Tell us about your PDF experience.

Azure Pipelines documentation

Implement continuous integration and continuous delivery (CI/CD) for the app and platform of your choice.

Get started

CONCEPT

What is Azure Pipelines?

Use Azure Pipelines

Key concepts for user new to Azure Pipelines

Classic release pipelines

QUICKSTART

Sign up for free

Create your first pipeline

Clone or import a pipeline

Customize your pipeline

Build, deploy, test-any language, any ecosystem

TUTORIAL

Container image

.NET Core apps

Anaconda

Android

Azure Kubernetes Service

Java

JavaScript & Node.js apps

Python apps

PHP

See more >

Key concepts

CONCEPT

Agents

Conditions

Expressions

Environments

Jobs

Runtime parameters

Stages

Tasks

Templates

Triggers

Reference guidance

REFERENCE

YAML schema

Predefined variables

Release & artifacts variables

Variable groups

Task index

Build any repository

HOW-TO GUIDE

GitHub

Azure Repos Git

Team Foundation version control

Bitbucket Cloud

Subversion

Generic Git

TUTORIAL

Build multiple Git branches

Deploy to Azure

OVERVIEW

Deploy to Azure

TUTORIAL

Connect to Azure

Deploy to Azure SQL Database

Deploy to Azure Kubernetes Service

Use secrets from Azure Key Vault

Manage agents & self-hosted agents

HOW-TO GUIDE

Deploy on Windows

Microsoft-hosted agents

Self-hosted Linux agents

Self-hosted macOS agents

Set up continuous testing

TUTORIAL

Configure for UI testing

UI testing using Selenium

Review code coverage results

Review test results

Manage flaky tests

Analyze test results

Configure resources

HOW-TO GUIDE

Create and manage agent pools

Create and manage deployment groups

Create and manage service connections

Set build and release retention policies

Troubleshoot

HOW-TO GUIDE

Troubleshoot builds and releases

Run a release in debug mode

Troubleshoot Azure Resource Manager service connections

Publish packages

TUTORIAL

Publish & download artifacts

Publish NuGet packages

Publish npm packages

Publish Python packages

Publish and download Universal Packages

Publish Maven Packages

Publish symbols for debugging

Integrate with other apps

HOW-TO GUIDE

Integrate with Slack

Integrate with Microsoft Teams

Integrate with service hooks

What is Azure Pipelines?

Article • 04/12/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Azure Pipelines automatically builds and tests code projects. It supports all major languages and project types and combines continuous integration, continuous delivery, and continuous testing to build, test, and deliver your code to any destination.

Continuous Integration

Continuous Integration (CI) is the practice used by development teams of automating, merging, and testing code. CI helps to catch bugs early in the development cycle, which makes them less expensive to fix. Automated tests execute as part of the CI process to ensure quality. CI systems produce artifacts and feed them to release processes to drive frequent deployments.

Continuous Delivery

Continuous Delivery (CD) is a process by which code is built, tested, and deployed to one or more test and production environments. Deploying and testing in multiple environments increases quality. CD systems produce deployable artifacts, including infrastructure and apps. Automated release processes consume these artifacts to release

new versions and fixes to existing systems. Systems that monitor and send alerts run continually to drive visibility into the entire CD process.

Continuous Testing

Whether your app is on-premises or in the cloud, you can automate build-deploy-test workflows and choose the technologies and frameworks. Then, you can test your changes continuously in a fast, scalable, and efficient manner. Continuous testing offers the following benefits.

- Maintain quality and find problems as you develop. Continuous testing with Azure DevOps Server ensures your app still works after every check-in and build, enabling you to find problems earlier by running tests automatically with each build.
- Use any test type and any test framework. Choose your preferred test technologies and frameworks.
- View rich analytics and reporting. When your build is done, review your test results to resolve any issues. Actionable build-on-build reports let you instantly see if your builds are getting healthier. But it's not just about speed - detailed and customizable test results measure the quality of your app.

Version control systems

Azure Pipelines requires your source code to be in a version control system. Azure DevOps supports two forms of version control - Git and Azure Repos. Any changes you push to your version control repository are automatically built and validated.

Languages and applications

You can build, test, and deploy Node.js, Python, Java, PHP, Ruby, C#, C++, Go, XCode, .NET, Android, and iOS applications. Run these apps in parallel on Linux, macOS, and Windows.

Azure DevOps offers tasks to build and test .NET, Java, Node, Android, Xcode, and C++ applications. Similarly, there are tasks to run tests using many testing frameworks and services. You can also run command line, PowerShell, or Shell scripts in your automation.

Deployment targets

Use Azure Pipelines to deploy your code to multiple targets. Targets include virtual machines, environments, containers, on-premises and cloud platforms, or PaaS services. You can also publish your mobile application to a store.

Once you have continuous integration in place, create a release definition to automate the deployment of your application to one or more environments. This automation process is defined as a collection of tasks.

Package formats

To produce packages that can be consumed by others, you can publish NuGet, npm, or Maven packages to the built-in package management repository in Azure Pipelines. You also can use any other package management repository of your choice.

What do I need to use Azure Pipelines?

To use Azure Pipelines, complete the following tasks:

- Have an organization in Azure DevOps. If you don't have one, create an organization now.
- Store your source code in a version control system.

Pricing for Azure DevOps

Azure DevOps Services

If you use public projects, Azure Pipelines is free, but you will need to request the free grant of parallel jobs. You can request this grant by submitting a request. Existing organizations and projects are not affected.

For more information, see What is a public project. If you use private projects, you can run up to 1,800 minutes (30 hours) of pipeline jobs for free every month.

For more information, see Pricing based on parallel jobs and Pricing for Azure DevOps Services.

Azure DevOps Server

With five or less active users, Azure DevOps Express is free, simple to set up, and installs on both client and server operating systems. It supports all the same features as Azure DevOps Server 2019.

For more information, see Pricing for Azure DevOps Server.

Why should I use Azure Pipelines?

Azure Pipelines provides a quick, easy, and safe way to automate building your projects with consistent and quality code that's readily available to users.

Use Azure Pipelines to support the following scenarios:

- Works with any language or platform
- Deploys to different types of targets at the same time
- Integrates with Azure deployments
- Builds on Windows, Linux, or Mac machines
- Integrates with GitHub
- Works with open-source projects

Next steps

Use Azure Pipelines

Related articles

- Sign up for Azure Pipelines
- Create your first pipeline
- Customize your pipeline

Use Azure Pipelines

Article • 06/07/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Azure Pipelines supports continuous integration (CI) and continuous delivery (CD) to continuously test, build, and deploy your code. You accomplish this by defining a pipeline.

The latest way to build pipelines is with the YAML pipeline editor. You can also use Classic pipelines with the Classic editor.

Automate tests, builds, and delivery

Continuous integration (CI) automates tests and builds for your project. CI helps to catch bugs or issues early in the development cycle, when they're easier and faster to fix. Items known as artifacts are produced from CI systems. They're used by the continuous delivery release pipelines to drive automatic deployments.

Continuous delivery automatically deploys and tests code in multiple stages to help drive quality. Continuous integration systems produce deployable artifacts, which include infrastructure and apps. Automated release pipelines consume these artifacts to release new versions and fixes to the target of your choice.

Continuous integration (CI)	Continuous delivery (CD)
- Increase code coverage - Build faster by splitting test and build runs - Automatically ensure you don't ship broken code - Run tests continually	- Automatically deploy code to production - Ensure deployment targets have latest code - Use tested code from CI process

Define pipelines using YAML syntax

You define your pipeline in a YAML file called `azure-pipelines.yml` with the rest of your app.

- The pipeline is versioned with your code. It follows the same branching structure. You get validation of your changes through code reviews in pull requests and branch build policies.
- Every branch you use can modify the pipeline by modifying the `azure-pipelines.yml` file. Learn more about branch consideration for YAML pipelines.
- A change to the build process might cause a break or result in an unexpected outcome. Because the change is in version control with the rest of your codebase, you can more easily identify the issue.

Follow these basic steps:

1. Configure Azure Pipelines to use your Git repo.
2. Edit your `azure-pipelines.yml` file to define your build.
3. Push your code to your version control repository. This action kicks off the default trigger to build and deploy and then monitor the results.

Your code is now updated, built, tested, and packaged. It can be deployed to any target.

Define pipelines using the Classic interface

Create and configure pipelines in the Azure DevOps web portal with the Classic user interface editor. You define a *build pipeline* to build and test your code, and then to publish artifacts. You also define a *release pipeline* to consume and deploy those artifacts to deployment targets.

Follow these basic steps:

1. Configure Azure Pipelines to use your Git repo.
2. Use the Azure Pipelines classic editor to create and configure your build and release pipelines.
3. Push your code to your version control repository. This action triggers your pipeline and runs tasks such as building or testing code.

The build creates an artifact that's used by the rest of your pipeline to run tasks such as deploying to staging or production.

Your code is now updated, built, tested, and packaged. It can be deployed to any target.

Feature availability

Certain pipeline features are only available when using YAML or when defining build or release pipelines with the Classic interface. The following table indicates which features are supported and for which tasks and methods.

Feature	YAML	Classic Build	Classic Release	Notes
Agents	✅	✅	✅	Specifies a required resource on which the pipeline runs.
Approvals	✅	❌	✅	Defines a set of validations required prior to completing a deployment stage.
Artifacts	✅	✅	✅	Supports publishing or consuming different package types.
Caching	✅	✅	❌	Reduces build time by allowing outputs or downloaded dependencies from one run to be reused in later runs. In Preview, available with Azure Pipelines only.
Conditions	✅	✅	✅	Specifies conditions to be met prior to running a job.
Container jobs	✅	❌	❌	Specifies jobs to run in a container.
Demands	✅	✅	✅	Ensures pipeline requirements are met before running a pipeline stage. Requires self-hosted agents.

Feature	YAML	Classic Build	Classic Release	Notes
Dependencies	✅	✅	✅	Specifies a requirement that must be met in order to run the next job or stage.
Deployment groups	❌	❌	✅	Defines a logical set of deployment target machines.
Deployment group jobs	❌	❌	✅	Specifies a job to release to a deployment group.
Deployment jobs	✅	❌	❌	Defines the deployment steps.
Environment	✅	❌	❌	Represents a collection of resources targeted for deployment. Available with Azure Pipelines only.
Gates	❌	❌	✅	Supports automatic collection and evaluation of external health signals prior to completing a release stage. Available with Classic Release only.
Jobs	✅	✅	✅	Defines the execution sequence of a set of steps.
Service connections	✅	✅	✅	Enables a connection to a remote service that is required to execute tasks in a job.
Service containers	✅	❌	❌	Enables you to manage the lifecycle of a containerized service.
Stages	✅	❌	✅	Organizes jobs within a pipeline.
Task groups	❌	✅	✅	Encapsulates a sequence of tasks into a single reusable task. If using YAML, see templates.
Tasks	✅	✅	✅	Defines the building blocks that make up a pipeline.
Templates	✅	❌	❌	Defines reusable content, logic, and parameters.
Triggers	✅	✅	✅	Defines the event that causes a pipeline to run.
Variables	✅	✅	✅	Represents a value to be replaced by data to pass to the pipeline.
Variable groups	✅	✅	✅	Use to store values that you want to control and make available across multiple pipelines.

Next steps

Create your first pipeline

Related articles

- Key concepts for new Azure Pipelines users

Sign up for Azure Pipelines

Article • 11/08/2022

Azure DevOps Services

Sign up for an Azure DevOps organization and Azure Pipelines to begin managing CI/CD to deploy your code with high-performance pipelines.

For more information about Azure Pipelines, see What is Azure Pipelines.

You can sign up with either a Microsoft account or a GitHub account.

Sign up with a Microsoft account

To sign up for Azure DevOps with a Microsoft account, complete the following steps.

1. Check that your account is up to date by logging into your Microsoft account.

2. Open Azure Pipelines and select **Start free**.

3. Log into your Microsoft account.

4. To get started with Azure Pipelines, select **Continue**.

5. Enter a name for your organization, select a host location from the drop-down menu, enter the characters you see, and then select **Continue**.

Use the following URL to sign in to your organization at any time:
`https://dev.azure.com/{yourorganization}`

You're now prompted to create a project.

Sign up with a GitHub account

To sign up for Azure DevOps with a GitHub account, complete the following steps.

> **Important**
>
> If your GitHub email address is associated with an Azure AD-backed organization in Azure DevOps, you can't sign in with your GitHub account, rather you must sign in with your Azure AD account.

1. Check that your account is up to date by logging into your GitHub account.

2. Open Azure Pipelines and select **Start free with GitHub**. If you're already part of an Azure DevOps organization, choose **Start free**.

3. Enter your GitHub account credentials, and then select **Sign in**.

4. Select **Authorize Microsoft-corp**.

5. Select **Next** to create a new Microsoft account linked to your GitHub credentials.

For more information about GitHub authentication, see FAQs.

6. Fill in your name, email address, and country/region.

7. To get started with Azure Pipelines, select **Continue**.

8. Enter a name for your organization, select a host location from the drop-down menu, enter the characters you see, and then select **Continue**.

Use the following URL to sign in to your organization at any time:
`https://dev.azure.com/{yourorganization}`

You're now prompted to create a project.

Create a project

You can create public or private projects. To learn more about public projects, see What is a public project?.

1. Enter a name for your project, select the visibility, and optionally provide a description. Then choose **Create project**.

   Special characters aren't allowed in the project name (such as / : \ ~ & % ; @ ' " ? < > | # $ * } { , + = [ ]). The project name also can't begin with an underscore, can't begin or end with a period, and must be 64 characters or less. Set your project visibility to either public or private. Public visibility allows for anyone on the internet to view your project. Private visibility is for only people who you give access to your project.

2. When your project is created, if you signed up with a Microsoft account, the wizard to create a new pipeline automatically starts. If you signed up with a GitHub account, you're asked to select which services to use.

You're now set to create your first pipeline, or invite other users to collaborate with your project.

Invite team members - optional

Add and invite others to work on your project by adding their email address to your organization and project.

1. From your project web portal, choose Azure DevOps > **Organization settings**.

2. Select **Users** > **Add users**.

3. Complete the form by entering or selecting the following information:

- **Users:** Enter the email addresses (Microsoft accounts) or GitHub IDs for the users. You can add several email addresses by separating them with a semicolon (;).
- **Access level:** Assign one of the following access levels:
  - **Basic:** Assign to users who must have access to all Azure Pipelines features. You can grant up to five users Basic access for free.
  - **Stakeholder:** Assign to users for limited access to features to view, add, and modify work items. You can assign an unlimited amount of users Stakeholder access for free.
  - **Visual Studio Subscriber:** Assign to users who already have a Visual Studio subscription.
- **Add to project:** Select the project you named in the preceding procedure.
- **Azure DevOps groups:** Select one of the following security groups, which will determine the permissions the users have to do select tasks. To learn more, see Azure Pipelines resources.
  - **Project Readers:** Assign to users who only require read-only access.
  - **Project Contributors:** Assign to users who will contribute fully to the project.
  - **Project Administrators:** Assign to users who will configure project resources.

> **Note**
>
> Add email addresses for **Microsoft accounts** and IDs for GitHub accounts unless you plan to use **Azure Active Directory (Azure AD)** to authenticate users and control organization access. If a user doesn't have a Microsoft or GitHub account, ask the user to **sign up** for a Microsoft account or a GitHub account.

4. When you're done, select **Add** to complete your invitation.

For more information, see Add organization users for Azure DevOps Services.

Change organization or project settings

You can rename and delete your organization, or change the organization location. For more information, see the following articles:

- Manage organizations
- Rename an organization
- Change the location of your organization

You can rename your project or change its visibility. To learn more about managing projects, see the following articles:

- Manage projects
- Rename a project
- Change the project visibility, public or private

Next steps

Create your first pipeline

Related articles

- What is Azure Pipelines?
- Key concepts for new Azure Pipelines users
- Customize your pipeline

Create your first pipeline

Article • 08/18/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

This is a step-by-step guide to using Azure Pipelines to build a sample application from a Git repository. This guide uses YAML pipelines configured with the YAML pipeline editor. If you'd like to use Classic pipelines instead, see Define your Classic pipeline. For guidance on using TFVC, see Build TFVC repositories.

Prerequisites - Azure DevOps

Make sure you have the following items:

- A GitHub account where you can create a repository. Create one for free.
- An Azure DevOps organization. Create one for free. If your team already has one, then make sure you're an administrator of the Azure DevOps project that you want to use.
- An ability to run pipelines on Microsoft-hosted agents. To use Microsoft-hosted agents, your Azure DevOps organization must have access to Microsoft-hosted parallel jobs. You can either purchase a parallel job or you can request a free grant.

Create your first pipeline

Java

Get the Java sample code

To get started, fork the following repository into your GitHub account.

```
https://github.com/MicrosoftDocs/pipelines-java
```

Create your first Java pipeline

1. Sign-in to your Azure DevOps organization and go to your project.
2. Go to **Pipelines**, and then select **New pipeline**.
3. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.
4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.
5. When you see the list of repositories, select your repository.
6. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.
7. Azure Pipelines will analyze your repository and recommend the **Maven** pipeline template.
8. When your new pipeline appears, take a look at the YAML to see what it does. When you're ready, select **Save and run**.
9. You're prompted to commit a new `azure-pipelines.yml` file to your repository. After you're happy with the message, select **Save and run** again.

   If you want to watch your pipeline in action, select the build job.

   You just created and ran a pipeline that we automatically created for you, because your code appeared to be a good match for the Maven template.

   You now have a working YAML pipeline (`azure-pipelines.yml`) in your repository that's ready for you to customize!
10. When you're ready to make changes to your pipeline, select it in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file.

Learn more about working with Java in your pipeline.

View and manage your pipelines

You can view and manage your pipelines by choosing **Pipelines** from the left-hand menu to go to the pipelines landing page.

From the pipelines landing page you can view pipelines and pipeline runs, create and import pipelines, manage security, and drill down into pipeline and run details.

Choose **Recent** to view recently run pipelines (the default view), or choose **All** to view all pipelines.

Select a pipeline to manage that pipeline and view the runs. Select the build number for the last run to view the results of that build, select the branch name to view the branch for that run, or select the context menu to run the pipeline and perform other management actions.

Select **Runs** to view all pipeline runs. You can optionally filter the displayed runs.

Select a pipeline run to view information about that run.

You can choose to **Retain** or **Delete** a run from the context menu. For more information on run retention, see Build and release retention policies.

View pipeline details

The details page for a pipeline allows you to view and manage that pipeline.

Choose **Edit** to edit your pipeline. For more information, see YAML pipeline editor. You can also edit your pipeline by modifying the **azure-pipelines.yml** file directly in the repository that hosts the pipeline.

View pipeline run details

From the pipeline run summary you can view the status of your run, both while it is running and when it is complete.

From the summary pane you can view job and stage details, download artifacts, and navigate to linked commits, test results, and work items.

Jobs and stages

The jobs pane displays an overview of the status of your stages and jobs. This pane may have multiple tabs depending on whether your pipeline has stages and jobs, or just jobs. In this example, the pipeline has two stages named **Build** and **Deploy**. You can drill down into the pipeline steps by choosing the job from either the **Stages** or **Jobs** pane.

Choose a job to see the steps for that job.

From the steps view, you can review the status and details of each step. From the **More actions** ⋮ you can toggle timestamps or view a raw log of all steps in the pipeline.

Cancel and re-run a pipeline

If the pipeline is running, you can cancel it by choosing **Cancel**. If the run has completed, you can re-run the pipeline by choosing **Run new**.

Pipeline run more actions menu

From the **More actions** menu you can download logs, add tags, edit the pipeline, delete the run, and configure retention for the run.

> **Note**
>
> You can't delete a run if the run is retained. If you don't see **Delete**, choose **Stop retaining run**, and then delete the run. If you see both **Delete** and **View retention releases**, one or more configured retention policies still apply to your run. Choose

View retention releases, delete the policies (only the policies for the selected run are removed), and then delete the run.

Add a status badge to your repository

Many developers like to show that they're keeping their code quality high by displaying a status badge in their repo.

To copy the status badge to your clipboard:

1. In Azure Pipelines, go to the **Pipelines** page to view the list of pipelines. Select the pipeline you created in the previous section.
2. Select ⋮, and then select **Status badge**.
3. Select **Status badge**.
4. Copy the sample Markdown from the Sample markdown section.

Now with the badge Markdown in your clipboard, take the following steps in GitHub:

1. Go to the list of files and select `Readme.md`. Select the pencil icon to edit.
2. Paste the status badge Markdown at the beginning of the file.
3. Commit the change to the `main` branch.
4. Notice that the status badge appears in the description of your repository.

To configure anonymous access to badges for private projects:

1. Navigate to **Project Settings** in the bottom left corner of the page
2. Open the **Settings** tab under **Pipelines**
3. Toggle the **Disable anonymous access to badges** slider under **General**

> ⓘ **Note**
>
> Even in a private project, anonymous badge access is enabled by default. With anonymous badge access enabled, users outside your organization might be able

to query information such as project names, branch names, job names, and build status through the badge status API.

Because you just changed the `Readme.md` file in this repository, Azure Pipelines automatically builds your code, according to the configuration in the `azure-pipelines.yml` file at the root of your repository. Back in Azure Pipelines, observe that a new run appears. Each time you make an edit, Azure Pipelines starts a new run.

Next steps

You've just learned how to create your first pipeline in Azure. Learn more about configuring pipelines in the language of your choice:

- .NET Core
- Go
- Java
- Node.js
- Python
- Containers

Or, you can proceed to customize the pipeline you just created.

To run your pipeline in a container, see Container jobs.

For details about building GitHub repositories, see Build GitHub repositories.

To learn how to publish your Pipeline Artifacts, see Publish Pipeline Artifacts.

To find out what else you can do in YAML pipelines, see YAML schema reference.

Clean up

If you created any test pipelines, they are easy to delete when you are done with them.

Browser

To delete a pipeline, navigate to the summary page for that pipeline, and choose **Delete** from the **...** menu at the top-right of the page. Type the name of the pipeline to confirm, and choose **Delete**.

FAQ

Where can I read articles about DevOps and CI/CD?

What is Continuous Integration?

What is Continuous Delivery?

What is DevOps?

What version control system can I use?

When you're ready to get going with CI/CD for your app, you can use the version control system of your choice:

- Clients
  - Visual Studio Code for Windows, macOS, and Linux
  - Visual Studio with Git for Windows or Visual Studio for Mac
  - Eclipse
  - Xcode
  - IntelliJ
  - Command line
- Services
  - Azure Pipelines
  - Git service providers such as Azure Repos Git, GitHub, and Bitbucket Cloud
  - Subversion

How can I delete a pipeline?

To delete a pipeline, navigate to the summary page for that pipeline, and choose **Delete** from the **...** menu in the top-right of the page. Type the name of the pipeline to confirm, and choose **Delete**.

What else can I do when I queue a build?

You can queue builds automatically or manually.

When you manually queue a build, you can, for a single run of the build:

- Specify the pool into which the build goes.
- Add and modify some variables.
- Add demands.
- In a Git repository
  - Build a branch or a tag.
  - Build a commit.

Where can I learn more about pipeline settings?

To learn more about pipeline settings, see:

- Getting sources
- Tasks
- Variables
- Triggers
- Retention
- History

How do I programmatically create a build pipeline?

REST API Reference: Create a build pipeline

> **Note**
>
> You can also manage builds and build pipelines from the command line or scripts using the **Azure Pipelines CLI**.

Customize your pipeline

Article • 08/17/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

This is a step-by-step guide on common ways to customize your pipeline.

Prerequisite

Follow instructions in Create your first pipeline to create a working pipeline.

Understand the `azure-pipelines.yml` file

A pipeline is defined using a YAML file in your repo. Usually, this file is named `azure-pipelines.yml` and is located at the root of your repo.

Navigate to the **Pipelines** page in Azure Pipelines, select the pipeline you created, and choose **Edit** in the context menu of the pipeline to open the YAML editor for the pipeline.

> ⓘ **Note**
>
> For instructions on how to view and manage your pipelines in the Azure DevOps portal, see **View and manage your pipelines**.

Examine the contents of the YAML file.

YAML

```yaml
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: Maven@4
  inputs:
    mavenPomFile: 'pom.xml'
    mavenOptions: '-Xmx3072m'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.11'
    jdkArchitectureOption: 'x64'
    publishJUnitResults: false
```

```
    testResultsFiles: '**/surefire-reports/TEST-*.xml'
    goals: 'package'
```

> **Note**
>
> The contents of your YAML file may be different depending on the sample repo you started with, or upgrades made in Azure Pipelines.

This pipeline runs whenever your team pushes a change to the main branch of your repo or creates a pull request. It runs on a Microsoft-hosted Linux machine. The pipeline process has a single step, which is to run the Maven task.

Change the platform to build on

You can build your project on Microsoft-hosted agents that already include SDKs and tools for various development languages. Or, you can use self-hosted agents with specific tools that you need.

- Navigate to the editor for your pipeline by selecting **Edit pipeline** action on the build, or by selecting **Edit** from the pipeline's main page.
- Currently the pipeline runs on a Linux agent:

  YAML
  ```yaml
  pool:
    vmImage: "ubuntu-latest"
  ```

- To choose a different platform like Windows or Mac, change the `vmImage` value:

  YAML
  ```yaml
  pool:
    vmImage: "windows-latest"
  ```

  YAML
  ```yaml
  pool:
    vmImage: "macos-latest"
  ```

- Select **Save** and then confirm the changes to see your pipeline run on a different platform.

Add steps

You can add more **scripts** or **tasks** as steps to your pipeline. A task is a pre-packaged script. You can use tasks for building, testing, publishing, or deploying your app. For Java, the Maven task we used handles testing and publishing results, however, you can use a task to publish code coverage results too.

- Open the YAML editor for your pipeline.
- Add the following snippet to the end of your YAML file.

  YAML

  ```yaml
  - task: PublishCodeCoverageResults@1
    inputs:
      codeCoverageTool: "JaCoCo"
      summaryFileLocation:
  "$(System.DefaultWorkingDirectory)/**/site/jacoco/jacoco.xml"
      reportDirectory: "$(System.DefaultWorkingDirectory)/**/site/jacoco"
      failIfCoverageEmpty: true
  ```

- Select **Save** and then confirm the changes.
- You can view your test and code coverage results by selecting your build and going to the **Test** and **Coverage** tabs.

Build across multiple platforms

You can build and test your project on multiple platforms. One way to do it is with `strategy` and `matrix`. You can use variables to conveniently put data into various parts of a pipeline. For this example, we'll use a variable to pass in the name of the image we want to use.

- In your `azure-pipelines.yml` file, replace this content:

  YAML

  ```yaml
  pool:
    vmImage: "ubuntu-latest"
  ```

  with the following content:

  YAML

```yaml
strategy:
  matrix:
    linux:
      imageName: "ubuntu-latest"
    mac:
      imageName: "macOS-latest"
    windows:
      imageName: "windows-latest"
  maxParallel: 3

pool:
  vmImage: $(imageName)
```

- Select **Save** and then confirm the changes to see your build run up to three jobs on three different platforms.

> Each agent can run only one job at a time. To run multiple jobs in parallel you must configure multiple agents. You also need sufficient parallel jobs.

Build using multiple versions

To build a project using different versions of that language, you can use a `matrix` of versions and a variable. In this step, you can either build the Java project with two different versions of Java on a single platform or run different versions of Java on different platforms.

> **Note**
>
> You cannot use `strategy` multiples times in a context.

- If you want to build on a single platform and multiple versions, add the following matrix to your `azure-pipelines.yml` file before the Maven task and after the `vmImage`.

YAML

```yaml
strategy:
  matrix:
    jdk10:
      jdkVersion: "1.10"
    jdk11:
      jdkVersion: "1.11"
  maxParallel: 2
```

- Then replace this line in your maven task:

  YAML

  ```yaml
  jdkVersionOption: "1.11"
  ```

  with this line:

  YAML

  ```yaml
  jdkVersionOption: $(jdkVersion)
  ```

- Make sure to change the `$(imageName)` variable back to the platform of your choice.

- If you want to build on multiple platforms and versions, replace the entire content in your `azure-pipelines.yml` file before the publishing task with the following snippet:

  YAML

  ```yaml
  trigger:
  - main

  strategy:
    matrix:
      jdk10_linux:
        imageName: "ubuntu-latest"
        jdkVersion: "1.10"
      jdk11_windows:
        imageName: "windows-latest"
        jdkVersion: "1.11"
    maxParallel: 2

  pool:
    vmImage: $(imageName)

  steps:
  - task: Maven@4
    inputs:
      mavenPomFile: "pom.xml"
      mavenOptions: "-Xmx3072m"
      javaHomeOption: "JDKVersion"
      jdkVersionOption: $(jdkVersion)
      jdkArchitectureOption: "x64"
      publishJUnitResults: true
      testResultsFiles: "**/TEST-*.xml"
      goals: "package"
  ```

- Select **Save** and then confirm the changes to see your build run two jobs on two different platforms and SDKs.

Customize CI triggers

Pipeline triggers cause a pipeline to run. You can use `trigger:` to cause a pipeline to run whenever you push an update to a branch. YAML pipelines are configured by default with a CI trigger on your default branch (which is usually `main`). You can set up triggers for specific branches or for pull request validation. For a pull request validation trigger, just replace the `trigger:` step with `pr:` as shown in the two examples below. By default, the pipeline runs for each pull request change.

- If you'd like to set up triggers, add either of the following snippets at the beginning of your `azure-pipelines.yml` file.

  YAML

  ```yaml
  trigger:
    - main
    - releases/*
  ```

  YAML

  ```yaml
  pr:
    - main
    - releases/*
  ```

  You can specify the full name of the branch (for example, `main`) or a prefix-matching wildcard (for example, `releases/*`).

Pipeline settings

You can view and configure pipeline settings from the **More actions** ⋮ menu on the pipeline details page.

- **Manage security** - Manage security
- **Rename/move** - Edit your pipeline name and folder location.

- **Status badge** - Add a status badge to your repository
- **Delete** - Deletes the pipeline including all builds and associated artifacts.
- **Scheduled runs** - Scheduled runs view

Choose **Settings** to configure the following pipeline settings.

From the **Pipeline settings** pane you can configure the following settings.

- **Processing of new run requests** - Sometimes you'll want to prevent new runs from starting on your pipeline.
  - By default, the processing of new run requests is **Enabled**. This setting allows standard processing of all trigger types, including manual runs.
  - **Paused** pipelines allow run requests to be processed, but those requests are queued without actually starting. When new request processing is enabled, run processing resumes starting with the first request in the queue.
  - **Disabled** pipelines prevent users from starting new runs. All triggers are also disabled while this setting is applied.

- **YAML file path** - If you ever need to direct your pipeline to use a different YAML file, you can specify the path to that file. This setting can also be useful if you need to move/rename your YAML file.

- **Automatically link work items included in this run** - The changes associated with a given pipeline run may have work items associated with them. Select this option to link those work items to the run. When **Automatically link work items included in this run** is selected, you must specify either a specific branch, or `*` for all branches, which is the default. If you specify a branch, work items are only associated with runs of that branch. If you specify `*`, work items are associated for all runs.

- To get notifications when your runs fail, see how to Manage notifications for a team

Manage security

You can configure pipelines security on a project level from the **More actions** ⋮ on the pipelines landing page, and on a pipeline level on the pipeline details page.

To support security of your pipeline operations, you can add users to a built-in security group, set individual permissions for a user or group, or add users to predefined roles. You can manage security for Azure Pipelines in the web portal, either from the user or admin context. For more information on configuring pipelines security, see Pipeline permissions and security roles.

Create work item on failure

YAML pipelines don't have a Create work item on failure setting like classic build pipelines. Classic build pipelines are single stage, and **Create work item on failure** applies to the whole pipeline. YAML pipelines can be multi-stage, and a pipeline level setting may not be appropriate. To implement **Create work item on failure** in a YAML pipeline, you can use methods such as the Work Items - Create REST API call or the Azure DevOps CLI az boards work-item create command at the desired point in your pipeline.

The following example has two jobs. The first job represents the work of the pipeline, but if it fails, the second job runs, and creates a bug in the same project as the pipeline.

yml

```yml
# When manually running the pipeline, you can select whether it
# succeeds or fails.
parameters:
- name: succeed
  displayName: Succeed or fail
  type: boolean
  default: false

trigger:
- main

pool:
  vmImage: ubuntu-latest

jobs:
- job: Work
  steps:
  - script: echo Hello, world!
    displayName: 'Run a one-line script'

  # This malformed command causes the job to fail
  # Only run this command if the succeed variable is set to false
  - script: git clone malformed input
    condition: eq(${{ parameters.succeed }}, false)

# This job creates a work item, and only runs if the previous job failed
- job: ErrorHandler
  dependsOn: Work
  condition: failed()
  steps:
  - bash: |
      az boards work-item create \
        --title "Build $(build.buildNumber) failed" \
        --type bug \
        --org $(System.TeamFoundationCollectionUri) \
        --project $(System.TeamProject)
    env:
```

```
      AZURE_DEVOPS_EXT_PAT: $(System.AccessToken)
    displayName: 'Create work item on failure'
```

> **Note**
>
> Azure Boards allows you to configure your work item tracking using several different processes, such as Agile or Basic. Each process has different work item types, and not every work item type is available in each process. For a list of work item types supported by each process, see **Work item types (WITs)**.

The previous example uses Runtime parameters to configure whether the pipeline succeeds or fails. When manually running the pipeline, you can set the value of the `succeed` parameter. The second `script` step in the first job of the pipeline evaluates the `succeed` parameter and only runs when `succeed` is set to false.

The second job in the pipeline has a dependency on the first job and only runs if the first job fails. The second job uses the Azure DevOps CLI az boards work-item create command to create a bug. For more information on running Azure DevOps CLI commands from a pipeline, see Run commands in a YAML pipeline.

This example uses two jobs, but this same approach could be used across multiple stages.

> **Note**
>
> You can also use a marketplace extension like **Create Bug on Release failure** which has support for YAML multi-stage pipelines.

Next steps

You've learned the basics of customizing your pipeline. Next we recommend that you learn more about customizing a pipeline for the language you use:

- .NET Core
- Containers
- Go
- Java
- Node.js
- Python

Or, to grow your CI pipeline to a CI/CD pipeline, include a deployment job with steps to deploy your app to an environment.

To learn more about the topics in this guide see Jobs, Tasks, Catalog of Tasks, Variables, Triggers, or Troubleshooting.

To learn what else you can do in YAML pipelines, see YAML schema reference.

Key concepts for new Azure Pipelines users

Article • 12/20/2022

Azure DevOps Services

Learn about the key concepts and components that make up a pipeline. Understanding the basic terms and parts of a pipeline can help you deliver better code more efficiently and reliably.

https://www.microsoft.com/en-us/videoplayer/embed/RWMlMo?postJsllMsg=true

Key concepts overview

- A trigger tells a Pipeline to run.
- A pipeline is made up of one or more stages. A pipeline can deploy to one or more environments.
- A stage is a way of organizing jobs in a pipeline and each stage can have one or more jobs.
- Each job runs on one agent. A job can also be agentless.
- Each agent runs a job that contains one or more steps.
- A step can be a task or script and is the smallest building block of a pipeline.
- A task is a pre-packaged script that performs an action, such as invoking a REST API or publishing a build artifact.
- An artifact is a collection of files or packages published by a run.

Azure Pipelines terms

Agent

When your build or deployment runs, the system begins one or more jobs. An agent is computing infrastructure with installed agent software that runs one job at a time. For example, your job could run on a Microsoft-hosted Ubuntu agent.

For more in-depth information about the different types of agents and how to use them, see Azure Pipelines Agents.

Approvals

Approvals define a set of validations required before a deployment runs. Manual approval is a common check performed to control deployments to production environments. When checks are configured on an environment, pipelines will stop before starting a stage that deploys to the environment until all the checks are completed successfully.

Artifact

An artifact is a collection of files or packages published by a run. Artifacts are made available to subsequent tasks, such as distribution or deployment. For more information, see Artifacts in Azure Pipelines.

Continuous delivery

Continuous delivery (CD) is a process by which code is built, tested, and deployed to one or more test and production stages. Deploying and testing in multiple stages helps drive quality. Continuous integration systems produce deployable artifacts, which include infrastructure and apps. Automated release pipelines consume these artifacts to release new versions and fixes to existing systems. Monitoring and alerting systems run constantly to drive visibility into the entire CD process. This process ensures that errors are caught often and early.

Continuous integration

Continuous integration (CI) is the practice used by development teams to simplify the testing and building of code. CI helps to catch bugs or problems early in the development cycle, which makes them easier and faster to fix. Automated tests and builds are run as part of the CI process. The process can run on a set schedule, whenever code is pushed, or both. Items known as artifacts are produced from CI systems. They're used by the continuous delivery release pipelines to drive automatic deployments.

Deployment

For Classic pipelines, a deployment is the action of running the tasks for one stage, which can include running automated tests, deploying build artifacts, and any other actions are specified for that stage.

For YAML pipelines, a deployment typically refers to a deployment job. A deployment job is a collection of steps that are run sequentially against an environment. You can use strategies like run once, rolling, and canary for deployment jobs.

Deployment group

A deployment group is a set of deployment target machines that have agents installed. A deployment group is just another grouping of agents, like an agent pool. You can set the deployment targets in a pipeline for a job using a deployment group. Learn more about provisioning agents for deployment groups.

Environment

An environment is a collection of resources, where you deploy your application. It can contain one or more virtual machines, containers, web apps, or any service that's used to host the application being developed. A pipeline might deploy the app to one or more environments after build is completed and tests are run.

Job

A stage contains one or more jobs. Each job runs on an agent. A job represents an execution boundary of a set of steps. All of the steps run together on the same agent. Jobs are most useful when you want to run a series of steps in different environments. For example, you might want to build two configurations - x86 and x64. In this case, you have one stage and two jobs. One job would be for x86 and the other job would be for x64.

Pipeline

A pipeline defines the continuous integration and deployment process for your app. It's made up of one or more stages. It can be thought of as a workflow that defines how your test, build, and deployment steps are run.

For Classic pipelines, a pipeline can also be referred to as a definition.

Release

For Classic pipelines, a release is a versioned set of artifacts specified in a pipeline. The release includes a snapshot of all the information required to carry out all the tasks and actions in the release pipeline, such as stages, tasks, policies such as triggers and approvers, and deployment options. You can create a release manually, with a deployment trigger, or with the REST API.

For YAML pipelines, the build and release stages are in one, multi-stage pipeline.

Run

A run represents one execution of a pipeline. It collects the logs associated with running the steps and the results of running tests. During a run, Azure Pipelines will first process the pipeline and then send the run to one or more agents. Each agent will run jobs. Learn more about the pipeline run sequence.

For Classic pipelines, a build represents one execution of a pipeline.

Script

A script runs code as a step in your pipeline using command line, PowerShell, or Bash. You can write cross-platform scripts for macOS, Linux, and Windows. Unlike a task, a script is custom code that is specific to your pipeline.

Stage

A stage is a logical boundary in the pipeline. It can be used to mark separation of concerns (for example, Build, QA, and production). Each stage contains one or more jobs. When you define multiple stages in a pipeline, by default, they run one after the other. You can specify the conditions for when a stage runs. When you are thinking about whether you need a stage, ask yourself:

- Do separate groups manage different parts of this pipeline? For example, you could have a test manager that manages the jobs that relate to testing and a different manager that manages jobs related to production deployment. In this case, it makes sense to have separate stages for testing and production.
- Is there a set of approvals that are connected to a specific job or set of jobs? If so, you can use stages to break your jobs into logical groups that require approvals.
- Are there jobs that need to run a long time? If you have part of your pipeline that will have an extended run time, it makes sense to divide them into their own stage.

Step

A step is the smallest building block of a pipeline. For example, a pipeline might consist of build and test steps. A step can either be a script or a task. A task is simply a pre-created script offered as a convenience to you. To view the available tasks, see the Build and release tasks reference. For information on creating custom tasks, see Create a custom task.

Task

A task is the building block for defining automation in a pipeline. A task is packaged script or procedure that has been abstracted with a set of inputs.

Trigger

A trigger is something that's set up to tell the pipeline when to run. You can configure a pipeline to run upon a push to a repository, at scheduled times, or upon the completion of another build. All of these actions are known as triggers. For more information, see build triggers and release triggers.

Library

The Library includes **secure files** and **variable groups**. Secure files are a way to store files and share them across pipelines. You may need to save a file at the DevOps level and then use it during build or deployment. In that case, you can save the file within Library and use it when you need it. Variable groups store values and secrets that you might want to be passed into a YAML pipeline or make available across multiple pipelines.

About the authors

- Dave Jarvis contributed to the key concepts overview graphic.

YAML pipeline editor

Article • 08/17/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Azure Pipelines provides a YAML pipeline editor that you can use to author and edit your pipelines. The YAML editor is based on the Monaco Editor. The editor provides tools like Intellisense support and a task assistant to provide guidance while you edit a pipeline.

This article shows you how to edit your pipelines using the YAML Pipeline editor, but you can also edit pipelines by modifying the **azure-pipelines.yml** file directly in your pipeline's repository using a text editor of your choice, or by using a tool like Visual Studio Code and the Azure Pipelines for VS Code extension.

Edit a YAML pipeline

To access the YAML pipeline editor, do the following steps.

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

2. Select your project, choose **Pipelines**, and then select the pipeline you want to edit. You can browse pipelines by **Recent**, **All**, and **Runs**. For more information, see view and manage your pipelines.

3. Choose **Edit**.

4. Make edits to your pipeline using Intellisense and the task assistant for guidance.

5. Choose **Save**. You can commit directly to your branch, or create a new branch and optionally start a pull request.

Use keyboard shortcuts

The YAML pipeline editor provides several keyboard shortcuts, which we show in the following examples.

- Choose **Ctrl**+**Space** for Intellisense support while you're editing the YAML pipeline.

- Choose **F1** (**Fn+F1** on Mac) to display the command palette and view the available keyboard shortcuts.

Use task assistant

The task assistant provides a method for adding tasks to your YAML pipeline.

- To display the task assistant, edit your YAML pipeline and choose **Show assistant**.

- To hide the task assistant, choose **Hide assistant**.

- To use the task assistant, browse or search for tasks in the **Tasks** pane.

- Select the desired task and configure its inputs.

- Choose **Add** to insert the task YAML into your pipeline.

- You can edit the YAML to make more configuration changes to the task, or you can choose **Settings** above the task in the YAML pipeline editor to configure the inserted task in the task assistant.

Validate

Validate your changes to catch syntax errors in your pipeline that prevent it from starting. Choose **More actions** > **Validate**.

Download full YAML

You can preview the fully parsed YAML document without committing or running the pipeline. Choose **More actions** > **Download full YAML**.

Download full YAML Runs the Azure DevOps REST API for Azure Pipelines and initiates a download of the rendered YAML from the editor.

Manage pipeline variables

You can manage pipeline variables both from within your YAML pipeline and from the pipeline settings UI.

To manage pipeline variables, do the following steps.

1. Edit your YAML pipeline and choose **Variables** to manage pipeline variables.

2. Choose from the following functions:

   - **New variable**: to add your first variable.
   - **Add** + : to add subsequent variables.
   - *Variable name* to edit a variable.

- **Delete** 🗑 : to delete a variable.

To manage pipelines variables in the pipeline settings UI, do the following steps.

1. Edit the pipeline and choose **More actions** > **Triggers**.

2. Choose **Variables**.

For more information on working with pipeline variables, see Define variables.

Configure the default agent pool

If a YAML pipeline doesn't specify an agent pool, the agent pool configured in the **Default agent pool for YAML** setting is used. This pool is also used for post-run cleanup tasks.

To view and configure the **Default agent pool for YAML** setting:

1. Edit the pipeline and choose **More actions** > **Triggers**.

2. Choose **YAML**, and select the desired agent pool using the **Default agent pool for YAML** dropdown list.

Default agent pool for YAML is configured on a per-pipeline basis.

Manage settings using the pipeline settings UI

Some YAML pipeline settings are configured using the pipeline settings UI instead of in the YAML file.

1. Edit the pipeline and choose **More actions** > **Triggers**.

2. From the pipeline settings UI, choose the tab of the setting to configure.

View and edit templates

Templates are a commonly used feature in YAML pipelines. They're an easy way to share pipeline snippets and are a powerful mechanism for verifying and enforcing security and governance in your pipeline. Previously, the editor didn't support templates, so authors of YAML pipelines couldn't get intellisense assistance. Now Azure Pipelines supports a YAML editor, for which we're previewing support. To enable this preview, go to preview features in your Azure DevOps organization, and enable **YAML templates editor**.

> **Important**
>
> This feature has the following limitations.
>
> - If the template has required parameters that aren't provided as inputs in the main YAML file, then the validation fails and prompts you to provide those inputs.
> - You can't create a new template from the editor. You can only use or edit existing templates.

As you edit your main Azure Pipelines YAML file, you can either *include* or *extend* a template. As you enter the name of your template, you may be prompted to validate your template. Once validated, the YAML editor understands the schema of the template, including the input parameters.

main azure-devops-cli-in-yaml / azure-pipelines.yml *

```yaml
trigger: none

jobs:
# Running Azure DevOps extension commands on a hosted Mac agent
- job:
  condition: false
  displayName: 'macOS'
  pool:
    vmImage: 'macOS-latest'
  steps:
    View template
  - template: azure-pipelines-steps-mac.yml

# Running Azure DevOps extension commands on a hosted Linux agent
- job:
  condition: true
  displayName: 'Linux'
  pool:
    vmImage: 'ubuntu-16.04'
  steps:
    View template
  - template: azure-pipelines-steps-linux.yml
```

Post validation, you can go into the template by choosing **View template**, which opens the template in a new browser tab. You can make changes to the template using all the features of the YAML editor.

Next steps

Customize your pipeline

Related articles

- Learn how to navigate and view your pipelines
- Create your first pipeline

Supported source repositories

Article • 01/26/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Azure Pipelines, Azure DevOps Server, and TFS integrate with a number of version control systems. When you use any of these version control systems, you can configure a pipeline to build, test, and deploy your application.

YAML pipelines are a new form of pipelines that have been introduced in Azure DevOps Server 2019 and in Azure Pipelines. YAML pipelines only work with certain version control systems. The following table shows all the supported version control systems and the ones that support YAML pipelines.

Repository type	Azure Pipelines (YAML)	Azure Pipelines (classic editor)	Azure DevOps Server 2022, 2020, 2019, TFS 2018
Azure Repos Git	Yes	Yes	Yes
Azure Repos TFVC	No	Yes	Yes
GitHub	Yes	Yes	No
GitHub Enterprise Server	Yes	Yes	TFS 2018.2 and higher
Bitbucket Cloud	Yes	Yes	No
Bitbucket Server	No	Yes	Yes
Subversion	No	Yes	Yes

Build Azure Repos Git or TFS Git repositories

Article • 01/24/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Azure Pipelines can automatically build and validate every pull request and commit to your Azure Repos Git repository.

Choose a repository to build

YAML

You create a new pipeline by first selecting a repository and then a YAML file in that repository. The repository in which the YAML file is present is called `self` repository. By default, this is the repository that your pipeline builds.

You can later configure your pipeline to check out a different repository or multiple repositories. To learn how to do this, see multi-repo checkout.

Azure Pipelines must be granted access to your repositories to trigger their builds and fetch their code during builds. Normally, a pipeline has access to repositories in the same project. But, if you wish to access repositories in a different project, then you need to update the permissions granted to job access tokens.

CI triggers

Continuous integration (CI) triggers cause a pipeline to run whenever you push an update to the specified branches or you push specified tags.

YAML

YAML pipelines are configured by default with a CI trigger on all branches.

Branches

You can control which branches get CI triggers with a simple syntax:

YAML

```yaml
trigger:
- main
- releases/*
```

You can specify the full name of the branch (for example, `main`) or a wildcard (for example, `releases/*`). See Wildcards for information on the wildcard syntax.

> **Note**
>
> You cannot use **variables** in triggers, as variables are evaluated at runtime (after the trigger has fired).

> **Note**
>
> If you use **templates** to author YAML files, then you can only specify triggers in the main YAML file for the pipeline. You cannot specify triggers in the template files.

For more complex triggers that use `exclude` or `batch`, you must use the full syntax as shown in the following example.

YAML

```yaml
# specific branch build
trigger:
  branches:
    include:
    - main
    - releases/*
    exclude:
    - releases/old*
```

In the above example, the pipeline will be triggered if a change is pushed to `main` or to any releases branch. However, it won't be triggered if a change is made to a releases branch that starts with `old`.

If you specify an `exclude` clause without an `include` clause, then it is equivalent to specifying `*` in the `include` clause.

In addition to specifying branch names in the `branches` lists, you can also configure triggers based on tags by using the following format:

YAML

```yaml
trigger:
  branches:
    include:
      - refs/tags/{tagname}
    exclude:
      - refs/tags/{othertagname}
```

If you don't specify any triggers, the default is as if you wrote:

YAML

```yaml
trigger:
  branches:
    include:
    - '*'  # must quote since "*" is a YAML reserved character; we want
a string
```

> **Important**
>
> When you specify a trigger, it replaces the default implicit trigger, and only pushes to branches that are explicitly configured to be included will trigger a pipeline. Includes are processed first, and then excludes are removed from that list.

Batching CI runs

If you have many team members uploading changes often, you may want to reduce the number of runs you start. If you set `batch` to `true`, when a pipeline is running, the system waits until the run is completed, then starts another run with all changes that have not yet been built.

YAML

```yaml
# specific branch build with batching
trigger:
  batch: true
  branches:
    include:
    - main
```

> **Note**
>
> `batch` is not supported in repository resource triggers.

To clarify this example, let us say that a push `A` to `main` caused the above pipeline to run. While that pipeline is running, additional pushes `B` and `C` occur into the repository. These updates do not start new independent runs immediately. But after the first run is completed, all pushes until that point of time are batched together and a new run is started.

> **Note**
>
> If the pipeline has multiple jobs and stages, then the first run should still reach a terminal state by completing or skipping all its jobs and stages before the second run can start. For this reason, you must exercise caution when using this feature in a pipeline with multiple stages or approvals. If you wish to batch your builds in such cases, it is recommended that you split your CI/CD process into two pipelines - one for build (with batching) and one for deployments.

Paths

You can specify file paths to include or exclude.

YAML

```yaml
# specific path build
trigger:
  branches:
    include:
    - main
    - releases/*
  paths:
    include:
    - docs
    exclude:
    - docs/README.md
```

When you specify paths, you must explicitly specify branches to trigger on if you are using Azure DevOps Server 2019.1 or lower. You can't trigger a pipeline with only a path filter; you must also have a branch filter, and the changed files that match the path filter must be from a branch that matches the branch filter. If you

are using Azure DevOps Server 2020 or newer, you can omit `branches` to filter on all branches in conjunction with the path filter.

Wilds cards are supported for path filters. For instance, you can include all paths that match `src/app/**/myapp*`. You can use wild card characters (`**`, `*`, or `?`) when specifying path filters.

- Paths are always specified relative to the root of the repository.
- If you don't set path filters, then the root folder of the repo is implicitly included by default.
- If you exclude a path, you cannot also include it unless you qualify it to a deeper folder. For example if you exclude */tools* then you could include */tools/trigger-runs-on-these*
- The order of path filters doesn't matter.
- Paths in Git *are case-sensitive*. Be sure to use the same case as the real folders.
- You cannot use variables in paths, as variables are evaluated at runtime (after the trigger has fired).

Tags

In addition to specifying tags in the `branches` lists as covered in the previous section, you can directly specify tags to include or exclude:

YAML

```yaml
# specific tag
trigger:
  tags:
    include:
    - v2.*
    exclude:
    - v2.0
```

If you don't specify any tag triggers, then by default, tags will not trigger pipelines.

> **Important**
>
> If you specify tags in combination with branch filters, the trigger will fire if either the branch filter is satisfied or the tag filter is satisfied. For example, if a pushed tag satisfies the branch filter, the pipeline triggers even if the tag is excluded by the tag filter, because the push satisfied the branch filter.

Opting out of CI

Disabling the CI trigger

You can opt out of CI triggers entirely by specifying `trigger: none`.

YAML

```yaml
# A pipeline with no CI trigger
trigger: none
```

> ⓘ **Important**
>
> When you push a change to a branch, the YAML file in that branch is evaluated to determine if a CI run should be started.

Skipping CI for individual pushes

You can also tell Azure Pipelines to skip running a pipeline that a push would normally trigger. Just include `[skip ci]` in the message or description of any of the commits that are part of a push, and Azure Pipelines will skip running CI for this push. You can also use any of the following variations.

- `[skip ci]` or `[ci skip]`
- `skip-checks: true` or `skip-checks:true`
- `[skip azurepipelines]` or `[azurepipelines skip]`
- `[skip azpipelines]` or `[azpipelines skip]`
- `[skip azp]` or `[azp skip]`
- `***NO_CI***`

Using the trigger type in conditions

It is a common scenario to run different steps, jobs, or stages in your pipeline depending on the type of trigger that started the run. You can do this using the system variable `Build.Reason`. For example, add the following condition to your step, job, or stage to exclude it from PR validations.

```
condition: and(succeeded(), ne(variables['Build.Reason'], 'PullRequest'))
```

Behavior of triggers when new branches are created

It is common to configure multiple pipelines for the same repository. For instance, you may have one pipeline to build the docs for your app and another to build the source code. You may configure CI triggers with appropriate branch filters and path filters in each of these pipelines. For instance, you may want one pipeline to trigger when you push an update to the `docs` folder, and another one to trigger when you push an update to your application code. In these cases, you need to understand how the pipelines are triggered when a new branch is created.

Here is the behavior when you push a new branch (that matches the branch filters) to your repository:

- If your pipeline has path filters, it will be triggered only if the new branch has changes to files that match that path filter.
- If your pipeline does not have path filters, it will be triggered even if there are no changes in the new branch.

Wildcards

When specifying a branch, tag, or path, you may use an exact name or a wildcard. Wildcards patterns allow `*` to match zero or more characters and `?` to match a single character.

- If you start your pattern with `*` in a YAML pipeline, you must wrap the pattern in quotes, like `"*-releases"`.
- For branches and tags:
  - A wildcard may appear anywhere in the pattern.
- For paths:
  - In Azure DevOps Server 2022 and higher, including Azure DevOps Services, a wildcard may appear anywhere within a path pattern and you may use `*` or `?`.
  - In Azure DevOps Server 2020 and lower, you may include `*` as the final character, but it doesn't do anything differently from specifying the directory name by itself. You may **not** include `*` in the middle of a path filter, and you may not use `?`.

YAML

```yaml
trigger:
  branches:
    include:
    - main
    - releases/*
```

```
    - feature/*
    exclude:
    - releases/old*
    - feature/*-working
  paths:
    include:
    - docs/*.md
```

PR triggers

Pull request (PR) triggers cause a pipeline to run whenever you open a pull request, or when you push changes to it. In Azure Repos Git, this functionality is implemented using branch policies. To enable PR validation, navigate to the branch policies for the desired branch, and configure the Build validation policy for that branch. For more information, see Configure branch policies.

If you have an open PR and you push changes to its source branch, multiple pipelines may run:

- The pipelines specified by the target branch's build validation policy will run on the *merge commit* (the merged code between the source and target branches of the pull request), regardless if there exist pushed commits whose messages or descriptions contain `[skip ci]` (or any of its variants).
- The pipelines triggered by changes to the PR's source branch, if there are **no** pushed commits whose messages or descriptions contain `[skip ci]` (or any of its variants). If at least one pushed commit contains `[skip ci]`, the pipelines will not run.

Finally, after you merge the PR, Azure Pipelines will run the CI pipelines triggered by pushes to the target branch, even if some of the merged commits' messages or descriptions contain `[skip ci]` (or any of its variants).

> ⓘ **Note**
>
> To configure validation builds for an Azure Repos Git repository, you must be a project administrator of its project.

> ⓘ **Note**
>
> **Draft pull requests** do not trigger a pipeline even if you configure a branch policy.

Validate contributions from forks

Building pull requests from Azure Repos forks is no different from building pull requests within the same repository or project. You can create forks only within the same organization that your project is part of.

Limit job authorization scope

Azure Pipelines provides several security settings to configure the job authorization scope that your pipelines run with.

- Limit job authorization scope to current project
- Protect access to repositories in YAML pipelines

Limit job authorization scope to current project

Azure Pipelines provides two **Limit job authorization scope to current project** settings:

- **Limit job authorization scope to current project for non-release pipelines** - This setting applies to YAML pipelines and classic build pipelines. This setting does not apply to classic release pipelines.
- **Limit job authorization scope to current project for release pipelines** - This setting applies to classic release pipelines only.

Pipelines run with collection scoped access tokens unless the relevant setting for the pipeline type is enabled. The **Limit job authorization scope** settings allow you to reduce the scope of access for all pipelines to the current project. This can impact your pipeline if you are accessing an Azure Repos Git repository in a different project in your organization.

If your Azure Repos Git repository is in a different project than your pipeline, and the **Limit job authorization scope** setting for your pipeline type is enabled, you must grant permission to the build service identity for your pipeline to the second project. For more information, see Manage build service account permissions.

For more information on **Limit job authorization scope**, see Understand job access tokens.

Protect access to repositories in YAML pipelines

Pipelines can access any Azure DevOps repositories in authorized projects, as described in the previous Limit job authorization scope to current project section, unless **Protect**

access to repositories in YAML pipelines is enabled. With this option enabled, you can reduce the scope of access for all pipelines to only Azure DevOps repositories explicitly referenced by a `checkout` step or a `uses` statement in the pipeline job that uses that repository.

To configure this setting, navigate to **Pipelines**, **Settings** at either **Organization settings** or **Project settings**. If enabled at the organization level, the setting is grayed out and unavailable at the project settings level.

> ⓘ **Important**
>
> **Protect access to repositories in YAML pipelines** is enabled by default for new organizations and projects created after May 2020. When **Protect access to repositories in YAML pipelines** is enabled, your YAML pipelines must explicitly reference any Azure Repos Git repositories you want to use in the pipeline as a checkout step in the job that uses the repository. You won't be able to fetch code using scripting tasks and git commands for an Azure Repos Git repository unless that repo is first explicitly referenced.

There are a few exceptions where you don't need to explicitly reference an Azure Repos Git repository before using it in your pipeline when **Protect access to repositories in YAML pipelines** is enabled.

- If you do not have an explicit checkout step in your pipeline, it is as if you have a `checkout: self` step, and the `self` repository is checked out.
- If you are using a script to perform read-only operations on a repository in a public project, you don't need to reference the public project repository in a `checkout` step.
- If you are using a script that provides its own authentication to the repo, such as a PAT, you don't need to reference that repository in a `checkout` step.

For example, when **Protect access to repositories in YAML pipelines** is enabled, if your pipeline is in the `FabrikamProject/Fabrikam` repo in your organization, and you want to use a script to check out the `FabrikamProject/FabrikamTools` repo, you must either reference this repository in a `checkout` step or with a `uses` statement.

If you are already checking out the `FabrikamTools` repository in your pipeline using a checkout step, you may subsequently use scripts to interact with that repository.

```yml
```

```
steps:
- checkout: git://FabrikamFiber/FabrikamTools # Azure Repos Git repository
in the same organization
- script: # Do something with that repo
# Or you can reference it with a uses statement in the job
uses:
  repositories: # List of referenced repositories
  - FabrikamTools # Repository reference to FabrikamTools
steps:
- script: # Do something with that repo like clone it
```

> **Note**
>
> For many scenarios, multi-repo checkout can be leveraged, removing the need to use scripts to check out additional repositories in your pipeline. For more information, see **Check out multiple repositories in your pipeline**.

Checkout

When a pipeline is triggered, Azure Pipelines pulls your source code from the Azure Repos Git repository. You can control various aspects of how this happens.

> **Note**
>
> When you include a checkout step in your pipeline, we run the following command: `git -c fetch --force --tags --prune --prune-tags --progress --no-recurse-submodules origin --depth=1`. If this does not meet your needs, you can choose to exclude built-in checkout by `checkout: none` and then use a script task to perform your own checkout.

Preferred version of Git

The Windows agent comes with its own copy of Git. If you prefer to supply your own Git rather than use the included copy, set `System.PreferGitFromPath` to `true`. This setting is always true on non-Windows agents.

Checkout path

YAML

If you are checking out a single repository, by default, your source code will be checked out into a directory called `s`. For YAML pipelines, you can change this by specifying `checkout` with a `path`. The specified path is relative to `$(Agent.BuildDirectory)`. For example: if the checkout path value is `mycustompath` and `$(Agent.BuildDirectory)` is `C:\agent\_work\1`, then the source code will be checked out into `C:\agent\_work\1\mycustompath`.

If you are using multiple `checkout` steps and checking out multiple repositories, and not explicitly specifying the folder using `path`, each repository is placed in a subfolder of `s` named after the repository. For example if you check out two repositories named `tools` and `code`, the source code will be checked out into `C:\agent\_work\1\s\tools` and `C:\agent\_work\1\s\code`.

Please note that the checkout path value cannot be set to go up any directory levels above `$(Agent.BuildDirectory)`, so `path\..\anotherpath` will result in a valid checkout path (i.e. `C:\agent\_work\1\anotherpath`), but a value like `..\invalidpath` will not (i.e. `C:\agent\_work\invalidpath`).

You can configure the `path` setting in the Checkout step of your pipeline.

YAML

```yaml
steps:
- checkout: self  # self represents the repo where the initial Pipelines
YAML file was found
  clean: boolean  # whether to fetch clean each time
  fetchDepth: number  # the depth of commits to ask Git to fetch
  lfs: boolean  # whether to download Git-LFS files
  submodules: true | recursive  # set to 'true' for a single level of
submodules or 'recursive' to get submodules of submodules
  path: string  # path to check out source code, relative to the agent's
build directory (e.g. \_work\1)
  persistCredentials: boolean  # set to 'true' to leave the OAuth token
in the Git config after the initial fetch
```

Submodules

YAML

You can configure the `submodules` setting in the Checkout step of your pipeline if you want to download files from submodules.

YAML

```
steps:
- checkout: self  # self represents the repo where the initial Pipelines
YAML file was found
  clean: boolean  # whether to fetch clean each time
  fetchDepth: number  # the depth of commits to ask Git to fetch
  lfs: boolean  # whether to download Git-LFS files
  submodules: true | recursive  # set to 'true' for a single level of
submodules or 'recursive' to get submodules of submodules
  path: string  # path to check out source code, relative to the agent's
build directory (e.g. \_work\1)
  persistCredentials: boolean  # set to 'true' to leave the OAuth token
in the Git config after the initial fetch
```

The build pipeline will check out your Git submodules as long as they are:

- **Unauthenticated:** A public, unauthenticated repo with no credentials required to clone or fetch.
- **Authenticated:**
  - Contained in the same project as the Azure Repos Git repo specified above. The same credentials that are used by the agent to get the sources from the main repository are also used to get the sources for submodules.
  - Added by using a URL relative to the main repository. For example
    - This one would be checked out: `git submodule add ../../../FabrikamFiberProject/_git/FabrikamFiber FabrikamFiber`

      In this example the submodule refers to a repo (FabrikamFiber) in the same Azure DevOps organization, but in a different project (FabrikamFiberProject). The same credentials that are used by the agent to get the sources from the main repository are also used to get the sources for submodules. This requires that the job access token has access to the repository in the second project. If you restricted the job access token as explained in the section above, then you won't be able to do this. You can allow the job access token to access the repo in the second project by either (a) explicitly granting access to the project build service account in the second project or (b) using collection-scoped access tokens instead of project-scoped tokens for the entire organization. For more information about these options and their security implications, see Access repositories, artifacts, and other resources.
    - This one would not be checked out: `git submodule add https://fabrikam-fiber@dev.azure.com/fabrikam-`

`fiber/FabrikamFiberProject/_git/FabrikamFiber FabrikamFiber`

Alternative to using the Checkout submodules option

In some cases you can't use the **Checkout submodules** option. You might have a scenario where a different set of credentials are needed to access the submodules. This can happen, for example, if your main repository and submodule repositories aren't stored in the same Azure DevOps organization, or if your job access token does not have access to the repository in a different project.

If you can't use the **Checkout submodules** option, then you can instead use a custom script step to fetch submodules. First, get a personal access token (PAT) and prefix it with `pat:`. Next, base64-encode this prefixed string to create a basic auth token. Finally, add this script to your pipeline:

```
git -c http.https://<url of submodule
repository>.extraheader="AUTHORIZATION: Basic <BASE64_ENCODED_STRING>"
submodule update --init --recursive
```

Be sure to replace "<BASE64_ENCODED_STRING>" with your Base64-encoded "pat:token" string.

Use a secret variable in your project or build pipeline to store the basic auth token that you generated. Use that variable to populate the secret in the above Git command.

> **Note**
>
> **Q: Why can't I use a Git credential manager on the agent? A:** Storing the submodule credentials in a Git credential manager installed on your private build agent is usually not effective as the credential manager may prompt you to re-enter the credentials whenever the submodule is updated. This isn't desirable during automated builds when user interaction isn't possible.

Sync tags

The checkout step uses the `--tags` option when fetching the contents of a Git repository. This causes the server to fetch all tags as well as all objects that are pointed to by those tags. This increases the time to run the task in a pipeline, particularly if you have a large repository with a number of tags. Furthermore, the checkout step syncs

tags even when you enable the shallow fetch option, thereby possibly defeating its purpose. To reduce the amount of data fetched or pulled from a Git repository, Microsoft has added a new option to checkout to control the behavior of syncing tags. This option is available both in classic and YAML pipelines.

Whether to synchronize tags when checking out a repository can be configured in YAML by setting the `fetchTags` property, and in the UI by configuring the **Sync tags** setting.

YAML

You can configure the `fetchTags` setting in the Checkout step of your pipeline.

To configure the setting in YAML, set the `fetchTags` property.

YAML

```yaml
steps:
- checkout: self
  fetchTags: true
```

You can also configure this setting by using the **Sync tags** option in the pipeline settings UI.

1. Edit your YAML pipeline and choose **More actions**, **Triggers**.

2. Choose **YAML**, **Get sources**.

3. Configure the **Sync tags** setting.

> **Note**
>
> If you explicitly set `fetchTags` in your `checkout` step, that setting takes priority over the setting configured in the pipeline settings UI.

Default behavior

- For existing pipelines created before the release of Azure DevOps sprint 209, released in September 2022, the default for syncing tags remains the same as the existing behavior before the **Sync tags** options was added, which is `true`.
- For new pipelines created after Azure DevOps sprint release 209, the default for syncing tags is `false`.

> **Note**

If you explicitly set `fetchTags` in your `checkout` step, that setting takes priority over the setting configured in the pipeline settings UI.

Shallow fetch

You may want to limit how far back in history to download. Effectively this results in `git fetch --depth=n`. If your repository is large, this option might make your build pipeline more efficient. Your repository might be large if it has been in use for a long time and has sizeable history. It also might be large if you added and later deleted large files.

> **Important**
>
> New pipelines created after the **September 2022 Azure DevOps sprint 209 update** have **Shallow fetch** enabled by default and configured with a depth of 1. Previously the default was not to shallow fetch. To check your pipeline, view the **Shallow fetch** setting in the pipeline settings UI as described in the following section.

YAML

You can configure the `fetchDepth` setting in the Checkout step of your pipeline.

YAML

```yaml
steps:
- checkout: self  # self represents the repo where the initial Pipelines
YAML file was found
  clean: boolean  # whether to fetch clean each time
  fetchDepth: number  # the depth of commits to ask Git to fetch
  lfs: boolean  # whether to download Git-LFS files
  submodules: true | recursive  # set to 'true' for a single level of
submodules or 'recursive' to get submodules of submodules
  path: string  # path to check out source code, relative to the agent's
build directory (e.g. \_work\1)
  persistCredentials: boolean  # set to 'true' to leave the OAuth token
in the Git config after the initial fetch
```

You can also configure fetch depth by setting the **Shallow depth** option in the pipeline settings UI.

1. Edit your YAML pipeline and choose **More actions**, **Triggers**.

2. Choose **YAML**, **Get sources**.

3. Configure the **Shallow fetch** setting. Uncheck **Shallow fetch** to disable shallow fetch, or check the box and enter a **Depth** to enable shallow fetch.

> **Note**
>
> If you explicitly set `fetchDepth` in your `checkout` step, that setting takes priority over the setting configured in the pipeline settings UI. Setting `fetchDepth: 0`

fetches all history and overrides the **Shallow fetch** setting.

In these cases this option can help you conserve network and storage resources. It might also save time. The reason it doesn't always save time is because in some situations the server might need to spend time calculating the commits to download for the depth you specify.

> **Note**
>
> When the pipeline is started, the branch to build is resolved to a commit ID. Then, the agent fetches the branch and checks out the desired commit. There is a small window between when a branch is resolved to a commit ID and when the agent performs the checkout. If the branch updates rapidly and you set a very small value for shallow fetch, the commit may not exist when the agent attempts to check it out. If that happens, increase the shallow fetch depth setting.

Don't sync sources

You may want to skip fetching new commits. This option can be useful in cases when you want to:

- Git init, config, and fetch using your own custom options.
- Use a build pipeline to just run automation (for example some scripts) that do not depend on code in version control.

YAML

You can configure the **Don't sync sources** setting in the Checkout step of your pipeline, by setting `checkout: none`.

YAML

```yaml
steps:
- checkout: none  # Don't sync sources
```

> **Note**

When you use this option, the agent also skips running Git commands that clean the repo.

Clean build

You can perform different forms of cleaning the working directory of your self-hosted agent before a build runs.

In general, for faster performance of your self-hosted agents, don't clean the repo. In this case, to get the best performance, make sure you're also building incrementally by disabling any **Clean** option of the task or tool you're using to build.

If you do need to clean the repo (for example to avoid problems caused by residual files from a previous build), your options are below.

> **Note**
>
> Cleaning is not effective if you're using a **Microsoft-hosted agent** because you'll get a new agent every time.

YAML

You can configure the `clean` setting in the Checkout step of your pipeline.

YAML

```yaml
steps:
- checkout: self  # self represents the repo where the initial Pipelines YAML file was found
  clean: boolean  # whether to fetch clean each time
  fetchDepth: number  # the depth of commits to ask Git to fetch
  lfs: boolean  # whether to download Git-LFS files
  submodules: true | recursive  # set to 'true' for a single level of submodules or 'recursive' to get submodules of submodules
  path: string  # path to check out source code, relative to the agent's build directory (e.g. \_work\1)
  persistCredentials: boolean  # set to 'true' to leave the OAuth token in the Git config after the initial fetch
```

When `clean` is set to `true` the build pipeline performs an undo of any changes in `$(Build.SourcesDirectory)`. More specifically, the following Git commands are executed prior to fetching the source.

```
git clean -ffdx
git reset --hard HEAD
```

For more options, you can configure the `workspace` setting of a Job.

YAML

```yaml
jobs:
- job: string  # name of the job, A-Z, a-z, 0-9, and underscore
  ...
  workspace:
    clean: outputs | resources | all # what to clean up before the job
runs
```

This gives the following clean options.

- **outputs**: Same operation as the clean setting described in the previous checkout task, plus: Deletes and recreates `$(Build.BinariesDirectory)`. Note that the `$(Build.ArtifactStagingDirectory)` and `$(Common.TestResultsDirectory)` are always deleted and recreated prior to every build regardless of any of these settings.
- **resources**: Deletes and recreates `$(Build.SourcesDirectory)`. This results in initializing a new, local Git repository for every build.
- **all**: Deletes and recreates `$(Agent.BuildDirectory)`. This results in initializing a new, local Git repository for every build.

Label sources

You may want to label your source code files to enable your team to easily identify which version of each file is included in the completed build. You also have the option to specify whether the source code should be labeled for all builds or only for successful builds.

YAML

You can't currently configure this setting in YAML but you can in the classic editor. When editing a YAML pipeline, you can access the classic editor by choosing either **Triggers** from the YAML editor menu.

From the classic editor, choose **YAML**, choose the **Get sources** task, and then configure the desired properties there.

In the **Tag format** you can use user-defined and predefined variables that have a scope of "All." For example:

```
$(Build.DefinitionName)_$(Build.DefinitionVersion)_$(Build.BuildId)_$(Build.BuildNumber)_$(My.Variable)
```

The first four variables are predefined. `My.Variable` can be defined by you on the variables tab.

The build pipeline labels your sources with a Git tag.

Some build variables might yield a value that is not a valid label. For example, variables such as `$(Build.RequestedFor)` and `$(Build.DefinitionName)` can contain white space. If the value contains white space, the tag is not created.

After the sources are tagged by your build pipeline, an artifact with the Git ref `refs/tags/{tag}` is automatically added to the completed build. This gives your team additional traceability and a more user-friendly way to navigate from the build to the code that was built. The tag is considered a build artifact since it is produced by the build. When the build is deleted either manually or through a retention policy, the tag is also deleted.

FAQ

Problems related to Azure Repos integration fall into three categories:

- **Failing triggers**: My pipeline is not being triggered when I push an update to the repo.
- **Failing checkout**: My pipeline is being triggered, but it fails in the checkout step.
- **Wrong version**: My pipeline runs, but it is using an unexpected version of the source/YAML.

Failing triggers

I just created a new YAML pipeline with CI/PR triggers, but the pipeline is not being triggered.

Follow each of these steps to troubleshoot your failing triggers:

- Are your YAML CI or PR triggers being overridden by pipeline settings in the UI? While editing your pipeline, choose **...** and then **Triggers**.

Check the **Override the YAML trigger from here** setting for the types of trigger (**Continuous integration** or **Pull request validation**) available for your repo.

- Are you configuring the PR trigger in the YAML file or in branch policies for the repo? For an Azure Repos Git repo, you cannot configure a PR trigger in the YAML file. You need to use branch policies.

- Is your pipeline paused or disabled? Open the editor for the pipeline, and then select **Settings** to check. If your pipeline is paused or disabled, then triggers do not work.

- Have you updated the YAML file in the correct branch? If you push an update to a branch, then the YAML file in that same branch governs the CI behavior. If you push an update to a source branch, then the YAML file resulting from merging the source branch with the target branch governs the PR behavior. Make sure that the YAML file in the correct branch has the necessary CI or PR configuration.

- Have you configured the trigger correctly? When you define a YAML trigger, you can specify both include and exclude clauses for branches, tags, and paths. Ensure that the include clause matches the details of your commit and that the exclude

clause doesn't exclude them. Check the syntax for the triggers and make sure that it is accurate.

- Have you used variables in defining the trigger or the paths? That is not supported.
- Did you use templates for your YAML file? If so, make sure that your triggers are defined in the main YAML file. Triggers defined inside template files are not supported.
- Have you excluded the branches or paths to which you pushed your changes? Test by pushing a change to an included path in an included branch. Note that paths in triggers are case-sensitive. Make sure that you use the same case as those of real folders when specifying the paths in triggers.
- Did you just push a new branch? If so, the new branch may not start a new run. See the section "Behavior of triggers when new branches are created".

My CI or PR triggers have been working fine. But, they stopped working now.

First go through the troubleshooting steps in the previous question. Then, follow these additional steps:

- Do you have merge conflicts in your PR? For a PR that did not trigger a pipeline, open it and check whether it has a merge conflict. Resolve the merge conflict.
- Are you experiencing a delay in the processing of push or PR events? You can usually verify this by seeing if the issue is specific to a single pipeline or is common to all pipelines or repos in your project. If a push or a PR update to any of the repos exhibits this symptom, we might be experiencing delays in processing the update events. Check if we are experiencing a service outage on our status page. If the status page shows an issue, then our team must have already started working on it. Check the page frequently for updates on the issue.

I do not want users to override the list of branches for triggers when they update the YAML file. How can I do this?

Users with permissions to contribute code can update the YAML file and include/exclude additional branches. As a result, users can include their own feature or user branch in their YAML file and push that update to a feature or user branch. This may cause the pipeline to be triggered for all updates to that branch. If you want to prevent this behavior, then you can:

1. Edit the pipeline in the Azure Pipelines UI.
2. Navigate to the **Triggers** menu.
3. Select **Override the YAML continuous Integration trigger from here**.
4. Specify the branches to include or exclude for the trigger.

When you follow these steps, any CI triggers specified in the YAML file are ignored.

I have multiple repositories in my YAML pipeline. How do I set up triggers for each repository?

See triggers in Using multiple repositories.

Failing checkout

I see the following error in the log file during checkout step. How do I fix it?

log

```
remote: TF401019: The Git repository with name or identifier XYZ does not
exist or you do not have permissions for the operation you are attempting.
fatal: repository 'XYZ' not found
##[error] Git fetch failed with exit code: 128
```

Follow each of these steps to troubleshoot your failing checkout:

- Does the repository still exist? First, make sure it does by opening it in the **Repos** page.
- Are you accessing the repository using a script? If so, check the Limit job authorization scope to referenced Azure DevOps repositories setting. When **Limit job authorization scope to referenced Azure DevOps repositories** is enabled, you won't be able to check out Azure Repos Git repositories using a script unless they are explicitly referenced first in the pipeline.
- What is the job authorization scope of the pipeline?
  - If the scope is **collection**:
    - This may be an intermittent error. Re-run the pipeline.
    - Someone may have removed the access to **Project Collection Build Service account**.
      - Go to **Project settings** for the project in which the repository exists. Select **Repos** > **Repositories** > **specific repository**, and then **Security**.

        - Check if **Project Collection Build Service (your-collection-name)** exists in the list of users.
        - Check if that account has **Create tag** and **Read** access.

    - If the scope is **project**:
        - Is the repo in the same project as the pipeline?
            - Yes:
                - This may be an intermittent error. Re-run the pipeline.
                - Someone may have removed the access to **Project Build Service account**.
                - Go to **Project settings** for the project in which the repository exists. Select **Repos** > **Repositories** > **specific repository**, and then **Security**.
                - Check if **your-project-name Build Service (your-collection-name)** exists in the list of users.
                - Check if that account has **Create tag** and **Read** access.
            - No:
                - Is your pipeline in a public project?
                - Yes: You cannot access resources outside of your public project. Make the project private.
                - No: You need to configure permissions to access another repo in the same project collection.

Wrong version

A wrong version of the YAML file is being used in the pipeline. Why is that?

- For CI triggers, the YAML file that is in the branch you are pushing is evaluated to see if a CI build should be run.
- For PR triggers, the YAML file resulting from merging the source and target branches of the PR is evaluated to see if a PR build should be run.

Related articles

- Scheduled triggers
- Pipeline completion triggers

Build GitHub repositories

Article • 11/27/2023

Azure DevOps Services

Azure Pipelines can automatically build and validate every pull request and commit to your GitHub repository. This article describes how to configure the integration between GitHub and Azure Pipelines.

If you're new to pipelines integration with GitHub, follow the steps in Create your first pipeline. Come back to this article to learn more about configuring and customizing the integration between GitHub and Azure Pipelines.

Organizations and users

GitHub and Azure Pipelines are two independent services that integrate well together. Each of them have their own organization and user management. This section makes a recommendation on how to replicate the organization and users from GitHub to Azure Pipelines.

Organizations

GitHub's structure consists of **organizations and user accounts** that contain **repositories**. See GitHub's documentation .

Azure DevOps' structure consists of **organizations** that contain **projects**. See Plan your organizational structure.

Azure DevOps can reflect your GitHub structure with:

- A DevOps **organization** for your GitHub **organization or user account**
- DevOps **Projects** for your GitHub **repositories**

To set up an identical structure in Azure DevOps:

1. Create a DevOps organization named after your GitHub organization or user account. It will have a URL like `https://dev.azure.com/your-organization`.
2. In the DevOps organization, create projects named after your repositories. They'll have URLs like `https://dev.azure.com/your-organization/your-repository`.

3. In the DevOps Project, create pipelines named after the GitHub organization and repository they build, such as `your-organization.your-repository`. Then, it's clear which repositories they're for.

Following this pattern, your GitHub repositories and Azure DevOps Projects will have matching URL paths. For example:

Expand table

Service	URL
GitHub	`https://github.com/python/cpython`
Azure DevOps	`https://dev.azure.com/python/cpython`

Users

Your GitHub users don't automatically get access to Azure Pipelines. Azure Pipelines is unaware of GitHub identities. For this reason, there's no way to configure Azure Pipelines to automatically notify users of a build failure or a PR validation failure using their GitHub identity and email address. You must explicitly create new users in Azure Pipelines to replicate GitHub users. Once you create new users, you can configure their permissions in Azure DevOps to reflect their permissions in GitHub. You can also configure notifications in DevOps using their DevOps identity.

GitHub organization roles

GitHub organization member roles are found at `https://github.com/orgs/your-organization/people` (replace `your-organization`).

DevOps organization member permissions are found at `https://dev.azure.com/your-organization/_settings/security` (replace `your-organization`).

Roles in a GitHub organization and equivalent roles in an Azure DevOps organization are shown below.

Expand table

GitHub organization role	DevOps organization equivalent
Owner	Member of `Project Collection Administrators`

GitHub organization role	DevOps organization equivalent
Billing manager	Member of `Project Collection Administrators`
Member	Member of `Project Collection Valid Users`. By default, the Member group lacks permission to create new projects. To change the permission, set the group's `Create new projects` permission to `Allow`, or create a new group with permissions you need.

GitHub user account roles

A GitHub user account has one role, which is ownership of the account.

DevOps organization member permissions are found at `https://dev.azure.com/your-organization/_settings/security` (replace `your-organization`).

The GitHub user account role maps to DevOps organization permissions as follows.

Expand table

GitHub user account role	DevOps organization equivalent
Owner	Member of `Project Collection Administrators`

GitHub repository permissions

GitHub repository permissions are found at `https://github.com/your-organization/your-repository/settings/collaboration` (replace `your-organization` and `your-repository`).

DevOps project permissions are found at `https://dev.azure.com/your-organization/your-project/_settings/security` (replace `your-organization` and `your-project`).

Equivalent permissions between GitHub repositories and Azure DevOps Projects are as follows.

Expand table

GitHub repository permission	DevOps project equivalent
Admin	Member of `Project Administrators`

GitHub repository permission	DevOps project equivalent
Write	Member of `Contributors`
Read	Member of `Readers`

If your GitHub repository grants permission to teams, you can create matching teams in the `Teams` section of your Azure DevOps project settings. Then, add the teams to the security groups above, just like users.

Pipeline-specific permissions

To grant permissions to users or teams for specific pipelines in a DevOps project, follow these steps:

1. Visit the project's Pipelines page (for example, `https://dev.azure.com/your-organization/your-project/_build`).
2. Select the pipeline for which to set specific permissions.
3. From the '...' context menu, select **Security**.
4. Select **Add...** to add a specific user, team, or group and customize their permissions for the pipeline.

Access to GitHub repositories

YAML

You create a new pipeline by first selecting a GitHub repository and then a YAML file in that repository. The repository in which the YAML file is present is called `self` repository. By default, this is the repository that your pipeline builds.

You can later configure your pipeline to check out a different repository or multiple repositories. To learn how to do this, see multi-repo checkout.

Azure Pipelines must be granted access to your repositories to trigger their builds, and fetch their code during builds.

There are three authentication types for granting Azure Pipelines access to your GitHub repositories while creating a pipeline.

Expand table

Authentication type	Pipelines run using	Works with GitHub Checks
1. GitHub App	The Azure Pipelines identity	Yes
2. OAuth	Your personal GitHub identity	No
3. Personal access token (PAT)	Your personal GitHub identity	No

GitHub app authentication

The Azure Pipelines GitHub App is the **recommended** authentication type for continuous integration pipelines. After you install the GitHub App in your GitHub account or organization, your pipeline will run without using your personal GitHub identity. Builds and GitHub status updates will be performed using the Azure Pipelines identity. The app works with GitHub Checks to display build, test, and code coverage results in GitHub.

To use the GitHub App, install it in your GitHub organization or user account for some or all repositories. The GitHub App can be installed and uninstalled from the app's homepage.

After installation, the GitHub App will become Azure Pipelines' default method of authentication to GitHub (instead of OAuth) when pipelines are created for the repositories.

If you install the GitHub App for all repositories in a GitHub organization, you don't need to worry about Azure Pipelines sending mass emails or automatically setting up pipelines on your behalf. As an alternative to installing the app for all repositories, repository admins can install it one at a time for individual repositories. This requires more work for admins, but has no advantage nor disadvantage.

Permissions needed in GitHub

Installation of Azure Pipelines GitHub app requires you to be a GitHub organization owner or repository admin. In addition, to create a pipeline for a GitHub repository with continuous integration and pull request triggers, you must have the required GitHub permissions configured. Otherwise, **the repository will not appear** in the repository list while creating a pipeline. Depending on the authentication type and ownership of the repository, ensure that the appropriate access is configured.

- If the repo is in your personal GitHub account, install the Azure Pipelines GitHub App in your personal GitHub account, and you'll be able to list this repository when creating the pipeline in Azure Pipelines.

- If the repo is in someone else's personal GitHub account, the other person must install the Azure Pipelines GitHub App in their personal GitHub account. You must be added as a collaborator in the repository's settings under "Collaborators". Accept the invitation to be a collaborator using the link that is emailed to you. Once you've done so, you can create a pipeline for that repository.
- If the repo is in a GitHub organization that you own, install the Azure Pipelines GitHub App in the GitHub organization. You must also be added as a collaborator, or your team must be added, in the repository's settings under "Collaborators and teams".
- If the repo is in a GitHub organization that someone else owns, a GitHub organization owner or repository admin must install the Azure Pipelines GitHub App in the organization. You must be added as a collaborator, or your team must be added, in the repository's settings under "Collaborators and teams". Accept the invitation to be a collaborator using the link that is emailed to you.

GitHub App permissions

The GitHub App requests the following permissions during installation:

Expand table

Permission	How Azure Pipelines uses the permission
Write access to code	Only upon your deliberate action, Azure Pipelines will simplify creating a pipeline by committing a YAML file to a selected branch of your GitHub repository.
Read access to metadata	Azure Pipelines will retrieve GitHub metadata for displaying the repository, branches, and issues associated with a build in the build's summary.
Read and write access to checks	Azure Pipelines will read and write its own build, test, and code coverage results to be displayed in GitHub.
Read and write access to pull requests	Only upon your deliberate action, Azure Pipelines will simplify creating a pipeline by creating a pull request for a YAML file that was committed to a selected branch of your GitHub repository. Pipelines retrieves request metadata to display in build summaries associated with pull requests.

Troubleshooting GitHub App installation

GitHub may display an error such as:

`You do not have permission to modify this app on your-organization. Please contact an Organization Owner.`

This means that the GitHub App is likely already installed for your organization. When you create a pipeline for a repository in the organization, the GitHub App will automatically be used to connect to GitHub.

Create pipelines in multiple Azure DevOps organizations and projects

Once the GitHub App is installed, pipelines can be created for the organization's repositories in different Azure DevOps organizations and projects. However, if you create pipelines for a single repository in multiple Azure DevOps organizations, only the first organization's pipelines can be automatically triggered by GitHub commits or pull requests. Manual or scheduled builds are still possible in secondary Azure DevOps organizations.

OAuth authentication

OAuth is the simplest authentication type to get started with for repositories in your personal GitHub account. GitHub status updates will be performed on behalf of your personal GitHub identity. For pipelines to keep working, your repository access must remain active. Some GitHub features, like Checks, are unavailable with OAuth and require the GitHub App.

To use OAuth, select **Choose a different connection** below the list of repositories while creating a pipeline. Then, select **Authorize** to sign into GitHub and authorize with OAuth. An OAuth connection will be saved in your Azure DevOps project for later use, and used in the pipeline being created.

Permissions needed in GitHub

To create a pipeline for a GitHub repository with continuous integration and pull request triggers, you must have the required GitHub permissions configured. Otherwise, **the repository will not appear** in the repository list while creating a pipeline. Depending on the authentication type and ownership of the repository, ensure that the appropriate access is configured.

- If the repo is in your personal GitHub account, at least once, authenticate to GitHub with OAuth using your personal GitHub account credentials. This can be done in Azure DevOps project settings under Pipelines > Service connections >

New service connection > GitHub > Authorize. Grant Azure Pipelines access to your repositories under "Permissions" here.

- If the repo is in someone else's personal GitHub account, at least once, the other person must authenticate to GitHub with OAuth using their personal GitHub account credentials. This can be done in Azure DevOps project settings under Pipelines > Service connections > New service connection > GitHub > Authorize. The other person must grant Azure Pipelines access to their repositories under "Permissions" here. You must be added as a collaborator in the repository's settings under "Collaborators". Accept the invitation to be a collaborator using the link that is emailed to you.

- If the repo is in a GitHub organization that you own, at least once, authenticate to GitHub with OAuth using your personal GitHub account credentials. This can be done in Azure DevOps project settings under Pipelines > Service connections > New service connection > GitHub > Authorize. Grant Azure Pipelines access to your organization under "Organization access" here. You must be added as a collaborator, or your team must be added, in the repository's settings under "Collaborators and teams".

- If the repo is in a GitHub organization that someone else owns, at least once, a GitHub organization owner must authenticate to GitHub with OAuth using their personal GitHub account credentials. This can be done in Azure DevOps project settings under Pipelines > Service connections > New service connection > GitHub > Authorize. The organization owner must grant Azure Pipelines access to the organization under "Organization access" here. You must be added as a collaborator, or your team must be added, in the repository's settings under "Collaborators and teams". Accept the invitation to be a collaborator using the link that is emailed to you.

Revoke OAuth access

After authorizing Azure Pipelines to use OAuth, to later revoke it and prevent further use, visit OAuth Apps in your GitHub settings. You can also delete it from the list of GitHub service connections in your Azure DevOps project settings.

Personal access token (PAT) authentication

PATs are effectively the same as OAuth, but allow you to control which permissions are granted to Azure Pipelines. Builds and GitHub status updates will be performed on

behalf of your personal GitHub identity. For builds to keep working, your repository access must remain active.

To create a PAT, visit Personal access tokens in your GitHub settings. The required permissions are `repo`, `admin:repo_hook`, `read:user`, and `user:email`. These are the same permissions required when using OAuth above. Copy the generated PAT to the clipboard and paste it into a new GitHub service connection in your Azure DevOps project settings. For future recall, name the service connection after your GitHub username. It will be available in your Azure DevOps project for later use when creating pipelines.

Permissions needed in GitHub

To create a pipeline for a GitHub repository with continuous integration and pull request triggers, you must have the required GitHub permissions configured. Otherwise, **the repository will not appear** in the repository list while creating a pipeline. Depending on the authentication type and ownership of the repository, ensure that the following access is configured.

- If the repo is in your personal GitHub account, the PAT must have the required access scopes under Personal access tokens: `repo`, `admin:repo_hook`, `read:user`, and `user:email`.
- If the repo is in someone else's personal GitHub account, the PAT must have the required access scopes under Personal access tokens: `repo`, `admin:repo_hook`, `read:user`, and `user:email`. You must be added as a collaborator in the repository's settings under "Collaborators". Accept the invitation to be a collaborator using the link that is emailed to you.
- If the repo is in a GitHub organization that you own, the PAT must have the required access scopes under Personal access tokens: `repo`, `admin:repo_hook`, `read:user`, and `user:email`. You must be added as a collaborator, or your team must be added, in the repository's settings under "Collaborators and teams".
- If the repo is in a GitHub organization that someone else owns, the PAT must have the required access scopes under Personal access tokens: `repo`, `admin:repo_hook`, `read:user`, and `user:email`. You must be added as a collaborator, or your team must be added, in the repository's settings under "Collaborators and teams". Accept the invitation to be a collaborator using the link that is emailed to you.

Revoke PAT access

After authorizing Azure Pipelines to use a PAT, to later delete it and prevent further use, visit Personal access tokens in your GitHub settings. You can also delete it from the list of GitHub service connections in your Azure DevOps project settings.

CI triggers

Continuous integration (CI) triggers cause a pipeline to run whenever you push an update to the specified branches or you push specified tags.

YAML

YAML pipelines are configured by default with a CI trigger on all branches.

Branches

You can control which branches get CI triggers with a simple syntax:

YAML

```yaml
trigger:
- main
- releases/*
```

You can specify the full name of the branch (for example, `main`) or a wildcard (for example, `releases/*`). See Wildcards for information on the wildcard syntax.

> **Note**
>
> You cannot use **variables** in triggers, as variables are evaluated at runtime (after the trigger has fired).

> **Note**
>
> If you use **templates** to author YAML files, then you can only specify triggers in the main YAML file for the pipeline. You cannot specify triggers in the template files.

For more complex triggers that use `exclude` or `batch`, you must use the full syntax as shown in the following example.

YAML

```yaml
# specific branch build
trigger:
  branches:
    include:
    - main
    - releases/*
    exclude:
    - releases/old*
```

In the above example, the pipeline will be triggered if a change is pushed to `main` or to any releases branch. However, it won't be triggered if a change is made to a releases branch that starts with `old`.

If you specify an `exclude` clause without an `include` clause, then it is equivalent to specifying `*` in the `include` clause.

In addition to specifying branch names in the `branches` lists, you can also configure triggers based on tags by using the following format:

YAML

```yaml
trigger:
  branches:
    include:
      - refs/tags/{tagname}
    exclude:
      - refs/tags/{othertagname}
```

If you don't specify any triggers, the default is as if you wrote:

YAML

```yaml
trigger:
  branches:
    include:
    - '*'  # must quote since "*" is a YAML reserved character; we want
a string
```

ⓘ **Important**

When you specify a trigger, it replaces the default implicit trigger, and only pushes to branches that are explicitly configured to be included will trigger a pipeline. Includes are processed first, and then excludes are removed from that list.

Batching CI runs

If you have many team members uploading changes often, you may want to reduce the number of runs you start. If you set `batch` to `true`, when a pipeline is running, the system waits until the run is completed, then starts another run with all changes that have not yet been built.

YAML

```yaml
# specific branch build with batching
trigger:
  batch: true
  branches:
    include:
    - main
```

> **Note**
>
> `batch` is not supported in repository resource triggers.

To clarify this example, let us say that a push `A` to `main` caused the above pipeline to run. While that pipeline is running, additional pushes `B` and `C` occur into the repository. These updates do not start new independent runs immediately. But after the first run is completed, all pushes until that point of time are batched together and a new run is started.

> **Note**
>
> If the pipeline has multiple jobs and stages, then the first run should still reach a terminal state by completing or skipping all its jobs and stages before the second run can start. For this reason, you must exercise caution when using this feature in a pipeline with multiple stages or approvals. If you wish to batch your builds in such cases, it is recommended that you split your CI/CD process into two pipelines - one for build (with batching) and one for deployments.

Paths

You can specify file paths to include or exclude.

YAML

```yaml
# specific path build
trigger:
  branches:
    include:
    - main
    - releases/*
  paths:
    include:
    - docs
    exclude:
    - docs/README.md
```

When you specify paths, you must explicitly specify branches to trigger on if you are using Azure DevOps Server 2019.1 or lower. You can't trigger a pipeline with only a path filter; you must also have a branch filter, and the changed files that match the path filter must be from a branch that matches the branch filter. If you are using Azure DevOps Server 2020 or newer, you can omit `branches` to filter on all branches in conjunction with the path filter.

Wilds cards are supported for path filters. For instance, you can include all paths that match `src/app/**/myapp*`. You can use wild card characters (`**`, `*`, or `?`) when specifying path filters.

- Paths are always specified relative to the root of the repository.
- If you don't set path filters, then the root folder of the repo is implicitly included by default.
- If you exclude a path, you cannot also include it unless you qualify it to a deeper folder. For example if you exclude */tools* then you could include */tools/trigger-runs-on-these*
- The order of path filters doesn't matter.
- Paths in Git *are case-sensitive*. Be sure to use the same case as the real folders.
- You cannot use variables in paths, as variables are evaluated at runtime (after the trigger has fired).

Tags

In addition to specifying tags in the `branches` lists as covered in the previous section, you can directly specify tags to include or exclude:

YAML

```yaml
# specific tag
trigger:
  tags:
    include:
    - v2.*
    exclude:
    - v2.0
```

If you don't specify any tag triggers, then by default, tags will not trigger pipelines.

> **Important**
>
> If you specify tags in combination with branch filters, the trigger will fire if either the branch filter is satisfied or the tag filter is satisfied. For example, if a pushed tag satisfies the branch filter, the pipeline triggers even if the tag is excluded by the tag filter, because the push satisfied the branch filter.

Opting out of CI

Disabling the CI trigger

You can opt out of CI triggers entirely by specifying `trigger: none`.

YAML

```yaml
# A pipeline with no CI trigger
trigger: none
```

> **Important**
>
> When you push a change to a branch, the YAML file in that branch is evaluated to determine if a CI run should be started.

Skipping CI for individual commits

You can also tell Azure Pipelines to skip running a pipeline that a push would normally trigger. Just include `[skip ci]` in the message or description of any of the commits that

are part of a push, and Azure Pipelines will skip running CI for this push. You can also use any of the following variations.

- `[skip ci]` or `[ci skip]`
- `skip-checks: true` or `skip-checks:true`
- `[skip azurepipelines]` or `[azurepipelines skip]`
- `[skip azpipelines]` or `[azpipelines skip]`
- `[skip azp]` or `[azp skip]`
- `***NO_CI***`

Using the trigger type in conditions

It is a common scenario to run different steps, jobs, or stages in your pipeline depending on the type of trigger that started the run. You can do this using the system variable `Build.Reason`. For example, add the following condition to your step, job, or stage to exclude it from PR validations.

```
condition: and(succeeded(), ne(variables['Build.Reason'], 'PullRequest'))
```

Behavior of triggers when new branches are created

It is common to configure multiple pipelines for the same repository. For instance, you may have one pipeline to build the docs for your app and another to build the source code. You may configure CI triggers with appropriate branch filters and path filters in each of these pipelines. For instance, you may want one pipeline to trigger when you push an update to the `docs` folder, and another one to trigger when you push an update to your application code. In these cases, you need to understand how the pipelines are triggered when a new branch is created.

Here is the behavior when you push a new branch (that matches the branch filters) to your repository:

- If your pipeline has path filters, it will be triggered only if the new branch has changes to files that match that path filter.
- If your pipeline does not have path filters, it will be triggered even if there are no changes in the new branch.

Wildcards

When specifying a branch, tag, or path, you may use an exact name or a wildcard. Wildcards patterns allow `*` to match zero or more characters and `?` to match a single

character.

- If you start your pattern with `*` in a YAML pipeline, you must wrap the pattern in quotes, like `"*-releases"`.
- For branches and tags:
  - A wildcard may appear anywhere in the pattern.
- For paths:
  - In Azure DevOps Server 2022 and higher, including Azure DevOps Services, a wildcard may appear anywhere within a path pattern and you may use `*` or `?`.
  - In Azure DevOps Server 2020 and lower, you may include `*` as the final character, but it doesn't do anything differently from specifying the directory name by itself. You may **not** include `*` in the middle of a path filter, and you may not use `?`.

YAML

```yaml
trigger:
  branches:
    include:
    - main
    - releases/*
    - feature/*
    exclude:
    - releases/old*
    - feature/*-working
  paths:
    include:
    - docs/*.md
```

PR triggers

Pull request (PR) triggers cause a pipeline to run whenever a pull request is opened with one of the specified target branches, or when updates are made to such a pull request.

YAML

Branches

You can specify the target branches when validating your pull requests. For example, to validate pull requests that target `main` and `releases/*`, you can use the following `pr` trigger.

YAML

```yaml
pr:
- main
- releases/*
```

This configuration starts a new run the first time a new pull request is created, and after every update made to the pull request.

You can specify the full name of the branch (for example, `main`) or a wildcard (for example, `releases/*`).

> **Note**
>
> You cannot use **variables** in triggers, as variables are evaluated at runtime (after the trigger has fired).

> **Note**
>
> If you use **templates** to author YAML files, then you can only specify triggers in the main YAML file for the pipeline. You cannot specify triggers in the template files.

GitHub creates a new *ref* when a pull request is created. The ref points to a *merge commit*, which is the merged code between the source and target branches of the pull request. The PR validation pipeline builds the commit that this *ref* points to. This means that the YAML file that is used to run the pipeline is also a merge between the source and the target branch. As a result, the changes you make to the YAML file in source branch of the pull request can override the behavior defined by the YAML file in target branch.

If no `pr` triggers appear in your YAML file, pull request validations are automatically enabled for all branches, as if you wrote the following `pr` trigger. This configuration triggers a build when any pull request is created, and when commits come into the source branch of any active pull request.

YAML

```yaml
pr:
  branches:
    include:
```

```
    - '*'  # must quote since "*" is a YAML reserved character; we want
a string
```

> **Important**
>
> When you specify a `pr` trigger with a subset of branches, a pipeline is triggered only when updates are pushed to those branches.

For more complex triggers that need to exclude certain branches, you must use the full syntax as shown in the following example. In this example, pull requests are validated that target `main` or `releases/*` and the branch `releases/old*` is excluded.

YAML

```yaml
# specific branch
pr:
  branches:
    include:
    - main
    - releases/*
    exclude:
    - releases/old*
```

Paths

You can specify file paths to include or exclude. For example:

YAML

```yaml
# specific path
pr:
  branches:
    include:
    - main
    - releases/*
  paths:
    include:
    - docs
    exclude:
    - docs/README.md
```

> **Tips:**
>
> - Azure Pipelines posts a neutral status back to GitHub when it decides not to run a validation build because of a path exclusion rule. This provides a

clear direction to GitHub indicating that Azure Pipelines has completed its processing. For more information, see Post neutral status to GitHub when a build is skipped.

- Wild cards are now supported with path filters.
- Paths are always specified relative to the root of the repository.
- If you don't set path filters, then the root folder of the repo is implicitly included by default.
- If you exclude a path, you cannot also include it unless you qualify it to a deeper folder. For example if you exclude */tools* then you could include */tools/trigger-runs-on-these*
- The order of path filters doesn't matter.
- Paths in Git *are case-sensitive*. Be sure to use the same case as the real folders.
- You cannot use variables in paths, as variables are evaluated at runtime (after the trigger has fired).
- Azure Pipelines posts a neutral status back to GitHub when it decides not to run a validation build because of a path exclusion rule.

Multiple PR updates

You can specify whether more updates to a PR should cancel in-progress validation runs for the same PR. The default is `true`.

YAML

```yaml
# auto cancel false
pr:
  autoCancel: false
  branches:
    include:
    - main
```

Draft PR validation

By default, pull request triggers fire on draft pull requests and pull requests that are ready for review. To disable pull request triggers for draft pull requests, set the `drafts` property to `false`.

YAML

```yaml
pr:
  autoCancel: boolean # indicates whether additional pushes to a PR
should cancel in-progress runs for the same PR. Defaults to true
```

```
  branches:
    include: [ string ] # branch names which will trigger a build
    exclude: [ string ] # branch names which will not
  paths:
    include: [ string ] # file paths which must match to trigger a build
    exclude: [ string ] # file paths which will not trigger a build
  drafts: boolean # whether to build draft PRs, defaults to true
```

Opting out of PR validation

You can opt out of pull request validation entirely by specifying `pr: none`.

YAML

```yaml
# no PR triggers
pr: none
```

For more information, see PR trigger in the YAML schema.

> **Note**
>
> If your `pr` trigger isn't firing, follow the troubleshooting steps in the **FAQ**.

> **Note**
>
> **Draft pull requests** do not trigger a pipeline.

If you have an open PR and you push changes to its source branch, multiple pipelines may run:

- The pipelines that have a PR trigger on the PR's target branch will run on the *merge commit* (the merged code between the source and target branches of the pull request), regardless if there exist pushed commits whose messages or descriptions contain `[skip ci]` (or any of its variants).
- The pipelines triggered by changes to the PR's source branch, if there are **no** pushed commits whose messages or descriptions contain `[skip ci]` (or any of its variants). If at least one pushed commit contains `[skip ci]`, the pipelines will not run.

Finally, after you merge the PR, Azure Pipelines will run the CI pipelines triggered by pushes to the target branch, if the merge commit's message or description doesn't

contain `[skip ci]` (or any of its variants).

Protected branches

You can run a validation build with each commit or pull request that targets a branch, and even prevent pull requests from merging until a validation build succeeds.

To configure mandatory validation builds for a GitHub repository, you must be its owner, a collaborator with the Admin role, or a GitHub organization member with the Write role.

1. First, create a pipeline for the repository and build it at least once so that its status is posted to GitHub, thereby making GitHub aware of the pipeline's name.

2. Next, follow GitHub's documentation for configuring protected branches in the repository's settings.

   For the status check, select the name of your pipeline in the **Status checks** list.

> **Important**
>
> If your pipeline doesn't show up in this list, please ensure the following:
>
> - You are using **GitHub app authentication**
> - Your pipeline has run at least once in the last week

Contributions from external sources

If your GitHub repository is open source, you can make your Azure DevOps project public so that anyone can view your pipeline's build results, logs, and test results without signing in. When users outside your organization fork your repository and

submit pull requests, they can view the status of builds that automatically validate those pull requests.

You should keep in mind the following considerations when using Azure Pipelines in a public project when accepting contributions from external sources.

- Access restrictions
- Validate contributions from forks
- Important security considerations

Access restrictions

Be aware of the following access restrictions when you're running pipelines in Azure DevOps public projects:

- **Secrets:** By default, secrets associated with your pipeline aren't made available to pull request validations of forks. See Validate contributions from forks.
- **Cross-project access:** All pipelines in an Azure DevOps public project run with an access token restricted to the project. Pipelines in a public project can access resources such as build artifacts or test results only within the project and not in other projects of the Azure DevOps organization.
- **Azure Artifacts packages:** If your pipelines need access to packages from Azure Artifacts, you must explicitly grant permission to the **Project Build Service** account to access the package feeds.

Contributions from forks

> **Important**
>
> These settings affect the security of your pipeline.

When you create a pipeline, it's automatically triggered for pull requests from forks of your repository. You can change this behavior, carefully considering how it affects security. To enable or disable this behavior:

1. Go to your Azure DevOps project. Select **Pipelines**, locate your pipeline, and select **Edit**.
2. Select the **Triggers** tab. After enabling the **Pull request trigger**, enable or disable the **Build pull requests from forks of this repository** check box.

By default with GitHub pipelines, secrets associated with your build pipeline aren't made available to pull request builds of forks. These secrets are enabled by default with

GitHub Enterprise Server pipelines. Secrets include:

- A security token with access to your GitHub repository.
- These items, if your pipeline uses them:
  - Service connection credentials
  - Files from the secure files library
  - Build variables marked **secret**

To bypass this precaution on GitHub pipelines, enable the **Make secrets available to builds of forks** check box. Be aware of this setting's effect on security.

> **Note**
>
> When you enable fork builds to access secrets, Azure Pipelines by default restricts the access token used for fork builds. It has more limited access to open resources than a normal access token. To give fork builds the same permissions as regular builds, enable the **Make fork builds have the same permissions as regular builds** setting.

For more information, see Repository protection - Forks.

You can define centrally how pipelines build PRs from forked GitHub repositories using the **Limit building pull requests from forked GitHub repositories** control. It's available at organization and project level. You can choose to:

- Disable building pull requests from forked repositories
- Securely build pull requests from forked repositories
- Customize rules for building pull requests from forked repositories

Starting with Sprint 229, to improve the security of your pipelines, Azure Pipelines no longer automatically builds pull requests from forked GitHub repositories. For new projects and organizations, the default value of the **Limit building pull requests from**

forked GitHub repositories setting is **Disable building pull requests from forked repositories**.

When you choose the **Securely build pull requests from forked repositories** option, all pipelines, organization or project-wide, *cannot* make secrets available to builds of PRs from forked repositories, *cannot* make these builds have the same permissions as normal builds, and *must* be triggered by a PR comment. Projects can still decide to *not* allow pipelines to build such PRs.

When you choose the **Customize** option, you can define how to restrict pipeline settings. For example, you can ensure that all pipelines require a comment in order to build a PR from a forked GitHub repo, when the PR belongs to non-team members and non-contributors. But, you can choose to allow them to make secrets available to such builds. Projects can decide to *not* allow pipelines to build such PRs, or to build them securely, or have even more restrictive settings than what is specified at the organization level.

The control is off for existing organizations. Starting September 2023, new organizations have **Securely build pull requests from forked repositories** turned on by default.

Important security considerations

A GitHub user can fork your repository, change it, and create a pull request to propose changes to your repository. This pull request could contain malicious code to run as part of your triggered build. Such code can cause harm in the following ways:

- Leak secrets from your pipeline. To mitigate this risk, don't enable the **Make secrets available to builds of forks** check box if your repository is public or untrusted users can submit pull requests that automatically trigger builds. This option is disabled by default.
- Compromise the machine running the agent to steal code or secrets from other pipelines. To mitigate this:
  - Use a Microsoft-hosted agent pool to build pull requests from forks. Microsoft-hosted agent machines are immediately deleted after they complete a build, so there's no lasting impact if they're compromised.
  - If you must use a self-hosted agent, don't store any secrets or perform other builds and releases that use secrets on the same agent, unless your repository is private and you trust pull request creators.

Comment triggers

Repository collaborators can comment on a pull request to manually run a pipeline. Here are a few common reasons for why you might want to do this:

- You may not want to automatically build pull requests from unknown users until their changes can be reviewed. You want one of your team members to first review their code and then run the pipeline. This is commonly used as a security measure when building contributed code from forked repositories.
- You may want to run an optional test suite or one more validation build.

To enable comment triggers, you must follow the following two steps:

1. Enable pull request triggers for your pipeline, and make sure that you didn't exclude the target branch.
2. In the Azure Pipelines web portal, edit your pipeline and choose **More actions**, **Triggers**. Then, under **Pull request validation**, enable **Require a team member's comment before building a pull request**.

    - Choose **On all pull requests** to require a team member's comment before building a pull request. With this workflow, a team member reviews the pull request and triggers the build with a comment once the pull request is deemed safe.
    - Choose **Only on pull requests from non-team members** to require a team member's comment only when a PR is made by a non-team member. In this workflow, a team member doesn't need a secondary team member's review to trigger a build.

With these two changes, the pull request validation build won't be triggered automatically, unless **Only on pull requests from non-team members** is selected and the PR is made by a team member. Only repository owners and collaborators with 'Write' permission can trigger the build by commenting on the pull request with `/AzurePipelines run` or `/AzurePipelines run <pipeline-name>`.

The following commands can be issued to Azure Pipelines in comments:

Expand table

Command	Result
`/AzurePipelines help`	Display help for all supported commands.
`/AzurePipelines help <command-name>`	Display help for the specified command.

Command	Result
`/AzurePipelines run`	Run all pipelines that are associated with this repository and whose triggers don't exclude this pull request.
`/AzurePipelines run <pipeline-name>`	Run the specified pipeline unless its triggers exclude this pull request.

> **Note**
>
> For brevity, you can comment using `/azp` instead of `/AzurePipelines`.

> **Important**
>
> Responses to these commands will appear in the pull request discussion only if your pipeline uses the **Azure Pipelines GitHub App**.

Troubleshoot pull request comment triggers

If you have the necessary repository permissions, but pipelines aren't getting triggered by your comments, make sure that your membership is **public** in the repository's organization, or directly add yourself as a repository collaborator. Pipelines can't see private organization members unless they are direct collaborators or belong to a team that is a direct collaborator. You can change your GitHub organization membership from private to public here (replace `Your-Organization` with your organization name): `https://github.com/orgs/Your-Organization/people`.

Informational runs

An informational run tells you Azure DevOps failed to retrieve a YAML pipeline's source code. Source code retrieval happens in response to external events, for example, a pushed commit. It also happens in response to internal triggers, for example, to check if there are code changes and start a scheduled run or not. Source code retrieval can fail for multiple reasons, with a frequent one being request throttling by the git repository provider. The existence of an informational run doesn't necessarily mean Azure DevOps was going to run the pipeline.

An informational run looks like in the following screenshot.

You can recognize an informational run by the following attributes:

- Status is `Canceled`
- Duration is `< 1s`
- Run name contains one of the following texts:
  - `Could not retrieve file content for {file_path} from repository {repo_name} hosted on {host} using commit {commit_sha}.`
  - `Could not retrieve content for object {commit_sha} from repository {repo_name} hosted on {host}.`
  - `Could not retrieve the tree object {tree_sha} from the repository {repo_name} hosted on {host}.`
  - `Could not find {file_path} from repository {repo_name} hosted on {host} using version {commit_sha}. One of the directories in the path contains too many files or subdirectories.`
- Run name generally contains the BitBucket / GitHub error that caused the YAML pipeline load to fail
- No stages / jobs / steps

Learn more about informational runs.

Checkout

When a pipeline is triggered, Azure Pipelines pulls your source code from the Azure Repos Git repository. You can control various aspects of how this happens.

> **Note**
>
> When you include a checkout step in your pipeline, we run the following command: `git -c fetch --force --tags --prune --prune-tags --progress --no-recurse-submodules origin --depth=1`. If this does not meet your needs, you can choose to exclude built-in checkout by `checkout: none` and then use a script task to perform your own checkout.

Preferred version of Git

The Windows agent comes with its own copy of Git. If you prefer to supply your own Git rather than use the included copy, set `System.PreferGitFromPath` to `true`. This setting is always true on non-Windows agents.

Checkout path

YAML

If you are checking out a single repository, by default, your source code will be checked out into a directory called `s`. For YAML pipelines, you can change this by specifying `checkout` with a `path`. The specified path is relative to `$(Agent.BuildDirectory)`. For example: if the checkout path value is `mycustompath` and `$(Agent.BuildDirectory)` is `C:\agent\_work\1`, then the source code will be checked out into `C:\agent\_work\1\mycustompath`.

If you are using multiple `checkout` steps and checking out multiple repositories, and not explicitly specifying the folder using `path`, each repository is placed in a subfolder of `s` named after the repository. For example if you check out two repositories named `tools` and `code`, the source code will be checked out into `C:\agent\_work\1\s\tools` and `C:\agent\_work\1\s\code`.

Please note that the checkout path value cannot be set to go up any directory levels above `$(Agent.BuildDirectory)`, so `path\..\anotherpath` will result in a valid checkout path (i.e. `C:\agent\_work\1\anotherpath`), but a value like `..\invalidpath` will not (i.e. `C:\agent\_work\invalidpath`).

You can configure the `path` setting in the Checkout step of your pipeline.

YAML

```yaml
steps:
- checkout: self  # self represents the repo where the initial Pipelines
YAML file was found
  clean: boolean  # whether to fetch clean each time
  fetchDepth: number  # the depth of commits to ask Git to fetch
  lfs: boolean  # whether to download Git-LFS files
  submodules: true | recursive  # set to 'true' for a single level of
submodules or 'recursive' to get submodules of submodules
  path: string  # path to check out source code, relative to the agent's
build directory (e.g. \_work\1)
```

```
  persistCredentials: boolean  # set to 'true' to leave the OAuth token
in the Git config after the initial fetch
```

Submodules

YAML

You can configure the `submodules` setting in the Checkout step of your pipeline if you want to download files from submodules.

YAML

```
steps:
- checkout: self  # self represents the repo where the initial Pipelines
YAML file was found
  clean: boolean  # whether to fetch clean each time
  fetchDepth: number  # the depth of commits to ask Git to fetch
  lfs: boolean  # whether to download Git-LFS files
  submodules: true | recursive  # set to 'true' for a single level of
submodules or 'recursive' to get submodules of submodules
  path: string  # path to check out source code, relative to the agent's
build directory (e.g. \_work\1)
  persistCredentials: boolean  # set to 'true' to leave the OAuth token
in the Git config after the initial fetch
```

The build pipeline will check out your Git submodules as long as they are:

- **Unauthenticated:** A public, unauthenticated repo with no credentials required to clone or fetch.
- **Authenticated:**
  - Contained in the same project as the Azure Repos Git repo specified above. The same credentials that are used by the agent to get the sources from the main repository are also used to get the sources for submodules.
  - Added by using a URL relative to the main repository. For example
    - This one would be checked out: `git submodule add ../../../FabrikamFiberProject/_git/FabrikamFiber FabrikamFiber`

      In this example the submodule refers to a repo (FabrikamFiber) in the same Azure DevOps organization, but in a different project (FabrikamFiberProject). The same credentials that are used by the agent to get the sources from the

main repository are also used to get the sources for submodules. This requires that the job access token has access to the repository in the second project. If you restricted the job access token as explained in the section above, then you won't be able to do this. You can allow the job access token to access the repo in the second project by either (a) explicitly granting access to the project build service account in the second project or (b) using collection-scoped access tokens instead of project-scoped tokens for the entire organization. For more information about these options and their security implications, see Access repositories, artifacts, and other resources.

- This one would not be checked out: `git submodule add https://fabrikam-fiber@dev.azure.com/fabrikam-fiber/FabrikamFiberProject/_git/FabrikamFiber FabrikamFiber`

Alternative to using the Checkout submodules option

In some cases you can't use the **Checkout submodules** option. You might have a scenario where a different set of credentials are needed to access the submodules. This can happen, for example, if your main repository and submodule repositories aren't stored in the same Azure DevOps organization, or if your job access token does not have access to the repository in a different project.

If you can't use the **Checkout submodules** option, then you can instead use a custom script step to fetch submodules. First, get a personal access token (PAT) and prefix it with `pat:`. Next, base64-encode this prefixed string to create a basic auth token. Finally, add this script to your pipeline:

```
git -c http.https://<url of submodule
repository>.extraheader="AUTHORIZATION: Basic <BASE64_ENCODED_STRING>"
submodule update --init --recursive
```

Be sure to replace "<BASE64_ENCODED_STRING>" with your Base64-encoded "pat:token" string.

Use a secret variable in your project or build pipeline to store the basic auth token that you generated. Use that variable to populate the secret in the above Git command.

> **Note**

Q: Why can't I use a Git credential manager on the agent? A: Storing the submodule credentials in a Git credential manager installed on your private build agent is usually not effective as the credential manager may prompt you to re-enter the credentials whenever the submodule is updated. This isn't desirable during automated builds when user interaction isn't possible.

Sync tags

> **Important**
>
> The sync tags feature is supported in Azure Repos Git with Azure DevOps Server 2022.1 and higher.

The checkout step uses the `--tags` option when fetching the contents of a Git repository. This causes the server to fetch all tags as well as all objects that are pointed to by those tags. This increases the time to run the task in a pipeline, particularly if you have a large repository with a number of tags. Furthermore, the checkout step syncs tags even when you enable the shallow fetch option, thereby possibly defeating its purpose. To reduce the amount of data fetched or pulled from a Git repository, Microsoft has added a new option to checkout to control the behavior of syncing tags. This option is available both in classic and YAML pipelines.

Whether to synchronize tags when checking out a repository can be configured in YAML by setting the `fetchTags` property, and in the UI by configuring the **Sync tags** setting.

YAML

You can configure the `fetchTags` setting in the Checkout step of your pipeline.

To configure the setting in YAML, set the `fetchTags` property.

YAML

```yaml
steps:
- checkout: self
  fetchTags: true
```

You can also configure this setting by using the **Sync tags** option in the pipeline settings UI.

1. Edit your YAML pipeline and choose **More actions**, **Triggers**.

2. Choose **YAML**, **Get sources**.

3. Configure the **Sync tags** setting.

> **Note**
>
> If you explicitly set `fetchTags` in your `checkout` step, that setting takes priority over the setting configured in the pipeline settings UI.

Default behavior

- For existing pipelines created before the release of Azure DevOps sprint 209, released in September 2022, the default for syncing tags remains the same as the existing behavior before the **Sync tags** options was added, which is `true`.
- For new pipelines created after Azure DevOps sprint release 209, the default for syncing tags is `false`.

> **Note**
>
> If you explicitly set `fetchTags` in your `checkout` step, that setting takes priority over the setting configured in the pipeline settings UI.

Shallow fetch

You may want to limit how far back in history to download. Effectively this results in `git fetch --depth=n`. If your repository is large, this option might make your build pipeline more efficient. Your repository might be large if it has been in use for a long time and has sizeable history. It also might be large if you added and later deleted large files.

> **Important**
>
> New pipelines created after the **September 2022 Azure DevOps sprint 209 update** have **Shallow fetch** enabled by default and configured with a depth of 1. Previously the default was not to shallow fetch. To check your pipeline, view the **Shallow fetch** setting in the pipeline settings UI as described in the following section.

YAML

You can configure the `fetchDepth` setting in the Checkout step of your pipeline.

YAML

```yaml
steps:
- checkout: self  # self represents the repo where the initial Pipelines
YAML file was found
  clean: boolean  # whether to fetch clean each time
  fetchDepth: number  # the depth of commits to ask Git to fetch
  lfs: boolean  # whether to download Git-LFS files
  submodules: true | recursive  # set to 'true' for a single level of
submodules or 'recursive' to get submodules of submodules
  path: string  # path to check out source code, relative to the agent's
```

```
build directory (e.g. \_work\1)
  persistCredentials: boolean  # set to 'true' to leave the OAuth token
in the Git config after the initial fetch
```

You can also configure fetch depth by setting the **Shallow depth** option in the pipeline settings UI.

1. Edit your YAML pipeline and choose **More actions**, **Triggers**.

2. Choose **YAML**, **Get sources**.

3. Configure the **Shallow fetch** setting. Uncheck **Shallow fetch** to disable shallow fetch, or check the box and enter a **Depth** to enable shallow fetch.

> **Note**
>
> If you explicitly set `fetchDepth` in your `checkout` step, that setting takes priority over the setting configured in the pipeline settings UI. Setting `fetchDepth: 0` fetches all history and overrides the **Shallow fetch** setting.

In these cases this option can help you conserve network and storage resources. It might also save time. The reason it doesn't always save time is because in some situations the server might need to spend time calculating the commits to download for the depth you specify.

> **Note**
>
> When the pipeline is started, the branch to build is resolved to a commit ID. Then, the agent fetches the branch and checks out the desired commit. There is a small window between when a branch is resolved to a commit ID and when the agent performs the checkout. If the branch updates rapidly and you set a very small value for shallow fetch, the commit may not exist when the agent attempts to check it out. If that happens, increase the shallow fetch depth setting.

Don't sync sources

You may want to skip fetching new commits. This option can be useful in cases when you want to:

- Git init, config, and fetch using your own custom options.

- Use a build pipeline to just run automation (for example some scripts) that do not depend on code in version control.

YAML

You can configure the **Don't sync sources** setting in the Checkout step of your pipeline, by setting `checkout: none`.

YAML

```yaml
steps:
- checkout: none  # Don't sync sources
```

> **Note**
>
> When you use this option, the agent also skips running Git commands that clean the repo.

Clean build

You can perform different forms of cleaning the working directory of your self-hosted agent before a build runs.

In general, for faster performance of your self-hosted agents, don't clean the repo. In this case, to get the best performance, make sure you're also building incrementally by disabling any **Clean** option of the task or tool you're using to build.

If you do need to clean the repo (for example to avoid problems caused by residual files from a previous build), your options are below.

> **Note**
>
> Cleaning is not effective if you're using a **Microsoft-hosted agent** because you'll get a new agent every time.

YAML

You can configure the `clean` setting in the Checkout step of your pipeline.

YAML

```yaml
steps:
- checkout: self  # self represents the repo where the initial Pipelines
YAML file was found
  clean: boolean  # whether to fetch clean each time
  fetchDepth: number  # the depth of commits to ask Git to fetch
  lfs: boolean  # whether to download Git-LFS files
  submodules: true | recursive  # set to 'true' for a single level of
submodules or 'recursive' to get submodules of submodules
  path: string  # path to check out source code, relative to the agent's
build directory (e.g. \_work\1)
  persistCredentials: boolean  # set to 'true' to leave the OAuth token
in the Git config after the initial fetch
```

When `clean` is set to `true` the build pipeline performs an undo of any changes in `$(Build.SourcesDirectory)`. More specifically, the following Git commands are executed prior to fetching the source.

```
git clean -ffdx
git reset --hard HEAD
```

For more options, you can configure the `workspace` setting of a Job.

YAML

```yaml
jobs:
- job: string  # name of the job, A-Z, a-z, 0-9, and underscore
  ...
  workspace:
    clean: outputs | resources | all # what to clean up before the job
runs
```

This gives the following clean options.

- **outputs**: Same operation as the clean setting described in the previous checkout task, plus: Deletes and recreates `$(Build.BinariesDirectory)`. Note that the `$(Build.ArtifactStagingDirectory)` and `$(Common.TestResultsDirectory)` are always deleted and recreated prior to every build regardless of any of these settings.
- **resources**: Deletes and recreates `$(Build.SourcesDirectory)`. This results in initializing a new, local Git repository for every build.

- **all**: Deletes and recreates `$(Agent.BuildDirectory)`. This results in initializing a new, local Git repository for every build.

Label sources

You may want to label your source code files to enable your team to easily identify which version of each file is included in the completed build. You also have the option to specify whether the source code should be labeled for all builds or only for successful builds.

YAML

You can't currently configure this setting in YAML but you can in the classic editor. When editing a YAML pipeline, you can access the classic editor by choosing either **Triggers** from the YAML editor menu.

From the classic editor, choose **YAML**, choose the **Get sources** task, and then configure the desired properties there.

In the **Tag format** you can use user-defined and predefined variables that have a scope of "All." For example:

```
$(Build.DefinitionName)_$(Build.DefinitionVersion)_$(Build.BuildId)_$(Build.BuildNumber)_$(My.Variable)
```

The first four variables are predefined. `My.Variable` can be defined by you on the variables tab.

The build pipeline labels your sources with a Git tag.

Some build variables might yield a value that is not a valid label. For example, variables such as `$(Build.RequestedFor)` and `$(Build.DefinitionName)` can contain white space. If the value contains white space, the tag is not created.

After the sources are tagged by your build pipeline, an artifact with the Git ref `refs/tags/{tag}` is automatically added to the completed build. This gives your team additional traceability and a more user-friendly way to navigate from the build to the code that was built. The tag is considered a build artifact since it is produced by the

build. When the build is deleted either manually or through a retention policy, the tag is also deleted.

Pre-defined variables

When you build a GitHub repository, most of the predefined variables are available to your jobs. However, since Azure Pipelines doesn't recognize the identity of a user making an update in GitHub, the following variables are set to system identity instead of user's identity:

- `Build.RequestedFor`
- `Build.RequestedForId`
- `Build.RequestedForEmail`

Status updates

There are two types of statuses that Azure Pipelines posts back to GitHub - basic statuses and GitHub Check Runs. GitHub Checks functionality is only available with GitHub Apps.

Pipeline statuses show up in various places in the GitHub UI.

- For PRs, they're displayed on the PR conversations tab.
- For individual commits, they're displayed when hovering over the status mark after the commit time on the repo's commits tab.

PAT or OAuth GitHub connections

For pipelines using PAT or OAuth GitHub connections, statuses are posted back to the commit/PR that triggered the run. The GitHub status API is used to post such updates. These statuses contain limited information: pipeline status (failed, success), URL to link back to the build pipeline, and a brief description of the status.

Statuses for PAT or OAuth GitHub connections are only sent at the run level. In other words, you can have a single status updated for an entire run. If you have multiple jobs in a run, you can't post a separate status for each job. However, multiple pipelines can post separate statuses to the same commit.

GitHub Checks

For pipelines set up using the Azure Pipelines GitHub app, the status is posted back in the form of GitHub Checks. GitHub Checks allow for sending detailed information about the pipeline status and test, code coverage, and errors. The GitHub Checks API can be found here.

For every pipeline using the GitHub App, Checks are posted back for the overall run and each job in that run.

GitHub allows three options when one or more Check Runs fail for a PR/commit. You can choose to "rerun" the individual Check, rerun all the failing Checks on that PR/commit, or rerun all the Checks, whether they succeeded initially or not.

Clicking on the "Rerun" link next to the Check Run name will result in Azure Pipelines retrying the run that generated the Check Run. The resultant run will have the same run number and will use the same version of the source code, configuration, and YAML file as the initial build. Only those jobs that failed in the initial run and any dependent downstream jobs will be run again. Clicking on the "Rerun all failing checks" link will have the same effect. This is the same behavior as clicking "Retry run" in the Azure Pipelines UI. Clicking on "Rerun all checks" will result in a new run, with a new run number and will pick up changes in the configuration or YAML file.

Limitations

- For best performance, we recommend a maximum of 50 pipelines in a single repository. For acceptable performance, we recommend a maximum of 100 pipelines in a single repository. The time required to process a push to a repository increases with the number of pipelines in that repository. Whenever there's push to a repository, Azure Pipelines needs to load all YAML pipelines in that repository to

figure out if any of them need to run, and each loaded pipeline incurs a performance penalty.
- Azure Pipelines loads a maximum of 2000 branches from a repository into dropdown lists in the Azure Devops Portal, for example into the **Default branch for manual and scheduled builds** setting, or when choosing a branch when running a pipeline manually. If you don't see your desired branch in the list, type the desired branch name manually.

FAQ

Problems related to GitHub integration fall into the following categories:

- **Connection types**: I'm not sure what connection type I'm using to connect my pipeline to GitHub.
- **Failing triggers**: My pipeline isn't being triggered when I push an update to the repo.
- **Failing checkout**: My pipeline is being triggered, but it fails in the checkout step.
- **Wrong version**: My pipeline runs, but it's using an unexpected version of the source/YAML.
- **Missing status updates**: My GitHub PRs are blocked because Azure Pipelines didn't report a status update.

Connection types

To troubleshoot triggers, how do I know the type of GitHub connection I'm using for my pipeline?

Troubleshooting problems with triggers very much depends on the type of GitHub connection you use in your pipeline. There are two ways to determine the type of connection - from GitHub and from Azure Pipelines.

- From GitHub: If a repo is set up to use the GitHub app, then the statuses on PRs and commits will be Check Runs. If the repo has Azure Pipelines set up with OAuth or PAT connections, the statuses will be the "old" style of statuses. A quick way to determine if the statuses are Check Runs or simple statuses is to look at the "conversation" tab on a GitHub PR.
  - If the "Details" link redirects to the Checks tab, it's a Check Run and the repo is using the app.
  - If the "Details" link redirects to the Azure DevOps pipeline, then the status is an "old style" status and the repo isn't using the app.

- From Azure Pipelines: You can also determine the type of connection by inspecting the pipeline in Azure Pipelines UI. Open the editor for the pipeline. Select **Triggers** to open the classic editor for the pipeline. Then, select **YAML** tab and then the **Get sources** step. You'll notice a banner **Authorized using connection:** indicating the service connection that was used to integrate the pipeline with GitHub. The name of the service connection is a hyperlink. Select it to navigate to the service connection properties. The properties of the service connection will indicate the type of connection being used:
  - **Azure Pipelines app** indicates GitHub app connection
  - **oauth** indicates OAuth connection
  - **personalaccesstoken** indicates PAT authentication

How do I switch my pipeline to use GitHub app instead of OAuth?

Using a GitHub app instead of OAuth or PAT connection is the recommended integration between GitHub and Azure Pipelines. To switch to GitHub app, follow these steps:

1. Navigate here and install the app in the GitHub organization of your repository.
2. During installation, you'll be redirected to Azure DevOps to choose an Azure DevOps organization and project. Choose the organization and project that contain the classic build pipeline you want to use the app for. This choice associates the GitHub App installation with your Azure DevOps organization. If you choose incorrectly, you can visit this page to uninstall the GitHub app from your GitHub org and start over.
3. In the next page that appears, you don't need to proceed creating a new pipeline.
4. Edit your pipeline by visiting the Pipelines page (e.g., https://dev.azure.com/YOUR_ORG_NAME/YOUR_PROJECT_NAME/_build), selecting your pipeline, and clicking Edit.
5. If this is a YAML pipeline, select the **Triggers** menu to open the classic editor.
6. Select the "Get sources" step in the pipeline.
7. On the green bar with text "Authorized using connection", select "Change" and select the GitHub App connection with the same name as the GitHub organization in which you installed the app.
8. On the toolbar, select "Save and queue" and then "Save and queue". Select the link to the pipeline run that was queued to make sure it succeeds.
9. Create (or close and reopen) a pull request in your GitHub repository to verify that a build is successfully queued in its "Checks" section.

Why isn't a GitHub repository displayed for me to choose in Azure Pipelines?

Depending on the authentication type and ownership of the repository, specific permissions are required.

- If you're using the GitHub App, see GitHub App authentication.
- If you're using OAuth, see OAuth authentication.
- If you're using PATs, see Personal access token (PAT) authentication.

When I select a repository during pipeline creation, I get an error "The repository {repo-name} is in use with the Azure Pipelines GitHub App in another Azure DevOps organization."

This means that your repository is already associated with a pipeline in a different organization. CI and PR events from this repository won't work as they'll be delivered to the other organization. Here are the steps you should take to remove the mapping to the other organization before proceeding to create a pipeline.

1. Open a pull request in your GitHub repository, and make the comment `/azp where`. This reports back the Azure DevOps organization that the repository is mapped to.

2. To change the mapping, uninstall the app from the GitHub organization, and reinstall it. As you reinstall it, make sure to select the correct organization when you're redirected to Azure DevOps.

Failing triggers

I just created a new YAML pipeline with CI/PR triggers, but the pipeline isn't being triggered.

Follow each of these steps to troubleshoot your failing triggers:

- Are your YAML CI or PR triggers overridden by pipeline settings in the UI? While editing your pipeline, choose **...** and then **Triggers**.

Check the **Override the YAML trigger from here** setting for the types of trigger (**Continuous integration** or **Pull request validation**) available for your repo.

- Are you using the GitHub app connection to connect the pipeline to GitHub? See Connection types to determine the type of connection you have. If you're using a GitHub app connection, follow these steps:
  - Is the mapping set up properly between GitHub and Azure DevOps? Open a pull request in your GitHub repository, and make the comment `/azp where`. This reports back the Azure DevOps organization that the repository is mapped to.
    - If no organizations are set up to build this repository using the app, go to `https://github.com/<org_name>/<repo_name>/settings/installations` and complete the configuration of the app.
    - If a different Azure DevOps organization is reported, then someone has already established a pipeline for this repo in a different organization. We currently have the limitation that we can only map a GitHub repo to a single DevOps org. Only the pipelines in the first Azure DevOps org can be automatically triggered. To change the mapping, uninstall the app from the

GitHub organization, and reinstall it. As you reinstall it, make sure to select the correct organization when you're redirected to Azure DevOps.

- Are you using OAuth or PAT to connect the pipeline to GitHub? See Connection types to determine the type of connection you have. If you're using a GitHub connection, follow these steps:

  1. OAuth and PAT connections rely on webhooks to communicate updates to Azure Pipelines. In GitHub, navigate to the settings for your repository, then to Webhooks. Verify that the webhooks exist. Usually you should see three webhooks - push, pull_request, and issue_comment. If you don't, then you must re-create the service connection and update the pipeline to use the new service connection.

  2. Select each of the webhooks in GitHub and verify that the payload that corresponds to the user's commit exists and was sent successfully to Azure DevOps. You may see an error here if the event couldn't be communicated to Azure DevOps.

- The traffic from Azure DevOps could be throttled by GitHub. When Azure Pipelines receives a notification from GitHub, it tries to contact GitHub and fetch more information about the repo and YAML file. If you have a repo with a large number of updates and pull requests, this call may fail due to such throttling. In this case, see if you can reduce the frequency of builds by using batching or stricter path/branch filters.

- Is your pipeline paused or disabled? Open the editor for the pipeline, and then select **Settings** to check. If your pipeline is paused or disabled, then triggers do not work.

- Have you updated the YAML file in the correct branch? If you push an update to a branch, then the YAML file in that same branch governs the CI behavior. If you push an update to a source branch, then the YAML file resulting from merging the source branch with the target branch governs the PR behavior. Make sure that the YAML file in the correct branch has the necessary CI or PR configuration.

- Have you configured the trigger correctly? When you define a YAML trigger, you can specify both include and exclude clauses for branches, tags, and paths. Ensure that the include clause matches the details of your commit and that the exclude clause doesn't exclude them. Check the syntax for the triggers and make sure that it is accurate.

- Have you used variables in defining the trigger or the paths? That is not supported.

- Did you use templates for your YAML file? If so, make sure that your triggers are defined in the main YAML file. Triggers defined inside template files are not supported.

- Have you excluded the branches or paths to which you pushed your changes? Test by pushing a change to an included path in an included branch. Note that paths in triggers are case-sensitive. Make sure that you use the same case as those of real folders when specifying the paths in triggers.

- Did you just push a new branch? If so, the new branch may not start a new run. See the section "Behavior of triggers when new branches are created".

My CI or PR triggers have been working fine. But, they stopped working now.

First, go through the troubleshooting steps in the previous question, then follow these additional steps:

- Do you have merge conflicts in your PR? For a PR that didn't trigger a pipeline, open it and check whether it has a merge conflict. Resolve the merge conflict.

- Are you experiencing a delay in the processing of push or PR events? You can usually verify a delay by seeing if the issue is specific to a single pipeline or is common to all pipelines or repos in your project. If a push or a PR update to any of the repos exhibits this symptom, we might be experiencing delays in processing the update events. Here are some reasons why a delay may be happening:
  - We are experiencing a service outage on our status page. If the status page shows an issue, then our team must have already started working on it. Check the page frequently for updates on the issue.
  - Your repository contains too many YAML pipelines. For best performance, we recommend a maximum of 50 pipelines in a single repository. For acceptable performance, we recommend a maximum of 100 pipelines in a single repository. The more pipelines there are, the slower the processing of a push to that repository. Whenever there is push to a repository, Azure Pipelines needs to load all YAML pipelines in that repository, to figure out if any of them need to run, and each new pipeline incurs a performance penalty.

I do not want users to override the list of branches for triggers when they update the YAML file. How can I do this?

Users with permissions to contribute code can update the YAML file and include/exclude additional branches. As a result, users can include their own feature or user branch in

their YAML file and push that update to a feature or user branch. This may cause the pipeline to be triggered for all updates to that branch. If you want to prevent this behavior, then you can:

1. Edit the pipeline in the Azure Pipelines UI.
2. Navigate to the **Triggers** menu.
3. Select **Override the YAML continuous Integration trigger from here**.
4. Specify the branches to include or exclude for the trigger.

When you follow these steps, any CI triggers specified in the YAML file are ignored.

Failing checkout

I see the following error in the log file during checkout step. How do I fix it?

log

```
remote: Repository not found.
fatal: repository <repo> not found
```

This could be caused by an outage of GitHub. Try to access the repository in GitHub and make sure that you're able to.

Wrong version

A wrong version of the YAML file is being used in the pipeline. Why is that?

- For CI triggers, the YAML file that is in the branch you are pushing is evaluated to see if a CI build should be run.
- For PR triggers, the YAML file resulting from merging the source and target branches of the PR is evaluated to see if a PR build should be run.

Missing status updates

My PR in GitHub is blocked since Azure Pipelines didn't update the status.

This could be a transient error that resulted in Azure DevOps not being able to communicate with GitHub. Retry the check-in GitHub if you use the GitHub app. Or, make a trivial update to the PR to see if the problem can be resolved.

Related articles

- Scheduled triggers
- Pipeline completion triggers

Build GitHub Enterprise Server repositories

Article • 09/28/2023

Azure DevOps Services

You can integrate your on-premises GitHub Enterprise Server with Azure Pipelines. Your on-premises server may be exposed to the Internet or it may not be.

If your GitHub Enterprise Server is reachable from the servers that run Azure Pipelines service, then:

- you can set up classic build and YAML pipelines
- you can configure CI, PR, and scheduled triggers

If your GitHub Enterprise Server is not reachable from the servers that run Azure Pipelines service, then:

- you can only set up classic build pipelines
- you can only start manual or scheduled builds
- you cannot set up YAML pipelines
- you cannot configure CI or PR triggers for your classic build pipelines

If your on-premises server is reachable from Microsoft-hosted agents, then you can use them to run your pipelines. Otherwise, you must set up self-hosted agents that can access your on-premises server and fetch the code.

Reachable from Azure Pipelines

The first thing to check is whether your GitHub Enterprise Server is reachable from Azure Pipelines service.

1. In your Azure DevOps UI, navigate to your project settings, and select **Service Connections** under **Pipelines**.
2. Select **New service connection** and choose **GitHub Enterprise Server** as the connection type.
3. Enter the required information to create a connection to your GitHub Enterprise Server.
4. Select **Verify** in the service connection panel.

If the verification passes, then the servers that run Azure Pipelines service are able to reach your on-premises GitHub Enterprise Server. You can proceed and set up the connection. Then, you can use this service connection when creating a classic build or YAML pipeline. You can also configure CI and PR triggers for the pipeline. A majority of features in Azure Pipelines that work with GitHub also work with GitHub Enterprise Server. Review the documentation for GitHub to understand these features. Here are some differences:

- The integration between GitHub and Azure Pipelines is made easier through an Azure Pipelines app in GitHub marketplace. This app allows you to set up an integration without having to rely on a particular user's OAuth token. We do not have a similar app that works with GitHub Enterprise Server. So, you must use a PAT, username and password, or OAuth to set up the connection between Azure Pipelines and GitHub Enterprise server.
- Azure Pipelines supports a number of GitHub security features to validate contributions from external forks. For instance, secrets stored in a pipeline are not made available to a running job. These protections are not available when working with GitHub Enterprise server.
- Comment triggers are not available with GitHub Enterprise server. You cannot use comments in a GitHub Enterprise server repo pull request to trigger a pipeline.
- GitHub Checks are not available in GitHub Enterprise server. All status updates are through basic statuses.

Not reachable from Azure Pipelines

When the verification of a GitHub Enterprise Server connection as explained in the above section fails, then Azure Pipelines cannot communicate with your server. This is likely caused by how your enterprise network is set up. For instance, a firewall in your network may prevent external traffic from reaching your servers. You have two options in this case:

- Work with your IT department to open a network path between Azure Pipelines and GitHub Enterprise Server. For example, you can add exceptions to your firewall rules to allow traffic from Azure Pipelines to flow through. See the section on Azure DevOps IPs to see which IP addresses you need to allow. Furthermore, you need to have a public DNS entry for the GitHub Enterprise Server so that Azure Pipelines can resolve the FQDN of your server to an IP address. With all of these changes, attempt to create and verify a GitHub Enterprise Server connection in Azure Pipelines.

- Instead of a using a GitHub Enterprise Server connection, you can use a **Other Git** connection. Make sure to uncheck the option to **Attempt accessing this Git server from Azure Pipelines**. With this connection type, you can only configure a classic build pipeline. CI and PR triggers will not work in this configuration. You can only start manual or scheduled pipeline runs.

Reachable from Microsoft-hosted agents

Another decision you possibly have to make is whether to use Microsoft-hosted agents or self-hosted agents to run your pipelines. This often comes down to whether Microsoft-hosted agents can reach your server. To check whether they can, create a simple pipeline to use Microsoft-hosted agents and make sure to add a step to check out source code from your server. If this passes, then you can continue using Microsoft-hosted agents.

Not reachable from Microsoft-hosted agents

If the simple test pipeline mentioned in the above section fails with the error `TF401019: The Git repository with name or identifier <your repo name> does not exist or you do not have permissions for the operation you are attempting`, then the GitHub Enterprise Server is not reachable from Microsoft-hosted agents. This is again probably caused by a firewall blocking traffic from these servers. You have two options in this case:

- Work with your IT department to open a network path between Microsoft-hosted agents and GitHub Enterprise Server. See the section on networking in Microsoft-hosted agents.

- Switch to using self-hosted agents or scale-set agents. These agents can be set up within your network and hence will have access to the GitHub Enterprise Server. These agents only require outbound connections to Azure Pipelines. There is no need to open a firewall for inbound connections. Make sure that the name of the server you specified when creating the GitHub Enterprise Server connection is resolvable from the self-hosted agents.

Azure DevOps IP addresses

Azure Pipelines sends requests to GitHub Enterprise Server to:

- Query for a list of repositories during pipeline creation (classic and YAML pipelines)

- Look for existing YAML files during pipeline creation (YAML pipelines)
- Check-in YAML files (YAML pipelines)
- Register a webhook during pipeline creation (classic and YAML pipelines)
- Present an editor for YAML files (YAML pipelines)
- Resolve templates and expand YAML files prior to execution (YAML pipelines)
- Check if there are any changes since the last scheduled run (classic and YAML pipelines)
- Fetch details about latest commit and display that in the user interface (classic and YAML pipelines)

You can observe that YAML pipelines fundamentally require communication between Azure Pipelines and GitHub Enterprise Server. Hence, it is not possible to set up a YAML pipeline if the GitHub Enterprise Server is not visible to Azure Pipelines.

When you use **Other Git** connection to set up a classic pipeline, disable communication between Azure Pipelines service and GitHub Enterprise Server, and use self-hosted agents to build code, you will get a degraded experience:

- You will have to type in the name of the repository manually during pipeline creation
- You cannot use CI or PR triggers as Azure Pipelines cannot register a webhook in GitHub Enterprise Server
- You cannot use scheduled triggers with the option to build only when there are changes
- You cannot view information about the latest commit in the user interface

If you want to set up YAML pipelines or if you want to enhance the experience with classic pipelines, it is important that you enable communication from Azure Pipelines to GitHub Enterprise Server.

To allow traffic from Azure DevOps to reach your GitHub Enterprise Server, add the IP addresses or service tags specified in Inbound connections to your firewall's allowlist. If you use ExpressRoute, make sure to also include ExpressRoute IP ranges to your firewall's allowlist.

Limitations

- For best performance, we recommend a maximum of 50 pipelines in a single repository. For acceptable performance, we recommend a maximum of 100 pipelines in a single repository. The more pipelines there are, the slower the processing of a push to that repository. Whenever there is push to a repository,

Azure Pipelines needs to load all YAML pipelines in that repository to figure out if any of them need to run, and each loaded pipeline incurs a performance penalty.
- Azure Pipelines loads a maximum of 2000 branches from a repository.

FAQ

Problems related to GitHub Enterprise integration fall into the following categories:

- **Failing triggers**: My pipeline is not being triggered when I push an update to the repo.
- **Failing checkout**: My pipeline is being triggered, but it fails in the checkout step.
- **Wrong version**: My pipeline runs, but it is using an unexpected version of the source/YAML.

Failing triggers

I just created a new YAML pipeline with CI/PR triggers, but the pipeline is not being triggered.

Follow each of these steps to troubleshoot your failing triggers:

- Are your YAML CI or PR triggers being overridden by pipeline settings in the UI? While editing your pipeline, choose **...** and then **Triggers**.

Check the **Override the YAML trigger from here** setting for the types of trigger (**Continuous integration** or **Pull request validation**) available for your repo.

- Webhooks are used to communicate updates from GitHub Enterprise to Azure Pipelines. In GitHub Enterprise, navigate to the settings for your repository, then to Webhooks. Verify that the webhooks exist. Usually you should see two webhooks - push, pull_request. If you don't, then you must re-create the service connection and update the pipeline to use the new service connection.

- Select each of the webhooks in GitHub Enterprise and verify that the payload that corresponds to the user's commit exists and was sent successfully to Azure DevOps. You may see an error here if the event could not be communicated to Azure DevOps.

- When Azure Pipelines receives a notification from GitHub, it tries to contact GitHub and fetch more information about the repo and YAML file. If the GitHub Enterprise Server is behind a firewall, this traffic may not reach your server. See Azure DevOps IP Addresses and verify that you have granted exceptions to all the required IP addresses. These IP addresses may have changed since you have originally set up the exception rules.

- Is your pipeline paused or disabled? Open the editor for the pipeline, and then select **Settings** to check. If your pipeline is paused or disabled, then triggers do not work.

- Have you updated the YAML file in the correct branch? If you push an update to a branch, then the YAML file in that same branch governs the CI behavior. If you push an update to a source branch, then the YAML file resulting from merging the source branch with the target branch governs the PR behavior. Make sure that the YAML file in the correct branch has the necessary CI or PR configuration.

- Have you configured the trigger correctly? When you define a YAML trigger, you can specify both include and exclude clauses for branches, tags, and paths. Ensure that the include clause matches the details of your commit and that the exclude clause doesn't exclude them. Check the syntax for the triggers and make sure that it is accurate.

- Have you used variables in defining the trigger or the paths? That is not supported.
- Did you use templates for your YAML file? If so, make sure that your triggers are defined in the main YAML file. Triggers defined inside template files are not supported.
- Have you excluded the branches or paths to which you pushed your changes? Test by pushing a change to an included path in an included branch. Note that paths in triggers are case-sensitive. Make sure that you use the same case as those of real folders when specifying the paths in triggers.
- Did you just push a new branch? If so, the new branch may not start a new run. See the section "Behavior of triggers when new branches are created".

My CI or PR triggers have been working fine. But, they stopped working now.

First, go through the troubleshooting steps in the previous question, then follow these additional steps:

- Do you have merge conflicts in your PR? For a PR that did not trigger a pipeline, open it and check whether it has a merge conflict. Resolve the merge conflict.
- Are you experiencing a delay in the processing of push or PR events? You can usually verify a delay by seeing if the issue is specific to a single pipeline or is common to all pipelines or repos in your project. If a push or a PR update to any of the repos exhibits this symptom, we might be experiencing delays in processing the update events. Here are some reasons why a delay may be happening:
  - We are experiencing a service outage on our status page. If the status page shows an issue, then our team must have already started working on it. Check the page frequently for updates on the issue.
  - Your repository contains too many YAML pipelines. For best performance, we recommend a maximum of 50 pipelines in a single repository. For acceptable performance, we recommend a maximum of 100 pipelines in a single repository. The more pipelines there are, the slower the processing of a push to that repository. Whenever there is push to a repository, Azure Pipelines needs to load all YAML pipelines in that repository, to figure out if any of them need to run, and each new pipeline incurs a performance penalty.
  - Your GitHub Enterprise Server instance may be underprovisioned, slowing down processing requests from Azure Pipelines. Read more about hardware considerations for GitHub Enterprise Server.

Failing checkout

Do you use Microsoft-hosted agents? If so, these agents may not be able to reach your GitHub Enterprise Server. See Not reachable from Microsoft-hosted agents for more information.

Wrong version

A wrong version of the YAML file is being used in the pipeline. Why is that?

- For CI triggers, the YAML file that is in the branch you are pushing is evaluated to see if a CI build should be run.
- For PR triggers, the YAML file resulting from merging the source and target branches of the PR is evaluated to see if a PR build should be run.

Pipeline options for Git repositories

Article • 05/16/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

While editing a pipeline that uses a Git repo—in an Azure DevOps project, GitHub, GitHub Enterprise Server, Bitbucket Cloud, or another Git repo—you have the following options.

Feature	Azure Pipelines	Azure DevOps Server 2019 and higher	TFS 2018
Branch	Yes	Yes	Yes
Clean	Yes	Yes	Yes
Tag or label sources	Project; Classic only	Team project	Team project
Report build status	Yes	Yes	Yes
Check out submodules	Yes	Yes	Yes
Check out files from LFS	Yes	Yes	Yes
Clone a second repo	Yes	Yes	Yes
Don't sync sources	Yes	Yes	Yes
Shallow fetch	Yes	Yes	Yes

> **Note**
>
> Click **Advanced settings** in the **Get Sources** task to see some of the above options.

Branch

This is the branch that you want to be the default when you manually queue this build. If you set a scheduled trigger for the build, this is the branch from which your build will get the latest sources. The default branch has no bearing when the build is triggered

through continuous integration (CI). Usually you'll set this to be the same as the default branch of the repository (for example, "master").

Clean the local repo on the agent

You can perform different forms of cleaning the working directory of your self-hosted agent before a build runs.

In general, for faster performance of your self-hosted agents, don't clean the repo. In this case, to get the best performance, make sure you're also building incrementally by disabling any **Clean** option of the task or tool you're using to build.

If you do need to clean the repo (for example to avoid problems caused by residual files from a previous build), your options are below.

> ⓘ **Note**
>
> Cleaning is not effective if you're using a **Microsoft-hosted agent** because you'll get a new agent every time. When using self-hosted agents, depending on how your agents pools are configured, you may get a new agent for subsequent pipeline runs (or stages or jobs in the same pipeline), so **not** cleaning is not a guarantee that subsequent runs, jobs, or stages will be able to access outputs from previous runs, jobs, or stages.

YAML

Azure Pipelines, Azure DevOps Server 2019 and newer

There are several different clean options available for YAML pipelines.

- The `checkout` step has a `clean` option. When set to `true`, the pipeline runs `execute git clean -ffdx && git reset --hard HEAD` before fetching the repo. For more information, see Checkout.
- The `workspace` setting for `job` has multiple clean options (outputs, resources, all). For more information, see Workspace.
- The pipeline settings UI has a **Clean** setting, that when set to true is equivalent of specifying `clean: true` for every `checkout` step in your pipeline. To configure the **Clean** setting:

    1. Edit your pipeline, choose **...**, and select **Triggers**.

2. Select **YAML**, **Get sources**, and configure your desired **Clean** setting. The default is **true**.

To override clean settings when manually running a pipeline, you can use runtime parameters. In the following example, a runtime parameter is used to configure the checkout clean setting.

yml

```yml
parameters:
- name: clean
  displayName: Checkout clean
  type: boolean
  default: true
  values:
  - false
  - true

trigger:
- main

pool: FabrikamPool
#  vmImage: 'ubuntu-latest'
```

```yaml
steps:
- checkout: self
  clean: ${{ parameters.clean }}
```

By default, `clean` is set to `true` but can be overridden when manually running the pipeline by unchecking the **Checkout clean** checkbox that is added for the runtime parameter.

Label sources

You may want to label your source code files to enable your team to easily identify which version of each file is included in the completed build. You also have the option to specify whether the source code should be labeled for all builds or only for successful builds.

> **Note**
>
> You can only use this feature when the source repository in your build is a GitHub repository, or a Git or TFVC repository from your project.

In the **Label format** you can use user-defined and predefined variables that have a scope of "All." For example:

```
$(Build.DefinitionName)_$(Build.DefinitionVersion)_$(Build.BuildId)_$(Build.BuildNumber)_$(My.Variable)
```

The first four variables are predefined. `My.Variable` can be defined by you on the variables tab.

The build pipeline labels your sources with a Git tag.

Some build variables might yield a value that is not a valid label. For example, variables such as `$(Build.RequestedFor)` and `$(Build.DefinitionName)` can contain white space. If the value contains white space, the tag is not created.

After the sources are tagged by your build pipeline, an artifact with the Git ref `refs/tags/{tag}` is automatically added to the completed build. This gives your team additional traceability and a more user-friendly way to navigate from the build to the code that was built. The tag is considered a build artifact since it is produced by the

build. When the build is deleted either manually or through a retention policy, the tag is also deleted.

Report build status (Azure Pipelines, TFS 2018 and newer)

You've got the option to give your team a view of the build status from your remote source repository.

If your sources are in an Azure Repos Git repository in your project, then this option displays a badge on the **Code** page to indicate whether the build is passing or failing. The build status is displayed in the following tabs:

- **Files**: Indicates the status of the latest build for the selected branch.
- **Commits**: Indicates the build status of each commit (this requires the continuous integration (CI) trigger to be enabled for your builds).
- **Branches**: Indicates the status of the latest build for each branch.

If you use multiple build pipelines for the same repository in your project, then you may choose to enable this option for one or more of the pipelines. In the case when this option is enabled on multiple pipelines, the badge on the **Code** page indicates the status of the latest build across all the pipelines. Your team members can click the build status badge to view the latest build status for each one of the build pipelines.

GitHub

If your sources are in GitHub, then this option publishes the status of your build to GitHub using GitHub Checks or Status APIs. If your build is triggered from a GitHub pull request, then you can view the status on the GitHub pull requests page. This also allows you to set status policies within GitHub and automate merges. If your build is triggered by continuous integration (CI), then you can view the build status on the commit or branch in GitHub.

Other types of Git remote repositories

If your source is in any other type of remote repository, then you cannot use Azure Pipelines or TFS to automatically publish the build status to that repository. However, you can use a build badge as a way to integrate and show build status within your version control experiences.

Checkout path

If you are checking out a single repository, by default, your source code will be checked out into a directory called `s`. For YAML pipelines, you can change this by specifying `checkout` with a `path`. The specified path is relative to `$(Agent.BuildDirectory)`. For example: if the checkout path value is `mycustompath` and `$(Agent.BuildDirectory)` is `C:\agent\_work\1`, then the source code will be checked out into `C:\agent\_work\1\mycustompath`.

If you are using multiple `checkout` steps and checking out multiple repositories, and not explicitly specifying the folder using `path`, each repository is placed in a subfolder of `s` named after the repository. For example if you check out two repositories named `tools` and `code`, the source code will be checked out into `C:\agent\_work\1\s\tools` and `C:\agent\_work\1\s\code`.

Please note that the checkout path value cannot be set to go up any directory levels above `$(Agent.BuildDirectory)`, so `path\..\anotherpath` will result in a valid checkout path (i.e. `C:\agent\_work\1\anotherpath`), but a value like `..\invalidpath` will not (i.e. `C:\agent\_work\invalidpath`).

If you are using multiple `checkout` steps and checking out multiple repositories, and want to explicitly specify the folder using `path`, consider avoiding setting path which is subfolder of another checkout step's path (i.e. `C:\agent\_work\1\s\repo1` and `C:\agent\_work\1\s\repo1\repo2`), otherwise, the subfolder of the checkout step will be cleared by another repo's cleaning. Please note that this case is valid if the clean option is true for `repo1`)

> **Note**
>
> The checkout path can only be specified for YAML pipelines. For more information, see **Checkout** in the **YAML schema**.

Checkout submodules

Select if you want to download files from submodules. You can either choose to get the immediate submodules or all submodules nested to any depth of recursion. If you want to use LFS with submodules, be sure to see the note about using LFS with submodules.

> **Note**
>
> For more information about the YAML syntax for checking out submodules, see **Checkout in the YAML schema**.

The build pipeline will check out your Git submodules as long as they are:

- **Unauthenticated:** A public, unauthenticated repo with no credentials required to clone or fetch.
- **Authenticated:**
  - Contained in the same project, GitHub organization, or Bitbucket Cloud account as the Git repo specified above.
  - Added by using a URL relative to the main repository. For example, this one would be checked out: `git submodule add /../../submodule.git mymodule` This one would not be checked out: `git submodule add https://dev.azure.com/fabrikamfiber/_git/ConsoleApp mymodule`

Authenticated submodules

> **Note**
>
> Make sure that you have registered your submodules using HTTPS and not using SSH.

The same credentials that are used by the agent to get the sources from the main repository are also used to get the sources for submodules.

If your main repository and submodules are in an Azure Repos Git repository in your Azure DevOps project, then you can select the account used to access the sources. On the **Options** tab, on the **Build job authorization scope** menu, select either:

- **Project collection** to use the Project Collection Build service account
- **Current project** to use the Project Build Service account.

Make sure that whichever account you use has access to both the main repository as well as the submodules.

If your main repository and submodules are in the same GitHub organization, then the token stored in the GitHub service connection is used to access the sources.

Alternative to using the Checkout submodules option

In some cases you can't use the **Checkout submodules** option. You might have a scenario where a different set of credentials are needed to access the submodules. This can happen, for example, if your main repository and submodule repositories aren't stored in the same Azure DevOps organization or Git service.

If you can't use the **Checkout submodules** option, then you can instead use a custom script step to fetch submodules. First, get a personal access token (PAT) and prefix it with `pat:`. Next, base64-encode this prefixed string to create a basic auth token. Finally, add this script to your pipeline:

```
git -c http.https://<url of submodule
repository>.extraheader="AUTHORIZATION: basic
<BASE64_ENCODED_TOKEN_DESCRIBED_ABOVE>" submodule update --init --recursive
```

Be sure to replace "<BASIC_AUTH_TOKEN>" with your Base64-encoded token.

Use a secret variable in your project or build pipeline to store the basic auth token that you generated. Use that variable to populate the secret in the above Git command.

> **Note**
>
> **Q: Why can't I use a Git credential manager on the agent? A:** Storing the submodule credentials in a Git credential manager installed on your private build agent is usually not effective as the credential manager may prompt you to re-enter the credentials whenever the submodule is updated. This isn't desirable during automated builds when user interaction isn't possible.

Check out files from LFS

Select if you want to download files from large file storage (LFS).

In the classic editor, select the check box to enable this option.

In a YAML build, add a checkout step with `lfs` set to `true`:

```yaml
steps:
- checkout: self
  lfs: true
```

If you're using TFS, or if you're using Azure Pipelines with a self-hosted agent, then you must install `git-lfs` on the agent for this option to work. If your hosted agents use Windows, consider using the `System.PreferGitFromPath` variable to ensure that pipelines use the versions of git and git-lfs you installed on the machine. For more information, see What version of Git does my agent run?

Using Git LFS with submodules

If a submodule contains LFS files, Git LFS must be configured prior to checking out submodules. The Microsoft-hosted macOS and Linux agents come preconfigured this way. Windows agents and self-hosted macOS / Linux agents may not.

As a workaround, if you're using YAML, you can add the following step before your `checkout`:

```yaml
steps:
- script: |
    git config --global --add filter.lfs.required true
    git config --global --add filter.lfs.smudge "git-lfs smudge -- %f"
    git config --global --add filter.lfs.process "git-lfs filter-process"
    git config --global --add filter.lfs.clean "git-lfs clean -- %f"
  displayName: Configure LFS for use with submodules
- checkout: self
  lfs: true
  submodules: true
# ... rest of steps ...
```

Clone a second repo

By default, your pipeline is associated with one repo from Azure Repos or an external provider. This is the repo that can trigger builds on commits and pull requests.

You may want to include sources from a second repo in your pipeline. You can do this by writing a script.

```
git clone https://github.com/Microsoft/TypeScript.git
```

If the repo is not public, you will need to pass authentication to the Git command.

Azure Repos

You can clone multiple repositories in the same project as your pipeline by using multi-repo checkout.

If you need to clone a repo from another project that is not public, you will need to authenticate as a user who has access to that project.

> ⓘ **Note**
>
> Use a **secret variable** to store credentials securely.
>
> Secret variables are not automatically made available to scripts as environment variables. See **Secret variables** on how to map them in.

For Azure Repos, you can use a personal access token with the **Code (Read)** permission. Send this as the password field in a "Basic" authorization header without a username. (In other words, base64-encode the value of `:<PAT>`, including the colon.)

```
AUTH=$(echo -n ":$REPO_PAT" | openssl base64 | tr -d '\n')
git -c http.<repo URL>.extraheader="AUTHORIZATION: basic $AUTH" clone <repo
URL> --no-checkout --branch master
```

Don't sync sources

Non-deployment jobs automatically fetch sources. Use this option if you want to skip that behavior. This option can be useful in cases when you want to:

- Git init, config, and fetch using your own custom options.
- Use a build pipeline to just run automation (for example some scripts) that does not depend on code in version control.

If you want to disable downloading sources:

- **Azure Pipelines, TFS 2018, and newer:** Click **Advanced settings**, and then select **Don't sync sources**.

> **Note**
>
> When you use this option, the agent also skips running Git commands that clean the repo.

Shallow fetch

Select if you want to limit how far back in history to download. Effectively this results in `git fetch --depth=n`. If your repository is large, this option might make your build pipeline more efficient. Your repository might be large if it has been in use for a long time and has sizeable history. It also might be large if you added and later deleted large files.

In these cases this option can help you conserve network and storage resources. It might also save time. The reason it doesn't always save time is because in some situations the server might need to spend time calculating the commits to download for the depth you specify.

> **Note**
>
> When the build is queued, the branch to build is resolved to a commit ID. Then, the agent fetches the branch and checks out the desired commit. There is a small window between when a branch is resolved to a commit ID and when the agent performs the checkout. If the branch updates rapidly and you set a very small value for shallow fetch, the commit may not exist when the agent attempts to check it out. If that happens, increase the shallow fetch depth setting.

After you select the check box to enable this option, in the **Depth** box specify the number of commits.

> **Tip**
>
> The `Agent.Source.Git.ShallowFetchDepth` variable mentioned below also works and overrides the check box controls. This way you can modify the setting when you queue the build.

Prefer Git from path

By default, the Windows agent uses the version of Git that is bundled with the agent software. Microsoft recommends using the version of Git that is bundled with the agent, but you have several options to override this default behavior and use the version of Git that the agent machine has installed in the path.

- Set a pipeline variable named `System.PreferGitFromPath` to `true` in your pipelines.
- On self-hosted agents, you can create a file named **.env** in the agent root directory and add a `System.PreferGitFromPath=true` line to the file. For more information, see How do I set different environment variables for each individual agent?

To see the version of Git used by a pipeline, you can look at the logs for a `checkout` step in your pipeline, as shown in the following example.

```
Syncing repository: PathFilter (Git)
Prepending Path environment variable with directory containing 'git.exe'.
git version
git version 2.26.2.windows.1
```

This setting is always true on non-Windows agents.

Trigger Options for Other Git

When an Other/external Git repository is specified, CI builds require that the repository is accessible from the internet. If the repository is behind a firewall or proxy, then only scheduled and manual builds will work.

FAQ

What protocols can the build agent use with Git?

The agent supports HTTPS.

The agent does not yet support SSH. See Allow build to use SSH authentication while checking out Git submodules.

Build Bitbucket Cloud repositories

Article • 10/27/2023

Azure DevOps Services

Azure Pipelines can automatically build and validate every pull request and commit to your Bitbucket Cloud repository. This article describes how to configure the integration between Bitbucket Cloud and Azure Pipelines.

Bitbucket and Azure Pipelines are two independent services that integrate well together. Your Bitbucket Cloud users do not automatically get access to Azure Pipelines. You must add them explicitly to Azure Pipelines.

Access to Bitbucket repositories

YAML

You create a new pipeline by first selecting a Bitbucket Cloud repository and then a YAML file in that repository. The repository in which the YAML file is present is called `self` repository. By default, this is the repository that your pipeline builds.

You can later configure your pipeline to check out a different repository or multiple repositories. To learn how to do this, see multi-repo checkout.

Azure Pipelines must be granted access to your repositories to fetch the code during builds. In addition, the user setting up the pipeline must have admin access to Bitbucket, since that identity is used to register a webhook in Bitbucket.

There are 2 authentication types for granting Azure Pipelines access to your Bitbucket Cloud repositories while creating a pipeline.

Authentication type	Pipelines run using
1. OAuth	Your personal Bitbucket identity
2. Username and password	Your personal Bitbucket identity

OAuth authentication

OAuth is the simplest authentication type to get started with for repositories in your Bitbucket account. Bitbucket status updates will be performed on behalf of your

personal Bitbucket identity. For pipelines to keep working, your repository access must remain active.

To use OAuth, login to Bitbucket when prompted during pipeline creation. Then, click **Authorize** to authorize with OAuth. An OAuth connection will be saved in your Azure DevOps project for later use, as well as used in the pipeline being created.

> **Note**
>
> The maximum number of Bitbucket repositories that the Azure DevOps Services user interface can load is 2,000.

Password authentication

Builds and Bitbucket status updates will be performed on behalf of your personal identity. For builds to keep working, your repository access must remain active.

To create a password connection, visit Service connections in your Azure DevOps project settings. Create a new Bitbucket service connection and provide the user name and password to connect to your Bitbucket Cloud repository.

CI triggers

Continuous integration (CI) triggers cause a pipeline to run whenever you push an update to the specified branches or you push specified tags.

YAML

YAML pipelines are configured by default with a CI trigger on all branches.

Branches

You can control which branches get CI triggers with a simple syntax:

YAML

```yaml
trigger:
- main
- releases/*
```

You can specify the full name of the branch (for example, `main`) or a wildcard (for example, `releases/*`). See Wildcards for information on the wildcard syntax.

> **Note**
>
> You cannot use **variables** in triggers, as variables are evaluated at runtime (after the trigger has fired).

> **Note**
>
> If you use **templates** to author YAML files, then you can only specify triggers in the main YAML file for the pipeline. You cannot specify triggers in the template files.

For more complex triggers that use `exclude` or `batch`, you must use the full syntax as shown in the following example.

YAML

```yaml
# specific branch build
trigger:
  branches:
    include:
    - main
    - releases/*
    exclude:
    - releases/old*
```

In the above example, the pipeline will be triggered if a change is pushed to `main` or to any releases branch. However, it won't be triggered if a change is made to a releases branch that starts with `old`.

If you specify an `exclude` clause without an `include` clause, then it is equivalent to specifying `*` in the `include` clause.

In addition to specifying branch names in the `branches` lists, you can also configure triggers based on tags by using the following format:

YAML

```yaml
trigger:
  branches:
    include:
      - refs/tags/{tagname}
```

```
    exclude:
      - refs/tags/{othertagname}
```

If you don't specify any triggers, the default is as if you wrote:

YAML

```yaml
trigger:
  branches:
    include:
    - '*'  # must quote since "*" is a YAML reserved character; we want
a string
```

> **Important**
>
> When you specify a trigger, it replaces the default implicit trigger, and only pushes to branches that are explicitly configured to be included will trigger a pipeline. Includes are processed first, and then excludes are removed from that list.

Batching CI runs

If you have many team members uploading changes often, you may want to reduce the number of runs you start. If you set `batch` to `true`, when a pipeline is running, the system waits until the run is completed, then starts another run with all changes that have not yet been built.

YAML

```yaml
# specific branch build with batching
trigger:
  batch: true
  branches:
    include:
    - main
```

> **Note**
>
> `batch` is not supported in repository resource triggers.

To clarify this example, let us say that a push `A` to `main` caused the above pipeline to run. While that pipeline is running, additional pushes `B` and `C` occur into the

repository. These updates do not start new independent runs immediately. But after the first run is completed, all pushes until that point of time are batched together and a new run is started.

> ⓘ **Note**
>
> If the pipeline has multiple jobs and stages, then the first run should still reach a terminal state by completing or skipping all its jobs and stages before the second run can start. For this reason, you must exercise caution when using this feature in a pipeline with multiple stages or approvals. If you wish to batch your builds in such cases, it is recommended that you split your CI/CD process into two pipelines - one for build (with batching) and one for deployments.

Paths

You can specify file paths to include or exclude.

YAML

```yaml
# specific path build
trigger:
  branches:
    include:
    - main
    - releases/*
  paths:
    include:
    - docs
    exclude:
    - docs/README.md
```

When you specify paths, you must explicitly specify branches to trigger on if you are using Azure DevOps Server 2019.1 or lower. You can't trigger a pipeline with only a path filter; you must also have a branch filter, and the changed files that match the path filter must be from a branch that matches the branch filter. If you are using Azure DevOps Server 2020 or newer, you can omit `branches` to filter on all branches in conjunction with the path filter.

Wilds cards are supported for path filters. For instance, you can include all paths that match `src/app/**/myapp*`. You can use wild card characters (`**`, `*`, or `?`) when specifying path filters.

- Paths are always specified relative to the root of the repository.

- If you don't set path filters, then the root folder of the repo is implicitly included by default.
- If you exclude a path, you cannot also include it unless you qualify it to a deeper folder. For example if you exclude */tools* then you could include */tools/trigger-runs-on-these*
- The order of path filters doesn't matter.
- Paths in Git *are case-sensitive*. Be sure to use the same case as the real folders.
- You cannot use variables in paths, as variables are evaluated at runtime (after the trigger has fired).

> **Note**
>
> For Bitbucket Cloud repos, using `branches` syntax is the only way to specify tag triggers. The `tags:` syntax is not supported for Bitbucket.

Opting out of CI

Disabling the CI trigger

You can opt out of CI triggers entirely by specifying `trigger: none`.

YAML

```yaml
# A pipeline with no CI trigger
trigger: none
```

> **Important**
>
> When you push a change to a branch, the YAML file in that branch is evaluated to determine if a CI run should be started.

Skipping CI for individual commits

You can also tell Azure Pipelines to skip running a pipeline that a push would normally trigger. Just include `[skip ci]` in the message or description of any of the commits that are part of a push, and Azure Pipelines will skip running CI for this push. You can also use any of the following variations.

- `[skip ci]` or `[ci skip]`
- `skip-checks: true` or `skip-checks:true`
- `[skip azurepipelines]` or `[azurepipelines skip]`
- `[skip azpipelines]` or `[azpipelines skip]`
- `[skip azp]` or `[azp skip]`
- `***NO_CI***`

Using the trigger type in conditions

It is a common scenario to run different steps, jobs, or stages in your pipeline depending on the type of trigger that started the run. You can do this using the system variable `Build.Reason`. For example, add the following condition to your step, job, or stage to exclude it from PR validations.

```
condition: and(succeeded(), ne(variables['Build.Reason'], 'PullRequest'))
```

Behavior of triggers when new branches are created

It is common to configure multiple pipelines for the same repository. For instance, you may have one pipeline to build the docs for your app and another to build the source code. You may configure CI triggers with appropriate branch filters and path filters in each of these pipelines. For instance, you may want one pipeline to trigger when you push an update to the `docs` folder, and another one to trigger when you push an update to your application code. In these cases, you need to understand how the pipelines are triggered when a new branch is created.

Here is the behavior when you push a new branch (that matches the branch filters) to your repository:

- If your pipeline has path filters, it will be triggered only if the new branch has changes to files that match that path filter.
- If your pipeline does not have path filters, it will be triggered even if there are no changes in the new branch.

Wildcards

When specifying a branch, tag, or path, you may use an exact name or a wildcard. Wildcards patterns allow `*` to match zero or more characters and `?` to match a single character.

- If you start your pattern with `*` in a YAML pipeline, you must wrap the pattern in quotes, like `"*-releases"`.
- For branches and tags:
  - A wildcard may appear anywhere in the pattern.
- For paths:
  - In Azure DevOps Server 2022 and higher, including Azure DevOps Services, a wildcard may appear anywhere within a path pattern and you may use `*` or `?`.
  - In Azure DevOps Server 2020 and lower, you may include `*` as the final character, but it doesn't do anything differently from specifying the directory name by itself. You may **not** include `*` in the middle of a path filter, and you may not use `?`.

YAML

```yaml
trigger:
  branches:
    include:
    - main
    - releases/*
    - feature/*
    exclude:
    - releases/old*
    - feature/*-working
  paths:
    include:
    - docs/*.md
```

PR triggers

Pull request (PR) triggers cause a pipeline to run whenever a pull request is opened with one of the specified target branches, or when updates are made to such a pull request.

YAML

Branches

You can specify the target branches when validating your pull requests. For example, to validate pull requests that target `master` and `releases/*`, you can use the following `pr` trigger.

YAML

```yaml
pr:
- main
- releases/*
```

This configuration starts a new run the first time a new pull request is created, and after every update made to the pull request.

You can specify the full name of the branch (for example, `master`) or a wildcard (for example, `releases/*`).

> **Note**
>
> You cannot use **variables** in triggers, as variables are evaluated at runtime (after the trigger has fired).

> **Note**
>
> If you use **templates** to author YAML files, then you can only specify triggers in the main YAML file for the pipeline. You cannot specify triggers in the template files.

Each new run builds the latest commit from the source branch of the pull request. This is different from how Azure Pipelines builds pull requests in other repositories (e.g., Azure Repos or GitHub), where it builds the merge commit. Unfortunately, Bitbucket does not expose information about the merge commit, which contains the merged code between the source and target branches of the pull request.

If no `pr` triggers appear in your YAML file, pull request validations are automatically enabled for all branches, as if you wrote the following `pr` trigger. This configuration triggers a build when any pull request is created, and when commits come into the source branch of any active pull request.

YAML

```yaml
pr:
  branches:
    include:
    - '*'  # must quote since "*" is a YAML reserved character; we want
a string
```

> **Important**

When you specify a `pr` trigger, it replaces the default implicit `pr` trigger, and only pushes to branches that are explicitly configured to be included will trigger a pipeline.

For more complex triggers that need to exclude certain branches, you must use the full syntax as shown in the following example.

YAML

```yaml
# specific branch
pr:
  branches:
    include:
    - main
    - releases/*
    exclude:
    - releases/old*
```

Paths

You can specify file paths to include or exclude. For example:

YAML

```yaml
# specific path
pr:
  branches:
    include:
    - main
    - releases/*
  paths:
    include:
    - docs
    exclude:
    - docs/README.md
```

> **Tips:**
>
> - Wild cards are not supported with path filters.
> - Paths are always specified relative to the root of the repository.
> - If you don't set path filters, then the root folder of the repo is implicitly included by default.
> - If you exclude a path, you cannot also include it unless you qualify it to a deeper folder. For example if you exclude */tools* then you could include */tools/trigger-runs-on-these*

- The order of path filters doesn't matter.
- Paths in Git *are case-sensitive*. Be sure to use the same case as the real folders.
- You cannot use variables in paths, as variables are evaluated at runtime (after the trigger has fired).

Multiple PR updates

You can specify whether additional updates to a PR should cancel in-progress validation runs for the same PR. The default is `true`.

YAML

```yaml
# auto cancel false
pr:
  autoCancel: false
  branches:
    include:
    - main
```

Opting out of PR validation

You can opt out of pull request validation entirely by specifying `pr: none`.

YAML

```yaml
# no PR triggers
pr: none
```

For more information, see PR trigger in the YAML schema.

> **Note**
>
> If your `pr` trigger isn't firing, ensure that you have not **overridden YAML PR triggers in the UI**.

Informational runs

An informational run tells you Azure DevOps failed to retrieve a YAML pipeline's source code. Source code retrieval happens in response to external events, for example, a

pushed commit. It also happens in response to internal triggers, for example, to check if there are code changes and start a scheduled run or not. Source code retrieval can fail for multiple reasons, with a frequent one being request throttling by the git repository provider. The existence of an informational run doesn't necessarily mean Azure DevOps was going to run the pipeline.

An informational run looks like in the following screenshot.

You can recognize an informational run by the following attributes:

- Status is `Canceled`
- Duration is `< 1s`
- Run name contains one of the following texts:
  - `Could not retrieve file content for {file_path} from repository {repo_name} hosted on {host} using commit {commit_sha}.`
  - `Could not retrieve content for object {commit_sha} from repository {repo_name} hosted on {host}.`
  - `Could not retrieve the tree object {tree_sha} from the repository {repo_name} hosted on {host}.`
  - `Could not find {file_path} from repository {repo_name} hosted on {host} using version {commit_sha}. One of the directories in the path contains too many files or subdirectories.`
- Run name generally contains the BitBucket / GitHub error that caused the YAML pipeline load to fail
- No stages / jobs / steps

Learn more about informational runs.

Limitations

Azure Pipelines loads a maximum of 2000 branches from a repository into dropdown lists in the Azure Devops Portal, for example into the **Default branch for manual and**

scheduled builds setting, or when choosing a branch when running a pipeline manually. If you don't see your desired branch in the list, type the desired branch name manually.

FAQ

Problems related to Bitbucket integration fall into the following categories:

- **Failing triggers**: My pipeline is not being triggered when I push an update to the repo.
- **Wrong version**: My pipeline runs, but it is using an unexpected version of the source/YAML.

Failing triggers

I just created a new YAML pipeline with CI/PR triggers, but the pipeline isn't being triggered.

Follow each of these steps to troubleshoot your failing triggers:

- Are your YAML CI or PR triggers overridden by pipeline settings in the UI? While editing your pipeline, choose **...** and then **Triggers**.

  Check the **Override the YAML trigger from here** setting for the types of trigger (**Continuous integration** or **Pull request validation**) available for your repo.

- Webhooks are used to communicate updates from Bitbucket to Azure Pipelines. In Bitbucket, navigate to the settings for your repository, then to Webhooks. Verify that the webhooks exist.

- Is your pipeline paused or disabled? Open the editor for the pipeline, and then select **Settings** to check. If your pipeline is paused or disabled, then triggers do not work.

- Have you updated the YAML file in the correct branch? If you push an update to a branch, then the YAML file in that same branch governs the CI behavior. If you push an update to a source branch, then the YAML file resulting from merging the source branch with the target branch governs the PR behavior. Make sure that the YAML file in the correct branch has the necessary CI or PR configuration.

- Have you configured the trigger correctly? When you define a YAML trigger, you can specify both include and exclude clauses for branches, tags, and paths. Ensure that the include clause matches the details of your commit and that the exclude clause doesn't exclude them. Check the syntax for the triggers and make sure that it is accurate.

- Have you used variables in defining the trigger or the paths? That is not supported.

- Did you use templates for your YAML file? If so, make sure that your triggers are defined in the main YAML file. Triggers defined inside template files are not supported.

- Have you excluded the branches or paths to which you pushed your changes? Test by pushing a change to an included path in an included branch. Note that paths in triggers are case-sensitive. Make sure that you use the same case as those of real folders when specifying the paths in triggers.

- Did you just push a new branch? If so, the new branch may not start a new run. See the section "Behavior of triggers when new branches are created".

My CI or PR triggers have been working fine. But, they stopped working now.

First go through the troubleshooting steps in the previous question. Then, follow these additional steps:

- Do you have merge conflicts in your PR? For a PR that did not trigger a pipeline, open it and check whether it has a merge conflict. Resolve the merge conflict.
- Are you experiencing a delay in the processing of push or PR events? You can usually verify this by seeing if the issue is specific to a single pipeline or is common to all pipelines or repos in your project. If a push or a PR update to any of the repos exhibits this symptom, we might be experiencing delays in processing the update events. Check if we are experiencing a service outage on our status page. If the status page shows an issue, then our team must have already started working on it. Check the page frequently for updates on the issue.

I do not want users to override the list of branches for triggers when they update the YAML file. How can I do this?

Users with permissions to contribute code can update the YAML file and include/exclude additional branches. As a result, users can include their own feature or user branch in their YAML file and push that update to a feature or user branch. This may cause the pipeline to be triggered for all updates to that branch. If you want to prevent this behavior, then you can:

1. Edit the pipeline in the Azure Pipelines UI.
2. Navigate to the **Triggers** menu.
3. Select **Override the YAML continuous Integration trigger from here**.
4. Specify the branches to include or exclude for the trigger.

When you follow these steps, any CI triggers specified in the YAML file are ignored.

Wrong version

A wrong version of the YAML file is being used in the pipeline. Why is that?

- For CI triggers, the YAML file that is in the branch you are pushing is evaluated to see if a CI build should be run.
- For PR triggers, the YAML file resulting from merging the source and target branches of the PR is evaluated to see if a PR build should be run.

Build on-premises Bitbucket repositories

Article • 10/27/2023

Azure DevOps Services

> ⓘ **Note**
>
> To integrate Bitbucket Cloud with Azure Pipelines, see **Bitbucket Cloud**.

You can integrate your on-premises Bitbucket server or another Git server with Azure Pipelines. Your on-premises server might be exposed to the Internet or it might not be.

If your on-premises server is reachable from the servers that run Azure Pipelines service, then:

- you can set up classic build and configure CI triggers

If your on-premises server isn't reachable from the servers that run Azure Pipelines service, then:

- you can set up classic build pipelines and start manual builds
- you can't configure CI triggers

> ⓘ **Note**
>
> YAML pipelines do not work with on-premises Bitbucket repositories.

> ⓘ **Note**
>
> PR triggers are not available with on-premises Bitbucket repositories.

If your on-premises server is reachable from the hosted agents, then you can use the hosted agents to run manual, scheduled, or CI builds. Otherwise, you must set up self-hosted agents that can access your on-premises server and fetch the code.

Reachable from Azure Pipelines

If your on-premises Bitbucket server is reachable from Azure Pipelines service, create a Other Git service connection and use that to create a pipeline. Check the option to **Attempt accessing this Git server from Azure Pipelines**.

CI triggers work through polling and not through webhooks. In other words, Azure Pipelines periodically checks the Bitbucket server if there are any updates to code. If there are, then Azure Pipelines starts a new run.

Not reachable from Azure Pipelines

If the Bitbucket server can't be reached from Azure Pipelines, you have two options:

- Work with your IT department to open a network path between Azure Pipelines and on-premises Git server. For example, you can add exceptions to your firewall rules to allow traffic from Azure Pipelines to flow through. See the section on Azure DevOps IPs to see which IP addresses you need to allow. Furthermore, you need to have a public DNS entry for the Bitbucket server so that Azure Pipelines can resolve the FQDN of your server to an IP address.

- You can use a **Other Git** connection but tell Azure Pipelines not to **attempt accessing this Git server from Azure Pipelines**. CI and PR triggers aren't available with **Other Git** repositories. You can only start manual or scheduled pipeline runs.

Reachable from Microsoft-hosted agents

Another decision you possibly have to make is whether to use Microsoft-hosted agents or self-hosted agents to run your pipelines. This choice often depends on to whether Microsoft-hosted agents can reach your server. To check whether they can, create a pipeline to use Microsoft-hosted agents and make sure to add a step to check out source code from your server. If this passes, then you can continue using Microsoft-hosted agents.

Not reachable from Microsoft-hosted agents

If the simple test pipeline mentioned in the above section fails with the error `TF401019: The Git repository with name or identifier <your repo name> does not exist or you do not have permissions for the operation you are attempting`, then the Bitbucket server isn't reachable from Microsoft-hosted agents. This is again probably caused by a firewall blocking traffic from these servers. You have two options in this case:

- Work with your IT department to open a network path between Microsoft-hosted agents and Bitbucket server. See the section on networking in Microsoft-hosted agents.

- Switch to using self-hosted agents or scale-set agents. These agents can be set up within your network and hence will have access to the Bitbucket server. These

agents only require outbound connections to Azure Pipelines. There is no need to open a firewall for inbound connections. Make sure that the name of the server you specified when creating the service connection is resolvable from the self-hosted agents.

Azure DevOps IP addresses

When you use **Other Git** connection to set up a classic pipeline, disable communication between Azure Pipelines service and Bitbucket server, and use self-hosted agents to build code, you get a degraded experience:

- You have to type in the name of the repository manually during pipeline creation
- You can't use CI triggers as Azure Pipelines won't be able to poll for changes to the code
- You can't use scheduled triggers with the option to build only when there are changes
- You can't view information about the latest commit in the user interface

If you want to enhance this experience, it is important that you enable communication from Azure Pipelines to Bitbucket Server.

To allow traffic from Azure DevOps to reach your Bitbucket Server, add the IP addresses or service tags specified in Inbound connections to your firewall's allowlist. If you use ExpressRoute, make sure to also include ExpressRoute IP ranges to your firewall's allowlist.

Allow Azure Pipelines to attempt accessing the Git server in the **Other Git** service connection.

Informational runs

An informational run tells you Azure DevOps failed to retrieve a YAML pipeline's source code. Source code retrieval happens in response to external events, for example, a pushed commit. It also happens in response to internal triggers, for example, to check if there are code changes and start a scheduled run or not. Source code retrieval can fail for multiple reasons, with a frequent one being request throttling by the git repository provider. The existence of an informational run doesn't necessarily mean Azure DevOps was going to run the pipeline.

An informational run looks like in the following screenshot.

You can recognize an informational run by the following attributes:

- Status is `Canceled`
- Duration is `< 1s`
- Run name contains one of the following texts:
  - `Could not retrieve file content for {file_path} from repository {repo_name} hosted on {host} using commit {commit_sha}.`
  - `Could not retrieve content for object {commit_sha} from repository {repo_name} hosted on {host}.`
  - `Could not retrieve the tree object {tree_sha} from the repository {repo_name} hosted on {host}.`
  - `Could not find {file_path} from repository {repo_name} hosted on {host} using version {commit_sha}. One of the directories in the path contains too many files or subdirectories.`
- Run name generally contains the BitBucket / GitHub error that caused the YAML pipeline load to fail
- No stages / jobs / steps

Learn more about informational runs.

Limitations

Azure Pipelines loads a maximum of 2000 branches from a repository into dropdown lists in the Azure Devops Portal, for example into the **Default branch for manual and scheduled builds** setting, or when choosing a branch when running a pipeline manually. If you don't see your desired branch in the list, type the desired branch name manually.

FAQ

Problems related to Bitbucket Server integration fall into the following categories:

- **Failing triggers**: My pipeline isn't being triggered when I push an update to the repo.

- **Failing checkout:** My pipeline is being triggered, but it fails in the checkout step.

Failing triggers

I pushed a change to my server, but the pipeline isn't being triggered.

Follow each of these steps to troubleshoot your failing triggers:

- Is your Bitbucket server accessible from Azure Pipelines? Azure Pipelines periodically polls Bitbucket server for changes. If the Bitbucket server is behind a firewall, this traffic might not reach your server. For more information, see Azure DevOps IP Addresses and verify that you have granted exceptions to all the required IP addresses. These IP addresses might have changed since you have originally set up the exception rules. You can only start manual runs if you used an external Git connection and if your server isn't accessible from Azure Pipelines.

- Is your pipeline paused or disabled? Open the editor for the pipeline, and then select **Settings** to check. If your pipeline is paused or disabled, then triggers do not work.

- Have you excluded the branches or paths to which you pushed your changes? Test by pushing a change to an included path in an included branch. Note that paths in triggers are case-sensitive. Make sure that you use the same case as those of real folders when specifying the paths in triggers.

I did not push any updates to my code, however the pipeline is still being triggered.

- The continuous integration trigger for Bitbucket works through polling. After each polling interval, Azure Pipelines attempts to contact the Bitbucket server to check if there have been any updates to the code. If Azure Pipelines is unable to reach the Bitbucket server (possibly due to a network issue), then we start a new run anyway assuming that there might have been code changes. When Azure Pipelines can't retrieve a YAML pipeline's code, it will create an informational run.

Failing checkout

When I attempt to start a new run manually, there is a delay of 4-8 minutes before it starts.

- Your Bitbucket server isn't reachable from Azure Pipelines. Make sure that you have not selected the option to **attempt accessing this Git server from Azure Pipelines** in the Bitbucket service connection. If that option is selected, Azure Pipelines will attempt to contact to your server and since your server is unreachable, it eventually times out and starts the run anyway. Unchecking that option speeds up your manual runs.

The checkout step fails with the error that the server can't be resolved.

Do you use Microsoft-hosted agents? If so, these agents might not be able to reach your Bitbucket server. See Not reachable from Microsoft-hosted agents for more information.

Build TFVC repositories

Article • 08/18/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> **Important**
>
> TFVC is supported by classic pipelines only, and does not support YAML.

Choose the repository to build

While editing a pipeline that uses a TFVC repo, you have the following options.

- Clean
- Specify local path
- Label sources

Repository name

Name of the TFVC repository.

Mappings (workspace)

Include with a type value of **Map** only the folders that your build pipeline requires. If a subfolder of a mapped folder contains files that the build pipeline does not require, map it with a type value of **Cloak**.

Make sure that you **Map** all folders that contain files that your build pipeline requires. For example, if you add another project, you might have to add another mapping to the workspace.

Cloak folders you don't need. By default the root folder of project is mapped in the workspace. This configuration results in the build agent downloading all the files in the version control folder of your project. If this folder contains lots of data, your build could waste build system resources and slow down your build pipeline by downloading large amounts of data that it does not require.

When you remove projects, look for mappings that you can remove from the workspace.

If this is a CI build, in most cases you should make sure that these mappings match the filter settings of your CI trigger on the Triggers tab.

For more information on how to optimize a TFVC workspace, see Optimize your workspace.

Clean the local repo on the agent

You can perform different forms of cleaning the working directory of your self-hosted agent before a build runs.

In general, for faster performance of your self-hosted agents, don't clean the repo. In this case, to get the best performance, make sure you're also building incrementally by disabling any **Clean** option of the task or tool you're using to build.

If you do need to clean the repo (for example to avoid problems caused by residual files from a previous build), your options are below.

> ⓘ **Note**
>
> Cleaning is not relevant if you are using a **Microsoft-hosted agent** because you get a new agent every time in that case.

If you want to clean the repo, then select **true**, and then select one of the following options:

- **Sources**: The build pipeline performs an undo of any changes and scorches the current workspace under `$(Build.SourcesDirectory)`.
- **Sources and output directory**: Same operation as **Sources** option above, plus: Deletes and recreates `$(Build.BinariesDirectory)`.
- **Sources directory**: Deletes and recreates `$(Build.SourcesDirectory)`.
- **All build directories**: Deletes and recreates `$(Agent.BuildDirectory)`.

CI triggers

Select **Enable continuous integration** on the **Triggers** tab to enable this trigger if you want the build to run whenever someone checks in code.

Batch changes

Select this check box if you have many team members uploading changes often and you want to reduce the number of builds you are running. If you select this option, when a build is running, the system waits until the build is completed and then queues another build of all changes that have not yet been built.

> You can batch changes and build them together.

Path filters

Select the version control paths you want to include and exclude. In most cases, you should make sure that these filters are consistent with your TFVC mappings. You can use path filters to reduce the set of files that you want to trigger a build.

> **Tips:**
>
> - Paths are always specified relative to the root of the workspace.
> - If you don't set path filters, then the root folder of the workspace is implicitly included by default.
> - If you exclude a path, you cannot also include it unless you qualify it to a deeper folder. For example if you exclude */tools* then you could include */tools/trigger-runs-on-these*
> - The order of path filters doesn't matter.

Gated check-in

You can use gated check-in to protect against breaking changes.

By default **Use workspace mappings for filters** is selected. Builds are triggered whenever a change is checked in under a path specified in your source mappings.

Otherwise, you can clear this check box and specify the paths in the trigger.

How it affects your developers

When developers try to check-in, they are prompted to build their changes.

The system then creates a shelveset and builds it.

> ⓘ **Note**
>
> If you receive an error such as `The shelveset _Build_95;Build\6bc8a077-3f27-4936-82e6-415fbd53ba07 could not be found for check-in`, check the **Limit job authorization scope to current project for non-release pipelines** setting and ensure it is not enabled.

For details on the gated check-in experience, see Check in to a folder that is controlled by a gated check-in build pipeline.

Option to run CI builds

By default, CI builds are not run after the gated check-in process is complete and the changes are checked in.

However, if you **do** want CI builds to run after a gated check-in, select the **Run CI triggers for committed changes** check box. When you do this, the build pipeline does not add ***NO_CI*** to the changeset description. As a result, CI builds that are affected by the check-in are run.

A few other things to know

- Make sure the folders you include in your trigger are also included in your workspace mappings.
- You can run gated builds on either a Microsoft-hosted agent or a self-hosted agent.

FAQ

I get the following error when running a pipeline:

`The shelveset <xyz> could not be found for check-in`

- Is your job authorization scope set to **collection**? TFVC repositories are usually spread across the projects in your collection. You may be reading or writing to a folder that can only be accessed when the scope is the entire collection. You can set this in organization settings or in project setting under the **Pipelines** tab.

I get the following error when running a pipeline:

`The underlying connection was closed: An unexpected error occurred on a receive. ## [error]Exit code 100 returned from process: file name 'tf', arguments 'vc workspace /new /location:local /permission:Public`

- This is usually an intermittent error caused when the service is experiencing technical issues. Please re-run the pipeline.

What is scorch?

Scorch is a TFVC power tool that ensures source control on the server and the local disk are identical. See Microsoft Visual Studio Team Foundation Server 2015 Power Tools.

Build Subversion repositories

Article • 01/26/2023

Azure DevOps Services

You can integrate your on-premises Subversion server with Azure Pipelines. The Subversion server must be accessible to Azure Pipelines.

> **Note**
>
> YAML pipelines do not work with Subversion repositories.

If your server is reachable from the hosted agents, then you can use the hosted agents to run manual, scheduled, or CI builds. Otherwise, you must set up self-hosted agents that can access your on-premises server and fetch the code.

To integrate with Subversion, create a **Subversion** service connection and use that to create a pipeline. CI triggers work through polling. In other words, Azure Pipelines periodically checks the Subversion server if there are any updates to code. If there are, then Azure Pipelines will start a new run.

If the Subversion server cannot be reached from Azure Pipelines, work with your IT department to open a network path between Azure Pipelines and your server. For example, you can add exceptions to your firewall rules to allow traffic from Azure Pipelines to flow through. See the section on Azure DevOps IPs to see which IP addresses you need to allow. Furthermore, you need to have a public DNS entry for the Subversion server so that Azure Pipelines can resolve the FQDN of your server to an IP address.

Reachable from Microsoft-hosted agents

A decision you have to make is whether to use Microsoft-hosted agents or self-hosted agents to run your pipelines. This often comes down to whether Microsoft-hosted agents can reach your server. To check whether they can, create a simple pipeline to use Microsoft-hosted agents and make sure to add a step to check out source code from your server. If this passes, then you can continue using Microsoft-hosted agents.

Not reachable from Microsoft-hosted agents

If the simple test pipeline mentioned in the above section fails with an error, then the Subversion server is probably not reachable from Microsoft-hosted agents. This is probably caused by a firewall blocking traffic from these servers. You have two options in this case:

- Work with your IT department to open a network path between Microsoft-hosted agents and Subversion server. See the section on networking in Microsoft-hosted agents.
- Switch to using self-hosted agents or scale-set agents. These agents can be set up within your network and hence will have access to the Subversion server. These agents only require outbound connections to Azure Pipelines. There is no need to open a firewall for inbound connections. Make sure that the name of the server you specified when creating the service connection is resolvable from the self-hosted agents.

Azure DevOps IP addresses

To allow traffic from Azure DevOps to reach your Subversion Server, add the IP addresses or service tags specified in Inbound connections to your firewall's allow-list. If you use ExpressRoute, make sure to also include ExpressRoute IP ranges to your firewall's allow-list.

FAQ

Problems related to Subversion server integration fall into the following categories:

- **Failing triggers**: My pipeline is not being triggered when I push an update to the repo.
- **Failing checkout**: My pipeline is being triggered, but it fails in the checkout step.

Failing triggers

I pushed a change to my server, but the pipeline is not being triggered.

Follow each of these steps to troubleshoot your failing triggers:

- Is your Subversion server accessible from Azure Pipelines? Azure Pipelines periodically polls Subversion server for changes. If the Subversion server is behind a firewall, this traffic may not reach your server. See Azure DevOps IP Addresses

and verify that you have granted exceptions to all the required IP addresses. These IP addresses may have changed since you have originally set up the exception rules.

- Is your pipeline paused or disabled? Open the editor for the pipeline, and then select **Settings** to check. If your pipeline is paused or disabled, then triggers do not work.

I did not push any updates to my code, however the pipeline is still being triggered.

- The continuous integration trigger for Subversion works through polling. After each polling interval, Azure Pipelines attempts to contact the Subversion server to check if there have been any updates to the code. If Azure Pipelines is unable to reach the server (possibly due to a network issue), then we start a new run anyway assuming that there might have been code changes. In a few cases, Azure Pipelines may also create a dummy failed build with an error message to indicate that it was unable to reach the server.

Failing checkout

The checkout step fails with the error that the server cannot be resolved.

Do you use Microsoft-hosted agents? If so, these agents may not be able to reach your Bitbucket server. See Not reachable from Microsoft-hosted agents for more information.

Check out multiple repositories in your pipeline

Article • 05/03/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Pipelines often rely on multiple repositories that contain source, tools, scripts, or other items that you need to build your code. By using multiple `checkout` steps in your pipeline, you can fetch and check out other repositories in addition to the one you use to store your YAML pipeline.

Specify multiple repositories

Repositories can be specified as a repository resource, or inline with the `checkout` step.

The following repository types are supported.

Azure Repos Git (`git`)

- Azure DevOps Server (limited to repositories in the same organization)
- Azure DevOps Services

GitHub (`github`)

- Azure DevOps Services

GitHubEnterprise (`githubenterprise`)

- Azure DevOps Services

Bitbucket Cloud (`bitbucket`)

- Azure DevOps Services

> **Important**
>
> Only **Azure Repos Git** (`git`) repositories in the same organization as the pipeline are supported for multi-repo checkout in Azure DevOps Server.

> **Note**

Azure Pipelines provides **Limit job scope** settings for Azure Repos Git repositories. To check out Azure Repos Git repositories hosted in another project, **Limit job scope** must be configured to allow access. For more information, see **Limit job authorization scope**.

The following combinations of `checkout` steps are supported.

No `checkout` steps

The default behavior is as if `checkout: self` were the first step, and the current repository is checked out.

A single `checkout: none` step

No repositories are synced or checked out.

A single `checkout: self` step

The current repository is checked out.

A single `checkout` step that isn't `self` or `none`

The designated repository is checked out instead of `self`.

Multiple `checkout` steps

Each designated repository is checked out to a folder named after the repository, unless a different `path` is specified in the `checkout` step. To check out `self` as one of the repositories, use `checkout: self` as one of the `checkout` steps.

> **Note**
>
> When you check out Azure Repos Git repositories other than the one containing the pipeline, you may be prompted to authorize access to that resource before the pipeline runs for the first time. For more information, see **Why am I prompted to authorize resources the first time I try to check out a different repository?** in the **FAQ** section.

Repository resource definition

You must use a repository resource if your repository type requires a service connection or other extended resources field. The following repository types require a service connection.

Repository type	Service connection
Bitbucket Cloud	Bitbucket Cloud
GitHub	GitHub
GitHub Enterprise Server	GitHub Enterprise Server
Azure Repos Git repositories in a different organization than your pipeline	Azure Repos/Team Foundation Server

You may use a repository resource even if your repository type doesn't require a service connection, for example if you have a repository resource defined already for templates in a different repository.

In the following example, three repositories are declared as repository resources. The Azure Repos Git repository in another organization, GitHub, and Bitbucket Cloud repository resources require service connections, which are specified as the `endpoint` for those repository resources. This example has four `checkout` steps, which checks out the three repositories declared as repository resources along with the current `self` repository that contains the pipeline YAML.

YAML

```yaml
resources:
  repositories:
  - repository: MyGitHubRepo # The name used to reference this repository in the checkout step
    type: github
    endpoint: MyGitHubServiceConnection
    name: MyGitHubOrgOrUser/MyGitHubRepo
  - repository: MyBitbucketRepo
    type: bitbucket
    endpoint: MyBitbucketServiceConnection
    name: MyBitbucketOrgOrUser/MyBitbucketRepo
  - repository: MyAzureReposGitRepository # In a different organization
    endpoint: MyAzureReposGitServiceConnection
    type: git
    name: OtherProject/MyAzureReposGitRepo

trigger:
- main

pool:
  vmImage: 'ubuntu-latest'
```

```yaml
steps:
- checkout: self
- checkout: MyGitHubRepo
- checkout: MyBitbucketRepo
- checkout: MyAzureReposGitRepository

- script: dir $(Build.SourcesDirectory)
```

If the `self` repository is named `CurrentRepo`, the `script` command produces the following output: `CurrentRepo MyAzureReposGitRepo MyBitbucketRepo MyGitHubRepo`. In this example, the names of the repositories (as specified by the `name` property in the repository resource) are used for the folders, because no `path` is specified in the checkout step. For more information on repository folder names and locations, see the following Checkout path section.

Inline syntax checkout

If your repository doesn't require a service connection, you can declare it inline with your `checkout` step.

> **Note**
>
> Only Azure Repos Git repositories in the same organization can use the inline syntax. Azure Repos Git repositories in a different organization, and other supported repository types require a **service connection** and must be declared as a **repository resource**.

YAML

```yaml
steps:
- checkout: git://MyProject/MyRepo # Azure Repos Git repository in the same organization
```

> **Note**
>
> In the previous example, the `self` repository is not checked out. If you specify any `checkout` steps, you must include `checkout: self` in order for `self` to be checked out.

Checkout path

Unless a `path` is specified in the `checkout` step, source code is placed in a default directory. This directory is different depending on whether you are checking out a single repository or multiple repositories.

- **Single repository**: If you have a single `checkout` step in your job, or you have no checkout step which is equivalent to `checkout: self`, your source code is checked out into a directory called `s` located as a subfolder of `(Agent.BuildDirectory)`. If `(Agent.BuildDirectory)` is `C:\agent\_work\1`, your code is checked out to `C:\agent\_work\1\s`.

- **Multiple repositories**: If you have multiple `checkout` steps in your job, your source code is checked out into directories named after the repositories as a subfolder of `s` in `(Agent.BuildDirectory)`. If `(Agent.BuildDirectory)` is `C:\agent\_work\1` and your repositories are named `tools` and `code`, your code is checked out to `C:\agent\_work\1\s\tools` and `C:\agent\_work\1\s\code`.

  > ⓘ **Note**
  >
  > If no `path` is specified in the `checkout` step, the name of the repository is used for the folder, not the `repository` value which is used to reference the repository in the `checkout` step.

If a `path` is specified for a `checkout` step, that path is used, relative to `(Agent.BuildDirectory)`.

> ⓘ **Note**
>
> If you are using default paths, adding a second repository `checkout` step changes the default path of the code for the first repository. For example, the code for a repository named `tools` would be checked out to `C:\agent\_work\1\s` when `tools` is the only repository, but if a second repository is added, `tools` would then be checked out to `C:\agent\_work\1\s\tools`. If you have any steps that depend on the source code being in the original location, those steps must be updated.

Checking out a specific ref

The default branch is checked out unless you designate a specific ref.

If you are using inline syntax, designate the ref by appending `@<ref>`. For example:

YAML

```yaml
- checkout: git://MyProject/MyRepo@features/tools # checks out the
features/tools branch
- checkout: git://MyProject/MyRepo@refs/heads/features/tools # also checks
out the features/tools branch
- checkout: git://MyProject/MyRepo@refs/tags/MyTag # checks out the commit
referenced by MyTag.
```

When using a repository resource, specify the ref using the `ref` property. The following example checks out the `features/tools/` branch of the designated repository.

YAML

```yaml
resources:
  repositories:
  - repository: MyGitHubRepo
    type: github
    endpoint: MyGitHubServiceConnection
    name: MyGitHubOrgOrUser/MyGitHubRepo
    ref: features/tools

steps:
- checkout: MyGitHubRepo
```

The following example uses tags to check out the commit referenced by `MyTag`.

YAML

```yaml
resources:
  repositories:
  - repository: MyGitHubRepo
    type: github
    endpoint: MyGitHubServiceConnection
    name: MyGitHubOrgOrUser/MyGitHubRepo
    ref: refs/tags/MyTag

steps:
- checkout: MyGitHubRepo
```

Triggers

You can trigger a pipeline when an update is pushed to the `self` repository or to any of the repositories declared as resources. This is useful, for instance, in the following

scenarios:

- You consume a tool or a library from a different repository. You want to run tests for your application whenever the tool or library is updated.
- You keep your YAML file in a separate repository from the application code. You want to trigger the pipeline every time an update is pushed to the application repository.

> **Important**
>
> Repository resource triggers only work for Azure Repos Git repositories in the same organization at present. They do not work for GitHub or Bitbucket repository resources.
>
> `batch` is not supported in repository resource triggers.

If you do not specify a `trigger` section in a repository resource, then the pipeline won't be triggered by changes to that repository. If you specify a `trigger` section, then the behavior for triggering is similar to how CI triggers work for the self repository.

If you specify a `trigger` section for multiple repository resources, then a change to any of them will start a new run.

When a pipeline is triggered, Azure Pipelines has to determine the version of the YAML file that should be used and a version for each repository that should be checked out. If a change to the `self` repository triggers a pipeline, then the commit that triggered the pipeline is used to determine the version of the YAML file. If a change to any other repository resource triggers the pipeline, then the latest version of YAML from the **default branch** of `self` repository is used.

When an update to one of the repositories triggers a pipeline, then the following variables are set based on triggering repository:

- `Build.Repository.ID`
- `Build.Repository.Name`
- `Build.Repository.Provider`
- `Build.Repository.Uri`
- `Build.SourceBranch`
- `Build.SourceBranchName`
- `Build.SourceVersion`
- `Build.SourceVersionMessage`

For the triggering repository, the commit that triggered the pipeline determines the version of the code that is checked out. For other repositories, the `ref` defined in the YAML for that repository resource determines the default version that is checked out.

Consider the following example, where the `self` repository contains the YAML file and repositories `A` and `B` contain additional source code.

YAML

```yaml
trigger:
- main
- feature

resources:
  repositories:
  - repository: A
    type: git
    name: MyProject/A
    ref: main
    trigger:
    - main

  - repository: B
    type: git
    name: MyProject/B
    ref: release
    trigger:
    - main
    - release
steps:
- checkout: self
- checkout: A
- checkout: B
```

The following table shows which versions are checked out for each repository by a pipeline using the above YAML file.

Change made to	Pipeline triggered	Version of YAML	Version of `self`	Version of `A`	Version of `B`
`main` in `self`	Yes	commit from `main` that triggered the pipeline	commit from `main` that triggered the pipeline	latest from `main`	latest from `release`

Change made to	Pipeline triggered	Version of YAML	Version of `self`	Version of `A`	Version of `B`
`feature` in `self`	Yes	commit from `feature` that triggered the pipeline	commit from `feature` that triggered the pipeline	latest from `main`	latest from `release`
`main` in `A`	Yes	latest from `main`	latest from `main`	commit from `main` that triggered the pipeline	latest from `release`
`main` in `B`	Yes	latest from `main`	latest from `main`	latest from `main`	commit from `main` that triggered the pipeline
`release` in `B`	Yes	latest from `main`	latest from `main`	latest from `main`	commit from `release` that triggered the pipeline

You can also trigger the pipeline when you create or update a pull request in any of the repositories. To do this, declare the repository resources in the YAML files as in the examples above, and configure a branch policy in the repository (Azure Repos only).

Repository details

When you check out multiple repositories, some details about the `self` repository are available as variables. When you use multi-repo triggers, some of those variables have information about the triggering repository instead. Details about all of the repositories consumed by the job are available as a template context object called `resources.repositories`.

For example, to get the ref of a non-`self` repository, you could write a pipeline like this:

YAML

```yaml
resources:
  repositories:
  - repository: other
    type: git
    name: MyProject/OtherTools

variables:
```

```
  tools.ref: $[ resources.repositories['other'].ref ]

steps:
- checkout: self
- checkout: other
- bash: |
    echo "Tools version: $TOOLS_REF"
```

FAQ

- Why can't I check out a repository from another project? It used to work.
- Why am I prompted to authorize resources the first time I try to check out a different repository?

Why can't I check out a repository from another project? It used to work.

Azure Pipelines provides a **Limit job authorization scope to current project** setting, that when enabled, doesn't permit the pipeline to access resources outside of the project that contains the pipeline. This setting can be set at either the organization or project level. If this setting is enabled, you won't be able to check out a repository in another project unless you explicitly grant access. For more information, see Job authorization scope.

Why am I prompted to authorize resources the first time I try to check out a different repository?

When you check out Azure Repos Git repositories other than the one containing the pipeline, you may be prompted to authorize access to that resource before the pipeline runs for the first time. These prompts are displayed on the pipeline run summary page.

Choose **View** or **Authorize resources**, and follow the prompts to authorize the resources.

For more information, see Troubleshooting authorization for a YAML pipeline.

Build pipeline history

Article • 04/05/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

From the **History** tab you can see a list of changes that includes who made the change and when the change occurred.

To work with a change, select it, click **...**, and then click **Compare Difference** or **Revert Pipeline**.

FAQ

Can I edit the JSON source directly?

No

Do I need an agent?

You need at least one agent to run your build or release.

I'm having problems. How can I troubleshoot them?

See Troubleshoot Build and Release.

I can't select a default agent pool and I can't queue my build or release. How do I fix this?

See Agent pools.

My NuGet push task is failing with the following error: "Error: unable to get local issuer certificate". How can I fix this?

This can be fixed by adding a trusted root certificate. You can either add the `NODE_EXTRA_CA_CERTS=file` environment variable to your build agent, or you can add the `NODE.EXTRA.CA.CERTS=file` task variable in your pipeline. See Node.js documentation

for more details about this variable. See Set variables in a pipeline for instructions on setting a variable in your pipeline.

Specify events that trigger pipelines

Article • 02/10/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Use triggers to run a pipeline automatically. Azure Pipelines supports many types of triggers. Based on your pipeline's type, select the appropriate trigger from the lists below.

> **Note**
>
> All trigger paths are case-sensitive.

Classic build pipelines and YAML pipelines

Continuous integration (CI) triggers vary based on the type of repository you build in your pipeline.

- CI triggers in Azure Repos Git
- CI triggers in GitHub
- CI triggers in Bitbucket Cloud
- CI triggers in TFVC

Pull request validation (PR) triggers also vary based on the type of repository.

- PR triggers in Azure Repos Git
- PR triggers in GitHub
- PR triggers in Bitbucket Cloud

Gated check-in is supported for TFVC repositories.

Comment triggers are supported only for GitHub repositories.

Scheduled triggers are independent of the repository and allow you to run a pipeline according to a schedule.

Pipeline triggers in YAML pipelines and build completion triggers in classic build pipelines allow you to trigger one pipeline upon the completion of another.

Branch consideration for triggers in YAML pipelines

YAML pipelines can have different versions of the pipeline in different branches, which can affect which version of the pipeline's triggers are evaluated and which version of the pipeline should run.

Trigger type	Pipeline YAML version
CI triggers (`trigger`)	The version of the pipeline in the pushed branch is used.
PR triggers (`pr`)	The version of the pipeline in the source branch for the pull request is used.
GitHub pull request comment triggers	The version of the pipeline in the source branch for the pull request is used.
Scheduled triggers	See Branch considerations for scheduled triggers.
Pipeline completion triggers	See Branch considerations for pipeline completion triggers.
Azure Pipelines Action	See Branch considerations.

Classic release pipelines

Continuous deployment triggers help you start classic releases after a classic build or YAML pipeline completes.

Scheduled release triggers allow you to run a release pipeline according to a schedule.

Pull request release triggers are used to deploy a pull request directly using classic releases.

Stage triggers in classic release are used to configure how each stage in a classic release is triggered.

Configure schedules for pipelines

Article • 10/10/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Azure Pipelines provides several types of triggers to configure how your pipeline starts.

- Scheduled triggers start your pipeline based on a schedule, such as a nightly build. This article provides guidance on using scheduled triggers to run your pipelines based on a schedule.
- Event-based triggers start your pipeline in response to events, such as creating a pull request or pushing to a branch. For information on using event-based triggers, see Triggers in Azure Pipelines.

You can combine scheduled and event-based triggers in your pipelines, for example to validate the build every time a push is made (CI trigger), when a pull request is made (PR trigger), and a nightly build (Scheduled trigger). If you want to build your pipeline only on a schedule, and not in response to event-based triggers, ensure that your pipeline doesn't have any other triggers enabled. For example, YAML pipelines in a GitHub repository have CI triggers and PR triggers enabled by default. For information on disabling default triggers, see Triggers in Azure Pipelines and navigate to the section that covers your repository type.

Scheduled triggers

YAML

> **Important**
>
> Scheduled triggers defined using the pipeline settings UI take precedence over YAML scheduled triggers.
>
> If your YAML pipeline has both YAML scheduled triggers and UI defined scheduled triggers, only the UI defined scheduled triggers are run. To run the YAML defined scheduled triggers in your YAML pipeline, you must remove the scheduled triggers defined in the pipeline settings UI. Once all UI scheduled triggers are removed, a push must be made in order for the YAML scheduled triggers to start being evaluated.

To delete UI scheduled triggers from a YAML pipeline, see **UI settings override YAML scheduled triggers**.

Scheduled triggers configure a pipeline to run on a schedule defined using cron syntax.

YAML

```yaml
schedules:
- cron: string # cron syntax defining a schedule
  displayName: string # friendly name given to a specific schedule
  branches:
    include: [ string ] # which branches the schedule applies to
    exclude: [ string ] # which branches to exclude from the schedule
  always: boolean # whether to always run the pipeline or only if there
have been source code changes since the last successful scheduled run.
The default is false.
  batch: boolean # Whether to run the pipeline if the previously
scheduled run is in-progress; the default is false.
  # batch is available in Azure DevOps Server 2022.1 and higher
```

Scheduled pipelines in YAML have the following constraints.

- The time zone for cron schedules is UTC.
- If you specify an `exclude` clause without an `include` clause for `branches`, it's equivalent to specifying `*` in the `include` clause.
- You can't use pipeline variables when specifying schedules.
- If you use templates in your YAML file, then the schedules must be specified in the main YAML file and not in the template files.

Branch considerations for scheduled triggers

Scheduled triggers are evaluated for a branch when the following events occur.

- A pipeline is created.
- A pipeline's YAML file is updated, either from a push, or by editing it in the pipeline editor.
- A pipeline's YAML file path is updated to reference a different YAML file. This change only updates the default branch, and therefore only picks up schedules in the updated YAML file for the default branch. If any other branches subsequently merge the default branch, for example `git pull origin main`, the scheduled triggers from the newly referenced YAML file are evaluated for that branch.
- A new branch is created.

After one of these events occurs in a branch, any scheduled runs for that branch are added, if that branch matches the branch filters for the scheduled triggers contained in the YAML file in that branch.

> **Important**
>
> Scheduled runs for a branch are added only if the branch matches the branch filters for the scheduled triggers in the YAML file **in that particular branch**.

For example, a pipeline is created with the following schedule, and this version of the YAML file is checked into the `main` branch. This schedule builds the `main` branch on a daily basis.

YAML

```yaml
# YAML file in the main branch
schedules:
- cron: '0 0 * * *'
  displayName: Daily midnight build
  branches:
    include:
    - main
```

Next, a new branch is created based off of `main`, named `new-feature`. The scheduled triggers from the YAML file in the new branch are read, and since there's no match for the `new-feature` branch, no changes are made to the scheduled builds, and the `new-feature` branch isn't built using a scheduled trigger.

If `new-feature` is added to the `branches` list and this change is pushed to the `new-feature` branch, the YAML file is read, and since `new-feature` is now in the branches list, a scheduled build is added for the `new-feature` branch.

YAML

```yaml
# YAML file in the new-feature-branch
schedules:
- cron: '0 0 * * *'
  displayName: Daily midnight build
  branches:
    include:
    - main
    - new-feature
```

Now consider that a branch named `release` is created based off `main`, and then `release` is added to the branch filters in the YAML file in the `main` branch, but not in the newly created `release` branch.

YAML

```yaml
# YAML file in the release branch
schedules:
- cron: '0 0 * * *'
  displayName: Daily midnight build
  branches:
    include:
    - main

# YAML file in the main branch with release added to the branches list
schedules:
- cron: '0 0 * * *'
  displayName: Daily midnight build
  branches:
    include:
    - main
    - release
```

Because `release` was added to the branch filters in the `main` branch, but **not** to the branch filters in the `release` branch, the `release` branch won't be built on that schedule. Only when the `release` branch is added to the branch filters in the YAML file **in the release branch** will the scheduled build be added to the scheduler.

Batch considerations for scheduled triggers

The `batch` property configures whether to run the pipeline if the previously scheduled run is in-progress; the default is `false`. This is regardless of the version of the pipeline repository.

The following table describes how `always` and `batch` interact.

Always	Batch	Behavior
`false`	`false`	Pipeline runs only if there's a change with respect to the last successful scheduled pipeline run.
`false`	`true`	Pipeline runs only if there's a change with respect to the last successful scheduled pipeline run, and there's no in-progress scheduled pipeline run.
`true`	`false`	Pipeline runs according to the cron schedule.
`true`	`true`	Pipeline runs according to the cron schedule.

> **Important**
>
> When `always` is `true`, the pipeline runs according to the cron schedule, even when `batch` is `true`.

Build.CronSchedule.DisplayName variable

When a pipeline is running due to a cron scheduled trigger, the pre-defined `Build.CronSchedule.DisplayName` variable contains the `displayName` of the cron schedule that triggered the pipeline run.

Your YAML pipeline may contain multiple cron schedules, and you may want your pipeline to run different stages or jobs based on which cron schedule runs. For example, you have a nightly build and a weekly build, and you want to run a certain stage only during the nightly build. You can use the `Build.CronSchedule.DisplayName` variable in a job or stage condition to determine whether to run that job or stage.

yml

```yml
- stage: stage1
  # Run this stage only when the pipeline is triggered by the
  # "Daily midnight build" cron schedule
  condition: eq(variables['Build.CronSchedule.DisplayName'], 'Daily
midnight build')
```

For more examples, see schedules.cron examples.

Examples

YAML

The following example defines two schedules:

YAML

```yaml
schedules:
- cron: '0 0 * * *'
  displayName: Daily midnight build
  branches:
    include:
```

```
    - main
    - releases/*
    exclude:
    - releases/ancient/*
- cron: '0 12 * * 0'
  displayName: Weekly Sunday build
  branches:
    include:
    - releases/*
  always: true
```

The first schedule, **Daily midnight build**, runs a pipeline at midnight every day, but only if the code has changed since the last successful scheduled run, for `main` and all `releases/*` branches, except the branches under `releases/ancient/*`.

The second schedule, **Weekly Sunday build**, runs a pipeline at noon on Sundays, whether the code has changed or not since the last run, for all `releases/*` branches.

> **Note**
>
> The time zone for cron schedules is UTC, so in these examples, the midnight build and the noon build are at midnight and noon in UTC.

For more examples, see Migrating from the classic editor.

Cron syntax

YAML

Each Azure Pipelines scheduled trigger cron expression is a space-delimited expression with five entries in the following order. The expression is enclosed in single quotes `'`.

```
mm HH DD MM DW
 \  \  \  \  \__ Days of week
  \  \  \  \____ Months
   \  \  \______ Days
    \  \________ Hours
     \__________ Minutes
```

Field	Accepted values
Minutes	0 through 59
Hours	0 through 23
Days	1 through 31
Months	1 through 12, full English names, first three letters of English names
Days of week	0 through 6 (starting with Sunday), full English names, first three letters of English names

Values can be in the following formats.

Format	Example	Description
Wildcard	`*`	Matches all values for this field
Single value	`5`	Specifies a single value for this field
Comma delimited	`3,5,6`	Specifies multiple values for this field. Multiple formats can be combined, like `1,3-6`
Ranges	`1-3`	The inclusive range of values for this field
Intervals	`*/4` or `1-5/2`	Intervals to match for this field, such as every fourth value or the range 1-5 with a step interval of 2

Example	Cron expression
Build every Monday, Wednesday, and Friday at 6:00 PM	`0 18 * * Mon,Wed,Fri`, `0 18 * * 1,3,5`, or `0 18 * * 1-5/2`
Build every 6 hours	`0 0,6,12,18 * * *`, `0 */6 * * *` or `0 0-18/6 * * *`
Build every 6 hours starting at 9:00 AM	`0 9,15,21 * * *` or `0 9-21/6 * * *`

For more information on supported formats, see Crontab Expression.

Scheduled runs view

YAML

You can view a preview of upcoming scheduled builds by choosing **Scheduled runs** from the context menu on the pipeline details page for your pipeline.

> **Important**
>
> The scheduled runs view only shows pipelines scheduled to run within seven days from the current date. If your cron schedule has an interval longer than 7 days and the next run is scheduled to start after seven days from the current date, it won't be displayed in the scheduled runs view.

After you create or update your scheduled triggers, you can verify them using this view.

This example displays the scheduled runs for the following schedule.

YAML

```yaml
schedules:
- cron: '0 0 * * *'
  displayName: Daily midnight build
  branches:
    include:
    - main
```

The **Scheduled runs** windows displays the times converted to the local time zone set on the computer used to browse to the Azure DevOps portal. This example displays a screenshot taken in the EST time zone.

Running even when there are no code changes

By default, your pipeline doesn't run as scheduled if there have been no code changes since the last successful scheduled run. For instance, consider that you've scheduled a pipeline to run every night at 9:00pm. During the weekdays, you push various changes to your code. The pipeline runs as per schedule. During the weekends, you don't make

any changes to your code. If there have been no code changes since the scheduled run on Friday, then the pipeline doesn't run as scheduled during the weekend.

YAML

To force a pipeline to run even when there are no code changes, you can use the `always` keyword.

YAML

```yaml
schedules:
- cron: ...
  ...
  always: true
```

Limits on the number of scheduled runs in YAML pipelines

There are certain limits on how often you can schedule a pipeline to run. These limits have been put in place to prevent misuse of Azure Pipelines resources, particularly the Microsoft-hosted agents. The limits are:

- around 1000 runs per pipeline per week
- 10 runs per pipeline per 15 minutes

Migrating from the classic editor

The following examples show you how to migrate your schedules from the classic editor to YAML.

- Example: Nightly build of Git repo in multiple time zones
- Example: Nightly build with different frequencies

Example: Nightly build of Git repo in multiple time zones

In this example, the classic editor scheduled trigger has two entries, producing the following builds.

- Every Monday - Friday at 3:00 AM (UTC + 5:30 time zone), build branches that meet the `features/india/*` branch filter criteria

- Every Monday - Friday at 3:00 AM (UTC - 5:00 time zone), build branches that meet the `features/nc/*` branch filter criteria

The equivalent YAML scheduled trigger is:

YAML

```yaml
schedules:
- cron: '30 21 * * Sun-Thu'
  displayName: M-F 3:00 AM (UTC + 5:30) India daily build
  branches:
    include:
    - /features/india/*
- cron: '0 8 * * Mon-Fri'
```

```
  displayName: M-F 3:00 AM (UTC - 5) NC daily build
  branches:
    include:
    - /features/nc/*
```

In the first schedule, **M-F 3:00 AM (UTC + 5:30) India daily build**, the cron syntax (`mm HH DD MM DW`) is `30 21 * * Sun-Thu`.

- Minutes and Hours - `30 21` - This maps to `21:30 UTC` (`9:30 PM UTC`). Since the specified time zone in the classic editor is **UTC + 5:30**, we need to subtract 5 hours and 30 minutes from the desired build time of 3:00 AM to arrive at the desired UTC time to specify for the YAML trigger.
- Days and Months are specified as wildcards since this schedule doesn't specify to run only on certain days of the month or on a specific month.
- Days of the week - `Sun-Thu` - because of the timezone conversion, for our builds to run at 3:00 AM in the UTC + 5:30 India time zone, we need to specify starting them the previous day in UTC time. We could also specify the days of the week as `0-4` or `0,1,2,3,4`.

In the second schedule, **M-F 3:00 AM (UTC - 5) NC daily build**, the cron syntax is `0 8 * * Mon-Fri`.

- Minutes and Hours - `0 8` - This maps to `8:00 AM UTC`. Since the specified time zone in the classic editor is **UTC - 5:00**, we need to add 5 hours from the desired build time of 3:00 AM to arrive at the desired UTC time to specify for the YAML trigger.
- Days and Months are specified as wildcards since this schedule doesn't specify to run only on certain days of the month or on a specific month.
- Days of the week - `Mon-Fri` - Because our timezone conversions don't span multiple days of the week for our desired schedule, we don't need to do any conversion here. We could also specify the days of the week as `1-5` or `1,2,3,4,5`.

> **Important**
>
> The UTC time zones in YAML scheduled triggers don't account for daylight saving time.

> **Tip**
>
> When using 3 letter days of the week and wanting a span of multiple days through Sun, Sun should be considered the first day of the week e.g. For a schedule of

midnight EST, Thursday to Sunday, the cron syntax is `0 5 * * Sun,Thu-Sat`.

Example: Nightly build with different frequencies

In this example, the classic editor scheduled trigger has two entries, producing the following builds.

- Every Monday - Friday at 3:00 AM UTC, build branches that meet the `main` and `releases/*` branch filter criteria

- Every Sunday at 3:00 AM UTC, build the `releases/lastversion` branch, even if the source or pipeline hasn't changed

The equivalent YAML scheduled trigger is:

YAML

```yaml
schedules:
- cron: '0 3 * * Mon-Fri'
  displayName: M-F 3:00 AM (UTC) daily build
  branches:
    include:
    - main
    - /releases/*
- cron: '0 3 * * Sun'
  displayName: Sunday 3:00 AM (UTC) weekly latest version build
  branches:
    include:
    - /releases/lastversion
  always: true
```

In the first schedule, **M-F 3:00 AM (UTC) daily build**, the cron syntax is `0 3 * * Mon-Fri`.

- Minutes and Hours - `0 3` - This maps to `3:00 AM UTC`. Since the specified time zone in the classic editor is **UTC**, we don't need to do any time zone conversions.
- Days and Months are specified as wildcards since this schedule doesn't specify to run only on certain days of the month or on a specific month.
- Days of the week - `Mon-Fri` - because there's no timezone conversion, the days of the week map directly from the classic editor schedule. We could also specify the days of the week as `1,2,3,4,5`.

In the second schedule, **Sunday 3:00 AM (UTC) weekly latest version build**, the cron syntax is `0 3 * * Sun`.

- Minutes and Hours - `0 3` - This maps to `3:00 AM UTC`. Since the specified time zone in the classic editor is **UTC**, we don't need to do any time zone conversions.
- Days and Months are specified as wildcards since this schedule doesn't specify to run only on certain days of the month or on a specific month.
- Days of the week - `Sun` - Because our timezone conversions don't span multiple days of the week for our desired schedule, we don't need to do any conversion here. We could also specify the days of the week as `0`.
- We also specify `always: true` since this build is scheduled to run whether or not the source code has been updated.

FAQ

- I want my pipeline to run only on the schedule and not when someone pushes a change to a branch
- I defined a schedule in the YAML file. But it didn't run. What happened?
- My YAML schedules were working fine. But, they stopped working now. How do I debug this?
- My code hasn't changed, yet a scheduled build is triggered. Why?
- I see the planned run in the Scheduled runs panel. However, it doesn't run at that time. Why?
- Schedules defined in YAML pipeline work for one branch but not the other. How do I fix this?

I want my pipeline to run only on the schedule and not when someone pushes a change to a branch

If you want your pipeline to run only on the schedule, and not when someone pushes a change to a branch or merges a change to the main branch, you must explicitly disabled the default CI and PR triggers on the pipeline.

To disable the default CI and PR triggers, add the following statements to your YAML pipeline, and verify that you haven't overridden the YAML pipeline triggers with UI triggers.

YAML

```yaml
trigger: none
```

```
pr: none
```

For more information, see pr definition and trigger definition.

I defined a schedule in the YAML file. But it didn't run. What happened?

- Check the next few runs that Azure Pipelines has scheduled for your pipeline. You can find these runs by selecting the **Scheduled runs** action in your pipeline. The list is filtered down to only show you the upcoming few runs over the next few days. If this doesn't meet your expectation, it's probably the case that you've mistyped your cron schedule, or you don't have the schedule defined in the correct branch. Read the topic above to understand how to configure schedules. Reevaluate your cron syntax. All the times for cron schedules are in UTC.
- Make a small trivial change to your YAML file and push that update into your repository. If there was any problem in reading the schedules from the YAML file earlier, it should be fixed now.
- If you have any schedules defined in the UI, then your YAML schedules aren't honored. Ensure that you don't have any UI schedules by navigating to the editor for your pipeline and then selecting **Triggers**.
- There's a limit on the number of runs you can schedule for a pipeline. Read more about limits.
- If there are no changes to your code, they Azure Pipelines may not start new runs. Learn how to override this behavior.

My YAML schedules were working fine. But, they stopped working now. How do I debug this?

- If you didn't specify `always:true`, your pipeline won't be scheduled unless there are any updates made to your code. Check whether there have been any code changes and how you configured the schedules.
- There's a limit on how many times you can schedule your pipeline. Check if you've exceeded those limits.
- Check if someone enabled more schedules in the UI. Open the editor for your pipeline, and select **Triggers**. If they defined schedules in the UI, then your YAML schedules won't be honored.

- Check if your pipeline is paused or disabled. Select **Settings** for your pipeline.
- Check the next few runs that Azure Pipelines has scheduled for your pipeline. You can find these runs by selecting the **Scheduled runs** action in your pipeline. If you don't see the schedules that you expected, make a small trivial change to your YAML file, and push the update to your repository. This should resync the schedules.
- If you use GitHub for storing your code, it's possible that Azure Pipelines may have been throttled by GitHub when it tried to start a new run. Check if you can start a new run manually.

My code hasn't changed, yet a scheduled build is triggered. Why?

- You might have enabled an option to **always** run a scheduled build even if there are no code changes. If you use a YAML file, verify the syntax for the schedule in the YAML file. If you use classic pipelines, verify if you checked this option in the scheduled triggers.
- You might have updated the build pipeline or some property of the pipeline. This will cause a new run to be scheduled even if you haven't updated your source code. Verify the **History** of changes in the pipeline using the classic editor.
- You might have updated the service connection used to connect to the repository. This will cause a new run to be scheduled even if you haven't updated your source code.
- Azure Pipelines first checks if there are any updates to your code. If Azure Pipelines is unable to reach your repository or get this information, it will create an informational run. It's a dummy build to let you know that Azure Pipelines is unable to reach your repository.
- Your pipeline may not have a completely successful build. In order to determine whether to schedule a new build or not, Azure DevOps looks up the last completely successful scheduled build. If it doesn't find one, it triggers a new scheduled build. Partially successful scheduled builds aren't considered successful, so if your pipeline only has partially successful builds, Azure DevOps will trigger scheduled builds, even if your code hasn't changed.

I see the planned run in the Scheduled runs panel. However, it doesn't run at that time. Why?

- The **Scheduled runs** panel shows all potential schedules. However, it may not actually run unless you have made real updates to the code. To force a schedule to always run, ensure that you have set the **always** property in the YAML pipeline, or checked the option to always run in a classic pipeline.

Schedules defined in YAML pipeline work for one branch but not the other. How do I fix this?

Schedules are defined in YAML files, and these files are associated with branches. If you want a pipeline to be scheduled for a particular branch, say `features/X`, then make sure that the YAML file **in that branch** has the cron schedule defined in it, and that it has the correct branch inclusions for the schedule. The YAML file in the `features/X` branch should have the following `schedule` in this example:

YAML

```yaml
schedules:
- cron: '0 12 * * 0'   # replace with your schedule
  branches:
    include:
    - features/X
```

For more information, see Branch considerations for scheduled triggers.

Trigger one pipeline after another

Article • 10/10/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Large products have several components that are dependent on each other. These components are often independently built. When an upstream component (a library, for example) changes, the downstream dependencies have to be rebuilt and revalidated.

In situations like these, add a pipeline trigger to run your pipeline upon the successful completion of the **triggering pipeline**.

> **Note**
>
> Previously, you may have navigated to the classic editor for your YAML pipeline and configured **build completion triggers** in the UI. While that model still works, it is no longer recommended. The recommended approach is to specify **pipeline triggers** directly within the YAML file. Build completion triggers as defined in the classic editor have various drawbacks, which have now been addressed in pipeline triggers. For instance, there is no way to trigger a pipeline on the same branch as that of the triggering pipeline using build completion triggers.

Configure pipeline resource triggers

To trigger a pipeline upon the completion of another pipeline, configure a pipeline resource trigger.

The following example configures a pipeline resource trigger so that a pipeline named `app-ci` runs after any run of the `security-lib-ci` pipeline completes.

This example has the following two pipelines.

- `security-lib-ci` - This pipeline runs first.

  yml

  ```yml
  # security-lib-ci YAML pipeline
  steps:
  - bash: echo "The security-lib-ci pipeline runs first"
  ```

- `app-ci` - This pipeline has a pipeline resource trigger that configures the `app-ci` pipeline to run automatically every time a run of the `security-lib-ci` pipeline completes.

  YAML

  ```yaml
  # app-ci YAML pipeline
  # We are setting up a pipeline resource that references the security-
  lib-ci
  # pipeline and setting up a pipeline completion trigger so that our
  app-ci
  # pipeline runs when a run of the security-lib-ci pipeline completes
  resources:
    pipelines:
    - pipeline: securitylib # Name of the pipeline resource.
      source: security-lib-ci # The name of the pipeline referenced by
  this pipeline resource.
      project: FabrikamProject # Required only if the source pipeline is
  in another project
      trigger: true # Run app-ci pipeline when any run of security-lib-ci
  completes

  steps:
  - bash: echo "app-ci runs after security-lib-ci completes"
  ```

- `- pipeline: securitylib` specifies the name of the pipeline resource. Use the label defined here when referring to the pipeline resource from other parts of the pipeline, such as when using pipeline resource variables or downloading artifacts.
- `source: security-lib-ci` specifies the name of the pipeline referenced by this pipeline resource. You can retrieve a pipeline's name from the Azure DevOps portal in several places, such as the Pipelines landing page. By default, pipelines are named after the repository that contains the pipeline. To update a pipeline's name, see Pipeline settings. If the pipeline is contained in a folder, include the folder name, including the leading `\`, for example `\security pipelines\security-lib-ci`.
- `project: FabrikamProject` - If the triggering pipeline is in another Azure DevOps project, you must specify the project name. This property is optional if both the source pipeline and the triggered pipeline are in the same project. If you specify this value and your pipeline doesn't trigger, see the note at the end of this section.
- `trigger: true` - Use this syntax to trigger the pipeline when any version of the source pipeline completes. See the following sections in this article to learn how to filter which versions of the source pipeline completing will trigger a run. When filters are specified, the source pipeline run must match all of the filters to trigger a run.

If the triggering pipeline and the triggered pipeline use the same repository, both pipelines will run using the same commit when one triggers the other. This is helpful if your first pipeline builds the code and the second pipeline tests it. However, if the two pipelines use different repositories, the triggered pipeline will use the version of the code in the branch specified by the `Default branch for manual and scheduled builds` setting, as described in Branch considerations for pipeline completion triggers.

> **Note**
>
> In some scenarios, the default branch for manual builds and scheduled builds doesn't include a `refs/heads` prefix. For example, the default branch might be set to `main` instead of to `refs/heads/main`. In this scenario, *a trigger from a different project doesn't work*. If you encounter issues when you set `project` to a value other than the target pipeline's, you can update the default branch to include `refs/heads` by changing its value to a different branch, and then by changing it back to the default branch you want to use.

Configuring pipeline completion triggers is not supported in YAML templates. You can still define pipeline resources in templates.

Branch filters

You can optionally specify the branches to include or exclude when configuring the trigger. If you specify branch filters, a new pipeline is triggered whenever a source pipeline run is successfully completed that matches the branch filters. In the following example, the `app-ci` pipeline runs if the `security-lib-ci` completes on any `releases/*` branch, except for `releases/old*`.

YAML

```yaml
# app-ci YAML pipeline
resources:
  pipelines:
  - pipeline: securitylib
    source: security-lib-ci
    trigger:
      branches:
        include:
        - releases/*
        exclude:
        - releases/old*
```

> **Note**
>
> If your branch filters aren't working, try using the prefix `refs/heads/`. For example, use `refs/heads/releases/old*` instead of `releases/old*`.

Tag filters

The `tags` property of the `trigger` filters which pipeline completion events can trigger your pipeline. If the triggering pipeline matches all of the tags in the `tags` list, the pipeline runs.

yml

```yml
resources:
  pipelines:
  - pipeline: MyCIAlias
    source: Farbrikam-CI
    trigger:
      tags:        # This filter is used for triggering the pipeline run
      - Production # Tags are AND'ed
      - Signed
```

> **Note**
>
> The pipeline resource also has a `tags` property. The `tags` property of the pipeline resource is used to determine which pipeline run to retrieve artifacts from, when the pipeline is triggered manually or by a scheduled trigger. For more information, see **Resources: pipelines** and **Evaluation of artifact version**.

Stage filters

You can trigger your pipeline when one or more stages of the triggering pipeline complete by using the `stages` filter. If you provide multiple stages, the triggered pipeline runs when all of the listed stages complete.

yml

```yml
resources:
  pipelines:
  - pipeline: MyCIAlias
    source: Farbrikam-CI
```

```
    trigger:
      stages:         # This stage filter is used when evaluating conditions
for
      - PreProduction # triggering your pipeline. On successful completion
of all the stages
      - Production    # provided, your pipeline will be triggered.
```

Branch considerations

Pipeline completion triggers use the Default branch for manual and scheduled builds setting to determine which branch's version of a YAML pipeline's branch filters to evaluate when determining whether to run a pipeline as the result of another pipeline completing. By default this setting points to the default branch of the repository.

When a pipeline completes, the Azure DevOps runtime evaluates the pipeline resource trigger branch filters of any pipelines with pipeline completion triggers that reference the completed pipeline. A pipeline can have multiple versions in different branches, so the runtime evaluates the branch filters in the pipeline version in the branch specified by the `Default branch for manual and scheduled builds` setting. If there is a match, the pipeline runs, but the version of the pipeline that runs may be in a different branch depending on whether the triggered pipeline is in the same repository as the completed pipeline.

- If the two pipelines are in different repositories, the triggered pipeline version in the branch specified by `Default branch for manual and scheduled builds` is run.
- If the two pipelines are in the same repository, the triggered pipeline version in the same branch as the triggering pipeline is run (using the version of the pipeline from that branch at the time that the trigger condition is met), even if that branch is different than the `Default branch for manual and scheduled builds`, and even if that version does not have branch filters that match the completed pipeline's branch. This is because the branch filters from the `Default branch for manual and scheduled builds` branch are used to determine if the pipeline should run, and not the branch filters in the version that is in the completed pipeline branch.

If your pipeline completion triggers don't seem to be firing, check the value of the Default branch for manual and scheduled builds setting for the triggered pipeline. The branch filters in that branch's version of the pipeline are used to determine whether the pipeline completion trigger initiates a run of the pipeline. By default, `Default branch for manual and scheduled builds` is set to the default branch of the repository, but you can change it after the pipeline is created.

A typical scenario in which the pipeline completion trigger doesn't fire is when a new branch is created, the pipeline completion trigger branch filters are modified to include this new branch, but when the first pipeline completes on a branch that matches the new branch filters, the second pipeline doesn't trigger. This happens if the branch filters in the pipeline version in the `Default branch for manual and scheduled builds` branch don't match the new branch. To resolve this trigger issue you have the following two options.

- Update the branch filters in the pipeline in the `Default branch for manual and scheduled builds` branch so that they match the new branch.
- Update the Default branch for manual and scheduled builds setting to a branch that has a version of the pipeline with the branch filters that match the new branch.

Combining trigger types

When you specify both CI triggers and pipeline triggers in your pipeline, you can expect new runs to be started every time a push is made that matches the filters of the CI trigger, and a run of the source pipeline is completed that matches the filters of the pipeline completion trigger.

For example, consider two pipelines named `A` and `B` that are in the same repository, both have CI triggers, and `B` has a pipeline completion trigger configured for the completion of pipeline `A`. If you make a push to the repository:

- A new run of `A` is started, based on its CI trigger.
- At the same time, a new run of `B` is started, based on its CI trigger. This run consumes the artifacts from a previous run of pipeline `A`.
- When `A` completes, it triggers another run of `B`, based on the pipeline completion trigger in `B`.

To prevent triggering two runs of `B` in this example, you must disable its CI trigger (`trigger: none`) or pipeline trigger (`pr: none`).

Trigger one pipeline after another (classic)

Article • 01/26/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Large products have several components that are dependent on each other. These components are often independently built. When an upstream component (a library, for example) changes, the downstream dependencies have to be rebuilt and revalidated.

In situations like these, add a pipeline trigger to run your pipeline upon the successful completion of the **triggering pipeline**.

Add a build completion trigger

In the classic editor, pipeline triggers are called **build completion triggers**. You can select any other build in the same project to be the triggering pipeline.

After you add a **build completion** trigger, select the **triggering build**. If the triggering build is sourced from a Git repo, you can also specify **branch filters**. If you want to use wildcard characters, then type the branch specification (for example, `features/modules/*`) and then press Enter.

> ⓘ **Note**
>
> Keep in mind that in some cases, a single **multi-job build** could meet your needs. However, a build completion trigger is useful if your requirements include different configuration settings, options, or a different team to own the dependent pipeline.

Download artifacts from the triggering build

In many cases, you'll want to download artifacts from the triggering build. To do this:

1. Edit your build pipeline.
2. Add the **Download Build Artifacts** task to one of your jobs under **Tasks**.
3. For **Download artifacts produced by**, select **Specific build**.
4. Select the team **Project** that contains the triggering build pipeline.

5. Select the triggering **Build pipeline**.

6. Select **When appropriate, download artifacts from the triggering build**.

7. Even though you specified that you want to download artifacts from the triggering build, you must still select a value for **Build**. The option you choose here determines which build will be the source of the artifacts whenever your triggered build is run because of any other reason than `BuildCompletion` (e.g. `Manual`, `IndividualCI`, `Schedule`, and so on).

8. Specify the **Artifact name** and make sure it matches the name of the artifact published by the triggering build.

9. Specify the **Destination directory** to which you want to download the artifacts. For example: `$(Build.BinariesDirectory)`

Release triggers

Article • 12/15/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> **Note**
>
> This topic covers classic release pipelines. To understand triggers in YAML pipelines, see **pipeline triggers**.

Release triggers are an automation tool to deploy your application. When the trigger conditions are met, the pipeline will deploy your artifacts to the environment/stages you already specified.

Continuous deployment triggers

Continuous deployment triggers allow you to create a release every time a new build artifact is available. Using the build branch filters you can trigger deployment for a specific target branch. A release will be triggered only if the Git push contains a commit on the specified branch. For example, selecting `main` will trigger a release for a Git push which contains one or more commits to the main branch. To trigger a release for any commit to branches under `features/`, enter `features/*`. To trigger a release for commits to all branches, enter `*`. Note that all specified filters will be OR'ed.

> **Note**
>
> Automatically creating a release does not mean it will be automatically deployed to a stage. You must set up triggers to deploy your app to the various stages.

Scheduled release triggers

Scheduled release triggers allow you to create new releases at specific times.

Select the schedule icon under the **Artifacts** section. Toggle the Enabled/Disabled button and specify your release schedule. You can set up multiple schedules to trigger a release.

Pull request triggers

If you chose to enable the pull-request triggers, a release will be created every time a selected artifact is available as part of a pull request workflow.

To use a pull request trigger, you must also enable it for specific stages. We will go through stage triggers in the next section. You may also want to set up a branch policies for your branches.

You can also use **Build tags** to organize your workflow and tag specific runs. The following pull request trigger will create a release every time a new artifact version is available as part of a pull request to the *main* branch with the tags *Migration* and *Deployment*.

Stage triggers

Stage triggers allow you set up specific conditions to trigger deployment to a specific stage.

- **Select trigger**: Set the trigger that will start the deployment to your stage automatically. Use the **Stages** dropdown to trigger a release after a successful deployment to the selected stage. Select **Manual only** to only allow manual trigger.

- **Artifacts filter**: Enable the toggle button to trigger a new deployment based on specific artifacts. In this example, a release will be deployed when a new artifact is available from the specified branch.

- **Schedule**: Trigger a new deployment to your specified stage at a specific time.

- **Pull-request deployment**: Enable the toggle button to trigger a new release every time a new pull request is created. It's recommended to disable this feature for production environment.

- **Pre-deployment approvals**: Select the users who can approve or reject deployments to your selected stage. By default, when this feature is enabled, all project users must approve the deployment. If a group is added to the approvers list, at least one user in the group must approve the deployment. You can also specify the *Approval policies* and *Timeout* (the maximum time for an approval to remain in pending state before it is automatically rejected).

- **Gates**: Enable the toggle button to set up specific gates to evaluate before trigger deployment.

- **Deployment queue settings**: Configure specific actions when multiple releases are queued for deployment.

Template usage reference

Article • 07/03/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Templates let you define reusable content, logic, and parameters in YAML pipelines. To work with templates effectively, you'll need to have a basic understanding of Azure Pipelines key concepts such as stages, steps, and jobs.

Templates can help you speed up development. For example, you can have a series of the same tasks in a template and then include the template multiple times in different stages of your YAML pipeline.

Templates can also help you secure your pipeline. When a template controls what is allowed in a pipeline, the template defines logic that another file must follow. For example, you may want to restrict what tasks are allowed to run. For that scenario, you can use template to prevent someone from successfully running a task that violates your organization's security policies.

There are two types of templates: includes and extends.

- **Includes templates** let you insert reusable content with a template. If a template is used to include content, it functions like an include directive in many programming languages. Content from one file is inserted into another file.
- **Extends template** control what is allowed in a pipeline. When an extends template controls what is allowed in a pipeline, the template defines logic that another file must follow.

To take full advantage of templates, you should also use template expressions and template parameters.

Insert a template

You can copy content from one YAML and reuse it in a different YAML. Copying content from one YAML to another saves you from having to manually include the same logic in multiple places. The `include-npm-steps.yml` file template contains steps that are reused in `azure-pipelines.yml`.

> ⓘ **Note**

Template files need to exist on your filesystem at the start of a pipeline run. You can't reference templates in an artifact.

YAML

```yaml
# File: templates/include-npm-steps.yml

steps:
- script: npm install
- script: yarn install
- script: npm run compile
```

YAML

```yaml
# File: azure-pipelines.yml

jobs:
- job: Linux
  pool:
    vmImage: 'ubuntu-latest'
  steps:
  - template: templates/include-npm-steps.yml  # Template reference
- job: Windows
  pool:
    vmImage: 'windows-latest'
  steps:
  - template: templates/include-npm-steps.yml  # Template reference
```

Step reuse

You can insert a template to reuse one or more steps across several jobs. In addition to the steps from the template, each job can define more steps.

YAML

```yaml
# File: templates/npm-steps.yml
steps:
- script: npm install
- script: npm test
```

YAML

```yaml
# File: azure-pipelines.yml

jobs:
- job: Linux
  pool:
```

```yaml
    vmImage: 'ubuntu-latest'
  steps:
  - template: templates/npm-steps.yml  # Template reference

- job: macOS
  pool:
    vmImage: 'macOS-latest'
  steps:
  - template: templates/npm-steps.yml  # Template reference

- job: Windows
  pool:
    vmImage: 'windows-latest'
  steps:
  - script: echo This script runs before the template's steps, only on
Windows.
  - template: templates/npm-steps.yml  # Template reference
  - script: echo This step runs after the template's steps.
```

Job reuse

Much like steps, jobs can be reused with templates.

YAML

```yaml
# File: templates/jobs.yml
jobs:
- job: Ubuntu
  pool:
    vmImage: 'ubuntu-latest'
  steps:
  - bash: echo "Hello Ubuntu"

- job: Windows
  pool:
    vmImage: 'windows-latest'
  steps:
  - bash: echo "Hello Windows"
```

YAML

```yaml
# File: azure-pipelines.yml

jobs:
- template: templates/jobs.yml  # Template reference
```

When working with multiple jobs, remember to remove the name of the job in the template file, so as to avoid conflict

YAML

```yaml
# File: templates/jobs.yml
jobs:
- job:
  pool:
    vmImage: 'ubuntu-latest'
  steps:
  - bash: echo "Hello Ubuntu"

- job:
  pool:
    vmImage: 'windows-latest'
  steps:
  - bash: echo "Hello Windows"
```

YAML

```yaml
# File: azure-pipelines.yml

jobs:
- template: templates/jobs.yml  # Template reference
- template: templates/jobs.yml  # Template reference
- template: templates/jobs.yml  # Template reference
```

Stage reuse

Stages can also be reused with templates.

YAML

```yaml
# File: templates/stages1.yml
stages:
- stage: Angular
  jobs:
  - job: angularinstall
    steps:
    - script: npm install angular
```

YAML

```yaml
# File: templates/stages2.yml
stages:
- stage: Build
  jobs:
  - job: build
    steps:
    - script: npm run build
```

YAML

```yaml
# File: azure-pipelines.yml
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

stages:
- stage: Install
  jobs:
  - job: npminstall
    steps:
    - task: Npm@1
      inputs:
        command: 'install'
- template: templates/stages1.yml
- template: templates/stages2.yml
```

Job, stage, and step templates with parameters

YAML

```yaml
# File: templates/npm-with-params.yml

parameters:
- name: name  # defaults for any parameters that aren't specified
  default: ''
- name: vmImage
  default: ''

jobs:
- job: ${{ parameters.name }}
  pool:
    vmImage: ${{ parameters.vmImage }}
  steps:
  - script: npm install
  - script: npm test
```

When you consume the template in your pipeline, specify values for the template parameters.

YAML

```yaml
# File: azure-pipelines.yml

jobs:
- template: templates/npm-with-params.yml  # Template reference
  parameters:
```

```
    name: Linux
    vmImage: 'ubuntu-latest'

- template: templates/npm-with-params.yml  # Template reference
  parameters:
    name: macOS
    vmImage: 'macOS-latest'

- template: templates/npm-with-params.yml  # Template reference
  parameters:
    name: Windows
    vmImage: 'windows-latest'
```

You can also use parameters with step or stage templates. For example, steps with parameters:

YAML

```
# File: templates/steps-with-params.yml

parameters:
- name: 'runExtendedTests'  # defaults for any parameters that aren't
specified
  type: boolean
  default: false

steps:
- script: npm test
- ${{ if eq(parameters.runExtendedTests, true) }}:
  - script: npm test --extended
```

When you consume the template in your pipeline, specify values for the template parameters.

YAML

```
# File: azure-pipelines.yml

steps:
- script: npm install

- template: templates/steps-with-params.yml  # Template reference
  parameters:
    runExtendedTests: 'true'
```

Note

Scalar parameters without a specified type are treated as strings. For example, `eq(true, parameters['myparam'])` will return `true`, even if the `myparam` parameter is the word `false`, if `myparam` is not explicitly made `boolean`. Non-empty strings are cast to `true` in a Boolean context. That **expression** could be rewritten to explicitly compare strings: `eq(parameters['myparam'], 'true')`.

Parameters aren't limited to scalar strings. See the list of data types. For example, using the `object` type:

YAML

```yaml
# azure-pipelines.yml
jobs:
- template: process.yml
  parameters:
    pool:   # this parameter is called `pool`
      vmImage: ubuntu-latest  # and it's a mapping rather than a string


# process.yml
parameters:
- name: 'pool'
  type: object
  default: {}

jobs:
- job: build
  pool: ${{ parameters.pool }}
```

Variable reuse

Variables can be defined in one YAML and included in another template. This could be useful if you want to store all of your variables in one file. If you're using a template to include variables in a pipeline, the included template can only be used to define variables. You can use steps and more complex logic when you're extending from a template. Use parameters instead of variables when you want to restrict type.

In this example, the variable `favoriteVeggie` is included in `azure-pipelines.yml`.

YAML

```yaml
# File: vars.yml
variables:
  favoriteVeggie: 'brussels sprouts'
```

YAML

```yaml
# File: azure-pipelines.yml

variables:
- template: vars.yml  # Template reference

steps:
- script: echo My favorite vegetable is ${{ variables.favoriteVeggie }}.
```

Variable templates with parameter

You can pass parameters to variables with templates. In this example, you're passing the `DIRECTORY` parameter to a `RELEASE_COMMAND` variable.

YAML

```yaml
# File: templates/package-release-with-params.yml

parameters:
- name: DIRECTORY
  type: string
  default: "." # defaults for any parameters that specified with "."
(current directory)

variables:
- name: RELEASE_COMMAND
  value: grep version ${{ parameters.DIRECTORY }}/package.json | awk -F \"
'{print $4}'
```

When you consume the template in your pipeline, specify values for the template parameters.

YAML

```yaml
# File: azure-pipelines.yml

variables: # Global variables
  - template: package-release-with-params.yml # Template reference
    parameters:
      DIRECTORY: "azure/checker"

pool:
  vmImage: 'ubuntu-latest'

stages:
- stage: Release_Stage
  displayName: Release Version
  variables: # Stage variables
```

```
  - template: package-release-with-params.yml  # Template reference
    parameters:
      DIRECTORY: "azure/todo-list"
  jobs:
  - job: A
    steps:
    - bash: $(RELEASE_COMMAND) #output release command
```

Reference template paths

Template paths should be relative to the file that does the including. Here's an example nested hierarchy.

```
|
+-- fileA.yml
|
+-- dir1/
     |
     +-- fileB.yml
     |
     +-- dir2/
          |
          +-- fileC.yml
```

Then, in `fileA.yml` you can reference `fileB.yml` and `fileC.yml` like this.

YAML

```yaml
steps:
- template: dir1/fileB.yml
- template: dir1/dir2/fileC.yml
```

If `fileC.yml` is your starting point, you can include `fileA.yml` and `fileB.yml` like this.

YAML

```yaml
steps:
- template: ../../fileA.yml
- template: ../fileB.yml
```

When `fileB.yml` is your starting point, you can include `fileA.yml` and `fileC.yml` like this.

YAML

```yaml
steps:
- template: ../fileA.yml
- template: dir2/fileC.yml
```

Use other repositories

You can keep your templates in other repositories. For example, suppose you have a core pipeline that you want all of your app pipelines to use. You can put the template in a core repo and then refer to it from each of your app repos:

YAML

```yaml
# Repo: Contoso/BuildTemplates
# File: common.yml
parameters:
- name: 'vmImage'
  default: 'ubuntu 16.04'
  type: string

jobs:
- job: Build
  pool:
    vmImage: ${{ parameters.vmImage }}
  steps:
  - script: npm install
  - script: npm test
```

Now you can reuse this template in multiple pipelines. Use the `resources` specification to provide the location of the core repo. When you refer to the core repo, use `@` and the name you gave it in `resources`.

YAML

```yaml
# Repo: Contoso/LinuxProduct
# File: azure-pipelines.yml
resources:
  repositories:
    - repository: templates
      type: github
      name: Contoso/BuildTemplates

jobs:
- template: common.yml@templates  # Template reference
```

YAML

```yaml
# Repo: Contoso/WindowsProduct
# File: azure-pipelines.yml
resources:
  repositories:
    - repository: templates
      type: github
      name: Contoso/BuildTemplates
      ref: refs/tags/v1.0 # optional ref to pin to

jobs:
- template: common.yml@templates  # Template reference
  parameters:
    vmImage: 'windows-latest'
```

For `type: github`, `name` is `<identity>/<repo>` as in the examples above. For `type: git` (Azure Repos), `name` is `<project>/<repo>`. If that project is in a separate Azure DevOps organization, you'll need to configure a service connection of type `Azure Repos/Team Foundation Server` with access to the project and include that in YAML:

YAML

```yaml
resources:
  repositories:
  - repository: templates
    name: Contoso/BuildTemplates
    endpoint: myServiceConnection # Azure DevOps service connection
jobs:
- template: common.yml@templates
```

Repositories are resolved only once, when the pipeline starts up. After that, the same resource is used for the duration of the pipeline. Only the template files are used. Once the templates are fully expanded, the final pipeline runs as if it were defined entirely in the source repo. This means that you can't use scripts from the template repo in your pipeline.

If you want to use a particular, fixed version of the template, be sure to pin to a `ref`. The `refs` are either branches (`refs/heads/<name>`) or tags (`refs/tags/<name>`). If you want to pin a specific commit, first create a tag pointing to that commit, then pin to that tag.

> **Note**
>
> If no `ref` is specified, the pipeline will default to using `refs/heads/main`.

You may also use `@self` to refer to the repository where the original pipeline was found. This is convenient for use in `extends` templates if you want to refer back to contents in the extending pipeline's repository. For example:

YAML

```yaml
# Repo: Contoso/Central
# File: template.yml
jobs:
- job: PreBuild
  steps: []

  # Template reference to the repo where this template was
  # included from - consumers of the template are expected
  # to provide a "BuildJobs.yml"
- template: BuildJobs.yml@self

- job: PostBuild
  steps: []
```

YAML

```yaml
# Repo: Contoso/MyProduct
# File: azure-pipelines.yml
resources:
  repositories:
    - repository: templates
      type: git
      name: Contoso/Central

extends:
  template: template.yml@templates
```

YAML

```yaml
# Repo: Contoso/MyProduct
# File: BuildJobs.yml
jobs:
- job: Build
  steps: []
```

FAQ

How can I use variables inside of templates?

There are times when it may be useful to set parameters to values based on variables. Parameters are expanded early in processing a pipeline run so not all variables are available. To see what predefined variables are available in templates, see Use predefined variables.

In this example, the predefined variables `Build.SourceBranch` and `Build.Reason` are used in conditions in template.yml.

YAML

```yaml
# File: azure-pipelines.yml
trigger:
- main

extends:
  template: template.yml
```

YAML

```yaml
# File: template.yml
steps:
- script: echo Build.SourceBranch = $(Build.SourceBranch) # outputs
refs/heads/main
- script: echo Build.Reason = $(Build.Reason) # outputs IndividualCI
- ${{ if eq(variables['Build.SourceBranch'], 'refs/heads/main') }}:
  - script: echo I run only if Build.SourceBranch = refs/heads/main
- ${{ if eq(variables['Build.Reason'], 'IndividualCI') }}:
  - script: echo I run only if Build.Reason = IndividualCI
- script: echo I run after the conditions
```

Template parameters

Article • 07/03/2023

You can specify parameters and their data types in a template and reference those parameters in a pipeline. With templateContext, you can also pass properties to stages, steps, and jobs that are used as parameters in a template.

You can also use parameters outside of templates. You can only use literals for parameter default values.

Passing parameters

Parameters must contain a name and data type. In `azure-pipelines.yml`, when the parameter `yesNo` is set to a boolean value, the build succeeds. When `yesNo` is set to a string such as `apples`, the build fails.

YAML

```yaml
# File: simple-param.yml
parameters:
- name: yesNo # name of the parameter; required
  type: boolean # data type of the parameter; required
  default: false

steps:
- script: echo ${{ parameters.yesNo }}
```

YAML

```yaml
# File: azure-pipelines.yml
trigger:
- main

extends:
  template: simple-param.yml
  parameters:
      yesNo: false # set to a non-boolean value to have the build fail
```

Use templateContext to pass properties to templates

You can use `templateContext` to pass more properties to stages, steps, and jobs that are used as parameters in a template. Specifically, you can specify `templateContext` within the `jobList`, `deploymentList`, or `stageList` parameter data type.

You can use `templateContext` to make it easier to set up environments when processing each job. By bundling a job and its environment properties object together, `templateContext` can help you have more maintainable and easier to understand YAML.

In this example, the parameter `testSet` in `testing-template.yml` has the data type `jobList`. The template `testing-template.yml` creates a new variable `testJob` using the each keyword. The template then references the `testJob.templateContext.expectedHTTPResponseCode`, which gets set in `azure-pipeline.yml` and passed to the template.

When response code is 200, the template makes a REST request. When the response code is 500, the template outputs all of the environment variables for debugging.

`templateContext` can contain properties.

YAML

```yaml
#testing-template.yml

parameters:
- name: testSet
  type: jobList

jobs:
- ${{ each testJob in parameters.testSet }}:
  - ${{ if eq(testJob.templateContext.expectedHTTPResponseCode, 200) }}:
    - job:
      steps:
      - powershell: 'Invoke-RestMethod -Uri
https://blogs.msdn.microsoft.com/powershell/feed/ | Format-Table -Property
Title, pubDate'
      - ${{ testJob.steps }}
  - ${{ if eq(testJob.templateContext.expectedHTTPResponseCode, 500) }}:
    - job:
      steps:
      - powershell: 'Get-ChildItem -Path Env:\'
      - ${{ testJob.steps }}
```

YAML

```yaml
#azure-pipeline.yml

trigger: none
```

```yaml
pool:
  vmImage: ubuntu-latest

extends:
  template: testing-template.yml
  parameters:
    testSet:
    - job: positive_test
      templateContext:
        expectedHTTPResponseCode: 200
      steps:
      - script: echo "Run positive test"
    - job: negative_test
      templateContext:
        expectedHTTPResponseCode: 500
      steps:
      - script: echo "Run negative test"
```

Parameters to select a template at runtime

You can call different templates from a pipeline YAML depending on a condition. In this example, the `experimental.yml` YAML runs when the parameter `experimentalTemplate` is true.

yml

```yaml
#azure-pipeline.yml
parameters:
- name: experimentalTemplate
  displayName: 'Use experimental build process?'
  type: boolean
  default: false

steps:
- ${{ if eq(parameters.experimentalTemplate, true) }}:
  - template: experimental.yml
- ${{ if not(eq(parameters.experimentalTemplate, true)) }}:
  - template: stable.yml
```

Parameter data types

Data type	Notes
`string`	string
`number`	may be restricted to `values:`, otherwise any number-like string is accepted
`boolean`	`true` or `false`

Data type	Notes
`object`	any YAML structure
`step`	a single step
`stepList`	sequence of steps
`job`	a single job
`jobList`	sequence of jobs
`deployment`	a single deployment job
`deploymentList`	sequence of deployment jobs
`stage`	a single stage
`stageList`	sequence of stages

The step, stepList, job, jobList, deployment, deploymentList, stage, and stageList data types all use standard YAML schema format. This example includes string, number, boolean, object, step, and stepList.

YAML

```yaml
parameters:
- name: myString
  type: string
  default: a string
- name: myMultiString
  type: string
  default: default
  values:
  - default
  - ubuntu
- name: myNumber
  type: number
  default: 2
  values:
  - 1
  - 2
  - 4
  - 8
  - 16
- name: myBoolean
  type: boolean
  default: true
- name: myObject
  type: object
  default:
    foo: FOO
```

```
    bar: BAR
    things:
    - one
    - two
    - three
    nested:
      one: apple
      two: pear
      count: 3
- name: myStep
  type: step
  default:
    script: echo my step
- name: mySteplist
  type: stepList
  default:
    - script: echo step one
    - script: echo step two

trigger: none

jobs:
- job: stepList
  steps: ${{ parameters.mySteplist }}
- job: myStep
  steps:
    - ${{ parameters.myStep }}
```

You can iterate through an object and print each string in the object.

YAML

```
parameters:
- name: listOfStrings
  type: object
  default:
  - one
  - two

steps:
- ${{ each value in parameters.listOfStrings }}:
  - script: echo ${{ value }}
```

Additionally, you can iterate through nested elements within an object.

YAML

```
parameters:
- name: listOfFruits
  type: object
  default:
  - fruitName: 'apple'
```

```
    colors: ['red','green']
  - fruitName: 'lemon'
    colors: ['yellow']

steps:
- ${{ each fruit in parameters.listOfFruits }} :
  - ${{ each fruitColor in fruit.colors}} :
    - script: echo ${{ fruit.fruitName}} ${{ fruitColor }}
```

Required parameters

You can add a validation step at the beginning of your template to check for the parameters you require.

Here's an example that checks for the `solution` parameter using Bash (which enables it to work on any platform):

YAML

```
# File: steps/msbuild.yml

parameters:
- name: 'solution'
  default: ''
  type: string

steps:
- bash: |
    if [ -z "$SOLUTION" ]; then
      echo "##vso[task.logissue type=error;]Missing template parameter
\"solution\""
      echo "##vso[task.complete result=Failed;]"
    fi
  env:
    SOLUTION: ${{ parameters.solution }}
  displayName: Check for required parameters
- task: msbuild@1
  inputs:
    solution: ${{ parameters.solution }}
- task: vstest@2
  inputs:
    solution: ${{ parameters.solution }}
```

To show that the template fails if it's missing the required parameter:

YAML

```
# File: azure-pipelines.yml
```

```yaml
# This will fail since it doesn't set the "solution" parameter to anything,
# so the template will use its default of an empty string
steps:
- template: steps/msbuild.yml
```

Template expressions

Article • 07/03/2023

Use template expressions to specify how values are dynamically resolved during pipeline initialization. Wrap your template expression inside this syntax: `${{ }}`.

Template expressions can expand template parameters, and also variables. You can use parameters to influence how a template is expanded. The `parameters` object works like the variables object in an expression. Only predefined variables can be used in template expressions.

> **Note**
>
> Expressions are only expanded for `stages`, `jobs`, `steps`, and `containers` (inside `resources`). You cannot, for example, use an expression inside `trigger` or a resource like `repositories`. Additionally, on Azure DevOps 2020 RTW, you can't use template expressions inside `containers`.

For example, you define a template:

YAML

```yaml
# File: steps/msbuild.yml

parameters:
- name: 'solution'
  default: '**/*.sln'
  type: string

steps:
- task: msbuild@1
  inputs:
    solution: ${{ parameters['solution'] }}  # index syntax
- task: vstest@2
  inputs:
    solution: ${{ parameters.solution }}  # property dereference syntax
```

Then you reference the template and pass it the optional `solution` parameter:

YAML

```yaml
# File: azure-pipelines.yml

steps:
```

```
- template: steps/msbuild.yml
  parameters:
    solution: my.sln
```

Context

Within a template expression, you have access to the `parameters` context that contains the values of parameters passed in. Additionally, you have access to the `variables` context that contains all the variables specified in the YAML file plus many of the predefined variables (noted on each variable in that article). Importantly, it doesn't have runtime variables such as those stored on the pipeline or given when you start a run. Template expansion happens early in the run, so those variables aren't available.

Template expression functions

You can use general functions in your templates. You can also use a few template expression functions.

format

- Simple string token replacement
- Min parameters: 2. Max parameters: N
- Example: `${{ format('{0} Build', parameters.os) }}` → `'Windows Build'`

coalesce

- Evaluates to the first non-empty, non-null string argument
- Min parameters: 2. Max parameters: N
- Example:

YAML

```yaml
parameters:
- name: 'restoreProjects'
  default: ''
  type: string
- name: 'buildProjects'
  default: ''
  type: string

steps:
- script: echo ${{ coalesce(parameters.foo, parameters.bar, 'Nothing to
see') }}
```

Insertion

You can use template expressions to alter the structure of a YAML pipeline. For instance, to insert into a sequence:

YAML

```yaml
# File: jobs/build.yml

parameters:
- name: 'preBuild'
  type: stepList
  default: []
- name: 'preTest'
  type: stepList
  default: []
- name: 'preSign'
  type: stepList
  default: []

jobs:
- job: Build
  pool:
    vmImage: 'windows-latest'
  steps:
  - script: cred-scan
  - ${{ parameters.preBuild }}
  - task: msbuild@1
  - ${{ parameters.preTest }}
  - task: vstest@2
  - ${{ parameters.preSign }}
  - script: sign
```

YAML

```yaml
# File: .vsts.ci.yml

jobs:
- template: jobs/build.yml
  parameters:
    preBuild:
    - script: echo hello from pre-build
    preTest:
    - script: echo hello from pre-test
```

When an array is inserted into an array, the nested array is flattened.

To insert into a mapping, use the special property `${{ insert }}`.

YAML

```yaml
# Default values
parameters:
- name: 'additionalVariables'
  type: object
  default: {}

jobs:
- job: build
  variables:
    configuration: debug
    arch: x86
    ${{ insert }}: ${{ parameters.additionalVariables }}
  steps:
  - task: msbuild@1
  - task: vstest@2
```

YAML

```yaml
jobs:
- template: jobs/build.yml
  parameters:
    additionalVariables:
      TEST_SUITE: L0,L1
```

Conditional insertion

If you want to conditionally insert into a sequence or a mapping in a template, use insertions and expression evaluation. You can also use `if` statements outside of templates as long as you use template syntax.

For example, to insert into a sequence in a template:

YAML

```yaml
# File: steps/build.yml

parameters:
- name: 'toolset'
  default: msbuild
  type: string
  values:
  - msbuild
  - dotnet

steps:
# msbuild
- ${{ if eq(parameters.toolset, 'msbuild') }}:
  - task: msbuild@1
  - task: vstest@2
```

```
# dotnet
- ${{ if eq(parameters.toolset, 'dotnet') }}:
  - task: dotnet@1
    inputs:
      command: build
  - task: dotnet@1
    inputs:
      command: test
```

YAML

```yaml
# File: azure-pipelines.yml

steps:
- template: steps/build.yml
  parameters:
    toolset: dotnet
```

For example, to insert into a mapping in a template:

YAML

```yaml
# File: steps/build.yml

parameters:
- name: 'debug'
  type: boolean
  default: false

steps:
- script: tool
  env:
    ${{ if eq(parameters.debug, true) }}:
      TOOL_DEBUG: true
      TOOL_DEBUG_DIR: _dbg
```

YAML

```yaml
steps:
- template: steps/build.yml
  parameters:
    debug: true
```

You can also use conditional insertion for variables. In this example, `start` always prints and `this is a test` only prints when the `foo` variable equals `test`.

YAML

```yaml
variables:
  - name: foo
    value: test

pool:
  vmImage: 'ubuntu-latest'

steps:
- script: echo "start" # always runs
- ${{ if eq(variables.foo, 'test') }}:
  - script: echo "this is a test" # runs when foo=test
```

Iterative insertion

The `each` directive allows iterative insertion based on a YAML sequence (array) or mapping (key-value pairs).

For example, you can wrap the steps of each job with other pre- and post-steps:

YAML

```yaml
# job.yml
parameters:
- name: 'jobs'
  type: jobList
  default: []

jobs:
- ${{ each job in parameters.jobs }}: # Each job
  - ${{ each pair in job }}:          # Insert all properties other than
"steps"
      ${{ if ne(pair.key, 'steps') }}:
        ${{ pair.key }}: ${{ pair.value }}
    steps:                            # Wrap the steps
    - task: SetupMyBuildTools@1       # Pre steps
    - ${{ job.steps }}                # Users steps
    - task: PublishMyTelemetry@1      # Post steps
      condition: always()
```

YAML

```yaml
# azure-pipelines.yml
jobs:
- template: job.yml
  parameters:
    jobs:
    - job: A
      steps:
      - script: echo This will get sandwiched between SetupMyBuildTools and
PublishMyTelemetry.
```

```
    - job: B
      steps:
      - script: echo So will this!
```

You can also manipulate the properties of whatever you're iterating over. For example, to add more dependencies:

YAML

```yaml
# job.yml
parameters:
- name: 'jobs'
  type: jobList
  default: []

jobs:
- job: SomeSpecialTool                # Run your special tool in its own job
first
  steps:
  - task: RunSpecialTool@1
- ${{ each job in parameters.jobs }}: # Then do each job
  - ${{ each pair in job }}:          # Insert all properties other than
"dependsOn"
      ${{ if ne(pair.key, 'dependsOn') }}:
        ${{ pair.key }}: ${{ pair.value }}
    dependsOn:                        # Inject dependency
    - SomeSpecialTool
    - ${{ if job.dependsOn }}:
      - ${{ job.dependsOn }}
```

YAML

```yaml
# azure-pipelines.yml
jobs:
- template: job.yml
  parameters:
    jobs:
    - job: A
      steps:
      - script: echo This job depends on SomeSpecialTool, even though it's
not explicitly shown here.
    - job: B
      dependsOn:
      - A
      steps:
      - script: echo This job depends on both Job A and on SomeSpecialTool.
```

Escape a value

If you need to escape a value that literally contains `${{`, then wrap the value in an expression string. For example, `${{ 'my${{value' }}` or `${{ 'my${{value with a '' single quote too' }}`

Imposed limits

Templates and template expressions can cause explosive growth to the size and complexity of a pipeline. To help prevent runaway growth, Azure Pipelines imposes the following limits:

- No more than 100 separate YAML files may be included (directly or indirectly)
- No more than 20 levels of template nesting (templates including other templates)
- No more than 10 megabytes of memory consumed while parsing the YAML (in practice, this is typically between 600 KB - 2 MB of on-disk YAML, depending on the specific features used)

Task types & usage

Article • 04/11/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

A **task** is the building block for defining automation in a pipeline. A task is simply a packaged script or procedure that has been abstracted with a set of inputs.

When you add a task to your pipeline, it may also add a set of **demands** to the pipeline. The demands define the prerequisites that must be installed on the agent for the task to run. When you run the build or deployment, an agent that meets these demands will be chosen.

When you run a job, all the tasks are run in sequence, one after the other. To run the same set of tasks in parallel on multiple agents, or to run some tasks without using an agent, see jobs.

By default, all tasks run in the same context, whether that's on the host or in a job container. You may optionally use step targets to control context for an individual task.

Learn more about how to specify properties for a task with the built-in tasks.

Custom tasks

We provide some built-in tasks to enable fundamental build and deployment scenarios. We have also provided guidance for creating your own custom task.

In addition, Visual Studio Marketplace offers many extensions; each of which, when installed to your subscription or collection, extends the task catalog with one or more tasks. Furthermore, you can write your own custom extensions to add tasks to Azure Pipelines or TFS.

In YAML pipelines, you refer to tasks by name. If a name matches both an in-box task and a custom task, the in-box task will take precedence. You can use the task GUID or a fully qualified name for the custom task to avoid this risk:

YAML

```yaml
steps:
- task: myPublisherId.myExtensionId.myContributionId.myTaskName@1 #format
example
- task: qetza.replacetokens.replacetokens-task.replacetokens@3 #working
example
```

To find `myPublisherId` and `myExtensionId`, select **Get** on a task in the marketplace. The values after the `itemName` in your URL string are `myPublisherId` and `myExtensionId`. You can also find the fully qualified name by adding the task to a Release pipeline and selecting **View YAML** when editing the task.

Task versions

Tasks are versioned, and you must specify the major version of the task used in your pipeline. This can help to prevent issues when new versions of a task are released. Tasks are typically backwards compatible, but in some scenarios you may encounter unpredictable errors when a task is automatically updated.

When a new minor version is released (for example, 1.2 to 1.3), your build or release will automatically use the new version. However, if a new major version is released (for example 2.0), your build or release will continue to use the major version you specified until you edit the pipeline and manually change to the new major version. The build or release log will include an alert that a new major version is available.

You can set which minor version gets used by specifying the full version number of a task after the `@` sign (example: `GoTool@0.3.1`). You can only use task versions that exist for your organization.

YAML

In YAML, you specify the major version using `@` in the task name. For example, to pin to version 2 of the `PublishTestResults` task:

YAML

```yaml
steps:
- task: PublishTestResults@2
```

Task control options

Each task offers you some **Control Options**.

YAML

Control options are available as keys on the `task` section.

YAML

```yaml
- task: string  # reference to a task and version, e.g. "VSBuild@1"
  condition: expression     # see below
  continueOnError: boolean  # 'true' if future steps should run even if
this step fails; defaults to 'false'
  enabled: boolean          # whether or not to run this step; defaults
to 'true'
  retryCountOnTaskFailure: number # Max number of retries; default is
zero
  timeoutInMinutes: string  # how long to wait before timing out the
task
  target: string            # 'host' or the name of a container resource
to target
```

> **Note**
>
> A given task or job can't unilaterally decide whether the job/stage continues. What it can do is offer a status of **succeeded** or **failed**, and downstream tasks/jobs each have a condition computation that lets them decide whether to run or not. The default condition which is effectively "run if we're in a successful state".
>
> **Continue on error** alters this in a subtle way. It effectively "tricks" all downstream steps/jobs into treating any result as "success" for the purposes of making that decision. Or to put it another way, it says "don't consider the failure of this task when you're making a decision about the condition of the containing structure".

The timeout period begins when the task starts running. It doesn't include the time the task is queued or is waiting for an agent.

> **Note**
>
> Pipelines may have a job level timeout specified in addition to a task level timeout. If the job level timeout interval elapses before your step completes, the running job is terminated, even if the step is configured with a longer timeout interval. For more information, see **Timeouts**.

In this YAML, `PublishTestResults@2` will run even if the previous step fails because of the succeededOrFailed() condition.

```yaml
steps:
- task: UsePythonVersion@0
  inputs:
    versionSpec: '3.x'
    architecture: 'x64'
- task: PublishTestResults@2
  inputs:
    testResultsFiles: "**/TEST-*.xml"
  condition: succeededOrFailed()
```

Conditions

- Only when all previous direct and indirect dependencies with the same agent pool have succeeded. If you have different agent pools, those stages or jobs will run concurrently. This is the default if there is not a condition set in the YAML.
- Even if a previous dependency has failed, unless the run was canceled. Use `succeededOrFailed()` in the YAML for this condition.
- Even if a previous dependency has failed, even if the run was canceled. Use `always()` in the YAML for this condition.
- Only when a previous dependency has failed. Use `failed()` in the YAML for this condition.
- Custom conditions, which are composed of expressions

Step target

Tasks run in an execution context, which is either the agent host or a container. An individual step may override its context by specifying a `target`. Available options are the word `host` to target the agent host plus any containers defined in the pipeline. For example:

```yaml
resources:
  containers:
  - container: pycontainer
```

```yaml
    image: python:3.11

steps:
- task: SampleTask@1
  target: host
- task: AnotherTask@1
  target: pycontainer
```

Here, the `SampleTask` runs on the host and `AnotherTask` runs in a container.

Number of retries if task failed

Use `retryCountOnTaskFailure` to specify the number of retries if the task fails. The default is zero.

```yml
- task: <name of task>
   retryCountOnTaskFailure: <max number of retries>
   ...
```

Note

- Requires agent version 2.194.0 or later. Not supported for **agentless tasks**.
- The failing task retries in seconds. The wait time between each retry increases after each failed attempt.
- There is no assumption about the idempotency of the task. If the task has side-effects (for instance, if it created an external resource partially), then it may fail the second time it is run.
- There is no information about the retry count made available to the task.
- A warning is added to the task logs indicating that it has failed before it is retried.
- All of the attempts to retry a task are shown in the UI as part of the same task node.

Environment variables

YAML

Each task has an `env` property that is a list of string pairs that represent environment variables mapped into the task process.

```yml
task: AzureCLI@2
displayName: Azure CLI
inputs: # Specific to each task
env:
  ENV_VARIABLE_NAME: value
  ENV_VARIABLE_NAME2: value
  ...
```

The following example runs the `script` step, which is a shortcut for the Command line task, followed by the equivalent task syntax. This example assigns a value to the `AZURE_DEVOPS_EXT_PAT` environment variable, which is used to authenticating with Azure DevOps CLI.

```yml
# Using the script shortcut syntax
- script: az pipelines variable-group list --output table
  env:
    AZURE_DEVOPS_EXT_PAT: $(System.AccessToken)
  displayName: 'List variable groups using the script step'

# Using the task syntax
- task: CmdLine@2
  inputs:
    script: az pipelines variable-group list --output table
  env:
    AZURE_DEVOPS_EXT_PAT: $(System.AccessToken)
  displayName: 'List variable groups using the command line task'
```

Build tool installers (Azure Pipelines)

Tool installers enable your build pipeline to install and control your dependencies. Specifically, you can:

- Install a tool or runtime on the fly (even on Microsoft-hosted agents) just in time for your CI build.

- Validate your app or library against multiple versions of a dependency such as Node.js.

For example, you can set up your build pipeline to run and validate your app for multiple versions of Node.js.

Example: Test and validate your app on multiple versions of Node.js

YAML

Create an azure-pipelines.yml file in your project's base directory with the following contents.

YAML

```yaml
pool:
  vmImage: ubuntu-latest

steps:
# Node install
- task: NodeTool@0
  displayName: Node install
  inputs:
    versionSpec: '12.x' # The version we're installing
# Write the installed version to the command line
- script: which node
```

Create a new build pipeline and run it. Observe how the build is run. The Node.js Tool Installer downloads the Node.js version if it isn't already on the agent. The Command Line script logs the location of the Node.js version on disk.

Tool installer tasks

For a list of our tool installer tasks, see Tool installer tasks.

Disabling in-box and Marketplace tasks

On the organization settings page, you can disable Marketplace tasks, in-box tasks, or both. Disabling Marketplace tasks can help increase security of your pipelines. If you disable both in-box and Marketplace tasks, only tasks you install using tfx will be available.

Related articles

- Jobs
- Task groups
- Built-in task catalog

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and feel free to post your questions, search for answers, or suggest a feature on our Azure DevOps Developer Community. Support page.

Task groups for builds and releases (classic)

Article • 04/20/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> **Note**
>
> Task groups are not supported in YAML pipelines. Instead, in that case you can use templates. See **YAML schema reference**.

A *task group* allows you to encapsulate a sequence of tasks, already defined in a build or a release pipeline, into a single reusable task that can be added to a build or release pipeline, just like any other task. You can choose to extract the parameters from the encapsulated tasks as configuration variables, and abstract the rest of the task information.

The new task group is automatically added to the task catalog, ready to be added to other release and build pipelines. Task groups are stored at the project level, and are not accessible outside the project scope.

Task groups are a way to standardize and centrally manage deployment steps for all your applications. When you include a task group in your definitions, and then make a change centrally to the task group, the change is automatically reflected in all the definitions that use the task group. There is no need to change each one individually.

Before you create a task group...

- Ensure that all of the tasks you want to include in a task group have their parameters defined as variables, such as **$(MyVariable)**, where you want to be able to configure these parameters when you use the task group. Variables used in the tasks are automatically extracted and converted into parameters for the task group. Values of these configuration variables will be converted into default values for the task group.

- If you specify a value (instead of a variable) for a parameter, that value becomes a fixed parameter value and cannot be exposed as a parameter to the task group.

- Parameters of the encapsulated tasks for which you specified a value (instead of a variable), or you didn't provide a value for, are not configurable in the task group when added to a build or release pipeline.
- Task conditions (such as "Run this task only when a previous task has failed" for a **PowerShell Script** task) can be configured in a task group and these settings are persisted with the task group.
- When you save the task group, you can provide a name and a description for the new task group, and select a category where you want it to appear in the **Task catalog** dialog. You can also change the default values for each of the parameters.
- When you queue a build or a release, the encapsulated tasks are extracted and the values you entered for the task group parameters are applied to the tasks.
- Changes you make to a task group are reflected in every instance of the task group.

Create a task group

1. 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.

2. Select a sequence of tasks in a build or release pipeline, open the shortcut menu, and then choose **Create task group**.

3. Specify a name and description for the new task group, and the category (tab in the Add tasks panel) you want to add it to.

4. After you choose **Create**, the new task group is created and replaces the selected tasks in your pipeline.

5. All the '$(vars)' from the underlying tasks, excluding the predefined variables, will surface as the mandatory parameters for the newly created task group.

   For example, let's say you have a task input $(foobar), which you don't intend to parameterize. However, when you create a task group, the task input is converted into task group parameter 'foobar'. Now, you can provide the default value for the task group parameter 'foobar' as $(foobar). This ensures that at runtime, the expanded task gets the same input it's intended to.

6. Save your updated pipeline.

Manage task groups

All the task groups you create in the current project are listed in the **Task Groups** page of **Azure Pipelines**.

Use the **Export** shortcut command to save a copy of the task group as a JSON pipeline, and the **Import** icon to import previously saved task group definitions. Use this feature to transfer task groups between projects and enterprises, or replicate and save copies of your task groups.

Select a task group name to open the details page.

- In the **Tasks** page you can edit the tasks that make up the task group. For each encapsulated task you can change the parameter values for the non-variable parameters, edit the existing parameter variables, or convert parameter values to and from variables. When you save the changes, all definitions that use this task group will pick up the changes.

All the variable parameters of the task group will show up as mandatory parameters in the pipeline definition. You can also set the default value for the task group parameters.

- In the **History** tab you can see the history of changes to the group.
- In the **References** tab you can expand lists of all the build and release pipelines, and other task groups, that use (reference) this task group. This is useful to ensure changes do not have unexpected effects on other processes.

Create previews and updated versions of task groups

All of the built-in tasks in Azure Pipelines and TFS are versioned. This allows build and release pipelines to continue to use the existing version of a task while new versions are developed, tested, and released. In Azure Pipelines, you can version your own custom task groups so that they behave in the same way and provide the same advantages.

1. After you finish editing a task group, choose **Save as draft** instead of **Save**.

2. The string **-test** is appended to the task group version number. When you are happy with the changes, choose **Publish draft**. You can choose whether to publish it as a preview or as a production-ready version.

3. You can now use the updated task group in your build and release processes; either by changing the version number of the task group in an existing pipeline or by adding it from the **Add tasks** panel.

> As with the built-in tasks, the default when you add a task group is the highest non-preview version.

4. After you have finished testing the updated task group, choose **Publish preview**. The **Preview** string is removed from the version number string. It will now appear

in definitions as a "production-ready" version.

5. In a build or release pipeline that already contains this task group, you can now select the new "production-ready" version. When you add the task group from the **Add tasks** panel, it automatically selects the new "production-ready" version.

Working with task group versions

Any task group update can be a minor or major version update.

Minor version

Action: You directly save the task group after edit instead of saving it as draft.

Effect: The version number doesn't change. Let's say you have a task group of version `1.0`. You can have any number of minor version updates i.e. `1.1`, `1.2`, `1.3` etc. In your pipeline, the task group version shows as `1.*` The latest changes will show up in the pipeline definition automatically.

Reason: This is supposed to be a small change in the task group and you expect the pipelines to use this new change without editing the version in the pipeline definition.

Major version

Action: You save the task group as draft and then create a preview, validate the task group and then publish the preview as a major version.

Effect: The task group bumps up to a new version. Let's say you have a task group of version `1.*`. A new version gets published as `2.*`, `3.*`, `4.*` etc. And a notification about availability of new version shows up in all the pipeline definitions where this task group is used. User has to explicitly update to new version of the task group in pipelines.

Reason: When you have a substantial change which might break the existing pipelines, you would like to test it out and roll out as a new version. Users can choose to upgrade to new version or choose to stay on the same version. This functionality is same as a normal task version update.

However, if your task group update is not a breaking change but you would like to validate first and then enforce pipelines to consume the latest changes, you can follow below steps.

1. Update the task group with your desired changes and save it as a draft. A new draft task group '<Taskgroupname>-Draft' will be created which contains the changes you have done. And this draft task group is accessible for you to consume in your pipelines.
2. Now, instead of publishing as preview, you can directly consume this draft task group in your test pipeline.
3. Validate this new draft task group in your test pipeline and once you are confident, go back to your main task group and do the same changes and save it directly. This will be taken as minor version update.
4. The new changes will now show up in all the pipelines where this task group is used.
5. Now you can delete your draft task group.

Related topics

- Tasks
- Task jobs

Help and support

- See our troubleshooting page

- Get advice on Stack Overflow, and feel free to post your questions, search for answers, or suggest a feature on our Azure DevOps Developer Community. Support page.

Add a custom pipelines task extension

Article • 09/28/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Learn how to install extensions to your organization for custom build or release tasks in Azure DevOps. For more information about the new cross-platform build/release system, see What is Azure Pipelines?

> **Note**
>
> This article covers agent tasks in agent-based extensions. For information on server tasks/server-based extensions, check out the **Server Task GitHub Documentation**.

Prerequisites

To create extensions for Azure DevOps, you need the following software and tools.

Software/tool	Information
Azure DevOps organization	Create an organization.
A text editor	For many of the tutorials, we use Visual Studio Code, which provides intellisense and debugging support. Download the latest version.
Node.js	Download the latest version.
npmjs.com 4.0.2 or newer	TypeScript Compiler. Download the latest version.
tfx-cli	Package your extension with Cross-platform CLI for Azure DevOps. using `npm`, a component of Node.js, by running `npm i -g tfx-cli`.
Azure DevOps extension SDK	Install the azure-devops-extension-sdk package
A `home` directory for your project	The `home` directory of a build or release task extension should look like the following example after you complete the steps in this tutorial:

```
|--- README.md
|--- images
    |--- extension-icon.png
|--- buildandreleasetask            // where your task scripts are placed
|--- vss-extension.json             // extension's manifest
```

> **Important**
>
> The dev machine needs to run the **latest version of Node** to ensure that the written code is compatible with the production environment on the agent and the latest non-preview version of azure-pipelines-task-lib.

1. Create a custom task

Do every part of the steps withing this procedure in the `buildandreleasetask` folder.

> **Note**
>
> This example walk-through is on Windows with PowerShell. We made it generic for all platforms, but the syntax for getting environment variables is different. If you're using a Mac or Linux, replace any instances of `$env:<var>=<val>` with `export <var>=<val>`.

Create task scaffolding

1. Create the folder structure for the task and install the required libraries and dependencies.

2. Open a PowerShell command window, go to your `buildandreleasetask` folder, and run the following command.

   ```
   npm init --yes
   ```

   `npm init` creates the `package.json` file. We added the `--yes` parameter to accept all of the default `npm init` options.

   > **Tip**

The agent doesn't automatically install the required modules because it's expecting your task folder to include the node modules. To mitigate this, copy the `node_modules` to `buildandreleasetask`. As your task gets bigger, it's easy to exceed the size limit (50MB) of a VSIX file. Before you copy the node folder, you may want to run `npm install --production` or `npm prune --production`, or you can write a script to build and pack everything.

3. Add `azure-pipelines-task-lib` to your library.

```
npm install azure-pipelines-task-lib --save
```

4. Ensure that TypeScript typings are installed for external dependencies.

```
npm install @types/node --save-dev
npm install @types/q --save-dev
```

5. Create a `.gitignore` file and add node_modules to it. Your build process should do an `npm install` and a `typings install` so that node_modules are built each time and don't need to be checked in.

```
echo node_modules > .gitignore
```

6. Install Mocha as a development dependency.

```
npm install mocha --save-dev -g
npm install sync-request --save-dev
npm install @types/mocha --save-dev
```

7. Choose TypeScript version 2.3.4 or 4.6.3.

```
npm install typescript@4.6.3 -g --save-dev
```

> ⓘ **Note**
>
> Make sure that TypeScript is installed globally with npm in your development environment, so the `tsc` command is available. If you skip this step, TypeScript version 2.3.4 gets used by default, and you still have to install the package globally to have the `tsc` command available.

8. Create `tsconfig.json` compiler options. This file ensures that your TypeScript files are compiled to JavaScript files.

```
tsc --init --target es2023
```

Create custom task

Now that the scaffolding is complete, we can create our custom task.

1. Create a `task.json` file in the `buildandreleasetask` folder. The `task.json` file describes the build/release task and is what the build/release system uses to render configuration options to the user and to know which scripts to execute at build/release time.

2. Copy the following code and replace the `{{placeholders}}` with your task's information. The most important placeholder is the `taskguid`, and it must be unique.

```JSON
{
 "$schema": "https://raw.githubusercontent.com/Microsoft/azure-pipelines-task-lib/master/tasks.schema.json",
 "id": "{{taskguid}}",
 "name": "{{taskname}}",
 "friendlyName": "{{taskfriendlyname}}",
 "description": "{{taskdescription}}",
 "helpMarkDown": "",
 "category": "Utility",
 "author": "{{taskauthor}}",
 "version": {
     "Major": 0,
     "Minor": 1,
     "Patch": 0
 },
 "instanceNameFormat": "Echo $(samplestring)",
```

```
"inputs": [
    {
        "name": "samplestring",
        "type": "string",
        "label": "Sample String",
        "defaultValue": "",
        "required": true,
        "helpMarkDown": "A sample string"
    }
],
"execution": {
    "Node": {
        "target": "index.js"
    }
}
}
```

task.json components

See the following descriptions of some of the components of the `task.json` file.

Property	Description
`id`	A unique GUID for your task.
`name`	Name with no spaces.
`friendlyName`	Descriptive name (spaces allowed).
`description`	Detailed description of what your task does.
`author`	Short string describing the entity developing the build or release task, for example: "Microsoft Corporation."
`instanceNameFormat`	How the task displays within the build/release step list. You can use variable values by using **$(variablename)**.
`groups`	Describes groups that task properties may be logically grouped by in the UI.
`inputs`	Inputs to be used when your build or release task runs. This task expects an input with the name **samplestring**.
`execution`	Execution options for this task, including scripts.
`restrictions`	Restrictions being applied to the task about GitHub Codespaces commands task can call, and variables task can set. We recommend that you specify restriction mode for new tasks.

Note

- You can create an `id` with the following command in PowerShell:

  PowerShell

  ```
  (New-Guid).Guid
  ```

- For a more in-depth look into the task.json file, or to learn how to bundle multiple versions in your extension, see the **Build/release task reference**.

3. Create an `index.ts` file by using the following code as a reference. This code runs when the task gets called.

TypeScript

```
import tl = require('azure-pipelines-task-lib/task');

async function run() {
    try {
        const inputString: string | undefined = tl.getInput('samplestring',
true);
        if (inputString == 'bad') {
            tl.setResult(tl.TaskResult.Failed, 'Bad input was given');
            return;
        }
        console.log('Hello', inputString);
    }
    catch (err) {
        tl.setResult(tl.TaskResult.Failed, err.message);
    }
}

run();
```

4. Enter "tsc" from the `buildandreleasetask` folder to compile an `index.js` file from `index.ts`.

Run the task

1. Run the task with `node index.js` from PowerShell.

   In the following example, the task fails because inputs weren't supplied (`samplestring` is a required input).

```
node index.js
##vso[task.debug]agent.workFolder=undefined
##vso[task.debug]loading inputs and endpoints
##vso[task.debug]loaded 0
##vso[task.debug]task result: Failed
##vso[task.issue type=error;]Input required: samplestring
##vso[task.complete result=Failed;]Input required: samplestring
```

As a fix, we can set the `samplestring` input and run the task again.

```
$env:INPUT_SAMPLESTRING="Human"
node index.js
##vso[task.debug]agent.workFolder=undefined
##vso[task.debug]loading inputs and endpoints
##vso[task.debug]loading INPUT_SAMPLESTRING
##vso[task.debug]loaded 1
##vso[task.debug]Agent.ProxyUrl=undefined
##vso[task.debug]Agent.CAInfo=undefined
##vso[task.debug]Agent.ClientCert=undefined
##vso[task.debug]Agent.SkipCertValidation=undefined
##vso[task.debug]samplestring=Human
Hello Human
```

This time, the task succeeded because `samplestring` was supplied and it correctly outputted "Hello Human!"

> **Tip**
>
> For information about various task runners and how to include the latest node version in the task.json, see **Node runner update guidance for Azure Pipelines task authors**.

2. Unit test your task scripts

We unit test to quickly test the task script, and not the external tools that it's calling. We want to test all aspects of both success and failure paths.

1. Install test tools. We use Mocha as the test driver in this walk through.

```
npm install mocha --save-dev -g
npm install sync-request --save-dev
```

```
npm install @types/mocha --save-dev
```

2. Create a `tests` folder containing a `_suite.ts` file with the following contents:

TypeScript

```
import * as path from 'path';
import * as assert from 'assert';
import * as ttm from 'azure-pipelines-task-lib/mock-test';

describe('Sample task tests', function () {

    before( function() {

    });

    after(() => {

    });

    it('should succeed with simple inputs', function(done: Mocha.Done)
{
        // Add success test here
    });

    it('it should fail if tool returns 1', function(done: Mocha.Done) {
        // Add failure test here
    });
});
```

> **Tip**
>
> Your test folder should be located in the buildandreleasetask folder. If you get a sync-request error, you can work around it by adding sync-request to the buildandreleasetask folder with the command `npm i --save-dev sync-request`.

3. Create a `success.ts` file in your test directory with the following contents. This file creation simulates running the task and mocks all calls to outside methods.

TypeScript

```
import ma = require('azure-pipelines-task-lib/mock-answer');
import tmrm = require('azure-pipelines-task-lib/mock-run');
import path = require('path');

let taskPath = path.join(__dirname, '..', 'index.js');
let tmr: tmrm.TaskMockRunner = new tmrm.TaskMockRunner(taskPath);
```

```
tmr.setInput('samplestring', 'human');

tmr.run();
```

The success test validates that with the appropriate inputs, it succeeds with no errors or warnings and returns the correct output.

4. Add the following example success test to your `_suite.ts` file to run the task mock runner.

TypeScript

```
it('should succeed with simple inputs', function(done: Mocha.Done) {
    this.timeout(1000);

    let tp = path.join(__dirname, 'success.js');
    let tr: ttm.MockTestRunner = new ttm.MockTestRunner(tp);

    tr.run();
    console.log(tr.succeeded);
    assert.equal(tr.succeeded, true, 'should have succeeded');
    assert.equal(tr.warningIssues.length, 0, "should have no
warnings");
    assert.equal(tr.errorIssues.length, 0, "should have no errors");
    console.log(tr.stdout);
    assert.equal(tr.stdout.indexOf('Hello human') >= 0, true, "should
display Hello human");
    done();
});
```

5. Create a `failure.ts` file in your test directory as your task mock runner with the following contents:

TypeScript

```
import ma = require('azure-pipelines-task-lib/mock-answer');
import tmrm = require('azure-pipelines-task-lib/mock-run');
import path = require('path');

let taskPath = path.join(__dirname, '..', 'index.js');
let tmr: tmrm.TaskMockRunner = new tmrm.TaskMockRunner(taskPath);

tmr.setInput('samplestring', 'bad');

tmr.run();
```

The failure test validates that when the tool gets bad or incomplete input, it fails in the expected way with helpful output.

6. Add the following code to your `_suite.ts` file to run the task mock runner.

TypeScript

```
it('it should fail if tool returns 1', function(done: Mocha.Done) {
    this.timeout(1000);

    let tp = path.join(__dirname, 'failure.js');
    let tr: ttm.MockTestRunner = new ttm.MockTestRunner(tp);

    tr.run();
    console.log(tr.succeeded);
    assert.equal(tr.succeeded, false, 'should have failed');
    assert.equal(tr.warningIssues.length, 0, "should have no
warnings");
    assert.equal(tr.errorIssues.length, 1, "should have 1 error
issue");
    assert.equal(tr.errorIssues[0], 'Bad input was given', 'error issue
output');
    assert.equal(tr.stdout.indexOf('Hello bad'), -1, "Should not
display Hello bad");

    done();
});
```

7. Run the tests.

```
tsc
mocha tests/_suite.js
```

Both tests should pass. If you want to run the tests with more verbose output (what you'd see in the build console), set the environment variable: `TASK_TEST_TRACE=1`.

```
$env:TASK_TEST_TRACE=1
```

3. Create the extension manifest file

The extension manifest contains all of the information about your extension. It includes links to your files, including your task folders and images folders. Ensure you've created an images folder with extension-icon.png. The following example is an extension manifest that contains the build or release task.

1. Copy the following .json code and save it as your `vss-extension.json` file in your `home` directory. **Don't create this file in the buildandreleasetask folder.**

JavaScript

```
{
    "manifestVersion": 1,
    "id": "build-release-task",
    "name": "Fabrikam Build and Release Tools",
    "version": "0.0.1",
    "publisher": "fabrikam",
    "targets": [
        {
            "id": "Microsoft.VisualStudio.Services"
        }
    ],
    "description": "Tools for building/releasing with Fabrikam. Includes one
build/release task.",
    "categories": [
        "Azure Pipelines"
    ],
    "icons": {
        "default": "images/extension-icon.png"
    },
    "files": [
        {
            "path": "buildandreleasetask"
        }
    ],
    "contributions": [
        {
            "id": "custom-build-release-task",
            "type": "ms.vss-distributed-task.task",
            "targets": [
                "ms.vss-distributed-task.tasks"
            ],
            "properties": {
                "name": "buildandreleasetask"
            }
        }
    ]
}
```

> **Note**
>
> Change the **publisher** to your publisher name. For more information, see **Create a publisher**.

Contributions

Property	Description
`id`	Identifier of the contribution. Must be unique within the extension. Doesn't need to match the name of the build or release task. Typically the build or release task name is in the ID of the contribution.
`type`	Type of the contribution. Should be **ms.vss-distributed-task.task**.
`targets`	Contributions "targeted" by this contribution. Should be **ms.vss-distributed-task.tasks**.
`properties.name`	Name of the task. This name must match the folder name of the corresponding self-contained build or release pipeline task.

Files

Property	Description
`path`	Path of the file or folder relative to the `home` directory.

> ⓘ **Note**
>
> For more information about the **extension manifest file**, such as its properties and what they do, check out the extension manifest reference.

4. Package your extension

After you've written your extension, the next step toward getting it into the Visual Studio Marketplace is to package all of your files together. All extensions are packaged as VSIX 2.0-compatible .vsix files. Microsoft provides a cross-platform command-line interface (CLI) to package your extension.

1. Once you have the tfx-cli, go to your extension's home directory, and run the following command:

   no-highlight

   ```
   tfx extension create --manifest-globs vss-extension.json
   ```

> ⓘ **Note**

An extension or integration's version must be incremented on every update. When you're updating an existing extension, either update the version in the manifest or pass the `--rev-version` command line switch. This increments the *patch* version number of your extension and saves the new version to your manifest. You must rev both the task version and extension version for an update to occur. `tfx extension create --manifest-globs vss-extension.json --rev-version` only updates the extension version and not the task version. For more information, see **Build Task in GitHub**.

After you have your packaged extension in a .vsix file, you're ready to publish your extension to the Marketplace.

5. Publish your extension

To publish your extension, you first create your publisher, then upload your extension, and finally share it.

Create your publisher

All extensions, including extensions from Microsoft, are identified as being provided by a publisher. If you aren't already a member of an existing publisher, you create one.

1. Sign in to the Visual Studio Marketplace Publishing Portal.
2. If you aren't already a member of an existing publisher, you're prompted to create a publisher. If you're not prompted to create a publisher, scroll down to the bottom of the page and select **Publish extensions** under **Related Sites**.
   - Specify an identifier for your publisher, for example: `mycompany-myteam`.
     - This identifier is used as the value for the `publisher` attribute in your extensions' manifest file.
   - Specify a display name for your publisher, for example: `My Team`.
3. Review the Marketplace Publisher Agreement and select **Create**.

Your publisher is defined. In a future release, you can grant permissions to view and manage your publisher's extensions. It's easier and more secure to publish extensions under a common publisher, without the need to share a set of credentials across users.

Upload your extension

1. Find the **Upload new extension** button, go to your packaged .vsix file, and select **Upload**.

   You can also upload your extension via the command line interface (CLI) by using the `tfx extension publish` command instead of `tfx extension create` to package and publish your extension in one step. You can optionally use `--share-with` to share your extension with one or more accounts after it's published.

   no-highlight

   ```
   tfx extension publish --manifest-globs your-manifest.json --share-with
   yourOrganization
   ```

   - Create a personal access token (PAT). Select the "Marketplace (publish)" scope. This scope limits the token to only being able to publish extensions to the Marketplace.

Share your extension

Now that you've uploaded your extension, it's in the Marketplace, but no one can see it. Share it with your organization so that you can install and test it.

1. Right-click your extension and select **Share**, and enter your organization information. You can share it with other accounts that you want to have access to your extension, too.

> **Important**
>
> Publishers must be verified to share extensions publicly. To learn more, see **Package/Publish/Install**.

Now that your extension is shared in the Marketplace, anyone who wants to use it must install it.

6. Create a build and release pipeline to publish the extension to Marketplace

Create a build and release pipeline on Azure DevOps to help maintain the custom task on the Marketplace.

Prerequisites - Create a build and release pipeline to publish extension

Software/tool	Information
Azure DevOps project	Create a project
Azure DevOps Extension Tasks extension	Install for free, Azure DevOps Extension Tasks in your organization.
Pipeline library variable group.	Create a pipeline library variable group to hold the variables used by the pipeline. For more information, see Add and use variable groups. You can make variable groups from the Azure DevOps Library tab or through the CLI. Use the variables within this group in your pipeline. Also, declare the following variables in the variable group:

- `publisherId`: ID of your marketplace publisher
- `extensionId`: ID of your extension, as declared in the vss-extension.json file
- `extensionName`: Name of your extension, as declared in the vss-extension.json file
- `artifactName`: Name of the artifact being created for the VSIX file

  | |Service connection | Create a new Marketplace service connection and grant

access permissions for all pipelines.

| YAML pipeline | Use the following example to create a new pipeline with YAML. For more information, see Create your first pipeline and YAML schema.

YAML

```yaml
trigger:
- main

pool:
  vmImage: "ubuntu-latest"

variables:
  - group: variable-group # Rename to whatever you named your variable group in the prerequisite stage of step 6

stages:
  - stage: Run_and_publish_unit_tests
```

```yaml
    jobs:
      - job:
        steps:
          - task: TfxInstaller@4
            inputs:
              version: "v0.x"
          - task: Npm@1
            inputs:
              command: 'install'
              workingDir: '/TaskDirectory' # Update to the name of the
directory of your task
          - task: Bash@3
            displayName: Compile Javascript
            inputs:
              targetType: "inline"
              script: |
                cd TaskDirectory # Update to the name of the directory of
your task
                tsc
          - task: Npm@1
            inputs:
              command: 'custom'
              workingDir: '/TestsDirectory' # Update to the name of the
directory of your task's tests
              customCommand: 'testScript' # See the definition in the
explanation section below - it may be called test
          - task: PublishTestResults@2
            inputs:
              testResultsFormat: 'JUnit'
              testResultsFiles: '**/ResultsFile.xml'
  - stage: Package_extension_and_publish_build_artifacts
    jobs:
      - job:
        steps:
          - task: TfxInstaller@4
            inputs:
              version: "0.x"
          - task: Npm@1
            inputs:
              command: 'install'
              workingDir: '/TaskDirectory' # Update to the name of the
directory of your task
          - task: Bash@3
            displayName: Compile Javascript
            inputs:
              targetType: "inline"
              script: |
                cd TaskDirectory # Update to the name of the directory of
your task
                tsc
          - task: QueryAzureDevOpsExtensionVersion@4
            name: QueryVersion
            inputs:
              connectTo: 'VsTeam'
              connectedServiceName: 'ServiceConnection' # Change to whatever
```

```
you named the service connection
              publisherId: '$(PublisherID)'
              extensionId: '$(ExtensionID)'
              versionAction: 'Patch'
          - task: PackageAzureDevOpsExtension@4
            inputs:
              rootFolder: '$(System.DefaultWorkingDirectory)'
              publisherId: '$(PublisherID)'
              extensionId: '$(ExtensionID)'
              extensionName: '$(ExtensionName)'
              extensionVersion: '$(QueryVersion.Extension.Version)'
              updateTasksVersion: true
              updateTasksVersionType: 'patch'
              extensionVisibility: 'private' # Change to public if you're
publishing to the marketplace
              extensionPricing: 'free'
          - task: CopyFiles@2
            displayName: "Copy Files to: $(Build.ArtifactStagingDirectory)"
            inputs:
              Contents: "**/*.vsix"
              TargetFolder: "$(Build.ArtifactStagingDirectory)"
          - task: PublishBuildArtifacts@1
            inputs:
              PathtoPublish: '$(Build.ArtifactStagingDirectory)'
              ArtifactName: '$(ArtifactName)'
              publishLocation: 'Container'
  - stage: Download_build_artifacts_and_publish_the_extension
    jobs:
      - job:
        steps:
          - task: TfxInstaller@4
            inputs:
              version: "v0.x"
          - task: DownloadBuildArtifacts@0
            inputs:
              buildType: "current"
              downloadType: "single"
              artifactName: "$(ArtifactName)"
              downloadPath: "$(System.DefaultWorkingDirectory)"
          - task: PublishAzureDevOpsExtension@4
            inputs:
              connectTo: 'VsTeam'
              connectedServiceName: 'ServiceConnection' # Change to whatever
you named the service connection
              fileType: 'vsix'
              vsixFile:
'$(PublisherID).$(ExtensionName)/$(PublisherID)..vsix'
              publisherId: '$(PublisherID)'
              extensionId: '$(ExtensionID)'
              extensionName: '$(ExtensionName)'
              updateTasksVersion: false
              extensionVisibility: 'private' # Change to public if you're
publishing to the marketplace
              extensionPricing: 'free'
```

|

For more help with triggers, such as CI and PR triggers, see Specify events that trigger pipelines.

> **Note**
>
> Each job uses a new user agent and requires dependencies to be installed.

Pipeline stages

This section helps you understand how the pipeline stages work.

Stage: Run and publish unit tests

This stage runs unit tests and publishes test results to Azure DevOps.

To run unit tests, add a custom script to the package.json file. For example:

```JSON
"scripts": {
    "testScript": "mocha ./TestFile --reporter xunit --reporter-option output=ResultsFile.xml"
},
```

1. Add "Use Node CLI for Azure DevOps (tfx-cli)" to install the tfx-cli onto your build agent.
2. Add the "npm" task with the "install" command and target the folder with the package.json file.
3. Add the "Bash" task to compile the TypeScript into JavaScript.
4. Add the "npm" task with the "custom" command, target the folder that contains the unit tests, and input `testScript` as the command. Use the following inputs:
    - Command: custom
    - Working folder that contains package.json: /TestsDirectory
    - Command and arguments: `testScript`
5. Add the "Publish Test Results" task. If you're using the Mocha XUnit reporter, ensure that the result format is "JUnit" and not "XUnit." Set the search folder to the

root directory. Use the following inputs:

- Test result format: JUnit
- Test results files: **/ResultsFile.xml
- Search folder: `$(System.DefaultWorkingDirectory)`

After the test results have been published, the output under the tests tab should look like the following example.

Stage: Package the extension and publish build artifacts

1. Add "Use Node CLI for Azure DevOps (tfx-cli)" to install the tfx-cli onto your build agent.

2. Add the "npm" task with the "install" command and target the folder with the package.json file.

3. Add the "Bash" task to compile the TypeScript into JavaScript.

4. Add the "Query Extension Version" task to query the existing extension version. Use the following inputs:

   - Connect to: Visual Studio Marketplace
   - Visual Studio Marketplace (Service connection): Service Connection
   - Publisher ID: ID of your Visual Studio Marketplace publisher
   - Extension ID: ID of your extension in the vss-extension.json file
   - Increase version: Patch
   - Output Variable: Task.Extension.Version

5. Add the "Package Extension" task to package the extensions based on manifest Json. Use the following inputs:

   - Root manifests folder: Points to root directory that contains manifest file. For example, $(System.DefaultWorkingDirectory) is the root directory
   - Manifest file(s): vss-extension.json
   - Publisher ID: ID of your Visual Studio Marketplace publisher
   - Extension ID: ID of your extension in the vss-extension.json file
   - Extension Name: Name of your extension in the vss-extension.json file

- Extension Version: $(Task.Extension.Version)
- Override tasks version: checked (true)
- Override Type: Replace Only Patch (1.0.r)
- Extension Visibility: If the extension is still in development, set the value to private. To release the extension to the public, set the value to public

6. Add the "Copy files" task to copy published files. Use the following inputs:

    - Contents: All of the files to be copied for publishing them as an artifact
    - Target folder: The folder that the files get copied to
        - For example: $(Build.ArtifactStagingDirectory)

7. Add "Publish build artifacts" to publish the artifacts for use in other jobs or pipelines. Use the following inputs:

    - Path to publish: The path to the folder that contains the files that are being published
        - For example: $(Build.ArtifactStagingDirectory)
    - Artifact name: The name given to the artifact
    - Artifacts publish location: Choose "Azure Pipelines" to use the artifact in future jobs

Stage: Download build artifacts and publish the extension

1. Add "Use Node CLI for Azure DevOps (tfx-cli)" to install the tfx-cli onto your build agent.

2. Add the "Download build artifacts" task to download the artifacts onto a new job. Use the following inputs:

    - Download artifacts produced by: If you're downloading the artifact on a new job from the same pipeline, select "Current build." If you're downloading on a new pipeline, select "Specific build."
    - Download type: Choose "Specific artifact" to download all files that were published.
    - Artifact name: The published artifact's name.
    - Destination directory: The folder where the files should be downloaded.

3. The last task that you need is the "Publish Extension" task. Use the following inputs:

    - Connect to: Visual Studio Marketplace
    - Visual Studio Marketplace connection: ServiceConnection

- Input file type: VSIX file
- VSIX file: /Publisher.*.vsix
- Publisher ID: ID of your Visual Studio Marketplace publisher
- Extension ID: ID of your extension in the vss-extension.json file
- Extension Name: Name of your extension in the vss-extension.json file
- Extension visibility: Either private or public

Optional: Install and test your extension

Install an extension that is shared with you in just a few steps:

1. From your organization control panel (`https://dev.azure.com/{organization}/_admin`), go to the project collection administration page.
2. In the **Extensions** tab, find your extension in the "Extensions Shared With Me" group and select the extension link.
3. Install the extension.

If you can't see the **Extensions** tab, make sure you're in the control panel (the administration page at the project collection level, `https://dev.azure.com/{organization}/_admin`) and not the administration page for a project.

If you don't see the **Extensions** tab, then extensions aren't enabled for your organization. You can get early access to the extensions feature by joining the Visual Studio Partner Program.

To package and publish Azure DevOps Extensions to the Visual Studio Marketplace, you can download Azure DevOps Extension Tasks.

FAQs

See the following frequently asked questions (FAQs) about adding custom build or release tasks in extensions for Azure DevOps

Q: How can I restrict Azure Pipelines commands usage for task?

You can restrict Azure Pipelines commands usage and variables, which get set by task. This action could be useful to prevent unrestricted access to variables/vso commands

for custom scripts which task executes. We recommend that you set it up for new tasks. To apply, you may need to add the following statement to your task.json file:

JSON

```
  "restrictions": {
    "commands": {
      "mode": "restricted"
    },
    "settableVariables": {
      "allowed": ["variable1", "test*"]
    }
}
```

If `restricted` value is specified for `mode` - you can only execute the following commands by the task:

- `logdetail`
- `logissue`
- `complete`
- `setprogress`
- `setsecret`
- `setvariable`
- `debug`
- `settaskvariable`
- `prependpath`
- `publish`

`settableVariables` restrictions allow you to pass in an allowlist of variables, which get set by `setvariable` or `prependpath` commands. It also allows basic regular expressions. For example, if your allowlist was: `['abc', 'test*']`, setting `abc`, `test`, or `test1` as variables with any value or prepending them to the path would succeed, but if you try to set a variable proxy it would warn. Empty list means that no variables get changed by task.

If either the `settableVariables` or `commands` key is omitted, relevant restriction isn't applied.

The restriction feature is available from 2.182.1 agent version.

Q: How is the cancellation signal handled by a task?

The pipeline agent sends `SIGINT` and `SIGTERM` signals to the relevant child process. There are no explicit means in the task library to process. For more information, see Agent jobs cancellation.

Q: How can I remove the task from project collection?

We don't support the automatic deletion of tasks. Automatic deletion isn't safe and breaks existing pipelines that already use such tasks. But, you can mark tasks as deprecated. To do so, bump the task version and mark the task as deprecated.

Q: How can I upgrade a task to the latest Node?

We recommend upgrading to the latest Node version. For example information, see Upgrading tasks to Node 16.

Related articles

- Extension manifest reference
- Build/Release Task JSON Schema
- Build/Release Task Examples

Upload tasks to project collection

Article • 02/11/2022

Azure DevOps Services

Learn how to upload tasks to organization for custom tasks or in-the-box tasks in Azure DevOps using the Node CLI for Azure DevOps (tfx-cli).

For example, this guideline can help to update in-the-box tasks on Azure DevOps Server.

> **Important**
>
> For the case of in-the-box tasks being uploaded to on-prem instance, there could be some task capabilities not supported due to the old agent version/lack of support on Azure DevOps Server side.

For more information about **tfx-cli**, see the Node CLI for Azure DevOps on GitHub.

Prerequisites

To upload tasks to project collection, you need prerequisites:

- The latest version of **Node.js**.
- The **Node CLI for Azure DevOps** to upload tasks.
  - Install **tfx-cli** using `npm`, a component of Node.js by running:

    ```
    npm install -g tfx-cli
    ```

- Permissions to update required project collection, PAT generated with scope **Environment (Read & Write)** to be able to upload tasks to the project collection.

Tfx-cli sign in with personal access token

You need to login to Azure DevOps with tfx-cli - to be able to upload pipeline tasks to the project collection.

> **Important**
>
> A personal access token is required by default for authentication to project collection in Azure DevOps. You need to create **personal access token (PAT)** with scope **Environment (Read & manage)**.

> **Tip**
>
> You can use other ways to authorize with tfx-cli - see **Authenticate in Cross-platform CLI for Azure DevOps** for more details.

To login - you should specify the path to project collection as URL. The default name of the project collection is `DefaultCollection`.

For Azure DevOps Services, path to project collection would have the following format: `https://{Azure DevOps organization name}.visualstudio.com/DefaultCollection`

For Azure DevOps Server default project collection URL will depend on the url where the server is located and its template will be: `http://{Azure DevOps Server url}/DefaultCollection`

Enter the following command and provide requested information:

```
~$ tfx login
```

Uploading tasks to the project collection

> **Tip**
>
> If you need to update in-the-box pipeline tasks, you can clone **azure-pipelines-tasks** repository, and build required tasks following **the guideline - how to build tasks**.

Now you can start to upload task using `tfx-cli`.

Enter the following command:

```
tfx build tasks upload --task-path <PATH_TO_TASK>
```

Note

PATH_TO_TASK is the path to the folder with the compiled task. For more information about using tfx-cli, see **Node CLI for Azure DevOps documentation**.

Specify jobs in your pipeline

Article • 11/01/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

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.

To learn about the key concepts and components that make up a pipeline, see Key concepts for new Azure Pipelines users.

Azure Pipelines does not support job priority for YAML pipelines. To control when jobs run, you can specify conditions and dependencies.

Define a single job

YAML

In the simplest case, a pipeline has a single job. In that case, you don't have to explicitly use the `job` keyword unless you're using a template. You can directly specify the steps in your YAML file.

This YAML file has a job that runs on a Microsoft-hosted agent and outputs `Hello world`.

YAML

```yaml
pool:
  vmImage: 'ubuntu-latest'
steps:
- bash: echo "Hello world"
```

You might want to specify more properties on that job. In that case, you can use the `job` keyword.

YAML

```yaml
jobs:
- job: myJob
  timeoutInMinutes: 10
```

```
  pool:
    vmImage: 'ubuntu-latest'
  steps:
  - bash: echo "Hello world"
```

Your pipeline could have multiple jobs. In that case, use the `jobs` keyword.

YAML

```yaml
jobs:
- job: A
  steps:
  - bash: echo "A"

- job: B
  steps:
  - bash: echo "B"
```

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

YAML

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

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

The full syntax to specify a job is:

YAML

```yaml
- job: string  # name of the job, A-Z, a-z, 0-9, and underscore
  displayName: string  # friendly name to display in the UI
  dependsOn: string | [ string ]
  condition: string
  strategy:
    parallel: # parallel strategy
    matrix: # matrix strategy
    maxParallel: number # maximum number simultaneous matrix legs to run
    # note: `parallel` and `matrix` are mutually exclusive
    # you may specify one or the other; including both is an error
    # `maxParallel` is only valid with `matrix`
  continueOnError: boolean  # 'true' if future jobs should run even if
```

```
this job fails; defaults to 'false'
  pool: pool # agent pool
  workspace:
    clean: outputs | resources | all # what to clean up before the job
runs
  container: containerReference # container to run this job inside
  timeoutInMinutes: number # how long to run the job before
automatically cancelling
  cancelTimeoutInMinutes: number # how much time to give 'run always
even if cancelled tasks' before killing them
  variables: { string: string } | [ variable | variableReference ]
  steps: [ script | bash | pwsh | powershell | checkout | task |
templateReference ]
  services: { string: string | container } # container resources to run
as a service container
  uses: # Any resources (repos or pools) required by this job that are
not already referenced
    repositories: [ string ] # Repository references to Azure Git
repositories
    pools: [ string ] # Pool names, typically when using a matrix
strategy for the 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**.

The syntax for a deployment job is:

YAML

```yaml
- 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.

Types of jobs

Jobs can be of different types, depending on where they run.

YAML

- **Agent pool jobs** run on an agent in an agent pool.
- **Server jobs** run on the Azure DevOps Server.
- **Container jobs** run in a container on an agent in an agent pool. For more information about choosing containers, see Define container jobs.

Agent pool jobs

These are the most common type of jobs and they run on an agent in an agent pool.

- When using Microsoft-hosted agents, each job in a pipeline gets a fresh agent.
- Use demands with self-hosted agents to specify what capabilities an agent must have to run your job. You may get the same agent for consecutive jobs, depending on whether there's more than one agent in your agent pool that matches your pipeline's demands. If there's only one agent in your pool that matches the pipeline's demands, the pipeline waits until this agent is available.

> **Note**
>
> Demands and capabilities are designed for use with self-hosted agents so that jobs can be matched with an agent that meets the requirements of the job. When using Microsoft-hosted agents, you select an image for the agent that matches the requirements of the job, so although it is possible to add capabilities to a Microsoft-hosted agent, you don't need to use capabilities with Microsoft-hosted agents.

YAML

YAML

```yaml
pool:
  name: myPrivateAgents    # your job runs on an agent in this pool
  demands: agent.os -equals Windows_NT    # the agent must have this
capability to run the job
steps:
- script: echo hello world
```

Or multiple demands:

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

Learn more about agent capabilities.

Server jobs

Tasks in a server job are orchestrated by and executed on the server (Azure Pipelines or TFS). A server job doesn't require an agent or any target computers. Only a few tasks are supported in a server job now. The maximum time for a server job is 30 days.

Agentless jobs supported tasks

Currently, only the following tasks are supported out of the box for agentless jobs:

- Delay task
- Invoke Azure Function task
- Invoke REST API task
- Manual Validation task
- Publish To Azure Service Bus task
- Query Azure Monitor Alerts task
- Query Work Items task

Because tasks are extensible, you can add more agentless tasks by using extensions. The default timeout for agentless jobs is 60 minutes.

YAML

The full syntax to specify a server job is:

```yaml
jobs:
- job: string
```

```
  timeoutInMinutes: number
  cancelTimeoutInMinutes: number
  strategy:
    maxParallel: number
    matrix: { string: { string: string } }

  pool: server # note: the value 'server' is a reserved keyword which
indicates this is an agentless job
```

You can also use the simplified syntax:

YAML

```
jobs:
- job: string
  pool: server # note: the value 'server' is a reserved keyword which
indicates this is an agentless job
```

Dependencies

When you define multiple jobs in a single stage, you can specify dependencies between them. Pipelines must contain at least one job with no dependencies. By default Azure DevOps YAML pipeline jobs run in parallel unless the `dependsOn` value is set.

> **Note**
>
> Each agent can run only one job at a time. To run multiple jobs in parallel you must configure multiple agents. You also need sufficient **parallel jobs**.

YAML

The syntax for defining multiple jobs and their dependencies is:

YAML

```
jobs:
- job: string
  dependsOn: string
  condition: string
```

Example jobs that build sequentially:

YAML

```yaml
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):

YAML

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

Example of fan out:

YAML

```yaml
jobs:
- job: InitialJob
  steps:
  - script: echo hello from initial job
- job: SubsequentA
  dependsOn: InitialJob
  steps:
  - script: echo hello from subsequent A
- job: SubsequentB
  dependsOn: InitialJob
  steps:
  - script: echo hello from subsequent B
```

Example of fan-in:

YAML

```yaml
jobs:
- job: InitialA
  steps:
  - script: echo hello from initial A
- job: InitialB
  steps:
  - script: echo hello from initial B
- job: Subsequent
  dependsOn:
  - InitialA
  - InitialB
  steps:
  - script: echo hello from subsequent
```

Conditions

You can specify the conditions under which each job runs. By default, a job runs if it doesn't 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.

YAML

Example to run a job based upon the status of running a previous job:

YAML

```yaml
jobs:
- job: A
  steps:
  - script: exit 1

- job: B
  dependsOn: A
  condition: failed()
  steps:
  - script: echo this will run when A fails

- job: C
  dependsOn:
  - A
  - B
  condition: succeeded('B')
```

```
  steps:
  - script: echo this will run when B runs and succeeds
```

Example of using a custom condition:

YAML

```
jobs:
- job: A
  steps:
  - script: echo hello

- job: B
  dependsOn: A
  condition: and(succeeded(), eq(variables['build.sourceBranch'],
'refs/heads/main'))
  steps:
  - script: echo this only runs for master
```

You can specify that a job run based on the value of an output variable set in a previous job. In this case, you can only use variables set in directly dependent jobs:

YAML

```
jobs:
- job: A
  steps:
  - script: "echo '##vso[task.setvariable
variable=skipsubsequent;isOutput=true]false'"
    name: printvar

- job: B
  condition: and(succeeded(),
ne(dependencies.A.outputs['printvar.skipsubsequent'], 'true'))
  dependsOn: A
  steps:
  - script: echo hello from B
```

Timeouts

To avoid taking up resources when your job is unresponsive or waiting too long, it's a good idea to set a limit on how long your job is allowed to run. Use the job timeout setting to specify the limit in minutes for running the job. Setting the value to **zero** means that the job can run:

- Forever on self-hosted agents

- For 360 minutes (6 hours) on Microsoft-hosted agents with a public project and public repository
- For 60 minutes on Microsoft-hosted agents with a private project or private repository (unless additional capacity is paid for)

The timeout period begins when the job starts running. It doesn't include the time the job is queued or is waiting for an agent.

YAML

The `timeoutInMinutes` allows a limit to be set for the job execution time. When not specified, the default is 60 minutes. When `0` is specified, the maximum limit is used (described above).

The `cancelTimeoutInMinutes` allows a limit to be set for the job cancel time when the deployment task is set to keep running if a previous task failed. When not specified, the default is 5 minutes. The value should be in range from **1** to **35790** minutes.

YAML

```yaml
jobs:
- job: Test
  timeoutInMinutes: 10 # how long to run the job before automatically
cancelling
  cancelTimeoutInMinutes: 2 # how much time to give 'run always even if
cancelled tasks' before stopping them
```

Timeouts have the following level of precedence.

1. On Microsoft-hosted agents, jobs are limited in how long they can run based on project type and whether they're run using a paid parallel job. When the Microsoft-hosted job timeout interval elapses, the job is terminated. On Microsoft-hosted agents, jobs can't run longer than this interval, regardless of any job level timeouts specified in the job.
2. The timeout configured at the job level specifies the maximum duration for the job to run. When the job level timeout interval elapses, the job is terminated. If the job is run on a Microsoft-hosted agent, setting the job level timeout to an interval greater than the built-in Microsoft-hosted job level timeout has no effect and the Microsoft-hosted job timeout is used.
3. You can also set the timeout for each task individually - see task control options. If the job level timeout interval elapses before the task completes, the running job is

terminated, even if the task is configured with a longer timeout interval.

Multi-job configuration

From a single job you author, you can run multiple jobs on multiple agents in parallel. Some examples include:

- **Multi-configuration builds:** You can build multiple configurations in parallel. For example, you could build a Visual C++ app for both `debug` and `release` configurations on both `x86` and `x64` platforms. To learn more, see Visual Studio Build - multiple configurations for multiple platforms.
- **Multi-configuration deployments:** You can run multiple deployments in parallel, for example, to different geographic regions.
- **Multi-configuration testing:** You can run test multiple configurations in parallel.
- Multi-configuration will always generate at least one job, even if a multi-configuration variable is empty.

YAML

The `matrix` strategy enables a job to be dispatched multiple times, with different variable sets. The `maxParallel` tag restricts the amount of parallelism. The following job is dispatched three times with the values of Location and Browser set as specified. However, only two jobs run at the same time.

YAML

```yaml
jobs:
- job: Test
  strategy:
    maxParallel: 2
    matrix:
      US_IE:
        Location: US
        Browser: IE
      US_Chrome:
        Location: US
        Browser: Chrome
      Europe_Chrome:
        Location: Europe
        Browser: Chrome
```

> **Note**
>
> Matrix configuration names (like `US_IE` above) must contain only basic Latin alphabet letters (A-Z, a-z), numbers, and underscores (`_`). They must start with a letter. Also, they must be 100 characters or less.

It's also possible to use output variables to generate a matrix. This can be handy if you need to generate the matrix using a script.

`matrix` accepts a runtime expression containing a stringified JSON object. That JSON object, when expanded, must match the matrixing syntax. In the example below, we've hard-coded the JSON string, but it could be generated by a scripting language or command-line program.

YAML

```yaml
jobs:
- job: generator
  steps:
  - bash: echo "##vso[task.setVariable variable=legs;isOutput=true]{'a':
{'myvar':'A'}, 'b':{'myvar':'B'}}"
    name: mtrx
  # This expands to the matrix
  #   a:
  #     myvar: A
  #   b:
  #     myvar: B
- job: runner
  dependsOn: generator
  strategy:
    matrix: $[ dependencies.generator.outputs['mtrx.legs'] ]
  steps:
  - script: echo $(myvar) # echos A or B depending on which leg is
running
```

Slicing

An agent job can be used to run a suite of tests in parallel. For example, you can run a large suite of 1000 tests on a single agent. Or, you can use two agents and run 500 tests on each one in parallel.

To apply slicing, the tasks in the job should be smart enough to understand the slice they belong to.

The Visual Studio Test task is one such task that supports test slicing. If you installed multiple agents, you can specify how the Visual Studio Test task runs in parallel on these agents.

YAML

The `parallel` strategy enables a job to be duplicated many times. Variables `System.JobPositionInPhase` and `System.TotalJobsInPhase` are added to each job. The variables can then be used within your scripts to divide work among the jobs. See Parallel and multiple execution using agent jobs.

The following job is dispatched five times with the values of `System.JobPositionInPhase` and `System.TotalJobsInPhase` set appropriately.

YAML

```yaml
jobs:
- job: Test
  strategy:
    parallel: 5
```

Job variables

If you're using YAML, variables can be specified on the job. The variables can be passed to task inputs using the macro syntax $(variableName), or accessed within a script using the stage variable.

YAML

Here's an example of defining variables in a job and using them within tasks.

YAML

```yaml
variables:
  mySimpleVar: simple var value
  "my.dotted.var": dotted var value
  "my var with spaces": var with spaces value

steps:
- script: echo Input macro = $(mySimpleVar). Env var = %MYSIMPLEVAR%
  condition: eq(variables['agent.os'], 'Windows_NT')
- script: echo Input macro = $(mySimpleVar). Env var = $MYSIMPLEVAR
```

```
  condition: in(variables['agent.os'], 'Darwin', 'Linux')
- bash: echo Input macro = $(my.dotted.var). Env var = $MY_DOTTED_VAR
- powershell: Write-Host "Input macro = $(my var with spaces). Env var =
$env:MY_VAR_WITH_SPACES"
```

For information about using a **condition**, see Specify conditions.

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 subdirectories 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're published.
- `Build.BinariesDirectory` is where tasks write their outputs.
- `Common.TestResultsDirectory` is where tasks upload their test results.

YAML

The `$(Build.ArtifactStagingDirectory)` and `$(Common.TestResultsDirectory)` are always deleted and recreated prior to every build.

When you run a pipeline on a **self-hosted agent**, by default, none of the subdirectories other than `$(Build.ArtifactStagingDirectory)` and `$(Common.TestResultsDirectory)` 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.

> **Important**
>
> The workspace clean options are applicable only for self-hosted agents. Jobs are always run on a new agent with Microsoft-hosted agents.

YAML

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

When you specify one of the `clean` options, they're interpreted as follows:

- `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.

YAML

```yaml
jobs:
- deployment: MyDeploy
  pool:
    vmImage: 'ubuntu-latest'
  workspace:
    clean: all
  environment: staging
```

> **Note**
>
> Depending on your agent capabilities and pipeline demands, each job may be routed to a different agent in your self-hosted pool. As a result, you may get a new agent for subsequent pipeline runs (or stages or jobs in the same pipeline), so **not** cleaning is not a guarantee that subsequent runs, jobs, or stages will be able to access outputs from previous runs, jobs, or stages. You can configure agent capabilities and pipeline demands to specify which agents are used to run a pipeline job, but unless there is only a single agent in the pool that meets the demands, there is no guarantee that subsequent jobs will use the same agent as previous jobs. For more information, see **Specify demands**.

In addition to workspace clean, you can also configure cleaning by configuring the **Clean** setting in the pipeline settings UI. When the **Clean** setting is **true**, which is also its default value, it's equivalent to specifying `clean: true` for every checkout step in your pipeline. When you specify `clean: true`, you'll run `git clean -ffdx` followed by `git reset --hard HEAD` before git fetching. To configure the **Clean** setting:

1. Edit your pipeline, choose **...**, and select **Triggers**.

2. Select **YAML**, **Get sources**, and configure your desired **Clean** setting. The default is **true**.

Artifact download

This example YAML file publishes the artifact `WebSite` and then downloads the artifact to `$(Pipeline.Workspace)`. The Deploy job only runs if the Build job is successful.

YAML

```yaml
# test and upload my code as an artifact named WebSite
jobs:
- job: Build
  pool:
    vmImage: 'ubuntu-latest'
  steps:
  - script: npm test
```

```
  - task: PublishBuildArtifacts@1
    inputs:
      pathtoPublish: '$(System.DefaultWorkingDirectory)'
      artifactName: WebSite

# download the artifact and deploy it only if the build job succeeded
- job: Deploy
  pool:
    vmImage: 'ubuntu-latest'
  steps:
  - checkout: none #skip checking out the default repository resource
  - task: DownloadBuildArtifacts@0
    displayName: 'Download Build Artifacts'
    inputs:
      artifactName: WebSite
      downloadPath: $(Pipeline.Workspace)

  dependsOn: Build
  condition: succeeded()
```

For information about using **dependsOn** and **condition**, see Specify conditions.

Access to OAuth token

You can allow scripts running in a job to access the current Azure Pipelines or TFS OAuth security token. The token can be used to authenticate to the Azure Pipelines REST API.

YAML

The OAuth token is always available to YAML pipelines. It must be explicitly mapped into the task or step using `env`. Here's an example:

YAML

```
steps:
- powershell: |
    $url =
"$($env:SYSTEM_TEAMFOUNDATIONCOLLECTIONURI)$env:SYSTEM_TEAMPROJECTID/_ap
is/build/definitions/$($env:SYSTEM_DEFINITIONID)?api-version=4.1-
preview"
    Write-Host "URL: $url"
    $pipeline = Invoke-RestMethod -Uri $url -Headers @{
      Authorization = "Bearer $env:SYSTEM_ACCESSTOKEN"
    }
    Write-Host "Pipeline = $($pipeline | ConvertTo-Json -Depth 100)"
```

```
    env:
      SYSTEM_ACCESSTOKEN: $(system.accesstoken)
```

What's next

- Deployment group jobs
- Conditions

Define container jobs (YAML)

Article • 05/01/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

By default, jobs run on the host machine where the agent is installed. This is convenient and typically well-suited for projects that are just beginning to adopt Azure Pipelines. Over time, you may find that you want more control over the context where your tasks run. YAML pipelines offer container jobs for this level of control.

On Linux and Windows agents, jobs may be run on the host or in a container. (On macOS and Red Hat Enterprise Linux 6, container jobs are not available.) Containers provide isolation from the host and allow you to pin specific versions of tools and dependencies. Host jobs require less initial setup and infrastructure to maintain.

Containers offer a lightweight abstraction over the host operating system. You can select the exact versions of operating systems, tools, and dependencies that your build requires. When you specify a container in your pipeline, the agent will first fetch and start the container. Then, each step of the job will run inside the container. You can't have nested containers. Containers aren't supported when an agent is already running inside a container.

If you need fine-grained control at the individual step level, step targets allow you to choose container or host for each step.

Requirements

Linux-based containers

The Azure Pipelines system requires a few things in Linux-based containers:

- Bash
- glibc-based
- Can run Node.js (which the agent provides)
- Doesn't define an `ENTRYPOINT`
- `USER` has access to `groupadd` and other privileges commands without `sudo`

And on your agent host:

- Ensure Docker is installed
- The agent must have permission to access the Docker daemon

Be sure your container has each of these tools available. Some of the stripped-down containers available on Docker Hub, especially those based on Alpine Linux, don't satisfy these minimum requirements. Containers with a `ENTRYPOINT` might not work, since Azure Pipelines will `docker create` an awaiting container and `docker exec` a series of commands, which expect the container is always up and running.

> **Note**
>
> For Windows-based Linux containers, Node.js must be pre-installed.

Windows Containers

Azure Pipelines can also run Windows Containers. Windows Server version 1803 or higher is required. Docker must be installed. Be sure your pipelines agent has permission to access the Docker daemon.

The Windows container must support running Node.js. A base Windows Nano Server container is missing dependencies required to run Node.

Hosted agents

Only `windows-2019` and `ubuntu-*` images support running containers. The macOS image doesn't support running containers.

Single job

A simple example:

YAML

```yaml
pool:
  vmImage: 'ubuntu-latest'

container: ubuntu:18.04

steps:
- script: printenv
```

This tells the system to fetch the `ubuntu` image tagged `18.04` from Docker Hub and then start the container. When the `printenv` command runs, it will happen inside the `ubuntu:18.04` container.

A Windows example:

YAML

```yaml
pool:
  vmImage: 'windows-2019'

container: mcr.microsoft.com/windows/servercore:ltsc2019

steps:
- script: set
```

> **Note**
>
> Windows requires that the kernel version of the host and container match. Since this example uses the Windows 2019 image, we will use the `2019` tag for the container.

Multiple jobs

Containers are also useful for running the same steps in multiple jobs. In the following example, the same steps run in multiple versions of Ubuntu Linux. (And we don't have to mention the `jobs` keyword, since there's only a single job defined.)

YAML

```yaml
pool:
  vmImage: 'ubuntu-latest'

strategy:
  matrix:
    ubuntu16:
      containerImage: ubuntu:16.04
    ubuntu18:
      containerImage: ubuntu:18.04
    ubuntu20:
      containerImage: ubuntu:20.04

container: $[ variables['containerImage'] ]

steps:
- script: printenv
```

Endpoints

Containers can be hosted on registries other than public Docker Hub registries. To host an image on Azure Container Registry or another private container registry (including a private Docker Hub registry), add a service connection to the private registry. Then you can reference it in a container spec:

YAML

```yaml
container:
  image: registry:ubuntu1804
  endpoint: private_dockerhub_connection

steps:
- script: echo hello
```

or

YAML

```yaml
container:
  image: myprivate.azurecr.io/windowsservercore:1803
  endpoint: my_acr_connection

steps:
- script: echo hello
```

Other container registries may also work. Amazon ECR doesn't currently work, as there are other client tools required to convert AWS credentials into something Docker can use to authenticate.

> **Note**
>
> The Red Hat Enterprise Linux 6 build of the agent won't run container job. Choose another Linux flavor, such as Red Hat Enterprise Linux 7 or above.

Options

If you need to control container startup, you can specify `options`.

YAML

```yaml
container:
  image: ubuntu:18.04
  options: --hostname container-test --ip 192.168.0.1
```

```
steps:
- script: echo hello
```

Running `docker create --help` will give you the list of supported options. You can use any option available with the docker create command.

Reusable container definition

In the following example, the containers are defined in the resources section. Each container is then referenced later, by referring to its assigned alias. (Here, we explicitly list the `jobs` keyword for clarity.)

YAML

```yaml
resources:
  containers:
  - container: u16
    image: ubuntu:16.04

  - container: u18
    image: ubuntu:18.04

  - container: u20
    image: ubuntu:20.04

jobs:
- job: RunInContainer
  pool:
    vmImage: 'ubuntu-latest'

  strategy:
    matrix:
      ubuntu16:
        containerResource: u16
      ubuntu18:
        containerResource: u18
      ubuntu20:
        containerResource: u20

  container: $[ variables['containerResource'] ]

  steps:
  - script: printenv
```

Non glibc-based containers

The Azure Pipelines agent supplies a copy of Node.js, which is required to run tasks and scripts. To find out the version of Node.js for an hosted agent, see Microsoft-hosted agents. The version of Node.js is compiled against the C runtime we use in our hosted cloud, typically glibc. Some variants of Linux use other C runtimes. For instance, Alpine Linux uses musl.

If you want to use a non-glibc-based container as a job container, you will need to arrange a few things on your own. First, you must supply your own copy of Node.js. Second, you must add a label to your image telling the agent where to find the Node.js binary. Finally, stock Alpine doesn't come with other dependencies that Azure Pipelines depends on: bash, sudo, which, and groupadd.

Bring your own Node.js

You're responsible for adding a Node binary to your container. Node 14 is a safe choice. You can start from the `node:14-alpine` image.

Tell the agent about Node.js

The agent will read a container label "com.azure.dev.pipelines.handler.node.path". If this label exists, it must be the path to the Node.js binary. For example, in an image based on `node:10-alpine`, add this line to your Dockerfile:

```
LABEL
"com.azure.dev.pipelines.agent.handler.node.path"="/usr/local/bin/node"
```

Add requirements

Azure Pipelines assumes a Bash-based system with common administration packages installed. Alpine Linux in particular doesn't come with several of the packages needed. Installing `bash`, `sudo`, and `shadow` will cover the basic needs.

```
RUN apk add bash sudo shadow
```

If you depend on any in-box or Marketplace tasks, you'll also need to supply the binaries they require.

Full example of a Dockerfile

```
FROM node:10-alpine

RUN apk add --no-cache --virtual .pipeline-deps readline linux-pam \
  && apk add bash sudo shadow \
  && apk del .pipeline-deps

LABEL
"com.azure.dev.pipelines.agent.handler.node.path"="/usr/local/bin/node"

CMD [ "node" ]
```

Multiple jobs with agent pools on a single hosted agent

The container job uses the underlying host agent Docker config.json for image registry authorization, which logs out at the end of the Docker registry container initialization. Subsequent registry image pulls authorization might be denied for “unauthorized authentication” because the Docker config.json file registered in the system for authentication has already been logged out by one of the other container jobs that are running in parallel.

The solution is to set the Docker environment variable `DOCKER_CONFIG` that is specific to each agent pool service running on the hosted agent. Export the `DOCKER_CONFIG` in each agent pool’s runsvc.sh script:

```
#insert anything to set up env when running as a service
export DOCKER_CONFIG=./.docker
```

Add stages, dependencies, & conditions

Article • 12/22/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

A stage is a logical boundary in the pipeline. It can be used to mark separation of concerns (for example, Build, QA, and production). Each stage contains one or more jobs. When you define multiple stages in a pipeline, by default, they run one after the other.

For Classic pipelines, You can organize the deployment jobs in your release pipeline into stages.

To learn how stages work with parallel jobs and licensing, see Configure and pay for parallel jobs.

To find out how stages relate to other parts of a pipeline such as jobs, see Key pipelines concepts.

You can also learn more about how stages relate to parts of a pipeline in the YAML schema stages article.

YAML

You can organize pipeline jobs into stages. Stages are the major divisions in a pipeline: "build this app", "run these tests", and "deploy to pre-production" are good examples of stages. They're logical boundaries in your pipeline where you can pause the pipeline and perform various checks.

Every pipeline has at least one stage even if you don't explicitly define it. You can also arrange stages into a dependency graph so that one stage runs before another one. There is a limit of 256 jobs for a stage.

Specify stages

YAML

In the simplest case, you don't need any logical boundaries in your pipeline. In that case, you don't have to explicitly use the `stage` keyword. You can directly specify the jobs in your YAML file.

YAML

```yaml
# this has one implicit stage and one implicit job
pool:
  vmImage: 'ubuntu-latest'
steps:
- bash: echo "Hello world"
```

YAML

```yaml
# this pipeline has one implicit stage
jobs:
- job: A
  steps:
  - bash: echo "A"

- job: B
  steps:
  - bash: echo "B"
```

If you organize your pipeline into multiple stages, you use the `stages` keyword.

YAML

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

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

If you choose to specify a `pool` at the stage level, then all jobs defined in that stage will use that pool unless otherwise specified at the job-level.

YAML

```yaml
stages:
- stage: A
```

```
  pool: StageAPool
  jobs:
  - job: A1 # will run on "StageAPool" pool based on the pool defined on
the stage
  - job: A2 # will run on "JobPool" pool
    pool: JobPool
```

The full syntax to specify a stage is:

YAML

```yaml
stages:
- stage: string  # name of the stage, A-Z, a-z, 0-9, and underscore
  displayName: string  # friendly name to display in the UI
  dependsOn: string | [ string ]
  condition: string
  pool: string | pool
  variables: { string: string } | [ variable | variableReference ]
  jobs: [ job | templateReference]
```

Specify dependencies

YAML

When you define multiple stages in a pipeline, by default, they run sequentially in the order in which you define them in the YAML file. The exception to this is when you add dependencies. With dependencies, stages run in the order of the `dependsOn` requirements.

Pipelines must contain at least one stage with no dependencies.

The syntax for defining multiple stages and their dependencies is:

YAML

```yaml
stages:
- stage: string
  dependsOn: string
  condition: string
```

Example stages that run sequentially:

YAML

```
# if you do not use a dependsOn keyword, stages run in the order they
are defined
stages:
- stage: QA
  jobs:
  - job:
    ...

- stage: Prod
  jobs:
  - job:
    ...
```

Example stages that run in parallel:

YAML

```
stages:
- stage: FunctionalTest
  jobs:
  - job:
    ...

- stage: AcceptanceTest
  dependsOn: []    # this removes the implicit dependency on previous
stage and causes this to run in parallel
  jobs:
  - job:
    ...
```

Example of fan-out and fan-in:

YAML

```
stages:
- stage: Test

- stage: DeployUS1
  dependsOn: Test    # this stage runs after Test

- stage: DeployUS2
  dependsOn: Test    # this stage runs in parallel with DeployUS1, after
Test

- stage: DeployEurope
  dependsOn:         # this stage runs after DeployUS1 and DeployUS2
  - DeployUS1
  - DeployUS2
```

Conditions

You can specify the conditions under which each stage runs with expressions. By default, a stage runs if it doesn't depend on any other stage, or if all of the stages that it depends on have completed and succeeded. You can customize this behavior by forcing a stage to run even if a previous stage fails or by specifying a custom condition.

If you customize the default condition of the preceding steps for a stage, you remove the conditions for completion and success. So, if you use a custom condition, it's common to use `and(succeeded(),custom_condition)` to check whether the preceding stage ran successfully. Otherwise, the stage runs regardless of the outcome of the preceding stage.

> **Note**
>
> Conditions for failed ('JOBNAME/STAGENAME') and succeeded ('JOBNAME/STAGENAME') as shown in the following example work only for **YAML pipelines**.

YAML

Example to run a stage based upon the status of running a previous stage:

YAML

```yaml
stages:
- stage: A

# stage B runs if A fails
- stage: B
  condition: failed()

# stage C runs if B succeeds
- stage: C
  dependsOn:
  - A
  - B
  condition: succeeded('B')
```

Example of using a custom condition:

YAML

```
stages:
- stage: A

- stage: B
  condition: and(succeeded(), eq(variables['build.sourceBranch'],
'refs/heads/main'))
```

Specify queuing policies

YAML

YAML pipelines don't support queuing policies. Each run of a pipeline is independent from and unaware of other runs. In other words, your two successive commits may trigger two pipelines, and both of them will execute the same sequence of stages without waiting for each other. While we work to bring queuing policies to YAML pipelines, we recommend that you use manual approvals in order to manually sequence and control the order the execution if this is of importance.

Specify approvals

YAML

You can manually control when a stage should run using approval checks. This is commonly used to control deployments to production environments. Checks are a mechanism available to the *resource owner* to control if and when a stage in a pipeline can consume a resource. As an owner of a resource, such as an environment, you can define checks that must be satisfied before a stage consuming that resource can start.

Currently, manual approval checks are supported on environments. For more information, see Approvals.

Deployment jobs

Article • 05/18/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

> ⓘ **Important**
>
> - Job and stage names cannot contain keywords (example: `deployment`).
> - Each job in a stage must have a unique name.

In YAML pipelines, we recommend that you put your deployment steps in a special type of job called a deployment job. A deployment job is a collection of steps that are run sequentially against the environment. A deployment job and a traditional job can exist in the same stage. Azure DevOps supports the *runOnce*, *rolling*, and the *canary* strategies.

Deployment jobs provide the following benefits:

- **Deployment history**: You get the deployment history across pipelines, down to a specific resource and status of the deployments for auditing.
- **Apply deployment strategy**: You define how your application is rolled out.

A deployment job doesn't automatically clone the source repo. You can checkout the source repo within your job with `checkout: self`.

> ⓘ **Note**
>
> This article focuses on deployment with deployment jobs. To learn how to deploy to Azure with pipelines, see **Deploy to Azure overview**.

Schema

Here's the full syntax to specify a deployment job:

YAML

```yaml
jobs:
- deployment: string   # name of the deployment job, A-Z, a-z, 0-9, and
underscore. The word "deploy" is a keyword and is unsupported as the
deployment name.
  displayName: string  # friendly name to display in the UI
```

```
  pool:                 # not required for virtual machine resources
    name: string        # Use only global level variables for defining a pool
name. Stage/job level variables are not supported to define pool name.
    demands: string | [ string ]
  workspace:
    clean: outputs | resources | all # what to clean up before the job runs
  dependsOn: string
  condition: string
  continueOnError: boolean                # 'true' if future jobs should run
even if this job fails; defaults to 'false'
  container: containerReference # container to run this job inside
  services: { string: string | container } # container resources to run as a
service container
  timeoutInMinutes: nonEmptyString        # how long to run the job before
automatically cancelling
  cancelTimeoutInMinutes: nonEmptyString  # how much time to give 'run
always even if cancelled tasks' before killing them
  variables: # several syntaxes, see specific section
  environment: string  # target environment name and optionally a resource
name to record the deployment history; format: <environment-name>.<resource-
name>
  strategy:
    runOnce:    #rolling, canary are the other strategies that are supported
      deploy:
        steps: [ script | bash | pwsh | powershell | checkout | task |
templateReference ]
```

There is a more detailed, alternative syntax you can also use for the `environment` property.

YAML

```
environment:
    name: string # Name of environment.
    resourceName: string # Name of resource.
    resourceId: string # Id of resource.
    resourceType: string # Type of environment resource.
    tags: string # List of tag filters.
```

For virtual machines, you don't need to define a pool. Any steps that you define in a deployment job with a virtual machine resource will run against that virtual machine and not against the agent in the pool. For other resource types such as Kubernetes, you do need to define a pool so that tasks can run on that machine.

Deployment strategies

When you're deploying application updates, it's important that the technique you use to deliver the update will:

- Enable initialization.
- Deploy the update.
- Route traffic to the updated version.
- Test the updated version after routing traffic.
- In case of failure, run steps to restore to the last known good version.

We achieve this by using lifecycle hooks that can run steps during deployment. Each of the lifecycle hooks resolves into an agent job or a server job (or a container or validation job in the future), depending on the `pool` attribute. By default, the lifecycle hooks will inherit the `pool` specified by the `deployment` job.

Deployment jobs use the `$(Pipeline.Workspace)` system variable.

Descriptions of lifecycle hooks

`preDeploy`: Used to run steps that initialize resources before application deployment starts.

`deploy`: Used to run steps that deploy your application. Download artifact task will be auto injected only in the `deploy` hook for deployment jobs. To stop downloading artifacts, use `- download: none` or choose specific artifacts to download by specifying Download Pipeline Artifact task.

`routeTraffic`: Used to run steps that serve the traffic to the updated version.

`postRouteTraffic`: Used to run the steps after the traffic is routed. Typically, these tasks monitor the health of the updated version for defined interval.

`on: failure` or `on: success`: Used to run steps for rollback actions or clean-up.

RunOnce deployment strategy

`runOnce` is the simplest deployment strategy wherein all the lifecycle hooks, namely `preDeploy` `deploy`, `routeTraffic`, and `postRouteTraffic`, are executed once. Then, either `on:` `success` or `on:` `failure` is executed.

YAML

```yaml
strategy:
    runOnce:
      preDeploy:
        pool: [ server | pool ] # See pool schema.
        steps:
        - script: [ script | bash | pwsh | powershell | checkout | task |
```

```
templateReference ]
      deploy:
        pool: [ server | pool ] # See pool schema.
        steps:
        ...
      routeTraffic:
        pool: [ server | pool ]
        steps:
        ...
      postRouteTraffic:
        pool: [ server | pool ]
        steps:
        ...
      on:
        failure:
          pool: [ server | pool ]
          steps:
          ...
        success:
          pool: [ server | pool ]
          steps:
          ...
```

If you're using self-hosted agents, you can use the workspace clean options to clean your deployment workspace.

YAML

```yaml
jobs:
- deployment: MyDeploy
  pool:
    vmImage: 'ubuntu-latest'
  workspace:
    clean: all
  environment: staging
```

Rolling deployment strategy

A rolling deployment replaces instances of the previous version of an application with instances of the new version of the application on a fixed set of virtual machines (rolling set) in each iteration.

We currently only support the rolling strategy to VM resources.

For example, a rolling deployment typically waits for deployments on each set of virtual machines to complete before proceeding to the next set of deployments. You could do a health check after each iteration and if a significant issue occurs, the rolling deployment can be stopped.

Rolling deployments can be configured by specifying the keyword `rolling:` under the `strategy:` node. The `strategy.name` variable is available in this strategy block, which takes the name of the strategy. In this case, rolling.

YAML

```yaml
strategy:
  rolling:
    maxParallel: [ number or percentage as x% ]
    preDeploy:
      steps:
      - script: [ script | bash | pwsh | powershell | checkout | task |
templateReference ]
    deploy:
      steps:
      ...
    routeTraffic:
      steps:
      ...
    postRouteTraffic:
      steps:
      ...
    on:
      failure:
        steps:
        ...
      success:
        steps:
        ...
```

All the lifecycle hooks are supported and lifecycle hook jobs are created to run on each VM.

`preDeploy`, `deploy`, `routeTraffic`, and `postRouteTraffic` are executed once per batch size defined by `maxParallel`. Then, either `on: success` or `on: failure` is executed.

With `maxParallel: <# or % of VMs>`, you can control the number/percentage of virtual machine targets to deploy to in parallel. This ensures that the app is running on these machines and is capable of handling requests while the deployment is taking place on the rest of the machines, which reduces overall downtime.

> **Note**
>
> There are a few known gaps in this feature. For example, when you retry a stage, it will re-run the deployment on all VMs not just failed targets.

Canary deployment strategy

Canary deployment strategy is an advanced deployment strategy that helps mitigate the risk involved in rolling out new versions of applications. By using this strategy, you can roll out the changes to a small subset of servers first. As you gain more confidence in the new version, you can release it to more servers in your infrastructure and route more traffic to it.

YAML

```yaml
strategy:
    canary:
      increments: [ number ]
      preDeploy:
        pool: [ server | pool ] # See pool schema.
        steps:
        - script: [ script | bash | pwsh | powershell | checkout | task |
templateReference ]
      deploy:
        pool: [ server | pool ] # See pool schema.
        steps:
        ...
      routeTraffic:
        pool: [ server | pool ]
        steps:
        ...
      postRouteTraffic:
        pool: [ server | pool ]
        steps:
        ...
      on:
        failure:
          pool: [ server | pool ]
          steps:
          ...
        success:
          pool: [ server | pool ]
          steps:
          ...
```

Canary deployment strategy supports the `preDeploy` lifecycle hook (executed once) and iterates with the `deploy`, `routeTraffic`, and `postRouteTraffic` lifecycle hooks. It then exits with either the `success` or `failure` hook.

The following variables are available in this strategy:

`strategy.name`: Name of the strategy. For example, canary.
`strategy.action`: The action to be performed on the Kubernetes cluster. For example, deploy, promote, or reject.

`strategy.increment`: The increment value used in the current interaction. This variable is available only in `deploy`, `routeTraffic`, and `postRouteTraffic` lifecycle hooks.

Examples

RunOnce deployment strategy

The following example YAML snippet showcases a simple use of a deploy job by using the `runOnce` deployment strategy. The example includes a checkout step.

YAML

```yaml
jobs:
  # Track deployments on the environment.
- deployment: DeployWeb
  displayName: deploy Web App
  pool:
    vmImage: 'ubuntu-latest'
  # Creates an environment if it doesn't exist.
  environment: 'smarthotel-dev'
  strategy:
    # Default deployment strategy, more coming...
    runOnce:
      deploy:
        steps:
        - checkout: self
        - script: echo my first deployment
```

With each run of this job, deployment history is recorded against the `smarthotel-dev` environment.

> ⓘ **Note**
>
> - It's also possible to create an environment with empty resources and use that as an abstract shell to record deployment history, as shown in the previous example.

The next example demonstrates how a pipeline can refer both an environment and a resource to be used as the target for a deployment job.

YAML

```yaml
jobs:
- deployment: DeployWeb
  displayName: deploy Web App
  pool:
    vmImage: 'ubuntu-latest'
  # Records deployment against bookings resource - Kubernetes namespace.
  environment: 'smarthotel-dev.bookings'
  strategy:
    runOnce:
      deploy:
        steps:
          # No need to explicitly pass the connection details.
        - task: KubernetesManifest@0
          displayName: Deploy to Kubernetes cluster
          inputs:
            action: deploy
            namespace: $(k8sNamespace)
            manifests: |
              $(System.ArtifactsDirectory)/manifests/*
            imagePullSecrets: |
              $(imagePullSecret)
            containers: |
              $(containerRegistry)/$(imageRepository):$(tag)
```

This approach has the following benefits:

- Records deployment history on a specific resource within the environment, as opposed to recording the history on all resources within the environment.
- Steps in the deployment job **automatically inherit** the connection details of the resource (in this case, a Kubernetes namespace, `smarthotel-dev.bookings`), because the deployment job is linked to the environment. This is useful in the cases where the same connection detail is set for multiple steps of the job.

Rolling deployment strategy

The rolling strategy for VMs updates up to five targets in each iteration. `maxParallel` will determine the number of targets that can be deployed to, in parallel. The selection accounts for absolute number or percentage of targets that must remain available at any time excluding the targets that are being deployed to. It is also used to determine the success and failure conditions during deployment.

YAML

```yaml
jobs:
- deployment: VMDeploy
  displayName: web
  environment:
```

```
    name: smarthotel-dev
    resourceType: VirtualMachine
  strategy:
    rolling:
      maxParallel: 5  #for percentages, mention as x%
      preDeploy:
        steps:
        - download: current
          artifact: drop
        - script: echo initialize, cleanup, backup, install certs
      deploy:
        steps:
        - task: IISWebAppDeploymentOnMachineGroup@0
          displayName: 'Deploy application to Website'
          inputs:
            WebSiteName: 'Default Web Site'
            Package: '$(Pipeline.Workspace)/drop/**/*.zip'
      routeTraffic:
        steps:
        - script: echo routing traffic
      postRouteTraffic:
        steps:
        - script: echo health check post-route traffic
      on:
        failure:
          steps:
          - script: echo Restore from backup! This is on failure
        success:
          steps:
          - script: echo Notify! This is on success
```

Canary deployment strategy

In the next example, the canary strategy for AKS will first deploy the changes with 10-percent pods, followed by 20 percent, while monitoring the health during `postRouteTraffic`. If all goes well, it will promote to 100 percent.

YAML

```
jobs:
- deployment:
  environment: smarthotel-dev.bookings
  pool:
    name: smarthotel-devPool
  strategy:
    canary:
      increments: [10,20]
      preDeploy:
        steps:
        - script: initialize, cleanup....
      deploy:
```

```
        steps:
        - script: echo deploy updates...
        - task: KubernetesManifest@0
          inputs:
            action: $(strategy.action)
            namespace: 'default'
            strategy: $(strategy.name)
            percentage: $(strategy.increment)
            manifests: 'manifest.yml'
      postRouteTraffic:
        pool: server
        steps:
        - script: echo monitor application health...
      on:
        failure:
          steps:
          - script: echo clean-up, rollback...
        success:
          steps:
          - script: echo checks passed, notify...
```

Use pipeline decorators to inject steps automatically

Pipeline decorators can be used in deployment jobs to auto-inject any custom step (for example, vulnerability scanner) to every lifecycle hook execution of every deployment job. Since pipeline decorators can be applied to all pipelines in an organization, this can be applied as part of enforcing safe deployment practices.

In addition, deployment jobs can be run as a container job along with services side-car if defined.

Support for output variables

Define output variables in a deployment job's lifecycle hooks and consume them in other downstream steps and jobs within the same stage.

To share variables between stages, output an artifact in one stage and then consume it in a subsequent stage, or use the `stageDependencies` syntax described in variables.

While executing deployment strategies, you can access output variables across jobs using the following syntax.

- For **runOnce** strategy: `$[dependencies.<job-name>.outputs['<job-name>.<step-name>.<variable-name>']]` (for example,

`$[dependencies.JobA.outputs['Deploy.StepA.VariableA']]`)

- For **runOnce** strategy plus a resourceType: `$[dependencies.<job-name>.outputs['<job-name>_<resource-name>.<step-name>.<variable-name>']]`. (for example, `$[dependencies.JobA.outputs['Deploy_VM1.StepA.VariableA']]`)
- For **canary** strategy: `$[dependencies.<job-name>.outputs['<lifecycle-hookname>_<increment-value>.<step-name>.<variable-name>']]`
- For **rolling** strategy: `$[dependencies.<job-name>.outputs['<lifecycle-hookname>_<resource-name>.<step-name>.<variable-name>']]`

YAML

```yaml
# Set an output variable in a lifecycle hook of a deployment job executing
canary strategy.
- deployment: A
  pool:
    vmImage: 'ubuntu-latest'
  environment: staging
  strategy:
    canary:
      increments: [10,20]  # Creates multiple jobs, one for each increment.
Output variable can be referenced with this.
      deploy:
        steps:
        - bash: echo "##vso[task.setvariable
variable=myOutputVar;isOutput=true]this is the deployment variable value"
          name: setvarStep
        - bash: echo $(setvarStep.myOutputVar)
          name: echovar

# Map the variable from the job.
- job: B
  dependsOn: A
  pool:
    vmImage: 'ubuntu-latest'
  variables:
    myVarFromDeploymentJob: $[
dependencies.A.outputs['deploy_10.setvarStep.myOutputVar'] ]
  steps:
  - script: "echo $(myVarFromDeploymentJob)"
    name: echovar
```

For a `runOnce` job, specify the name of the job instead of the lifecycle hook:

YAML

```yaml
# Set an output variable in a lifecycle hook of a deployment job executing
runOnce strategy.
- deployment: A
  pool:
```

```
    vmImage: 'ubuntu-latest'
  environment: staging
  strategy:
    runOnce:
      deploy:
        steps:
        - bash: echo "##vso[task.setvariable
variable=myOutputVar;isOutput=true]this is the deployment variable value"
          name: setvarStep
        - bash: echo $(setvarStep.myOutputVar)
          name: echovar

# Map the variable from the job.
- job: B
  dependsOn: A
  pool:
    vmImage: 'ubuntu-latest'
  variables:
    myVarFromDeploymentJob: $[
dependencies.A.outputs['A.setvarStep.myOutputVar'] ]
  steps:
  - script: "echo $(myVarFromDeploymentJob)"
    name: echovar
```

When you define an environment in a deployment job, the syntax of the output variable varies depending on how the environment gets defined. In this example, `env1` uses shorthand notation and `env2` includes the full syntax with a defined resource type.

YAML

```
stages:
- stage: StageA
  jobs:
  - deployment: A1
    pool:
      vmImage: 'ubuntu-latest'
    environment: env1
    strategy:
      runOnce:
        deploy:
          steps:
          - bash: echo "##vso[task.setvariable
variable=myOutputVar;isOutput=true]this is the deployment variable value"
            name: setvarStep
          - bash: echo $(System.JobName)
  - deployment: A2
    pool:
      vmImage: 'ubuntu-latest'
    environment:
      name: env2
      resourceType: virtualmachine
    strategy:
```

```
      runOnce:
        deploy:
          steps:
          - script: echo "##vso[task.setvariable
variable=myOutputVarTwo;isOutput=true]this is the second deployment variable
value"
            name: setvarStepTwo

  - job: B1
    dependsOn: A1
    pool:
      vmImage: 'ubuntu-latest'
    variables:
      myVarFromDeploymentJob: $[
dependencies.A1.outputs['A1.setvarStep.myOutputVar'] ]

    steps:
    - script: "echo $(myVarFromDeploymentJob)"
      name: echovar

  - job: B2
    dependsOn: A2
    pool:
      vmImage: 'ubuntu-latest'
    variables:
      myVarFromDeploymentJob: $[
dependencies.A2.outputs['A2.setvarStepTwo.myOutputVar'] ]
      myOutputVarTwo: $[
dependencies.A2.outputs['Deploy_vmsfortesting.setvarStepTwo.myOutputVarTwo']
]

    steps:
    - script: "echo $(myOutputVarTwo)"
      name: echovartwo
```

When you output a variable from a deployment job, referencing it from the next job uses different syntax depending on if you want to set a variable or use it as a condition for the stage.

YAML

```
stages:
- stage: StageA
  jobs:
  - job: A1
    steps:
      - pwsh: echo "##vso[task.setvariable
variable=RunStageB;isOutput=true]true"
        name: setvarStep
      - bash: echo $(System.JobName)

- stage: StageB
```

```
  dependsOn:
    - StageA

  # when referring to another stage, stage name is included in variable path
  condition: eq(dependencies.StageA.outputs['A1.setvarStep.RunStageB'],
'true')

  # Variables reference syntax differs slightly from inter-stage condition
syntax
  variables:
    myOutputVar:
$[stageDependencies.StageA.A1.outputs['setvarStep.RunStageB']]
  jobs:
  - deployment: B1
    pool:
      vmImage: 'ubuntu-latest'
    environment: envB
    strategy:
      runOnce:
        deploy:
          steps:
          - bash: echo $(myOutputVar)
```

Learn more about how to set a multi-job output variable

FAQ

My pipeline is stuck with the message "Job is pending...". How can I fix this?

This can happen when there's a name conflict between two jobs. Verify that any deployment jobs in the same stage have a unique name and that job and stage names don't contain keywords. If renaming doesn't fix the problem, review troubleshooting pipeline runs.

Are decorators supported in deployment groups?

No. You can't use decorators in deployment groups.

Author a pipeline decorator

Article • 01/27/2023

Azure DevOps Services | Azure DevOps Server 2022

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.

> **Tip**
>
> Check out our newest documentation on extension development using the **Azure DevOps Extension SDK**.

1. Add contributions to an extension

The following example assumes you're familiar with the contribution models.

1. Create an extension. Once your extension gets created, you have a `vss-extension.json` file.
2. Add contributions to the `vss-extension.json` file for our new pipeline decorator.

vss-extension.json

JSON

```
{
    "manifestVersion": 1,
    "contributions": [
        {
            "id": "my-required-task",
            "type": "ms.azure-pipelines.pipeline-decorator",
            "targets": [
                "ms.azure-pipelines-agent-job.post-job-tasks"
            ],
            "properties": {
                "template": "my-decorator.yml"
            }
```

```
        }
    ],
    "files": [
        {
            "path": "my-decorator.yml",
            "addressable": true,
            "contentType": "text/plain"
        }
    ]
}
```

Contribution options

Let's take a look at the properties and what they're used for:

Property	Description
`id`	Contribution identifier. Must be unique among contributions in this extension.
`type`	Specifies that this contribution is a pipeline decorator. Must be the string `ms.azure-pipelines.pipeline-decorator`.
`targets`	Decorators can run before your job/specified task, after, or both. See the following table for available options.
`properties.template`	(Required) The template is a YAML file included in your extension, which defines the steps for your pipeline decorator. It's a relative path from the root of your extension folder.
`properties.targettask`	The target task ID used for `ms.azure-pipelines-agent-job.pre-task-tasks` or `ms.azure-pipelines-agent-job.post-task-tasks` targets. Must be GUID string like `89b8ac58-8cb7-4479-a362-1baaacc6c7ad`

Targets

Target	Description
`ms.azure-pipelines-agent-job.pre-job-tasks`	Run before other tasks in a classic build or YAML pipeline. Due to differences in how source code checkout happens, this target runs after checkout in a YAML pipeline but before checkout in a classic build pipeline.
`ms.azure-pipelines-agent-job.post-checkout-tasks`	Run after the last `checkout` task in a classic build or YAML pipeline.

Target	Description
`ms.azure-pipelines-agent-job.post-job-tasks`	Run after other tasks in a classic build or YAML pipeline.
`ms.azure-pipelines-agent-job.pre-task-tasks`	Run before specified task in a classic build or YAML pipeline.
`ms.azure-pipelines-agent-job.post-task-tasks`	Run after specified task in a classic build or YAML pipeline.
`ms.azure-release-pipelines-agent-job.pre-task-tasks`	Run before specified task in a classic RM pipeline.
`ms.azure-release-pipelines-agent-job.post-task-tasks`	Run after specified task in a classic RM pipeline.
`ms.azure-release-pipelines-agent-job.pre-job-tasks`	Run before other tasks in a classic RM pipeline.
`ms.azure-release-pipelines-agent-job.post-job-tasks`	Run after other tasks in a classic RM pipeline.

> **Note**
>
> **Deployment jobs** in a YAML pipeline only support `ms.azure-pipelines-agent-job.pre-job-tasks` and `ms.azure-pipelines-agent-job.post-job-tasks` targets. **Jobs** support all YAML pipeline targets.

In this example, we use `ms.azure-pipelines-agent-job.post-job-tasks` because we want to run at the end of all build jobs.

This extension contributes a pipeline decorator. Next, we create a template YAML file to define the decorator's behavior.

2. Create a decorator YAML file

In the extension's properties, we chose the name "my-decorator.yml". Create that file in the root of your contribution. It holds the set of steps to run after each job. We start with a basic example and work up to the full task.

my-decorator.yml (initial version)

YAML

```yaml
steps:
- task: CmdLine@2
  displayName: 'Run my script (injected from decorator)'
  inputs:
    script: dir
```

3. Install the decorator

To add a pipeline decorator to your organization, you must install an extension. **Only private extensions can contribute pipeline decorators.** The extension must be authored and shared with your organization before it can be used.

Once the extension has been shared with your organization, search for the extension and install it.

Save the file, then build and install the extension. Create and run a basic pipeline. The decorator automatically injects our `dir` script at the end of every job. A pipeline run looks similar to the following example.

> ⓘ **Note**
>
> The decorator runs on every job in every pipeline in the organization. In later steps, we add logic to control when and how the decorator runs.

4. Inject conditions

In our example, we only need to run the virus scanner if the build outputs might be released to the public. Let's say that only builds from the default branch (typically `main`) are ever released. We should limit the decorator to jobs running against the default branch.

The updated file looks like this:

my-decorator.yml (revised version)

YAML

```
steps:
- ${{ if eq(resources.repositories['self'].ref,
resources.repositories['self'].defaultBranch) }}:
  - script: dir
    displayName: 'Run my script (injected from decorator)'
```

You can start to see the power of this extensibility point. Use the context of the current job to conditionally inject steps at runtime. Use YAML expressions to make decisions about what steps to inject and when. See pipeline decorator expression context for a full list of available data.

There's another condition we need to consider: what if the user already included the virus scanning step? We shouldn't waste time running it again. In this simple example, we'll pretend that any `script` task found in the job is running the virus scanner. (In a real implementation, you'd have a custom task to check for that instead.)

The script task's ID is `d9bafed4-0b18-4f58-968d-86655b4d2ce9`. If we see another script task, we shouldn't inject ours.

my-decorator.yml (final version)

YAML

```
steps:
- ${{ if and(eq(resources.repositories['self'].ref,
resources.repositories['self'].defaultBranch),
not(containsValue(job.steps.*.task.id, 'd9bafed4-0b18-4f58-968d-
86655b4d2ce9'))) }}:
  - script: dir
    displayName: 'Run my script (injected from decorator)'
```

5. Specify a target task

You can specify target task ID, and inject tasks before or after this target task. To specify target task, you can modify vss-extension.json manifest file like the following example.

vss-extension.json

JSON

```
{
    "contributions": [
        {
```

```
            "id": "my-required-task",
            "type": "ms.azure-pipelines.pipeline-decorator",
            "targets": [
                "ms.azure-pipelines-agent-job.pre-task-tasks",
                "ms.azure-pipelines-agent-job.post-task-tasks"
            ],
            "properties": {
                "template": "my-decorator.yml",
                "targettask": "target-task-id"
            }
        }
    ],
    ...
}
```

When you set up the 'targettask' property, you can specify ID of a target task. Tasks will be injected before/after all instances of specified target task.

Specify target task's inputs injection

You can specify a list of inputs of the target task that you want to inject as inputs to the injected task.

This feature is designed to work with custom pipeline tasks. It isn't intended to provide access to target pipeline task inputs via pipeline variables.

To get access to the target pipeline task inputs (inputs with the `target_` prefix), the injected pipeline task should use methods from the azure-pipelines-tasks-task-lib, and not the pipeline variables, for example `const inputString = tl.getInput('target_targetInput')`).

To do so, you can create your own custom pipeline task and use the target inputs there. If you need the functionality of one of the out-of-box tasks, like `CmdLine@2`, you can create a copy of the CmdLine@2 task and publish it with your decorator extension.

> **Note**
>
> This functionality is only available for tasks that are injected before or after the target task.

To specify this list of inputs, you can modify vss-extension.json manifest file like the following example.

vss-extension.json (injected task inputs version)

JSON

```
{
    "contributions": [
        {
            "id": "my-required-task",
            "type": "ms.azure-pipelines.pipeline-decorator",
            "targets": [
                "ms.azure-pipelines-agent-job.pre-task-tasks",
                "ms.azure-pipelines-agent-job.post-task-tasks"
            ],
            "properties": {
                "template": "my-decorator.yml",
                "targettask": "target-task-id",
                "targettaskinputs": ["target-task-input", "target-task-
second-input"]
            }
        }
    ],
    ...
}
```

By setting up of 'targettaskinputs' property, you can specify the list of inputs that are expected to inject. These inputs will be injected into the task with the prefix "`target_`" and will be available in the injected task like `target_target-task-input`.

> **Note**
>
> Target task inputs that get secret values with variables or get them from other tasks won't be injected.

Debug

You might need to debug when you create your decorator. You also may want to see what data you have available in the context.

You can set the `system.debugContext` variable to `true` when you queue a pipeline. Then, look at the pipeline summary page.

You see something similar to the following image.

Select the task to see the logs, which show runtime values and that the context is available.

Related articles

- About YAML expression syntax
- Pipeline decorator expression context
- Develop a web extension
- Authentication guide

Pipeline decorator expression context

Article • 10/04/2022

Azure DevOps Services

Pipeline decorators have access to context about the pipeline in which they run. As a pipeline decorator author, you can use this context to make decisions about the decorator's behavior. The information available in context is different for pipelines and for release. Also, decorators run after task names are resolved to task GUIDs. When your decorator wants to reference a task, it should use the GUID rather than the name or keyword.

> **Tip**
>
> Check out our newest documentation on extension development using the **Azure DevOps Extension SDK**.

Resources

Pipeline resources are available on the `resources` object.

Repositories

Currently, there's only one key: `repositories`. `repositories` is a map from repo ID to information about the repository.

In a designer build, the primary repo alias is `__designer_repo`. In a YAML pipeline, the primary repo is called `self`. In a release pipeline, repositories aren't available. Release artifact variables are available.

For example, to print the name of the `self` repo in a YAML pipeline:

```
steps:
- script: echo ${{ resources.repositories['self'].name }}
```

Repositories contain these properties:

JavaScript

```
resources['repositories']['self'] =
{
    "alias": "self",
    "id": "<repo guid>",
    "type": "Git",
    "version": "<commit hash>",
    "name": "<repo name>",
    "project": "<project guid>",
    "defaultBranch": "<default ref of repo, like 'refs/heads/main'>",
    "ref": "<current pipeline ref, like 'refs/heads/topic'>",
    "versionInfo": {
        "author": "<author of tip commit>",
        "message": "<commit message of tip commit>"
    },
    "checkoutOptions": {}
}
```

Job

Job details are available on the `job` object.

The data looks similar to:

JavaScript

```
job =
{
    "steps": [
        {
            "environment": null,
            "inputs": {
                "script": "echo hi"
            },
            "type": "Task",
            "task": {
                "id": "d9bafed4-0b18-4f58-968d-86655b4d2ce9",
                "name": "CmdLine",
                "version": "2.146.1"
            },
            "condition": null,
            "continueOnError": false,
            "timeoutInMinutes": 0,
            "id": "5c09f0b5-9bc3-401f-8cfb-09c716403f48",
            "name": "CmdLine",
            "displayName": "CmdLine",
            "enabled": true
        }
    ]
}
```

For instance, to conditionally add a task only if it doesn't already exist:

```yaml
- ${{ if not(containsValue(job.steps.*.task.id, 'f3ab91e7-bed6-436a-b651-399a66fe6c2a')) }}:
  - script: echo conditionally inserted
```

Variables

Pipeline variables are also available.

For instance, if the pipeline had a variable called `myVar`, its value would be available to the decorator as `variables['myVar']`.

For example, to give a decorator an opt-out, we could look for a variable. Pipeline authors who wish to opt out of the decorator can set this variable, and the decorator won't be injected. If the variable isn't present, then the decorator is injected as usual.

my-decorator.yml

```yaml
- ${{ if ne(variables['skipInjecting'], 'true') }}:
  - script: echo Injected the decorator
```

Then, in a pipeline in the organization, the author can request the decorator not to inject itself.

pipeline-with-opt-out.yml

```yaml
variables:
  skipInjecting: true
steps:
- script: echo This is the only step. No decorator is added.
```

Task names and GUIDs

Decorators run after tasks have already been turned into GUIDs. Consider the following YAML:

YAML

```yaml
steps:
- checkout: self
- bash: echo This is the Bash task
- task: PowerShell@2
  inputs:
    targetType: inline
    script: Write-Host This is the PowerShell task
```

Each of those steps maps to a task. Each task has a unique GUID. Task names and keywords map to task GUIDs before decorators run. If a decorator wants to check for the existence of another task, it must search by task GUID rather than by name or keyword.

For normal tasks (which you specify with the `task` keyword), you can look at the task's `task.json` to determine its GUID. For special keywords like `checkout` and `bash` in the previous example, you can use the following GUIDs:

Keyword	GUID	Task Name
`checkout`	`6D15AF64-176C-496D-B583-FD2AE21D4DF4`	n/a, see note below
`bash`	`6C731C3C-3C68-459A-A5C9-BDE6E6595B5B`	Bash
`script`	`D9BAFED4-0B18-4F58-968D-86655B4D2CE9`	CmdLine
`powershell`	`E213FF0F-5D5C-4791-802D-52EA3E7BE1F1`	PowerShell
`pwsh`	`E213FF0F-5D5C-4791-802D-52EA3E7BE1F1`	PowerShell
`publish`	`ECDC45F6-832D-4AD9-B52B-EE49E94659BE`	PublishPipelineArtifact
`download`	`61F2A582-95AE-4948-B34D-A1B3C4F6A737`	DownloadPipelineArtifact

After resolving task names and keywords, the previous YAML becomes:

YAML

```yaml
steps:
- task: 6D15AF64-176C-496D-B583-FD2AE21D4DF4@1
  inputs:
    repository: self
- task: 6C731C3C-3C68-459A-A5C9-BDE6E6595B5B@3
  inputs:
    targetType: inline
    script: echo This is the Bash task
- task: E213FF0F-5D5C-4791-802D-52EA3E7BE1F1@2
  inputs:
```

```
    targetType: inline
    script: Write-Host This is the PowerShell task
```

Tip

Each of these GUIDs can be found in the `task.json` for the corresponding **in-box task**. The only exception is `checkout`, which is a native capability of the agent. Its GUID is built into the Azure Pipelines service and agent.

Specify conditions

Article • 03/21/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

You can specify the conditions under which each stage, job, or step runs. By default, a job or stage runs if it doesn't depend on any other job or stage, or if *all* of the jobs or stages it depends on have completed and succeeded. This includes not only direct dependencies, but their dependencies as well, computed recursively. By default, a step runs if nothing in its job has failed yet and the step immediately preceding it has finished. You can customize this behavior by forcing a stage, job, or step to run even if a previous dependency fails or by specifying a custom condition.

YAML

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

- Only when all previous direct and indirect dependencies with the same agent pool have succeeded. If you have different agent pools, those stages or jobs will run concurrently. This is the default if there is not a condition set in the YAML.
- Even if a previous dependency has failed, unless the run was canceled. Use `succeededOrFailed()` in the YAML for this condition.
- Even if a previous dependency has failed, even if the run was canceled. Use `always()` in the YAML for this condition.
- Only when a previous dependency has failed. Use `failed()` in the YAML for this condition.
- Custom conditions

By default, steps, jobs, and stages run if all previous steps/jobs have succeeded. It's as if you specified "condition: succeeded()" (see Job status functions).

YAML

```yaml
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
```

You can also use variables in conditions.

YAML

```yaml
variables:
  isMain: $[eq(variables['Build.SourceBranch'], 'refs/heads/main')]

stages:
- stage: A
  jobs:
  - job: A1
    steps:
      - script: echo Hello Stage A!

- stage: B
  condition: and(succeeded(), eq(variables.isMain, true))
  jobs:
  - job: B1
    steps:
      - script: echo Hello Stage B!
      - script: echo $(isMain)
```

Conditions are evaluated to decide whether to start a stage, job, or step. This means that nothing computed at runtime inside that unit of work will be available. For example, if you have a job that sets a variable using a runtime expression using `$[ ]` syntax, you can't use that variable in your custom condition.

Note

When you specify your own `condition` property for a stage / job / step, you overwrite its default `condition: succeeded()`. This can lead to your stage / job / step running even if the build is cancelled. Make sure you take into account the state of the parent stage / job when writing your own conditions.

Enable a custom condition

If the built-in conditions don't meet your needs, then you can specify **custom conditions**.

Conditions are written as expressions in YAML pipelines. The agent evaluates the expression beginning with the innermost function and works out its way. The final result is a boolean value that determines if the task, job, or stage should run or not. See the expressions article for a full guide to the syntax.

Do any of your conditions make it possible for the task to run even after the build is canceled by a user? If so, then specify a reasonable value for cancel timeout so that these kinds of tasks have enough time to complete after the user cancels a run.

Pipeline behavior when build is canceled

When a build is canceled, it doesn't mean all its stages, jobs, or steps stop running. The decision depends on the stage, job, or step `conditions` you specified and at what point of the pipeline's execution you canceled the build.

If your condition doesn't take into account the state of the parent of your stage / job / step, then if the condition evaluates to `true`, your stage, job, or step will run, *even if* its parent is canceled. If its parent is skipped, then your stage, job, or step won't run.

Let's look at some examples.

Stages

In this pipeline, by default, `stage2` depends on `stage1` and `stage2` has a `condition` set. `stage2` only runs when the source branch is `main`.

yml

```yml
stages:
- stage: stage1
  jobs:
  - job: A
    steps:
      - script: echo 1; sleep 30
- stage: stage2
  condition: eq(variables['Build.SourceBranch'], 'refs/heads/main')
  jobs:
  - job: B
    steps:
      - script: echo 2
```

If you queue a build on the `main` branch, and you cancel it while `stage1` is running, `stage2` will still run, because `eq(variables['Build.SourceBranch'], 'refs/heads/main')` evaluates to `true`.

In this pipeline, `stage1` depends on `stage2`. Job `B` has a `condition` set for it.

yml

```yml
stages:
- stage: stage1
  jobs:
  - job: A
    steps:
      - script: echo 1; sleep 30
- stage: stage2
  jobs:
  - job: B
    condition: eq(variables['Build.SourceBranch'], 'refs/heads/main')
    steps:
      - script: echo 2
```

If you queue a build on the `main` branch, and you cancel it while `stage1` is running, `stage2` *won't* run, even though it contains a job `A` whose condition evaluates to `true`. The reason is because `stage2` has the default `condition: succeeded()`, which evaluates to `false` when `stage1` is canceled. Therefore, `stage2` is skipped, and none of its jobs run.

Say you have the following YAML pipeline. Notice that, by default, `stage2` depends on `stage1` and that `script: echo 2` has a `condition` set for it.

YAML

```yaml
stages:
- stage: stage1
  jobs:
  - job: A
    steps:
      - script: echo 1; sleep 30
- stage: stage2
  jobs:
  - job: B
    steps:
      - script: echo 2
        condition: eq(variables['Build.SourceBranch'],
'refs/heads/main')
```

> If you queue a build on the `main` branch, and you cancel it while `stage1` is running, `stage2` *won't* run, even though it contains a step in job `B` whose condition evaluates to `true`. The reason is because `stage2` is skipped in response to `stage1` being canceled.

To prevent stages, jobs, or steps with `conditions` from running when a build is canceled, make sure you consider their parent's state when writing the `conditions`. For more information, see Job status functions.

Examples

Run for the main branch, even if canceled, even if failing

```
eq(variables['Build.SourceBranch'], 'refs/heads/main')
```

Run for the main branch, if succeeding

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

Run if the branch is not main, if succeeding

```
and(succeeded(), ne(variables['Build.SourceBranch'], 'refs/heads/main'))
```

Run for user topic branches, if succeeding

```
and(succeeded(), startsWith(variables['Build.SourceBranch'],
'refs/heads/users/'))
```

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

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

> **Release.Artifacts.{artifact-alias}.SourceBranch** is equivalent to **Build.SourceBranch**.

Always run if a variable is set to true, even if canceled, even if failing

```
eq(variables['System.debug'], true)
```

Run if a variable is null (empty string)

Since all variables are treated as strings in Azure Pipelines, an empty string is equivalent to `null` in this pipeline.

YAML

```yaml
variables:
- name: testEmpty
  value: ''

jobs:
  - job: A
    steps:
```

```
    - script: echo testEmpty is blank
    condition: eq(variables.testEmpty, '')
```

Use a template parameter as part of a condition

When you declare a parameter in the same pipeline that you have a condition, parameter expansion happens before conditions are considered. In this case, you can embed parameters inside conditions. The script in this YAML file will run because `parameters.doThing` is true.

The `condition` in the pipeline combines two functions: `succeeded()` and `eq('${{ parameters.doThing }}', true)`. The `succeeded()` function checks if the previous step succeeded. The `succeeded()` function returns true because there was no previous step.

The `eq('${{ parameters.doThing }}', true)` function checks whether the doThing parameter is equal to `true`. Since the default value for doThing is true, the condition will return true by default unless a different values gets set in the pipeline.

For more template parameter examples, see Template types & usage.

YAML

```yaml
parameters:
- name: doThing
  default: true
  type: boolean

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

When you pass a parameter to a template, you need to set the parameter's value in your template or use templateContext to pass properties to templates.

YAML

```yaml
# parameters.yml
parameters:
- name: doThing
  default: true # value passed to the condition
  type: boolean

jobs:
  - job: B
    steps:
```

```
    - script: echo I did a thing
    condition: ${{ eq(parameters.doThing, true) }}
```

YAML

```yaml
# azure-pipeline.yml
parameters:
- name: doThing
  default: true
  type: boolean

trigger:
- none

extends:
  template: parameters.yml
```

The output of this pipeline is `I did a thing` because the parameter `doThing` is true.

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 using `isOutput=true`.

YAML

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

Use the pipeline variable created from a step in a condition in a subsequent step

You can make a variable available to future steps and specify it in a condition. By default, variables created from a step are available to future steps and don't need to be marked as multi-job output variables using `isOutput=true`.

There are some important things to note regarding the above approach and scoping:

- Variables created in a step in a job will be scoped to the steps in the same job.
- Variables created in a step will only be available in subsequent steps as environment variables.
- Variables created in a step can't be used in the step that defines them.

Below is an example of creating a pipeline variable in a step and using the variable in a subsequent step's condition and script.

YAML

```yaml
steps:

# This step creates a new pipeline variable: doThing. This variable will be
available to subsquent steps.
- bash: |
    echo "##vso[task.setvariable variable=doThing]Yes"
  displayName: Step 1

# This step is able to use doThing, so it uses it in its condition
- script: |
    # You can access the variable from Step 1 as an environment variable.
    echo "Value of doThing (as DOTHING env var): $DOTHING."
  displayName: Step 2
  condition: and(succeeded(), eq(variables['doThing'], 'Yes')) # or
and(succeeded(), eq(variables.doThing, 'Yes'))
```

FAQ

How can I trigger a job if a previous job succeeded with issues?

You can use the result of the previous job. For example, in this YAML file, the condition `eq(dependencies.A.result,'SucceededWithIssues')` allows the job to run because Job A succeeded with issues.

YAML

```yaml
jobs:
- job: A
```

```
  displayName: Job A
  continueOnError: true # next job starts even if this one fails
  steps:
  - script: echo Job A ran
  - script: exit 1

- job: B
  dependsOn: A
  condition: eq(dependencies.A.result,'SucceededWithIssues') # targets the
result of the previous job
  displayName: Job B
  steps:
  - script: echo Job B ran
```

I canceled my build, but it's still running. What's happening?

You'll experience this issue if the condition that's configured in the stage doesn't include a job status check function. To resolve the issue, add a job status check function to the condition. If you cancel a job while it's in the queue, but not running, the entire job is canceled, including all the other stages.

Learn more about a pipeline's behavior when a build is canceled.

Related articles

- Specify jobs in your pipeline
- Add stages, dependencies, & conditions

pool.demands definition

Article • 12/21/2023

Demands (for a private pool).

Definitions that reference this definition: pool

Implementations

Expand table

Implementation	Description
demands: string	Specify a demand for a private pool.
demands: string list	Specify a list of demands for a private pool.

Remarks

Use demands to make sure that the capabilities your pipeline needs are present on the agents that run it. Demands are asserted automatically by tasks or manually by you.

> **Note**
>
> Demands and capabilities are designed for use with self-hosted agents so that jobs can be matched with an agent that meets the requirements of the job. When using Microsoft-hosted agents, you select an image for the agent that matches the requirements of the job, so although it is possible to add capabilities to a Microsoft-hosted agent, you don't need to use capabilities with Microsoft-hosted agents.

You can check for the presence of a capability (Exists operation) or you can check for a specific string in a capability (Equals operation). Checking for the existence of a capability (exists) and checking for a specific string in a capability (equals) are the only two supported operations for demands.

Task demands

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 agent 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.

Exists operation

The exists operation checks for the presence of a capability with the specific name. The comparison is not case sensitive.

YAML

```yaml
pool:
  name: MyPool
  demands: myCustomCapability # exists check for myCustomCapability
```

Equals operation

The equals operation checks for the existence of a capability, and if present, checks its value with the specified value. If the capability is not present or the values don't match, the operation evaluates to false. The comparisons are not case sensitive.

YAML

```yaml
pool:
  name: MyPool
  demands: Agent.Version -equals 2.144.0 # equals check for Agent.Version
2.144.0
```

Agent variables as system capabilities

Self-hosted agents have the following system capabilities with similar names to agent variables, but they are not variables and don't require variable syntax when checking for exists or equals in a demand.

- Agent.Name
- Agent.Version
- Agent.ComputerName

- Agent.HomeDirectory
- Agent.OS
- Agent.OSArchitecture
- Agent.OSVersion (Windows agents only)

demands: string

Specify a demand for a private pool.

YAML

```yaml
demands: string # Specify a demand for a private pool.
```

`demands` string.

Specify a demand for a private pool.

Examples

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

YAML

```yaml
pool:
  name: Default
  demands: SpecialSoftware # exists check for SpecialSoftware
```

demands: string list

Specify a list of demands for a private pool.

YAML

```yaml
demands: [ string ] # Specify a list of demands for a private pool.
```

List types

Expand table

Type	Description
string	Specify a list of demands for a private pool.

Examples

To specify multiple demands, add one per line.

YAML

```yaml
pool:
  name: MyPool
  demands:
  - myCustomCapability   # exists check for myCustomCapability
  - Agent.Version -equals 2.144.0 # equals check for Agent.Version 2.144.0
```

See also

- Agent capabilities

Library of assets

Article • 04/05/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

A *library* is a collection of build and release assets for an Azure DevOps project. Assets defined in a library can be used in multiple build and release pipelines of the project. The **Library** tab can be accessed directly in Azure Pipelines.

The library contains two types of assets: variable groups and secure files.

Variable groups are only available to release pipelines in TFS 2017 and earlier. They're available to build and release pipelines in TFS 2018 and in Azure Pipelines. Task groups and service connections are available to build and release pipelines in TFS 2015 and newer, and in Azure Pipelines.

Library security

All assets defined in the **Library** share a common security model. You can control who can define new items in a library, and who can use an existing item. **Roles** are defined for library items, and membership of these roles governs the operations you can perform on those items.

Role for library item	Description
Reader	Can view the item.
User	Can use the item when authoring build or release pipelines. For example, you must be a 'User' for a variable group to use it in a release pipeline.
Administrator	Can also manage membership of all other roles for the item. The user who created an item gets automatically added to the Administrator role for that item. By default, the following groups get added to the Administrator role of the library: Build Administrators, Release Administrators, and Project Administrators.
Creator	Can create new items in the library, but this role doesn't include Reader or User permissions. The Creator role can't manage permissions for other users.

The security settings for the **Library** tab control access for *all* items in the library. Role memberships for individual items get automatically inherited from the roles of the Library node.

For more information on pipeline security roles, see About pipeline security roles.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and feel free to post your questions, search for answers, or suggest a feature on our Azure DevOps Developer Community. Support page.

Related articles

- Create and target an environment
- Manage service connections
- Add and use variable groups
- Resources in YAML
- Agents and agent pools

Define variables

Article • 03/20/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Variables give you a convenient way to get key bits of data into various parts of the pipeline. The most common use of variables is to define a value that you can then use in your pipeline. All variables are strings and are mutable. The value of a variable can change from run to run or job to job of your pipeline.

When you define the same variable in multiple places with the same name, the most locally scoped variable wins. So, a variable defined at the job level can override a variable set at the stage level. A variable defined at the stage level overrides a variable set at the pipeline root level. A variable set in the pipeline root level overrides a variable set in the Pipeline settings UI.

You can use variables with expressions to conditionally assign values and further customize pipelines.

Variables are different from runtime parameters. Runtime parameters are typed and available during template parsing.

User-defined variables

When you define a variable, you can use different syntaxes (macro, template expression, or runtime) and what syntax you use determines where in the pipeline your variable renders.

In YAML pipelines, you can set variables at the root, stage, and job level. You can also specify variables outside of a YAML pipeline in the UI. When you set a variable in the UI, that variable can be encrypted and set as secret.

User-defined variables can be set as read-only. There are naming restrictions for variables (example: you can't use `secret` at the start of a variable name).

You can use a variable group to make variables available across multiple pipelines.

Use templates to define variables in one file that are used in multiple pipelines.

User-defined multi-line variables

Azure DevOps supports multi-line variables but there are a few limitations.

Downstream components such as pipeline tasks may not handle the variable values correctly.

Azure DevOps won't alter user-defined variable values. Variable values need to be formatted correctly before being passed as multi-line variables. When formatting your variable, avoid special characters, don't use restricted names, and make sure you use a line ending format that works for the operating system of your agent.

Multi-line variables behave differently depending on the operating system. To avoid this, make sure that you format multi-line variables correctly for the target operating system.

Azure DevOps never alters variable values, even if you provide unsupported formatting.

System variables

In addition to user-defined variables, Azure Pipelines has system variables with predefined values. If you're using YAML or classic build pipelines, see predefined variables for a comprehensive list of system variables. If you're using classic release pipelines, see release variables.

System variables get set with their current value when you run the pipeline. Some variables are set automatically. As a pipeline author or end user, you change the value of a system variable before the pipeline runs.

System variables are read-only.

Environment variables

Environment variables are specific to the operating system you're using. They're injected into a pipeline in platform-specific ways. The format corresponds to how environment variables get formatted for your specific scripting platform.

On UNIX systems (macOS and Linux), environment variables have the format `$NAME`. On Windows, the format is `%NAME%` for batch and `$env:NAME` in PowerShell.

System and user-defined variables also get injected as environment variables for your platform. When variables convert into environment variables, variable names become uppercase, and periods turn into underscores. For example, the variable name `any.variable` becomes the variable name `$ANY_VARIABLE`.

There are variable naming restrictions for environment variables (example: you can't use `secret` at the start of a variable name).

Variable naming restrictions

User-defined and environment variables can consist of letters, numbers, `.`, and `_` characters. Don't use variable prefixes reserved by the system. These are: `endpoint`, `input`, `secret`, `path`, and `securefile`. Any variable that begins with one of these strings (regardless of capitalization) won't be available to your tasks and scripts.

Understand variable syntax

Azure Pipelines supports three different ways to reference variables: macro, template expression, and runtime expression. You can use each syntax for a different purpose and each have some limitations.

In a pipeline, template expression variables (`${{ variables.var }}`) get processed at compile time, before runtime starts. Macro syntax variables (`$(var)`) get processed during runtime before a task runs. Runtime expressions (`$[variables.var]`) also get processed during runtime but are intended to be used with conditions and expressions. When you use a runtime expression, it must take up the entire right side of a definition.

In this example, you can see that the template expression still has the initial value of the variable after the variable is updated. The value of the macro syntax variable updates. The template expression value doesn't change because all template expression variables get processed at compile time before tasks run. In contrast, macro syntax variables evaluate before each task runs.

YAML

```yaml
variables:
- name: one
  value: initialValue

steps:
  - script: |
      echo ${{ variables.one }} # outputs initialValue
      echo $(one)
    displayName: First variable pass
  - bash: echo "##vso[task.setvariable variable=one]secondValue"
    displayName: Set new variable value
  - script: |
      echo ${{ variables.one }} # outputs initialValue
```

```
      echo $(one) # outputs secondValue
    displayName: Second variable pass
```

Macro syntax variables

Most documentation examples use macro syntax (`$(var)`). Macro syntax is designed to interpolate variable values into task inputs and into other variables.

Variables with macro syntax get processed before a task executes during runtime. Runtime happens after template expansion. When the system encounters a macro expression, it replaces the expression with the contents of the variable. If there's no variable by that name, then the macro expression does not change. For example, if `$(var)` can't be replaced, `$(var)` won't be replaced by anything.

Macro syntax variables remain unchanged with no value because an empty value like `$()` might mean something to the task you're running and the agent shouldn't assume you want that value replaced. For example, if you use `$(foo)` to reference variable `foo` in a Bash task, replacing all `$()` expressions in the input to the task could break your Bash scripts.

Macro variables are only expanded when they're used for a value, not as a keyword. Values appear on the right side of a pipeline definition. The following is valid: `key: $(value)`. The following isn't valid: `$(key): value`. Macro variables aren't expanded when used to display a job name inline. Instead, you must use the `displayName` property.

> **Note**
>
> Macro syntax variables are only expanded for `stages`, `jobs`, and `steps`. You cannot, for example, use macro syntax inside a `resource` or `trigger`.

This example uses macro syntax with Bash, PowerShell, and a script task. The syntax for calling a variable with macro syntax is the same for all three.

YAML

```yaml
variables:
 - name: projectName
   value: contoso

steps:
- bash: echo $(projectName)
- powershell: echo $(projectName)
- script: echo $(projectName)
```

Template expression syntax

You can use template expression syntax to expand both template parameters and variables (`${{ variables.var }}`). Template variables process at compile time, and get replaced before runtime starts. Template expressions are designed for reusing parts of YAML as templates.

Template variables silently coalesce to empty strings when a replacement value isn't found. Template expressions, unlike macro and runtime expressions, can appear as either keys (left side) or values (right side). The following is valid: `${{ variables.key }} : ${{ variables.value }}`.

Runtime expression syntax

You can use runtime expression syntax for variables that are expanded at runtime (`$[variables.var]`). Runtime expression variables silently coalesce to empty strings when a replacement value isn't found. Use runtime expressions in job conditions, to support conditional execution of jobs, or whole stages.

Runtime expression variables are only expanded when they're used for a value, not as a keyword. Values appear on the right side of a pipeline definition. The following is valid: `key: $[variables.value]`. The following isn't valid: `$[variables.key]: value`. The runtime expression must take up the entire right side of a key-value pair. For example, `key: $[variables.value]` is valid but `key: $[variables.value] foo` isn't.

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

What syntax should I use?

Use macro syntax if you're providing input for a task.

Choose a runtime expression if you're working with conditions and expressions. However, don't use a runtime expression if you don't want your empty variable to print (example: `$[variables.var]`). For example, if you have conditional logic that relies on a variable having a specific value or no value. In that case, you should use a macro expression.

If you're defining a variable in a template, use a template expression.

Set variables in pipeline

YAML

In the most common case, you set the variables and use them within the YAML file. This allows you to track changes to the variable in your version control system. You can also define variables in the pipeline settings UI (see the Classic tab) and reference them in your YAML.

Here's an example that shows how to set two variables, `configuration` and `platform`, and use them later in steps. To use a variable in a YAML statement, wrap it in `$()`. Variables can't be used to define a `repository` in a YAML statement.

YAML

```yaml
# Set variables once
variables:
  configuration: debug
  platform: x64

steps:

# Use them once
- task: MSBuild@1
  inputs:
    solution: solution1.sln
    configuration: $(configuration) # Use the variable
    platform: $(platform)

# Use them again
- task: MSBuild@1
  inputs:
    solution: solution2.sln
    configuration: $(configuration) # Use the variable
    platform: $(platform)
```

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.

When you define a variable at the top of a YAML, the variable is available to all jobs and stages in the pipeline and is a global variable. Global variables defined in a YAML aren't visible in the pipeline settings UI.

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

YAML

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

jobs:
- job: job1
  pool:
    vmImage: 'ubuntu-latest'
  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-latest'
  variables:
    job_variable2: value2    # this is only available in job2
  steps:
  - bash: echo $(global_variable)
  - bash: echo $(job_variable2)
  - bash: echo $GLOBAL_VARIABLE
```

The output from both jobs looks like this:

text

```text
# job1
value
value1
value1
```

```
# job2
value
value2
value
```

Specify variables

In the preceding examples, the `variables` keyword is followed by a list of key-value pairs. The keys are the variable names and the values are the variable values.

There's another syntax, useful when you want to use variable templates or variable groups. Use this syntax at the root level of a pipeline.

In this alternate syntax, the `variables` keyword takes a list of variable specifiers. The variable specifiers are `name` for a regular variable, `group` for a variable group, and `template` to include a variable template. The following example demonstrates all three.

YAML

```yaml
variables:
# a regular variable
- name: myvariable
  value: myvalue
# a variable group
- group: myvariablegroup
# a reference to a variable template
- template: myvariabletemplate.yml
```

Learn more about variable reuse with templates.

Access variables through the environment

Notice that variables are also made available to scripts through environment variables. The syntax for using these environment variables depends on the scripting language.

The name is upper-cased, and the `.` is replaced with the `_`. This is automatically inserted into the process environment. Here are some examples:

- Batch script: `%VARIABLE_NAME%`
- PowerShell script: `$env:VARIABLE_NAME`
- Bash script: `$VARIABLE_NAME`

> **Important**
>
> Predefined variables that contain file paths are translated to the appropriate styling (Windows style C:\foo\ versus Unix style /foo/) based on agent host type and shell type. If you are running bash script tasks on Windows, you should use the environment variable method for accessing these variables rather than the pipeline variable method to ensure you have the correct file path styling.

Set secret variables

YAML

Don't set secret variables in your YAML file. Operating systems often log commands for the processes that they run, and you wouldn't want the log to include a secret that you passed in as an input. Use the script's environment or map the variable within the `variables` block to pass secrets to your pipeline.

You need to set secret variables in the pipeline settings UI for your pipeline. These variables are scoped to the pipeline where they are set. You can also set secret variables in variable groups.

To set secrets in the web interface, follow these steps:

1. Go to the **Pipelines** page, select the appropriate pipeline, and then select **Edit**.
2. Locate the **Variables** for this pipeline.
3. Add or update the variable.
4. Select the 🔒 lock icon to store the variable in an encrypted manner.
5. 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. Be careful about who has access to alter your pipeline.

> **Important**
>
> We make an effort to mask secrets from appearing in Azure Pipelines output, but you still need to take precautions. Never echo secrets as output. Some operating systems log command line arguments. Never pass secrets on the

command line. Instead, we suggest that you map your secrets into environment variables.

We never mask substrings of secrets. If, for example, "abc123" is set as a secret, "abc" isn't masked from the logs. This is to avoid masking secrets at too granular of a level, making the logs unreadable. For this reason, secrets should not contain structured data. If, for example, "{ "foo": "bar" }" is set as a secret, "bar" isn't masked from the logs.

Unlike a normal variable, they are not automatically decrypted into environment variables for scripts. You need to explicitly map secret variables.

The following example shows how to use a secret variable called `mySecret` in PowerShell and Bash scripts. Unlike a normal pipeline variable, there's no environment variable called `MYSECRET`.

YAML

```yaml
variables:
 GLOBAL_MYSECRET: $(mySecret) # this will not work because the secret
variable needs to be mapped as env
 GLOBAL_MY_MAPPED_ENV_VAR: $(nonSecretVariable) # this works because
it's not a secret.

steps:

- powershell: |
    Write-Host "Using an input-macro works: $(mySecret)"
    Write-Host "Using the env var directly does not work: $env:MYSECRET"
    Write-Host "Using a global secret var mapped in the pipeline does
not work either: $env:GLOBAL_MYSECRET"
    Write-Host "Using a global non-secret var mapped in the pipeline
works: $env:GLOBAL_MY_MAPPED_ENV_VAR"
    Write-Host "Using the mapped env var for this task works and is
recommended: $env:MY_MAPPED_ENV_VAR"
  env:
    MY_MAPPED_ENV_VAR: $(mySecret) # the recommended way to map to an
env variable

- bash: |
    echo "Using an input-macro works: $(mySecret)"
    echo "Using the env var directly does not work: $MYSECRET"
    echo "Using a global secret var mapped in the pipeline does not work
either: $GLOBAL_MYSECRET"
    echo "Using a global non-secret var mapped in the pipeline works:
$GLOBAL_MY_MAPPED_ENV_VAR"
    echo "Using the mapped env var for this task works and is
recommended: $MY_MAPPED_ENV_VAR"
  env:
    MY_MAPPED_ENV_VAR: $(mySecret) # the recommended way to map to an
```

```
env variable
```

The output from both tasks in the preceding script would look like this:

```text
Using an input-macro works: ***
Using the env var directly does not work:
Using a global secret var mapped in the pipeline does not work either:
Using a global non-secret var mapped in the pipeline works: foo
Using the mapped env var for this task works and is recommended: ***
```

You can also use secret variables outside of scripts. For example, you can map secret variables to tasks using the `variables` definition. This example shows how to use secret variables `$(vmsUser)` and `$(vmsAdminPass)` in an Azure file copy task.

```yaml
variables:
  VMS_USER: $(vmsUser)
  VMS_PASS: $(vmsAdminPass)

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: AzureFileCopy@4
  inputs:
    SourcePath: 'my/path'
    azureSubscription: 'my-subscription'
    Destination: 'AzureVMs'
    storage: 'my-storage'
    resourceGroup: 'my-rg'
    vmsAdminUserName: $(VMS_USER)
    vmsAdminPassword: $(VMS_PASS)
```

Reference secret variables in variable groups

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 secret, and is mapped to the environment variable `$env:MY_MAPPED_TOKEN` so that it can be referenced in the YAML.

This YAML makes a REST call to retrieve a list of releases, and outputs the result.

YAML

```yaml
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: |
        # Encode the Personal Access Token (PAT)
        # $env:USER is a normal variable in the variable group
        # $env:MY_MAPPED_TOKEN is a mapped secret variable
        $base64AuthInfo =
[Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes(("{0}:{1}" -f
$env:USER,$env:MY_MAPPED_TOKEN)))

        # Get a list of releases
        $uri =
"https://vsrm.dev.azure.com/$(devopsAccount)/$(projectName)/_apis/releas
e/releases?api-version=5.1"

        # Invoke the REST call
        $result = Invoke-RestMethod -Uri $uri -Method Get -ContentType
"application/json" -Headers @{Authorization=("Basic {0}" -f
$base64AuthInfo)}

        # Output releases in JSON
        Write-Host $result.value
  env:
    MY_MAPPED_TOKEN: $(token) # Maps the secret variable $(token) from
my-var-group
```

Important

By default with GitHub repositories, secret variables associated with your pipeline aren't made available to pull request builds of forks. For more information, see Contributions from forks.

Share variables across pipelines

To share variables across multiple pipelines in your project, use the web interface. Under **Library**, use variable groups.

Use output variables from tasks

Some tasks define output variables, which you can consume in downstream steps, jobs, and stages. In YAML, you can access variables across jobs and stages by using dependencies.

When referencing matrix jobs in downstream tasks, you'll need to use a different syntax. See Set a multi-job output variable.

- To reference a variable from a different task within the same job, use `TASK.VARIABLE`.
- To reference a variable from a task from a different job, use `dependencies.JOB.outputs['TASK.VARIABLE']`.

> **Note**
>
> By default, each stage in a pipeline depends on the one just before it in the YAML file. If you need to refer to a stage that isn't immediately prior to the current one, you can override this automatic default by adding a `dependsOn` section to the stage.

> **Note**
>
> The following examples use standard pipeline syntax. If you're using deployment pipelines, both variable and conditional variable syntax will differ. For information about the specific syntax to use, see **Deployment jobs**.

YAML

For these examples, assume we have a task called `MyTask`, which sets an output variable called `MyVar`. Learn more about the syntax in Expressions - Dependencies.

Use outputs in the same job

YAML

```yaml
steps:
- task: MyTask@1  # this step generates the output variable
```

```
  name: ProduceVar  # because we're going to depend on it, we need to
name the step
- script: echo $(ProduceVar.MyVar) # this step uses the output variable
```

Use outputs in a different job

YAML

```yaml
jobs:
- job: A
  steps:
  # assume that MyTask generates an output variable called "MyVar"
  # (you would learn that from the task's documentation)
  - task: MyTask@1
    name: ProduceVar  # because we're going to depend on it, we need to
name the step
- job: B
  dependsOn: A
  variables:
    # map the output variable from A into this job
    varFromA: $[ dependencies.A.outputs['ProduceVar.MyVar'] ]
  steps:
  - script: echo $(varFromA) # this step uses the mapped-in variable
```

Use outputs in a different stage

To use the output from a different stage, you must use the syntax depending on whether you're at the stage or job level:

- At the stage level, the format for referencing variables from a different stage is `dependencies.STAGE.outputs['JOB.TASK.VARIABLE']`. You can use these variables in conditions.
- At the job level, the format for referencing variables from a different stage is `stageDependencies.STAGE.JOB.outputs['TASK.VARIABLE']`

Output variables are only available in the next downstream stage. If multiple stages consume the same output variable, use the `dependsOn` condition.

YAML

```yaml
stages:
- stage: One
  jobs:
  - job: A
    steps:
    - task: MyTask@1  # this step generates the output variable
```

```
      name: ProduceVar  # because we're going to depend on it, we need
to name the step

- stage: Two
  dependsOn:
  - One
  jobs:
  - job: B
    variables:
      # map the output variable from A into this job
      varFromA: $[ stageDependencies.One.A.outputs['ProduceVar.MyVar'] ]
    steps:
    - script: echo $(varFromA) # this step uses the mapped-in variable

- stage: Three
  dependsOn:
  - One
  - Two
  jobs:
  - job: C
    variables:
      # map the output variable from A into this job
      varFromA: $[ stageDependencies.One.A.outputs['ProduceVar.MyVar'] ]
    steps:
    - script: echo $(varFromA) # this step uses the mapped-in variable
```

You can also pass variables between stages with a file input. To do so, you'll need to define variables in the second stage at the job level, and then pass the variables as `env:` inputs.

Bash

```
## script-a.sh
echo "##vso[task.setvariable variable=sauce;isOutput=true]crushed
tomatoes"
```

Bash

```
## script-b.sh
echo 'Hello file version'
echo $skipMe
echo $StageSauce
```

YAML

```
## azure-pipelines.yml
stages:

- stage: one
  jobs:
```

```
  - job: A
    steps:
    - task: Bash@3
      inputs:
          filePath: 'script-a.sh'
      name: setvar
    - bash: |
       echo "##vso[task.setvariable
variable=skipsubsequent;isOutput=true]true"
      name: skipstep

- stage: two
  jobs:
  - job: B
    variables:
      - name: StageSauce
        value: $[ stageDependencies.one.A.outputs['setvar.sauce'] ]
      - name: skipMe
        value: $[
stageDependencies.one.A.outputs['skipstep.skipsubsequent'] ]
    steps:
    - task: Bash@3
      inputs:
        filePath: 'script-b.sh'
      name: fileversion
      env:
        StageSauce: $(StageSauce) # predefined in variables section
        skipMe: $(skipMe) # predefined in variables section
    - task: Bash@3
      inputs:
        targetType: 'inline'
        script: |
          echo 'Hello inline version'
          echo $(skipMe)
          echo $(StageSauce)
```

The output from stages in the preceding pipeline looks like this:

```text
Hello inline version
true
crushed tomatoes
```

List variables

You can list all of the variables in your pipeline with the az pipelines variable list command. To get started, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines variable list [--org]
                           [--pipeline-id]
                           [--pipeline-name]
                           [--project]
```

Parameters

- **org**: Azure DevOps organization URL. You can configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **pipeline-id**: Required if **pipeline-name** isn't supplied. ID of the pipeline.
- **pipeline-name**: Required if **pipeline-id** isn't supplied, but ignored if **pipeline-id** is supplied. Name of the pipeline.
- **project**: Name or ID of the project. You can configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up by using `git config`.

Example

The following command lists all of the variables in the pipeline with ID **12** and shows the result in table format.

Azure CLI

```
az pipelines variable list --pipeline-id 12 --output table

Name           Allow Override    Is Secret    Value
-------------  ----------------  -----------  ------------
MyVariable     False             False        platform
NextVariable   False             True         platform
Configuration  False             False        config.debug
```

Set variables in scripts

Scripts can define variables that are later consumed in subsequent steps in the pipeline. All variables set by this method are treated as strings. To set a variable from a script, you use a command syntax and print to stdout.

YAML

Set a job-scoped variable from a script

To set a variable from a script, you use the `task.setvariable` logging command. This updates the environment variables for subsequent jobs. Subsequent jobs have access to the new variable with macro syntax and in tasks as environment variables.

When `issecret` is true, the value of the variable will be saved as secret and masked from the log. For more information on secret variables, see logging commands.

YAML

```yaml
steps:
# Create a variable
- bash: |
    echo "##vso[task.setvariable variable=sauce]crushed tomatoes" #
remember to use double quotes

# Use the variable
# "$(sauce)" is replaced by the contents of the `sauce` variable by
Azure Pipelines
# before handing the body of the script to the shell.
- bash: |
    echo my pipeline variable is $(sauce)
```

Subsequent steps will also have the pipeline variable added to their environment. You can't use the variable in the step that it's defined.

YAML

```yaml
steps:
# Create a variable
# Note that this does not update the environment of the current script.
- bash: |
    echo "##vso[task.setvariable variable=sauce]crushed tomatoes"

# An environment variable called `SAUCE` has been added to all
downstream steps
- bash: |
    echo "my environment variable is $SAUCE"
- pwsh: |
    Write-Host "my environment variable is $env:SAUCE"
```

The output from the preceding pipeline.

text

```
my environment variable is crushed tomatoes
my environment variable is crushed tomatoes
```

Set a multi-job output variable

If you want to make a variable available to future jobs, you must mark it as an output variable by using `isOutput=true`. Then you can map it into future jobs by using the `$[]` syntax and including the step name that set the variable. Multi-job output variables only work for jobs in the same stage.

To pass variables to jobs in different stages, use the stage dependencies syntax.

> **Note**
>
> By default, each stage in a pipeline depends on the one just before it in the YAML file. Therefore, each stage can use output variables from the prior stage. To access further stages, you will need to alter the dependency graph, for instance, if stage 3 requires a variable from stage 1, you will need to declare an explicit dependency on stage 1.

When you create a multi-job output variable, you should assign the expression to a variable. In this YAML, `$[ dependencies.A.outputs['setvarStep.myOutputVar'] ]` is assigned to the variable `$(myVarFromJobA)`.

YAML

```yaml
jobs:
# Set an output variable from job A
- job: A
  pool:
    vmImage: 'windows-latest'
  steps:
  - powershell: echo "##vso[task.setvariable
variable=myOutputVar;isOutput=true]this is the value"
    name: setvarStep
  - script: echo $(setvarStep.myOutputVar)
    name: echovar

# Map the variable into job B
- job: B
  dependsOn: A
  pool:
    vmImage: 'ubuntu-18.04'
  variables:
    myVarFromJobA: $[ dependencies.A.outputs['setvarStep.myOutputVar'] ]
```

```
# map in the variable

# remember, expressions require single quotes
  steps:
  - script: echo $(myVarFromJobA)
    name: echovar
```

The output from the preceding pipeline.

```text
this is the value
this is the value
```

If you're setting a variable from one stage to another, use `stageDependencies`.

```YAML
stages:
- stage: A
  jobs:
  - job: A1
    steps:
     - bash: echo "##vso[task.setvariable
variable=myStageOutputVar;isOutput=true]this is a stage output var"
       name: printvar

- stage: B
  dependsOn: A
  variables:
    myVarfromStageA: $[
stageDependencies.A.A1.outputs['printvar.myStageOutputVar'] ]
  jobs:
  - job: B1
    steps:
    - script: echo $(myVarfromStageA)
```

If you're setting a variable from a matrix or slice then to reference the variable when you access it from a downstream job, you must include:

- The name of the job.
- The step.

```YAML
jobs:

# Set an output variable from a job with a matrix
```

```yaml
- job: A
  pool:
    vmImage: 'ubuntu-18.04'
  strategy:
    maxParallel: 2
    matrix:
      debugJob:
        configuration: debug
        platform: x64
      releaseJob:
        configuration: release
        platform: x64
  steps:
  - bash: echo "##vso[task.setvariable
variable=myOutputVar;isOutput=true]this is the $(configuration) value"
    name: setvarStep
  - bash: echo $(setvarStep.myOutputVar)
    name: echovar

# Map the variable from the debug job
- job: B
  dependsOn: A
  pool:
    vmImage: 'ubuntu-18.04'
  variables:
    myVarFromJobADebug: $[
dependencies.A.outputs['debugJob.setvarStep.myOutputVar'] ]
  steps:
  - script: echo $(myVarFromJobADebug)
    name: echovar
```

YAML

```yaml
jobs:

# Set an output variable from a job with slicing
- job: A
  pool:
    vmImage: 'ubuntu-18.04'
    parallel: 2 # Two slices
  steps:
  - bash: echo "##vso[task.setvariable
variable=myOutputVar;isOutput=true]this is the slice
$(system.jobPositionInPhase) value"
    name: setvarStep
  - script: echo $(setvarStep.myOutputVar)
    name: echovar

# Map the variable from the job for the first slice
- job: B
  dependsOn: A
  pool:
    vmImage: 'ubuntu-18.04'
```

```
  variables:
    myVarFromJobsA1: $[
dependencies.A.outputs['job1.setvarStep.myOutputVar'] ]
  steps:
  - script: "echo $(myVarFromJobsA1)"
    name: echovar
```

Be sure to prefix the job name to the output variables of a deployment job. In this case, the job name is `A`:

YAML

```
jobs:

# Set an output variable from a deployment
- deployment: A
  pool:
    vmImage: 'ubuntu-18.04'
  environment: staging
  strategy:
    runOnce:
      deploy:
        steps:
        - bash: echo "##vso[task.setvariable
variable=myOutputVar;isOutput=true]this is the deployment variable
value"
          name: setvarStep
        - bash: echo $(setvarStep.myOutputVar)
          name: echovar

# Map the variable from the job for the first slice
- job: B
  dependsOn: A
  pool:
    vmImage: 'ubuntu-18.04'
  variables:
    myVarFromDeploymentJob: $[
dependencies.A.outputs['A.setvarStep.myOutputVar'] ]
  steps:
  - bash: "echo $(myVarFromDeploymentJob)"
    name: echovar
```

Set variables by using expressions

YAML

You can set a variable by using an expression. We already encountered one case of this to set a variable to the output of another from a previous job.

YAML

```yaml
- job: B
  dependsOn: A
  variables:
    myVarFromJobsA1: $[
dependencies.A.outputs['job1.setvarStep.myOutputVar'] ] # remember to
use single quotes
```

You can use any of the supported expressions for setting a variable. Here's an example of setting a variable to act as a counter that starts at 100, gets incremented by 1 for every run, and gets reset to 100 every day.

YAML

```yaml
jobs:
- job:
  variables:
    a: $[counter(format('{0:yyyyMMdd}', pipeline.startTime), 100)]
  steps:
  - bash: echo $(a)
```

For more information about counters, dependencies, and other expressions, see expressions.

Configure settable variables for steps

You can define `settableVariables` within a step or specify that no variables can be set.

In this example, the script cannot set a variable.

YAML

```yaml
steps:
- script: echo This is a step
  target:
    settableVariables: none
```

In this example, the script allows the variable `sauce` but not the variable `secretSauce`. You'll see a warning on the pipeline run page.

YAML

```yaml
steps:
  - bash: |
      echo "##vso[task.setvariable variable=Sauce;]crushed tomatoes"
      echo "##vso[task.setvariable variable=secretSauce;]crushed tomatoes
with garlic"
    target:
     settableVariables:
      - sauce
    name: SetVars
  - bash:
      echo "Sauce is $(sauce)"
      echo "secretSauce is $(secretSauce)"
    name: OutputVars
```

Allow at queue time

YAML

If a variable appears in the `variables` block of a YAML file, its value is fixed and can't be overridden at queue time. Best practice is to define your variables in a YAML file but there are times when this doesn't make sense. For example, you may want to define a secret variable and not have the variable exposed in your YAML. Or, you may need to manually set a variable value during the pipeline run.

You have two options for defining queue-time values. You can define a variable in the UI and select the option to **Let users override this value when running this pipeline** or you can use runtime parameters instead. If your variable is not a secret, the best practice is to use runtime parameters.

To set a variable at queue time, add a new variable within your pipeline and select the override option.

To allow a variable to be set at queue time, make sure the variable doesn't also appear in the `variables` block of a pipeline or job. If you define a variable in both the variables block of a YAML and in the UI, the value in the YAML will have priority.

Expansion of variables

YAML

When you set a variable with the same name in multiple scopes, the following precedence applies (highest precedence first).

1. Job level variable set in the YAML file
2. Stage level variable set in the YAML file
3. Pipeline level variable set in the YAML file
4. Variable set at queue time
5. Pipeline variable set in Pipeline settings UI

In the following example, the same variable `a` is set at the pipeline level and job level in YAML file. It's also set in a variable group `G`, and as a variable in the Pipeline settings UI.

YAML

```yaml
variables:
  a: 'pipeline yaml'

stages:
- stage: one
  displayName: one
  variables:
  - name: a
    value: 'stage yaml'
```

```yaml
  jobs:
  - job: A
    variables:
    - name: a
      value: 'job yaml'
    steps:
    - bash: echo $(a)        # This will be 'job yaml'
```

When you set a variable with the same name in the same scope, the last set value will take precedence.

YAML

```yaml
stages:
- stage: one
  displayName: Stage One
  variables:
    - name: a
      value: alpha
    - name: a
      value: beta
  jobs:
  - job: I
    displayName: Job I
    variables:
      - name: b
        value: uno
      - name: b
        value: dos
    steps:
    - script: echo $(a) #outputs beta
    - script: echo $(b) #outputs dos
```

Note

When you set a variable in the YAML file, don't define it in the web editor as settable at queue time. You can't currently change variables that are set in the YAML file at queue time. If you need a variable to be settable at queue time, don't set it in the YAML file.

Variables are expanded once when the run is started, and again at the beginning of each step. For example:

YAML

```yaml
jobs:
- job: A
  variables:
```

```
    a: 10
  steps:
  - bash: |
      echo $(a)            # This will be 10
      echo '##vso[task.setvariable variable=a]20'
      echo $(a)            # This will also be 10, since the expansion
of $(a) happens before the step
  - bash: echo $(a)        # This will be 20, since the variables are
expanded just before the step
```

There are two steps in the preceding example. The expansion of `$(a)` happens once at the beginning of the job, and once at the beginning of each of the two steps.

Because variables are expanded at the beginning of a job, you can't use them in a strategy. In the following example, you can't use the variable `a` to expand the job matrix, because the variable is only available at the beginning of each expanded job.

YAML

```yaml
jobs:
- job: A
  variables:
    a: 10
  strategy:
    matrix:
      x:
        some_variable: $(a)    # This does not work
```

If the variable `a` is an output variable from a previous job, then you can use it in a future job.

YAML

```yaml
- job: A
  steps:
  - powershell: echo "##vso[task.setvariable
variable=a;isOutput=true]10"
    name: a_step

# Map the variable into job B
- job: B
  dependsOn: A
  variables:
    some_variable: $[ dependencies.A.outputs['a_step.a'] ]
```

Recursive expansion

On the agent, variables referenced using `$( )` syntax are recursively expanded. For example:

YAML

```yaml
variables:
  myInner: someValue
  myOuter: $(myInner)

steps:
- script: echo $(myOuter)  # prints "someValue"
  displayName: Variable is $(myOuter)  # display name is "Variable is
someValue"
```

Use a variable group's secret and nonsecret variables in an Azure Pipeline

Article • 01/13/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

In this sample, use the Microsoft Azure DevOps CLI (azure-devops extension) to create an Azure Pipeline that accesses a variable group containing both secret and nonsecret variables.

This script demonstrates three operations:

- Defining a Azure Pipeline using YAML files
- Creating a variable group with nonsecret and secret variables for use in a pipeline
- Running the pipeline using Azure DevOps CLI, which also opens a web page for monitoring the pipeline run's processing and output

Prerequisites

- Use the Bash environment in Azure Cloud Shell. For more information, see Quickstart for Bash in Azure Cloud Shell.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.
- Sign up for Azure Pipelines to get an Azure DevOps organization.
- Create an Azure DevOps personal access token (PAT) for authentication.
- Use or create a GitHub repository to place the Azure Pipeline in.

- In the GitHub Marketplace, install the Azure Pipelines application so that Azure Pipelines can access your GitHub repository.

Sample script

First, save the following YAML file, which defines the Azure Pipeline, to *azure-pipelines.yml* in the root directory of your local repository. Then add and commit the file in GitHub, and push the file to the remote GitHub repository.

yml

```yml
parameters:
- name: image
  displayName: 'Pool image'
  default: ubuntu-latest
  values:
  - windows-latest
  - windows-latest
  - ubuntu-latest
  - ubuntu-latest
  - macOS-latest
  - macOS-latest
- name: test
  displayName: Run Tests?
  type: boolean
  default: false

variables:
- group: "Contoso Variable Group"
- name: va
  value: $[variables.a]
- name: vb
  value: $[variables.b]
- name: vcontososecret
  value: $[variables.contososecret]

trigger:
- main

pool:
  vmImage: ubuntu-latest

steps:
- script: |
    echo "Hello, world!"
    echo "Pool image: ${{ parameters.image }}"
    echo "Run tests? ${{ parameters.test }}"
  displayName: 'Show runtime parameter values'

- script: |
    echo "a=$(va)"
```

```
    echo "b=$(vb)"
    echo "contososecret=$(vcontososecret)"
    echo
    echo "Count up to the value of the variable group's nonsecret variable
*a*:"
    for number in {1..$(va)}
    do
        echo "$number"
    done
    echo "Count up to the value of the variable group's nonsecret variable
*b*:"
    for number in {1..$(vb)}
    do
        echo "$number"
    done
    echo "Count up to the value of the variable group's secret variable
*contososecret*:"
    for number in {1..$(vcontososecret)}
    do
        echo "$number"
    done
  displayName: 'Test variable group variables (secret and nonsecret)'
  env:
    SYSTEM_ACCESSTOKEN: $(System.AccessToken)
```

After you've published the YAML file in GitHub, replace the placeholders in the following Bash script, and then run the script.

Azure CLI

```
#!/bin/bash

# Provide variable definitions.
devopsToken="<azure-devops-personal-access-token>"
devopsOrg="https://dev.azure.com/<my-azure-devops-account-or-organization-
name>"
resourceGroupLocation="<resource-group-location-name-or-id>"
storageAccountLocation="<storage-account-location-name-or-id>"
pipelineName="<my-build>"
githubOrg="<my-github-organization-name>"
githubRepo="<my-github-repository-name>"
repoName="$githubOrg/$githubRepo"
repoType="github"
branch="main"

# Declare other variables.
uniqueId=$RANDOM
resourceGroupName="contoso-storage-rg$uniqueId"
storageAccountName="contosostoracct$uniqueId"  # needs to be lowercase
devopsProject="Contoso DevOps Project $uniqueId"
serviceConnectionName="Contoso Service Connection $uniqueId"
variableGroupName="Contoso Variable Group"
```

```
# Sign in to Azure CLI and follow the directions. May be unnecessary in some
environments.
echo "Sign in. (For Cloud Shell, provide the device login code.)"
az login

# Sign in using an Azure DevOps personal access token (PAT). May be
unnecessary in some environments.
echo "Sign in to Azure DevOps."
az devops login

# Create a resource group and a storage account.
az group create --name "$resourceGroupName" --location
"$resourceGroupLocation"
az storage account create --name "$storageAccountName" \
    --resource-group "$resourceGroupName" --location
"$storageAccountLocation"

# Set the environment variable used for Azure DevOps token authentication.
export AZURE_DEVOPS_EXT_GITHUB_PAT=$devopsToken

# Create the Azure DevOps project. Set the default organization and project.
projectId=$(az devops project create \
    --name "$devopsProject" --organization "$devopsOrg" --visibility public
--query id)
projectId=${projectId:1:-1}  # Just set to GUID; drop enclosing quotes.
az devops configure --defaults organization="$devopsOrg"
project="$devopsProject"
pipelineRunUrlPrefix="$devopsOrg/$projectId/_build/results?buildId="

# Create GitHub service connection (requires AZURE_DEVOPS_EXT_GITHUB_PAT to
be set).
githubServiceEndpointId=$(az devops service-endpoint github create \
    --name "$serviceConnectionName" --github-url
"https://www.github.com/$repoName" --query id)
githubServiceEndpointId=${githubServiceEndpointId:1:-1}  # Just set to GUID;
drop enclosing quotes.

# Create the pipeline.
pipelineId=$(az pipelines create \
    --name "$pipelineName" \
    --skip-first-run \
    --repository $repoName \
    --repository-type $repoType \
    --branch $branch \
    --service-connection $githubServiceEndpointId \
    --yml-path azure-pipelines.yml \
    --query id)

# Create a variable group with 2 non-secret variables and 1 secret variable.
# (contososecret < a < b). Then run the pipeline.
variableGroupId=$(az pipelines variable-group create \
    --name "$variableGroupName" --authorize true --variables a=12 b=29 --
query id)
az pipelines variable-group variable create \
    --group-id $variableGroupId --name contososecret --secret true --value
```

```
pipelineRunId1=$(az pipelines run --id $pipelineId --open --query id)
echo "Go to the pipeline run's web page to view the output results for the 1st run."
echo "If the web page doesn't automatically appear, go to:"
echo "    ${pipelineRunUrlPrefix}${pipelineRunId1}"
read -p "Press Enter to change the value of one of the variable group's nonsecret variables, then run again:"

# Change the value of one of the variable group's nonsecret variables.
az pipelines variable-group variable update \
    --group-id $variableGroupId --name a --value 22
pipelineRunId2=$(az pipelines run --id $pipelineId --open --query id)
echo "Go to the pipeline run's web page to view the output results for the 2nd run."
echo "If the web page doesn't automatically appear, go to:"
echo "    ${pipelineRunUrlPrefix}${pipelineRunId2}"
read -p "Press Enter to change the value of the variable group's secret variable, then run once more:"

# Change the value of the variable group's secret variable.
az pipelines variable-group variable update \
    --group-id $variableGroupId --name contososecret --value 35
pipelineRunId3=$(az pipelines run --id $pipelineId --open --query id)
echo "Go to the pipeline run's web page to view the output results for the 3rd run."
echo "If the web page doesn't automatically appear, go to:"
echo "    ${pipelineRunUrlPrefix}${pipelineRunId3}"
read -p "Press Enter to continue:"
```

Clean up resources

After the script sample has been run, the following commands can be used to remove the resource group and all resources associated with it.

Azure CLI

```
az pipelines variable-group delete --group-id $variableGroupId --yes
az pipelines delete --id $pipelineId --yes
az devops service-endpoint delete --id $githubServiceEndpointId --yes
az devops project delete --id $projectId --yes
export AZURE_DEVOPS_EXT_GITHUB_PAT=""
az storage account delete --name $storageAccountName --resource-group $resourceGroupName --yes
az group delete --name $resourceGroupName --yes
az devops configure --defaults organization="" project=""
```

Azure CLI references used in this article

- az devops configure
- az devops project create
- az devops project delete
- az devops service-endpoint github create
- az group create
- az group delete
- az login
- az pipelines create
- az pipelines delete
- az pipelines run
- az pipelines variable-group create
- az pipelines variable-group delete
- az pipelines variable-group variable create
- az pipelines variable-group variable update
- az storage account create
- az storage account delete

Use predefined variables

Article • 07/20/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Variables give you a convenient way to get key bits of data into various parts of your pipeline. This is a list of predefined variables that are available for your use. There may be a few other predefined variables, but they're mostly for internal use.

These variables are automatically set by the system and read-only. (The exceptions are Build.Clean and System.Debug.)

In YAML pipelines, you can reference predefined variables as environment variables. For example, the variable `Build.ArtifactStagingDirectory` becomes the variable `BUILD_ARTIFACTSTAGINGDIRECTORY`.

For classic pipelines, you can use release variables in your deploy tasks to share the common information (for example, Environment Name, Resource Group, etc.).

Learn more about working with variables.

Build.Clean

This is a deprecated variable that modifies how the build agent cleans up source. To learn how to clean up source, see Clean the local repo on the agent.

System.AccessToken

`System.AccessToken` is a special variable that carries the security token used by the running build.

YAML

In YAML, you must explicitly map `System.AccessToken` into the pipeline using a variable. You can do this at the step or task level:

YAML

```yaml
steps:
  - bash: echo This script could use $SYSTEM_ACCESSTOKEN
    env:
      SYSTEM_ACCESSTOKEN: $(System.AccessToken)
  - powershell: |
      Write-Host "This is a script that could use $env:SYSTEM_ACCESSTOKEN"
      Write-Host "$env:SYSTEM_ACCESSTOKEN = $(System.AccessToken)"
    env:
      SYSTEM_ACCESSTOKEN: $(System.AccessToken)
```

You can configure the default scope for `System.AccessToken` using build job authorization scope.

System.Debug

For more detailed logs to debug pipeline problems, define `System.Debug` and set it to `true`.

1. Edit your pipeline.
2. Select **Variables**.
3. Add a new variable with the name `System.Debug` and value `true`.

4. Save the new variable.

Setting `System.Debug` to `true` configures verbose logs for all runs. You can also configure verbose logs for a single run with the **Enable system diagnostics** checkbox.

You can also set `System.Debug` to `true` as a variable in a pipeline or template.

YAML

```yaml
variables:
  system.debug: 'true'
```

When `System.Debug` is set to `true`, an extra variable named `Agent.Diagnostic` is set to `true`. When `Agent.Diagnostic` is `true`, the agent collects more logs that can be used for troubleshooting network issues for self-hosted agents. For more information, see Network diagnostics for self-hosted agents.

> **Note**
>
> The `Agent.Diagnostic` variable is available with **Agent v2.200.0** and higher.

For more information, see Review logs to diagnose pipeline issues.

Agent variables (DevOps Services)

> **Note**
>
> You can use agent variables as environment variables in your scripts and as parameters in your build tasks. You cannot use them to customize the build number or to apply a version control label or tag.

Variable	Description
Agent.BuildDirectory	The local path on the agent where all folders for a given build pipeline are created. This variable has the same value as `Pipeline.Workspace`. For example: `/home/vsts/work/1`
Agent.ContainerMapping	A mapping from container resource names in YAML to their Docker IDs at runtime. For example: `{ "one_container": { "id": "bdbb357d73a0bd3550a1a5b778b62a4c88ed2051c7802a0659f1ff6e76910190" }, "another_container": { "id": "82652975109ec494876a8ccbb875459c945982952e0a72ad74c91216707162bb" } }`
Agent.HomeDirectory	The directory the agent is installed into. This contains the agent software. For example: `c:\agent`.
Agent.Id	The ID of the agent.
Agent.JobName	The name of the running job. This will usually be "Job" or "__default", but in multi-config scenarios, will be the configuration.
Agent.JobStatus	The status of the build. • `Canceled` • `Failed` • `Succeeded` • `SucceededWithIssues` (partially successful) The environment variable should be referenced as `AGENT_JOBSTATUS`. The older `agent.jobstatus` is available for backwards compatibility.

Agent.MachineName	The name of the machine on which the agent is installed.
Agent.Name	The name of the agent that is registered with the pool. If you're using a self-hosted agent, then this name is specified by you. See agents.
Agent.OS	The operating system of the agent host. Valid values are: • `Windows_NT` • `Darwin` • `Linux` If you're running in a container, the agent host and container may be running different operating systems.
Agent.OSArchitecture	The operating system processor architecture of the agent host. Valid values are: • `X86` • `X64` • `ARM`
Agent.TempDirectory	A temporary folder that is cleaned after each pipeline job. This directory is used by tasks such as .NET Core CLI task to hold temporary items like test results before they're published. For example: `/home/vsts/work/_temp` for Ubuntu
Agent.ToolsDirectory	The directory used by tasks such as Node Tool Installer and Use Python Version to switch between multiple versions of a tool. These tasks add tools from this directory to `PATH` so that subsequent build steps can use them. Learn about managing this directory on a self-hosted agent.
Agent.WorkFolder	The working directory for this agent. For example: `c:\agent_work`. Note: This directory isn't guaranteed to be writable by pipeline tasks (for example, when mapped into a container)

Build variables (DevOps Services)

Variable	Description	Available in templates?
Build.ArtifactStagingDirectory	The local path on the agent where any artifacts are copied to before being pushed to their destination. For example: `c:\agent_work\1\a` A typical way to use this folder is to publish your build artifacts with the Copy files and Publish build artifacts tasks. Note: Build.ArtifactStagingDirectory and Build.StagingDirectory are interchangeable. This directory is	No

	purged before each new build, so you don't have to clean it up yourself. See Artifacts in Azure Pipelines. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	
Build.BuildId	The ID of the record for the completed build.	No
Build.BuildNumber	The name of the completed build, also known as the run number. You can specify what is included in this value. A typical use of this variable is to make it part of the label format, which you specify on the repository tab. Note: This value can contain whitespace or other invalid label characters. In these cases, the label format fails. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No
Build.BuildUri	The URI for the build. For example: `vstfs:///Build/Build/1430`. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No
Build.BinariesDirectory	The local path on the agent you can use as an output folder for compiled binaries. By default, new build pipelines aren't set up to clean this directory. You can define your build to clean it up on the Repository tab. For example: `c:\agent_work\1\b`. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No
Build.ContainerId	The ID of the container for your artifact. When you upload an artifact in your pipeline, it's added to a container that is specific for that particular artifact.	No

Build.CronSchedule.DisplayName	The `displayName` of the cron schedule that triggered the pipeline run. This variable is only set if the pipeline run is triggered by a YAML scheduled trigger. For more information, see schedules.cron definition - Build.CronSchedule.DisplayName variable	Yes
Build.DefinitionName	The name of the build pipeline. Note: This value can contain whitespace or other invalid label characters. In these cases, the label format fails.	Yes
Build.DefinitionVersion	The version of the build pipeline.	Yes
Build.QueuedBy	See "How are the identity variables set?". Note: This value can contain whitespace or other invalid label characters. In these cases, the label format fails.	Yes
Build.QueuedById	See "How are the identity variables set?".	Yes
Build.Reason	The event that caused the build to run. • `Manual`: A user manually queued the build. • `IndividualCI`: **Continuous integration (CI)** triggered by a Git push or a TFVC check-in. • `BatchedCI`: **Continuous integration (CI)** triggered by a Git push or a TFVC check-in, and the **Batch changes** was selected. • `Schedule`: **Scheduled** trigger. • `ValidateShelveset`: A user manually queued the build of a specific TFVC shelveset. • `CheckInShelveset`: **Gated check-in** trigger. • `PullRequest`: The build was triggered by a Git branch policy that requires a build. • `BuildCompletion`: The build was triggered by another build • `ResourceTrigger`: The build was triggered by a resource trigger or it was triggered by another build. See Build pipeline triggers, Improve code quality with branch policies.	Yes
Build.Repository.Clean	The value you've selected for **Clean** in the source repository settings. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No
Build.Repository.LocalPath	The local path on the agent where your source code files are downloaded. For example: `c:\agent_work\1\s` By default, new build pipelines update only the changed	No

	files. You can modify how files are downloaded on the Repository tab. Important note: If you check out only one Git repository, this path is the exact path to the code. If you check out multiple repositories, the behavior is as follows (and might differ from the value of the Build.SourcesDirectory variable): • If the checkout step for the self (primary) repository has no custom checkout path defined, or the checkout path is the multi-checkout default path `$(Pipeline.Workspace)/s/<RepoName>` for the self repository, the value of this variable reverts to its default value, which is `$(Pipeline.Workspace)/s`. • If the checkout step for the self (primary) repository does have a custom checkout path defined (and it's not its multi-checkout default path), this variable contains the exact path to the self repository. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	
Build.Repository.ID	The unique identifier of the repository. This won't change, even if the name of the repository does. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No
Build.Repository.Name	The name of the triggering repository. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No
Build.Repository.Provider	The type of the triggering repository. • `TfsGit`: TFS Git repository • `TfsVersionControl`: Team Foundation Version Control • `Git`: Git repository hosted on an external server • `GitHub` • `Svn`: Subversion This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No
Build.Repository.Tfvc.Workspace	Defined if your repository is Team Foundation Version Control. The name of the TFVC workspace used by the build agent.	No

	For example, if the Agent.BuildDirectory is `c:\agent_work\12` and the Agent.Id is `8`, the workspace name could be: `ws_12_8` This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	
Build.Repository.Uri	The URL for the triggering repository. For example: • Git: `https://dev.azure.com/fabrikamfiber/_git/Scripts` • TFVC: `https://dev.azure.com/fabrikamfiber/` This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No
Build.RequestedFor	See "How are the identity variables set?". Note: This value can contain whitespace or other invalid label characters. In these cases, the label format fails.	Yes
Build.RequestedForEmail	See "How are the identity variables set?".	Yes
Build.RequestedForId	See "How are the identity variables set?".	Yes
Build.SourceBranch	The branch of the triggering repo the build was queued for. Some examples: • Git repo branch: `refs/heads/main` • Git repo pull request: `refs/pull/1/merge` • TFVC repo branch: `$/teamproject/main` • TFVC repo gated check-in: `Gated_2016-06-06_05.20.51.4369;username@live.com` • TFVC repo shelveset build: `myshelveset;username@live.com` • When your pipeline is triggered by a tag: `refs/tags/your-tag-name` When you use this variable in your build number format, the forward slash characters (`/`) are replaced with underscore characters `_`). Note: In TFVC, if you're running a gated check-in build or manually building a shelveset, you can't use this variable in your build number format.	Yes
Build.SourceBranchName	The name of the branch in the triggering repo the build was queued for. • Git repo branch, pull request, or tag: The last path segment in the ref. For example, in `refs/heads/main` this value is `main`. In `refs/heads/feature/tools` this	Yes

	value is `tools`. In `refs/tags/your-tag-name` this value is `your-tag-name`. • TFVC repo branch: The last path segment in the root server path for the workspace. For example, in `$/teamproject/main` this value is `main`. • TFVC repo gated check-in or shelveset build is the name of the shelveset. For example, `Gated_2016-06-06_05.20.51.4369;username@live.com` or `myshelveset;username@live.com`. Note: In TFVC, if you're running a gated check-in build or manually building a shelveset, you can't use this variable in your build number format.	
Build.SourcesDirectory	The local path on the agent where your source code files are downloaded. For example: `c:\agent_work\1\s` By default, new build pipelines update only the changed files. Important note: If you check out only one Git repository, this path is the exact path to the code. If you check out multiple repositories, it reverts to its default value, which is `$(Pipeline.Workspace)/s`, even if the self (primary) repository is checked out to a custom path different from its multi-checkout default path `$(Pipeline.Workspace)/s/<RepoName>` (in this respect, the variable differs from the behavior of the Build.Repository.LocalPath variable). This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No
Build.SourceVersion	The latest version control change of the triggering repo that is included in this build. • Git: The commit ID. • TFVC: the changeset. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	Yes
Build.SourceVersionMessage	The comment of the commit or changeset for the triggering repo. We truncate the message to the first line or 200 characters, whichever is shorter. The `Build.SourceVersionMessage` corresponds to the message on `Build.SourceVersion` commit. The `Build.SourceVersion` commit for a PR build is the merge commit (not the commit on the source branch). This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a	No

	build task, but not as part of the build number or as a version control tag. Also, this variable is only available on the step level and is not available in the job or stage levels (that is, the message isn't extracted until the job starts and the code is checked out). Note: This variable is available in TFS 2015.4. ⓘ **Note** The **Build.SourceVersionMessage** variable does not work with classic build pipelines in Bitbucket repositories when **Batch changes while a build is in progress** is enabled.	
Build.StagingDirectory	The local path on the agent where any artifacts are copied to before being pushed to their destination. For example: `c:\agent_work\1\a` A typical way to use this folder is to publish your build artifacts with the Copy files and Publish build artifacts tasks. Note: Build.ArtifactStagingDirectory and Build.StagingDirectory are interchangeable. This directory is purged before each new build, so you don't have to clean it up yourself. See Artifacts in Azure Pipelines. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No
Build.Repository.Git.SubmoduleCheckout	The value you've selected for **Checkout submodules** on the repository tab. With multiple repos checked out, this value tracks the triggering repository's setting. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No
Build.SourceTfvcShelveset	Defined if your repository is Team Foundation Version Control. If you're running a gated build or a shelveset build, this is set to the name of the shelveset you're building. Note: This variable yields a value that is invalid for build use in a build number format.	No

Build.TriggeredBy.BuildId	If the build was triggered by another build, then this variable is set to the BuildID of the triggering build. In Classic pipelines, this variable is triggered by a build completion trigger. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag. If you're triggering a YAML pipeline using `resources`, you should use the resources variables instead.	No
Build.TriggeredBy.DefinitionId	If the build was triggered by another build, then this variable is set to the DefinitionID of the triggering build. In Classic pipelines, this variable is triggered by a build completion trigger. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag. If you're triggering a YAML pipeline using `resources`, you should use the resources variables instead.	No
Build.TriggeredBy.DefinitionName	If the build was triggered by another build, then this variable is set to the name of the triggering build pipeline. In Classic pipelines, this variable is triggered by a build completion trigger. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag. If you're triggering a YAML pipeline using `resources`, you should use the resources variables instead.	No
Build.TriggeredBy.BuildNumber	If the build was triggered by another build, then this variable is set to the number of the triggering build. In Classic pipelines, this variable is triggered by a build completion trigger. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag. If you're triggering a YAML pipeline using `resources`, you should use the resources variables instead.	No
Build.TriggeredBy.ProjectID	If the build was triggered by another build, then this variable is set to ID of the project that contains the triggering build. In Classic pipelines, this variable is	No

	triggered by a build completion trigger. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag. If you're triggering a YAML pipeline using `resources`, you should use the resources variables instead.	
Common.TestResultsDirectory	The local path on the agent where the test results are created. For example: `c:\agent_work\1\TestResults` This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No

Pipeline variables (DevOps Services)

Variable	Description
Pipeline.Workspace	Workspace directory for a particular pipeline. This variable has the same value as `Agent.BuildDirectory`. For example, `/home/vsts/work/1`.

> **Tip**
>
> If you are using classic release pipelines, you can use **classic releases and artifacts variables** to store and access data throughout your pipeline.

Deployment job variables (DevOps Services)

These variables are scoped to a specific Deployment job and will be resolved only at job execution time.

Variable	Description
Environment.Name	Name of the environment targeted in the deployment job to run the deployment steps and record the deployment history. For example, `smarthotel-dev`.
Environment.Id	ID of the environment targeted in the deployment job. For example, `10`.
Environment.ResourceName	Name of the specific resource within the environment targeted in the deployment job to run the deployment steps and record the deployment history. For example, `bookings` which is a Kubernetes namespace that has been added as a resource to the environment `smarthotel-dev`.

Environment.ResourceId	ID of the specific resource within the environment targeted in the deployment job to run the deployment steps. For example, `4`.
Strategy.Name	The name of the deployment strategy: `canary`, `runOnce`, or `rolling`.
Strategy.CycleName	The current cycle name in a deployment. Options are `PreIteration`, `Iteration`, or `PostIteration`.

System variables (DevOps Services)

Variable	Description	Available in templates?
System.AccessToken	Use the OAuth token to access the REST API. Use System.AccessToken from YAML scripts. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	Yes
System.CollectionId	The GUID of the TFS collection or Azure DevOps organization.	Yes
System.CollectionUri	The URI of the TFS collection or Azure DevOps organization. For example: `https://dev.azure.com/fabrikamfiber/`.	Yes
System.DefaultWorkingDirectory	The local path on the agent where your source code files are downloaded. For example: `c:\agent_work\1\s` By default, new build pipelines update only the changed files. You can modify how files are downloaded on the Repository tab. This variable is agent-scoped. It can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	Yes
System.DefinitionId	The ID of the build pipeline.	Yes
System.HostType	Set to `build` if the pipeline is a build. For a release, the values are `deployment` for a Deployment group job, `gates` during evaluation of gates, and `release` for other (Agent and Agentless) jobs.	Yes
System.JobAttempt	Set to 1 the first time this job is attempted, and increments every time the job is retried.	No

System.JobDisplayName	The human-readable name given to a job.	No
System.JobId	A unique identifier for a single attempt of a single job. The value is unique to the current pipeline.	No
System.JobName	The name of the job, typically used for expressing dependencies and accessing output variables.	No
System.PhaseAttempt	Set to 1 the first time this phase is attempted, and increments every time the job is retried. **Note:** "Phase" is a mostly redundant concept, which represents the design-time for a job (whereas job was the runtime version of a phase). We've mostly removed the concept of "phase" from Azure Pipelines. Matrix and multi-config jobs are the only place where "phase" is still distinct from "job." One phase can instantiate multiple jobs, which differ only in their inputs.	No
System.PhaseDisplayName	The human-readable name given to a phase.	No
System.PhaseName	A string-based identifier for a job, typically used for expressing dependencies and accessing output variables.	No
System.PlanId	A string-based identifier for a single pipeline run.	No
System.PullRequest.IsFork	If the pull request is from a fork of the repository, this variable is set to `True`. Otherwise, it's set to `False`.	Yes
System.PullRequest.PullRequestId	The ID of the pull request that caused this build. For example: `17`. (This variable is initialized only if the build ran because of a Git PR affected by a branch policy).	No
System.PullRequest.PullRequestNumber	The number of the pull request that caused this build. This variable is populated for pull requests from GitHub that have a different pull request ID and pull request number. This variable is only available in a YAML pipeline if the PR is affected by a branch policy.	No
System.PullRequest.targetBranchName	The name of the target branch for a pull request. This variable can be used in a pipeline to conditionally execute tasks or steps based on the target branch of the pull request. For example, you might want to trigger a different set of tests or code analysis tools depending on the branch that the changes are being merged into.	No
System.PullRequest.SourceBranch	The branch that is being reviewed in a pull request. For example: `refs/heads/users/raisa/new-feature` for Azure Repos. (This variable is initialized only if the build ran because of a Git PR affected by a	No

	branch policy). This variable is only available in a YAML pipeline if the PR is affected by a branch policy.	
System.PullRequest.SourceRepositoryURI	The URL to the repo that contains the pull request. For example: `https://dev.azure.com/ouraccount/_git/OurProject`.	No
System.PullRequest.TargetBranch	The branch that is the target of a pull request. For example: `refs/heads/main` when your repository is in Azure Repos and `main` when your repository is in GitHub. This variable is initialized only if the build ran because of a Git PR affected by a branch policy. This variable is only available in a YAML pipeline if the PR is affected by a branch policy.	No
System.StageAttempt	Set to 1 the first time this stage is attempted, and increments every time the job is retried.	No
System.StageDisplayName	The human-readable name given to a stage.	No
System.StageName	A string-based identifier for a stage, typically used for expressing dependencies and accessing output variables.	No
System.TeamFoundationCollectionUri	The URI of the TFS collection or Azure DevOps organization. For example: `https://dev.azure.com/fabrikamfiber/`. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	Yes
System.TeamProject	The name of the project that contains this build.	Yes
System.TeamProjectId	The ID of the project that this build belongs to.	Yes
System.TimelineId	A string-based identifier for the execution details and logs of a single pipeline run.	No
TF_BUILD	Set to `True` if the script is being run by a build task. This variable is agent-scoped, and can be used as an environment variable in a script and as a parameter in a build task, but not as part of the build number or as a version control tag.	No

Checks variables (DevOps Services)

Variable	Description
Checks.StageAttempt	Set to 1 the first time this stage is attempted, and increments every time the stage is retried.

This variable can only be used within an approval or check for an environment. For example, you could use `$(Checks.StageAttempt)` within an Invoke REST API check.

URL suffix and parameters

?attempt=$(Checks.StageAttempt)

How are the identity variables set?

The value depends on what caused the build and are specific to Azure Repos repositories.

If the build is triggered...	Then the Build.QueuedBy and Build.QueuedById values are based on...	Then the Build.RequestedFor and Build.RequestedForId values are based on...
In Git or by the Continuous integration (CI) triggers	The system identity, for example: `[DefaultCollection]\Project Collection Service Accounts`	The person who pushed or checked in the changes.
In Git or by a branch policy build.	The system identity, for example: `[DefaultCollection]\Project Collection Service Accounts`	The person who checked in the changes.
In TFVC by a gated check-in trigger	The person who checked in the changes.	The person who checked in the changes.
In Git or TFVC by the Scheduled triggers	The system identity, for example: `[DefaultCollection]\Project Collection Service Accounts`	The system identity, for example: `[DefaultCollection]\Project Collection Service Accounts`
Because you clicked the **Queue build** button	You	You

Set variables in scripts

Article • 05/10/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

When you use PowerShell and Bash scripts in your pipelines, it's often useful to be able to set variables that you can then use in future tasks. Newly set variables aren't available in the same task.

Scripts are great for when you want to do something that isn't supported by a task like calling a custom REST API and parsing the response.

You'll use the `task.setvariable` logging command to set variables in PowerShell and Bash scripts.

> **Note**
>
> Deployment jobs use a different syntax for output variables. To learn more about support for output variables in deployment jobs, see **Deployment jobs**.

To use a variable with a condition in a pipeline, see Specify conditions.

About `task.setvariable`

When you add a variable with `task.setvariable`, the following tasks can use the variable using macro syntax `$(myVar)`. The variable will only be available to tasks in the same job by default. If you add the parameter `isoutput`, the syntax to call your variable changes. See Set an output variable for use in the same job.

Bash

Set the variable `myVar` with the value `foo`.

YAML

```yaml
- bash: |
    echo "##vso[task.setvariable variable=myVar;]foo"
```

Read the variable `myVar`:

YAML

```yaml
- bash: |
    echo "You can use macro syntax for variables: $(myVar)"
```

Set variable properties

The `task.setvariable` command includes properties for setting a variable as secret, as an output variable, and as read only. The available properties include:

- `variable` = variable name (Required)
- `issecret` = boolean (Optional, defaults to false)
- `isoutput` = boolean (Optional, defaults to false)
- `isreadonly` = boolean (Optional, defaults to false)

To use the variable in the next stage, set the `isoutput` property to `true`. To reference a variable with the `isoutput` set to true, you'll include the task name. For example, `$(TaskName.myVar)`.

When you set a variable as read only, it can't be overwritten by downstream tasks. Set `isreadonly` to `true`. Setting a variable as read only enhances security by making that variable immutable.

Set a variable as secret

When `issecret` is set to true, the value of the variable will be saved as secret and masked out from log.

Bash

Set the secret variable `mySecretVal`.

YAML

```yaml
- bash: |
    echo "##vso[task.setvariable
variable=mySecretVal;issecret=true]secretvalue"
```

Get the secret variable `mySecretVal`.

YAML

```
- bash: |
    echo "##vso[task.setvariable
variable=mySecretVal;issecret=true]secretvalue"
- bash: |
    echo $(mySecretVal)
```

Secret variable output in bash.

```
Generating script.
Script contents:
echo ***
```

Levels of output variables

There are four different types of output variables with distinct syntaxes:

- Output variables set in the same job without the isoutput parameter. To reference these variables, you'll use macro syntax. Example: `$(myVar)`.
- Output variables set in the same job with the isoutput parameter. To reference these variables, you'll include the task name. Example: `$(myTask.myVar)`.
- Output variables set in a future job. To reference these variables, you'll reference the variable in the `variables` section with `dependency` syntax.
- Output variables set in future stages. To reference these variables, you'll reference the variable in the `variables` section with `stageDependencies` syntax.

Set an output variable for use in the same job

When you use an output variable in the same job, you don't have to use the `isoutput` property. By default, the variable will be available to downstream steps within the same job. However, if you do add the `isoutput` property, you'll need to reference the variable with the task name.

Bash

The script here sets the same-job output variable `myJobVar` without specifying `isoutput` and sets `myOutputJobVar` with `isoutput=true`.

YAML

```
jobs:
- job: A
```

```
  steps:
  - bash: |
      echo "##vso[task.setvariable variable=myJobVar]this is the same
job"
  - bash: |
      echo "##vso[task.setvariable
variable=myOutputJobVar;isoutput=true]this is the same job too"
    name: setOutput
```

This script gets the same-job variables `myJobVar` and `myOutputJobVar`. Notice that the syntax changes for referencing an output variable once `isoutput=true` is added.

YAML

```
jobs:
- job: A
  steps:
  - bash: |
      echo "##vso[task.setvariable variable=myJobVar]this is the same
job"
  - bash: |
      echo "##vso[task.setvariable
variable=myOutputJobVar;isoutput=true]this is the same job too"
    name: setOutput
  - bash: |
      echo $(myJobVar)
  - bash: |
      echo $(setOutput.myOutputJobVar)
```

Set an output variable for use in future jobs

When you use output variables across jobs, you'll reference them with `dependencies`. The syntax for accessing an output variable in a future job or stage varies based on the relationship between the setter and consumer of the variable. Learn about each case in dependencies.

Bash

First, set the output variable `myOutputVar`.

YAML

```
jobs:
- job: A
  steps:
  - bash: |
```

```
      echo "##vso[task.setvariable
variable=myOutputVar;isoutput=true]this is from job A"
    name: passOutput
```

Next, access `myOutputVar` in a future job and output the variable as `myVarFromJobA`. To use `dependencies`, you need to set the `dependsOn` property on the future job using the name of the past job in which the output variable was set.

YAML

```yaml
jobs:
- job: A
  steps:
  - bash: |
      echo "##vso[task.setvariable
variable=myOutputVar;isoutput=true]this is from job A"
    name: passOutput
- job: B
  dependsOn: A
  variables:
    myVarFromJobA: $[ dependencies.A.outputs['passOutput.myOutputVar'] ]
  steps:
  - bash: |
      echo $(myVarFromJobA)
```

Set an output variable for use in future stages

Output variables can be used across stages in pipelines. You can use output variables to pass useful information, such as the ID of a generated output, from one stage to the next.

When you set a variable with the `isoutput` property, you can reference that variable in later stages with the task name and the `stageDependencies` syntax. Learn more about dependencies.

Bash

First, set the output variable `myStageVal`.

YAML

```yaml
steps:
  - bash: echo "##vso[task.setvariable
```

```
variable=myStageVal;isOutput=true]this is a stage output variable"
    name: MyOutputVar
```

Then, in a future stage, map the output variable `myStageVal` to a stage, job, or task-scoped variable as, for example, `myStageAVar`. Note the mapping syntax uses a runtime expression `$[]` and traces the path from `stageDependencies` to the output variable using both the stage name (`A`) and the job name (`A1`) to fully qualify the variable.

YAML

```yaml
stages:
- stage: A
  jobs:
  - job: A1
    steps:
     - bash: echo "##vso[task.setvariable
variable=myStageVal;isOutput=true]this is a stage output variable"
       name: MyOutputVar
- stage: B
  dependsOn: A
  jobs:
  - job: B1
    variables:
      myStageAVar:
$[stageDependencies.A.A1.outputs['MyOutputVar.myStageVal']]
    steps:
      - bash: echo $(myStageAVar)
```

In case your value contains newlines, you can escape them and the agent will automatically unescape it:

YAML

```yaml
steps:
- bash: |
    escape_data() {
      local data=$1
      data="${data//'%'/'%AZP25'}"
      data="${data//$'\n'/'%0A'}"
      data="${data//$'\r'/'%0D'}"
      echo "$data"
    }
    echo "##vso[task.setvariable
variable=myStageVal;isOutput=true]$(escape_data $'foo\nbar')"
  name: MyOutputVar
```

FAQ

My output variable isn't rendering. What is going wrong?

There are a few reasons why your output variable may not appear.

- Output variables set with `isoutput` aren't available in the same job and instead are only available in downstream jobs.
- Depending on what variable syntax you use, a variable that sets an output variable's value may not be available at runtime. For example, variables with macro syntax (`$(var)`) get processed before a task runs. In contrast, variables with template syntax are processed at runtime (`$[variables.var]`). You'll usually want to use runtime syntax when setting output variables. For more information on variable syntax, see Define variables.
- There may be extra spaces within your expression. If your variable isn't rendering, check for extra spaces surrounding `isOutput=true`.

Set secret variables

Article • 11/01/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Secret variables are encrypted variables that you can use in pipelines without exposing their value. Secret variables can be used for private information like passwords, IDs, and other identifying data that you wouldn't want exposed in a pipeline. Secret variables are encrypted at rest with a 2048-bit RSA key and are available on the agent for tasks and scripts to use.

The recommended ways to set secret variables are in the UI, in a variable group, and in a variable group from Azure Key Vault. You can also set secret variables in a script with a logging command but this isn't recommended since anyone who can access your pipeline can also see the secret.

Secret variables set in the pipeline settings UI for a pipeline are scoped to the pipeline where they're set. You can use variable groups to share secret variables across pipelines.

Secret variable in the UI

You can set secret variables in the pipeline editor when you're editing an individual pipeline. You'll encrypt and make a pipeline variable secret by selecting the lock icon.

You set secret variables the same way for YAML and Classic.

To set secrets in the web interface, follow these steps:

1. Go to the **Pipelines** page, select the appropriate pipeline, and then select **Edit**.
2. Locate the **Variables** for this pipeline.
3. Add or update the variable.
4. Select the 🔒 lock icon to store the variable in an encrypted manner.
5. 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. Be careful about who has access to alter your pipeline.

> **ⓘ Important**
>
> We make an effort to mask secrets from appearing in Azure Pipelines output, but you still need to take precautions. Never echo secrets as output. Some operating

systems log command line arguments. Never pass secrets on the command line. Instead, we suggest that you map your secrets into environment variables.

We never mask substrings of secrets. If, for example, "abc123" is set as a secret, "abc" isn't masked from the logs. This is to avoid masking secrets at too granular of a level, making the logs unreadable. For this reason, secrets should not contain structured data. If, for example, "{ "foo": "bar" }" is set as a secret, "bar" isn't masked from the logs.

Unlike a normal variable, they are not automatically decrypted into environment variables for scripts. You need to explicitly map secret variables.

Use a secret variable in the UI

YAML

You'll need to map secret variable as environment variables to reference them in YAML pipelines. In this example, there are two secret variables defined in the UI, `SecretOne` and `SecretTwo`. The value of `SecretOne` is `foo` and the value of `SecretTwo` is `bar`.

yml

```yml
steps:
- powershell: |
      Write-Host "My first secret variable is $env:FOO_ONE"
      $env:FOO_ONE -eq "foo"
  env:
    FOO_ONE: $(SecretOne)
- bash: |
    echo "My second secret variable: $FOO_TWO"
    if [ "$FOO_TWO" = "bar" ]; then
        echo "Strings are equal."
    else
        echo "Strings are not equal."
    fi
  env:
    FOO_TWO: $(SecretTwo)
```

The pipeline outputs:

```
My first secret variable is ***
True
```

```
My second secret variable: ***
Strings are equal.
```

> **Note**
>
> Azure Pipelines makes an effort to mask secrets when emitting data to pipeline logs, so you may see additional variables and data masked in output and logs that are not set as secrets.

For a more detailed example, see Define variables.

Set a secret variable in a variable group

You can add secrets to a variable group or link secrets from an existing Azure Key Vault.

Create new variable groups

1. Select **Pipelines** > **Library** > **+ Variable group**.

2. Enter a name and description for the group.

3. Optional: Move the toggle to link secrets from an Azure key vault as variables. For more information, see Use Azure Key Vault secrets.

4. Enter the name and value for each variable to include in the group, choosing **+ Add** for each one.

5. To make your variable secure, choose the "lock" icon at the end of the row.

6. When you're finished adding variables, select **Save**.

Variable groups follow the library security model.

Link secrets from an Azure key vault

Link an existing Azure key vault to a variable group and map selective vault secrets to the variable group.

1. In the **Variable groups** page, enable **Link secrets from an Azure key vault as variables**. You'll need an existing key vault containing your secrets. Create a key vault using the Azure portal.

2. Specify your Azure subscription end point and the name of the vault containing your secrets.

Ensure the Azure service connection has at least **Get** and **List** management permissions on the vault for secrets. Enable Azure Pipelines to set these permissions by choosing **Authorize** next to the vault name. Or, set the permissions manually in the Azure portal:

a. Open **Settings** for the vault, and then choose **Access policies** > **Add new**.
b. Select **Select principal** and then choose the service principal for your client account.
c. Select **Secret permissions** and ensure that **Get** and **List** have check marks.
d. Select **OK** to save the changes.

3. On the **Variable groups** page, select **+ Add** to select specific secrets from your vault for mapping to this variable group.

Manage key vault secrets

See the following list of helpful tips for managing secrets.

- Only the secret *names* get mapped to the variable group, not the secret values. The latest secret value, fetched from the vault, is used in the pipeline run that's linked to the variable group.
- Any change made to *existing* secrets in the key vault is automatically available to all the pipelines the variable group's used in.
- When new secrets get added to or deleted from the vault, the associated variable groups aren't automatically updated. The secrets included in the variable group must be explicitly updated so the pipelines that are using the variable group get executed correctly.
- Azure Key Vault supports storing and managing cryptographic keys and secrets in Azure. Currently, Azure Pipelines variable group integration supports mapping only secrets from the Azure key vault. Cryptographic keys and certificates aren't supported.

Use the Azure Key Vault task

You can use the Azure Key Vault task to include secrets in your pipeline. This task allows the pipeline to connect to your Azure Key Vault and retrieve secrets to use as pipeline variables.

1. In the pipeline editor, select **Show assistant** to expand the assistant panel.

2. Search for `vault` and select the Azure Key Vault task.

The **Make secrets available to whole job** option isn't currently supported in Azure DevOps Server 2019 and 2020.

To learn more about the Azure Key Vault task, see Use Azure Key Vault secrets in Azure Pipelines.

Set secret variable in a script with logging commands

You can use the `task.setvariable` logging command to set variables in PowerShell and Bash scripts. This is the least secure way to work with secret variables but can be useful for debugging. The recommended ways to set secret variables are in the UI, in a variable group, and in a variable group from Azure Key Vault.

To set a variable as a script with a logging command, you'll need to pass the `issecret` flag.

When `issecret` is set to true, the value of the variable will be saved as secret and masked out from logs.

> **Note**

Azure Pipelines makes an effort to mask secrets when emitting data to pipeline logs, so you may see additional variables and data masked in output and logs that are not set as secrets.

Bash

Set the secret variable `mySecretVal`.

YAML

```yaml
- bash: |
    echo "##vso[task.setvariable variable=mySecretVal;issecret=true]secretvalue"
```

Get the secret variable `mySecretVal`.

YAML

```yaml
- bash: |
    echo "##vso[task.setvariable variable=mySecretVal;issecret=true]secretvalue"
- bash: |
    echo $(mySecretVal)
```

Secret variable output in bash.

```
Generating script.
Script contents:
echo ***
```

Learn more about setting and using variables in scripts.

Related articles

- Define variables
- Use variables in a variable group
- Use predefined variables
- Set variables in scripts

Runtime parameters

Article • 07/03/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Runtime parameters let you have more control over what values can be passed to a pipeline. With runtime parameters you can:

- Supply different values to scripts and tasks at runtime
- Control parameter types, allowed ranges, and defaults
- Dynamically select jobs and stages with template expressions

You can specify parameters in templates and in the pipeline. Parameters have data types such as number and string, and they can be restricted to a subset of values. The `parameters` section in a YAML defines what parameters are available.

Parameters are only available at template parsing time. Parameters are expanded just before the pipeline runs so that values surrounded by `${{ }}` are replaced with parameter values. Use variables if you need your values to be more widely available during your pipeline run.

> **Note**
>
> This guidance does not apply to classic pipelines. For parameters in classic pipelines, see **Process parameters (classic)**.

Parameters must contain a name and data type. Parameters can't be optional. A default value needs to be assigned in your YAML file or when you run your pipeline. If you don't assign a default value or set `default` to `false`, the first available value is used.

Use templateContext to pass extra properties to stages, steps, and jobs that are used as parameters in a template.

Use parameters in pipelines

Set runtime parameters at the beginning of a YAML.

This example pipeline includes an `image` parameter with three hosted agents as `string` options. In the jobs section, the `pool` value specifies the agent from the parameter used to run the job. The `trigger` is set to none so that you can select the value of `image` when you manually trigger your pipeline to run.

YAML

```yaml
parameters:
- name: image
  displayName: Pool Image
  type: string
  default: ubuntu-latest
  values:
  - windows-latest
  - ubuntu-latest
  - macOS-latest

trigger: none

jobs:
- job: build
  displayName: build
  pool:
    vmImage: ${{ parameters.image }}
  steps:
  - script: echo building $(Build.BuildNumber) with ${{ parameters.image }}
```

When the pipeline runs, you select the Pool Image. If you don't make a selection, the default option, `ubuntu-latest` gets used.

Use conditionals with parameters

You can also use parameters as part of conditional logic. With conditionals, part of a YAML runs if it meets the `if` criteria.

Use parameters to determine what steps run

This pipeline adds a second boolean parameter, `test`, which can be used to control whether or not to run tests in the pipeline. When the value of `test` is true, the step that outputs *Running all the tests* runs.

YAML

```yaml
parameters:
- name: image
  displayName: Pool Image
  values:
  - windows-latest
  - ubuntu-latest
  - macOS-latest
- name: test
  displayName: Run Tests?
  type: boolean
  default: false

trigger: none

jobs:
- job: build
  displayName: Build and Test
  pool:
    vmImage: ${{ parameters.image }}
  steps:
  - script: echo building $(Build.BuildNumber)
  - ${{ if eq(parameters.test, true) }}:
    - script: echo "Running all the tests"
```

Use parameters to set what configuration is used

You can also use parameters to set which job runs. In this example, different architectures build depending on the value of `config` parameter, which is a `string` type. By default, both the `x86` and `x64` architectures build.

YAML

```yaml
parameters:
- name: configs
  type: string
  default: 'x86,x64'

trigger: none

jobs:
- ${{ if contains(parameters.configs, 'x86') }}:
  - job: x86
    steps:
```

```
    - script: echo Building x86...
- ${{ if contains(parameters.configs, 'x64') }}:
  - job: x64
    steps:
    - script: echo Building x64...
- ${{ if contains(parameters.configs, 'arm') }}:
  - job: arm
    steps:
    - script: echo Building arm...
```

Selectively exclude a stage

You can also use parameters to set whether a stage runs. In this example, there's a pipeline with four stages and different jobs for each stage. The Performance Test stage runs if the parameter `runPerfTests` is true. The default value of `runPerfTests` is false so without any updates, only three of the four stages run.

YAML

```
parameters:
- name: runPerfTests
  type: boolean
  default: false

trigger: none

stages:
- stage: Build
  displayName: Build
  jobs:
  - job: Build
    steps:
    - script: echo running Build


- stage: UnitTest
  displayName: Unit Test
  dependsOn: Build
  jobs:
  - job: UnitTest
    steps:
    - script: echo running UnitTest


- ${{ if eq(parameters.runPerfTests, true) }}:
  - stage: PerfTest
    displayName: Performance Test
    dependsOn: Build
    jobs:
    - job: PerfTest
      steps:
```

```yaml
      - script: echo running PerfTest


  - stage: Deploy
    displayName: Deploy
    dependsOn: UnitTest
    jobs:
    - job: Deploy
      steps:
      - script: echo running UnitTest
```

Loop through parameters

You can also loop through your string, number, and boolean parameters.

Script

In this example, you loop through parameters and print the name and value of each parameter. There are four different parameters and each represents a different type. `myStringName` is a single-line string. `myMultiString` is a multi-line string. `myNumber` is a number. `myBoolean` is a boolean value. In the steps section, the script tasks output the key and value of each parameter.

YAML

```yaml
# start.yaml
parameters:
- name: myStringName
  type: string
  default: a string value
- name: myMultiString
  type: string
  default: default
  values:
  - default
  - ubuntu
- name: myNumber
  type: number
  default: 2
  values:
  - 1
  - 2
  - 4
  - 8
  - 16
- name: myBoolean
  type: boolean
  default: true
```

```
steps:
- ${{ each parameter in parameters }}:
  - script: echo ${{ parameter.Key }}
  - script: echo ${{ parameter.Value }}
```

YAML

```yaml
# azure-pipeline.yaml
trigger: none

extends:
  template: start.yaml
```

Check for an empty parameter object

You can use the `length()` expression to check whether an object parameter has no value.

YAML

```yaml
parameters:
- name: foo
  type: object
  default: []

steps:
- checkout: none
- ${{ if eq(length(parameters.foo), 0) }}:
  - script: echo Foo is empty
    displayName: Foo is empty
```

Parameter data types

Data type	Notes
`string`	string
`number`	may be restricted to `values:`, otherwise any number-like string is accepted
`boolean`	`true` or `false`
`object`	any YAML structure
`step`	a single step
`stepList`	sequence of steps

Data type	Notes
`job`	a single job
`jobList`	sequence of jobs
`deployment`	a single deployment job
`deploymentList`	sequence of deployment jobs
`stage`	a single stage
`stageList`	sequence of stages

The step, stepList, job, jobList, deployment, deploymentList, stage, and stageList data types all use standard YAML schema format. This example includes string, number, boolean, object, step, and stepList.

YAML

```yaml
parameters:
- name: myString
  type: string
  default: a string
- name: myMultiString
  type: string
  default: default
  values:
  - default
  - ubuntu
- name: myNumber
  type: number
  default: 2
  values:
  - 1
  - 2
  - 4
  - 8
  - 16
- name: myBoolean
  type: boolean
  default: true
- name: myObject
  type: object
  default:
    foo: FOO
    bar: BAR
    things:
    - one
    - two
    - three
    nested:
      one: apple
      two: pear
```

```yaml
      count: 3
- name: myStep
  type: step
  default:
    script: echo my step
- name: mySteplist
  type: stepList
  default:
    - script: echo step one
    - script: echo step two

trigger: none

jobs:
- job: stepList
  steps: ${{ parameters.mySteplist }}
- job: myStep
  steps:
    - ${{ parameters.myStep }}
```

Process parameters

Article • 02/06/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> ⓘ **Note**
>
> This guidances does not apply to YAML pipelines. For parameters in YAML pipelines, see **runtime parameters**.

Process parameters are used in classic pipelines and differ from variables in the kind of input supported by them. Variables only take in string inputs while process parameters in addition to string inputs support more data types like check boxes and drop-down list boxes.

You can link all important arguments for tasks used across the build definition as process parameters, which are then shown at one place - the Pipeline view. This means you can quickly edit these arguments without needing to click through all the tasks.

Templates come with a set of predefined process parameters.

> ⓘ **Note**
>
> The **Link** and **Unlink** functionality applies to build pipelines only. It does not apply to release pipelines.

To link more arguments across all tasks to new or existing process parameters, select **Link** from the task argument.

To set a process parameter, edit your pipeline and go to **Tasks** > **Pipeline**.

Select **Unlink** if you need to disconnect an argument from a process parameter.

Classic release and artifacts variables

Article • 02/28/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Classic release and artifacts variables are a convenient way to exchange and transport data throughout your pipeline. Each variable is stored as a string and its value can change between runs of your pipeline.

Variables are different from Runtime parameters which are only available at template parsing time.

As you compose the tasks for deploying your application into each stage in your DevOps CI/CD processes, variables will help you to:

- Define a more generic deployment pipeline once, and then customize it easily for each stage. For example, a variable can be used to represent the connection string for web deployment, and the value of this variable can be changed from one stage to another. These are **custom variables**.
- Use information about the context of the particular release, stage, artifacts, or agent in which the deployment pipeline is being run. For example, your script may need access to the location of the build to download it, or to the working directory on the agent to create temporary files. These are **default variables**.

> **Note**
>
> For YAML pipelines, see **user-defined variables** and **predefined variables** for more details.

Default variables

Information about the execution context is made available to running tasks through default variables. Your tasks and scripts can use these variables to find information about the system, release, stage, or agent they are running in. With the exception of **System.Debug**, these variables are read-only and their values are automatically set by the system. Some of the most significant variables are described in the following tables. To view the full list, see View the current values of all variables.

> **Tip**
>
> You can view the **current values of all variables** for a release, and use a default variable to **run a release in debug mode**.

System

Variable name	Description
System.TeamFoundationServerUri	The URL of the service connection in TFS or Azure Pipelines. Use this from your scripts or tasks to call Azure Pipelines REST APIs. Example: `https://fabrikam.vsrm.visualstudio.com/`
System.TeamFoundationCollectionUri	The URL of the Team Foundation collection or Azure Pipelines. Use this from your scripts or tasks to call REST APIs on other services such as Build and Version control. Example: `https://dev.azure.com/fabrikam/`
System.CollectionId	The ID of the collection to which this build or release belongs. Not available in TFS 2015. Example: `6c6f3423-1c84-4625-995a-f7f143a1e43d`
System.DefinitionId	The ID of the release pipeline to which the current release belongs. Not available in TFS 2015. Example: `1`
System.TeamProject	The name of the project to which this build or release belongs. Example: `Fabrikam`
System.TeamProjectId	The ID of the project to which this build or release belongs. Not available in TFS 2015. Example: `79f5c12e-3337-4151-be41-a268d2c73344`

Variable name	Description
System.ArtifactsDirectory	The directory to which artifacts are downloaded during deployment of a release. The directory is cleared before every deployment if it requires artifacts to be downloaded to the agent. Same as Agent.ReleaseDirectory and System.DefaultWorkingDirectory. Example: `C:\agent\_work\r1\a`
System.DefaultWorkingDirectory	The directory to which artifacts are downloaded during deployment of a release. The directory is cleared before every deployment if it requires artifacts to be downloaded to the agent. Same as Agent.ReleaseDirectory and System.ArtifactsDirectory. Example: `C:\agent\_work\r1\a`
System.WorkFolder	The working directory for this agent, where subfolders are created for every build or release. Same as Agent.RootDirectory and Agent.WorkFolder. Example: `C:\agent\_work`
System.Debug	This is the only system variable that can be *set* by the users. Set this to true to run the release in debug mode to assist in fault-finding. Example: `true`

Release

Variable name	Description
Release.AttemptNumber	The number of times this release is deployed in this stage. Not available in TFS 2015. Example: `1`
Release.DefinitionEnvironmentId	The ID of the stage in the corresponding release pipeline. Not available in TFS 2015. Example: `1`
Release.DefinitionId	The ID of the release pipeline to which the current release belongs. Not available in TFS 2015. Example: `1`

Variable name	Description
Release.DefinitionName	The name of the release pipeline to which the current release belongs. Example: `fabrikam-cd`
Release.Deployment.RequestedFor	The display name of the identity that triggered (started) the deployment currently in progress. Not available in TFS 2015. Example: `Mateo Escobedo`
Release.Deployment.RequestedForEmail	The email address of the identity that triggered (started) the deployment currently in progress. Not available in TFS 2015. Example: `mateo@fabrikam.com`
Release.Deployment.RequestedForId	The ID of the identity that triggered (started) the deployment currently in progress. Not available in TFS 2015. Example: `2f435d07-769f-4e46-849d-10d1ab9ba6ab`
Release.DeploymentID	The ID of the deployment. Unique per job. Example: `254`
Release.DeployPhaseID	The ID of the phase where deployment is running. Example: `127`
Release.EnvironmentId	The ID of the stage instance in a release to which the deployment is currently in progress. Example: `276`
Release.EnvironmentName	The name of stage to which deployment is currently in progress. Example: `Dev`
Release.EnvironmentUri	The URI of the stage instance in a release to which deployment is currently in progress. Example: `vstfs://ReleaseManagement/Environment/276`

Variable name	Description
Release.Environments.{stage-name}.status	The deployment status of the stage. Example: `InProgress`
Release.PrimaryArtifactSourceAlias	The alias of the primary artifact source Example: `fabrikam\_web`
Release.Reason	The reason for the deployment. Supported values are: `ContinuousIntegration` - the release started in Continuous Deployment after a build completed. `Manual` - the release started manually. `None` - the deployment reason has not been specified. `Schedule` - the release started from a schedule.
Release.ReleaseDescription	The text description provided at the time of the release. Example: `Critical security patch`
Release.ReleaseId	The identifier of the current release record. Example: `118`
Release.ReleaseName	The name of the current release. Example: `Release-47`
Release.ReleaseUri	The URI of current release. Example: `vstfs://ReleaseManagement/Release/118`
Release.ReleaseWebURL	The URL for this release. Example: `https://dev.azure.com/fabrikam/f3325c6c/_release?releaseId=392&_a=release-summary`
Release.RequestedFor	The display name of identity that triggered the release. Example: `Mateo Escobedo`
Release.RequestedForEmail	The email address of identity that triggered the release. Example: `mateo@fabrikam.com`
Release.RequestedForId	The ID of identity that triggered the release. Example: `2f435d07-769f-4e46-849d-10d1ab9ba6ab`

Variable name	Description
Release.SkipArtifactsDownload	Boolean value that specifies whether or not to skip downloading of artifacts to the agent. Example: `FALSE`
Release.TriggeringArtifact.Alias	The alias of the artifact which triggered the release. This is empty when the release was scheduled or triggered manually. Example: `fabrikam\_app`

Release-stage

Variable name	Description
Release.Environments.{stage name}.Status	The status of deployment of this release within a specified stage. Not available in TFS 2015. Example: `NotStarted`

Agent

Variable name	Description
Agent.Name	The name of the agent as registered with the agent pool. This is likely to be different from the computer name. Example: `fabrikam-agent`
Agent.MachineName	The name of the computer on which the agent is configured. Example: `fabrikam-agent`
Agent.Version	The version of the agent software. Example: `2.109.1`
Agent.JobName	The name of the job that is running, such as Release or Build. Example: `Release`

Variable name	Description
Agent.HomeDirectory	The folder where the agent is installed. This folder contains the code and resources for the agent. Example: `C:\agent`
Agent.ReleaseDirectory	The directory to which artifacts are downloaded during deployment of a release. The directory is cleared before every deployment if it requires artifacts to be downloaded to the agent. Same as System.ArtifactsDirectory and System.DefaultWorkingDirectory. Example: `C:\agent\_work\r1\a`
Agent.RootDirectory	The working directory for this agent, where subfolders are created for every build or release. Same as Agent.WorkFolder and System.WorkFolder. Example: `C:\agent\_work`
Agent.WorkFolder	The working directory for this agent, where subfolders are created for every build or release. Same as Agent.RootDirectory and System.WorkFolder. Example: `C:\agent\_work`
Agent.DeploymentGroupId	The ID of the deployment group the agent is registered with. This is available only in deployment group jobs. Not available in TFS 2018 Update 1. Example: `1`

General Artifact

For each artifact that is referenced in a release, you can use the following artifact variables. Not all variables are meaningful for each artifact type. The table below lists the default artifact variables and provides examples of the values that they have depending on the artifact type. If an example is empty, it implies that the variable is not populated for that artifact type.

Replace the `{alias}` placeholder with the value you specified for the artifact alias or with the default value generated for the release pipeline.

Variable name	Description

Variable name	Description
Release.Artifacts.{alias}.DefinitionId	The identifier of the build pipeline or repository. Azure Pipelines example: `1` GitHub example: `fabrikam/asp`
Release.Artifacts.{alias}.DefinitionName	The name of the build pipeline or repository. Azure Pipelines example: `fabrikam-ci` TFVC example: `$/fabrikam` Git example: `fabrikam` GitHub example: `fabrikam/asp (main)`
Release.Artifacts.{alias}.BuildNumber	The build number or the commit identifier. Azure Pipelines example: `20170112.1` Jenkins/TeamCity example: `20170112.1` TFVC example: `Changeset 3` Git example: `38629c964` GitHub example: `38629c964`
Release.Artifacts.{alias}.BuildId	The build identifier. Azure Pipelines example: `130` Jenkins/TeamCity example: `130` GitHub example: `38629c964d21fe405ef830b7d0220966b82c9e11`
Release.Artifacts.{alias}.BuildURI	The URL for the build. Azure Pipelines example: `vstfs://build-release/Build/130` GitHub example: `https://github.com/fabrikam/asp`
Release.Artifacts.{alias}.SourceBranch	The full path and name of the branch from which the source was built. Azure Pipelines example: `refs/heads/main`
Release.Artifacts.{alias}.SourceBranchName	The name only of the branch from which the source was built. Azure Pipelines example: `main`
Release.Artifacts.{alias}.SourceVersion	The commit that was built. Azure Pipelines example: `bc0044458ba1d9298cdc649cb5dcf013180706f7`

Variable name	Description
Release.Artifacts.{alias}.Repository.Provider	The type of repository from which the source was built. Azure Pipelines example: `Git`
Release.Artifacts.{alias}.RequestedForID	The identifier of the account that triggered the build. Azure Pipelines example: `2f435d07-769f-4e46-849d-10d1ab9ba6ab`
Release.Artifacts.{alias}.RequestedFor	The name of the account that requested the build. Azure Pipelines example: `Mateo Escobedo`
Release.Artifacts.{alias}.Type	The type of artifact source, such as Build. Azure Pipelines example: `Build` Jenkins example: `Jenkins` TeamCity example: `TeamCity` TFVC example: `TFVC` Git example: `Git` GitHub example: `GitHub`
Release.Artifacts.{alias}.PullRequest.TargetBranch	The full path and name of the branch that is the target of a pull request. This variable is initialized only if the release is triggered by a pull request flow. Azure Pipelines example: `refs/heads/main`
Release.Artifacts.{alias}.PullRequest.TargetBranchName	The name only of the branch that is the target of a pull request. This variable is initialized only if the release is triggered by a pull request flow. Azure Pipelines example: `main`

See also Artifact source alias

Primary Artifact

You designate one of the artifacts as a primary artifact in a release pipeline. For the designated primary artifact, Azure Pipelines populates the following variables.

Variable name	Same as
Build.DefinitionId	Release.Artifacts.{Primary artifact alias}.DefinitionId
Build.DefinitionName	Release.Artifacts.{Primary artifact alias}.DefinitionName

Variable name	Same as
Build.BuildNumber	Release.Artifacts.{Primary artifact alias}.BuildNumber
Build.BuildId	Release.Artifacts.{Primary artifact alias}.BuildId
Build.BuildURI	Release.Artifacts.{Primary artifact alias}.BuildURI
Build.SourceBranch	Release.Artifacts.{Primary artifact alias}.SourceBranch
Build.SourceBranchName	Release.Artifacts.{Primary artifact alias}.SourceBranchName
Build.SourceVersion	Release.Artifacts.{Primary artifact alias}.SourceVersion
Build.Repository.Provider	Release.Artifacts.{Primary artifact alias}.Repository.Provider
Build.RequestedForID	Release.Artifacts.{Primary artifact alias}.RequestedForID
Build.RequestedFor	Release.Artifacts.{Primary artifact alias}.RequestedFor
Build.Type	Release.Artifacts.{Primary artifact alias}.Type
Build.PullRequest.TargetBranch	Release.Artifacts.{Primary artifact alias}.PullRequest.TargetBranch
Build.PullRequest.TargetBranchName	Release.Artifacts.{Primary artifact alias}.PullRequest.TargetBranchName

Use default variables

You can use the default variables in two ways - as parameters to tasks in a release pipeline or in your scripts.

You can directly use a default variable as an input to a task. For example, to pass `Release.Artifacts.{Artifact alias}.DefinitionName` for the artifact source whose alias is **ASPNET4.CI** to a task, you would use `$(Release.Artifacts.ASPNET4.CI.DefinitionName)`.

To use a default variable in your script, you must first replace the `.` in the default variable names with `_`. For example, to print the value of artifact variable `Release.Artifacts.{Artifact alias}.DefinitionName` for the artifact source whose alias is **ASPNET4.CI** in a PowerShell script, you would use `$env:RELEASE_ARTIFACTS_ASPNET4_CI_DEFINITIONNAME`.

Note that the original name of the artifact source alias, `ASPNET4.CI`, is replaced by `ASPNET4_CI`.

View the current values of all variables

1. Open the pipelines view of the summary for the release, and choose the stage you are interested in. In the list of steps, choose **Initialize job**.

2. This opens the log for this step. Scroll down to see the values used by the agent for this job.

Run a release in debug mode

Show additional information as a release executes and in the log files by running the entire release, or just the tasks in an individual release stage, in debug mode. This can help you resolve issues and failures.

- To initiate debug mode for an entire release, add a variable named `System.Debug` with the value `true` to the **Variables** tab of a release pipeline.
- To initiate debug mode for a single stage, open the **Configure stage** dialog from the shortcut menu of the stage and add a variable named `System.Debug` with the value `true` to the **Variables** tab.
- Alternatively, create a variable group containing a variable named `System.Debug` with the value `true` and link this variable group to a release pipeline.

> **Tip**
>
> If you get an error related to an Azure RM service connection, see **How to: Troubleshoot Azure Resource Manager service connections**.

Custom variables

Custom variables can be defined at various scopes.

- Share values across all of the definitions in a project by using variable groups. Choose a variable group when you need to use the same values across all the

definitions, stages, and tasks in a project, and you want to be able to change the values in a single place. You define and manage variable groups in the **Library** tab.

- Share values across all of the stages by using **release pipeline variables**. Choose a release pipeline variable when you need to use the same value across all the stages and tasks in the release pipeline, and you want to be able to change the value in a single place. You define and manage these variables in the **Variables** tab in a release pipeline. In the Pipeline Variables page, open the Scope drop-down list and select "Release". By default, when you add a variable, it is set to Release scope.

- Share values across all of the tasks within one specific stage by using **stage variables**. Use a stage-level variable for values that vary from stage to stage (and are the same for all the tasks in an stage). You define and manage these variables in the **Variables** tab of a release pipeline. In the Pipeline Variables page, open the Scope drop-down list and select the required stage. When you add a variable, set the Scope to the appropriate environment.

Using custom variables at project, release pipeline, and stage scope helps you to:

- Avoid duplication of values, making it easier to update all occurrences as one operation.

- 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.

> **ⓘ Important**
>
> The values of the hidden (secret) variables are securely stored on the server and cannot be viewed by users after they are saved. During a deployment, the Azure Pipelines release service decrypts these values when referenced by the tasks and passes them to the agent over a secure HTTPS channel.

> **ⓘ Note**
>
> Creating custom variables can overwrite standard variables. For example, the PowerShell **Path** environment variable. If you create a custom `Path` variable on a Windows agent, it will overwrite the `$env:Path` variable and PowerShell won't be able to run.

Use custom variables

To use custom variables in your build and release tasks, simply enclose the variable name in parentheses and precede it with a $ character. For example, if you have a variable named **adminUserName**, you can insert the current value of that variable into a parameter of a task as `$(adminUserName)`.

> **Note**
>
> Variables in different groups that are linked to a pipeline in the same scope (for example, job or stage) will collide and the result may be unpredictable. Ensure that you use different names for variables across all your variable groups.

Define and modify your variables in a script

To define or modify a variable from a script, use the `task.setvariable` logging command. Note that the updated variable value is scoped to the job being executed, and does not flow across jobs or stages. Variable names are transformed to uppercase, and the characters "." and " " are replaced by "_".

For example, `Agent.WorkFolder` becomes `AGENT_WORKFOLDER`. On Windows, you access this as `%AGENT_WORKFOLDER%` or `$env:AGENT_WORKFOLDER`. On Linux and macOS, you use `$AGENT_WORKFOLDER`.

> **Tip**
>
> You can run a script on a:
>
> - **Windows agent** using either a **Batch script task** or **PowerShell script task**.
> - **macOS** or **Linux** agent using a **Shell script task**.

Batch

Batch script

Set the `sauce` and `secret.Sauce` variables

```
bat
```

```
@echo ##vso[task.setvariable variable=sauce]crushed tomatoes
@echo ##vso[task.setvariable variable=secret.Sauce;issecret=true]crushed
tomatoes with garlic
```

Read the variables

Arguments

arguments

```
"$(sauce)" "$(secret.Sauce)"
```

Script

bat

```
@echo off
set sauceArgument=%~1
set secretSauceArgument=%~2
@echo No problem reading %sauceArgument% or %SAUCE%
@echo But I cannot read %SECRET_SAUCE%
@echo But I can read %secretSauceArgument% (but the log is redacted so I
do not spoil
     the secret)
```

Console output from reading the variables:

Output

```
No problem reading crushed tomatoes or crushed tomatoes
But I cannot read
But I can read ******** (but the log is redacted so I do not spoil the
secret)
```

Use Azure Key Vault secrets in Azure Pipelines

Article • 01/20/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Azure Key Vault enables developers to securely store and manage secrets such as API keys, credentials or certificates. Azure Key Vault service supports two types of containers: vaults and managed HSM (hardware security module) pools. Vaults support storing software and HSM-backed keys, secrets, and certificates, while managed HSM pools only support HSM-backed keys.

In this tutorial, you will learn how to:

- ✓ Create an Azure Key Vault using Azure CLI
- ✓ Add a secret and configure access to Azure key vault
- ✓ Use secrets in your pipeline

Prerequisites

- An Azure DevOps organization. If you don't have one, you can create one for free.
- An Azure subscription. Create an Azure account for free if you don't have one already.

Create an Azure Key Vault

Sign in to the Azure Portal, and then select the Cloud Shell button in the upper-right corner.

1. If you have more than one Azure subscription associated with your account, use the command below to specify a default subscription. You can use `az account list` to generate a list of your subscriptions.

   Azure CLI

   ```
   az account set --subscription <your_subscription_name_or_ID>
   ```

2. Set your default Azure region. You can use `az account list-locations` to generate a list of available regions.

Azure CLI

```
az config set defaults.location=<your_region>
```

For example, this command will select the westus2 region:

Azure CLI

```
az config set defaults.location=westus2
```

3. Create a new resource group. A resource group is a container that holds related resources for an Azure solution.

Azure CLI

```
az group create --name <your-resource-group>
```

4. Create a new key vault.

Azure CLI

```
az keyvault create \
  --name <your-key-vault> \
  --resource-group <your-resource-group>
```

5. Create a new secret in your Azure key vault.

Azure CLI

```
az keyvault secret set \
  --name "Password" \
  --value "mysecretpassword" \
  --vault-name <your-key-vault-name>
```

Create a project

1. Sign in to your Azure DevOps organization.

2. If you don't have any projects in your organization yet, select **Create a project to get started**. Otherwise, select **New project** in the upper-right corner.

Create a repo

We will use YAML to create our pipeline but first we need to create a new repo.

1. Sign in to your Azure DevOps organization and navigate to your project.
2. Select **Repos**, and then select **Initialize** to initialize a new repo with a README.

Create a new pipeline

YAML

1. Select **Pipelines**, and then select **New Pipeline**.
2. Select **Azure Repos Git** (YAML).

3. Select the repository you created in the previous step.
4. Select the **Starter pipeline** template.
5. The default pipeline will include a few scripts that run echo commands. Those are not needed so we can delete them. Your new YAML file should look like this:

YAML

```yaml
trigger:
- main

pool:
```

```
    vmImage: 'ubuntu-latest'

steps:
```

6. Select **Show assistant** to expand the assistant panel. This panel provides convenient and searchable list of pipeline tasks.

7. Search for **vault** and select the **Azure Key Vault** task.

8. Select your **Azure subscription** and then select **Authorize**. Select your **Key vault** from the dropdown menu, and then select **Add** to add the task to your YAML pipeline.

> **Note**
>
> The **Make secrets available to whole job** feature is not supported in Azure DevOps Server 2019 and 2020.

9. Your YAML file should look like the following:

YAML

```yaml
trigger:
- main

pool:
  vmImage: ubuntu-latest

steps:
- task: AzureKeyVault@2
  inputs:
    azureSubscription: 'Your-Azure-Subscription'
    KeyVaultName: 'Your-Key-Vault-Name'
    SecretsFilter: '*'
    RunAsPreJob: false

- task: CmdLine@2
  inputs:
    script: 'echo $(Your-Secret-Name) > secret.txt'

- task: CopyFiles@2
```

```
    inputs:
      Contents: secret.txt
      targetFolder: '$(Build.ArtifactStagingDirectory)'

  - task: PublishBuildArtifacts@1
    inputs:
      PathtoPublish: '$(Build.ArtifactStagingDirectory)'
      ArtifactName: 'drop'
      publishLocation: 'Container'
```

Don't save or queue your pipeline just yet. We must first give our pipeline the right permissions to access Azure Key Vault. Keep your browser tab open, we will resume the remaining steps once we set up the key vault permissions.

Set up Azure Key Vault access policies

In order to access our Azure Key Vault, we must first set up a service principal to give access to Azure Pipelines. Follow this guide to create your service principal and then proceed with the next steps in this section.

1. Navigate to Azure portal.

2. Use the search bar to search for the key vault you created earlier.

3. Under **Settings** Select **Access policies**.

4. Select **Add Access Policy** to add a new policy.

5. For **Secret permissions**, select **Get** and **List**.

6. Select the option to select a service principal and search for the one you created in the beginning of this section. A security principal is an object that represents a user, group, service, or application that's requesting access to Azure resources.

7. Select **Add** to create the access policy, then select **Save** when you are done.

Run and review the pipeline

1. Return to the previous tab where we left off.

2. Select **Save**, and then select **Save** again to commit your changes and trigger the pipeline. You may be asked to allow the pipeline access to Azure resources, if prompted select **Allow**. You will only have to approve your pipeline once.

3. Select the **CmdLine** task to view the logs.

4. Return to pipeline summary and select the published artifact.

5. Select the **secret.txt** artifact to open it.

6. The text file should contain our secret: *mysecretpassword*.

> ⚠ **Warning**

This tutorial is for educational purposes only. For security best practices and how to safely work with secrets, see **Manage secrets in your server apps with Azure Key Vault**.

Clean up resources

Follow the steps below to delete the resources you created:

1. If you created a new organization to host your project, see how to delete your organization, otherwise delete your project.

2. All Azure resources created during this tutorial are hosted under a single resource group *PipelinesKeyVaultResourceGroup*. Run the following command to delete the resource group and all of its resources.

   Azure CLI

   ```
   az group delete --name PipelinesKeyVaultResourceGroup
   ```

FAQ

Q: I'm getting the following error: "the user or group does not have secrets list permission" what should I do?

A: If you encounter an error indicating that the user or group does not have secrets list permission on key vault, run the following commands to authorize your application to access the key or secret in the Azure Key Vault:

PowerShell

```
$ErrorActionPreference="Stop";
$Credential = Get-Credential;
Connect-AzAccount -SubscriptionId <YOUR_SUBSCRIPTION_ID> -Credential
$Credential;
$spn=(Get-AzureRmADServicePrincipal -SPN <YOUR_SERVICE_PRINCIPAL_ID>);
$spnObjectId=$spn.Id;
Set-AzureRmKeyVaultAccessPolicy -VaultName key-vault-tutorial -ObjectId
$spnObjectId -PermissionsToSecrets get,list;
```

Next steps

Artifacts in Azure Pipelines

Publish and download artifacts in Azure Pipelines

Release artifacts and artifact sources

Use Azure Key Vault secrets in your Pipeline

Article • 10/31/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

With Azure Key Vault, you can securely store and manage your sensitive information such as passwords, API keys, certificates, etc. using Azure Key Vault, you can easily create and manage encryption keys to encrypt your data. Azure Key Vault can also be used to manage certificates for all your resources. In this article, you'll learn how to:

- ✓ Create an Azure Key Vault.
- ✓ Configure your Key Vault permissions.
- ✓ Create a new service connection.
- ✓ Query for secrets from your Azure Pipeline.

Prerequisites

- An Azure DevOps organization. Create one for free if you don't already have one.
- Your own project. Create a project if you don't already have one.
- Your own repository. Create a new Git repo if you don't already have one.
- An Azure subscription. Create a free Azure account if you don't already have one.

Create an Azure Key Vault

Azure portal

1. Navigate to Azure portal.

2. Select **Create a resource** in the left navigation pane.

3. Search for **Key Vault** and then press Enter.

4. Select **Create** to create a new Azure Key Vault.

5. Select your **Subscription** and then add a new **Resource group**. Enter a **Key vault name** and select a **Region** and a **Pricing tier**. Select **Review + create** when you're done.

6. Select **Go to resource** when the deployment of your new resource is completed.

Create a service principal

In this step, we will create a new service principal in Azure, enabling us to query our Azure Key Vault from Azure Pipelines.

1. Navigate to Azure portal.

2. From the menu bar, select the >_ icon to open the **Cloud Shell**.

3. Select **PowerShell** or leave it as **Bash** based on your preference.

4. Run the following command to create a new service principal:

Azure

```
az ad sp create-for-rbac -name YOUR_SERVICE_PRINCIPAL_NAME
```

5. Your output should match the example below. Be sure to copy the output of your command, as you will need it to create the service connection in the upcoming step.

JSON

```
{
  "appId": "p951q3e2-8e5r-z697-e9q52aviu8a2",
  "displayName": "MyServicePrincipal",
  "password": "**********************************",
```

```
  "tenant": "85wes2u6-63sh-95zx-2as3-qw58wex269df"
}
```

Configure Key Vault access permissions

1. Navigate to Azure portal.

2. Select the key vault you created in the previous step.

3. Select **Access policies**.

4. Select **Add Access Policy** to add a new policy.

5. Add a **Get** and **List** to **Secret permissions**.

6. Under **Select principal**, select to add a service principal and choose the one you created earlier.

7. Select **Save** when you're done.

> ⓘ **Note**
>
> Azure Key Vaults that use Azure role-based access control (Azure RBAC) are not supported.

Create a new service connection

1. Sign in to your Azure DevOps organization, and then navigate to your project.

2. Select ⚙ **Project settings**, and then select **Service connections**.

3. If you're setting up a service connection for the first time in your project, select **Create service connection**. If you've made service connections before, select **New service connection**.

4. Select **Azure Resource Manager**, and then select **Next**.

5. Select **Service principal (manual)**, and then select **Next**.

6. Select **Subscription** for the **Scope Level**, and fill in the required fields with information from the previously created service principal. Select **Verify** when you're done:

    - **Service Principal Id**: Your service principal **appId**.
    - **Service Principal key**: Your service principal **password**.
    - **Tenant ID**: Your service principal **tenant**.

7. Provide a name for your service connection, and make sure you check the **Grant access permission to all pipelines** checkbox.

8. Select **Verify and save** when you're done.

Query and use secrets in your pipeline

Using the Azure Key Vault task we can fetch the value of our secret and use it in subsequent tasks in our pipeline. One thing to keep in mind is that secrets must be

explicitly mapped to env variable as shown in the example below.

YAML

```yaml
pool:
  vmImage: 'ubuntu-latest'

steps:
- task: AzureKeyVault@1
  inputs:
    azureSubscription: 'repo-kv-demo'                    ##
YOUR_SERVICE_CONNECTION_NAME
    KeyVaultName: 'kv-demo-repo'                         ##
YOUR_KEY_VAULT_NAME
    SecretsFilter: 'secretDemo'                          ##
YOUR_SECRET_NAME. Default value: *
    RunAsPreJob: false                                   ## Make the
secret(s) available to the whole job

- task: DotNetCoreCLI@2
  inputs:
    command: 'build'
    projects: '**/*.csproj'

- task: DotNetCoreCLI@2
  inputs:
    command: 'run'
    projects: '**/*.csproj'
  env:
    mySecret: $(secretDemo)

- bash: |
    echo "Secret Found! $MY_MAPPED_ENV_VAR"
  env:
    MY_MAPPED_ENV_VAR: $(mySecret)
```

The output from the last bash command should look like this:

```
Secret Found! ***
```

> **Note**
>
> If you want to query for multiple secrets from your Azure Key Vault, use the `SecretsFilter` argument to pass a comma-separated list of secret names: '*secret1, secret2*'.

Related articles

- Manage service connections
- Define variables
- Publish Pipeline Artifacts

Release gates and approvals overview

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Release pipelines enable teams to continuously deploy their application across different stages with lower risk and with faster pace. Deployments to each stage can be fully automated by using jobs and tasks.

Teams can also take advantage of the Approvals and Gates feature to control the workflow of the deployment pipeline. Each stage in a release pipeline can be configured with pre-deployment and post-deployment conditions that can include waiting for users to manually approve or reject deployments, and checking with other automated systems that specific conditions are met. In addition, teams can configure manual validations to pause the deployment pipeline and prompt users to carry out manual tasks then resume or reject the deployment.

The following diagram illustrates the release pipeline workflow.

By using gates, approvals, and manual intervention you can take full control of your releases to meet a wide range of deployment requirements. Typical scenarios where approvals, gates, and manual intervention are useful include the following.

Scenario	Feature(s) to use
A user must manually validate the change request and approve the deployment to a certain stage.	Pre-deployment approvals
A user must manually sign out after deployment before the release is triggered to other stages.	Post-deployment approvals
A team wants to ensure there are no active issues in the work item or problem management system before deploying a build to a stage.	Pre-deployment gates

Scenario	Feature(s) to use
A team wants to ensure there are no reported incidents after deployment, before triggering a release.	Post-deployment gates
After deployment, a team wants to wait for a specified time before prompting users to sign out.	Post-deployment gates and post-deployment approvals
During deployment, a user must manually follow specific instructions and then resume the deployment.	Manual Intervention or Manual Validation
During deployment, a team wants to prompt users to enter a value for a parameter used by the deployment tasks, or allow users to edit the release.	Manual Intervention or Manual Validation
During deployment, a team wants to wait for monitoring or information portals to detect any active incidents, before continuing with other deployment jobs.	Planned

You can combine all three techniques within a release pipeline to fully achieve your own deployment requirements.

In addition, you can install an extension that integrates with *ServiceNow* to help you control and manage your deployments through Service Management methodologies such as ITIL. For more information, see Integrate with ServiceNow change management.

> **Note**
>
> The time delay before pre-deployment gates are executed is capped at 48 hours. If you need to delay the overall launch of your gates instead, it is recommended to use a **delay task** in your release pipeline.

YAML

```yaml
# Delay further execution of a workflow by a fixed time
pool: server
steps:
- task: Delay@1
  displayName: 'Delay by 5 minutes'
  inputs:
    delayForMinutes: 5
```

> **Note**
>
> The **delay task** can only be used in an **agentless job**.

Related articles

- Release deployment control using approvals
- Release deployment control using gates
- Configure a manual intervention
- Add stages, dependencies, & conditions
- Release triggers
- Releases in Azure Pipelines

Next steps

Define approvals and checks

Use approvals and gates to control your deployment

Define approvals and checks

Article • 10/30/2023

Azure DevOps Services

A pipeline is made up of stages. A pipeline author can control whether a stage should run by defining conditions on the stage. Another way to control if and when a stage should run is through **approvals and checks**.

Pipelines rely on resources such as environments, service connections, agent pools, variable groups, and secure files. Checks enable the *resource owner* to control if and when a stage in any pipeline can consume a resource. As an owner of a resource, you can define checks that must be satisfied before a stage consuming that resource can start. For example, a *manual approval check* on an environment would ensure that deployment to that environment only happens after the designated user(s) has reviewed the changes being deployed.

A stage can consist of many jobs, and each job can consume several resources. Before the execution of a stage can begin, all checks on all the resources used in that stage must be satisfied. Azure Pipelines pauses the execution of a pipeline prior to each stage, and waits for all pending checks to be completed. Checks are reevaluated based on the retry interval specified in each check. If all checks aren't successful until the **timeout** specified, then that stage isn't executed. If any of the checks terminally fails (for example, if you reject an approval on one of the resources), then that stage isn't executed. However, you can retry a stage when approvals and checks time out.

Approvals and other checks aren't defined in the yaml file. Users modifying the pipeline yaml file can't modify the checks performed before start of a stage. Administrators of resources manage checks using the web interface of Azure Pipelines.

> **Important**
>
> Checks can be configured on environments, service connections, repositories, variable groups, secure files, and agent pools.
>
> Service connections cannot be specified by variable.

Approvals

You can manually control when a stage should run using approval and checks. This check is commonly used to control deployments to production environments.

1. Sign in to your Azure DevOps organization, and then navigate to your project.

2. Select **Pipelines** > **Environments**, and then select your environment.

3. Select the **Approvals and checks** tab, and then select the + sign to add a new check.

4. Select **Approvals**, and then select **Next**.

5. Add users or groups as your designated **Approvers**, and, if desired, provide **instructions for the approvers**. Specify if you want to permit or restrict approvers from approving their own runs, and specify your desired **Timeout**. If approvals aren't completed within the specified Timeout, the stage is marked as skipped.

6. Select **Create** when you're done.

7. Once the approval check is triggered, a prompt window, as shown in the example below, is presented in the user interface. This window provides the option for approvers to either reject or approve the run, along with any accompanying instructions.

> ⓘ **Note**
>
> If a group is designated as an approver, only one user within the group needs to approve for the run to proceed.

Branch control

Using the branch control check, you can ensure all the resources linked with the pipeline are built from the **allowed** branches and that the branches have protection enabled. This check helps in controlling the release readiness and quality of deployments. In case multiple resources are linked with the pipeline, source for all the resources is verified. If you've linked another pipeline, then the branch of the specific run being deployed is verified for protection.

To define the branch control check:

1. In your Azure DevOps project, go to the resource (for example, environment) that needs to be protected.

2. Navigate to **Approvals and Checks** for the resource.

3. Choose the **Branch control** check and provide a comma-separated list of allowed branches. You can mandate that the branch should have protection enabled. You can also define the behavior of the check in case protection status for one of the branches isn't known.

At run time, the check would validate branches for all linked resources in the run against the allowed list. If any of the branches doesn't match the criteria, the check fails and the stage is marked failed.

> **Note**
>
> The check requires the branch names to be fully qualified. Make sure the format for branch name is `refs/heads/<branch name>`

Business hours

In case you want all deployments to your environment to happen in a specific time window only, then business hours check is the ideal solution. When you run a pipeline, the execution of the stage that uses the resource waits for business hours. If you have multiple runs executing simultaneously, each of them is independently verified. At the start of the business hours, the check is marked successful for all the runs.

If execution of the stage hasn't started at the end of business hours (held up by to some other check), then the business hours approval is automatically withdrawn and a reevaluation is scheduled for the next day. The check fails if execution of the stage doesn't start within the **Timeout** period specified for the check, and the stage is marked as failed.

Invoke Azure function

Azure functions are the serverless computation platform offered by Azure. With Azure functions, you can run small pieces of code (called "functions") without worrying about application infrastructure. Given the high flexibility, Azure functions provide a great way to author your own checks. You include the logic of the check-in Azure function such that each execution is triggered on http request, has a short execution time and returns a response. While defining the check, you can parse the response body to infer if the check is successful. The evaluation can be repeated periodically using the Time between evaluations setting in control options. Learn More

If your check doesn't succeed within the configured **Timeout**, the associated stage is skipped. Stages depending on it are skipped as well. For more information, see the Azure Function App task.

> **Note**
>
> User defined pipeline variables are accessible to the check starting with **Sprint 215**.

Read more about the recommended way to use Invoke Azure Function checks. Checks need to follow specific rules depending on their mode and the number of retries to be compliant.

Invoke REST API

Invoke REST API check enables you to integrate with any of your existing services. Periodically, make a call to a REST API and continue if it returns a successful response. Learn More

The evaluation can be repeated periodically using the **Time between evaluations** setting in control options. If your check doesn't succeed within the configured **Timeout**, the associated stage is skipped. Stages depending on it are skipped as well. For more information, see Invoke REST API task.

> ⓘ **Note**
>
> User defined pipeline variables are accessible to the check starting with **Sprint 215**.

Read more about the recommended way to use Invoke REST API checks.

Query Azure Monitor Alerts

Azure Monitor offers visualization, query, routing, alerting, autoscale, and automation on data from the Azure infrastructure and each individual Azure resource. Alerts are a standard means to detect issues with the health of infrastructure or application, and take corrective actions. Canary deployments and staged rollouts are common deployment strategies used to lower risk of regressions to critical applications. After deploying to a stage (set of customers), the application is observed for a period of time. Health of the application after deployment is used to decide whether the update should be made to the next stage or not.

Query Azure Monitor Alerts helps you observe Azure Monitor and ensure no alerts are raised for the application after a deployment. The check succeeds if no alert rules are activated at the time of evaluation. Learn More

The evaluation is repeated after **Time between evaluations** setting in control options. The checks fail if the stage hasn't started execution within the specified **Timeout** period.

Required template

With the required template check, you can enforce pipelines to use a specific YAML template. When this check is in place, a pipeline fails if it doesn't extend from the referenced template.

To define a required template approval:

1. In your Azure DevOps project, go to the service connection that you want to restrict.
2. Open **Approvals and Checks** in the menu next to **Edit**.
3. In the **Add your first check** menu, select **Required template**.
4. Enter details on how to get to your required template file.
    - **Repository type**: The location of your repository (GitHub, Azure, or Bitbucket).
    - **Repository**: The name of your repository that contains your template.
    - **Ref**: The branch or tag of the required template.
    - **Path to required template**: The name of your template.

You can have multiple required templates for the same service connection. In this example, the required template is `required.yml`.

Disable a check

When debugging a check, you might want to temporarily disable and then enable it again. To disable or enable a check:

1. In your Azure DevOps project, go to the resource with a check.
2. Open the **Approvals and Checks** tab.
3. In the contextual menu, select **Disable** or **Enable**.

Evaluate artifact

You can evaluate artifact(s) to be deployed to an environment against custom policies.

> ⓘ **Note**
>
> Currently, this works with container image artifacts only

To define a custom policy evaluation over the artifact(s), follow the below steps.

1. In your Azure DevOps Services project, navigate to the environment that needs to be protected. Learn more about creating an environment.

2. Navigate to **Approvals and checks** for the environment.

3. Select **Evaluate artifact**.

4. Paste the policy definition and click **Save**. See more about writing policy definitions.

When you run a pipeline, the execution of that run pauses before entering a stage that uses the environment. The specified policy is evaluated against the available image metadata. The check passes when the policy is successful and fails otherwise. The stage is marked failed if the check fails.

Passed

You can also see the complete logs of the policy checks from the pipeline view.

Exclusive lock

The **exclusive lock** check allows only a single run from the pipeline to proceed. All stages in all runs of that pipeline that use the resource are paused. When the stage using the lock completes, then another stage can proceed to use the resource. Also, only one stage will be allowed to continue.

The behavior of any other stages that attempt to take a lock is configured by the `lockBehavior` value that is configured in the YAML file for the pipeline.

- `runLatest` - Only the latest run acquires the lock to the resource. `runLatest` is the default if no `lockBehavior` is specified.
- `sequential` - All runs acquire the lock sequentially to the protected resource.

To use exclusive lock check with `sequential` deployments or `runLatest`, follow these steps:

1. Enable the exclusive lock check on the environment (or another protected resource).
2. In the YAML file for the pipeline, specify a property called `lockBehavior`. This can be specified for the whole pipeline or for a given stage:

Set on a stage:

YAML

```yaml
stages:
- stage: A
  lockBehavior: sequential
  jobs:
```

```
  - job: Job
    steps:
    - script: Hey!
```

Set on the pipeline:

YAML

```yaml
lockBehavior: runLatest
stages:
- stage: A
  jobs:
  - job: Job
    steps:
    - script: Hey!
```

If you don't specify a `lockBehavior`, the default value of `runLatest` is used.

ServiceNow Change Management

This checks needs the ServiceNow Change Management extension to be installed from the marketplace

The **servicenow change management** check allows for an integration of ServiceNow change management process in the pipelines. By adding the check, a new change request in ServiceNow can be automatically created at the start of the stage. The pipeline waits for the change process to complete before starting the stage. More details are available here.

Multiple Approvals and Checks

A stage can consist of many jobs, and each job can consume several resources. Before the execution of a stage can begin, all checks on all the resources used in that stage must be satisfied. Azure Pipelines pauses the execution of a pipeline prior to each stage, and waits for all pending checks to be completed.

A single final negative decision causes the pipeline to be denied access and the stage to fail. The decisions of all Approvals and Checks except for Invoke Azure Function / REST API and Exclusive lock are final.

When using Invoke Azure Function / REST API checks in the recommended way, their access decisions are also final.

When you specify *Time between evaluations* for an Invoke Azure Function / REST API check to be non-zero, the check's decision is non-final. This scenario is worth exploring.

Let us look at an example. Imagine your YAML pipeline has a stage that uses a Service Connection. This Service Connection has two checks configured for it:

1. An asynchronous check, named *External Approval Granted*, that verifies that an external approval is given and is configured in the recommended way.
2. A synchronous check, named *Deployment Reason Valid*, that verifies that the deployment reason is valid and for which you set the *Time between evaluations* to 7 minutes.

A possible checks execution is shown in the following diagram.

In this execution:

- Both checks, *External Approval Granted* and *Deployment Reason Valid*, are invoked at the same time. *Deployment Reason Valid* fails immediately, but because *External Approval Granted* is pending, it will be retried.
- At minute 7, *Deployment Reason Valid* is retried and this time it passes.
- At minute 15, *External Approval Granted* calls back into Azure Pipelines with a successful decision. Now, both checks pass, so the pipeline is allowed to continue to deploy the stage.

Let us look at another example, involving two synchronous checks. Assume your YAML pipeline has a stage that uses a Service Connection. This Service Connection has two checks configured for it:

1. A synchronous check, named *Sync Check 1*, for which you set the *Time between evaluations* to 5 minutes.

2. A synchronous check, named *Sync Check 2*, for which you set the *Time between evaluations* to 7 minutes.

A possible checks execution is shown in the following diagram.

In this execution:

- Both checks, *Sync Check 1* and *Sync Check 2*, are invoked at the same time. *Sync Check 1* passes, but it will be retried, because *Sync Check 2* fails.
- At minute 5, *Sync Check 1* is retried but fails, so it will be retried.
- At minute 7, *Sync Check 2* is retried and succeeds. The pass decision is valid for 7 minutes. If *Sync Check 1* doesn't pass in this time interval, *Sync Check 2* will be retried.
- At minute 10, *Sync Check 1* is retried but fails, so it will be retried.
- At minute 14, *Sync Check 2* is retried and succeeds. The pass decision is valid for 7 minutes. If *Sync Check 1* doesn't pass in this time interval, *Sync Check 2* will be retried.
- At minute 15, *Sync Check 1* is retried and succeeds. Now, both checks pass, so the pipeline is allowed to continue to deploy the stage.

Let us look at an example that involves an Approval and a synchronous check. Imagine you configured a synchronous check and an Approval for a Service Connection with a *Time between evaluations* of 5 minutes. Until the approval is given, your check will run every 5 minutes, regardless of decision.

FAQ

The checks defined did not start. What happened?

The evaluation of checks starts once the stage conditions are satisfied. You should confirm run of the stage started after the checks were added on the resource and that the resource is consumed in the stage.

How can I use checks for scheduling a stage?

Using the business hours check, you can control the time for start of stage execution. You can achieve the same behavior as predefined schedule on a stage in designer releases.

How can I take advance approvals for a stage scheduled to run in future?

This scenario can be enabled

1. The business hours check enables all stages deploying to a resource to be scheduled for execution between the time window
2. When approvals are configured on the same resource, the stage would wait for approvals before starting.
3. You can configure both the checks on a resource. The stage would wait on approvals and business hours. It would start in the next scheduled window after approvals are complete.

Can I wait for completion of security scanning on the artifact being deployed?

In order to wait for completion of security scanning on the artifact being deployed, you would need to use an external scanning service like AquaScan. The artifact being deployed would need to be uploaded at a location accessible to the scanning service before the start of checks, and can be identified using predefined variables. Using the Invoke REST API check, you can add a check to wait on the API in the security service and pass the artifact identifier as an input.

How can I use output variables from previous stages in a check?

By default, only predefined variables are available to checks. You can use a linked variable group to access other variables. The output variable from the previous stage can be written to the variable group and accessed in the check.

Learn more

- Invoke Azure Function / REST API checks
- Approvals and Checks REST API
- Approvals Query REST API

Invoke Azure Function / REST API checks

Article • 07/24/2023

The Invoke Azure Function / REST API Checks allow you to write code to decide if a specific pipeline stage is allowed to access a protected resource or not. These checks can run in two modes:

- **Asynchronous (Recommended)**: push mode, in which Azure DevOps awaits for the Azure Function / REST API implementation to call back into Azure DevOps with a stage access decision
- **Synchronous**: poll mode, in which Azure DevOps periodically calls the Azure Function / REST API to get a stage access decision

In the rest of this guide, we refer to Azure Function / REST API Checks simply as checks.

The recommended way to use checks is in asynchronous mode. This mode offers you the highest level of control over the check logic, makes it easy to reason about what state the system is in, and decouples Azure Pipelines from your checks implementation, providing the best scalability. All synchronous checks can be implemented using the asynchronous checks mode.

Asynchronous checks

In asynchronous mode, Azure DevOps makes a call to the Azure Function / REST API check and awaits a callback with the resource access decision. There's no open HTTP connection between Azure DevOps and your check implementation during the waiting period.

The rest of this section talks about Azure Function checks, but unless otherwise noted, the guidance applies to Invoke REST API checks as well.

Recommended implementation of asynchronous checks

The recommended asynchronous mode has two communication steps:

1. **Deliver the check payload**. Azure Pipelines makes an HTTP call to your Azure Function / REST API *only* to deliver the check payload, and *not* to receive a decision on the spot. Your function should just acknowledge receipt of the information and

terminate the HTTP connection with Azure DevOps. Your check implementation should process the HTTP request within 3 seconds.

2. **Deliver access decision through a callback**. Your check should run asynchronously, evaluate the condition necessary for the pipeline to access the protected resource (possibly doing multiple evaluations at different points in time). Once it reaches a final decision, your Azure Function makes an HTTP REST call into Azure DevOps to communicate the decision. At any point in time, there should be a single open HTTP connection between Azure DevOps and your check implementation. Doing so saves resources on both your Azure Function side and on Azure Pipelines's side.

If a check passes, then the pipeline is allowed access to a protected resource and stage deployment can proceed. If a check fails, then the stage fails. If there are multiple checks in a single stage, all need to pass before access to protected resources is allowed, but a single failure is enough to fail the stage.

The recommended implementation of the async mode for a single Azure Function check is depicted in the following diagram.

The steps in the diagram are:

1. Check Delivery. Azure Pipelines prepares to deploy a pipeline stage and requires access to a protected resource. It invokes the corresponding Azure Function check and expects receipt confirmation, by the call ending with an HTTP 200 status code. Stage deployment is paused pending a decision.
2. Check Evaluation. This step happens inside your Azure Function implementation, which runs on your own Azure resources and the code of which is completely under your control. We recommend your Azure Function follow these steps:

    - 2.1 Start an *async* task and return HTTP status code `200`
    - 2.2 Enter an inner loop, in which it can do multiple condition evaluations
    - 2.3 Evaluate the access conditions
    - 2.4 If it can't reach a final decision, reschedule a reevaluation of the conditions for a later point, then go to step 2.3

3. Decision Communication. The Azure function calls back into Azure Pipelines with the access decision. Stage deployment can proceed

When you use the recommended implementation, the pipeline run details page shows the latest check log.

Recommended configuration for asynchronous checks

In the Azure Function / REST API check configuration panel, make sure you:

- Selected *Callback* for the *Completion event*
- Set *Time between evaluations (minutes)* to *0*

Setting the *Time between evaluations* to a nonzero value means the check decision (pass / fail) isn't final. The check is reevaluated until all other Approvals & Checks reach a final state.

Pass pipeline run information to checks

When configuring the check, you can specify the pipeline run information you wish to send to your check. At a minimum, you should send:

- `"PlanUrl": "$(system.CollectionUri)"`
- `"ProjectId": "$(system.TeamProjectId)"`
- `"HubName": "$(system.HostType)"`
- `"PlanId": "$(system.PlanId)"`
- `"JobId": "$(system.JobId)"`
- `"TaskInstanceId": "$(system.TaskInstanceId)"`
- `"AuthToken": "$(system.AccessToken)"`

These key-value pairs are set, by default, in the `Headers` of the REST call made by Azure Pipelines.

You should use `AuthToken` to make calls into Azure DevOps, such as when your check calls back with a decision.

Call into Azure DevOps

To reach a decision, your check may need information about the current pipeline run that can't be passed to the check, so the check needs to retrieve it. Imagine your check verifies that a pipeline run executed a particular task, for example a static analysis task. Your check needs to call into Azure DevOps and get the list of already executed tasks.

To call into Azure DevOps, we recommend you use the job access token issued for the check execution, instead of a personal access token (PAT). The token is already provided to your checks by default, in the `AuthToken` header.

Compared to PATs, the job access token is less prone to getting throttled, doesn't need manual refresh, and isn't tied to a personal account. The token is valid for 48 hours.

Using the job access token all but removes Azure DevOps REST API throttling issues. When you use a PAT, you're using the same PAT for all runs of your pipeline. If you run a large number of pipelines, then the PAT gets throttled. This is less of an issue with the job access token since a new token is generated for each check execution.

The token is issued for the build identity used to run a pipeline, for example, *FabrikamFiberChat build service (FabrikamFiber)*. In other words, the token can be used to access the same repositories or pipeline runs that the host pipeline can. If you

enabled *Protect access to repositories in YAML pipelines*, its scope is further restricted to only the repositories referenced in the pipeline run.

If your check needs to access non-Pipelines related resources, for example, Boards user stories, you should grant such permissions to pipelines' build identities. If your check can be triggered from multiple projects, make sure that all pipelines in all projects can access the required resources.

Send a decision back to Azure DevOps

Your check implementation must use the Post Event REST API call to communicate a decision back to Azure Pipelines. Make sure you specify the following properties:

- `Headers`: `Basic: {AuthToken}`
- `Body`:

JSON

```
{
    "name": "TaskCompleted",
    "taskId": "{TaskInstanceId}",
    "jobId": "{JobId}",
    "result": "succeeded|failed"
}
```

Send status updates to Azure DevOps from checks

You can provide status updates to Azure Pipelines users from within your checks using Azure Pipelines REST APIs. This functionality is useful, for example, if you wish to let users know the check is waiting on an external action, such as someone needs to approve a ServiceNow ticket.

The steps to send status updates are:

1. Create a task log
2. Append to the task log
3. Update timeline record

All REST API calls need to be authenticated.

Examples

Basic Azure Function check

In this basic example, the Azure Function checks that the invoking pipeline run executed a `CmdLine` task, prior to granting it access to a protected resource.

The Azure Function goes through the following steps:

1. Confirms the receipt of the check payload
2. Sends a status update to Azure Pipelines that the check started
3. Uses `{AuthToken}` to make a callback into Azure Pipelines to retrieve the pipeline run's Timeline entry
4. Checks if the Timeline contains a task with `"id": "D9BAFED4-0B18-4F58-968D-86655B4D2CE9"` (the ID of the `CmdLine` task)
5. Sends a status update with the result of the search
6. Sends a check decision to Azure Pipelines

You can download this example from GitHub.

To use this Azure Function check, you need to specify the following `Headers` when configuring the check:

JSON

```
{
    "Content-Type":"application/json",
    "PlanUrl": "$(system.CollectionUri)",
    "ProjectId": "$(system.TeamProjectId)",
    "HubName": "$(system.HostType)",
    "PlanId": "$(system.PlanId)",
    "JobId": "$(system.JobId)",
    "TimelineId": "$(system.TimelineId)",
    "TaskInstanceId": "$(system.TaskInstanceId)",
    "AuthToken": "$(system.AccessToken)",
    "BuildId": "$(build.BuildId)"
}
```

Advanced Azure Function check

In this advanced example, the Azure Function checks that the Azure Boards work item referenced in the commit message that triggered the pipeline run is in the correct state.

The Azure Function goes through the following steps:

1. Confirms the receipt of the check payload
2. Sends a status update to Azure Pipelines that the check started
3. Uses `{AuthToken}` to make a callback into Azure Pipelines to retrieve the state of the Azure Boards work item referenced in the commit message that triggered the

pipeline run
4. Checks if the work item is in the `Completed` state
5. Sends a status update with the result of the check
6. If the work item isn't in the `Completed` state, it reschedules another evaluation in 1 minute
7. Once the work item is in the correct state, it sends a positive decision to Azure Pipelines

You can download this example from GitHub.

To use this Azure Function check, you need to specify the following `Headers` when configuring the check:

JSON

```
{
    "Content-Type":"application/json",
    "PlanUrl": "$(system.CollectionUri)",
    "ProjectId": "$(system.TeamProjectId)",
    "HubName": "$(system.HostType)",
    "PlanId": "$(system.PlanId)",
    "JobId": "$(system.JobId)",
    "TimelineId": "$(system.TimelineId)",
    "TaskInstanceId": "$(system.TaskInstanceId)",
    "AuthToken": "$(system.AccessToken)",
    "BuildId": "$(build.BuildId)"
}
```

Error handling

Currently, Azure Pipelines evaluates a single check instance at most 2,000 times.

If your check doesn't call back into Azure Pipelines within the configured timeout, the associated stage is skipped. Stages depending on it are skipped as well.

Synchronous checks

In synchronous mode, Azure DevOps makes a call to the Azure Function / REST API check to get an immediate decision whether access to a protected resource is permitted or not.

The implementation of the sync mode for a single Azure Function check is depicted in the following diagram.

The steps are:

1. Azure Pipelines prepares to deploy a pipeline stage and requires access to a protected resource
2. It enters an inner loop in which:

- 2.1. Azure Pipelines invokes the corresponding Azure Function check and waits for a decision
- 2.2. Your Azure Function evaluates the conditions necessary to permit access and returns a decision
- 2.3. If the Azure Function response body doesn't satisfy the *Success criteria* you defined and *Time between evaluations* is non-zero, Azure Pipelines reschedules another check evaluation after *Time between evaluations*

Configure synchronous checks

To use the synchronous mode for the Azure Function / REST API, in the check configuration panel, make sure you:

- Selected *ApiResponse* for the *Completion event* under *Advanced*
- Specified the *Success criteria* that define when to pass the check based on the check's response body
- Set *Time between evaluations* to *0* under *Control options*
- Set *Timeout* to how long you wish to wait for a check to succeed. If there's no positive decision and *Timeout* is reached, the corresponding pipeline stage is skipped

The *Time between evaluations* setting defines how long the check's decision is valid. A value of 0 means the decision is final. A non-zero value means the check will be retried after the configured interval, when its decision is negative. When multiple Approvals and Checks are running, the check is retried regardless of decision.

The maximum number of evaluations is defined by the ratio between the *Timeout* and *Time between evaluations* values. We recommend you ensure this ratio is at most 10.

Pass pipeline run information to checks

When configuring the check, you can specify the pipeline run information you wish to send to your Azure Function / REST API check. By default, Azure Pipeline adds the following information in the `Headers` of the HTTP call it makes.

- `"PlanUrl": "$(system.CollectionUri)"`
- `"ProjectId": "$(system.TeamProjectId)"`
- `"HubName": "$(system.HostType)"`
- `"PlanId": "$(system.PlanId)"`
- `"JobId": "$(system.JobId)"`
- `"TaskInstanceId": "$(system.TaskInstanceId)"`
- `"AuthToken": "$(system.AccessToken)"`

We don't recommend making calls into Azure DevOps in synchronous mode, because it will most likely cause your check to take more than 3 seconds to reply, so the check fails.

Error handling

Currently, Azure Pipelines evaluates a single check instance at most 2,000 times.

When to use asynchronous vs synchronous checks

Let's look at some example use cases and what are the recommended type of checks to use.

External information must be correct

Say you have a Service Connection to a production resource, and you wish to ensure that access to it's permitted only if the information in a ServiceNow ticket is correct. In this case, the flow would be as follows:

- You add an *asynchronous* Azure Function check that verifies the correctness of the ServiceNow ticket
- When a pipeline that wants to use the Service Connection runs:
  - Azure Pipelines calls your check function
  - If the information is incorrect, the check returns a negative decision. Assume this outcome
  - The pipeline stage fails
  - You update the information in the ServiceNow ticket
  - You restart the failed stage
  - The check runs again and this time it succeeds
  - The pipeline run continues

External approval must be granted

Say you have a Service Connection to a production resource, and you wish to ensure that access to it's permitted only after an administrator approved a ServiceNow ticket. In this case, the flow would be as follows:

- You add an *asynchronous* Azure Function check that verifies the ServiceNow ticket has been approved
- When a pipeline that wants to use the Service Connection runs:
  - Azure Pipelines calls your check function.
  - If the ServiceNow ticket isn't approved, the Azure Function sends an update to Azure Pipelines, and reschedules itself to check the state of the ticket in 15 minutes
  - Once the ticket is approved, the check calls back into Azure Pipelines with a positive decision
  - The pipeline run continues

Development process was followed

Say you have a Service Connection to a production resource, and you wish to ensure that access to it's permitted only if the code coverage is above 80%. In this case, the flow would be as follows:

- You write your pipeline in such a way that stage failures cause the build to fail
- You add an *asynchronous* Azure Function check that verifies the code coverage for the associated pipeline run
- When a pipeline that wants to use the Service Connection runs:
  - Azure Pipelines calls your check function

    - If the code coverage condition isn't met, the check returns a negative decision. Assume this outcome
    - The check failure causes your stage to fail, which causes your pipeline run to fail
- The engineering team adds the necessary unit tests to reach 80% code coverage
- A new pipeline run is triggered, and this time, the check passes
    - The pipeline run continues

Performance criteria must be met

Say you deploy new versions of your system in multiple steps, starting with a canary deployment. You wish to ensure your canary deployment's performance is adequate. In this case, the flow would be as follows:

- You add an *asynchronous* Azure Function check
- When a pipeline that wants to use the Service Connection runs:
    - Azure Pipelines calls your check function
    - The check starts a monitor of the canary deployment's performance
    - The check schedules multiple evaluation checkpoints, to see how the performance evolved
    - Once you gain enough confidence in the canary deployment's performance, your Azure Function calls back into Azure Pipelines with a positive decision
    - The pipeline run continues

Deployment reason must be valid

Say you have a Service Connection to a production environment resource, and you wish to ensure that access to it happens only for manually queued builds. In this case, the flow would be as follows:

- You add a *synchronous* Azure Function check that verifies that `Build.Reason` for the pipeline run is `Manual`
- You configure the Azure Function check to pass `Build.Reason` in its `Headers`
- You set the check's *Time between evaluations* to *0*, so failure or pass is final and no reevaluation is necessary
- When a pipeline that wants to use the Service Connection runs:
    - Azure Pipelines calls your check function
    - If the reason is other than `Manual`, the check fails, and the pipeline run fails

Check compliance

Invoke Azure Function and REST API checks now include rules to match recommended usage. Checks need to follow these rules depending on mode and the number of retries:

- **Asynchronous checks (Callback mode)**: Azure Pipelines doesn't retry asynchronous checks. You should provide a result asynchronously when an evaluation is final. For asynchronous checks to be considered compliant, the time interval between evaluations needs to be 0.

- **Synchronous checks (ApiResponse mode)**: The maximum number of retries for your check is 10. You can do set in a number of ways. For example, you can configure timeout to 20 and time interval between evaluations to 2. Or, you can configure timeout to 100 and time interval between evaluations to 10. But, if you configure timeout to 100 and set the time interval between evaluations to 2, your check won't be compliant because your asking for 50 retries. The ratio of timeout to time interval between evaluations should be less than or equal to 10.

Learn more about the rollout of check compliance.

Multiple checks

Before Azure Pipelines deploys a stage in a pipeline run, multiple checks may need to pass. A protected resource may have one or more Checks associated to it. A stage may use multiple protected resources. Azure Pipelines collects all the checks associated to each protected resource used in a stage and evaluates them concurrently.

A pipeline run is allowed to deploy to a stage only when *all* checks pass at the same time. A single final negative decision causes the pipeline to be denied access and the stage to fail.

When you use checks in the recommended way (asynchronous, with final states) makes their access decisions final, and eases understanding the state of the system.

Check out the Multiple Approvals and Checks section for examples.

Learn more

- Approvals and Checks
- Invoke Azure Function Task
- Invoke REST API Task

Deployment gates

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Gates allow automatic collection of health signals from external services and then promote the release when all the signals are successful or stop the deployment on timeout. Typically, gates are used in connection with incident management, problem management, change management, monitoring, and external approval systems.

Use cases

Some common use cases for deployment gates are:

- **Incident management**: Ensure certain criteria are met before proceeding with deployment. For example, ensure deployment occurs only if no priority zero bugs exist.
- **Seek approvals**: Notify external users such as legal departments, auditors, or IT managers about a deployment by integrating with other services such as Microsoft Teams or Slack and wait for their approvals.
- **Quality validation**: Query pipeline metrics such as pass rate or code coverage and deploy only if they are within a predefined threshold.
- **Security scan**: Perform security checks such as artifacts scanning, code signing, and policy checking. A deployment gate might initiate the scan and wait for it to complete, or just check for completion.
- **User experience relative to baseline**: Using product telemetry, ensure the user experience hasn't regressed from the baseline state. The user experience metrics before the deployment could be used as baseline.
- **Change management**: Wait for change management procedures in a system such as ServiceNow to complete before proceeding with deployment.
- **Infrastructure health**: Execute monitoring and validate the infrastructure against compliance rules after deployment, or wait for healthy resource utilization and a positive security report.

Most of the health parameters vary over time, regularly changing their status from healthy to unhealthy and back to healthy. To account for such variations, all the gates are periodically reevaluated until all of them are successful at the same time. The release execution and deployment does not proceed if all gates do not succeed in the same interval and before the configured timeout.

Define a gate for a stage

You can enable gates at the start of a stage (Pre-deployment conditions) or at the end of a stage (Post-deployment conditions) or for both. See Set up gates for more details.

The **Delay before evaluation** is a time delay at the beginning of the gate evaluation process that allows the gates to initialize, stabilize, and begin providing accurate results for the current deployment. see Gate evaluation flows for more details.

- For **pre-deployment gates**, the delay would be the time required for all bugs to be logged against the artifacts being deployed.
- For **post-deployment gates**, the delay would be the maximum of the time taken for the deployed app to reach a steady operational state, the time taken for execution of all the required tests on the deployed stage, and the time it takes for incidents to be logged after the deployment.

The following gates are available by default:

- **Invoke Azure function**: Trigger execution of an Azure function and ensure a successful completion. See Azure function task for more details.
- **Query Azure monitor alerts**: Observe the configured Azure monitor alert rules for active alerts. See Azure monitor task for more details.
- **Invoke REST API**: Make a call to a REST API and continue if it returns a successful response. See Invoke REST API task for more details.
- **Query Work items**: Ensure the number of matching work items returned from a query is within a threshold. See Query Work Items task for more details.
- **Security and compliance assessment**: Assess Azure Policy compliance on resources within the scope of a given subscription and resource group, and optionally at a specific resource level. See Check Azure Policy compliance task for more details.

You can also create your own gates with Marketplace extensions.

The evaluation options that apply to all the gates are:

- **Time between re-evaluation of gates**. The time interval between successive evaluations of the gates. At each sampling interval, new requests are sent concurrently to each gate and the new results are evaluated. It is recommended that the sampling interval is greater than the longest typical response time of the configured gates to allow time for all responses to be received for evaluation.
- **Timeout after which gates fail**. The maximum evaluation period for all gates. The deployment will be rejected if the timeout is reached before all gates succeed during the same sampling interval.
- **Gates and approvals**. Select the required order of execution for gates and approvals if you have configured both. For pre-deployment conditions, the default is to prompt for manual (user) approvals first, then evaluate gates afterwards. This saves the system from evaluating the gate functions if the release is rejected by the user. For post-deployment conditions, the default is to evaluate gates and prompt for manual approvals only when all gates are successful. This ensures the approvers have all the information required to approve.

See View approvals logs and Monitor and track deployments for more information on gates analytics.

Gate evaluation flow examples

The following diagram illustrates the flow of gate evaluation where, after the initial stabilization delay period and three sampling intervals, the deployment is approved.

The following diagram illustrates the flow of gate evaluation where, after the initial stabilization delay period, not all gates have succeeded at each sampling interval. In this case, after the timeout period expires, the deployment is rejected.

Resources

- Create custom gates
- Twitter sentiment as a release gate
- GitHub issues as a release gate

Related articles

- Release gates and approvals overview
- Use gates and approvals to control your deployment
- Add stages, dependencies, & conditions
- Release triggers

Use gates and approvals to control your deployment

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

By using a combination of manual deployment approvals, gates, and manual intervention in your release pipeline, you can quickly and easily configure your deployment to meet all the specific pre-deployment requirements for your workflow.

In this tutorial, you will learn about:

- ✓ Pre-deployment gates
- ✓ Manual intervention
- ✓ Manual validation
- ✓ Deployment logs

Prerequisites

- Complete the Define your multi-stage release pipeline tutorial.
- A work item query. Create a work item query in Azure Boards if you don't have one already.

Set up gates

You can use gates to ensure that the release pipeline meets specific criteria before deployment without requiring user intervention.

1. Select **Pipelines** > **Releases**, and then select your release pipeline. Select **Edit** to open the pipeline editor.

2. Select the pre-deployment icon for your stage, and then select the toggle button to enable **Gates**.

3. Specify the delay time before the added gates are evaluated. This time is to allow gate functions to initialize and stabilize before returning results.

4. Select **Add**, and then select **Query Work Items**.

5. Select an existing work item query from the dropdown menu. Depending on how many work items you expect the query to return, set your maximum and minimum thresholds.

6. Select the **Evaluation options** section, and then specify the timeout and sampling interval. The minimum values you can specify are 6-minutes timeout and 5-minutes sampling interval.

7. Select **Save** when you're done.

Set up manual intervention

Depending on the scenario, sometimes you may need to add manual intervention to your release pipeline. You can do this by adding the **Manual Intervention** task to your pipeline.

1. Select **Pipelines** > **Releases**. Select your release pipeline, and then select **Tasks** and choose your stage.

2. Select the ellipses (...), and then select **Add an agentless job**.

3. Drag and drop the agentless job to the top of your deployment process. Select the (+) sign, and then select **Add** the Manual Intervention task.

4. Enter a **Display name** and the instructions that will be displayed when the task is triggered. You can also specify a list of users to be notified and a timeout action (reject or resume) if no intervention occurred within the timeout period.

5. Select **Save** when you're done.

> ⓘ **Note**
>
> The **Manual Intervention** task can only be used in an **agentless job**.

Set up manual validation

You can use the Manual Validation task in your YAML pipeline to pause and wait for manual approval. Manual validation is especially useful in scenarios where you want to validate configuration settings or build packages before starting a computation-intensive job.

The `waitForValidation` job pauses the run and triggers a UI prompt to review and validate the task. The email addresses listed in `notifyUsers` receive a notification to approve or deny the pipeline run.

YAML

```yaml
pool:
   vmImage: ubuntu-latest

jobs:
- job: waitForValidation
  displayName: Wait for external validation
  pool: server
  timeoutInMinutes: 4320 # job times out in 3 days
  steps:
   - task: ManualValidation@0
     timeoutInMinutes: 1440 # task times out in 1 day
     inputs:
         notifyUsers: |
            someone@example.com
         instructions: 'Please validate the build configuration and resume'
         onTimeout: 'resume'
```

View deployment logs

Deployment logs are useful to debug deployment issues but you can also use them to audit your pipeline runs and verify approvals and how they were granted and by whom.

1. Select **Pipelines** > **Releases**, and then select your release pipeline.

2. This view will show you a live status of each stage in your pipeline. The QA stage in this example is pending intervention. Select **Resume**.

3. Enter your comment, and then select **Resume**.

4. The **QA** stage deployment succeeded and the pre-deployment approvals are triggered for the **Production** stage.

5. Select **Approve**, enter your comment and then select **Approve** to continue deployment.

6. The live status indicates that the gates are being processed for the **Production** stage before the release continues.

7. Return to your release pipeline, hover over your stage and then select **Logs** to view the deployment logs.

Related articles

- Release triggers
- Deploy pull request Artifacts
- Add stages, dependencies, & conditions

Deployment control using approvals

Article • 11/02/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

With Azure release pipelines, You can enable manual deployment approvals for each stage in a release pipeline to control your deployment workflow. When using approvals in your pipeline, the deployment is paused at each point where approval is required until the specified approver grants approval, rejects the release, or reassigns the approval to another user.

Deployment approvals

You can set up approvals at the start of a stage (pre-deployment approvals), at the end of a stage (post-deployment approvals), or for both.

Pre-deployment approvals

1. Select your classic release pipeline, and then select the **Pre-deployment conditions** icon and then click the toggle button to enable **Pre-deployment approvals**.

2. Add your **Approvers** and then choose the **Timeout** period. You can add multiple users or groups to the list of approvers. You can also select your **Approval policies** depending on your deployment workflow.

> **Note**
>
> Azure DevOps doesn't expand Azure Active Directory groups when delivering Notifications. If you must use Azure AD groups, we suggest that you add an email alias as an explicit recipient to your subscription and associate that alias with your AD group, if applicable to your scenario.

Post-deployment approvals

1. Select your classic release pipeline, and then select the **Post-deployment conditions** icon and then click the toggle button to enable **Post-deployment approvals**.

2. Add your **Approvers** and then choose the **Timeout** period. You can add multiple users or groups to the list of approvers. You can also select your **Approval policies** depending on your deployment workflow.

> **Note**
>
> Deployment approvers must have **View releases permissions**.

- **Approvers**: When a group is specified as approvers, only one user from that group is needed to approve, resume, or reject deployment.

- **Timeout**: If no approval is granted within the **Timeout** period, the deployment will be rejected.

- **Approval policies**:

- For added security, you can add this approval policy to prevent the user who requested the release from approving it. If you're experimenting with approvals, uncheck this option so that you can approve or reject your own deployments. See How are the identity variables set? to learn more about identity variables.
- This policy lets you enforce multi-factor authentication in the release approval flow. If this policy is checked it will prompt approvers to re-sign in before approving releases. This feature is only available in Azure DevOps Services for Microsoft Entra backed accounts only.
- Reduce user workload by automatically approving subsequent prompts if the specified user has already approved the deployment to a previous stage in the pipeline (applies to pre-deployment approvals only).

Approval notifications

You can enable notifications from your project settings to subscribe to release events. Emails are sent to approvers with links to the summary page where they can approve/reject the release.

1. From your project, select **Project settings**.

2. Select **Notifications** from the left navigation pane, and then select **New subscription** > **Release** to add a new event subscription.

Related articles

- Release gates and approvals
- Use gates and approvals to control your deployment
- Add stages, dependencies, & conditions
- Release triggers

Pipeline run sequence

Article • 03/06/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Runs represent one execution of a pipeline. During a run, the pipeline is processed, and agents process one or more jobs. A pipeline run includes jobs, steps, and tasks. Runs power both continuous integration (CI) and continuous delivery (CD) pipelines.

When you run a pipeline, many things happen under the covers. While you often won't need to know about them, occasionally it's useful to have the big picture. At a high level, Azure Pipelines will:

- Process the pipeline
- Request one or more agents to run jobs
- Hand off jobs to agents and collect the results

On the agent side, for each job, an agent will:

- Get ready for the job
- Run each step in the job
- Report results to Azure Pipelines

Jobs may succeed, fail, or be canceled. There are also situations where a job may not complete. Understanding how this happens can help you troubleshoot issues.

Let's break down each action one by one.

Process the pipeline

To turn a pipeline into a run, Azure Pipelines goes through several steps in this order:

1. First, expand templates and evaluate template expressions.
2. Next, evaluate dependencies at the stage level to pick the first stage(s) to run.
3. For each stage selected to run, two things happen:
    - All resources used in all jobs are gathered up and validated for authorization to run.
    - Evaluate dependencies at the job level to pick the first job(s) to run.
4. For each job selected to run, expand multi-configs (`strategy: matrix` or `strategy: parallel` in YAML) into multiple runtime jobs.
5. For each runtime job, evaluate conditions to decide whether that job is eligible to run.
6. Request an agent for each eligible runtime job.

As runtime jobs complete, Azure Pipelines will see if there are new jobs eligible to run. If so, steps 4 - 6 repeat with the new jobs. Similarly, as stages complete, steps 2 - 6 will be repeated for any new stages.

This ordering helps answer a common question: why can't I use certain variables in my template parameters? Step 1, template expansion, operates solely on the text of the YAML document. Runtime variables don't exist during that step. After step 1, template parameters have been resolved and no longer exist.

It also answers another common issue: why can't I use variables to resolve service connection / environment names? Resources are authorized before a stage can start running, so stage- and job-level variables aren't available. Pipeline-level variables can be used, but only those variables explicitly included in the pipeline. Variable groups are themselves a resource subject to authorization, so their data is likewise not available when checking resource authorization.

Request an agent

Whenever Azure Pipelines needs to run a job, it will ask the pool for an agent. (Server jobs are an exception, since they run on the Azure Pipelines server itself.) Microsoft-hosted and self-hosted agent pools work slightly differently.

Microsoft-hosted agent pool requests

First, the service checks on your organization's parallel jobs. It adds up all running jobs on all Microsoft-hosted agents and compares that with the number of parallel jobs purchased. If there are no available parallel slots, the job has to wait on a slot to free up.

Once a parallel slot is available, the job is routed to the requested agent type. Conceptually, the Microsoft-hosted pool is one giant, global pool of machines. (In reality, it's many different physical pools split by geography and operating system type.) Based on the `vmImage` (in YAML) or pool name (in the classic editor) requested, an agent is selected.

All agents in the Microsoft pool are fresh, new virtual machines that haven't run any pipelines before. When the job completes, the agent VM will be discarded.

Self-hosted agent pool requests

Similar to the Microsoft-hosted pool, the service first checks on your organization's parallel jobs. It adds up all running jobs on all self-hosted agents and compares that with the number of parallel jobs purchased. If there are no available parallel slots, the job has to wait on a slot to free up.

Once a parallel slot is available, the self-hosted pool is examined for a compatible agent. Self-hosted agents offer capabilities, which are strings indicating that particular software is installed or settings are configured. The pipeline has demands, which are the

capabilities required to run the job. If a free agent whose capabilities match the pipeline's demands can't be found, the job will continue waiting. If there are no agents in the pool whose capabilities match the demands, the job will fail.

Self-hosted agents are typically reused from run to run. For self-hosted agents, a pipeline job can have side effects such as warming up caches or having most commits already available in the local repo.

Prepare to run a job

Once an agent has accepted a job, it has some preparation work to do. The agent downloads (and caches for next time) all the tasks needed to run the job. It creates working space on disk to hold the source code, artifacts, and outputs used in the run. Then it begins running steps.

Run each step

Steps are run sequentially, one after another. Before a step can start, all the previous steps must be finished (or skipped).

Steps are implemented by tasks. Tasks themselves are implemented as Node.js or PowerShell scripts. The task system routes inputs and outputs to the backing scripts. It also provides some common services such as altering the system path and creating new pipeline variables.

Each step runs in its own process, isolating it from the environment left by previous steps. Because of this process-per-step model, environment variables aren't preserved between steps. However, tasks and scripts have a mechanism to communicate back to the agent: logging commands. When a task or script writes a logging command to standard out, the agent will take whatever action is requested.

There's an agent command to create new pipeline variables. Pipeline variables will be automatically converted into environment variables in the next step. In order to set a new variable `myVar` with a value of `myValue`, a script can do this:

Bash

```
echo '##vso[task.setVariable variable=myVar]myValue'
```

PowerShell

```
Write-Host "##vso[task.setVariable variable=myVar]myValue"
```

Report and collect results

Each step can report warnings, errors, and failures. Errors and warnings are reported to the pipeline summary page, marking the task as "succeeded with issues". Failures are also reported to the summary page, but they mark the task as "failed". A step is a failure if it either explicitly reports failure (using a `##vso` command) or ends the script with a non-zero exit code.

As steps run, the agent is constantly sending output lines to the service. That's why you can see a live feed of the console. At the end of each step, the entire output from the step is also uploaded as a log file. Logs can be downloaded once the pipeline has finished. Other items that the agent can upload include artifacts and test results. These are also available after the pipeline completes.

State and conditions

The agent keeps track of each step's success or failure. As steps succeed with issues or fail, the job's status will be updated. The job always reflects the "worst" outcome from

each of its steps: if a step fails, the job also fails.

Before running a step, the agent will check that step's condition to determine whether it should run. By default, a step will only run when the job's status is succeeded or succeeded with issues. Many jobs have cleanup steps that need to run no matter what else happened, so they can specify a condition of "always()". Cleanup steps might also be set to run only on cancellation. A succeeding cleanup step can't save the job from failing; jobs can never go back to success after entering failure.

Timeouts and disconnects

Each job has a timeout. If the job hasn't completed in the specified time, the server will cancel the job. It will attempt to signal the agent to stop, and it will mark the job as canceled. On the agent side, this means canceling all remaining steps and uploading any remaining results.

Jobs have a grace period known as the cancel timeout in which to complete any cancellation work. (Remember, steps can be marked to run even on cancellation.) After the timeout plus the cancel timeout, if the agent hasn't reported that work has stopped, the server will mark the job as a failure.

Because Azure Pipelines distributes work to agent machines, from time to time, agents may stop responding to the server. This can happen if the agent's host machine goes away (power loss, VM turned off) or if there's a network failure. To help detect these conditions, the agent sends a heartbeat message once per minute to let the server know it's still operating. If the server doesn't receive a heartbeat for five consecutive minutes, it assumes the agent won't come back. The job is marked as a failure, letting the user know they should retry the pipeline.

Manage runs through the CLI

Using the Azure DevOps CLI, you can list the pipeline runs in your project and view details about a specific run. You can also add and delete tags in your pipeline run.

Prerequisites

- You must have installed the Azure DevOps CLI extension as described in Get started with Azure DevOps CLI.
- Sign into Azure DevOps using `az login`.
- For the examples in this article, set the default organization using `az devops configure --defaults organization=YourOrganizationURL`.

List pipeline runs

List the pipeline runs in your project with the az pipelines runs list command. To get started, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines runs list [--branch]
                       [--org]
                       [--pipeline-ids]
                       [--project]
                       [--query-order {FinishTimeAsc, FinishTimeDesc,
QueueTimeAsc, QueueTimeDesc, StartTimeAsc, StartTimeDesc}]
                       [--reason {all, batchedCI, buildCompletion,
checkInShelveset, individualCI, manual, pullRequest, schedule, triggered,
userCreated, validateShelveset}]
                       [--requested-for]
                       [--result {canceled, failed, none,
partiallySucceeded, succeeded}]
                       [--status {all, cancelling, completed, inProgress,
none, notStarted, postponed}]
                       [--tags]
                       [--top]
```

Optional parameters

- **branch**: Filter by builds for this branch.
- **org**: Azure DevOps organization URL. You can configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **pipeline-ids**: Space-separated IDs of definitions for which to list builds.
- **project**: Name or ID of the project. You can configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.
- **query-order**: Define the order in which pipeline runs are listed. Accepted values are *FinishTimeAsc*, *FinishTimeDesc*, *QueueTimeAsc*, *QueueTimeDesc*, *StartTimeAsc*, and *StartTimeDesc*.
- **reason**: Only list builds for this specified reason. Accepted values are *batchedCI*, *buildCompletion*, *checkInShelveset*, *individualCI*, *manual*, *pullRequest*, *schedule*, *triggered*, *userCreated*, and *validateShelveset*.
- **requested-for**: Limit to the builds requested for a specified user or group.
- **result**: Limit to the builds with a specified result. Accepted values are *canceled*, *failed*, *none*, *partiallySucceeded*, and *succeeded*.

- **status**: Limit to the builds with a specified status. Accepted values are *all*, *cancelling*, *completed*, *inProgress*, *none*, *notStarted*, and *postponed*.
- **tags**: Limit to the builds with each of the specified tags. Space separated.
- **top**: Maximum number of builds to list.

Example

The following command lists the first three pipeline runs that have a status of **completed** and a result of **succeeded**, and returns the result in table format.

Azure CLI

```
az pipelines runs list --status completed --result succeeded --top 3 --
output table

Run ID    Number      Status     Result     Pipeline ID    Pipeline Name
Source Branch    Queued Time                 Reason
--------  ----------  ---------  ---------  -------------  -----------------
---------  ---------------  --------------------------  ------
125       20200124.1  completed  succeeded  12
Githubname.pipelines-java  master           2020-01-23 18:56:10.067588
manual
123       20200123.2  completed  succeeded  12
Githubname.pipelines-java  master           2020-01-23 11:55:56.633450
manual
122       20200123.1  completed  succeeded  12
Githubname.pipelines-java  master           2020-01-23 11:48:05.574742
manual
```

Show pipeline run details

Show the details for a pipeline run in your project with the az pipelines runs show command. To get started, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines runs show --id
                       [--open]
                       [--org]
                       [--project]
```

Parameters

- **id**: Required. ID of the pipeline run.
- **open**: Optional. Opens the build results page in your web browser.

- **org**: Azure DevOps organization URL. You can configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. You can configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.

Example

The following command shows details for the pipeline run with the ID **123** and returns the results in table format. It also opens your web browser to the build results page.

Azure CLI

```
az pipelines runs show --id 122 --open --output table

Run ID    Number      Status     Result     Pipeline ID    Pipeline Name
Source Branch    Queued Time                  Reason
--------  ----------  ---------  ---------  -------------  -----------------
---------  ---------------  --------------------------  --------
123       20200123.2  completed  succeeded  12
Githubname.pipelines-java  master           2020-01-23 11:55:56.633450
manual
```

Add tag to pipeline run

Add a tag to a pipeline run in your project with the az pipelines runs tag add command. To get started, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines runs tag add --run-id
                          --tags
                          [--org]
                          [--project]
```

Parameters

- **run-id**: Required. ID of the pipeline run.
- **tags**: Required. Tags to be added to the pipeline run (comma-separated values).
- **org**: Azure DevOps organization URL. You can configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as

default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.

- **project**: Name or ID of the project. You can configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.

Example

The following command adds the tag **YAML** to the pipeline run with the ID **123** and returns the result in JSON format.

Azure CLI

```
az pipelines runs tag add --run-id 123 --tags YAML --output json

[
  "YAML"
]
```

List pipeline run tags

List the tags for a pipeline run in your project with the az pipelines runs tag list command. To get started, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines runs tag list --run-id
                           [--org]
                           [--project]
```

Parameters

- **run-id**: Required. ID of the pipeline run.
- **org**: Azure DevOps organization URL. You can configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. You can configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.

Example

The following command lists the tags for the pipeline run with the ID **123** and returns the result in table format.

Azure CLI

```
az pipelines runs tag list --run-id 123 --output table

Tags
------
YAML
```

Delete tag from pipeline run

Delete a tag from a pipeline run in your project with the az pipelines runs tag delete command. To get started, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines runs tag delete --run-id
                             --tag
                             [--org]
                             [--project]
```

Parameters

- **run-id**: Required. ID of the pipeline run.
- **tag**: Required. Tag to be deleted from the pipeline run.
- **org**: Azure DevOps organization URL. You can configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. You can configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.

Example

The following command deletes the **YAML** tag from the pipeline run with ID **123**.

Azure CLI

```
az pipelines runs tag delete --run-id 123 --tag YAML
```

Access repositories, artifacts, and other resources

Article • 08/03/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

At run-time, each job in a pipeline may access other resources in Azure DevOps. For example, a job may:

- Check out source code from a Git repository
- Add a tag to the repository
- Access a feed in Azure Artifacts
- Upload logs from the agent to the service
- Upload test results and other artifacts from the agent to the service
- Update a work item

Azure Pipelines uses job access tokens to perform these tasks. A **job access token** is a security token that is dynamically generated by Azure Pipelines for each job at run time. The agent on which the job is running uses the job access token in order to access these resources in Azure DevOps. You can control which resources your pipeline has access to by controlling how permissions are granted to job access tokens.

The token's permissions are derived from (a) job authorization scope and (b) the permissions you set on project or collection build service account.

Job authorization scope

You can set the job authorization scope to be **collection** or **project**. By setting the scope to **collection**, you choose to let pipelines access all repositories in the collection or organization. By setting the scope to **project**, you choose to restrict access to only those repositories that are in the same project as the pipeline.

YAML

Job authorization scope can be set for the entire Azure DevOps organization or for a specific project.

- To set job authorization scope at the organization level for all projects, choose **Organization settings** > **Pipelines** > **Settings**.

- To set job authorization scope for a specific project, choose **Project settings** > **Pipelines** > **Settings**.

Enable one or more of the following settings. Enabling these settings are recommended, as it enhances security for your pipelines.

- **Limit job authorization scope to current project for non-release pipelines** - This setting applies to YAML pipelines and classic build pipelines, and does not apply to classic release pipelines.
- **Limit job authorization scope to current project for release pipelines** - This setting applies to classic release pipelines only.

> **Note**
>
> If the scope is set to **project** at the organization level, you cannot change the scope in each project.

> **Important**
>
> If the scope is not restricted at either the organization level or project level, then every job in your YAML pipeline gets a collection scoped job access token. In other words, your pipeline has access to any repository in any project of your organization. If an adversary is able to gain access to a single pipeline in a single project, they will be able to gain access to any repository in your organization. This is why, it is recommended that you restrict the scope at the highest level (organization settings) in order to contain the attack to a single project.

> **Note**
>
> If your pipeline is in a **public project**, then the job authorization scope is automatically restricted to **project** no matter what you configure in any setting. Jobs in a public project can access resources such as build artifacts or test results only within the project and not from other projects of the organization.

Protect access to repositories in YAML pipelines

In addition to the job authorization scope settings described in the previous section, Azure Pipelines provides a **Protect access to repositories in YAML pipelines** setting.

Pipelines can access any Azure DevOps repositories in authorized projects unless **Protect access to repositories in YAML pipelines** is enabled. With this option enabled, you can reduce the scope of access for all pipelines to only Azure DevOps repositories explicitly referenced by a `checkout` step or a `uses` statement in the pipeline job that uses that repository.

For more information, see Azure Repos Git repositories - Protect access to repositories in YAML pipelines.

> ⓘ **Important**
>
> **Protect access to repositories in YAML pipelines** is enabled by default for new organizations and projects created after May 2020.

Scoped build identities

Azure DevOps uses two built-in identities to execute pipelines.

- A **collection-scoped identity**, which has access to all projects in the collection (or organization for Azure DevOps Services)
- A **project-scoped identity**, which has access to a single project

These identities are allocated permissions necessary to perform build/release execution time activities when calling back to the Azure DevOps system. There are built-in default permissions, and you may also manage your own permissions as needed.

The **collection-scoped identity** name has the following format:

- `Project Collection Build Service ({OrgName})`
- For example, if the organization name is `fabrikam-tailspin`, this account has the name `Project Collection Build Service (fabrikam-tailspin)`.

The **project-scoped identity** name has the following format:

- `{Project Name} Build Service ({Org Name})`
- For example, if the organization name is `fabrikam-tailspin` and the project name is `SpaceGameWeb`, this account has the name `SpaceGameWeb Build Service (fabrikam-tailspin)`.

By default, the collection-scoped identity is used, unless configured otherwise as described in the previous Job authorization scope section.

Manage build service account permissions

One result of setting project-scoped access may be that the project-scoped identity may not have permissions to a resource that the collection-scoped one did have.

You may want to change the permissions of job access token in scenarios such as the following:

- You want your pipeline to access a feed that is in a different project.
- You want your pipeline to be restricted from changing code in the repository.
- You want your pipeline to be restricted from creating work items.

To update the permissions of the job access token:

- First, determine the job authorization scope for your pipeline. See the section above to understand job authorization scope. If the job authorization scope is **collection**, then the corresponding build service account to manage permissions on is **Project Collection Build Service (your-collection-name)**. If the job authorization scope is **project**, then the build service account to manage permissions on is **Your-project-name Build Service (your-collection-name)**.

- To restrict or grant additional access to **Project Collection Build Service (your-collection-name)**:
  - Select **Manage security** in the overflow menu on **Pipelines** page.
  - Under **Users**, select **Project Collection Build Service (your-collection-name)**.
  - Make any changes to the pipelines-related permissions for this account.
  - Navigate to organization settings for your Azure DevOps organization (or collection settings for your project collection).
  - Select **Permissions** under **Security**.
  - Under the **Users** tab, look for **Project Collection Build Service (your-collection-name)**.
  - Make any changes to the non-pipelines-related permissions for this account.
  - Since **Project Collection Build Service (your-collection-name)** is a user in your organization or collection, you can add this account explicitly to any resource - for e.g., to a feed in Azure Artifacts.

- To restrict or grant additional access to **Your-project-name Build Service (your-collection-name)**:
  - The build service account on which you can manage permissions will only be created after you run the pipeline once. Make sure that you already ran the pipeline once.
  - Select **Manage security** in the overflow menu on **Pipelines** page.
  - Under **Users**, select **Your-project-name Build Service (your-collection-name)**.

- Make any changes to the pipelines-related permissions for this account.
- Navigate to organization settings for your Azure DevOps organization (or collection settings for your project collection).
- Select **Permissions** under **Security**.
- Under the **Users** tab, look for **Your-project-name build service (your-collection-name)**.
- Make any changes to the non-pipelines-related permissions for this account.
- Since **Your-project-name Build Service (your-collection-name)** is a user in your organization or collection, you can add this account explicitly to any resource - for e.g., to a feed in Azure Artifacts.

Configure permissions for a project to access another project in the same project collection

In this example, the `fabrikam-tailspin/SpaceGameWeb` project-scoped build identity is granted permissions to access the `fabrikam-tailspin/FabrikamFiber` project.

1. In the **FabrikamFiber** project, navigate to **Project settings**, **Permissions**.

2. Create a new Group named *External Projects* and add the **SpaceGameWeb Build Service** account.

3. Choose **Users**, start to type in the name **SpaceGameWeb**, and select the **SpaceGameWeb Build Service** account. If you don't see any search results initially, select **Expand search**.

4. Grant the *View project-level information* permission for that user.

Example - Configure permissions to access another repo in the same project collection

In this example, the `fabrikam-tailspin/SpaceGameWeb` project-scoped build identity is granted permission to access the `FabrikamFiber` repository in the `fabrikam-tailspin/FabrikamFiber` project.

1. Follow the steps to grant the SpaceGameWeb project-scoped build identity permission to access the FabrikamFiber project.

2. In the **FabrikamFiber** project, navigate to **Project settings**, **Repositories**, **FabrikamFiber**.

3. Start to type in the name **SpaceGameWeb**, and select the **SpaceGameWeb Build Service** account.

4. Grant *Read* permissions for that user.

Example - Configure permissions to access other resources in the same project collection

In this example, the `fabrikam-tailspin/SpaceGameWeb` project-scoped build identity is granted permissions to access other resources in the `fabrikam-tailspin/FabrikamFiber` project.

1. Follow the steps to grant the SpaceGameWeb project-scoped build identity permission to access the FabrikamFiber project.

2. Configure the desired permissions for that user.

FAQ

How do I determine the job authorization scope of my YAML pipeline?

- If your project is a public project, the job authorization scope is always **project** regardless of any other settings.
- If the pipeline is in a private project, check the Pipeline settings under your Azure DevOps **Organization settings**:
  - If **Limit job authorization scope to current project for non-release pipelines** is enabled, then the scope is **project**.
  - If **Limit job authorization scope to current project for non-release pipelines** is not enabled, then check the Pipeline settings under your **Project settings** in Azure DevOps:
    - If **Limit job authorization scope to current project for non-release pipelines** is enabled, then the scope is **project**.
    - Otherwise, the scope is **collection**.

How do I determine the job authorization scope of my classic build pipeline?

- If the pipeline is in a public project, then the job authorization scope is **project** regardless of any other settings.

- If the pipeline is in a private project, check the Pipeline settings under your Azure DevOps **Organization settings**:
  - If **Limit job authorization scope to current project for non-release pipelines** is enabled, then the scope is **project**.
  - If **Limit job authorization scope to current project for non-release pipelines** is not enabled, then check the Pipeline settings under your **Project settings** in Azure DevOps:
    - If **Limit job authorization scope to current project for non-release pipelines** is enabled, then the scope is **project**.
    - If **Limit job authorization scope to current project for non-release pipelines** is not enabled, open the editor for the pipeline, and navigate to the **Options** tab.
      - If the **Build job authorization scope** is **Current project**, then scope is **project**.
      - Or else, scope is **collection**.

How do I determine the job authorization scope of my classic release pipeline?

- If the pipeline is in a public project, then the job authorization scope is **project** regardless of any other settings.

- If the pipeline is in a private project, check the Pipeline settings under your Azure DevOps **Organization settings**:
  - If **Limit job authorization scope to current project for release pipelines** is enabled, then the scope is **project**.
  - If **Limit job authorization scope to current project for release pipelines** is not enabled, then check the Pipeline settings under your **Project settings** in Azure DevOps:
    - If **Limit job authorization scope to current project for release pipelines** is enabled, then the scope is **project**.
    - Otherwise, the scope is **collection**.

Informational runs

Article • 05/31/2022

An informational run tells you Azure DevOps failed to retrieve a YAML pipeline's source code. Source code retrieval happens in response to external events, for example, a pushed commit. It also happens in response to internal triggers, for example, to check if there are code changes and start a scheduled run or not. Source code retrieval can fail for multiple reasons, with a frequent one being request throttling by the git repository provider. The existence of an informational run doesn't necessarily mean Azure DevOps was going to run the pipeline.

An informational run looks like in the following screenshot.

You can recognize an informational run by the following attributes:

- Status is `Canceled`
- Duration is `< 1s`
- Run name contains one of the following texts:
  - `Could not retrieve file content for {file_path} from repository {repo_name} hosted on {host} using commit {commit_sha}.`
  - `Could not retrieve content for object {commit_sha} from repository {repo_name} hosted on {host}.`
  - `Could not retrieve the tree object {tree_sha} from the repository {repo_name} hosted on {host}.`
  - `Could not find {file_path} from repository {repo_name} hosted on {host} using version {commit_sha}. One of the directories in the path contains too many files or subdirectories.`
- Run name generally contains the BitBucket / GitHub error that caused the YAML pipeline load to fail
- No stages / jobs / steps

When is an informational run created?

The first step of running a YAML pipeline is to retrieve its source code. When this step fails, the system creates an informational run. These runs are created only if the pipeline's code is in a GitHub or BitBucket repository.

Retrieving a pipeline's YAML code can fail due to:

- Repository provider experiencing an outage
- Request throttling
- Authentication issues
- Unable to retrieve the content of the pipeline's `.yml` file

A pipeline may run in response to:

- Pushes to branches in its `trigger` branch list
- Creating or updating Pull Requests that target branches in its `pr` branch list
- Scheduled runs
- Webhooks called
- Resource repository updates
- Resource external builds complete
- Resource pipelines complete
- New resource package versions are available
- Resource containers changes

Here's an example of when an informational run is created. Suppose you have a repo in your local BitBucket Server and a pipeline that builds the code in that repo. Assume you scheduled your pipeline to run every day, at 03:00. Now, imagine it's 03:00 and your BitBucket Server is experiencing an outage. Azure DevOps reaches out to your local BitBucket Server to fetch the pipeline's YAML code, but it can't, because of the outage. At this moment, the system creates an informational run, similar to the one shown in the previous screenshot.

Request throttling by the git repository provider is a frequent cause of Azure DevOps Services creating an informational run. Throttling occurs when Azure DevOps makes too many requests to the repository in a short amount of time. These requests can be due to a spike in commit activity, for example. Throttling issues are transitory.

Next Steps

Learn more about Triggers and building your GitHub or BitBucket repositories.

Pipeline reports

Article • 02/11/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Teams track their pipeline health and efficiency to ensure continuous delivery to their customers. You can gain visibility into your team's pipeline(s) using Pipeline analytics. The source of information for pipeline analytics is the set of runs for your pipeline. These analytics are accrued over a period of time, and form the basis of the rich insights offered. Pipelines reports show you metrics, trends, and can help you identify insights to improve the efficiency of your pipeline.

View pipeline reports

A summary of the pass rate and duration can be viewed in the **Analytics** tab of a pipeline. To drill into the trend and insights, click on the card to view the full report.

Pipeline pass rate report

The **Pipeline pass rate** report provides a granular view of the pipeline pass rate and its trend over time. You can also view which specific task failure contributes to a high number of pipeline run failures, and use that insight to fix the top failing tasks.

The report contain the following sections:

- **Summary**: Provides the key metrics of pass rate of the pipeline over the specified period. The default view shows data for 14 days, which you can modify.

- **Failure trend**: Shows the number of failures per day. This data is divided by stages if multiple stages are applicable for the pipeline.

- **Top failing tasks & their failed runs**: Lists the top failing tasks, their trend and provides pointers to their failed runs. Analyze the failures in the build to fix your failing task and improve the pass rate of the pipeline.

Pipeline duration report

The **Pipeline duration** report shows how long your pipeline typically takes to complete successfully. You can review the duration trend and analyze the top tasks by duration to optimize the duration of the pipeline.

Test failures report

The **Test failures** report provides a granular view of the top failing tests in the pipeline, along with the failure details. For more information on this report, see Test failures.

Filters

Pipelines reports can be further filtered by date range or branch.

- **Date range**: The default view shows data from the last 14 days. The filter helps change this range.

- **Branch filter**: View the report for a particular branch or a set of branches.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

Add widgets to a dashboard

Article • 11/03/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Widgets smartly format data to provide access to easily consumable data. You add widgets to your team dashboards to gain visibility into the status and trends occurring as you develop your software project.

Each widget provides access to a chart, user-configurable information, or a set of links that open a feature or function. You can add one or more charts or widgets to your dashboard. Up to 200 widgets total. You add several widgets at a time simply by selecting each one. See Manage dashboards to determine the permissions you need to add and remove widgets from a dashboard.

Prerequisites

- You must be a member of the project.
- Anyone with access to a project, including **Stakeholders**, can view dashboards.
- To add, edit, or manage a team dashboard, you must have **Basic** access, be a member of the team, a member of the Project Administrators group, or have dashboard permissions granted to you.
- To add, edit, or manage a project dashboard, you must have **Basic** access or have dashboard permissions granted to you for the select project dashboard.

> **Note**
>
> Data that's displayed within a chart or widget is subject to permissions granted to the signed in user. For example, if a user doesn't have permissions to view work items under an area path, then those items don't display in a query results widget in a dashboard. For more information, see **FAQs on Azure DevOps dashboards, charts, and reports, Access and permissions**.

> **Note**
>
> Widgets specific to a service are disabled if the service they depend on has been disabled. For example, if **Boards** is disabled, New Work item and all work tracking

Analytics widgets are disabled and won't appear in the **widget catalog**. If Analytics is disabled or not installed, then all Analytics widgets are disabled.

To re-enable a service, see **Turn an Azure DevOps service on or off**. For Analytics, see **enable or install Analytics**].

Open a dashboard

All dashboards are associated with a team. You must be a team administrator, project administrator, or a team member with permissions to modify a dashboard.

1. Open a web browser, connect to your project, and select **Overview** > **Dashboards**. The dashboard directory page opens.

If you need to switch to a different project, select the Azure DevOps logo to browse all projects.

2. Select the dashboard you want to modify.

Add a widget

To add widgets to the dashboard, select **Edit**.

The widget catalog automatically opens. Add all the widgets that you want and drag their tiles into the sequence you want.

When you're finished with your additions, select **Done editing** to exit dashboard editing. The widget catalog closes. You can then configure the widgets as needed.

> **Tip**
>
> When you're in dashboard edit mode, you can remove, rearrange, and configure widgets, as well as add new widgets. Once you leave edit mode, the widget tiles remain locked, reducing the chances of accidentally moving a widget.

To remove a widget, select **More actions** and select **Delete** from the menu.

Or, you can drag and drop a widget from the catalog onto the dashboard.

Add an Analytics widget

This example shows how to add the Velocity widget available from Analytics to a dashboard.

1. Select **Edit** to modify a dashboard. The widget catalog opens.
2. In the right pane search box, enter **Velocity** to quickly locate the Velocity widget within the widget catalog.

3. Select the widget, then **Add** to add it to the dashboard. Or, you can drag-and-drop it onto the dashboard.

4. Configure the widget.

Configure a widget

Most widgets support configuration, which might include specifying the title, setting the widget size, and other widget-specific variables.

To configure a widget, add the widget to a dashboard, select open the ··· menu, and select **Configure**.

For more information about configuring widgets, see the following articles:

- Burndown/burnup
- Cumulative flow
- Lead time or cycle time
- Velocity widget
- Test trend results

Move or delete a widget

To move a widget, you need to enable the dashboard edit mode. To delete a widget, select the delete option provided from the widget's **More actions** menu.

Select **Edit** to modify your dashboard. You can then add widgets or drag tiles to reorder their sequence on the dashboard.

To remove a widget, select the ··· actions icon and then **Delete** it.

When you're finished with your changes, select **Done Editing** to exit dashboard editing.

Copy a widget

You can copy a widget to the same dashboard or to another team dashboard. If you want to move widgets you configured to another dashboard, here's how you do it. Before you begin, add the dashboard that you want to copy or move the widget to. Once you copy the widget, you can delete it from the current dashboard.

To copy a configured widget to another team dashboard, select the ··· actions icon and select **Copy to dashboard** and then the dashboard to copy it to.

Widget size

Some widgets are presized and can't be changed. Others are configurable through their configuration dialog.

For example, the Chart for work items widget allows you to select an area size ranging from 2 x 2 to 4 x 4 (tiles).

Extensibility and Marketplace widgets

In addition to the widgets described in the Widget catalog, you can:

- Add widgets from the Marketplace.
- Create your own widgets using the Widget REST APIs.

Disabled Marketplace widgets

If your organization owner or project collection administrator disables a marketplace widget, the following notification displays:

To regain access to it, request your admin to reinstate or reinstall the widget.

Next steps

Review the widget catalog or Review Marketplace widgets

Related articles

- FAQs on Azure DevOps dashboards, charts, and reports
- Analytics-based widgets
- What is Analytics?
- Burndown guidance
- Cumulative flow & lead/cycle time guidance

Widgets based on Analytics data

Article • 02/24/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Analytics supports several dashboard widgets that take advantage of the power of the service. Using these widgets, you and your team can gain valuable insights into the health and status of your work.

You add an Analytics widget to a dashboard the same way you add any other type of widget. For details, see Add a widget to your dashboard.

Prerequisites

- Analytics widget data is calculated from the Analytics service. The Analytics service is enabled for all Azure DevOps organizations.
- To view Analytics data, you must have the **View analytics** project-level permission set to **Allow**. By default, this permission is set for all project members in all security groups. Users granted **Stakeholder** access or greater can view Analytics widgets.

> **Note**
>
> If **Boards** is disabled, then **Analytics views** will also be disabled and all widgets associated with work item tracking won't appear in the widget catalog and will become disabled. To re-enable a service, see **Turn an Azure DevOps service on or off**.

Burndown widget

The Burndown widget lets you display a trend of remaining work across multiple teams and multiple sprints. You can use it to create a release burndown, a bug burndown, or a burndown on any scope of work over time. It will help you answer questions like:

- Will we complete the scope of work by the targeted completion date? If not, what is the projected completion date?
- What kind of scope creep does my project have?
- What is the projected completion date for my project?

Burndown widget showing a release Burndown

To learn more, see Configure a burndown or burnup widget.

Burnup widget

The Burnup widget lets you display a trend of completed work across multiple teams and multiple sprints. You can use it to create a release burnup, a bug burnup, or a burnup on any scope of work over time. When completed work meets total scope, your project is done!

Burnup widget showing a release Burnup

To learn more, see Configure a burndown or burnup widget.

Sprint Burndown widget

The Analytics-based Sprint Burndown widget adds a team's burndown chart for a sprint to the dashboard. This widget supports several configuration options, including selecting a team, iteration, and time period. Teams use the burndown chart to mitigate risk and check for scope creep throughout the sprint cycle.

Sprint Burndown widget

To learn more, see Configure and monitor sprint burndown.

Cumulative Flow Diagram (CFD) widget

The CFD widget shows the count of work items (over time) for each column of a Kanban board. It allows you to see patterns in your team's development cycle over time. It will help you answer questions like:

- Is there a bottleneck in my process?
- Am I consistently delivering value to my users?

Cumulative flow diagram widget showing 30 days of data

To learn more, see Cumulative flow diagram widget.

Cycle Time widget

The Cycle Time widget will help you analyze the time it takes for your team to complete work items once they begin actively working on them. A lower cycle time is typically indicative of a healthier team process. Using the Cycle Time widget you can answer questions like:

- On average, how long does it take my team to build a feature or fix a bug?
- Are bugs costing my team many development hours?

Cycle Time widget showing 30 days of data

To learn more, see Cycle Time and lead time control charts.

Lead Time widget

The Lead Time widget will help you analyze the time it takes to deliver work from your backlog. Lead Time measures the total time elapsed from the creation of work items to their completion. Using the Lead Time widget, you can answer questions like:

- How long does it take for work requested by a customer to be delivered?
- Did work items take longer than usual to complete?

Lead Time widget showing 60 days of data

To learn more, see Cycle Time and lead time control charts.

Velocity widget

The Velocity widget will help you learn how much work your team can complete during a sprint. The widget shows the team's velocity by Story Points, work item count, or any custom field. It allows you to compare the work delivered against your plan and track work that's completed late. Using the Velocity widget, you can answer questions like:

- On average, what is the velocity of my team?
- Is my team consistently delivering what we planned?
- How much work can we commit to deliver in upcoming sprints?

Velocity widget showing 8 sprints of data based on Story Points

To learn more, see Configure and view Velocity widgets.

Test Results Trend (Advanced)

With the Test Results Trend (Advanced) widget, you can track the test quality of your pipelines over time. Tracking test quality and improving test collateral are essential tasks to maintaining a healthy DevOps pipeline.

The widget shows a trend of your test results for either build or release pipelines. You can track the daily count of tests, pass rates, and test duration. The highly configurable widget allows you to use it for a wide variety of scenarios.

You can find outliers in your test results and answer questions like:

- *What tests taking longer to run than usual?*
- *What micro services are affecting my pass rate?*

Test trend widget showing passed test results and pass rate for the last 7 days grouped by Priority

To learn more, see Configure a test results widget.

Manage your pipeline with Azure CLI

Article • 03/30/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

You can manage the pipelines in your organization using these `az pipelines` commands:

- az pipelines run: Run an existing pipeline
- az pipelines update: Update an existing pipeline
- az pipelines show: Show the details of an existing pipeline

These commands require either the name or ID of the pipeline you want to manage. You can get the ID of a pipeline using the az pipelines list command.

To get started, see Get started with Azure DevOps CLI.

Run a pipeline

You can queue (run) an existing pipeline with the az pipelines run command.

Azure CLI

```
az pipelines run [--branch]
                 [--commit-id]
                 [--folder-path]
                 [--id]
                 [--name]
                 [--open]
                 [--org]
                 [--project]
                 [--variables]
```

Parameters

- **branch**: Name of the branch on which the pipeline run is to be queued, for example, *refs/heads/main*.
- **commit-id**: Commit-id on which the pipeline run is to be queued.
- **folder-path**: Folder path of pipeline. Default is root level folder.
- **id**: Required if **name** is not supplied. ID of the pipeline to queue.
- **name**: Required if **ID** is not supplied, but ignored if **ID** is supplied. Name of the pipeline to queue.
- **open**: Open the pipeline results page in your web browser.

- **org**: Azure DevOps organization URL. You can configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. You can configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.
- **variables**: Space separated "name=value" pairs for the variables you would like to set.

Example

The following command runs the pipeline named **myGithubname.pipelines-java** in the branch **pipeline** and shows the result in table format.

Azure CLI

```
az pipelines run --name myGithubname.pipelines-java --branch pipeline --
output table

Run ID    Number      Status      Result    Pipeline ID    Pipeline Name
Source Branch    Queued Time               Reason
--------  ----------  ----------  --------  -------------  -----------------
----------  ---------------  --------------------------  --------
123       20200123.2  notStarted            12
myGithubname.pipelines-java  pipeline           2020-01-23 11:55:56.633450
manual
```

Update a pipeline

You can update an existing pipeline with the az pipelines update command. To get started, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines update [--branch]
                    [--description]
                    [--id]
                    [--name]
                    [--new-folder-path]
                    [--new-name]
                    [--org]
                    [--project]
                    [--queue-id]
                    [--yaml-path]
```

Parameters

- **branch**: Name of the branch on which the pipeline run is to be configured, for example, *refs/heads/main*.
- **description**: New description for the pipeline.
- **id**: Required if **name** is not supplied. ID of the pipeline to update.
- **name**: Required if **ID** is not supplied. Name of the pipeline to update.
- **new-folder-path**: New full path of the folder to which the pipeline is moved, for example, *user1/production_pipelines*.
- **new-name**: New updated name of the pipeline.
- **org**: Azure DevOps organization URL. You can configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. You can configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.
- **queue-id**: Queue ID of the agent pool where the pipeline needs to run.
- **yaml-path**: Path of the pipeline's yaml file in the repo.

Global parameters include `debug`, `help`, `only-show-errors`, `query`, `output`, and `verbose`.

> **Tip**
>
> There are also global parameters you can use such as `--output`. The `--output` parameter is available for all commands. The **table** value presents output in a friendly format. For more information, see **Output formats for Azure CLI commands**.

Example

The following command updates the pipeline with the **ID** of 12 with a new name and description and shows the result in table format.

Azure CLI

```
az pipelines update --id 12 --description "rename pipeline" --new-name
updatedname.pipelines-java --output table

ID    Name                          Status    Default Queue
```

```
----  -------------------------  --------  ------------------
12    updatedname.pipelines-java  enabled   Hosted Ubuntu 1604
```

Show pipeline

You can view the details of an existing pipeline with the az pipelines show command. To get started, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines show [--folder-path]
                  [--id]
                  [--name]
                  [--open]
                  [--org]
                  [--project]
                  [--query-examples]
                  [--subscription]
```

Parameters

- **folder-path**: Folder path of pipeline. Default is root level folder.
- **id**: Required if **name** is not supplied. ID of the pipeline to show details.
- **name**: Required if **name** is not supplied, but ignored if **ID** is supplied. Name of the pipeline to show details.
- **open**: Open the pipeline summary page in your web browser.
- **org**: Azure DevOps organization URL. You can configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. You can configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.
- **query examples**: Recommend a JMESPath string for you.
- **subscription**: Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

Example

The following command shows the details of the pipeline with the **ID** of 12 and returns the result in table format.

Azure CLI

```
az pipelines show --id 12 --output table

ID    Name                        Status    Default Queue
----  --------------------------  --------  ------------------
12    updatedname.pipelines-java  enabled   Hosted Ubuntu 1604
```

Next steps

You can customize your pipeline or learn more about configuring pipelines in the language of your choice:

- .NET Core
- Go
- Java
- Node.js
- Python
- Containers

FAQ

How do I programmatically create a build pipeline?

REST API Reference: Create a build pipeline

> **Note**
>
> You can also manage builds and build pipelines from the command line or scripts using the **Azure Pipelines CLI**.

Can I run multiple pipelines in Azure DevOps Services using a single command?

Currently, the Azure CLI and Azure APIs don't offer commands that run multiple pipelines from the command line. You can use Azure CLI commands to list all pipelines and definitions and provide a *single* release or build ID as a parameter. All commands are designed to work for independent runs of independent pipelines, and they require unique ID requests that allow only one, unique value. To learn about pipeline triggers, see Specify events that trigger pipelines.

Clone or import a pipeline

Article • 11/01/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

One approach to creating a pipeline is to copy an existing pipeline and use it as a starting point. For YAML pipelines, the process is as easy as copying the YAML from one pipeline to another. For pipelines created in the classic editor, the procedure depends on whether the pipeline to copy is in the same project as the new pipeline. If the pipeline to copy is in the same project, you can clone it, and if it is in a different project you can export it from that project and import it into your project.

For information in migrating a classic build pipeline to YAML using **Export to YAML**, see Migrate from classic pipelines.

Clone a pipeline

YAML

For YAML pipelines, the process for cloning is to copy the YAML from the source pipeline and use it as the basis for the new pipeline.

1. Go to the pipeline details for your pipeline, and choose **Edit**.

2. Copy the pipeline YAML from the editor, and paste it into the YAML editor for your new pipeline.

3. To customize your newly cloned pipeline, see Customize your pipeline.

Export and Import a pipeline

You can create a new classic pipeline by exporting an existing one and then importing it. This is useful in cases where the new pipeline has to be created in a separate project.

YAML

In a YAML pipeline, exporting from one project and importing into another is the same process as cloning. You can simply copy the pipeline YAML from the editor and paste it into the YAML editor for your new pipeline.

1. Go to the pipeline details for your pipeline, and choose **Edit**.

2. Copy the pipeline YAML from the editor, and paste it into the YAML editor for your new pipeline.

3. To customize your newly cloned pipeline, see Customize your pipeline.

Next steps

Learn to customize the pipeline you just cloned or imported.

Configure pipelines to support work tracking

Article • 02/24/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

To support integration and traceability across Azure DevOps Services with pipelines, you can configure several options. You can report pipeline status, copy the syntax for status badges, and set up automatic linking of work items to builds and releases.

Supported pipeline and work tracking integration features

Several features provide support for end-to-end traceability as user stories and features move through the development cycle. As with Azure Repos, you can link work items to pipeline objects with the following link types: *Build, Integrated in build*, and *Integrated in release*. Note, that the *Integrated in release environment* link can only be created by enabling the **Report release status to Boards** option in Classic release pipelines.

The following table summarizes the integration points between Azure Boards and Azure Pipelines. Options and configuration steps differ depending on whether you are configuring a YAML or Classic pipeline, and your Azure DevOps version. Most options are supported for pipelines run against an Azure Repos Git repository unless otherwise noted.

Feature	Description	Supported versions
Manually link work items to builds	You can link from a work item to builds within the same project or other projects within	

the organization. For details, see Link to work items from other objects.

All versions

View builds linked to from a work item

You can view all builds linked to from a work item, whether manual or automatically linked, from the Links tab. For details, see Link to work items from other objects, View list of linked objects.

All versions

Automatically link work items to builds

Required to populate the **Development** control with *Integrated in build* links. The work items or commits that are part of a release are computed from the versions of artifacts. For example, each build in Azure Pipelines is associated with a set of work items and commits. For details, see Automatically link work items later in this article.

YAML, Azure DevOps Server 2020 and later
Classic, TFS 2017.2 and later

Automatically link work items to releases and report deployment status to a work item (Classic only)

Required to populate **Deployment** control in work item form with *Integrated in release stage* links. For details, see Report deployment status to Boards later in this article.

Azure DevOps Server 2020 and later

View list of work items linked to a build or release

Review and open the work items included in a build or release.

YAML, Azure DevOps Server 2020 and later
Classic, TFS 2017.2 and later

Create work item on failure (Classic)

Automatically create a work item when a build fails, and optionally set values for work item fields. For details, see Create work item on failure later in this article.

TFS 2018 and later versions

Query Work Items task, ensure the number of matching work items returned from a query is within a threshold.

Use this task to ensure the number of matching items returned by a work item query is within the configured thresholds. For details, see Query Work Items task, Control deployments with gates and approvals.

Azure DevOps Server 2020 and later versions

Prerequisites

- To configure the integration options for a Classic release pipeline, you must have permissions to edit the release.
- To link work items to commits and pull requests, you must have your **Edit work items in this node** permissions set to **Allow** for the Area Path assigned to the work item. By default, the Contributors group has this permission set.
- To view work items, you must have your **View work items in this node** permissions set to **Allow** for the Area Path assigned to the work item.

Open pipeline settings, build options, or integration options

YAML

Open Pipeline settings

For YAML-defined release pipelines, you configure the integration through the **Pipeline settings** dialog.

1. Open the pipeline, choose ⋮ **More actions**, and then choose **Settings**.

The Pipeline Settings dialog appears. For details on automatic linking, see Automatically link work items later in this article.

Automatically link work items to builds or releases

By enabling automatic linking, you can track the builds or releases that have incorporated work without having to manually search through a large set of builds or releases. Each successful build associated with the work item automatically appears in the **Development** control of the work item form. Each release stage associated with the work item automatically appears in the **Deployment** control of the work item form.

YAML

1. Open **Pipeline settings** as describe in Open Pipeline settings.
2. Enable **Automatically link new work in this build**.

Once enabled, **Integrated in build** links are generated for all work items linked to the selected pull request with each release run.

What work items are included in automatic linking?

When developing your software, you can link work items when you create a branch, commit, or pull request. Or, you can initiate a branch, commit, or pull request from a work item, automatically linking these objects as described in Drive Git development from a work item. For example, here we create a new branch from the Cancel order form user story.

When automatically linking work items to builds, the following computations are made:

- For a first-time build:
  - Identify all work items linked to the branch, commits, and pull requests associated with the build.
- For subsequent builds:
  - Identify all work items associated with the current commit (C1) being built.
  - Identify all work items associated with the commit (C2) of the last successful build of the same source branch.
  - Identify all work items associated with the commits between C1 and C2 in the commit tree.

Create work item on build failure (Classic)

If a build pipeline fails, you can automatically create a work item to track getting the problem fixed. You can specify the work item type and set options to automatically assign it to the requestor or other fields. The requestor corresponds to the person that triggered the build.

> **Tip**
>
> The option to **Create work item on failure** is only supported for Classic pipelines. To accomplish this with a YAML pipeline, you can use a marketplace extension like **Create Bug on Release failure** or you can **implement it yourself using Azure CLI or REST API calls**.

1. Open pipeline build options as describe in Build properties.

2. Enable **Create work item on failure** and choose the type of work item to create. Optionally check the **Assign to requestor** checkbox to set the **Assign To** field and add fields to set within the work item to create.

   For example, here we choose the Bug work item type and specify the Priority and Tags fields and their values.

3. Save your pipeline.

To learn the reference name for a field, look it up from the Work item field index. For custom fields you add through an inherited process, Azure DevOps assigns a reference name based on friendly field name prefixed with *Custom.*. For example, you add a field

named **DevOps Triage**, the reference name is Custom.DevOpsTriage. No spaces are allowed within the reference name.

Get or enable a status badge

YAML

1. Open pipeline ⋮ **More Actions** and choose **Status badge**.

2. Choose the branch and scope of interest, and then choose **Copy to clipboard** to copy the image or Markdown syntax.

Related articles

- Define your multi-stage continuous deployment (CD) pipeline
- Link and view work items to builds and deployments
- Release pipelines (Classic) overview
- Configure repositories to support work tracking.
- How to retrieve all work items associated with a release pipeline using Azure DevOps API
- Drive Git development from a work item
- Link to work items from other objects
- End-to-end traceability
- Linking, traceability, and managing dependencies
- Link type reference

The pipeline default branch

Article • 10/27/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

A pipeline's default branch defines the pipeline version used for manual builds, scheduled builds, retention policies, and in pipeline resource triggers.

By default, a pipeline's default branch is the default branch of the repository.

To view and update the **Default branch for manual and scheduled builds** setting:

1. Go to the pipeline details for your pipeline, and choose **Edit**.

2. Choose **...** and select **Triggers**.

3. Select **YAML**, **Get sources**, and view the **Default branch for manual and scheduled builds** setting. If you change it, choose **Save** or **Save & queue** to save the change.

> **Important**
>
> Azure Pipelines loads a maximum of 2000 branches from a repository into the **Default branch for manual and scheduled builds**. If you don't see your desired branch in the list, type the desired branch name manually.

Create your Azure Pipelines ecosystem

Article • 08/30/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

You can select from the following languages and platforms to find guidance for building and deploying your app.

Build your app

Build your app with any of the following languages.

.NET Core

Anaconda

Android

ASP.NET

C/C++ with GCC

C/C++ with VC++

Containers

Go

Java

JavaScript and Node.js

PHP

Python

Ruby

UWP

Xamarin

Xcode

Deploy your app

Deploy your app to any of the following platforms.

Kubernetes

Azure Stack

Azure SQL database

Azure Web Apps

Linux VM

npm

NuGet

Virtual machine resources

Virtual machine resources

Web App for Containers

Windows VM

Build, test, and deploy .NET Core apps

Article • 12/01/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Use a pipeline to automatically build and test your .NET Core projects. Learn how to do the following tasks:

- Set up your build environment with Microsoft-hosted or self-hosted agents.
- Restore dependencies, build your project, and test with the .NET Core CLI task or a script.
- Use the publish code coverage task to publish code coverage results.
- Package and deliver your code with the .NET Core CLI task and the publish build artifacts task.
- Publish to a NuGet feed.
- Deploy your web app to Azure.

> **Note**
>
> For help with .NET Framework projects, see **Build ASP.NET apps with .NET Framework**.

Create your first pipeline

Are you new to Azure Pipelines? If so, then we recommend you try the following section first.

Create a .NET project

If you don't have a .NET project to work with create a new one. Start by installing the latest .NET 6.0 SDK.

1. Open a terminal window

2. Create a project directory and navigate into it.

3. Create a new .NET 6 webapp.

```
.NET CLI
```

```
dotnet new webapp -f net6.0
```

4. From the same terminal session, run the application locally using the dotnet run command from your project directory.

   .NET CLI

   ```
   dotnet run
   ```

5. Once the application has start press Ctrl-C to shut it down.

Connect git repo to GitHub

1. From the project directory, create a local git repository and commit the application code to the main branch.
2. Connect your local Git repo to a GitHub repo.

Sign in to Azure Pipelines

Sign-in to Azure Pipelines. After you sign in, your browser goes to `https://dev.azure.com/my-organization-name` and displays your Azure DevOps dashboard.

Within your selected organization, create a *project*. If you don't have any projects in your organization, you see a **Create a project to get started** screen. Otherwise, select the **New Project** button in the upper-right corner of the dashboard.

Create the pipeline

1. Sign-in to your Azure DevOps organization and go to your project.

2. Go to **Pipelines**, and then select **New pipeline** or **Create pipeline** if creating your first pipeline.

3. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.

4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.

5. When you see the list of repositories, select your repository.

6. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.

Confgure and run the pipeline

1. When the **Configure** tab appears, select **Show more** and select ASP.NET Core pipeline template from the list.

2. Examine your new pipeline to see what the YAML does. When you're ready, select **Save and run**.

3. After you're happy with the message, commit the new *azure-pipelines.yml* file to your repository by selecting **Save and run** again.

   If you want to watch your pipeline in action, select the job in the **Jobs** section.

You now have a working YAML pipeline (`azure-pipelines.yml`) in your repository that's ready for you to customize! When you're ready to make changes to your pipeline, select it in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file. Read further to learn some of the more common ways to customize your pipeline.

Build environment

Use Azure Pipelines to build your .NET Core projects. Build your projects on Windows, Linux, or macOS without the need to set up infrastructure. The Microsoft-hosted agents in Azure Pipelines include several preinstalled versions of the .NET Core SDKs.

Ubuntu is set here in the YAML file.

YAML

```yaml
pool:
  vmImage: 'ubuntu-latest'
```

See Microsoft-hosted agents for a complete list of images and Pool for further examples.

The Microsoft-hosted agents don't include some of the older versions of the .NET Core SDK. They also don't typically include prerelease versions. If you need these kinds of SDKs on Microsoft-hosted agents, add the UseDotNet@2 task to your YAML file.

To install 6.0.x SDK for building, add the following snippet:

YAML

```yaml
steps:
- task: UseDotNet@2
  inputs:
    version: '5.x'
```

Windows agents already include a .NET Core runtime. To install a newer SDK, set `performMultiLevelLookup` to `true` in the following snippet:

YAML

```yaml
steps:
- task: UseDotNet@2
  displayName: 'Install .NET Core SDK'
  inputs:
    version: 5.x
    performMultiLevelLookup: true
    includePreviewVersions: true # Required for preview versions
```

Tip

To save the cost of running the tool installer, you can set up a **self-hosted agent**. See **Linux**, **macOS**, or **Windows**. You can also use self-hosted agents to save additional time if you have a large repository or you run incremental builds. A self-hosted agent can also help you in using the preview or private SDKs that aren't officially supported by Azure DevOps or you have available on your corporate or on-premises environments only.

Restore dependencies

NuGet is a popular way to depend on code that you don't build. You can download NuGet packages and project-specific tools that are specified in the project file by

running the `dotnet restore` command either through the .NET Core task or directly in a script in your pipeline.

You can download NuGet packages from Azure Artifacts, NuGet.org, or some other external or internal NuGet repository. The **.NET Core** task is especially useful to restore packages from authenticated NuGet feeds. If your feed is in the same project as your pipeline, you don't need to authenticate.

This pipeline uses an artifact feed for `dotnet restore` in the .NET Core CLI task.

YAML

```yaml
trigger:
- main

pool:
  vmImage: 'windows-latest'steps:
- task: UseDotNet@2
  displayName: 'Install .NET Core SDK'
  inputs:
    version: 6.x
    performMultiLevelLookup: true
    includePreviewVersions: true # Required for preview versions

variables:
  buildConfiguration: 'Release'

steps:
- task: DotNetCoreCLI@2
  inputs:
    command: 'restore'
    feedsToUse: 'select'
    vstsFeed: 'my-vsts-feed' # A series of numbers and letters

- task: DotNetCoreCLI@2
  inputs:
    command: 'build'
    arguments: '--configuration $(buildConfiguration)'
  displayName: 'dotnet build $(buildConfiguration)'
```

`dotnet restore` internally uses a version of `NuGet.exe` packaged with the .NET Core SDK. `dotnet restore` can only restore packages specified in the .NET Core project `.csproj` files. If you also have a Microsoft .NET Framework project in your solution or use `package.json` to specify your dependencies, use the **NuGet** task to restore those dependencies.

In .NET Core SDK version 2.0 and newer, packages get restored automatically when running other commands such as `dotnet build`. However, you might still need to use

the **.NET Core** task to restore packages if you use an authenticated feed.

Your builds might sometimes fail because of connection issues when you restore packages from NuGet.org. You can use Azure Artifacts with upstream sources to cache the packages. The credentials of the pipeline are automatically used when it connects to Azure Artifacts. These credentials are typically derived from the **Project Collection Build Service** account.

If you want to specify a NuGet repository, put the URLs in a `NuGet.config` file in your repository. If your feed is authenticated, manage its credentials by creating a NuGet service connection in the **Services** tab under **Project Settings**.

If you use Microsoft-hosted agents, you get a new machine every time your run a build, which restores the packages with each run. Restoration can take a significant amount of time. To mitigate, you can either use Azure Artifacts or a self-hosted agent with the benefit of using the package cache.

To restore packages from an external custom feed, use the following **.NET Core** task:

YAML

```yaml
# do this before your build tasks
steps:
- task: DotNetCoreCLI@2
  displayName: Restore
  inputs:
    command: restore
    projects: '**/*.csproj'
    feedsToUse: config
    nugetConfigPath: NuGet.config    # Relative to root of the repository
    externalFeedCredentials: <Name of the NuGet service connection>
# ...
```

For more information about NuGet service connections, see publish to NuGet feeds.

Build your project

Build your .NET Core project either by running the `dotnet build` command in your pipeline or by using the .NET Core task.

To build your project by using the .NET Core task, add the following snippet to your `azure-pipelines.yml` file:

YAML

```yaml
steps:
- task: DotNetCoreCLI@2
  displayName: Build
  inputs:
    command: build
    projects: '**/*.csproj'
    arguments: '--configuration $(buildConfiguration)' # Update this to
match your need
```

You can run any custom dotnet command in your pipeline. The following example shows how to install and use a .NET global tool, dotnetsay:

YAML

```yaml
steps:
- task: DotNetCoreCLI@2
  displayName: 'Install dotnetsay'
  inputs:
    command: custom
    custom: tool
    arguments: 'install -g dotnetsay'
```

Run your tests

If you have test projects in your repository, then use the **.NET Core** task to run unit tests by using testing frameworks like MSTest, xUnit, and NUnit.The test project must reference Microsoft.NET.Test.SDK version 15.8.0 or higher. Test results get automatically published to the service. These results are available to you in the build summary and can be used for troubleshooting failed tests and test-timing analysis.

Add the following snippet to your `azure-pipelines.yml` file:

YAML

```yaml
steps:
# ...
# do this after other tasks such as building
- task: DotNetCoreCLI@2
  inputs:
    command: test
    projects: '**/*Tests/*.csproj'
    arguments: '--configuration $(buildConfiguration)'
```

An alternative is to run the `dotnet test` command with a specific logger and then use the **Publish Test Results** task:

YAML

```yaml
steps:
# ...
# do this after your tests have run
- script: dotnet test <test-project> --logger trx
- task: PublishTestResults@2
  condition: succeededOrFailed()
  inputs:
    testRunner: VSTest
    testResultsFiles: '**/*.trx'
```

Collect code coverage

If you're building on the Windows platform, code coverage metrics can be collected by using the built-in coverage data collector. The test project must reference Microsoft.NET.Test.SDK version 15.8.0 or higher. If you use the **.NET Core** task to run tests, coverage data is automatically published to the server. The **.coverage** file can be downloaded from the build summary for viewing in Visual Studio.

Add the following snippet to your `azure-pipelines.yml` file:

YAML

```yaml
steps:
# ...
# do this after other tasks such as building
- task: DotNetCoreCLI@2
  inputs:
    command: test
    projects: '**/*Tests/*.csproj'
    arguments: '--configuration $(buildConfiguration) --collect "Code
coverage"'
```

If you choose to run the `dotnet test` command, specify the test results logger and coverage options. Then use the Publish Test Results task:

YAML

```yaml
steps:
# ...
# do this after your tests have run
- script: dotnet test <test-project> --logger trx --collect "Code coverage"
- task: PublishTestResults@2
  inputs:
```

```
    testRunner: VSTest
    testResultsFiles: '**/*.trx'
```

Collect code coverage metrics with Coverlet

If you're building on Linux or macOS, you can use Coverlet or a similar tool to collect code coverage metrics.

You can publish code coverage results to the server with the Publish Code Coverage Results task. The coverage tool must be configured to generate results in Cobertura or JaCoCo coverage format.

To run tests and publish code coverage with Coverlet, do the following tasks:

- Add a reference to the `coverlet.msbuild` NuGet package in your test project(s) for .NET projects before .NET 5. For .NET 5, add a reference to the `coverlet.collector` NuGet package.
- Add the following snippet to your `azure-pipelines.yml` file:

.NET >= 5

YAML

```yaml
- task: UseDotNet@2
  inputs:
    version: '6.x'
    includePreviewVersions: true # Required for preview versions

- task: DotNetCoreCLI@2
  displayName: 'dotnet build'
  inputs:
    command: 'build'
    configuration: $(buildConfiguration)

- task: DotNetCoreCLI@2
  displayName: 'dotnet test'
  inputs:
    command: 'test'
    arguments: '--configuration $(buildConfiguration) --collect:"XPlat
Code Coverage" --
DataCollectionRunSettings.DataCollectors.DataCollector.Configuration.For
mat=cobertura'
    publishTestResults: true
    projects: 'MyTestLibrary' # update with your test project directory

- task: PublishCodeCoverageResults@1
  displayName: 'Publish code coverage report'
  inputs:
```

```
    codeCoverageTool: 'Cobertura'
    summaryFileLocation:
'$(Agent.TempDirectory)/**/coverage.cobertura.xml'
```

Package and deliver your code

Upload the build output to Azure Pipelines. You can create and publish a NuGet package, or package the build output into a .zip file to deploy to a web application.

Publish artifacts to Azure Pipelines

To publish the output of your .NET **build**, do the following tasks:

- Run `dotnet publish --output $(Build.ArtifactStagingDirectory)` on CLI or add the DotNetCoreCLI@2 task with publish command.
- Publish the artifact by using Publish artifact task.

Add the following snippet to your `azure-pipelines.yml` file:

YAML

```yaml
steps:

- task: DotNetCoreCLI@2
  inputs:
    command: publish
    publishWebProjects: True
    arguments: '--configuration $(BuildConfiguration) --output
$(Build.ArtifactStagingDirectory)'
    zipAfterPublish: True

# this code takes all the files in $(Build.ArtifactStagingDirectory) and
uploads them as an artifact of your build.
- task: PublishPipelineArtifact@1
  inputs:
    targetPath: '$(Build.ArtifactStagingDirectory)'
    artifactName: 'myWebsiteName'
```

> **Note**
>
> The `dotNetCoreCLI@2` task has a `publishWebProjects` input that is set to **true** by default. This task publishes *all* web projects in your repo by default. You can find more help and information in the **open source task on GitHub**.

To copy more files to Build directory before publishing, use Utility: copy files.

Publish to a NuGet feed

To create and publish a NuGet package, add the following snippet:

YAML

```yaml
steps:
# ...
# do this near the end of your pipeline in most cases
- script: dotnet pack /p:PackageVersion=$(version)  # define version
variable elsewhere in your pipeline
- task: NuGetAuthenticate@0
  input:
    nuGetServiceConnections: '<Name of the NuGet service connection>'
- task: NuGetCommand@2
  inputs:
    command: push
    nuGetFeedType: external
    publishFeedCredentials: '<Name of the NuGet service connection>'
    versioningScheme: byEnvVar
    versionEnvVar: version
```

For more information about versioning and publishing NuGet packages, see publish to NuGet feeds.

Deploy a web app

To create a .zip file archive that's ready for publishing to a web app, add the following snippet:

YAML

```yaml
steps:
# ...
# do this after you've built your app, near the end of your pipeline in most
cases
# for example, you do this before you deploy to an Azure web app on Windows
- task: DotNetCoreCLI@2
  inputs:
    command: publish
    publishWebProjects: True
    arguments: '--configuration $(BuildConfiguration) --output
$(Build.ArtifactStagingDirectory)'
    zipAfterPublish: True
```

To publish this archive to a web app, see Azure Web Apps deployment.

Build an image and push to container registry

For your app, you can also build an image and push it to a container registry.

Troubleshoot

If you can build your project on your development machine, but you're having trouble building it on Azure Pipelines, explore the following potential causes and corrective actions:

- We don't install prerelease versions of the .NET Core SDK on Microsoft-hosted agents. After a new version of the .NET Core SDK gets released, it can take a few weeks to roll out to all the Azure Pipelines data centers. You don't have to wait for this rollout to complete. You can use the **.NET Core Tool Installer** to install the version you want of the .NET Core SDK on Microsoft-hosted agents.

- Check the .NET Core SDK versions and runtime on your development machine and make sure they match the agent. You can include a command-line script `dotnet --version` in your pipeline to print the version of the .NET Core SDK. Either use the **.NET Core Tool Installer** to deploy the same version on the agent, or update your projects and development machine to the newer version of the .NET Core SDK.

- You might be using some logic in the Visual Studio IDE that isn't encoded in your pipeline. Azure Pipelines runs each of the commands you specify in the tasks one after the other in a new process. Examine the logs from the pipelines build to see the exact commands that ran as part of the build. Repeat the same commands in the same order on your development machine to locate the problem.

- If you have a mixed solution that includes some .NET Core projects and some .NET Framework projects, you should also use the **NuGet** task to restore packages specified in `packages.config` files. Add the **MSBuild** or **Visual Studio Build** task to build the .NET Framework projects.

- Your builds might fail intermittently while restoring packages: either NuGet.org is having issues, or there are networking problems between the Azure data center and NuGet.org. You can explore whether using Azure Artifacts with NuGet.org as an upstream source improves the reliability of your builds, as it's not in our control.

- Occasionally, when we roll out a new version of the .NET Core SDK or Visual Studio, your build might break. For example, if a newer version or feature of the NuGet tool gets shipped with the SDK. To isolate this issue, use the **.NET Core Tool Installer** task to specify the version of the .NET Core SDK used in your build.

FAQ

Q: Where can I learn more about Azure Artifacts and the TFS Package Management service?

A: Package Management in Azure Artifacts

Q: Where can I learn more about .NET Core commands?

A: .NET Core CLI tools

Q: Where can I learn more about running tests in my solution?

A: Unit testing in .NET Core projects

Q: Where can I learn more about tasks?

A: Build and release tasks

Build ASP.NET apps with .NET Framework

Article • 05/30/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> **Note**
>
> This article focuses on building .NET Framework projects with Azure Pipelines. For help with .NET Core projects, see **.NET Core**.

Create your first pipeline

> Are you new to Azure Pipelines? If so, then we recommend you try this section before moving on to other sections.

Get the code

Fork the following repo at GitHub:

```
https://github.com/Azure-Samples/app-service-web-dotnet-get-started
```

The sample app is a Visual Studio solution that uses .NET 4.8.

Sign in to Azure Pipelines

Sign-in to Azure Pipelines. After you sign in, your browser goes to `https://dev.azure.com/my-organization-name` and displays your Azure DevOps dashboard.

Within your selected organization, create a *project*. If you don't have any projects in your organization, you see a **Create a project to get started** screen. Otherwise, select the **New Project** button in the upper-right corner of the dashboard.

- After you have the sample code in your own repository, create a pipeline using the instructions in Create your first pipeline and select the **ASP.NET** template. This

automatically adds the tasks required to build the code in the sample repository.

- Save the pipeline and queue a build to see it in action.

Build environment

You can use Azure Pipelines to build your .NET Framework projects without needing to set up any infrastructure of your own. The Microsoft-hosted agents in Azure Pipelines have several released versions of Visual Studio pre-installed to help you build your projects.

- Use `windows-2022` for Windows Server 2022 with Visual Studio 2022

You can also use a self-hosted agent to run your builds. This is helpful if you have a large repository and you want to avoid downloading the source code to a fresh machine for every build.

Restore dependencies

You can use the NuGet task to install and update NuGet package dependencies. You can also download NuGet packages from Azure Artifacts, NuGet.org, or some other external or internal NuGet repository with the NuGet task.

This code restores a solution from a project-scoped feed in the same organization.

YAML

```yaml
# Restore from a project scoped feed in the same organization
- task: NuGetCommand@2
  inputs:
    command: 'restore'
    feedsToUse: 'select'
    vstsFeed: 'my-project/my-project-scoped-feed'
    includeNuGetOrg: false
    restoreSolution: '**/*.sln'
```

Quickstart - Use Azure Pipelines to build and publish a Node.js package

Article • 04/17/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

You can use an Azure DevOps pipeline to build, deploy, and test JavaScript apps.

This quickstart walks through how to use a pipeline to create a Node.js package with Node Package Manager (npm) and publish a pipeline artifact.

Prerequisites

You must have the following items in Azure DevOps:

- A GitHub account where you can create a repository. Create one for free.
- An Azure DevOps organization and project. Create one for free.
- An ability to run pipelines on Microsoft-hosted agents. You can either purchase a parallel job or you can request a free tier.

1 - Fork the sample code

Fork the following sample Express.js server app at GitHub.

```
https://github.com/Azure-Samples/js-e2e-express-server
```

2 - Create your pipeline

1. Sign in to Azure Pipelines. Your browser will go to `https://dev.azure.com/my-organization-name` and display your Azure DevOps dashboard.

2. Go to your project and select **Pipelines** > **Create a new pipeline**.

3. Select **GitHub** as the location of your source code.

4. If you're redirected to GitHub to sign in, enter your GitHub credentials.

5. When the list of repositories appears, select your Node.js sample repository.
6. Azure Pipelines analyzes the code in your repository and recommends the `Node.js` template for your pipeline. Select that template.
7. Azure Pipelines generates a YAML file for your pipeline. Select **Save and run** > **Commit directly to the main branch**, and then choose **Save and run** again.
8. A new run starts. Wait for the run to finish.

When you're done, you have a working YAML file *azure-pipelines.yml* in your repository that's ready for you to customize.

3 - Build your package and publish an artifact

1. **Edit** your *azure-pipelines.yml* file.
2. Update the Node.js Tool Installer task to use Node.js version 16 LTS.

YAML

```yaml
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: NodeTool@0
  inputs:
    versionSpec: '16.x'
  displayName: 'Install Node.js'

- script: |
    npm install
  displayName: 'npm install'

- script: |
    npm run build
  displayName: 'npm build'
```

3. Add new tasks to your pipeline to copy your npm package, package.json, and to publish your artifact. The Copy Files task copies files from local path on the agent where your source code files are downloaded and saves files to a local path on the agent where any artifacts are copied to before being pushed to their destination. Only the `src` and `public` folders get copies. The Publish Pipeline Artifact task

downloads the files from the earlier Copy Files tasks and makes them available as pipeline artifacts that will be published with your pipeline build.

YAML

```yaml
- task: CopyFiles@2
  inputs:
    sourceFolder: '$(Build.SourcesDirectory)'
    contents: |
       src/*
       public/*
    targetFolder: '$(Build.ArtifactStagingDirectory)'
  displayName: 'Copy project files'

- task: PublishPipelineArtifact@1
  inputs:
    artifactName: e2e-server
    targetPath: '$(Build.ArtifactStagingDirectory)'
    publishLocation: 'pipeline'
  displayName: 'Publish npm artifact'
```

4 - Run your pipeline

Save and run your pipeline. After your pipeline runs, verify that the job ran successfully and that you see a published artifact.

Next steps

Congratulations, you've successfully completed this quickstart!

Configure JavaScript

Customize JavaScript for Azure Pipelines

Article • 11/02/2023

You can use Azure Pipelines to build your JavaScript apps without having to set up any infrastructure of your own. Tools that you commonly use to build, test, and run JavaScript apps - like npm, Node, Yarn, and Gulp - get pre-installed on Microsoft-hosted agents in Azure Pipelines.

For the version of Node.js and npm that is preinstalled, refer to Microsoft-hosted agents. To install a specific version of these tools on Microsoft-hosted agents, add the **Node Tool Installer** task to the beginning of your process. You can also use a self-hosted agent.

To create your first pipeline with JavaScript, see the JavaScript quickstart.

Use a specific version of Node.js

If you need a version of Node.js and npm that isn't already installed on the Microsoft-hosted agent, use the Node tool installer task. Add the following snippet to your `azure-pipelines.yml` file.

> **Note**
>
> The hosted agents are regularly updated, and setting up this task results in spending significant time updating to a newer minor version every time the pipeline is run. Use this task only when you need a specific Node version in your pipeline.

YAML

```yaml
- task: NodeTool@1
  inputs:
    version: '16.x' # replace this value with the version that you need for your project
```

To update just the npm tool, run the `npm i -g npm@version-number` command in your build process.

Use multiple node versions

You can build and test your app on multiple versions of Node with the Node tool installer task.

YAML

```yaml
pool:
  vmImage: 'ubuntu-latest'
strategy:
  matrix:
    node_16_x:
      node_version: 16.x
    node_13_x:
      node_version: 18.x

steps:
- task: NodeTool@1
  inputs:
    version: $(node_version)

- script: npm install
```

Install tools on your build agent

If you have tools that are development dependencies in your project `package.json` or `package-lock.json` file, install your tools and dependencies through npm. The exact version of the tools gets defined in the project, isolated from other versions that exist on the build agent.

Use a script or the npm task.

Use a script to install with package.json

YAML

```yaml
- script: npm install --only=dev
```

Use the npm task to install with package.json

YAML

```yaml
- task: Npm@1
  inputs:
     command: 'install'
```

Run tools installed this way by using the npm `npx` package runner, which detects tools installed this way in its path resolution. The following example calls the `mocha` test runner but looks for the version installed as a development dependency before using a globally installed (through `npm install -g`) version.

YAML

```yaml
- script: npx mocha
```

To install tools that your project needs but that aren't set as development dependencies in `package.json`, call `npm install -g` from a script stage in your pipeline.

The following example installs the latest version of the Angular CLI by using `npm`. The rest of the pipeline can then use the `ng` tool from other `script` stages.

> **Note**
>
> On Microsoft-hosted Linux agents, preface the command with `sudo`, like `sudo npm install -g`.

YAML

```yaml
- script: npm install -g @angular/cli
```

> **Tip**
>
> These tasks run every time your pipeline runs, so be mindful of the impact that installing tools has on build times. Consider configuring **self-hosted agents** with the version of the tools you need if overhead becomes a serious impact to your build performance.

Manage dependencies

In your build, use Yarn or Azure Artifacts to download packages from the public npm registry. This registry is a type of private npm registry that you specify in the `.npmrc` file.

Use npm

You can use npm in the following ways to download packages for your build:

- Directly run `npm install` in your pipeline, as it's the simplest way to download packages from a registry without authentication. If your build doesn't need development dependencies on the agent to run, you can speed up build times with the `--only=prod` option to `npm install`.
- Use an npm task. This task is useful when you're using an authenticated registry.
- Use an npm Authenticate task. This task is useful when you run `npm install` from inside your task runners - Gulp, Grunt, or Maven.

If you want to specify an npm registry, put the URLs in an `.npmrc` file in your repository. If your feed gets authenticated, create an npm service connection on the **Services** tab in **Project settings** to manage its credentials.

To install npm packages with a script in your pipeline, add the following snippet to `azure-pipelines.yml`.

YAML

```yaml
- script: npm install
```

To use a private registry specified in your `.npmrc` file, add the following snippet to `azure-pipelines.yml`.

YAML

```yaml
- task: Npm@1
  inputs:
    customEndpoint: <Name of npm service connection>
```

To pass registry credentials to npm commands via task runners such as Gulp, add the following task to `azure-pipelines.yml` before you call the task runner.

YAML

```yaml
- task: npmAuthenticate@0
  inputs:
    customEndpoint: <Name of npm service connection>
```

If your builds occasionally fail because of connection issues when you restore packages from the npm registry, you can use Azure Artifacts with upstream sources, and cache the packages. The credentials of the pipeline automatically get used when you connect to Azure Artifacts. These credentials are typically derived from the **Project Collection Build Service** account.

If you're using Microsoft-hosted agents, you get a new machine every time you run a build - which means restoring the dependencies every time, which can take a significant amount of time. To mitigate, you can use Azure Artifacts or a self-hosted agent - then you get the benefit of using the package cache.

Use Yarn

Use a script stage to invoke Yarn to restore dependencies. Yarn gets preinstalled on some Microsoft-hosted agents. You can install and configure it on self-hosted agents like any other tool.

YAML

```yaml
- script: yarn install
```

Run JavaScript compilers

Use compilers such as Babel and the TypeScript `tsc` compiler to convert your source code into versions usable by the Node.js runtime or in web browsers.

If you have a script object set up in your project `package.json` file that runs your compiler, invoke it in your pipeline by using a script task.

YAML

```yaml
- script: npm run compile
```

You can call compilers directly from the pipeline by using the script task. These commands run from the root of the cloned source-code repository.

YAML

```yaml
- script: tsc --target ES6 --strict true --project tsconfigs/production.json
```

Run unit tests

Configure your pipelines to run your JavaScript tests so that they produce results formatted in the JUnit XML format. You can then publish the results using the built-in publish test results task.

If your test framework doesn't support JUnit output, add support through a partner reporting module, such as mocha-junit-reporter. You can either update your test script to use the JUnit reporter, or if the reporter supports command-line options, pass those options into the task definition.

The following table lists the most commonly used test runners and the reporters that can be used to produce XML results:

Test runner	Reporters to produce XML reports
mocha	mocha-junit-reporter cypress-multi-reporters
jasmine	jasmine-reporters
jest	jest-junit jest-junit-reporter
karma	karma-junit-reporter
Ava	tap-xunit

The following example uses the mocha-junit-reporter and invokes `mocha test` directly by using a script. This script produces the JUnit XML output at the default location of `./test-results.xml`.

YAML

```yaml
- script: mocha test --reporter mocha-junit-reporter
```

If you defined a `test` script in your project package.json file, you can invoke it by using `npm test`.

YAML

```yaml
- script: npm test
```

Publish test results

To publish the results, use the Publish Test Results task.

YAML

```yaml
- task: PublishTestResults@2
  condition: succeededOrFailed()
```

```
  inputs:
    testRunner: JUnit
    testResultsFiles: '**/test-results.xml'
```

Publish code coverage results

If your test scripts run a code coverage tool, such as Istanbul, add the Publish Code Coverage Results task. When you do so, you can find coverage metrics in the build summary and download HTML reports for further analysis. The task expects Cobertura or JaCoCo reporting output, so ensure that your code coverage tool runs with the necessary options to generate the right output. For example, `--report cobertura`.

The following example uses nyc, the Istanbul command-line interface, along with mocha-junit-reporter and invokes `npm test` command.

YAML

```yaml
- script: |
    nyc --reporter=cobertura --reporter=html \
    npm test -- --reporter mocha-junit-reporter --reporter-options
mochaFile=./test-results.xml
  displayName: 'Build code coverage report'

- task: PublishCodeCoverageResults@2
  inputs:
    summaryFileLocation:
'$(System.DefaultWorkingDirectory)/**/*coverage.xml'
```

Test browser end-to-end

Run tests in headless browsers as part of your pipeline with tools like Protractor or Karma. Then publish the results for the build to Azure DevOps with the following steps:

1. Install a headless browser testing driver, such as headless Chrome or Firefox, or a browser-mocking tool such as PhantomJS, on the build agent.
2. Configure your test framework to use the headless browser/driver option of your choice according to the tool's documentation.
3. Configure your test framework (usually with a reporter plug-in or configuration) to output JUnit-formatted test results.
4. Set up a script task to run any CLI commands needed to start the headless browser instances.
5. Run the end-to-end tests in the pipeline stages along with your unit tests.

6. Publish the results by using the same Publish Test Results task alongside your unit tests.

Package web apps

Package applications to bundle all your application modules with intermediate outputs and dependencies into static assets ready for deployment. Add a pipeline stage after your compilation and tests to run a tool like webpack or ng build by using the Angular CLI.

The first example calls `webpack`. To have this work, make sure that `webpack` is configured as a development dependency in your package.json project file. This runs `webpack` with the default configuration unless you have a `webpack.config.js` file in the root folder of your project.

```yaml
- script: webpack
```

The next example uses the npm task to call `npm run build` to call the `build` script object defined in the project package.json. Using script objects in your project moves the logic for the build into the source code and out of the pipeline.

```yaml
- script: npm run build
```

Implement JavaScript frameworks

Angular

For Angular apps, you can include Angular-specific commands such as **ng test**, **ng build**, and **ng e2e**. To use Angular CLI commands in your pipeline, install the angular/cli npm package on the build agent.

> **Note**
>
> On Microsoft-hosted Linux agents, preface the command with `sudo`, like `sudo npm install -g`.

YAML

```yaml
- script: |
    npm install -g @angular/cli
    npm install
    ng build --prod
```

For tests in your pipeline that require a browser to run, such as the **ng test** command in the starter app, which runs Karma, use a headless browser instead of a standard browser. In the Angular starter app:

1. Change the `browsers` entry in your *karma.conf.js* project file from `browsers: ['Chrome']` to `browsers: ['ChromeHeadless']`.

2. Change the `singleRun` entry in your *karma.conf.js* project file from a value of `false` to `true`. This change helps make sure that the Karma process stops after it runs.

React and Vue

All the dependencies for your React and Vue apps are captured in your *package.json* file. Your *azure-pipelines.yml* file contains the standard Node.js script:

YAML

```yaml
- script: |
    npm install
  displayName: 'npm install'

- script: |
    npm run build
  displayName: 'npm build'
```

The build files are in a new folder, `dist` (for Vue) or `build` (for React). This snippet builds an artifact - `www` - that is ready for release. It uses the Node Installer, Copy Files, and Publish Build Artifacts tasks.

YAML

```yaml
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: NodeTool@1
```

```yaml
  inputs:
    version: '16.x'
  displayName: 'Install Node.js'

- script: |
    npm install
  displayName: 'npm install'

- script: |
    npm run build
  displayName: 'npm build'

- task: CopyFiles@2
  inputs:
    Contents: 'build/**' # Pull the build directory (React)
    TargetFolder: '$(Build.ArtifactStagingDirectory)'

- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: $(Build.ArtifactStagingDirectory) # dist or build files
    ArtifactName: 'www' # output artifact named www
```

To release, point your release task to the `dist` or `build` artifact and use the Azure Web App Deploy task.

Webpack

You can use a webpack configuration file to specify a compiler, such as Babel or TypeScript, to transpile JSX or TypeScript to plain JavaScript, and to bundle your app.

YAML

```yaml
- script: |
    npm install webpack webpack-cli --save-dev
    npx webpack --config webpack.config.js
```

Build task runners

It's common to use Gulp or Grunt as a task runner to build and test a JavaScript app.

Gulp

Gulp gets preinstalled on Microsoft-hosted agents. Run the `gulp` command in the YAML file:

YAML

```yaml
- script: gulp                        # include any additional options that
are needed
```

If the steps in your gulpfile.js file require authentication with an npm registry:

YAML

```yaml
- task: npmAuthenticate@0
  inputs:
    customEndpoint: <Name of npm service connection>

- script: gulp                        # include any additional options that
are needed
```

Add the Publish Test Results task to publish JUnit or xUnit test results to the server.

YAML

```yaml
- task: PublishTestResults@2
  inputs:
    testResultsFiles: '**/TEST-RESULTS.xml'
    testRunTitle: 'Test results for JavaScript using gulp'
```

Add the Publish Code Coverage Results task to publish code coverage results to the server. You can find coverage metrics in the build summary, and you can download HTML reports for further analysis.

YAML

```yaml
- task: PublishCodeCoverageResults@1
  inputs:
    codeCoverageTool: Cobertura
    summaryFileLocation:
'$(System.DefaultWorkingDirectory)/**/*coverage.xml'
    reportDirectory: '$(System.DefaultWorkingDirectory)/**/coverage'
```

Grunt

Grunt gets preinstalled on Microsoft-hosted agents. To run the grunt command in the YAML file:

YAML

```
- script: grunt                      # include any additional options that
are needed
```

If the steps in your `Gruntfile.js` file require authentication with an npm registry:

YAML

```yaml
- task: npmAuthenticate@0
  inputs:
    customEndpoint: <Name of npm service connection>

- script: grunt                      # include any additional options that
are needed
```

Package and deliver your code

After you've built and tested your app, you can upload the build output to Azure Pipelines, create and publish an npm or Maven package, or package the build output into a .zip file for deployment to a web application.

Publish files to Azure Pipelines

To upload the entire working directory of files, use the Publish Build Artifacts task and add the following to your `azure-pipelines.yml` file.

YAML

```yaml
- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(System.DefaultWorkingDirectory)'
```

To upload a subset of files, first copy the necessary files from the working directory to a staging directory with the Copy Files task, and then use the Publish Build Artifacts task.

YAML

```yaml
- task: CopyFiles@2
  inputs:
    SourceFolder: '$(System.DefaultWorkingDirectory)'
    Contents: |
      **\*.js
      package.json
    TargetFolder: '$(Build.ArtifactStagingDirectory)'
```

```yaml
- task: PublishBuildArtifacts@1
```

Publish a module to an npm registry

If your project's output is an `npm` module for use by other projects and not a web application, use the npm task to publish the module to a local registry or to the public npm registry. Provide a unique name/version combination each time you publish.

Examples

The first example assumes that you manage version information (such as through an npm version) through changes to your `package.json` file in version control. The following example uses the script task to publish to the public registry.

YAML

```yaml
- script: npm publish
```

The next example publishes to a custom registry defined in your repo's `.npmrc` file. Set up an npm service connection to inject authentication credentials into the connection as the build runs.

YAML

```yaml
- task: Npm@1
  inputs:
     command: publish
     publishRegistry: useExternalRegistry
     publishEndpoint: https://my.npmregistry.com
```

The final example publishes the module to an Azure DevOps Services package management feed.

YAML

```yaml
- task: Npm@1
  inputs:
     command: publish
     publishRegistry: useFeed
     publishFeed: https://my.npmregistry.com
```

For more information about versioning and publishing npm packages, see Publish npm packages and How can I version my npm packages as part of the build process?.

Deploy a web app

To create a .zip file archive that is ready for publishing to a web app, use the Archive Files task:

YAML

```yaml
- task: ArchiveFiles@2
  inputs:
    rootFolderOrFile: '$(System.DefaultWorkingDirectory)'
    includeRootFolder: false
```

To publish this archive to a web app, see Azure web app deployment.

Build and push image to container registry

Once your source code builds successfully and your unit tests are in place and successful, you can also build an image and push it to a container registry.

Troubleshoot

If you can build your project on your development machine but are having trouble building it on Azure Pipelines, explore the following potential causes and corrective actions:

- Check that the versions of **Node.js** and the task runner on your development machine match those on the agent. You can include command-line scripts such as `node --version` in your pipeline to check what is installed on the agent. Either use the **Node Tool Installer** (as explained in this guidance) to deploy the same version on the agent, or run `npm install` commands to update the tools to wanted versions.

- If your builds fail intermittently while you restore packages, either the npm registry has issues or there are networking problems between the Azure data center and the registry. We can't control these factors. Explore whether using Azure Artifacts with an npm registry as an upstream source improves the reliability of your builds.

- If you're using nvm to manage different versions of Node.js, consider switching to the **Node Tool Installer** task instead. (`nvm` is installed for historical reasons on

the macOS image.) `nvm` manages multiple Node.js versions by adding shell aliases and altering `PATH`, which interacts poorly with the way Azure Pipelines runs each task in a new process.

The **Node Tool Installer** task handles this model correctly. However, if your work requires the use of `nvm`, you can add the following script to the beginning of each pipeline:

YAML

```yaml
steps:
- bash: |
    NODE_VERSION=16  # or whatever your preferred version is
    npm config delete prefix  # avoid a warning
    . ${NVM_DIR}/nvm.sh
    nvm use ${NODE_VERSION}
    nvm alias default ${NODE_VERSION}
    VERSION_PATH="$(nvm_version_path ${NODE_VERSION})"
    echo "##vso[task.prependPath]$VERSION_PATH"
```

Then, `node` and other command-line tools work for the rest of the pipeline job. In each step where you use the `nvm` command, start the script with the following code:

YAML

```yaml
- bash: |
    . ${NVM_DIR}/nvm.sh
    nvm <command>
```

FAQ

Q: Where can I learn more about Azure Artifacts and the Package Management service?

A: Package Management in Azure Artifacts

Q: Where can I learn more about tasks?

A: Build, release, and test tasks

Q: How do I fix a pipeline failure with the message 'FATAL ERROR: CALL_AND_RETRY_LAST Allocation failed - JavaScript heap out of memory'?

A: This failure type happens when the Node.js package exceeds the memory usage limit. To resolve the issue, add a variable like `NODE_OPTIONS` and assign it a value of *--max_old_space_size=16384*.

Q: How can I version my npm packages as part of the build process?

A: One option is to use a combination of version control and npm version. At the end of a pipeline run, you can update your repo with the new version. In this YAML, there's a GitHub repo and the package gets deployed to npmjs. Your build fails if there's a mismatch between your package version on npmjs and your `package.json` file.

YAML

```yaml
variables:
    MAP_NPMTOKEN: $(NPMTOKEN) # Mapping secret var

trigger:
- none

pool:
  vmImage: 'ubuntu-latest'

steps: # Checking out connected repo
- checkout: self
  persistCredentials: true
  clean: true

- task: npmAuthenticate@0
  inputs:
    workingFile: .npmrc
    customEndpoint: 'my-npm-connection'

- task: NodeTool@1
  inputs:
    version: '16.x'
  displayName: 'Install Node.js'

- script: |
    npm install
  displayName: 'npm install'

- script: |
    npm pack
```

```yaml
  displayName: 'Package for release'

- bash: | # Grab the package version
    v=`node -p "const p = require('./package.json'); p.version;"`
    echo "##vso[task.setvariable variable=packageVersion]$v"

- task: CopyFiles@2
  inputs:
      contents: '*.tgz'
      targetFolder: $(Build.ArtifactStagingDirectory)/npm
  displayName: 'Copy archives to artifacts staging directory'

- task: CopyFiles@2
  inputs:
    sourceFolder: '$(Build.SourcesDirectory)'
    contents: 'package.json'
    targetFolder: $(Build.ArtifactStagingDirectory)/npm
  displayName: 'Copy package.json'

- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory)/npm'
    artifactName: npm
  displayName: 'Publish npm artifact'

- script: |  # Config can be set in .npmrc
    npm config set //registry.npmjs.org/:_authToken=$(MAP_NPMTOKEN)
    npm config set scope "@myscope"
    # npm config list
    # npm --version
    npm version patch --force
    npm publish --access public

- task: CmdLine@2 # Push changes to GitHub (substitute your repo)
  inputs:
    script: |
      git config --global user.email "username@contoso.com"
      git config --global user.name "Azure Pipeline"
      git add package.json
      git commit -a -m "Test Commit from Azure DevOps"
      git push -u origin HEAD:main
```

Tutorial: Automate Node.js deployments with Azure Pipelines

Article • 07/13/2023

In this tutorial, you'll learn how Azure and Azure DevOps support Node.js applications by building an Azure DevOps pipeline that deploys a Node.js app to Azure App Service. With Azure Pipelines, you'll be able to deploy your Node.js app automatically and reduce the risk of errors and downtime with continuous integration and continuous delivery.

In this tutorial, you learn how to:

- ✓ Fork a sample application in GitHub
- ✓ Create Azure App Service resources
- ✓ Select the Azure Pipelines Node.js Express Web App to Linux on Azure Azure DevOps template
- ✓ Deploy a basic Node.js application to Azure App Service

Prerequisites

- An Azure subscription. You need your own Azure subscription to complete this tutorial.
- A GitHub account where you can create a repository. Create one for free.
- An Azure DevOps organization with access to parallel jobs. If your organization doesn't have access to parallel jobs, you can request parallel jobs for free for public or private projects using this form. Your request takes 2-3 business days.
- Familiarity with Azure App Service and Azure DevOps.

Fork the sample application

To fork the sample application, you'll need to sign in to GitHub, go to the sample repository, and create a fork.

1. Go to GitHub, and sign in.

2. Navigate to the Node.js sample project.

3. Select **Fork**.

4. Select **Create fork** to create a fork in your repository.

5. Verify that you now have a version of the repository in your GitHub account by checking the URL path in a browser. Your URL should be `https://github.com/{GitHub User}/nodejs-docs-hello-world`.

Create Azure App Service resources

Use the Azure CLI to add the resources needed to deploy and run an App Service instance. You'll access Azure CLI from Azure Cloud Shell.

1. Go to the [Azure portal][https://portal.azure.com?azure-portal=true] and sign in.

2. From the menu, select **Cloud Shell**. When prompted, select the **Bash** experience.

> ⓘ **Note**
>
> Cloud Shell requires an Azure storage resource to persist any files that you create in Cloud Shell. When you first open Cloud Shell, you're prompted to create a resource group, storage account, and Azure Files share. This setup is automatically used for all future Cloud Shell sessions.

3. From Cloud Shell, run the following az account list-locations command to list the regions that are available from your Azure subscription.

Azure CLI

```
az account list-locations \
  --query "[].{Name: name, DisplayName: displayName}" \
  --output table
```

4. From the `Name` column in the output, choose a region that's close to you. For example, choose `eastasia` or `westus2`.

5. Run az configure to set your default region. Replace `<REGION>` with the name of the region you chose. This example sets `westus2` as the default region:

Azure CLI

```
az configure --defaults location=westus2
```

6. Generate a random number to make your resource names unique. The advantage of having a unique value is that your App Service instance won't have a name conflict with other learners completing this tutorial.

Bash

```
resourceSuffix=$RANDOM
```

7. Create globally unique names for your App Service Web App, resource group, and App Service plan.

Bash

```
webName="helloworld-nodejs-${resourceSuffix}"
rgName='hello-world-nodejs-rg'
planName='helloworld-nodejs-plan'
```

8. Run the following az group create command to create a resource group using the name defined earlier.

Azure CLI

```
az group create --name $rgName
```

9. Run the following az appservice plan create command to create an App Service plan using the name defined earlier.

Azure CLI

```
az appservice plan create \
  --name $planName \
  --resource-group $rgName \
  --sku B1 \
  --is-linux
```

The `--sku` argument specifies the B1 plan. This plan runs on the Basic tier. The `--is-linux` argument specifies to use Linux workers.

> **Important**
>
> If the B1 SKU isn't available in your Azure subscription, **choose a different plan**, such as S1 (Standard).

10. Run the following az webapp create command to create the App Service instance.

Azure CLI

```
az webapp create \
  --name $webName \
  --resource-group $rgName \
  --plan $planName \
  --runtime "node|16-lts"
```

11. Run the following az webapp list command to list the host name and state of the App Service instance.

Azure CLI

```
az webapp list \
  --resource-group $rgName \
  --query "[].{hostName: defaultHostName, state: state}" \
  --output table
```

Create the pipeline from a template

This process generates a pipelines configuration file named *azure-pipelines.yml*, which lives in the root directory of your Git repository.

1. Sign into your account at dev.azure.com.

2. Select + **New project**.

   The **Create new project** dialog box opens.

3. Create a new project with the following options.

Field	Description
Project name	Enter a name such as *nodejs-hello-world*.
Visibility	Choose whether to make your project public or private.
Advanced > **Version control**	Select **Git**.

4. Go to your ***nodejs-hello-world*** project.

5. Go to **Pipelines**, and then select **Create pipeline**.

6. Complete the steps of the wizard by first selecting **GitHub** as the location of your source code.

7. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.

8. When you see the list of repositories, select your repository.

9. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.

10. On the **Select** tab, select your ***nodejs-docs-hello-world*** repository.

11. On the **Configure** tab, select **Node.js Express Web App to Linux on Azure**.

    When prompted:
    a. Select the Azure subscription from which you created the resources earlier.
    b. Select **Continue**.
    c. Select the app name you created earlier, for example **helloworld-nodejs-16353**.
    d. Select **Validate and configure**.

12. On the **Review** tab, review the code for your pipeline configuration.

13. Select **Save** to save your changes.

Deploy the Node.js app to Azure App Service

Once you run your pipeline, the code deploys to Azure App Service. The pipeline is configured to run whenever a change is committed to the `main` branch.

1. Make a small, insignificant change to your pipeline YAML such as adding a space. Select **Save and run** again to commit git changes and trigger the pipeline to run.

2. In Azure DevOps, go to **Pipelines** > **Pipelines** and select your pipeline.

3. Watch your pipeline run and trace its build.

4. After the build succeeds, select the deploy task, and select the URL to view the deployed website.

5. Go to the deployed website URL and verify that you see the site running on App Service.

Clean up resources

If you're not going to continue to use this application, delete the resource group in Azure portal and the project in Azure DevOps with the following steps:

To clean up your resource group:

1. Go to the Azure portal and sign in.

2. From the menu bar, select Cloud Shell. When prompted, select the **Bash** experience.

3. Run the following az group delete command to delete the resource group that you used, `hello-world-nodejs-rg`.

```
Azure CLI

az group delete --name hello-world-nodejs-rg
```

To delete your Azure DevOps project, including the build pipeline:

1. In Azure DevOps, navigate to your project.
2. Select **Project settings**.
3. In the **Project details**, select **Delete**.

Related content

- Customize JavaScript
- Deploy to App Service using Azure Pipelines

Build Python apps

Article • 12/20/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

You can use Azure Pipelines to build, test, and deploy Python apps and scripts as part of your CI/CD system. This article focuses on creating a basic pipeline. This quickstart walks through how to create a simple Flask app with three pages that use a common base template and deploy it with Azure DevOps.

You don't have to set up anything for Azure Pipelines to build Python projects. Python is preinstalled on Microsoft-hosted build agents for Linux, macOS, or Windows. To see which Python versions are preinstalled, see Use a Microsoft-hosted agent.

To learn about configuring Python in pipelines, see Customize Python.

If you want a more complex example, see Use CI/CD to deploy a Python web app to Azure App Service on Linux.

Prerequisites

You must have the following items in Azure DevOps:

- A GitHub account where you can create a repository. Create one for free.
- An Azure DevOps organization and project. Create one for free.
- An ability to run pipelines on Microsoft-hosted agents. You can either purchase a parallel job or you can request a free tier.

1 - Fork the sample code

For the following sample Python Flask tutorial:

```
https://github.com/Microsoft/python-sample-vscode-flask-tutorial
```

2 - Create your pipeline

1. Sign in to Azure Pipelines. Your browser will go to `https://dev.azure.com/my-organization-name` and display your Azure DevOps dashboard.

2. Go to your project and select **Pipelines** > **Create a new pipeline**.
3. Select **GitHub** as the location of your source code.
4. If you're redirected to GitHub to sign in, enter your GitHub credentials.
5. When the list of repositories appears, select your Python sample repository.
6. Azure Pipelines analyzes the code in your repository and recommends the `Python package` template for your pipeline. Select that template.
7. Azure Pipelines generates a YAML file for your pipeline. Select **Save and run** > **Commit directly to the main branch**, and then choose **Save and run** again.
8. A new run starts. Wait for the run to finish.

When you're done, you have a YAML file *azure-pipelines.yml* in your repository that's ready for you to customize.

Customize your pipeline

1. Edit the `azure-pipelines.yml` file in your repository and update the Python version references.

YAML

```yaml
trigger:
- main

pool:
  vmImage: ubuntu-latest
strategy:
  matrix:
    Python38:
      python.version: '3.8'
    Python39:
      python.version: '3.9'
    Python310:
      python.version: '3.10'

steps:
- task: UsePythonVersion@0
  inputs:
    versionSpec: '$(python.version)'
  displayName: 'Use Python $(python.version)'

- script: |
    python -m pip install --upgrade pip
    pip install -r requirements.txt
```

```
  displayName: 'Install dependencies'

- script: |
    pip install pytest pytest-azurepipelines
    pytest
  displayName: 'pytest'
```

3 - Run your pipeline

Save and run your pipeline. After your pipeline runs, verify that the jobs ran successfully.

Next steps

Congratulations, you've successfully completed this quickstart! To run Python scripts or run specific versions of Python, see Configure Python.

Configure Python

Customize Python for Azure Pipelines

Article • 06/30/2023

You can use Azure Pipelines to build your Python apps without having to set up any infrastructure of your own. Tools that you commonly use to build, test, and run Python apps - like pip - get pre-installed on Microsoft-hosted agents in Azure Pipelines.

To create your first pipeline with Python, see the Python quickstart.

Use a specific Python version

To use a specific version of Python in your pipeline, add the Use Python Version task to *azure-pipelines.yml*. This snippet sets the pipeline to use Python 3.11:

YAML

```yaml
steps:
- task: UsePythonVersion@0
  inputs:
    versionSpec: '3.11'
```

Use multiple Python versions

To run a pipeline with multiple Python versions, for example to test a package against those versions, define a `job` with a `matrix` of Python versions. Then set the `UsePythonVersion` task to reference the `matrix` variable.

YAML

```yaml
jobs:
- job: 'Test'
  pool:
    vmImage: 'ubuntu-latest' # other options: 'macOS-latest', 'windows-
latest'
  strategy:
    matrix:
      Python38:
        python.version: '3.8'
      Python39:
        python.version: '3.9'
      Python310:
        python.version: '3.10'

  steps:
  - task: UsePythonVersion@0
```

```
    inputs:
      versionSpec: '$(python.version)'
```

You can add tasks to run using each Python version in the matrix.

Run Python scripts

To run Python scripts in your repository, use a `script` element and specify a filename. For example:

YAML

```yaml
- script: python src/example.py
```

You can also run inline Python scripts with the Python Script task:

YAML

```yaml
- task: PythonScript@0
  inputs:
    scriptSource: 'inline'
    script: |
      print('Hello world 1')
      print('Hello world 2')
```

To parameterize script execution, use the `PythonScript` task with `arguments` values to pass arguments into the executing process. You can use `sys.argv` or the more sophisticated `argparse` library to parse the arguments.

YAML

```yaml
- task: PythonScript@0
  inputs:
    scriptSource: inline
    script: |
      import sys
      print ('Executing script file is:', str(sys.argv[0]))
      print ('The arguments are:', str(sys.argv))
      import argparse
      parser = argparse.ArgumentParser()
      parser.add_argument("--world", help="Provide the name of the world to
greet.")
      args = parser.parse_args()
      print ('Hello ', args.world)
    arguments: --world Venus
```

Install dependencies

You can use scripts to install specific PyPI packages with `pip`. For example, this YAML installs or upgrades `pip` and the `setuptools` and `wheel` packages.

YAML

```yaml
- script: python -m pip install --upgrade pip setuptools wheel
  displayName: 'Install tools'
```

Install requirements

After you update `pip` and friends, a typical next step is to install dependencies from *requirements.txt*:

YAML

```yaml
- script: pip install -r requirements.txt
  displayName: 'Install requirements'
```

Run tests

Use scripts to install and run various tests in your pipeline.

Run lint tests with flake8

To install or upgrade `flake8` and use it to run lint tests, use this YAML:

YAML

```yaml
- script: |
    python -m pip install flake8
    flake8 .
  displayName: 'Run lint tests'
```

Test with pytest and collect coverage metrics with pytest-cov

Use this YAML to install `pytest` and `pytest-cov`, run tests, output test results in JUnit format, and output code coverage results in Cobertura XML format:

YAML

```yaml
- script: |
    pip install pytest pytest-azurepipelines
    pip install pytest-cov
    pytest --doctest-modules --junitxml=junit/test-results.xml --cov=. --
cov-report=xml
  displayName: 'pytest'
```

Run tests with Tox

Azure Pipelines can run parallel Tox test jobs to split up the work. On a development computer, you have to run your test environments in series. This sample uses `tox -e py` to run whichever version of Python is active for the current job.

YAML

```yaml
- job:

  pool:
    vmImage: 'ubuntu-latest'
  strategy:
    matrix:
      Python38:
        python.version: '3.8'
      Python39:
        python.version: '3.9'
      Python310:
        python.version: '3.10'

  steps:
  - task: UsePythonVersion@0
    displayName: 'Use Python $(python.version)'
    inputs:
      versionSpec: '$(python.version)'

  - script: pip install tox
    displayName: 'Install Tox'

  - script: tox -e py
    displayName: 'Run Tox'
```

Publish test results

Add the Publish Test Results task to publish JUnit or xUnit test results to the server:

YAML

```yaml
- task: PublishTestResults@2
  condition: succeededOrFailed()
  inputs:
    testResultsFiles: '**/test-*.xml'
    testRunTitle: 'Publish test results for Python $(python.version)'
```

Publish code coverage results

Add the Publish Code Coverage Results task to publish code coverage results to the server. You can see coverage metrics in the build summary and download HTML reports for further analysis.

YAML

```yaml
- task: PublishCodeCoverageResults@1
  inputs:
    codeCoverageTool: Cobertura
    summaryFileLocation: '$(System.DefaultWorkingDirectory)/**/coverage.xml'
```

Package and deliver code

To authenticate with `twine`, use the Twine Authenticate task to store authentication credentials in the `PYPIRC_PATH` environment variable.

YAML

```yaml
- task: TwineAuthenticate@0
  inputs:
    artifactFeed: '<Azure Artifacts feed name>'
    pythonUploadServiceConnection: '<twine service connection from external
organization>'
```

Then, add a custom script that uses `twine` to publish your packages.

YAML

```yaml
- script: |
   twine upload -r "<feed or service connection name>" --config-file
$(PYPIRC_PATH) <package path/files>
```

You can also use Azure Pipelines to build an image for your Python app and push it to a container registry.

Related extensions

- Azure DevOps plugin for PyCharm (IntelliJ) (Microsoft)
- Python in Visual Studio Code (Microsoft)

Use CI/CD to deploy a Python web app to Azure App Service on Linux

Article • 05/26/2023

Azure DevOps Services

Use Azure Pipelines continuous integration and continuous delivery (CI/CD) to deploy a Python web app to Azure App Service on Linux. Your pipeline automatically builds the code and deploys it to the App Service whenever there's a commit to the repository. You can add other functionalities in your pipeline, such as test scripts, security checks, multistages deployment, and so on.

Create a repository for your app code

If you already have a Python web app to use, make sure it's committed to a GitHub repository.

> **Note**
>
> If your app uses Django and a SQLite database, it won't work for this tutorial. For more information, see **considerations for Django** later in this article. If your Django app uses a separate database, you can use it with this tutorial.

If you need an app to work with, you can fork and clone the repository at https://github.com/Microsoft/python-sample-vscode-flask-tutorial. The code is from the tutorial Flask in Visual Studio Code.

To test the example app locally, from the folder containing the code, run the following appropriate commands for your operating system:

Bash

```bash
# Linux
sudo apt-get install python3-venv  # If needed
python3 -m venv .env
source .env/bin/activate
python3 -m pip install --upgrade pip
python3 -m pip install -r ./requirements.txt
export set FLASK_APP=hello_app.webapp
python3 -m flask run
```

PowerShell

```
# Windows
py -3 -m venv .env
.env\scripts\activate
pip install --target="./.python_packages/lib/site-packages" -r
./requirements.txt
$env:FLASK_APP = "hello_app.webapp"
python -m flask run
```

Open a browser and go to *http://localhost:5000* to view the app. Verify that you see the title `Visual Studio Flask Tutorial`.

When you're finished, close the browser and stop the Flask server with **Ctrl**+**C**.

Provision the target Azure App Service

The quickest way to create an App Service instance is to use the Azure command-line interface (CLI) through the interactive Azure Cloud Shell. In the following steps, you use az webapp up to both provision the App Service and do the first deployment of your app.

1. Sign in to the Azure portal at https://portal.azure.com.

2. Open the Azure CLI by selecting the Cloud Shell button on the portal's toolbar:

3. The Cloud Shell appears along the bottom of the browser. Select **Bash** from the dropdown:

4. In the Cloud Shell, clone your repository using `git clone`. For the example app, use:

```bash
git clone https://github.com/<your-alias>/python-sample-vscode-flask-tutorial
```

> **Note**
>
> The Cloud Shell is backed by an Azure Storage account in a resource group called *cloud-shell-storage-<your-region>*. That storage account contains an image of the Cloud Shell's file system, which stores the cloned repository. There's a small cost for this storage. You can delete the storage account at the end of this article, along with other resources you create.

Replace `<your-alias>` with the name of the GitHub account you used to fork the repository.

> **Tip**
>
> To paste into the Cloud Shell, use **Ctrl**+**Shift**+**V**, or right-click and select **Paste** from the context menu.

5. In the Cloud Shell, change directories into the repository folder that has your Python app, so the `az webapp up` command will recognize the app as Python.

```bash
cd python-sample-vscode-flask-tutorial
```

6. In the Cloud Shell, use `az webapp up` to create an App Service and initially deploy your app.

```azure cli
az webapp up -n <your-appservice>
```

Change `<your-appservice>` to a name for your app service that's unique across Azure. Typically, you use a personal or company name along with an app identifier,

such as `<your-name>-flaskpipelines`. The app URL becomes *<your-appservice>.azurewebsites.net*.

When the command completes, it shows JSON output in the Cloud Shell.

> **Tip**
>
> If you encounter a "Permission denied" error with a *.zip* file, you may have tried to run the command from a folder that doesn't contain a Python app. The `az webapp up` command then tries to create a Windows app service plan, and fails.

7. If your app uses a custom startup command, set the az webapp config property. For example, the *python-sample-vscode-flask-tutorial* app contains a file named *startup.txt* that contains its specific startup command, so you set the `az webapp config` property to `startup.txt`.

   a. From the first line of output from the previous `az webapp up` command, copy the name of your resource group, which is similar to **<your-name>_rg_Linux_<your-region>**.

   b. Enter the following command, using your resource group name, your app service name (`<your-appservice>`), and your startup file or command (`startup.txt`).

   Azure CLI

   ```
   az webapp config set -g <your-resource-group> -n <your-appservice> --startup-file <your-startup-file-or-command>
   ```

   When the command completes, it shows JSON output in the Cloud Shell.

8. To see the running app, open a browser and go to *http://<your-appservice>.azurewebsites.net*. If you see a generic page, wait a few seconds for the App Service to start, and refresh the page. Verify that you see the title `Visual Studio Flask Tutorial`.

> **Note**
>
> For a detailed description of the specific tasks performed by the `az webapp up` command, see **Provision an App Service with single commands** at the end of this article.

Create an Azure DevOps project and connect to Azure

To deploy to Azure App Service from Azure Pipelines, you need to establish a *service connection* between the two services.

1. In a browser, go to dev.azure.com. If you don't yet have an account on Azure DevOps, select **Start free** and get a free account. If you have an account already, select **Sign in to Azure DevOps**.

   > **Important**
   >
   > To simplify the service connection, use the same email address for Azure DevOps as you use for Azure.

2. Once you sign in, the browser displays your Azure DevOps dashboard, at the URL *https://dev.azure.com/<your-organization-name>*. If more than one organization is listed, select the one you want to use for this tutorial. By default, Azure DevOps creates a new organization using the email alias you used to sign in.

   A project is a grouping for boards, repositories, pipelines, and other aspects of Azure DevOps. If your organization doesn't have any projects, enter the project name *Flask Pipelines* under **Create a project to get started**, and then select **Create project**.

If your organization already has projects, select **New project** on the organization page. In the **Create new project** dialog box, enter the project name *Flask Pipelines*, and select **Create**.

3. From the new project page, select **Project settings** from the left navigation.

4. On the **Project Settings** page, select **Pipelines** > **Service connections**, then select **New service connection**, and then select **Azure Resource Manager** from the dropdown.

5. In the **Add an Azure Resource Manager service connection** dialog box:
   a. Give the connection a name. Make note of the name to use later in the pipeline.
   b. For **Scope level**, select **Subscription**.
   c. Select the subscription for your App Service from the **Subscription** drop-down list.
   d. Under **Resource Group**, select your resource group from the dropdown.
   e. Make sure the option **Allow all pipelines to use this connection** is selected, and then select **OK**.

The new connection appears in the **Service connections** list, and is ready for Azure Pipelines to use from the project.

> **Note**
>
> If you want to use an Azure subscription from a different email account, follow the instructions on **Create an Azure Resource Manager service connection with an existing service principal**.

Create a Python-specific pipeline to deploy to App Service

1. From your project page left navigation, select **Pipelines**.

2. Select **Create Pipeline**:

3. On the **Where is your code** screen, select **GitHub**. You may be prompted to sign into GitHub.

4. On the **Select a repository** screen, select the repository that contains your app, such as your fork of the example app.

5. You may be prompted to enter your GitHub password again as a confirmation, and then GitHub prompts you to install the **Azure Pipelines** extension:

On this screen, scroll down to the **Repository access** section, choose whether to install the extension on all repositories or only selected ones, and then select **Approve and install**:

6. On the **Configure your pipeline** screen, select **Python to Linux Web App on Azure**.

   Your new pipeline appears. When prompted, select the Azure subscription in which you created your Web App.

   - Select the Web App
   - Select Validate and configure

   Azure Pipelines creates an **azure-pipelines.yml** file that defines your CI/CD pipeline as a series of *stages*, *Jobs*, and *steps*, where each step contains the details for different *tasks* and *scripts*. Take a look at the pipeline to see what it does. Make sure all the default inputs are appropriate for your code.

YAML pipeline explained

The YAML file contains the following key elements:

- The `trigger` at the top indicates the commits that trigger the pipeline, such as commits to the `main` branch.
- The `variables` that parameterize the YAML template

> **Tip**
>
> To avoid hard-coding specific variable values in your YAML file, you can define variables in the pipeline's web interface instead. For more information, see **Variables - Secrets**.

- The `stages`
  - Build `stage`, which builds your project, and a Deploy stage, which deploys it to Azure as a Linux web app.
  - Deploy `stage` that also creates an Environment with default name same as the Web App. You can choose to modify the environment name.
- Each stage has a `pool` element that specifies one or more virtual machines (VMs) in which the pipeline runs the `steps`. By default, the `pool` element contains only a single entry for an Ubuntu VM. You can use a pool to run tests in multiple environments as part of the build, such as using different Python versions for creating a package.
- The `steps` element can contain children like `task`, which runs a specific task as defined in the Azure Pipelines task reference, and `script`, which runs an arbitrary set of commands.
- The first task under Build stage is UsePythonVersion, which specifies the version of Python to use on the build agent. The `@<n>` suffix indicates the version of the task. The `@0` indicates preview version. Then we have script-based task that creates a virtual environment and installs dependencies from file (requirements.txt).

YAML

```yaml
steps:
- task: UsePythonVersion@0
   inputs:
     versionSpec: '$(pythonVersion)'
     displayName: 'Use Python $(pythonVersion)'
- script: |
     python -m venv antenv
     source antenv/bin/activate
     python -m pip install --upgrade pip
     pip install setup
```

```
          pip install --target="./.python_packages/lib/site-packages" -r
./requirements.txt
        workingDirectory: $(projectRoot)
        displayName: "Install requirements"
```

- Next step creates the *.zip* file that the steps under deploy stage of the pipeline deploys. To create the *.zip* file, add an ArchiveFiles task to the end of the YAML file:

```yaml
- task: ArchiveFiles@2
   inputs:
     rootFolderOrFile: '$(Build.SourcesDirectory)'
     includeRootFolder: false
     archiveType: 'zip'
     archiveFile:
'$(Build.ArtifactStagingDirectory)/Application$(Build.BuildId).zip'
     replaceExistingArchive: true
     verbose: # (no value); this input is optional
- publish:
$(Build.ArtifactStagingDirectory)/Application$(Build.BuildId).zip
   displayName: 'Upload package'
   artifact: drop
```

You use `$()` in a parameter value to reference variables. The built-in `Build.SourcesDirectory` variable contains the location on the build agent where the pipeline cloned the app code. The `archiveFile` parameter indicates where to place the *.zip* file. In this case, the `archiveFile` parameter uses the built-in variable `Build.ArtifactsStagingDirectory`.

> **Important**
>
> When deploying to Azure App Service, be sure to use `includeRootFolder: false`. Otherwise, the contents of the *.zip* file are put in a folder named *s*, for "sources," which is replicated on the App Service. The App Service on Linux container then can't find the app code.

- Then we have the task to upload the artifacts.

- In the Deploy stage, we use the `deployment` keyword to define a deployment job targeting an environment. By using the template, an environment with same name as the Web app is automatically created if it doesn't already exist. Instead, you can pre-create the environment and provide the `environmentName`.

- Within the deployment job, first task is UsePythonVersion, which specifies the version of Python to use on the build agent.
- We then use the AzureWebApp task to deploy the *.zip* file to the App Service you identified by the `azureServiceConnectionId` and `webAppName` variables at the beginning of the pipeline file. If you need to use a different service connection, select **Settings** for the `AzureWebApp@1` task and update the **Azure subscription** value. Paste the following code at the end of the file:

YAML

```yaml
jobs:
- deployment: DeploymentJob
pool:
  vmImage: $(vmImageName)
environment: $(environmentName)
strategy:
  runOnce:
    deploy:
      steps:

      - task: UsePythonVersion@0
        inputs:
          versionSpec: '$(pythonVersion)'
        displayName: 'Use Python version'

      - task: AzureWebApp@1
        displayName: 'Deploy Azure Web App : {{ webAppName }}'
        inputs:
          azureSubscription: $(azureServiceConnectionId)
          appName: $(webAppName)
          package: $(Pipeline.Workspace)/drop/$(Build.BuildId).zip
          deploymentMethod: zipDeploy

          # The following parameter is specific to the Flask example
code. You may
          # or may not need a startup command for your app.

          startUpCommand: 'gunicorn --bind=0.0.0.0 --workers=4
startup:app'
```

The `StartupCommand` parameter shown is specific to the *python-vscode-flask-tutorial* example code, which defines the app in the *startup.py* file. By default, Azure App Service looks for the Flask app object in a file named *app.py* or *application.py*. If your code doesn't follow this pattern, customize the startup command. Django apps may not need customization at all. For more information, see How to configure Python on Azure App Service - Customize startup command.

Also, because the *python-vscode-flask-tutorial* repository contains the same startup command in a file named *startup.txt*, you could specify that file in the `StartupCommand` parameter rather than the command, by using `StartupCommand: 'startup.txt'`.

Run the pipeline

You're now ready to try it out!

1. Select **Save** at upper right in the editor, and in the pop-up window, add a commit message and select **Save**.

2. Select **Run** on the pipeline editor, and select **Run** again in the **Run pipeline** dialog box. Azure Pipelines queues another pipeline run, acquires an available build agent, and has that build agent run the pipeline.

   The pipeline takes a few minutes to complete, especially the deployment steps. You should see green checkmarks next to each of the steps.

   If there's an error, you can quickly return to the YAML editor by selecting the vertical dots at upper right and selecting **Edit pipeline**:

3. From the build page, select the **Azure Web App task** to display its output. To visit the deployed site, hold down the **Ctrl** key and select the URL after **App Service Application URL**.

   If you're using the Flask example, the app should appear as follows:

> **Important**
>
> If your app fails because of a missing dependency, then your *requirements.txt* file was not processed during deployment. This behavior happens if you created the web app directly on the portal rather than using the `az webapp up` command as shown in this article.
>
> The `az webapp up` command specifically sets the build action `SCM_DO_BUILD_DURING_DEPLOYMENT` to `true`. If you provisioned the app service through the portal, however, this action is not automatically set.
>
> The following steps set the action:
>
> 1. Open the **Azure portal**, select your App Service, then select **Configuration**.
> 2. Under the **Application Settings** tab, select **New Application Setting**.
> 3. In the popup that appears, set **Name** to `SCM_DO_BUILD_DURING_DEPLOYMENT`, set **Value** to `true`, and select **OK**.
> 4. Select **Save** at the top of the **Configuration** page.
> 5. Run the pipeline again. Your dependencies should be installed during deployment.

Run a post-deployment script

A post-deployment script can, for example, define environment variables expected by the app code. Add the script as part of the app code and execute it using startup command.

To avoid hard-coding specific variable values in your YAML file, you can instead define variables in the pipeline's web interface and then refer to the variable name in the script. For more information, see Variables - Secrets.

Considerations for Django

As noted earlier in this article, you can use Azure Pipelines to deploy Django apps to Azure App Service on Linux, if you're using a separate database. You can't use a SQLite database, because App Service locks the *db.sqlite3* file, preventing both reads and writes. This behavior doesn't affect an external database.

As described in Configure Python app on App Service - Container startup process, App Service automatically looks for a *wsgi.py* file within your app code, which typically contains the app object. If you want to customize the startup command in any way, use the `StartupCommand` parameter in the `AzureWebApp@1` step of your YAML pipeline file, as described in the previous section.

When using Django, you typically want to migrate the data models using `manage.py migrate` after deploying the app code. You can add `startUpCommand` with post-deployment script for this purpose:

YAML

```yaml
startUpCommand:  python3.7 manage.py migrate
```

Run tests on the build agent

As part of your build process, you may want to run tests on your app code. Tests run on the build agent, so you probably need to first install your dependencies into a virtual environment on the build agent computer. After the tests run, delete the virtual environment before you create the *.zip* file for deployment. The following script elements illustrate this process. Place them before the `ArchiveFiles@2` task in the *azure-pipelines.yml* file. For more information, see Run cross-platform scripts.

YAML

```yaml
# The | symbol is a continuation character, indicating a multi-line script.
# A single-line script can immediately follow "- script:".
- script: |
    python3.7 -m venv .env
    source .env/bin/activate
    pip3.7 install setuptools
    pip3.7 install -r requirements.txt

  # The displayName shows in the pipeline UI when a build runs
  displayName: 'Install dependencies on build agent'

- script: |
    # Put commands to run tests here
  displayName: 'Run tests'
```

```
- script: |
    echo Deleting .env
    deactivate
    rm -rf .env
  displayName: 'Remove .env before zip'
```

You can also use a task like PublishTestResults@2 to make test results appear in the pipeline results screen. For more information, see Build Python apps - Run tests.

Provision an App Service with single commands

The az webapp up command is a convenient method to provision the App Service and initially deploy your app in a single step. If you want more control over the deployment process, you can use single commands to accomplish the same tasks. For example, you might want to use a specific name for the resource group, or create an App Service within an existing App Service Plan.

The following steps do the equivalent of the `az webapp up` command:

1. Create a resource group.

   A resource group is a collection of related Azure resources. Creating a resource group makes it easy to delete all those resources at once when you no longer need them. In the Cloud Shell, run the following command to create a resource group in your Azure subscription. Set a location for the resource group by specifying the value of `<your-region>`. JSON output appears in the Cloud Shell when the command completes successfully.

   Azure CLI

   ```
   az group create -l <your-region> -n <your-resource-group>
   ```

2. Create an App Service Plan.

   An App Service runs inside a VM defined by an App Service Plan. Run the following command to create an App Service Plan, replacing your own values for `<your-resource-group>` and `<your-appservice-plan>`. The `--is-linux` is required for Python deployments. If you want a pricing plan other than the default F1 Free plan, use the `sku` argument. The `--sku B1` specifies the lower-price compute tier for the VM. You can easily delete the plan later by deleting the resource group.

Azure CLI

```
az appservice plan create -g <your-resource-group> -n <your-appservice-plan> --is-linux --sku B1
```

Again, you see JSON output in the Cloud Shell when the command completes successfully.

3. Create an App Service instance in the plan.

   Run the following command to create the App Service instance in the plan, replacing `<your-appservice>` with a name that's unique across Azure. Typically, you use a personal or company name along with an app identifier, such as `<your-name>-flaskpipelines`. The command fails if the name is already in use. When you assign the App Service to the same resource group as the plan, it's easy to clean up all the resources at once.

   > **Note**
   >
   > If you want to deploy your code at the same time you create the app service, you can use the `--deployment-source-url` and `--deployment-source-branch` arguments with the `az webapp create` command. For more information, see **az webapp create**.

   Azure CLI

   ```
   az webapp create -g <your-resource-group> -p <your-appservice-plan> -n <your-appservice> --runtime "Python|3.7"
   ```

   > **Tip**
   >
   > If you see the error message "The plan (name) doesn't exist", and you're sure that the plan name is correct, check that the resource group specified with the `-g` argument is also correct, and the plan you identify is part of that resource group. If you misspell the resource group name, the command doesn't find the plan in that nonexistent resource group, and gives this particular error.

4. If your app requires a custom startup command, use the `az webapp config set` command, as described earlier in Provision the target Azure App Service. For example, to customize the App Service with your resource group, app name, and startup command, run:

Azure CLI

```
az webapp config set -g <your-resource-group> -n <your-appservice> --
startup-file <your-startup-command-or-file>
```

The App Service at this point contains only default app code. You can now use Azure Pipelines to deploy your specific app code.

Clean up resources

To avoid incurring charges on the Azure resources created in this tutorial, delete the resource group that contains the App Service and the App Service Plan. To delete the resource group from the Azure portal, select **Resource groups** in the left navigation. In the resource group list, select the **...** to the right of the resource group you want to delete, select **Delete resource group**, and follow the prompts.

You can also use az group delete in the Cloud Shell to delete resource groups.

To delete the storage account that maintains the file system for Cloud Shell, which incurs a small monthly charge, delete the resource group that begins with **cloud-shell-storage-**.

Next steps

- Build Python apps
- Learn about build agents
- Configure Python app on App Service

Run pipelines with Anaconda environments

Article • 05/30/2023

Azure DevOps Services

Learn how to set up and use Anaconda with Python in your pipeline. Anaconda is a Python distribution for data science and machine learning.

Get started

Follow these instructions to set up a pipeline for a sample Python app with Anaconda environment.

1. Sign in to your Azure DevOps organization and navigate to your project.
2. In your project, navigate to the **Pipelines** page. Then choose the action to create a new pipeline.
3. Walk through the steps of the wizard by first selecting **GitHub** as the location of your source code.
4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.
5. When the list of repositories appears, select your Anaconda sample repository.
6. Azure Pipelines will analyze the code in your repository and detect an existing `azure-pipelines.yml` file.
7. Select **Run**.
8. A new run is started. Wait for the run to finish.

> **Tip**
>
> To make changes to the YAML file as described in this topic, select the pipeline in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file.

Add conda to your system path

On hosted agents, conda is left out of `PATH` by default to keep its Python version from conflicting with other installed versions. The `task.prependpath` agent command will make it available to all subsequent steps.

Windows

YAML

```yaml
- powershell: Write-Host "##vso[task.prependpath]$env:CONDA\Scripts"
  displayName: Add conda to PATH
```

Create an environment

From command-line arguments

The `conda create` command will create an environment with the arguments you pass it.

Windows

YAML

```yaml
- script: conda create --yes --quiet --name myEnvironment
  displayName: Create Anaconda environment
```

> **Note**
>
> To add specific conda channels, you need to add an extra line for conda config: `conda config --add channels conda-forge`

From YAML

You can check in an environment.yml file to your repo that defines the configuration for an Anaconda environment.

YAML

```yaml
- script: conda env create --quiet --file environment.yml
  displayName: Create Anaconda environment
```

> **Note**
>
> If you are using a self-hosted agent and don't remove the environment at the end, you'll get an error on the next build since the environment already exists. To resolve, use the `--force` argument: `conda env create --quiet --force --file environment.yml`.

> **Note**
>
> If you are using self-hosted agents that are sharing storage, and running jobs in parallel using the same Anaconda environments, there may be clashes between those environments. To resolve, use the `--name` argument and a unique identifier as the argument value, like a concatenation with the `$(Build.BuildNumber)` build variable.

Install packages from Anaconda

The following YAML installs the `scipy` package in the conda environment named `myEnvironment`.

Windows

YAML

```yaml
- script: |
    call activate myEnvironment
    conda install --yes --quiet --name myEnvironment scipy
  displayName: Install Anaconda packages
```

Run pipeline steps in an Anaconda environment

> **Note**
>
> Each build step runs in its own process. When you activate an Anaconda environment, it will edit `PATH` and make other changes to its current process. Therefore, an Anaconda environment must be activated separately for each step.

Windows

YAML

```yaml
- script: |
    call activate myEnvironment
    pytest --junitxml=junit/unit-test.xml
  displayName: pytest

- task: PublishTestResults@2
  inputs:
    testResultsFiles: 'junit/*.xml'
  condition: succeededOrFailed()
```

FAQs

Why am I getting a "Permission denied" error?

On Hosted macOS, the agent user does not have ownership of the directory where Miniconda is installed. For a fix, see the "Hosted macOS" tab under Add conda to your system path.

Why does my build stop responding on a `conda create` or `conda install` step?

If you forget to pass `--yes`, conda will stop and wait for user interaction.

Why is my script on Windows stopping after it activates the environment?

On Windows, `activate` is a Batch script. You must use the call command to resume running your script after activating. See examples of using `call` in a pipeline.

How can I run my tests with multiple versions of Python?

See Build Python apps in Azure Pipelines.

Build C++ Windows apps

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

This article explains how to build C++ projects on Windows automatically.

Example

This example shows how to build a C++ project. To start, import (into Azure Repos or TFS) or fork (into GitHub) this repo:

```
https://github.com/MicrosoftDocs/pipelines-cpp
```

- After you have the sample code in your own repository, create a pipeline using the instructions in Create your first pipeline and select the **.NET Desktop** template. This automatically adds the tasks required to build the code in the sample repository.
- Save the pipeline and queue a build to see it in action.

Build multiple configurations

It's often required to build your app in multiple configurations. The following steps extend the example above to build the app on four configurations: [Debug, x86], [Debug, x64], [Release, x86], [Release, x64].

1. Select the **Variables** tab and modify these variables:
   - `BuildConfiguration` = `debug, release`
   - `BuildPlatform` = `x86, x64`
2. Select **Tasks** and click on the **agent job**. From the **Execution plan** section, select **Multi-configuration** to change the options for the job:
   - Specify **Multipliers:** `BuildConfiguration, BuildPlatform`
   - Specify **Maximum number of agents**

3. Select **Parallel** if you have multiple build agents and want to build your configuration/platform pairings in parallel.

Copy output

To copy the results of the build to Azure Pipelines select the Copy Files task. Specify the following arguments:

```
contents: '**\$(BuildConfiguration)\**\?(*.exe|*.dll|*.pdb)'
```

Build Java apps

Article • 05/30/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

You can use a pipeline to automatically build and test your Java projects. After you build and test your app, you can deploy your app to Azure App Service, Azure Functions, or Azure Kubernetes Service. If you're working on an Android project, see Build, test, and deploy Android apps.

Prerequisites

You must have the following items in Azure DevOps:

- A project. If you don't have one, Create a project now.
- A pipeline. If you don't have one, Create a pipeline now.

Create a pipeline

1. Fork the following repo at GitHub:

   ```
   https://github.com/MicrosoftDocs/pipelines-java
   ```

2. Sign in to your Azure DevOps organization and go to your project.

3. Go to **Pipelines**, and then select **New pipeline**.

4. Perform the steps of the wizard by first selecting **GitHub** as the location of your source code. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.

5. Select your repo. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.

6. When you see the **Configure** tab, select **Maven** or **Gradle** or **Ant** depending on how you want to build your code.

7. When you're ready, select **Save and run**.

8. Commit a new *azure-pipelines.yml* file to your repo. Select **Save and run** again.

   If you want to watch your pipeline in action, select the build job.

   You just created and ran a pipeline, because your code appeared to be a good match for the Maven template that we automatically created for you.

   You now have a working YAML pipeline (`azure-pipelines.yml`) in your repo that's ready for you to customize!

9. When you're ready to make changes to your pipeline, select it in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file.

Read further to learn some of the more common ways to customize your pipeline.

Build environment

You can use Azure Pipelines to build Java apps without needing to set up any infrastructure of your own. You can build on Windows, Linux, or macOS images. The Microsoft-hosted agents in Azure Pipelines have modern JDKs and other tools for Java pre-installed. To know which versions of Java are installed, see Microsoft-hosted agents.

Update the following snippet in your `azure-pipelines.yml` file to select the appropriate image.

YAML

```yaml
pool:
  vmImage: 'ubuntu-latest' # other options: 'macOS-latest', 'windows-latest'
```

See Microsoft-hosted agents for a complete list of images.

As an alternative to using Microsoft-hosted agents, you can set up self-hosted agents with Java installed. You can also use self-hosted agents to save more time if you have a large repo or you can run incremental builds.

Build your code

Maven

With your Maven build, the following snippet gets added to your `azure-pipelines.yml` file. You can change values, such as the path to your `pom.xml` file, to match your project

configuration. See the Maven task for more information about these options.

YAML

```yaml
steps:
- task: Maven@4
  inputs:
    mavenPomFile: 'pom.xml'
    mavenOptions: '-Xmx3072m'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.8'
    jdkArchitectureOption: 'x64'
    publishJUnitResults: true
    testResultsFiles: '**/TEST-*.xml'
    goals: 'package'
```

For Spring Boot, you can use the Maven task as well. Make sure that your `mavenPomFile` value reflects the path to your `pom.xml` file. For example, if you're using the Spring Boot sample repo, your path will be `complete/pom.xml`.

Customize the build path

Adjust the `mavenPomFile` value if your `pom.xml` file isn't in the root of the repo. The file path value should be relative to the root of the repo, such as `IdentityService/pom.xml` or `$(system.defaultWorkingDirectory)/IdentityService/pom.xml`.

Customize Maven goals

Set the **goals** value to a space-separated list of goals for Maven to execute, such as `clean package`.

For details about common Java phases and goals, see Apache's Maven documentation.

Gradle

With the Gradle build, the following snippet gets added to your `azure-pipelines.yml` file. For more information about these options, see the Gradle task.

YAML

```yaml
steps:
- task: Gradle@2
  inputs:
    workingDirectory: ''
```

```
    gradleWrapperFile: 'gradlew'
    gradleOptions: '-Xmx3072m'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.8'
    jdkArchitectureOption: 'x64'
    publishJUnitResults: true
    testResultsFiles: '**/TEST-*.xml'
    tasks: 'build'
```

Choose the version of Gradle

The version of Gradle installed on the agent machine will be used unless your repo's `gradle/wrapper/gradle-wrapper.properties` file has a `distributionUrl` property that specifies a different Gradle version to download and use during the build.

Adjust the build path

Adjust the `workingDirectory` value if your `gradlew` file isn't in the root of the repo. The directory value should be relative to the root of the repo, such as `IdentityService` or `$(system.defaultWorkingDirectory)/IdentityService`.

Adjust the `gradleWrapperFile` value if your `gradlew` file isn't in the root of the repo. The file path value should be relative to the root of the repo, such as `IdentityService/gradlew` or `$(system.defaultWorkingDirectory)/IdentityService/gradlew`.

Adjust Gradle tasks

Adjust the **tasks** value for the tasks that Gradle should execute, such as `build` or `check`.

For more information about common Java Plugin tasks for Gradle, see Gradle's documentation.

Ant

With Ant build, the following snippet is added to your `azure-pipelines.yml` file. Change values, such as the path to your `build.xml` file to match your project configuration. For more information about these options, see the Ant task.

YAML

```
steps:
- task: Ant@1
```

```
  inputs:
    workingDirectory: ''
    buildFile: 'build.xml'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.8'
    jdkArchitectureOption: 'x64'
    publishJUnitResults: false
    testResultsFiles: '**/TEST-*.xml'
```

Script

To build with a command line or script, add one of the following snippets to your `azure-pipelines.yml` file.

Inline script

The `script:` step runs an inline script using Bash on Linux and macOS and Command Prompt on Windows. For details, see the Bash or Command line task.

YAML

```
steps:
- script: |
    echo Starting the build
    mvn package
  displayName: 'Build with Maven'
```

Script file

This snippet runs a script file that is in your repo. For details, see the Shell Script, Batch script, or PowerShell task.

YAML

```
steps:
- task: ShellScript@2
  inputs:
    scriptPath: 'build.sh'
```

Next steps

After you've built and tested your app, you can upload the build output to Azure Pipelines, create and publish a Maven package, or package the build output into a

.war/jar file to be deployed to a web application.

Learn more about creating a CI/CD pipeline for your deployment target:

- Build and deploy to a Java web app
- Build and deploy Java to Azure Functions
- Build and deploy Java to Azure Kubernetes service

Build & deploy to Java web app

Article • 02/11/2022

Azure DevOps Services

A web app is a lightweight way to host a web application. In this step-by-step guide, learn how to create a pipeline that continuously builds and deploys a Java app. Each commit can automatically build at GitHub and deploy to an Azure App Service. You can use whatever runtime you prefer, Tomcat, or Java SE.

For more information, see Java for Azure App Service.

> **Tip**
>
> If you only want to build a Java app, see **Build Java apps**.

Prerequisites

Make sure you have the following items:

- A GitHub account where you can create a repository. Create one for free.
- An Azure DevOps organization. Create one for free. If your team already has one, then make sure you're an administrator of the Azure DevOps project that you want to use.
- An ability to run pipelines on Microsoft-hosted agents. To use Microsoft-hosted agents, your Azure DevOps organization must have access to Microsoft-hosted parallel jobs. You can either purchase a parallel job or you can request a free grant.
- An Azure account. If you don't have one, you can create one for free.

  > **Tip**
  >
  > If you're new at this, the easiest way to get started is to use the same email address as the owner of both the Azure Pipelines organization and the Azure subscription.

Get the code

Select the runtime you want to use.

Tomcat

If you already have an app in GitHub that you want to deploy, you can create a pipeline for that code.

If you are a new user, fork this repo in GitHub:

```
https://github.com/spring-petclinic/spring-framework-petclinic
```

Create an Azure App Service

Sign in to the Azure Portal, and then select the Cloud Shell button in the upper-right corner.

Create an Azure App Service on Linux.

Tomcat

Azure CLI

```
# Create a resource group
az group create --location eastus2 --name myapp-rg

# Create an app service plan of type Linux
az appservice plan create -g myapp-rg -n myapp-service-plan --is-linux

# Create an App Service from the plan with Tomcat and JRE 8 as the
runtime
az webapp create -g myapp-rg -p myapp-service-plan -n my-app-name --
runtime "TOMCAT|8.5-jre8"
```

Create the pipeline

1. Sign-in to your Azure DevOps organization and go to your project.

2. Go to **Pipelines**, and then select **New pipeline**.

3. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.

4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.

5. When you see the list of repositories, select your repository.

6. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.

7. When the **Configure** tab appears, select **Show more**, and then select **Maven package Java project Web App to Linux on Azure**.

8. You can automatically create an Azure Resource Manager service connection when you create your pipeline. To get started, select your Azure subscription where you created a resource group.

9. Select **Validate and configure**. The new pipeline includes a new Azure Resource Manager service connection.

   As Azure Pipelines creates an azure-pipelines.yml file, which defines your CI/CD pipeline, it:

   - Includes a Build stage, which builds your project, and a Deploy stage, which deploys it to Azure as a Linux web app.
   - As part of the Deploy stage, it also creates an Environment with default name same as the Web App. You can choose to modify the environment name.

10. Make sure that all the default inputs are appropriate for your code.

11. Select **Save and run**, after which you're prompted for a commit message because the *azure-pipelines.yml* file gets added to your repository. After editing the message, select **Save and run** again to see your pipeline in action.

See the pipeline run, and your app deployed

As your pipeline runs, your build and deployment stages go from blue (running) to green (completed). To watch your pipeline in action, you can select stages and jobs.

Tomcat

After the pipeline runs, check out your site!

`https://my-app-name.azurewebsites.net/petclinic`

Also explore deployment history for the app by going to the "environment". From the pipeline summary:

1. Select the **Environments** tab.
2. Select **View environment**.

Clean up resources

Whenever you're done with the resources you created, you can use the following command to delete them:

Azure CLI

```
az group delete --name myapp-rg
```

Enter `y` when you're prompted.

Next steps

- Azure for Java developer documentation
- Create a Java app on Azure App Service

CI/CD for MicroProfile apps using Azure Pipelines

Article • 05/30/2023

This tutorial shows you how to easily set up an Azure Pipelines continuous integration and continuous deployment (CI/CD) release cycle to deploy your MicroProfile Java EE application to an Azure Web App for Containers. The MicroProfile app in this tutorial uses a Payara Micro base image to create a WAR file.

Dockerfile

```
FROM payara/micro:5.182
COPY target/*.war $DEPLOY_DIR/ROOT.war
EXPOSE 8080
```

You start the Azure Pipelines containerization process by building a Docker image and pushing the container image to an Azure Container Registry (ACR). You complete the process by creating an Azure Pipelines release pipeline and deploying the container image to a web app.

Prerequisites

1. In the Azure portal, create an Azure Container Registry.
2. In the Azure portal, create an Azure Web App for Containers. Select **Linux** for the **OS**, and for **Configure container**, select **Quickstart** as the **Image source**.
3. Copy and save the clone URL from the sample GitHub repository at https://github.com/Azure-Samples/microprofile-hello-azure.
4. Register or log into your Azure DevOps organization, and create a new project.
5. Import the sample GitHub repository into Azure Repos:
   a. From your Azure DevOps project page, select **Repos** in the left navigation.
   b. Under **or import a repository**, select **Import**.
   c. Under **Clone URL**, enter the Git clone URL you saved, and select **Import**.

Create a build pipeline

The continuous integration Build pipeline in Azure Pipelines automatically executes all build tasks each time there's a commit in the Java EE source app. In this example, Azure

Pipelines uses Maven to build the Java MicroProfile project.

1. From your Azure DevOps project page, select **Pipelines** > **Builds** in the left navigation.

2. Select **New Pipeline**.

3. Select **Use the classic editor to create a pipeline without YAML**.

4. Make sure your project name and imported GitHub repository appear in the fields, and select **Continue**.

5. Select **Maven** from the list of templates, and then select **Apply**.

6. In the right pane, make sure **Hosted Ubuntu 1604** appears in the **Agent pool** dropdown.

   > ⓘ **Note**
   >
   > This setting lets Azure Pipelines know which build server to use. You can also use your private, customized build server.

7. To configure the pipeline for continuous integration, select the **Triggers** tab on the left pane, and then select the checkbox next to **Enable continuous integration**.

8. At the top of the page, select the dropdown next to **Save & queue**, and select **Save**.

Create a Docker build image

Azure Pipelines uses a Dockerfile with a base image from Payara Micro to create a Docker image.

1. Select the **Tasks** tab, and then select the plus sign + next to **Agent job 1** to add a task.

2. In the right pane, select **Docker** from the list of templates, and then select **Add**.

3. Select **buildAndPush** in the left pane, and in the right pane, enter a description in the **Display name** field.

4. Under **Container Repository**, select **New** next to the **Container Registry** field.

5. Fill out the **Add a Docker Registry service connection** dialog as follows:

Field	Value
Registry type	Select **Azure Container Registry**.
Connection Name	Enter a name for the connection.
Azure subscription	Select your Azure subscription from the dropdown, and if necessary, select **Authorize**.
Azure container registry	Select your Azure Container Registry name from the dropdown.

6. Select **OK**.

> **Note**
>
> If you're using Docker Hub or another registry, select **Docker Hub** or **Others** instead of **Azure Container Registry** next to **Registry type**. Then provide the credentials and connection information for your container registry.

7. Under **Commands**, select **build** from the **Command** dropdown.

8. Select the ellipsis **...** next to the **Dockerfile** field, browse to and select the **Dockerfile** from your GitHub repository, and then select **OK**.

9. Under **Tags**, enter *latest* on a new line.

10. At the top of the page, select the dropdown next to **Save & queue**, and select **Save**.

Push the Docker image to ACR

Azure Pipelines pushes the Docker image to your Azure Container Registry, and uses it to run the MicroProfile API app as a containerized Java web app.

1. Since you're using Docker in Azure Pipelines, create another Docker template by repeating the steps under Create a Docker build image. This time, select **push** in the **Command** dropdown.

2. Select the dropdown next to **Save & queue**, and select **Save & queue**.

3. In the **Run pipeline** popup, make sure **Hosted Ubuntu 1604** is selected under **Agent pool**, and select **Save and run**.

4. After the build finishes, you can select the hyperlink on the **Build** page to verify build success and see other details.

Create a release pipeline

An Azure Pipelines continuous Release pipeline automatically triggers deployment to a target environment like Azure as soon as a build succeeds. You can create release pipelines for environments like dev, test, staging, or production.

1. On your Azure DevOps project page, select **Pipelines** > **Releases** in the left navigation.

2. Select **New Pipeline**.

3. Select **Deploy a Java app to Azure App Service** in the list of templates, and then select **Apply**.

4. In the popup window, change **Stage 1** to a stage name like *Dev*, *Test*, *Staging*, or *Production*, and then close the window.

5. Under **Artifacts** in the left pane, select **Add** to link artifacts from the build pipeline to the release pipeline.

6. In the right pane, select your build pipeline in the dropdown under **Source (build pipeline)**, and then select **Add**.

7. Select the hyperlink in the **Production** stage to **View stage tasks**.

8. In the right pane, fill out the form as follows:

Field	Value
Azure subscription	Select your Azure subscription from the dropdown.
App type	Select **Web App for Containers (Linux)** from the dropdown.
App service name	Select your ACR instance from the dropdown.
Registry or Namespaces	Enter your ACR name in the field. For example, enter *mymicroprofileregistry.azure.io*.

Field	Value
Repository	Enter the repository that contains your Docker image.

9. In the left pane, select **Deploy War to Azure App Service**, and in the right pane, enter *latest* tag in the **Tag** field.

10. In the left pane, select **Run on agent**, and in the right pane, select **Hosted Ubuntu 1604** from the **Agent pool** dropdown.

Set up environment variables

Add and define environment variables to connect to the container registry during deployment.

1. Select the **Variables** tab, and then select **Add** to add the following variables for your container registry URL, username, and password.

Name	Value
registry.url	Enter your container registry URL. For example: *https://mymicroprofileregistry.azure.io*
registry.username	Enter the username for the registry.
registry.password	Enter the password for the registry. For security, select the lock icon to keep the password value hidden.

2. On the **Tasks** tab, select **Deploy War to Azure App Service** in the left pane.

3. In the right pane, expand **Application and Configuration Settings**, and then select the ellipsis **...** next to the **App Settings** field.

4. In the **App settings** popup, select **Add** to define and assign the app setting variables:

Name	Value
DOCKER_REGISTRY_SERVER_URL	*$(registry.url)*
DOCKER_REGISTRY_SERVER_USERNAME	*$(registry.username)*
DOCKER_REGISTRY_SERVER_PASSWORD	*$(registry.password)*

5. Select **OK**.

Set up continuous deployment

To enable continuous deployment:

1. On the **Pipeline** tab, under **Artifacts**, select the lightning icon in the build artifact.

2. In the right pane, set the **Continuous deployment trigger** to **Enabled**.

3. Select **Save** at upper right, and then select **Save** again.

Deploy the Java app

Now that you enabled CI/CD, modifying the source code creates and runs builds and releases automatically. You can also create and run releases manually, as follows:

1. At the upper right on the release pipeline page, select **Create release** .

2. On the **Create a new release** page, select the stage name under **Stages for a trigger change from automated to manual**.

3. Select **Create**.

4. Select the release name, hover over or select the stage, and then select **Deploy**.

Test the Java web app

After deployment completes successfully, test your web app.

1. Copy the web app URL from the Azure portal.

2. Enter the URL in your web browser to run your app. The web page should say **Hello Azure!**

https://microprofilewebapp.azurewebsites.net/api/hello
Hello Azure!

Build, test, & deploy Android apps

Article • 03/27/2023

Azure DevOps Services

You can set up pipelines to automatically build, test, and deploy Android applications.

Prerequisites

You must have the following items:

- GitHub account. If you don't have a GitHub account, create one now.
- Azure DevOps project. If you don't have a project, create one now.

Set up pipeline

Do the following tasks to set up a pipeline for a sample Android application.

1. Fork the following repository to your GitHub account to get the code for a simple Android application.

   ```
   https://github.com/MicrosoftDocs/pipelines-android
   ```

2. Sign in to your Azure DevOps organization and go to your project.

3. Select **Pipelines** > **Create pipeline** or **New pipeline**.

4. Select **GitHub** as the location of your source code.

You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.

5. Select the **-android** repository that you previously forked.
6. Select **Approve and install** on the screen that follows.

   Azure Pipelines generates a YAML file for your pipeline.

7. Select **Run**.
8. **Commit directly to the main branch**, and then choose **Run** again.
9. Wait for the run to finish.

You have a working YAML file (`azure-pipelines.yml`) in your repository that's ready for you to customize.

> **Tip**
>
> To make changes to the YAML file, select the pipeline in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file.

Build with Gradle

Gradle is a common build tool used for building Android projects. For more information about your options, see the Gradle task.

YAML

```yaml
# https://learn.microsoft.com/azure/devops/pipelines/ecosystems/android
pool:
  vmImage: 'macOS-latest'

steps:
- task: Gradle@2
  inputs:
    workingDirectory: ''
    gradleWrapperFile: 'gradlew'
    gradleOptions: '-Xmx3072m'
    publishJUnitResults: false
    testResultsFiles: '**/TEST-*.xml'
    tasks: 'assembleDebug'
```

Adjust the build path

- Adjust the **workingDirectory** value if your `gradlew` file isn't in the root of the repository. The directory value should be similar to the root of the repository, such as `AndroidApps/MyApp` or `$(system.defaultWorkingDirectory)/AndroidApps/MyApp`.
- Adjust the **gradleWrapperFile** value if your `gradlew` file isn't in the root of the repository. The file path value should be similar to the root of the repository, such as `AndroidApps/MyApp/gradlew` or `$(system.defaultWorkingDirectory)/AndroidApps/MyApp/gradlew`.

Adjust Gradle tasks

Adjust the **tasks** value for the build variant you prefer, such as `assembleDebug` or `assembleRelease`. For more information, see the following Google Android development documentation:

- Build a debug APK and
- Configure build variants.

Sign and align an Android Package (APK)

If your build doesn't already sign and zipalign the APK, add the Android Signing task to the YAML. An APK must be signed to run on a device instead of an emulator. Zipaligning reduces the RAM consumed by the application.

> **Important**
>
> We recommend storing each of the following passwords in a **secret variable**.

YAML

```yaml
- task: AndroidSigning@2
  inputs:
    apkFiles: '**/*.apk'
    jarsign: true
    jarsignerKeystoreFile: 'pathToYourKeystoreFile'
    jarsignerKeystorePassword: '$(jarsignerKeystorePassword)'
    jarsignerKeystoreAlias: 'yourKeystoreAlias'
    jarsignerKeyPassword: '$(jarsignerKeyPassword)'
    zipalign: true
```

Test

Test on the Android Emulator

> **Note**
>
> The Android Emulator is currently available only on the **Hosted macOS** agent.

Create the Bash task and copy paste the code below in order to install and run the emulator. Don't forget to arrange the emulator parameters to fit your testing environment. The emulator starts as a background process and is available in later tasks.

Bash

```bash
#!/usr/bin/env bash

# Install AVD files
echo "y" | $ANDROID_HOME/cmdline-tools/latest/bin/sdkmanager --install
'system-images;android-27;google_apis;x86'

# Create emulator
echo "no" | $ANDROID_HOME/tools/bin/avdmanager create avd -n
xamarin_android_emulator -k 'system-images;android-27;google_apis;x86' --
force
```

```
$ANDROID_HOME/emulator/emulator -list-avds

echo "Starting emulator"

# Start emulator in background
nohup $ANDROID_HOME/emulator/emulator -avd xamarin_android_emulator -no-
snapshot > /dev/null 2>&1 &
$ANDROID_HOME/platform-tools/adb wait-for-device shell 'while [[ -z
$(getprop sys.boot_completed | tr -d '\r') ]]; do sleep 1; done; input
keyevent 82'

$ANDROID_HOME/platform-tools/adb devices

echo "Emulator started"
```

Test on Azure-hosted devices

Add the App Center Test task to test the application in a hosted lab of iOS and Android devices. An App Center free trial is required, which must later be converted to paid.

Sign up with App Center first.

YAML

```yaml
# App Center test
# Test app packages with Visual Studio App Center
- task: AppCenterTest@1
  inputs:
    appFile:
    #artifactsDirectory: '$(Build.ArtifactStagingDirectory)/AppCenterTest'
    #prepareTests: true # Optional
    #frameworkOption: 'appium' # Required when prepareTests == True#
Options: appium, espresso, calabash, uitest, xcuitest
    #appiumBuildDirectory: # Required when prepareTests == True && Framework
== Appium
    #espressoBuildDirectory: # Optional
    #espressoTestApkFile: # Optional
    #calabashProjectDirectory: # Required when prepareTests == True &&
Framework == Calabash
    #calabashConfigFile: # Optional
    #calabashProfile: # Optional
    #calabashSkipConfigCheck: # Optional
    #uiTestBuildDirectory: # Required when prepareTests == True && Framework
== Uitest
    #uitestStorePath: # Optional
    #uiTestStorePassword: # Optional
    #uitestKeyAlias: # Optional
    #uiTestKeyPassword: # Optional
    #uiTestToolsDirectory: # Optional
    #signInfo: # Optional
```

```
    #xcUITestBuildDirectory: # Optional
    #xcUITestIpaFile: # Optional
    #prepareOptions: # Optional
    #runTests: true # Optional
    #credentialsOption: 'serviceEndpoint' # Required when runTests == True# 
Options: serviceEndpoint, inputs
    #serverEndpoint: # Required when runTests == True && CredsType == 
ServiceEndpoint
    #username: # Required when runTests == True && CredsType == Inputs
    #password: # Required when runTests == True && CredsType == Inputs
    #appSlug: # Required when runTests == True
    #devices: # Required when runTests == True
    #series: 'master' # Optional
    #dsymDirectory: # Optional
    #localeOption: 'en_US' # Required when runTests == True# Options: da_DK, 
nl_NL, en_GB, en_US, fr_FR, de_DE, ja_JP, ru_RU, es_MX, es_ES, user
    #userDefinedLocale: # Optional
    #loginOptions: # Optional
    #runOptions: # Optional
    #skipWaitingForResults: # Optional
    #cliFile: # Optional
    #showDebugOutput: # Optional
```

Keep artifacts with the build record

Add the Copy Files and Publish Build Artifacts tasks. Your APK gets stored with the build record or test, and gets deployed in later pipelines. For more information, see Artifacts.

YAML

```yaml
- task: CopyFiles@2
  inputs:
    contents: '**/*.apk'
    targetFolder: '$(build.artifactStagingDirectory)'
- task: PublishBuildArtifacts@1
```

Deploy

Add App Center

Add the App Center Distribute task to distribute an application to a group of testers or beta users, or promote the application to Intune or Google Play. A free App Center account is required (no payment is necessary).

YAML

```yaml
# App Center distribute
# Distribute app builds to testers and users via Visual Studio App Center
- task: AppCenterDistribute@1
  inputs:
    serverEndpoint:
    appSlug:
    appFile:
    #symbolsOption: 'Apple' # Optional. Options: apple
    #symbolsPath: # Optional
    #symbolsPdbFiles: '**/*.pdb' # Optional
    #symbolsDsymFiles: # Optional
    #symbolsMappingTxtFile: # Optional
    #symbolsIncludeParentDirectory: # Optional
    #releaseNotesOption: 'input' # Options: input, file
    #releaseNotesInput: # Required when releaseNotesOption == Input
    #releaseNotesFile: # Required when releaseNotesOption == File
    #isMandatory: false # Optional
    #distributionGroupId: # Optional
```

Install Google Play

Install the Google Play extension and use the following tasks to automate interaction with Google Play. By default, these tasks authenticate to Google Play using a service connection that you configure.

Release

Add the Google Play Release task to release a new Android app version to the Google Play store.

YAML

```yaml
- task: GooglePlayRelease@4
  inputs:
    apkFile: '**/*.apk'
    serviceEndpoint: 'yourGooglePlayServiceConnectionName'
    track: 'internal'
```

Promote

Add the Google Play Promote task to promote a previously released Android application update from one track to another, such as `alpha` → `beta`.

YAML

```yaml
- task: GooglePlayPromote@3
  inputs:
    packageName: 'com.yourCompany.appPackageName'
    serviceEndpoint: 'yourGooglePlayServiceConnectionName'
    sourceTrack: 'internal'
    destinationTrack: 'alpha'
```

Increase rollout

Add the Google Play Increase Rollout task to increase the rollout percentage of an application that was previously released to the `rollout` track.

YAML

```yaml
- task: GooglePlayIncreaseRollout@2
  inputs:
    packageName: 'com.yourCompany.appPackageName'
    serviceEndpoint: 'yourGooglePlayServiceConnectionName'
    userFraction: '0.5' # 0.0 to 1.0 (0% to 100%)
```

Status update

Add the Google Play Status Update task to update the rollout status for the application that was previously released to the `rollout` track.

YAML

```yaml
  - task: GooglePlayStatusUpdate@2
    inputs:
      authType: ServiceEndpoint
      packageName: 'com.yourCompany.appPackageName'
      serviceEndpoint: 'yourGooglePlayServiceConnectionName'
      status: 'inProgress' # draft | inProgress | halted | completed
```

Related extensions

- Codified Security (Codified Security)
- Google Play (Microsoft)
- Mobile App Tasks for iOS and Android (James Montemagno)
- Mobile Testing Lab (Perfecto Mobile)
- React Native (Microsoft)

FAQ

Q: How do I create app bundles?

A: You can build and sign your app bundle with an inline script and a secure file. To do so, first download your keystore and store it as a secure file in the Library. Then, create variables for `keystore.password`, `key.alias`, and `key.password` in a variable group.

Next, use the Download Secure File and Bash tasks to download your keystore and build and sign your app bundle.

In this YAML file, download an `app.keystore` secure file and use a bash script to generate an app bundle. Then, use Copy Files to copy the app bundle. From there, create and save an artifact with Publish Build Artifact or use the Google Play extension to publish.

YAML

```yaml
- task: DownloadSecureFile@1
  name: keyStore
  displayName: "Download keystore from secure files"
  inputs:
    secureFile: app.keystore

- task: Bash@3
  displayName: "Build and sign App Bundle"
  inputs:
    targetType: "inline"
    script: |
      msbuild -restore $(Build.SourcesDirectory)/myAndroidApp/*.csproj -
t:SignAndroidPackage -p:AndroidPackageFormat=aab -
p:Configuration=$(buildConfiguration) -p:AndroidKeyStore=True -
p:AndroidSigningKeyStore=$(keyStore.secureFilePath) -
p:AndroidSigningStorePass=$(keystore.password) -
p:AndroidSigningKeyAlias=$(key.alias) -
p:AndroidSigningKeyPass=$(key.password)

- task: CopyFiles@2
  displayName: 'Copy deliverables'
  inputs:
    SourceFolder:
'$(Build.SourcesDirectory)/myAndroidApp/bin/$(buildConfiguration)'
    Contents: '*.aab'
    TargetFolder: 'drop'
```

Build and test Go projects

Article • 09/22/2022

Azure DevOps Services

Use a pipeline to automatically build and test your Go projects.

Create your first pipeline

New to Azure Pipelines? If so, then we recommend you try this section before moving on to other sections.

Fork the following repo at GitHub:

```
https://github.com/MicrosoftDocs/pipelines-go
```

Sign in to Azure Pipelines

Sign-in to Azure Pipelines. After you sign in, your browser goes to `https://dev.azure.com/my-organization-name` and displays your Azure DevOps dashboard.

Within your selected organization, create a *project*. If you don't have any projects in your organization, you see a **Create a project to get started** screen. Otherwise, select the **New Project** button in the upper-right corner of the dashboard.

Create the pipeline

1. Sign-in to your Azure DevOps organization and go to your project.
2. Go to **Pipelines**, and then select **New pipeline**.
3. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.
4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.
5. When you see the list of repositories, select your repository.

6. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.

When the **Configure** tab appears, select **Go**. Your new pipeline appears, with the `azure-pipelines.yml` YAML file ready to be configured. See the following sections to learn some of the more common ways to customize your pipeline.

Build environment

You can use Azure Pipelines to build your Go projects without setting up any infrastructure of your own. You can use Linux, macOS, or Windows agents to run your builds.

Update the following snippet in your `azure-pipelines.yml` file to select the appropriate image.

YAML

```yaml
pool:
  vmImage: 'ubuntu-latest'
```

Modern versions of Go are pre-installed on Microsoft-hosted agents. For the exact versions of pre-installed Go, refer to Microsoft-hosted agents in Azure Pipelines.

Set up Go

Go 1.11+

Starting with Go 1.11, you no longer need to define a `$GOPATH` environment, set up a workspace layout, or use the `dep` module. Dependency management is now built in.

This YAML implements the `go get` command to download Go packages and their dependencies. It then uses `go build` to generate the content that is published with `PublishBuildArtifacts@1` task.

YAML

```yaml
trigger:
 - main

pool:
```

```yaml
    vmImage: 'ubuntu-latest'

  steps:
  - task: GoTool@0
    inputs:
      version: '1.13.5'
  - task: Go@0
    inputs:
      command: 'get'
      arguments: '-d'
      workingDirectory: '$(System.DefaultWorkingDirectory)'
  - task: Go@0
    inputs:
      command: 'build'
      workingDirectory: '$(System.DefaultWorkingDirectory)'
  - task: CopyFiles@2
    inputs:
      TargetFolder: '$(Build.ArtifactStagingDirectory)'
  - task: PublishBuildArtifacts@1
    inputs:
       artifactName: drop
```

Build

Use `go build` to build your Go project. Add the following snippet to your `azure-pipelines.yml` file:

YAML

```yaml
- task: Go@0
  inputs:
    command: 'build'
    workingDirectory: '$(System.DefaultWorkingDirectory)'
```

Test

Use `go test` to test your go module and its subdirectories (`./...`). Add the following snippet to your `azure-pipelines.yml` file:

YAML

```yaml
- task: Go@0
  inputs:
    command: 'test'
    arguments: '-v'
    workingDirectory: '$(System.DefaultWorkingDirectory)'
```

When you're ready, Commit a new *azure-pipelines.yml* file to your repository and update the commit message. Select **Save and run**.

If you want to watch your pipeline in action, select the build in the **Jobs** option on your Azure Pipelines dashboard.

Because your code appeared to be a good match for the Go template, we automatically created your pipeline.

You now have a working YAML pipeline (`azure-pipelines.yml`) in your repository that's ready for you to customize!

When you're ready to make changes to your pipeline, select it in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file.

Tip

To make changes to the YAML file as described in this article, select the pipeline in **Pipelines** page, and then select **Edit** to open an editor for the `azure-pipelines.yml` file.

Build an image and push to container registry

For your Go app, you can also build an image and push it to a container registry.

Related extensions

Go extension for Visual Studio Code (Microsoft)

Build and test PHP apps

Article • 11/28/2022

Azure DevOps Services

Use Azure Pipelines continuous integration and continuous delivery (CI/CD) to build, deploy, and test your PHP projects.

Learn how to create a PHP pipeline, deploy a pipeline with a sample project to Azure App Service, and how to configure your environment.

To learn more about Azure App Service, see Create a PHP web app in Azure App Service.

Prerequisites

Make sure you have the following items:

- A GitHub account where you can create a repository. Create one for free.
- An Azure DevOps organization. Create one for free. If your team already has one, then make sure you're an administrator of the Azure DevOps project that you want to use.
- An ability to run pipelines on Microsoft-hosted agents. To use Microsoft-hosted agents, your Azure DevOps organization must have access to Microsoft-hosted parallel jobs. You can either purchase a parallel job or you can request a free grant.
- An Azure account. If you don't have one, you can create one for free.

> **Tip**
>
> If you're new at this, the easiest way to get started is to use the same email address as the owner of both the Azure Pipelines organization and the Azure subscription.

Get the code

If you already have an app at GitHub that you want to deploy, you can create a pipeline for that code. But, if you're a new user, you might get a better start by using our sample code. In that case, fork the following repo at GitHub:

```
https://github.com/Azure-Samples/basic-php-composer
```

Create a pipeline

1. Sign in to your Azure DevOps organization and go to your project.

2. Select **PHP** in the **Configure** tab.

3. Examine your new pipeline. When you're ready, select **Save and run**.

4. You're prompted to commit a new *azure-pipelines.yml* file to your repository. Select **Save and run** again.

   If you want to watch your pipeline in action, select the build job.

   You now have a working YAML pipeline (*azure-pipelines.yml*) in your repository that's ready for you to customize!

When you want to make changes to your pipeline, select your pipeline on the **Pipelines** page, and then **Edit** the *azure-pipelines.yml* file.

Read further to learn some of the more common ways to customize your pipeline.

Deploy to App Service

Use a pipeline to build a PHP web app and deploy to Azure App Service. Azure App Service is an HTTP-based service for hosting web applications, REST APIs, and mobile back ends.

You can use tasks to archive your files, publish a build artifact, and then use the Azure Web App task to deploy to Azure App Service.

This pipelines has two stages: Build and Deploy. In the Build stage, PHP 7.4 gets installed with composer. The app files are archived and uploaded into a package named `drop`. During the Deploy phase, the `drop` package gets deployed to Azure App Service as a web app.

YAML

```yaml
trigger:
- main

variables:
  # Azure Resource Manager connection created during pipeline creation
  azureSubscription: 'subscription-id'
  # Web app name
  webAppName: 'web-app-name'
  # Agent VM image name
  vmImageName: 'ubuntu-latest'
  # Environment name
  environmentName: 'environment-name'
  # Root folder under which your composer.json file is available.
  rootFolder: $(System.DefaultWorkingDirectory)

stages:
- stage: Build
  displayName: Build stage
  variables:
    phpVersion: '7.4'
  jobs:
  - job: BuildJob
    pool:
      vmImage: $(vmImageName)
    steps:
    - script: |
        sudo update-alternatives --set php /usr/bin/php$(phpVersion)
        sudo update-alternatives --set phar /usr/bin/phar$(phpVersion)
        sudo update-alternatives --set phpdbg /usr/bin/phpdbg$(phpVersion)
        sudo update-alternatives --set php-cgi /usr/bin/php-cgi$(phpVersion)
        sudo update-alternatives --set phar.phar
/usr/bin/phar.phar$(phpVersion)
        php -version
      workingDirectory: $(rootFolder)
      displayName: 'Use PHP version $(phpVersion)'

    - script: composer install --no-interaction --prefer-dist
      workingDirectory: $(rootFolder)
      displayName: 'Composer install'

    - task: ArchiveFiles@2
      displayName: 'Archive files'
      inputs:
        rootFolderOrFile: '$(rootFolder)'
        includeRootFolder: false
```

```yaml
          archiveType: zip
          archiveFile: $(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip
          replaceExistingArchive: true

      - upload: $(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip
        displayName: 'Upload package'
        artifact: drop

  - stage: Deploy
    displayName: 'Deploy Web App'
    dependsOn: Build
    condition: succeeded()
    jobs:
    - deployment: DeploymentJob
      pool:
        vmImage: $(vmImageName)
      environment: $(environmentName)
      strategy:
        runOnce:
          deploy:
            steps:
            - task: AzureWebApp@1
              displayName: 'Deploy Azure Web App'
              inputs:
                azureSubscription: $(azureSubscription)
                appName: $(webAppName)
                package: $(Pipeline.Workspace)/drop/$(Build.BuildId).zip
```

Configure build environment

Use Azure Pipelines to build your PHP projects without setting up infrastructure.

Use a specific PHP version

PHP is preinstalled on Microsoft-hosted agents, along with many common libraries per PHP version. You can use Linux, macOS, or Windows agents to run your builds. For more information and the exact versions of PHP that get preinstalled, see Microsoft-hosted agents.

On the Microsoft-hosted Ubuntu agent, multiple versions of PHP are installed. A symlink at `/usr/bin/php` points to the currently set PHP version, so that when you run `php`, the set version executes.

To use a PHP version other than the default, the symlink can be pointed to that version using the `update-alternatives` tool. Set the PHP version that you want by adding the following snippet to your `azure-pipelines.yml` file and change the value of the **phpVersion** variable.

YAML

```yaml
pool:
  vmImage: 'ubuntu-latest'

variables:
  phpVersion: 7.2

steps:
- script: |
    sudo update-alternatives --set php /usr/bin/php$(phpVersion)
    sudo update-alternatives --set phar /usr/bin/phar$(phpVersion)
    sudo update-alternatives --set phpdbg /usr/bin/phpdbg$(phpVersion)
    sudo update-alternatives --set php-cgi /usr/bin/php-cgi$(phpVersion)
    sudo update-alternatives --set phar.phar /usr/bin/phar.phar$(phpVersion)
    php -version
  displayName: 'Use PHP version $(phpVersion)'
```

Install dependencies

To use Composer to install dependencies, add the following snippet to your `azure-pipelines.yml` file.

YAML

```yaml
- script: composer install --no-interaction --prefer-dist
  displayName: 'composer install'
```

Test with phpunit

To run tests with phpunit, add the following snippet to your `azure-pipelines.yml` file.

YAML

```yaml
- script: ./phpunit
  displayName: 'Run tests with phpunit'
```

Retain the PHP app with the build record

To save the artifacts of this build with the build record, add the following snippet to your `azure-pipelines.yml` file. Optionally, customize the value of **rootFolderOrFile** to alter what is included in the archive.

YAML

```yaml
- task: ArchiveFiles@2
  inputs:
    rootFolderOrFile: '$(system.defaultWorkingDirectory)'
    includeRootFolder: false
- task: PublishBuildArtifacts@1
```

Using a custom composer location

If your composer.json is in a subfolder instead of the root directory, you can use the `--working-dir` argument to tell composer what directory to use. For example, if your composer.json is inside the subfolder `pkgs`

```
composer install --no-interaction --working-dir=pkgs
```

You can also specify the absolute path, using the built-in system variables:

```
composer install --no-interaction --working-dir='$(system.defaultWorkingDirectory)/pkgs'
```

Build and test Ruby apps

Article • 11/28/2022

Azure DevOps Services

This article explains how to automatically build Ruby projects.

Create the Azure Pipelines

Do the following steps to set up a pipeline for a Ruby app.

1. Sign in to your Azure DevOps organization and go to your project.
2. Select **Pipelines** > **New pipeline**.
3. Select **GitHub** as the location of your source code.

   You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.
4. Select your Ruby sample repository.
5. Select the `Ruby` template for your pipeline.
6. A YAML file gets generated. Select **Save and run** > **Commit directly to the main branch**, and then choose **Save and run** again.
7. Wait for the run to finish.

You have a working YAML file (`azure-pipelines.yml`) in your repository that's ready for you to customize.

> **Tip**
>
> To make changes to the YAML file as described in this article, select the pipeline in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file.

Build environment

You can use Azure Pipelines to build your Ruby projects without needing to set up any infrastructure of your own. Ruby is preinstalled on Microsoft-hosted agents in Azure Pipelines. You can use Linux, macOS, or Windows agents to run your builds.

For the exact versions of Ruby that are preinstalled, refer to Microsoft-hosted agents. To install a specific version of Ruby on Microsoft-hosted agents, add the Use Ruby Version task to the beginning of your pipeline.

Use a specific Ruby version

Add the Use Ruby Version task to set the version of Ruby used in your pipeline. This snippet adds Ruby 2.4 or later to the path and sets subsequent pipeline tasks to use it.

YAML

```yaml
# https://learn.microsoft.com/azure/devops/pipelines/ecosystems/ruby
pool:
  vmImage: 'ubuntu-latest' # other options: 'macOS-latest', 'windows-latest'

steps:
- task: UseRubyVersion@0
  inputs:
    versionSpec: '>= 2.5'
    addToPath: true
```

Install Rails

To install Rails, add the following snippet to your `azure-pipelines.yml` file.

YAML

```yaml
- script: gem install rails && rails -v
  displayName: 'gem install rails'
```

Install dependencies

To use Bundler to install dependencies, add the following snippet to your `azure-pipelines.yml` file.

YAML

```yaml
- script: |
    CALL gem install bundler
    bundle install --retry=3 --jobs=4
  displayName: 'bundle install'
```

Run Rake

To execute Rake in the context of the current bundle (as defined in your Gemfile), add the following snippet to your `azure-pipelines.yml` file.

YAML

```yaml
- script: bundle exec rake
  displayName: 'bundle exec rake'
```

Publish test results

The sample code includes unit tests written using RSpec. When Rake is run by the previous step, it runs the RSpec tests. The RSpec RakeTask in the Rakefile has been configured to produce JUnit style results using the RspecJUnitFormatter.

Add the Publish Test Results task to publish JUnit style test results to the server. You get a rich test reporting experience that you can use for troubleshooting any failed tests and for test timing analysis.

YAML

```yaml
- task: PublishTestResults@2
  condition: succeededOrFailed()
  inputs:
    testResultsFiles: '**/test-*.xml'
    testRunTitle: 'Ruby tests'
```

Publish code coverage results

The sample code uses SimpleCov to collect code coverage data when unit tests get run. SimpleCov is configured to use Cobertura and HTML report formatters.

Add the Publish Code Coverage Results task to publish code coverage results to the server. When you do so, coverage metrics can be seen in the build summary and HTML reports can be downloaded for further analysis.

YAML

```yaml
- task: PublishCodeCoverageResults@1
  inputs:
    codeCoverageTool: Cobertura
    summaryFileLocation: '$(System.DefaultWorkingDirectory)/**/coverage.xml'
    reportDirectory: '$(System.DefaultWorkingDirectory)/**/coverage'
```

Build an image and push to container registry

For your Ruby app, you can also build an image and push it to a container registry.

Build and deploy Xamarin apps with a pipeline

Article • 11/01/2023

Azure DevOps Services

Get started with Xamarin and Azure Pipelines by using a pipeline to deploy a Xamarin app. You can deploy Android and iOS apps in the same or separate pipelines.

Prerequisites

Have the following items:

- An Azure account with an active subscription. Create an account for free.
- An active Azure DevOps organization. Sign up for Azure Pipelines.

Get code

Fork the following repo at GitHub:

```
https://github.com/MicrosoftDocs/pipelines-xamarin
```

This sample is the `FirstApp` sample from the `https://github.com/xamarin/xamarin-forms-samples` repository. For more information on getting started with Xamarin, see Build your first Xamarin.Forms App.

Sign in to Azure Pipelines

Sign-in to Azure Pipelines. After you sign in, your browser goes to `https://dev.azure.com/my-organization-name` and displays your Azure DevOps dashboard.

Within your selected organization, create a *project*. If you don't have any projects in your organization, you see a **Create a project to get started** screen. Otherwise, select the **New Project** button in the upper-right corner of the dashboard.

Create the pipeline

1. Sign-in to your Azure DevOps organization and go to your project.
2. Go to **Pipelines**, and then select **New pipeline**.
3. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.
4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.
5. When you see the list of repositories, select your repository.
6. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.
7. On the **Configure** tab, select **Xamarin.Android** to build an Android project or **Xamarin.iOS** to build an iOS project. If you want to use the included sample `.yml` files, choose **Existing Azure Pipelines YAML file** and choose from one of the included sample pipelines (Android, iOS, or a combined pipeline that builds both).
8. When your new pipeline appears, take a look at the YAML to see what it does. When you're ready, select **Save and run**.

9. If you created a new YAML file, commit a new *azure-pipelines.yml* file to your repository. After you're happy with the message, select **Save and run** again.

   If you want to watch your pipeline in action, select the build job. You now have a working YAML pipeline (`azure-pipelines.yml`) in your repository that's ready for you to customize!

10. When you're ready to make changes to your pipeline, select it in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file.

Read further to learn some of the more common ways to customize your pipeline.

Set up Xamarin tools

You can use Azure Pipelines to build your Xamarin apps without setting up your own infrastructure. Xamarin tools are preinstalled on Microsoft-hosted agents in Azure Pipelines. You can use macOS or Windows agents to run Xamarin.Android builds, and macOS agents to run Xamarin.iOS builds. If you're using a self-hosted agent, install either of the following tools:

- Visual Studio Tools for Xamarin for Windows agents
- Visual Studio for Mac for macOS agents

For the exact versions of Xamarin that are preinstalled, refer to Microsoft-hosted agents.

Create a file named **azure-pipelines.yml** in the root of your repository. Then, add the following snippet to your `azure-pipelines.yml` file to select the appropriate agent pool:

YAML

```yaml
# https://learn.microsoft.com/azure/devops/pipelines/ecosystems/xamarin
pool:
  vmImage: 'macOS-12' # For Windows, use 'windows-2019'
```

Build a Xamarin.Android app

To build a Xamarin.Android app, add the following snippet to your `azure-pipelines.yml` file. Change values to match your project configuration. For more information about options, see the Xamarin.Android task.

YAML

```yaml
variables:
  buildConfiguration: 'Release'
  outputDirectory: '$(build.binariesDirectory)/$(buildConfiguration)'

steps:
- task: NuGetToolInstaller@1

- task: NuGetCommand@2
  inputs:
    restoreSolution: '**/*.sln'

- task: XamarinAndroid@1
  inputs:
    projectFile: '**/*Droid*.csproj'
    outputDirectory: '$(outputDirectory)'
    configuration: '$(buildConfiguration)'
    msbuildVersionOption: '16.0'
```

Sign a Xamarin.Android app

For information about signing your app, see Sign your mobile Android app during CI.

Build a Xamarin.iOS app

To build a Xamarin.iOS app, add the following snippet to your `azure-pipelines.yml` file. Change values to match your project configuration. For more information about options, see the Xamarin.iOS task.

YAML

```yaml
variables:
  buildConfiguration: 'Release'

steps:
- task: XamariniOS@2
  inputs:
    solutionFile: '**/*iOS.csproj'
    configuration: '$(buildConfiguration)'
    packageApp: false
    buildForSimulator: true
```

Sign and provision a Xamarin.iOS app - The PackageApp option

To generate a signed and provisioned Xamarin.iOS app .ipa package, set `packageApp` to `true`. Make sure you already installed the right Apple Provisioning Profile and Apple Certificates that match your App Bundle ID into the agent running the job.

To fulfill these mandatory requisites, use the Microsoft Provided tasks for installing an Apple Provisioning Profile and installing Apple Certificates.

YAML

```yaml
- task: XamariniOS@2
    inputs:
      solutionFile: '**/*iOS.csproj'
      configuration: 'AppStore'
      packageApp: true
```

Tip

The Xamarin.iOS build task only generates an .ipa package if the agent running the job has the **appropriate provisioning profile and Apple certificate installed**. If you enable the packageApp option and the agent doesn't have the appropriate apple provisioning profile(.mobileprovision) and apple certificate(.p12) the build may succeed, but the .ipa isn't generated.

For Microsoft Hosted agents, the .ipa package is by default located under the following path:
`{iOS.csproj root}/bin/{Configuration}/{iPhone/iPhoneSimulator}/`

You can configure the output path by adding an argument to the Xamarin.iOS task:

YAML

YAML

```yaml
- task: XamariniOS@2
    inputs:
      solutionFile: '**/*iOS.csproj'
      configuration: 'AppStore'
      packageApp: true
      args: /p:IpaPackageDir="/Users/vsts/agent/2.153.2/work/1/a"
```

This example locates the .ipa in the Build Artifact Staging Directory. It's ready to get pushed into Azure DevOps as an artifact to each build run. To push it into Azure DevOps, add a Publish Artifact task to the end of your pipeline.

For more information about signing and provisioning your iOS app, see Sign your mobile iOS app during CI.

Set the Xamarin SDK version on macOS

To set a specific Xamarin SDK version to use on the Microsoft-hosted macOS agent pool, add the following snippet before the `XamariniOS` task in your `azure-pipelines.yml` file. For details on properly formatting the version number (shown as **5_4_1** below), see How can I manually select versions of tools on the Hosted macOS agent?.

YAML

```yaml
- script: sudo $AGENT_HOMEDIRECTORY/scripts/select-xamarin-sdk.sh 5_4_1
  displayName: 'Select Xamarin SDK version'
```

Build Xamarin.Android and Xamarin.iOS apps with one pipeline

You can build and test your Xamarin.Android app, Xamarin.iOS app, and related apps in the same pipeline by defining multiple jobs in `azure-pipelines.yml`. These jobs can run in parallel to save time. The following complete example builds a Xamarin.Android app on Windows, and a Xamarin.iOS app on macOS, using two jobs.

YAML

```yaml
# Xamarin.Android and Xamarin.iOS
# Build a Xamarin.Android and Xamarin.iOS app.
# Add steps that test, sign, and distribute the app, save build
artifacts, and more:
# https://learn.microsoft.com/azure/devops/pipelines/ecosystems/xamarin

jobs:

- job: Android
  pool:
    vmImage: 'windows-2019'

  variables:
    buildConfiguration: 'Release'
    outputDirectory: '$(build.binariesDirectory)/$(buildConfiguration)'

  steps:
  - task: NuGetToolInstaller@1

  - task: NuGetCommand@2
    inputs:
      restoreSolution: '**/*.sln'

  - task: XamarinAndroid@1
    inputs:
      projectFile: '**/*droid*.csproj'
      outputDirectory: '$(outputDirectory)'
      configuration: '$(buildConfiguration)'
      msbuildVersionOption: '16.0'

  - task: AndroidSigning@3
    inputs:
      apksign: false
      zipalign: false
      apkFiles: '$(outputDirectory)/*.apk'

  - task: PublishBuildArtifacts@1
    inputs:
      pathtoPublish: '$(outputDirectory)'
```

```yaml
- job: iOS
  pool:
    vmImage: 'macOS-12'

  steps:
  # To manually select a Xamarin SDK version on the Hosted macOS agent,
enable this script with the SDK version you want to target
  # https://go.microsoft.com/fwlink/?linkid=871629
  - script: sudo $AGENT_HOMEDIRECTORY/scripts/select-xamarin-sdk.sh
5_4_1
    displayName: 'Select Xamarin SDK version'
    enabled: false

  - task: NuGetToolInstaller@1

  - task: NuGetCommand@2
    inputs:
      restoreSolution: '**/*.sln'

  - task: XamariniOS@2
    inputs:
      solutionFile: '**/*.sln'
      configuration: 'Release'
      buildForSimulator: true
      packageApp: false
```

Clean up resources

If you don't need the example code, delete your GitHub repository and Azure Pipelines project.

Next steps

Learn more about using Xcode in pipelines

Learn more about using Android in pipelines

Build, test, and deploy Xcode apps

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Learn how to build and deploy Xcode projects with Azure Pipelines.

Prerequisites

- An Xcode 9+ project in a GitHub repository. If you do not have a project, see Creating an Xcode Project for an App

Create the pipeline

1. Sign-in to your Azure DevOps organization and go to your project.
2. Go to **Pipelines**, and then select **New pipeline**.
3. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.
4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.
5. When you see the list of repositories, select your repository.
6. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.

> When the **Configure** tab appears, select **Xcode**.

7. When your new pipeline appears, take a look at the YAML to see what it does. When you're ready, select **Save and run**.

8. You're prompted to commit a new *azure-pipelines.yml* file to your repository. After you're happy with the message, select **Save and run** again.

   If you want to watch your pipeline in action, select the build job.

   > You just created and ran a pipeline that we automatically created for you, because your code appeared to be a good match for the Xcode template.

   You now have a working YAML pipeline (`azure-pipelines.yml`) in your repository that's ready for you to customize!

9. When you're ready to make changes to your pipeline, select it in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file.

See the sections below to learn some of the more common ways to customize your pipeline.

> **Tip**
>
> To make changes to the YAML file as described in this topic, select the pipeline in **Pipelines** page, and then select **Edit** to open an editor for the `azure-pipelines.yml` file.

Build environment

You can use Azure Pipelines to build your apps with Xcode without needing to set up any infrastructure of your own. Xcode is preinstalled on Microsoft-hosted macOS agents in Azure Pipelines. You can use the macOS agents to run your builds.

For the exact versions of Xcode that are preinstalled, refer to Microsoft-hosted agents.

Create a file named **azure-pipelines.yml** in the root of your repository. Then, add the following snippet to your `azure-pipelines.yml` file to select the appropriate agent pool:

YAML

```yaml
# https://learn.microsoft.com/azure/devops/pipelines/ecosystems/xcode
pool:
  vmImage: 'macOS-latest'
```

Build an app with Xcode

To build an app with Xcode, add the following snippet to your `azure-pipelines.yml` file. This is a minimal snippet for building an iOS project using its default scheme, for the Simulator, and without packaging. Change values to match your project configuration. See the Xcode task for more about these options.

YAML

```yaml
pool:
  vmImage: 'macos-latest'

steps:
- task: Xcode@5
  inputs:
    actions: 'build'
    scheme: ''
    sdk: 'iphoneos'
    configuration: 'Release'
    xcWorkspacePath: '**/*.xcodeproj/project.xcworkspace'
    xcodeVersion: 'default' # Options: 10, 11, 12, 13, 14, default,
specifyPath
```

Signing and provisioning

An Xcode app must be signed and provisioned to run on a device or be published to the App Store. The signing and provisioning process needs access to your P12 signing certificate and one or more provisioning profiles. The Install Apple Certificate and Install Apple Provisioning Profile tasks make these available to Xcode during a build.

See Sign your mobile app to learn more.

Carthage

If your project uses Carthage with a private Carthage repository, you can set up authentication by setting an environment variable named `GITHUB_ACCESS_TOKEN` with a value of a token that has access to the repository. Carthage will automatically detect and use this environment variable.

Do not add the secret token directly to your pipeline YAML. Instead, create a new pipeline variable with its lock enabled on the Variables pane to encrypt this value. See secret variables.

Here is an example that uses a secret variable named `myGitHubAccessToken` for the value of the `GITHUB_ACCESS_TOKEN` environment variable.

YAML

```
- script: carthage update --platform iOS
  env:
    GITHUB_ACCESS_TOKEN: $(myGitHubAccessToken)
```

Testing on Azure-hosted devices

Add the App Center Test task to test the app in a hosted lab of iOS and Android devices. An App Center free trial is required which must later be converted to paid.

Sign up with App Center first.

YAML

```
# App Center test
# Test app packages with Visual Studio App Center
- task: AppCenterTest@1
  inputs:
    appFile:
    #artifactsDirectory: '$(Build.ArtifactStagingDirectory)/AppCenterTest'
    #prepareTests: true # Optional
    #frameworkOption: 'appium' # Required when prepareTests == True#
Options: appium, espresso, calabash, uitest, xcuitest
    #appiumBuildDirectory: # Required when prepareTests == True && Framework
== Appium
    #espressoBuildDirectory: # Optional
    #espressoTestApkFile: # Optional
    #calabashProjectDirectory: # Required when prepareTests == True &&
Framework == Calabash
    #calabashConfigFile: # Optional
    #calabashProfile: # Optional
    #calabashSkipConfigCheck: # Optional
    #uiTestBuildDirectory: # Required when prepareTests == True && Framework
== Uitest
    #uitestStorePath: # Optional
    #uiTestStorePassword: # Optional
    #uitestKeyAlias: # Optional
    #uiTestKeyPassword: # Optional
    #uiTestToolsDirectory: # Optional
    #signInfo: # Optional
    #xcUITestBuildDirectory: # Optional
    #xcUITestIpaFile: # Optional
    #prepareOptions: # Optional
    #runTests: true # Optional
    #credentialsOption: 'serviceEndpoint' # Required when runTests == True#
Options: serviceEndpoint, inputs
    #serverEndpoint: # Required when runTests == True && CredsType ==
ServiceEndpoint
    #username: # Required when runTests == True && CredsType == Inputs
    #password: # Required when runTests == True && CredsType == Inputs
    #appSlug: # Required when runTests == True
```

```
    #devices: # Required when runTests == True
    #series: 'master' # Optional
    #dsymDirectory: # Optional
    #localeOption: 'en_US' # Required when runTests == True# Options: da_DK,
nl_NL, en_GB, en_US, fr_FR, de_DE, ja_JP, ru_RU, es_MX, es_ES, user
    #userDefinedLocale: # Optional
    #loginOptions: # Optional
    #runOptions: # Optional
    #skipWaitingForResults: # Optional
    #cliFile: # Optional
    #showDebugOutput: # Optional
```

Retain artifacts with the build record

Add the Copy Files and Publish Build Artifacts tasks to store your IPA with the build record or test and deploy it in subsequent pipelines. See Artifacts.

YAML

```yaml
- task: CopyFiles@2
  inputs:
    contents: '**/*.ipa'
    targetFolder: '$(build.artifactStagingDirectory)'
- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory)'
    ArtifactName: 'drop'
    publishLocation: 'Container'
```

Deploy

App Center

Add the App Center Distribute task to distribute an app to a group of testers or beta users, or promote the app to Intune or the Apple App Store. A free App Center account is required (no payment is necessary).

YAML

```yaml
# App Center distribute
# Distribute app builds to testers and users via Visual Studio App Center
- task: AppCenterDistribute@1
  inputs:
    serverEndpoint:
    appSlug:
    appFile:
```

```
    #symbolsOption: 'Apple' # Optional. Options: apple
    #symbolsPath: # Optional
    #symbolsPdbFiles: '**/*.pdb' # Optional
    #symbolsDsymFiles: # Optional
    #symbolsMappingTxtFile: # Optional
    #symbolsIncludeParentDirectory: # Optional
    #releaseNotesOption: 'input' # Options: input, file
    #releaseNotesInput: # Required when releaseNotesOption == Input
    #releaseNotesFile: # Required when releaseNotesOption == File
    #isMandatory: false # Optional
    #distributionGroupId: # Optional
```

Apple App Store

Install the Apple App Store extension and use the following tasks to automate interaction with the App Store. By default, these tasks authenticate to Apple using a service connection that you configure.

Release

Add the App Store Release task to automate the release of updates to existing iOS TestFlight beta apps or production apps in the App Store.

See limitations of using this task with Apple two-factor authentication, since Apple authentication is region-specific and fastlane session tokens expire quickly and must be recreated and reconfigured.

YAML

```yaml
- task: AppStoreRelease@1
  displayName: 'Publish to the App Store TestFlight track'
  inputs:
    serviceEndpoint: 'My Apple App Store service connection' # This service
connection must be added by you
    appIdentifier: com.yourorganization.testapplication.etc
    ipaPath: '$(build.artifactstagingdirectory)/**/*.ipa'
    shouldSkipWaitingForProcessing: true
    shouldSkipSubmission: true
```

Promote

Add the App Store Promote task to automate the promotion of a previously submitted app from iTunes Connect to the App Store.

YAML

```
- task: AppStorePromote@1
  displayName: 'Submit to the App Store for review'
  inputs:
    serviceEndpoint: 'My Apple App Store service connection' # This service
connection must be added by you
    appIdentifier: com.yourorganization.testapplication.etc
    shouldAutoRelease: false
```

Related extensions

- Apple App Store (Microsoft)
- Codified Security (Codified Security)
- MacinCloud (Moboware Inc.)
- Mobile App Tasks for iOS and Android (James Montemagno)
- Mobile Testing Lab (Perfecto Mobile)
- Raygun (Raygun)
- React Native (Microsoft)
- Version Setter (Tom Gilder)

Build multiple branches in Azure Pipelines

Article • 06/15/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Using Azure Pipelines, you can create triggers to build your project on every new commit and pull request to your repository. In this article, you will learn how to enable continuous integration and set up multiple branch builds for your repository.

Prerequisites

- An Azure DevOps organization and a project. Create an organization or a project if you haven't already.
- A working pipeline. Follow the instructions in Create your first pipeline to create your pipeline.

Enable CI trigger for a topic branch

When working with Git, it is a common practice to create temporary branches from the main branch to facilitate a streamlined workflow. These branches, often referred to as topic or feature branches, serve the purpose of isolating your work. Within this workflow, you create a branch dedicated to a specific feature or bug fix, and once completed, you merge the code back into the main branch before deleting the topic branch.

YAML

If no trigger is explicitly specified in your YAML file, any changes made to any branch will trigger a run. To add triggers for both the *main* branch and any *feature/* branches, include the following snippet in your YAML file. This will ensure that any modifications made to these branches will automatically trigger a pipeline run.

YAML

```yaml
trigger:
- main
- feature/*
```

Customize build tasks based on the branch being built

The main branch is usually responsible for generating deployable artifacts, such as binaries. For short-lived feature branches, there is no need to invest time in creating and storing these artifacts. In Azure Pipelines, you can implement custom conditions to ensure that specific tasks are executed only on the main branch.

YAML

Edit the *azure-pipelines.yml* file in your *main* branch, and add a condition to your desired task. For example, the following snippet adds a condition to the publish pipeline artifacts task.

YAML

```yaml
- task: PublishPipelineArtifact@1
  condition: and(succeeded(), eq(variables['Build.SourceBranch'],
'refs/heads/main'))
```

Validate pull requests

To ensure branch protection, you can utilize policies that mandate successful builds prior to merging pull requests. Using Azure Pipelines, you have the flexibility to configure the requirement of a new successful build for merging changes into crucial branches like the main branch.

GitHub repository

YAML

If you don't explicitly define **pr** triggers in your YAML file, pull request builds will be enabled by default for all branches. However, you have the flexibility to specify the target branches for your pull request builds. As an example, if you want to run the build exclusively for pull requests targeting the *main* branch and branches starting with *feature/*, you can specify the following configuration:

```yaml
pr:
- main
- feature/*
```

Azure Repos repository

1. Sign in to your Azure DevOps organization, and then navigate to your project.
2. Select **Repos** and then select **Branches**.
3. Select the ellipsis icon to the right of your branch name, and then select **Branch policies**.
4. Under the **Build validation** menu, select the + sign to add a build policy.
5. Select your **Build pipeline** from the dropdown menu and make sure that **Trigger** is set to automatic and the **Policy requirement** is set to required.
6. Enter a descriptive **Display name** to describe the policy.
7. Select **Save** to create and enable the policy. Select **Save changes** at the top left of your screen to save your changes.

1. To test the policy navigate to **Repos** > **Pull requests** in the Azure DevOps portal.
2. Select **New pull request** and make sure that your topic branch is set to merge into your main branch, and then Select **Create**.
3. On your screen, you can see the currently executing policy.
4. Select the **policy name** to examine the build. If the build succeeds your pull request will be merged. If the build fails the merge will be blocked.

Note

Azure Pipelines no longer supports per-pipeline retention policies. We recommend using project-level retention rules.

Related articles

- Deploy from multiple branches
- Deploy pull request Artifacts
- Configure retention policies

Publish Pipeline Artifacts

Article • 08/29/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Azure Artifacts enable developers to store and manage their packages and control who they want to share it with. Pipeline Artifacts are generated after you build your application. The output can then be deployed or consumed by another job in your pipeline.

Publish Artifacts

> **Note**
>
> Publish pipeline artifacts is not supported in release pipelines. It is only supported in build pipelines, multi-stage pipelines, and yaml pipelines.

You can publish your Artifacts at any stage of your pipeline using YAML or the classic editor. You won't be billed for storing your Pipeline Artifacts or using Pipeline caching.

YAML

YAML

```yaml
- task: PublishPipelineArtifact@1
  inputs:
    targetPath: '$(Pipeline.Workspace)'
    publishLocation: 'pipeline'
    artifact: 'drop'
```

- **targetPath**: (Required) The path of the file or directory to publish. Can be absolute or relative to the default working directory. Can include variables, but wildcards are not supported. Default value: $(Pipeline.Workspace).
- **publishLocation**: (Required). Artifacts publish location: choose whether to store the artifact in Azure Pipelines, or to copy it to a file share that must be accessible from the pipeline agent. Options: pipeline, filepath. Default value: pipeline.

- **artifact**: (Optional) Name of the artifact to publish. If not set, defaults to a unique ID scoped to the job.

Publish Artifacts from the command line

If you want to manually publish your Artifact, run the following command in an elevated command prompt:

Azure CLI

```
az pipelines runs artifact upload --artifact-name your_artifact_name --path
your_path_to_publish --run-id '<artifact_run_id>'
```

View published Artifacts

When your pipeline run is completed, you can view or download your published Artifact as follows

1. Select your pipeline run, and then select the **Summary** tab.

2. Select the published Artifact in the related section.

3. Expand the drop folder and find your Artifact.

4. Download your pipeline Artifact and explore its content.

Related articles

- Releases in Azure Pipelines
- Multi-stage release pipeline
- Deploy from multiple branches

Service containers

Article • 01/24/2023

Azure DevOps Services

If your pipeline requires the support of one or more services, in many cases you'll want to create, connect to, and clean up each service on a per-job basis. For instance, a pipeline may run integration tests that require access to a database and a memory cache. The database and memory cache need to be freshly created for each job in the pipeline.

A container provides a simple and portable way to run a service that your pipeline depends on. A *service container* enables you to automatically create, network, and manage the lifecycle of your containerized service. Each service container is accessible by only the job that requires it. Service containers work with any kind of job, but they're most commonly used with container jobs.

Requirements

Service containers must define a `CMD` or `ENTRYPOINT`. The pipeline will run `docker run` for the provided container without additional arguments.

Azure Pipelines can run Linux or Windows Containers. Use either hosted Ubuntu for Linux containers, or the Hosted Windows Container pool for Windows containers. (The Hosted macOS pool doesn't support running containers.)

YAML

Single container job

A simple example of using container jobs:

YAML

```yaml
resources:
  containers:
  - container: my_container
    image: buildpack-deps:focal
  - container: nginx
    image: nginx
```

```yaml
pool:
  vmImage: 'ubuntu-latest'

container: my_container
services:
  nginx: nginx

steps:
- script: |
    curl nginx
  displayName: Show that nginx is running
```

This pipeline fetches the `nginx` and `buildpack-deps` containers from Docker Hub and then starts the containers. The containers are networked together so that they can reach each other by their `services` name.

From inside this job container, the `nginx` host name resolves to the correct services using Docker networking. All containers on the network automatically expose all ports to each other.

Single job

You can also use service containers without a job container. A simple example:

YAML

```yaml
resources:
  containers:
  - container: nginx
    image: nginx
    ports:
    - 8080:80
    env:
      NGINX_PORT: 80
  - container: redis
    image: redis
    ports:
    - 6379

pool:
  vmImage: 'ubuntu-latest'

services:
  nginx: nginx
  redis: redis

steps:
- script: |
```

```
    curl localhost:8080
    echo $AGENT_SERVICES_REDIS_PORTS_6379
```

This pipeline starts the latest `nginx` containers. Since the job isn't running in a container, there's no automatic name resolution. This example shows how you can instead reach services by using `localhost`. In the above example, we provide the port explicitly (for example, `8080:80`).

An alternative approach is to let a random port get assigned dynamically at runtime. You can then access these dynamic ports by using variables. In a Bash script, you can access a variable by using the process environment. These variables take the form: `agent.services.<serviceName>.ports.<port>`. In the above example, `redis` is assigned a random available port on the host. The `agent.services.redis.ports.6379` variable contains the port number.

Multiple jobs

Service containers are also useful for running the same steps against multiple versions of the same service. In the following example, the same steps run against multiple versions of PostgreSQL.

YAML

```yaml
resources:
  containers:
  - container: my_container
    image: ubuntu:22.04
  - container: pg15
    image: postgres:15
  - container: pg14
    image: postgres:14

pool:
  vmImage: 'ubuntu-latest'

strategy:
  matrix:
    postgres15:
      postgresService: pg15
    postgres14:
      postgresService: pg14

container: my_container

services:
  postgres: $[ variables['postgresService'] ]
```

```yaml
steps:
- script: printenv
```

Ports

When specifying a container resource or an inline container, you can specify an array of `ports` to expose on the container.

YAML

```yaml
resources:
  containers:
  - container: my_service
    image: my_service:latest
    ports:
    - 8080:80
    - 5432

services:
  redis:
    image: redis
    ports:
    - 6379/tcp
```

Specifying `ports` is not required if your job is running in a container because containers on the same Docker network automatically expose all ports to each other by default.

If your job is running on the host, then `ports` are required to access the service. A port takes the form `<hostPort>:<containerPort>` or just `<containerPort>`, with an optional `/<protocol>` at the end, for example `6379/tcp` to expose `tcp` over port `6379`, bound to a random port on the host machine.

For ports bound to a random port on the host machine, the pipeline creates a variable of the form `agent.services.<serviceName>.ports.<port>` so that it can be accessed by the job. For example, `agent.services.redis.ports.6379` resolves to the randomly assigned port on the host machine.

Volumes

Volumes are useful for sharing data between services, or for persisting data between multiple runs of a job.

You can specify volume mounts as an array of `volumes`. Volumes can either be named Docker volumes, anonymous Docker volumes, or bind mounts on the host.

YAML

```yaml
services:
  my_service:
    image: myservice:latest
    volumes:
    - mydockervolume:/data/dir
    - /data/dir
    - /src/dir:/dst/dir
```

Volumes take the form `<source>:<destinationPath>`, where `<source>` can be a named volume or an absolute path on the host machine, and `<destinationPath>` is an absolute path in the container.

> **Note**
>
> If you use our hosted pools, then your volumes will not be persisted between jobs because the host machine is cleaned up after the job is completed.

Other options

Service containers share the same container resources as container jobs. This means that you can use the same additional options.

Healthcheck

Optionally, if any service container specifies a HEALTHCHECK, the agent waits until the container is healthy before running the job.

Multiple containers with services example

In this example, there's a Django Python web container connected to two database containers - PostgreSQL and MySQL. The PostgreSQL database is the primary database and its container has the name `db`. The `db` container uses volume `/data/db:/var/lib/postgresql/data` and there are three database variables passed to the container via `env`. The `mysql` container uses port `3306:3306` and there are also database variables passed via `env`. The `web` container is open with port `8000`. In the

steps, `pip` installs dependencies and then Django test are run. If you'd like to get a working example set up, you'll need a Django site set up with two databases. This example assumes your `manage.py` file is in the root directory and your Django project is within that directory. You may need to update the `/__w/1/s/` path in `/__w/1/s/manage.py test`.

YAML

```yaml
resources:
  containers:
    - container: db
      image: postgres
      volumes:
          - '/data/db:/var/lib/postgresql/data'
      env:
        POSTGRES_DB: postgres
        POSTGRES_USER: postgres
        POSTGRES_PASSWORD: postgres
    - container: mysql
      image: 'mysql:5.7'
      ports:
         - '3306:3306'
      env:
        MYSQL_DATABASE: users
        MYSQL_USER: mysql
        MYSQL_PASSWORD: mysql
        MYSQL_ROOT_PASSWORD: mysql
    - container: web
      image: python
      volumes:
      - '/code'
      ports:
        - '8000:8000'

pool:
  vmImage: 'ubuntu-latest'

container: web
services:
  db: db
  mysql: mysql

steps:
    - script: |
        pip install django
        pip install psycopg2
        pip install mysqlclient
      displayName: set up django
    - script: |
          python /__w/1/s/manage.py test
```

Run cross-platform scripts

Article • 11/02/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

With Azure Pipelines, you can run your builds on macOS, Linux, and Windows machines. If you develop on cross-platform technologies such as .NET Core, Node.js and Python, these capabilities bring both benefits and challenges.

For example, most pipelines include one or more scripts that you want to run during the build process. But scripts often don't run the same way on different platforms. You can use the `script` keyword shortcut to make writing scripts easier and also can use conditions to target specific platforms with your scripts.

Run cross-platform tools with a script step

The script keyword is a shortcut for the command line task. The `script` keyword runs Bash on Linux and macOS and cmd.exe on Windows.

Using `script` can be useful when your task just passes arguments to a cross-platform tool. For instance, calling `npm` with a set of arguments can be easily accomplished with a `script` step. `script` runs in each platform's native script interpreter: Bash on macOS and Linux, cmd.exe on Windows.

YAML

YAML

```yaml
steps:
- script: |
    npm install
    npm test
```

Handle environment variables

Environment variables throw the first wrinkle into writing cross-platform scripts. Command line, PowerShell, and Bash each have different ways of reading environment

variables. If you need to access an operating system-provided value like PATH, you'll need different techniques per platform.

However, Azure Pipelines offers a cross-platform way to refer to variables that it knows about called macro syntax. By surrounding a variable name in `$( )`, it's expanded before the platform's shell ever sees it. For instance, if you want to echo out the ID of the pipeline, the following script is cross-platform friendly:

YAML

YAML

```yaml
steps:
- script: echo This is pipeline $(System.DefinitionId)
```

This also works for variables you specify in the pipeline.

YAML

```yaml
variables:
  Example: 'myValue'

steps:
- script: echo The value passed in is $(Example)
```

Consider Bash or pwsh

If you have more complex scripting needs than the examples shown above, then consider writing them in Bash. Most macOS and Linux agents have Bash as an available shell, and Windows agents include Git Bash or Windows Subsystem for Linux Bash.

For Azure Pipelines, the Microsoft-hosted agents always have Bash available.

For example, if you need to make a decision about whether your build is triggered by a pull request:

YAML

YAML

```yaml
trigger:
    batch: true
    branches:
```

```
        include:
        - main
steps:
- bash: |
    echo "Hello world from $AGENT_NAME running on $AGENT_OS"
    case $BUILD_REASON in
            "Manual") echo "$BUILD_REQUESTEDFOR manually queued the
build." ;;
            "IndividualCI") echo "This is a CI build for
$BUILD_REQUESTEDFOR." ;;
            "BatchedCI") echo "This is a batched CI build for
$BUILD_REQUESTEDFOR." ;;
        *) $BUILD_REASON ;;
    esac
  displayName: Hello world
```

PowerShell Core (`pwsh`) is also an option. It requires each agent to have PowerShell Core installed.

Switch based on platform

In general, we recommend that you avoid platform-specific scripts to avoid problems such as duplication of your pipeline logic. Duplication causes extra work and extra risk of bugs. However, if there's no way to avoid platform-specific scripting, then you can use a `condition` to detect what platform you're on.

For example, suppose that for some reason you need the IP address of the build agent. On Windows, `ipconfig` gets that information. On macOS, it's `ifconfig`. And on Ubuntu Linux, it's `ip addr`.

Set up the below pipeline, then try running it against agents on different platforms.

YAML

YAML

```
steps:
# Linux
- bash: |
    export IPADDR=$(ip addr | grep 'state UP' -A2 | tail -n1 | awk
'{print $2}' | cut -f1  -d'/')
    echo "##vso[task.setvariable variable=IP_ADDR]$IPADDR"
  condition: eq( variables['Agent.OS'], 'Linux' )
  displayName: Get IP on Linux
# macOS
- bash: |
    export IPADDR=$(ifconfig | grep 'en0' -A3 | grep inet | tail -n1 |
```

```
awk '{print $2}')
    echo "##vso[task.setvariable variable=IP_ADDR]$IPADDR"
  condition: eq( variables['Agent.OS'], 'Darwin' )
  displayName: Get IP on macOS
# Windows
- powershell: |
    Set-Variable -Name IPADDR -Value ((Get-NetIPAddress | ?{
$_.AddressFamily -eq "IPv4" -and !($_.IPAddress -match "169") -and !
($_.IPaddress -match "127") } | Select-Object -First 1).IPAddress)
    Write-Host "##vso[task.setvariable variable=IP_ADDR]$IPADDR"
  condition: eq( variables['Agent.OS'], 'Windows_NT' )
  displayName: Get IP on Windows

# now we use the value, no matter where we got it
- script: |
    echo The IP address is $(IP_ADDR)
```

Use a PowerShell script to customize your pipeline

Article • 03/21/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

When you're ready to move beyond the basics of compiling and testing your code, use a PowerShell script to add your team's business logic to your build pipeline. You can run Windows PowerShell on a Windows build agent. PowerShell Core runs on any platform.

Add a PowerShell script

YAML

The syntax for including PowerShell Core is slightly different from the syntax for Windows PowerShell.

1. Push your PowerShell script to your repo.
2. Add a `pwsh` or `powershell` step. The `pwsh` keyword is a shortcut for the PowerShell task for PowerShell Core. The `powershell` keyword is another shortcut for the PowerShell task.

YAML

```yaml
# for PowerShell Core
steps:
- pwsh: ./my-script.ps1

# for Windows PowerShell
steps:
- powershell: .\my-script.ps1
```

Example PowerShell script: version assemblies

Here's an example script to version your assemblies. For the script to run successfully, you'll need to update your build number to use a format with four periods (example: `$(BuildDefinitionName)_$(Year:yyyy).$(Month).$(DayOfMonth)$(Rev:.r)`). Build number can also be referred to as run number.

YAML

You can customize your build number within a YAML pipeline with the `name` property.

YAML

```yaml
name:
$(BuildDefinitionName)_$(Year:yyyy).$(Month).$(DayOfMonth)$(Rev:.r)

pool:
  vmImage: windows-latest

steps:
- pwsh: echo $(Build.BuildNumber) //output updated build number
```

ps

```ps
# If found use it to version the assemblies.
#
# For example, if the 'Build number format' build pipeline parameter
# $(BuildDefinitionName)_$(Year:yyyy).$(Month).$(DayOfMonth)$(Rev:.r)
# then your build numbers come out like this:
# "Build HelloWorld_2013.07.19.1"
# This script would then apply version 2013.07.19.1 to your assemblies.

# Enable -Verbose option
[CmdletBinding()]

# Regular expression pattern to find the version in the build number
# and then apply it to the assemblies
$VersionRegex = "\d+\.\d+\.\d+\.\d+"

# If this script is not running on a build server, remind user to
# set environment variables so that this script can be debugged
if(-not ($Env:BUILD_SOURCESDIRECTORY -and $Env:BUILD_BUILDNUMBER))
{
    Write-Error "You must set the following environment variables"
    Write-Error "to test this script interactively."
    Write-Host '$Env:BUILD_SOURCESDIRECTORY - For example, enter something
like:'
    Write-Host '$Env:BUILD_SOURCESDIRECTORY =
"C:\code\FabrikamTFVC\HelloWorld"'
    Write-Host '$Env:BUILD_BUILDNUMBER - For example, enter something like:'
    Write-Host '$Env:BUILD_BUILDNUMBER = "Build HelloWorld_0000.00.00.0"'
    exit 1
}

# Make sure path to source code directory is available
if (-not $Env:BUILD_SOURCESDIRECTORY)
```

```
{
    Write-Error ("BUILD_SOURCESDIRECTORY environment variable is missing.")
    exit 1
}
elseif (-not (Test-Path $Env:BUILD_SOURCESDIRECTORY))
{
    Write-Error "BUILD_SOURCESDIRECTORY does not exist:
$Env:BUILD_SOURCESDIRECTORY"
    exit 1
}
Write-Verbose "BUILD_SOURCESDIRECTORY: $Env:BUILD_SOURCESDIRECTORY"

# Make sure there is a build number
if (-not $Env:BUILD_BUILDNUMBER)
{
    Write-Error ("BUILD_BUILDNUMBER environment variable is missing.")
    exit 1
}
Write-Verbose "BUILD_BUILDNUMBER: $Env:BUILD_BUILDNUMBER"

# Get and validate the version data
$VersionData = [regex]::matches($Env:BUILD_BUILDNUMBER,$VersionRegex)
switch($VersionData.Count)
{
   0
      {
         Write-Error "Could not find version number data in
BUILD_BUILDNUMBER."
         exit 1
      }
   1 {}
   default
      {
         Write-Warning "Found more than instance of version data in
BUILD_BUILDNUMBER."
         Write-Warning "Will assume first instance is version."
      }
}
$NewVersion = $VersionData[0]
Write-Verbose "Version: $NewVersion"

# Apply the version to the assembly property files
$files = gci $Env:BUILD_SOURCESDIRECTORY -recurse -include
"*Properties*","My Project" |
    ?{ $_.PSIsContainer } |
    foreach { gci -Path $_.FullName -Recurse -include AssemblyInfo.* }
if($files)
{
    Write-Verbose "Will apply $NewVersion to $($files.count) files."

    foreach ($file in $files) {
        $filecontent = Get-Content($file)
        attrib $file -r
        $filecontent -replace $VersionRegex, $NewVersion | Out-File $file
        Write-Verbose "$file.FullName - version applied"
```

```
    }
}
else
{
    Write-Warning "Found no files."
}
```

Example PowerShell script: access REST API

In this example, you'll use the `SYSTEM_ACCESSTOKEN` variable to access the Azure Pipelines REST API.

YAML

You can use `$env:SYSTEM_ACCESSTOKEN` in your script in a YAML pipeline to access the OAuth token.

YAML

```
- task: PowerShell@2
  inputs:
    targetType: 'inline'
    script: |
      $url =
"$($env:SYSTEM_TEAMFOUNDATIONCOLLECTIONURI)$env:SYSTEM_TEAMPROJECTID/_ap
is/build/definitions/$($env:SYSTEM_DEFINITIONID)?api-version=5.0"
              Write-Host "URL: $url"
              $pipeline = Invoke-RestMethod -Uri $url -Headers @{
                  Authorization = "Bearer $env:SYSTEM_ACCESSTOKEN"
              }
              Write-Host "Pipeline = $($pipeline | ConvertTo-Json -Depth
100)"
  env:
     SYSTEM_ACCESSTOKEN: $(System.AccessToken)
```

FAQ

What variables are available for me to use in my scripts?

You can use predefined variables in your scripts. For more information on available variables and how to use them, see Use predefined variables.

How do I set a variable so that it can be read by subsequent scripts and tasks?

To learn more about defining build variables in a script, see Define and modify your build variables in a script.

To learn more about defining release variables in a script, see Define and modify your release variables in a script

Which branch of the script does the build run?

The build will use the active branch of your code. If your pipeline run uses the `main` branch, your script will also use the `main` branch.

What kinds of parameters can I use?

You can use named parameters. Other kinds of parameters, such as switch parameters, aren't supported. You'll see errors if you try to use switch parameters.

Run Git commands in a script

Article • 07/03/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

For some workflows, you need your build pipeline to run Git commands. For example, after a CI build on a feature branch is done, the team might want to merge the branch to main.

Git is available on Microsoft-hosted agents and on on-premises agents.

Enable scripts to run Git commands

> **Note**
>
> Before you begin, be sure your account's default identity is set with the following code. This must be done as the very first step after checking out your code.
>
> ```
> git config --global user.email "you@example.com"
> git config --global user.name "Your Name"
> ```

Grant version control permissions to the build service

1. Go to the project settings page for your organization at **Organization Settings** > **General** > **Projects**.

2. Select the project you want to edit.

3. Within **Project Settings**, select **Repositories**. Select the repository you want to run Git commands on.

4. Select **Security** to edit your repository security.

5. Search for **Project Collection Build Service**. Choose the identity **{{your project name}} Build Service ({your organization})** (not the group **Project Collection Build Service Accounts ({your organization})**). By default, this identity can read from the repo but can't push any changes back to it. Grant permissions needed for the Git commands you want to run. Typically you'll want to grant:

   - **Create branch:** Allow
   - **Contribute:** Allow
   - **Read:** Allow
   - **Create tag:** Allow

Allow scripts to access the system token

YAML

Add a `checkout` section with `persistCredentials` set to `true`.

YAML

```yaml
steps:
- checkout: self
  persistCredentials: true
```

Learn more about checkout.

Make sure to clean up the local repo

Certain kinds of changes to the local repository aren't automatically cleaned up by the build pipeline. So make sure to:

- Delete local branches you create.
- Undo git config changes.

If you run into problems using an on-premises agent, make sure the repo is clean:

YAML

Make sure `checkout` has `clean` set to `true`.

YAML

```yaml
steps:
- checkout: self
  clean: true
```

Examples

List the files in your repo

On the build tab, add this task:

Task	Arguments
Utility: Command Line List the files in the Git repo.	Tool: `git` Arguments: `ls-files`

Merge a feature branch to main

You want a CI build to merge to main if the build succeeds.

On the Triggers tab, select **Continuous integration (CI)** and include the branches you want to build.

Create `merge.bat` at the root of your repo:

bat

```bat
@echo off
ECHO SOURCE BRANCH IS %BUILD_SOURCEBRANCH%
IF %BUILD_SOURCEBRANCH% == refs/heads/main (
   ECHO Building main branch so no merge is needed.
   EXIT
)
SET sourceBranch=origin/%BUILD_SOURCEBRANCH:refs/heads/=%
ECHO GIT CHECKOUT MAIN
git checkout main
ECHO GIT STATUS
git status
ECHO GIT MERGE
git merge %sourceBranch% -m "Merge to main"
ECHO GIT STATUS
git status
ECHO GIT PUSH
git push origin
ECHO GIT STATUS
git status
```

On the build tab add this as the last task:

Task	Arguments
Utility: Batch Script Run merge.bat.	Path: `merge.bat`

FAQ

Can I run Git commands if my remote repo is in GitHub or another Git service such as Bitbucket Cloud?

Yes

Which tasks can I use to run Git commands?

Batch Script

Command Line

PowerShell

Shell Script

How do I avoid triggering a CI build when the script pushes?

Add `[skip ci]` to your commit message or description. Here are examples:

- `git commit -m "This is a commit message [skip ci]"`
- `git merge origin/features/hello-world -m "Merge to main [skip ci]"`

You can also use any of these variations for commits to Azure Repos Git, Bitbucket Cloud, GitHub, and GitHub Enterprise Server.

- `[skip ci]` or `[ci skip]`
- `skip-checks: true` or `skip-checks:true`
- `[skip azurepipelines]` or `[azurepipelines skip]`
- `[skip azpipelines]` or `[azpipelines skip]`
- `[skip azp]` or `[azp skip]`
- `***NO_CI***`

Do I need an agent?

You need at least one agent to run your build or release.

I'm having problems. How can I troubleshoot them?

See Troubleshoot Build and Release.

I can't select a default agent pool and I can't queue my build or release. How do I fix this?

See Agent pools.

My NuGet push task is failing with the following error: "Error: unable to get local issuer certificate". How can I fix this?

This can be fixed by adding a trusted root certificate. You can either add the `NODE_EXTRA_CA_CERTS=file` environment variable to your build agent, or you can add the `NODE.EXTRA.CA.CERTS=file` task variable in your pipeline. See Node.js documentation

for more details about this variable. See Set variables in a pipeline for instructions on setting a variable in your pipeline.

Pipeline caching

Article • 03/22/2023

Azure DevOps Services

Pipeline caching can help reduce build time by allowing the outputs or downloaded dependencies from one run to be reused in later runs, thereby reducing or avoiding the cost to recreate or redownload the same files again. Caching is especially useful in scenarios where the same dependencies are downloaded over and over at the start of each run. This is often a time consuming process involving hundreds or thousands of network calls.

Caching can be effective at improving build time provided the time to restore and save the cache is less than the time to produce the output again from scratch. Because of this, caching may not be effective in all scenarios and may actually have a negative impact on build time.

Caching is currently supported in CI and deployment jobs, but not classic release jobs.

When to use artifacts versus caching

Pipeline caching and pipeline artifacts perform similar functions but are designed for different scenarios and shouldn't be used interchangeably.

- **Use pipeline artifacts** when you need to take specific files produced in one job and share them with other jobs (and these other jobs will likely fail without them).
- **Use pipeline caching** when you want to improve build time by reusing files from previous runs (and not having these files won't impact the job's ability to run).

> ⓘ **Note**
>
> Pipeline caching and pipeline artifacts are free for all tiers (free and paid). see **Artifacts storage consumption** for more details.

Cache task: how it works

Caching is added to a pipeline using the Cache task. This task works like any other task and is added to the `steps` section of a job.

When a cache step is encountered during a run, the task restores the cache based on the provided inputs. If no cache is found, the step completes and the next step in the job is run.

After all steps in the job have run and assuming a **successful** job status, a special **"Post-job: Cache"** step is automatically added and triggered for each **"restore cache"** step that wasn't skipped. This step is responsible for **saving the cache**.

> ⓘ **Note**
>
> Caches are immutable, meaning that once a cache is created, its contents cannot be changed.

Configure the Cache task

The Cache task has two required arguments: *key* and *path*:

- **path**: the path of the folder to cache. Can be an absolute or a relative path. Relative paths are resolved against `$(System.DefaultWorkingDirectory)`.

> ⓘ **Note**
>
> You can use **predefined variables** to store the path to the folder you want to cache, however wildcards are not supported.

- **key**: should be set to the identifier for the cache you want to restore or save. Keys are composed of a combination of string values, file paths, or file patterns, where each segment is separated by a `|` character.

- **Strings**:
  Fixed value (like the name of the cache or a tool name) or taken from an environment variable (like the current OS or current job name)

- **File paths**:
  Path to a specific file whose contents will be hashed. This file must exist at the time the task is run. Keep in mind that *any* key segment that "looks like a file path" will be treated like a file path. In particular, this includes segments containing a `.`. This could result in the task failing when this "file" doesn't exist.

  > 💡 **Tip**

> To avoid a path-like string segment from being treated like a file path, wrap it with double quotes, for example: `"my.key" | $(Agent.OS) | key.file`

- **File patterns**:
  Comma-separated list of glob-style wildcard pattern that must match at least one file. For example:
  - `**/yarn.lock`: all yarn.lock files under the sources directory
  - `*/asset.json, !bin/**`: all asset.json files located in a directory under the sources directory, except under the bin directory

The contents of any file identified by a file path or file pattern is hashed to produce a dynamic cache key. This is useful when your project has file(s) that uniquely identify what is being cached. For example, files like `package-lock.json`, `yarn.lock`, `Gemfile.lock`, or `Pipfile.lock` are commonly referenced in a cache key since they all represent a unique set of dependencies.

Relative file paths or file patterns are resolved against `$(System.DefaultWorkingDirectory)`.

Example:

Here's an example showing how to cache dependencies installed by Yarn:

YAML

```yaml
variables:
  YARN_CACHE_FOLDER: $(Pipeline.Workspace)/.yarn

steps:
- task: Cache@2
  inputs:
    key: '"yarn" | "$(Agent.OS)" | yarn.lock'
    restoreKeys: |
       "yarn" | "$(Agent.OS)"
       "yarn"
    path: $(YARN_CACHE_FOLDER)
  displayName: Cache Yarn packages

- script: yarn --frozen-lockfile
```

In this example, the cache key contains three parts: a static string ("yarn"), the OS the job is running on since this cache is unique per operating system, and the hash of the `yarn.lock` file that uniquely identifies the set of dependencies in the cache.

On the first run after the task is added, the cache step will report a "cache miss" since the cache identified by this key doesn't exist. After the last step, a cache will be created from the files in `$(Pipeline.Workspace)/.yarn` and uploaded. On the next run, the cache step will report a "cache hit" and the contents of the cache will be downloaded and restored.

> **Note**
>
> `Pipeline.Workspace` is the local path on the agent running your pipeline where all directories are created. This variable has the same value as `Agent.BuildDirectory`.

Restore keys

`restoreKeys` can be used if one wants to query against multiple exact keys or key prefixes. This is used to fall back to another key in the case that a `key` doesn't yield a hit. A restore key will search for a key by prefix and yield the latest created cache entry as a result. This is useful if the pipeline is unable to find an exact match but wants to use a partial cache hit instead. To insert multiple restore keys, simply delimit them by using a new line to indicate the restore key (see the example for more details). The order of which restore keys will be tried against will be from top to bottom.

Required software on self-hosted agent

Archive software / Platform	Windows	Linux	Mac
GNU Tar	Required	Required	No
BSD Tar	No	No	Required
7-Zip	Recommended	No	No

The above executables need to be in a folder listed in the PATH environment variable. Keep in mind that the hosted agents come with the software included, this is only applicable for self-hosted agents.

Example:

Here's an example of how to use restore keys by Yarn:

YAML

```yaml
variables:
  YARN_CACHE_FOLDER: $(Pipeline.Workspace)/.yarn
```

```yaml
steps:
- task: Cache@2
  inputs:
    key: '"yarn" | "$(Agent.OS)" | yarn.lock'
    restoreKeys: |
       yarn | "$(Agent.OS)"
       yarn
    path: $(YARN_CACHE_FOLDER)
  displayName: Cache Yarn packages

- script: yarn --frozen-lockfile
```

In this example, the cache task attempts to find if the key exists in the cache. If the key doesn't exist in the cache, it tries to use the first restore key `yarn | $(Agent.OS)`. This will attempt to search for all keys that either exactly match that key or has that key as a prefix. A prefix hit can happen if there was a different `yarn.lock` hash segment. For example, if the following key `yarn | $(Agent.OS) | old-yarn.lock` was in the cache where the old `yarn.lock` yielded a different hash than `yarn.lock`, the restore key will yield a partial hit. If there's a miss on the first restore key, it will then use the next restore key `yarn` which will try to find any key that starts with `yarn`. For prefix hits, the result will yield the most recently created cache key as the result.

> ⓘ **Note**
>
> A pipeline can have one or more caching task(s). There is no limit on the caching storage capacity, and jobs and tasks from the same pipeline can access and share the same cache.

Cache isolation and security

To ensure isolation between caches from different pipelines and different branches, every cache belongs to a logical container called a scope. Scopes provide a security boundary that ensures a job from one pipeline cannot access the caches from a different pipeline, and a job building a PR has read access to the caches for the PR's target branch (for the same pipeline), but cannot write (create) caches in the target branch's scope.

When a cache step is encountered during a run, the cache identified by the key is requested from the server. The server then looks for a cache with this key from the scopes visible to the job, and returns the cache (if available). On cache save (at the end

of the job), a cache is written to the scope representing the pipeline and branch. See below for more details.

CI, manual, and scheduled runs

Scope	Read	Write
Source branch	Yes	Yes
main branch (default branch)	Yes	No

Pull request runs

Scope	Read	Write
Source branch	Yes	No
Target branch	Yes	No
Intermediate branch (such as `refs/pull/1/merge`)	Yes	Yes
main branch (default branch)	Yes	No

Pull request fork runs

Branch	Read	Write
Target branch	Yes	No
Intermediate branch (such as `refs/pull/1/merge`)	Yes	Yes
main branch (default branch)	Yes	No

> **Tip**
>
> Because caches are already scoped to a project, pipeline, and branch, there is no need to include any project, pipeline, or branch identifiers in the cache key.

Conditioning on cache restoration

In some scenarios, the successful restoration of the cache should cause a different set of steps to be run. For example, a step that installs dependencies can be skipped if the

cache was restored. This is possible using the `cacheHitVar` task input. Setting this input to the name of an environment variable will cause the variable to be set to `true` when there's a cache hit, `inexact` on a restore key cache hit, otherwise it will be set to `false`. This variable can then be referenced in a step condition or from within a script.

In the following example, the `install-deps.sh` step is skipped when the cache is restored:

YAML

```yaml
steps:
- task: Cache@2
  inputs:
    key: mykey | mylockfile
    restoreKeys: mykey
    path: $(Pipeline.Workspace)/mycache
    cacheHitVar: CACHE_RESTORED

- script: install-deps.sh
  condition: ne(variables.CACHE_RESTORED, 'true')

- script: build.sh
```

Bundler

For Ruby projects using Bundler, override the `BUNDLE_PATH` environment variable used by Bundler to set the path Bundler will look for Gems in.

Example:

YAML

```yaml
variables:
  BUNDLE_PATH: $(Pipeline.Workspace)/.bundle

steps:
- task: Cache@2
  displayName: Bundler caching
  inputs:
    key: 'gems | "$(Agent.OS)" | Gemfile.lock'
    restoreKeys: |
      gems | "$(Agent.OS)"
      gems
    path: $(BUNDLE_PATH)
```

Ccache (C/C++)

Ccache is a compiler cache for C/C++. To use Ccache in your pipeline make sure `Ccache` is installed, and optionally added to your `PATH` (see Ccache run modes). Set the `CCACHE_DIR` environment variable to a path under `$(Pipeline.Workspace)` and cache this directory.

Example:

YAML

```yaml
variables:
  CCACHE_DIR: $(Pipeline.Workspace)/ccache

steps:
- bash: |
    sudo apt-get install ccache -y
    echo "##vso[task.prependpath]/usr/lib/ccache"
  displayName: Install ccache and update PATH to use linked versions of gcc,
cc, etc

- task: Cache@2
  inputs:
    key: 'ccache | "$(Agent.OS)"'
    path: $(CCACHE_DIR)
    restoreKeys: |
      ccache | "$(Agent.OS)"
  displayName: ccache
```

See Ccache configuration settings for more details.

Docker images

Caching Docker images dramatically reduces the time it takes to run your pipeline.

YAML

```yaml
variables:
  repository: 'myDockerImage'
  dockerfilePath: '$(Build.SourcesDirectory)/app/Dockerfile'
  tag: '$(Build.BuildId)'

pool:
  vmImage: 'ubuntu-latest'
steps:
  - task: Cache@2
    displayName: Cache task
    inputs:
      key: 'docker | "$(Agent.OS)" | cache'
      path: $(Pipeline.Workspace)/docker
      cacheHitVar: CACHE_RESTORED                #Variable to set to 'true'
```

```
when the cache is restored

  - script: |
      docker load -i $(Pipeline.Workspace)/docker/cache.tar
    displayName: Docker restore
    condition: and(not(canceled()), eq(variables.CACHE_RESTORED, 'true'))

  - task: Docker@2
    displayName: 'Build Docker'
    inputs:
      command: 'build'
      repository: '$(repository)'
      dockerfile: '$(dockerfilePath)'
      tags: |
        '$(tag)'

  - script: |
      mkdir -p $(Pipeline.Workspace)/docker
      docker save -o $(Pipeline.Workspace)/docker/cache.tar
$(repository):$(tag)
    displayName: Docker save
    condition: and(not(canceled()), not(failed()),
ne(variables.CACHE_RESTORED, 'true'))
```

- **key**: (required) - a unique identifier for the cache.
- **path**: (required) - path of the folder or file that you want to cache.

Golang

For Golang projects, you can specify the packages to be downloaded in the *go.mod* file. If your `GOCACHE` variable isn't already set, set it to where you want the cache to be downloaded.

Example:

YAML

```yaml
variables:
  GO_CACHE_DIR: $(Pipeline.Workspace)/.cache/go-build/

steps:
- task: Cache@2
  inputs:
    key: 'go | "$(Agent.OS)" | go.mod'
    restoreKeys: |
      go | "$(Agent.OS)"
    path: $(GO_CACHE_DIR)
  displayName: Cache GO packages
```

Gradle

Using Gradle's built-in caching support can have a significant impact on build time. To enable the build cache, set the `GRADLE_USER_HOME` environment variable to a path under `$(Pipeline.Workspace)` and either run your build with `--build-cache` or add `org.gradle.caching=true` to your `gradle.properties` file.

Example:

YAML

```yaml
variables:
  GRADLE_USER_HOME: $(Pipeline.Workspace)/.gradle

steps:
- task: Cache@2
  inputs:
    key: 'gradle | "$(Agent.OS)" | **/build.gradle.kts' # Swap
build.gradle.kts for build.gradle when using Groovy
    restoreKeys: |
      gradle | "$(Agent.OS)"
      gradle
    path: $(GRADLE_USER_HOME)
  displayName: Configure gradle caching

- task: Gradle@2
  inputs:
    gradleWrapperFile: 'gradlew'
    tasks: 'build'
    options: '--build-cache'
  displayName: Build

- script: |
    # stop the Gradle daemon to ensure no files are left open (impacting the
save cache operation later)
    ./gradlew --stop
  displayName: Gradlew stop
```

- **restoreKeys**: The fallback keys if the primary key fails (Optional)

> **Note**
>
> Caches are immutable, once a cache with a particular key is created for a specific scope (branch), the cache cannot be updated. This means that if the key is a fixed value, all subsequent builds for the same branch will not be able to update the cache even if the cache's contents have changed. If you want to use a fixed key value, you must use the `restoreKeys` argument as a fallback option.

Maven

Maven has a local repository where it stores downloads and built artifacts. To enable, set the `maven.repo.local` option to a path under `$(Pipeline.Workspace)` and cache this folder.

Example:

YAML

```yaml
variables:
  MAVEN_CACHE_FOLDER: $(Pipeline.Workspace)/.m2/repository
  MAVEN_OPTS: '-Dmaven.repo.local=$(MAVEN_CACHE_FOLDER)'

steps:
- task: Cache@2
  inputs:
    key: 'maven | "$(Agent.OS)" | **/pom.xml'
    restoreKeys: |
      maven | "$(Agent.OS)"
      maven
    path: $(MAVEN_CACHE_FOLDER)
  displayName: Cache Maven local repo

- script: mvn install -B -e
```

If you're using a Maven task, make sure to also pass the `MAVEN_OPTS` variable because it gets overwritten otherwise:

YAML

```yaml
- task: Maven@4
  inputs:
    mavenPomFile: 'pom.xml'
    mavenOptions: '-Xmx3072m $(MAVEN_OPTS)'
```

.NET/NuGet

If you use `PackageReferences` to manage NuGet dependencies directly within your project file and have a `packages.lock.json` file, you can enable caching by setting the `NUGET_PACKAGES` environment variable to a path under `$(UserProfile)` and caching this directory. See Package reference in project files for more details on how to lock dependencies. If you want to use multiple packages.lock.json, you can still use the following example without making any changes. The content of all the

packages.lock.json files will be hashed and if one of the files is changed, a new cache key will be generated.

Example:

YAML

```yaml
variables:
  NUGET_PACKAGES: $(Pipeline.Workspace)/.nuget/packages

steps:
- task: Cache@2
  inputs:
    key: 'nuget | "$(Agent.OS)" |
$(Build.SourcesDirectory)/**/packages.lock.json'
    restoreKeys: |
       nuget | "$(Agent.OS)"
       nuget
    path: $(NUGET_PACKAGES)
  displayName: Cache NuGet packages
```

Node.js/npm

There are different ways to enable caching in a Node.js project, but the recommended way is to cache npm's shared cache directory. This directory is managed by npm and contains a cached version of all downloaded modules. During install, npm checks this directory first (by default) for modules that can reduce or eliminate network calls to the public npm registry or to a private registry.

Because the default path to npm's shared cache directory is not the same across all platforms, it's recommended to override the `npm_config_cache` environment variable to a path under `$(Pipeline.Workspace)`. This also ensures the cache is accessible from container and non-container jobs.

Example:

YAML

```yaml
variables:
  npm_config_cache: $(Pipeline.Workspace)/.npm

steps:
- task: Cache@2
  inputs:
    key: 'npm | "$(Agent.OS)" | package-lock.json'
    restoreKeys: |
       npm | "$(Agent.OS)"
```

```
    path: $(npm_config_cache)
  displayName: Cache npm

- script: npm ci
```

If your project doesn't have a `package-lock.json` file, reference the `package.json` file in the cache key input instead.

> **Tip**
>
> Because `npm ci` deletes the `node_modules` folder to ensure that a consistent, repeatable set of modules is used, you should avoid caching `node_modules` when calling `npm ci`.

Node.js/Yarn

Like with npm, there are different ways to cache packages installed with Yarn. The recommended way is to cache Yarn's shared cache folder. This directory is managed by Yarn and contains a cached version of all downloaded packages. During install, Yarn checks this directory first (by default) for modules, which can reduce or eliminate network calls to public or private registries.

Example:

YAML

```yaml
variables:
  YARN_CACHE_FOLDER: $(Pipeline.Workspace)/.yarn

steps:
- task: Cache@2
  inputs:
    key: 'yarn | "$(Agent.OS)" | yarn.lock'
    restoreKeys: |
       yarn | "$(Agent.OS)"
       yarn
    path: $(YARN_CACHE_FOLDER)
  displayName: Cache Yarn packages

- script: yarn --frozen-lockfile
```

Python/Anaconda

Set up your pipeline caching with Anaconda environments:

Example

YAML

```yaml
variables:
  CONDA_CACHE_DIR: /usr/share/miniconda/envs

# Add conda to system path
steps:
- script: echo "##vso[task.prependpath]$CONDA/bin"
  displayName: Add conda to PATH

- bash: |
    sudo chown -R $(whoami):$(id -ng) $(CONDA_CACHE_DIR)
  displayName: Fix CONDA_CACHE_DIR directory permissions

- task: Cache@2
  displayName: Use cached Anaconda environment
  inputs:
    key: 'conda | "$(Agent.OS)" | environment.yml'
    restoreKeys: |
      python | "$(Agent.OS)"
      python
    path: $(CONDA_CACHE_DIR)
    cacheHitVar: CONDA_CACHE_RESTORED

- script: conda env create --quiet --file environment.yml
  displayName: Create Anaconda environment
  condition: eq(variables.CONDA_CACHE_RESTORED, 'false')
```

- **Windows**

  YAML

  ```yaml
  - task: Cache@2
    displayName: Cache Anaconda
    inputs:
      key: 'conda | "$(Agent.OS)" | environment.yml'
      restoreKeys: |
        python | "$(Agent.OS)"
        python
      path: $(CONDA)/envs
      cacheHitVar: CONDA_CACHE_RESTORED

  - script: conda env create --quiet --file environment.yml
    displayName: Create environment
    condition: eq(variables.CONDA_CACHE_RESTORED, 'false')
  ```

PHP/Composer

For PHP projects using Composer, override the `COMPOSER_CACHE_DIR` environment variable used by Composer.

Example:

```yaml
variables:
  COMPOSER_CACHE_DIR: $(Pipeline.Workspace)/.composer

steps:
- task: Cache@2
  inputs:
    key: 'composer | "$(Agent.OS)" | composer.lock'
    restoreKeys: |
      composer | "$(Agent.OS)"
      composer
    path: $(COMPOSER_CACHE_DIR)
  displayName: Cache composer

- script: composer install
```

Known issues and feedback

If you're experiencing issues setting up caching for your pipeline, check the list of open issues in the microsoft/azure-pipelines-tasks repo. If you don't see your issue listed, create a new one and provide the necessary information about your scenario.

Q&A

Q: Can I clear a cache?

A: Clearing a cache is currently not supported. However you can add a string literal (such as `version2`) to your existing cache key to change the key in a way that avoids any hits on existing caches. For example, change the following cache key from this:

```yaml
key: 'yarn | "$(Agent.OS)" | yarn.lock'
```

to this:

```yaml
key: 'version2 | yarn | "$(Agent.OS)" | yarn.lock'
```

Q: When does a cache expire?

A: Caches expire after seven days of no activity.

Q: When does the cache get uploaded?

A: After the last step fo your pipeline a cache will be created from your cache `path` and uploaded. See the example for more details.

Q: Is there a limit on the size of a cache?

A: There's no enforced limit on the size of individual caches or the total size of all caches in an organization.

Cache NuGet packages

Article • 10/06/2023

Azure DevOps Services

With pipeline caching, you can reduce your build time by caching your dependencies to be reused in later runs. In this article, you'll learn how to use the Cache task to cache and restore your NuGet packages.

Lock dependencies

To set up the cache task, we must first lock our project's dependencies and create a **package.lock.json** file. We'll use the hash of the content of this file to generate a unique key for our cache.

To lock your project's dependencies, set the **RestorePackagesWithLockFile** property in your *csproj* file to **true**. NuGet restore generates a lock file **packages.lock.json** at the root directory of your project. Make sure you check your **packages.lock.json** file into your source code.

XML

```
<PropertyGroup>
  <RestorePackagesWithLockFile>true</RestorePackagesWithLockFile>
</PropertyGroup>
```

Cache NuGet packages

We'll need to create a pipeline variable to point to the location of our packages on the agent running the pipeline.

In this example, the content of the **packages.lock.json** will be hashed to produce a dynamic cache key. This ensures that every time the file is modified, a new cache key is generated.

YAML

```yaml
variables:
  NUGET_PACKAGES: $(Pipeline.Workspace)/.nuget/packages

- task: Cache@2
  displayName: Cache
  inputs:
    key: 'nuget | "$(Agent.OS)" |
**/packages.lock.json,!**/bin/**,!**/obj/**'
    restoreKeys: |
       nuget | "$(Agent.OS)"
       nuget
    path: '$(NUGET_PACKAGES)'
    cacheHitVar: 'CACHE_RESTORED'
```

> ⓘ **Note**
>
> Caches are immutable, once a cache is created, its contents cannot be modified.

Restore cache

This task will only run if the `CACHE_RESTORED` variable is false.

YAML

```yaml
- task: NuGetCommand@2
  condition: ne(variables.CACHE_RESTORED, true)
  inputs:
    command: 'restore'
    restoreSolution: '**/*.sln'
```

If you encounter the error message "project.assets.json not found" during your build task, you can resolve it by removing the condition `condition: ne(variables.CACHE_RESTORED, true)` from your restore task. By doing so, the restore

command will be executed, generating your project.assets.json file. The restore task will not download packages that are already present in your corresponding folder.

> **Note**
>
> A pipeline can contain one or more caching tasks, and jobs and tasks within the same pipeline can access and share the same cache.

Performance comparison

Pipeline caching is a great way to speed up your pipeline execution. Here's a side-by-side performance comparison for two different pipelines. Before adding the caching task (right), the restore task took approximately 41 seconds. We added the caching task to a second pipeline (left) and configured the restore task to run when a cache miss is encountered. The restore task in this case took 8 seconds to complete.

Below is the full YAML pipeline for reference:

YAML

```yaml
pool:
  vmImage: 'windows-latest'

variables:
  solution: '**/*.sln'
  buildPlatform: 'Any CPU'
  buildConfiguration: 'Release'
  NUGET_PACKAGES: $(Pipeline.Workspace)/.nuget/packages
```

```yaml
steps:
- task: NuGetToolInstaller@1
  displayName: 'NuGet tool installer'

- task: Cache@2
  displayName: 'NuGet Cache'
  inputs:
    key: 'nuget | "$(Agent.OS)" |
**/packages.lock.json,!**/bin/**,!**/obj/**'
    restoreKeys: |
       nuget | "$(Agent.OS)"
       nuget
    path: '$(NUGET_PACKAGES)'
    cacheHitVar: 'CACHE_RESTORED'

- task: NuGetCommand@2
  displayName: 'NuGet restore'
  condition: ne(variables.CACHE_RESTORED, true)
  inputs:
    command: 'restore'
    restoreSolution: '$(solution)'

- task: VSBuild@1
  displayName: 'Visual Studio Build'
  inputs:
    solution: '$(solution)'
    platform: '$(buildPlatform)'
    configuration: '$(buildConfiguration)'
```

Related articles

- Pipeline caching
- Deploy from multiple branches
- Deploy pull request Artifacts

Configure run or build numbers

Article • 03/21/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

You can customize how your pipeline runs are numbered. The default value for run number is `$(Date:yyyyMMdd).$(Rev:r)`.

In Azure DevOps `$(Rev:r)` is a special variable format that only works in the build number field. When a build is completed, if nothing else in the build number has changed, the `Rev` integer value increases by one.

`$(Rev:r)` resets when you change part of the build number. For example, if you've configured your build number format as `$(BuildDefinitionName)_$(Date:yyyyMMdd)$(Rev:r)`, then the build number will reset when the date changes the next day. If your build number is `MyBuild_20230621.1`, the next build number that day is `MyBuild_20230621.2`. The next day, the build number is `MyBuild_20230622.1`.

If your build number format is `1.0.$(Rev:r)`, then the build number resets to `1.0.1` when you change part of the number. For example, if your last build number was `1.0.3`, and you change the build number to `1.1.$(Rev:r)` to indicate a version change, the next build number is `1.1.1`.

YAML

In YAML, this property is called `name` and must be at the root level of a pipeline. If not specified, your run is given a unique integer as its name. You can give runs much more useful names that are meaningful to your team. You can use a combination of tokens, variables, and underscore characters. The `name` property doesn't work in template files.

YAML

```yaml
name:
$(TeamProject)_$(Build.DefinitionName)_$(SourceBranchName)_$(Date:yyyyMM
dd)$(Rev:r)

steps:
  - script: echo '$(Build.BuildNumber)' # outputs customized build
number like project_def_master_20200828.1
```

Example

At the time, a run is started:

- Project name: Fabrikam
- Pipeline name: CIBuild
- Branch: main
- Build ID/Run ID: 752
- Date: May 5, 2019.
- Time: 9:07:03 PM.
- One run completed earlier today.

If you specify this build number format:

```
$(TeamProject)_$(Build.DefinitionName)_$(SourceBranchName)_$(Date:yyyyMMdd)$
(Rev:.r)
```

Then the second run on this day would be named:
Fabrikam_CIBuild_master_20190505.2

Tokens

The following table shows how each token is resolved based on the previous example. You can use these tokens only to define a run number; they don't work anywhere else in your pipeline.

Token	Example replacement value
`$(Build.DefinitionName)`	CIBuild Note: The pipeline name must not contain invalid or whitespace characters.

Token	Example replacement value
`$(Build.BuildId)`	752 $(Build.BuildId) is an internal immutable ID that is also referred to as the Run ID. It's unique across the organization.
`$(DayOfMonth)`	5
`$(DayOfYear)`	217
`$(Hours)`	21
`$(Minutes)`	7
`$(Month)`	8
`$(Rev:r)`	2 (The third run is 3, and so on.) Use **$(Rev:r)** to ensure that every completed build has a unique name. When a build starts, if nothing else in the build number has changed, the Rev integer value is incremented by one. If you want to show prefix zeros in the number, you can add more **'r'** characters. For example, specify **$(Rev:rr)** if you want the Rev number to begin with 01, 02, and so on. If you use a zero-padded Rev as part of a version numbering scheme, note that some pipeline tasks or popular tools, like NuGet packages, remove the leading zeros, which cause a version number mismatch in the artifacts that are produced.
`$(Date:yyyyMMdd)`	20090824 You can specify other date formats such as **$(Date:MMddyy)**
`$(Seconds)`	3
`$(SourceBranchName)`	main
`$(TeamProject)`	Fabrikam
`$(Year:yy)`	09
`$(Year:yyyy)`	2009

Variables

You can also use user-defined and predefined variables that have a scope of "All" in your number. For example, if you've defined `My.Variable`, you could specify the following number format:

```
$(Build.DefinitionName)_$(Build.DefinitionVersion)_$(Build.RequestedFor)_$(Build.BuildId)_$(My.Variable)
```

The first four variables are predefined. `My.Variable` is defined by you on the variables tab.

Expressions

If you use an expression to set the build number, you can't use some tokens because their values aren't set at the time expressions are evaluated. These tokens include `$(Build.BuildId)`, `$(Build.BuildURL)`, and `$(Build.BuildNumber)`.

FAQ

How large can a run number be and what characters can I use?

Runs may be up to 255 characters. Characters that aren't allowed include `"`, `/`, `:`, `<`, `>`, `'`, `|`, `?`, `@`, and `*`. You can't end with a `.`.

In what time zone are the build number time values expressed?

The time zone is UTC.

How can you reference the run number variable within a script?

The run number variable can be called with `$(Build.BuildNumber)`. You can define a new variable that includes the run number or call the run number directly. In this example, `$(MyRunNumber)` is a new variable that includes the run number.

YAML

```yaml
# Set MyRunNumber
variables:
  MyRunNumber: '1.0.0-CI+$(Build.BuildNumber)'
```

```yaml
steps:
- script: echo $(MyRunNumber) # display MyRunNumber
- script: echo $(Build.BuildNumber) #display Run Number
```

How can I set the build number dynamically with conditions?

You can use variables as part of your run number. In this example, the variable `why` changes depending on the `Build.Reason` and is used as part of the run number.

YAML

```yaml
variables:
  ${{ if eq(variables['Build.Reason'], 'PullRequest') }}:
    why: pr
  ${{ elseif eq(variables['Build.Reason'], 'Manual' ) }}:
    why: manual
  ${{ elseif eq(variables['Build.Reason'], 'IndividualCI' ) }}:
    why: indivci
  ${{ else }}:
    why: other

name: $(TeamProject)_$(SourceBranchName)_$(why)_$(Date:yyyyMMdd)$(Rev:.r)

pool:
  vmImage: 'ubuntu-latest'

steps:
- script: echo '$(Build.BuildNumber)' ## output run number
```

Build options

Article • 04/05/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Create a work item on failure

If the build pipeline fails, you can automatically create a work item to track getting the problem fixed. You can specify the work item type.

You can also select if you want to assign the work item to the requestor. For example, if this is a CI build, and a team member checks in some code that breaks the build, then the work item is assigned to that person.

Additional Fields: You can set the value of work item fields. For example:

Field	Value
`System.Title`	`Build $(Build.BuildNumber) failed`
`System.Reason`	`Build failure`

Q: What other work item fields can I set? **A:** Work item field index

Allow scripts to access the OAuth token

Select this check box in classic build pipelines if you want to enable your script to use the build pipeline OAuth token. This check box is located under the "additional settings" section after selecting the agent job in the pipeline.

For an example, see Use a script to customize your build pipeline.

Default agent pool

Select the pool that's attached to the pool that contains the agents you want to run this pipeline.

> **Tip**

If your code is in Azure Pipelines and you run your builds on Windows, in many cases the simplest option is to use the **Hosted pool**.

Build job authorization scope

Specify the authorization scope for a build job. Select:

- **Project Collection** if the build needs access to multiple projects.
- **Current Project** if you want to restrict this build to have access only the resources in the current project.

For more information, see Understand job access tokens.

Build (run) number

This documentation has moved to Build (run) number.

Pipeline Run Retention

Article • 06/14/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Retaining a pipeline run for longer than the configured project settings is handled by the creation of **retention leases**. Temporary retention leases are often created by automatic processes and more permanent leases by manipulating the UI or when Release Management retains artifacts, but they can also be manipulated through the REST API. Here are some examples of tasks that you can add to your yaml pipeline for retention.

Prerequisites

By default, members of the Contributors, Build Admins, Project Admins, and Release Admins groups can manage retention policies.

Example: Override a short project-level retention window

In this example, the project is configured to delete pipeline runs after only 30 days.

If a pipeline in this project is important and runs should be retained for longer than 30 days, this task ensures the run is valid for two years by adding a new retention lease.

PowerShell

YAML

```
  - task: PowerShell@2
    condition: and(succeeded(), not(canceled()))
    name: RetainOnSuccess
    displayName: Retain on Success
    inputs:
      failOnStderr: true
      targetType: 'inline'
      script: |
        $contentType = "application/json";
        $headers = @{ Authorization = 'Bearer $(System.AccessToken)' };
        $rawRequest = @{ daysValid = 365 * 2; definitionId =
  $(System.DefinitionId); ownerId = 'User:$(Build.RequestedForId)';
  protectPipeline = $false; runId = $(Build.BuildId) };
        $request = ConvertTo-Json @($rawRequest);
        $uri =
  "$(System.CollectionUri)$(System.TeamProject)/_apis/build/retention/leas
  es?api-version=6.0-preview.1";
        Invoke-RestMethod -uri $uri -method POST -Headers $headers -
  ContentType $contentType -Body $request;
```

Question: can a pipeline be retained for *less* than the configured project values?

No, leases don't work in the reverse. If a project is configured to retain for two years, pipeline runs the system won't remove runs for two years. To delete these runs earlier, manually delete them or use the equivalent REST API.

Example: Only runs on branches named `releases/*` should be retained for a long time

This is similar to above, only the condition needs to change:

YAML

```
- task: PowerShell@2
  condition: and(succeeded(), not(canceled()),
startsWith(variables['Build.SourceBranchName'], 'releases/'))
  name: RetainReleaseBuildOnSuccess
  displayName: Retain Release Build on Success
  inputs:
    failOnStderr: true
    targetType: 'inline'
    script: |
      $contentType = "application/json";
      $headers = @{ Authorization = 'Bearer $(System.AccessToken)' };
      $rawRequest = @{ daysValid = 365 * 2; definitionId =
$(System.DefinitionId); ownerId = 'User:$(Build.RequestedForId)';
```

```
protectPipeline = $false; runId = $(Build.BuildId) };
      $request = ConvertTo-Json @($rawRequest);
      $uri =
"$(System.CollectionUri)$(System.TeamProject)/_apis/build/retention/leases?
api-version=6.0-preview.1";
      Invoke-RestMethod -uri $uri -method POST -Headers $headers -
ContentType $contentType -Body $request;
```

Example: Updating the retention window for a multi-stage pipeline based on stage success

Consider a two-stage pipeline that first runs a build and then a release. When successful, the `Build` stage retains the run for three days, but the project administrator wants a successful `Release` stage to extend the lease to one year.

The `Build` stage can retain the pipeline as in the above examples, but with one addition: by saving the new lease's `Id` in an output variable, the lease can be updated later when the release stage runs.

YAML

```yaml
- task: PowerShell@2
  condition: and(succeeded(), not(canceled()))
  name: RetainOnSuccess
  displayName: Retain on Success
  inputs:
    failOnStderr: true
    targetType: 'inline'
    script: |
      $contentType = "application/json";
      $headers = @{ Authorization = 'Bearer $(System.AccessToken)' };
      $rawRequest = @{ daysValid = 365; definitionId =
$(System.DefinitionId); ownerId = 'User:$(Build.RequestedForId)';
protectPipeline = $false; runId = $(Build.BuildId) };
      $request = ConvertTo-Json @($rawRequest);
      $uri =
"$(System.CollectionUri)$(System.TeamProject)/_apis/build/retention/leases?
api-version=6.0-preview.1";
      $newLease = Invoke-RestMethod -uri $uri -method POST -Headers $headers
-ContentType $contentType -Body $request;
      $newLeaseId = $newLease.Value[0].LeaseId
      echo "##vso[task.setvariable
variable=newLeaseId;isOutput=true]$newLeaseId";
```

To *update* a retention lease requires a different REST API call.

YAML

```
- stage: Release
  dependsOn: Build
  jobs:
  - job: default
    variables:
    - name: NewLeaseId
      value: $[
stageDependencies.Build.default.outputs['RetainOnSuccess.newLeaseId']]
    steps:
    - task: PowerShell@2
      condition: and(succeeded(), not(canceled()))
      name: RetainOnSuccess
      displayName: Retain on Success
      inputs:
        failOnStderr: true
        targetType: 'inline'
        script: |
          $contentType = "application/json";
          $headers = @{ Authorization = 'Bearer $(System.AccessToken)' };
          $rawRequest = @{ daysValid = 365 };
          $request = ConvertTo-Json $rawRequest;
          $uri =
"$(System.CollectionUri)$(System.TeamProject)/_apis/build/retention/leases/$
newLeaseId?api-version=7.1-preview.2";
          Invoke-RestMethod -uri $uri -method PATCH -Headers $headers -
ContentType $contentType -Body $request;
```

Next steps

With these examples, you learned how to use custom pipeline tasks to manage run retention.

You learned how to:

- ✓ Override a short retention window
- ✓ Override retention for runs on specific branches
- ✓ Update a retention lease when a run should be retained even longer

About pipeline tests

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

This article describes commonly used terms used in pipeline test report and test analytics.

Term	Definition
Duration	Time elapsed in execution of a **test**, **test run**, or **entire test execution** in a build or release pipeline.
Owner	Owner of a **test** or **test run**. The test owner is typically specified as an attribute in the test code. See Publish Test Results task to view the mapping of the **Owner** attribute for supported test result formats.
Failing build	Reference to the **build** having the first occurrence of consecutive failures of a test case.
Failing release	Reference to the **release** having the first occurrence of consecutive failures of a test case.
Outcome	There are 15 possible outcomes for a test result: Aborted, Blocked, Error, Failed, Inconclusive, In progress, None, Not applicable, Not executed, Not impacted, Passed, Paused, Timeout, Unspecified, and Warning. Some of the commonly used outcomes are: - **Aborted**: Test execution terminated abruptly due to internal or external factors, e.g., bad code, environment issues. - **Failed**: Test not meeting the desired outcome. - **Inconclusive**: Test without a definitive outcome. - **Not executed**: Test marked as skipped for execution. - **Not impacted**: Test not impacted by the code change that triggered the pipeline. - **Passed**: Test executed successfully. - **Timeout**: Test execution duration exceeding the specified threshold.
Flaky test	A test with non-deterministic behavior. For example, the test may result in different outcomes for the same configuration, code, or inputs.
Filter	Mechanism to search for the test results within the result set, using the available attributes. Learn more.
Grouping	An aid to organizing the test results view based on available attributes such as **Requirement**, **Test files**, **Priority**, and more. Both test report and test analytics provide support for grouping test results.

Term	Definition
Pass percentage	Measure of the success of test outcome for a single instance of execution or over a period of time.
Priority	Specifies the degree of importance or criticality of a test. Priority is typically specified as an attribute in the test code. See Publish Test Results task to view the mapping of the **Priority** attribute for supported test result formats.
Test analytics	A view of the historical test data to provide meaningful insights.
Test case	Uniquely identifies a single test within the specified branch.
Test files	Group tests based on the way they are packaged; such as files, DLLs, or other formats.
Test report	A view of single instance of test execution in the pipeline that contains details of status and help for troubleshooting, traceability, and more.
Test result	Single instance of execution of a test case with a specific outcome and details.
Test run	Logical grouping of test results based on: - **Test executed using built-in tasks**: All tests executed using a single task such as Visual Studio Test, Ant, Maven, Gulp, Grunt or Xcode will be reported under a single test run - **Results published using Publish Test Results task**: Provides an option to group all test results from one or more test results files into a single run, or individual runs per file - **Tests results published using API(s)**: API(s) provide the flexibility to create test runs and organize test results for each run as required.
Traceability	Ability to trace forward or backward to a requirement, bug, or source code from a test result.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

Run tests in parallel using the Visual Studio Test task

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Running tests to validate changes to code is key to maintaining quality. For continuous integration practice to be successful, it is essential you have a good test suite that is run with every build. However, as the codebase grows, the regression test suite tends to grow as well and running a full regression test can take a long time. Sometimes, tests themselves may be long running - this is typically the case if you write end-to-end tests. This reduces the speed with which customer value can be delivered as pipelines cannot process builds quickly enough.

Running tests in parallel is a great way to improve the efficiency of CI/CD pipelines. This can be done easily by employing the additional capacity offered by the cloud. This article discusses how you can configure the Visual Studio Test task to run tests in parallel by using multiple agents.

Pre-requisite

Familiarize yourself with the concepts of agents and jobs. To run multiple jobs in parallel, you must configure multiple agents. You also need sufficient parallel jobs.

Test slicing

The Visual Studio Test task (version 2) is designed to work seamlessly with parallel job settings. When a pipeline job that contains the Visual Studio Test task (referred to as the "VSTest task" for simplicity) is configured to run on multiple agents in parallel, it automatically detects that multiple agents are involved and creates test slices that can be run in parallel across these agents.

The task can be configured to create test slices to suit different requirements such as batching based on the number of tests and agents, the previous test running times, or the location of tests in assemblies.

These options are explained in the following sections.

Simple slicing based on the number of tests and agents

This setting uses a simple slicing algorithm to divide up the number of tests 'T' across 'N' agents so that each agent runs T/N tests. For example, if your test suite contains 1000 tests, and you use two agents for parallel jobs, each agent will run 500 tests. Or you can reduce the amount of time taken to run the tests even further by using eight agents, in which case each agent runs 125 tests in parallel.

This option is typically used when all tests have similar running times. If test running times are not similar, agents may not be utilized effectively because some agents may receive slices with several long-running tests, while other agents may receive slices with short-running tests and finish much earlier than the rest of the agents.

Slicing based on the past running time of tests

This setting considers past running times to create slices of tests so that each slice has approximately the same running time. Short-running tests will be batched together, while long-running tests will be allocated to separate slices.

This option should be used when tests within an assembly do not have dependencies, and do not need to run on the same agent. This option results in the most efficient utilization of agents because every agent gets the same amount of 'work' and all finish at approximately the same time.

Slicing based on test assemblies

This setting uses a simple slicing algorithm that divides up the number of test assemblies (or files) 'A' across 'N' agents, so that each agent runs tests from A/N assemblies. The number of tests within an assembly is not taken into account when using this option. For example, if your test suite contains ten test assemblies and you use two agents for parallel jobs, each agent will receive five test assemblies to run. You

can reduce the amount of time taken to run the tests even further by using five agents, in which case each agent gets two test assemblies to run.

This option should be used when tests within an assembly have dependencies or utilize `AssemblyInitialize` and `AssemblyCleanup`, or `ClassInitialize` and `ClassCleanup` methods, to manage state in your test code.

Run tests in parallel in classic build pipelines

If you have a large test suite or long-running integration tests to run in your classic build pipeline, use the following steps.

> **Note**
>
> To use the multi-agent capability in build pipelines with on-premises TFS server, you must use TFS 2018 Update 2 or a later version.

1. **Build job using a single agent**. Build Visual Studio projects and publish build artifacts using the tasks shown in the following image. This uses the default job settings (single agent, no parallel jobs).

2. **Run tests in parallel using multiple agents**:

- Add an **agent job**

- Configure the job to use **multiple agents in parallel**. The example here uses three agents.

> **Tip**
>
> For massively parallel testing, you can specify as many as 99 agents.

- Add a **Download Build Artifacts** task to the job. This step is the link between the build job and the test job, and is necessary to ensure that the binaries generated in the build job are available on the agents used by the test job to run tests. Ensure that the task is set to download artifacts produced by the 'Current build' and the artifact name is the same as the artifact name used in the **Publish Build Artifacts** task in the build job.

- Add the **Visual Studio Test** task and configure it to use the required slicing strategy.

Setting up jobs for parallel testing in YAML pipelines

Specify the `parallel` strategy in the `job` and indicate how many jobs should be dispatched. You can specify as many as 99 agents to scale up testing for large test suites.

```
YAML
```

```yaml
jobs:
- job: ParallelTesting
  strategy:
    parallel: 2
```

For more information, see YAML schema - Job.

Run tests in parallel in classic release pipelines

Use the following steps if you have a large test suite or long-running functional tests to run after deploying your application. For example, you may want to deploy a web-application and run Selenium tests in a browser to validate the app functionality.

> **Note**
>
> To use the multi-agent capability in release pipelines with on-premises TFS server, you must use TFS 2017 Update 1 or a later version.

1. **Deploy app using a single agent**. Use the tasks shown in the image below to deploy a web app to Azure App Services. This uses the default job settings (single agent, no parallel jobs).

2. **Run tests in parallel using multiple agents**:

- Add an **agent job**

- Configure the job to use **multiple agents in parallel**. The example here uses three agents.

> **Tip**
>
> For massively parallel testing, you can specify as many as 99 agents.

- Add any **additional tasks** that must run before the Visual Studio test task is run. For example, run a PowerShell script to set up any data required by your tests.

> **Tip**
>
> Jobs in release pipelines download all artifacts linked to the release pipeline by default. To save time, you can configure the job to download only the test artifacts required by the job. For example, web app binaries are not required to run Selenium tests and downloading these can be skipped if the app and test artifacts are published separately by your build pipeline.

- Add the **Visual Studio Test** task and configure it to use the required slicing strategy.

> **Tip**

> If the test machines do not have Visual Studio installed, you can use the **Visual Studio Test Platform Installer task** to acquire the required version of the test platform.

Massively parallel testing by combining parallel pipeline jobs with parallel test execution

When parallel jobs are used in a pipeline, it employs multiple machines (agents) to run each job in parallel. Test frameworks and runners also provide the capability to run tests in parallel on a single machine, typically by creating multiple processes or threads that are run in parallel. Parallelism features can be combined in a layered fashion to achieve massively parallel testing. In the context of the Visual Studio Test task, parallelism can be combined in the following ways:

1. **Parallelism offered by test frameworks**. All modern test frameworks such as MSTest v2, NUnit, xUnit, and others provide the ability to run tests in parallel. Typically, tests in an assembly are run in parallel. These test frameworks interface with the Visual Studio Test platform using a test adapter and the test framework, together with the corresponding adapter, and work within a test host process that the Visual Studio Test Platform creates when tests are run. Therefore, parallelization at this layer is within a process for all frameworks and adapters.

2. **Parallelism offered by the Visual Studio Test Platform (vstest.console.exe)**. Visual Studio Test Platform can run test assemblies in parallel. Users of vstest.console.exe will recognize this as the /parallel switch. It does so by launching a test host process on each available core, and handing it tests in an assembly to execute. This works for any framework that has a test adapter for the Visual Studio test platform because the unit of parallelization is a test assembly or test file. This, when combined with the parallelism offered by test frameworks (described above), provides the maximum degree of parallelization when tests run on a single agent in the pipeline.

3. **Parallelism offered by the Visual Studio Test (VSTest) task**. The VSTest task supports running tests in parallel across multiple agents (or machines). Test slices are created, and each agent executes one slice at a time. The three different slicing strategies, when combined with the parallelism offered by the test platform and test framework (as described above), result in the following:

    - Slicing based on the number of tests and agents. Simple slicing where tests are grouped in equally sized slices. A slice contains tests from one or more

assemblies. Test execution on the agent then conforms to the parallelism described in **1** and **2** above.

- Slicing based on past running time. Based on the previous timings for running tests, and the number of available agents, tests are grouped into slices such that each slice requires approximately equal execution time. A slice contains tests from one or more assemblies. Test execution on the agent then conforms to the parallelism described in **1** and **2** above.

- Slicing based on assemblies. A slice is a test assembly, and so contains tests that all belong to the same assembly. Execution on the agent then conforms to the parallelism described in **1** and **2** above. However, **2** may not occur if an agent receives only one assembly to run.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

Run tests in parallel for any test runner

Article • 05/24/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Running tests to validate changes to code is key to maintaining quality. For continuous integration practice to be successful, it is essential you have a good test suite that is run with every build. However, as the codebase grows, the regression test suite tends to grow as well and running a full regression test can take a long time. Sometimes, tests themselves may be long running - this is typically the case if you write end-to-end tests. This reduces the speed with which customer value can be delivered as pipelines cannot process builds quickly enough.

Running tests in parallel is a great way to improve the efficiency of CI/CD pipelines. This can be done easily by employing the additional capacity offered by the cloud. This article discusses how you can parallelize tests by using multiple agents to process jobs.

Pre-requisite

Familiarize yourself with the concepts of agents and jobs. Each agent can run only one job at a time. To run multiple jobs in parallel, you must configure multiple agents. You also need sufficient parallel jobs.

Setting up parallel jobs

Specify 'parallel' strategy in the YAML and indicate how many jobs should be dispatched. The variables `System.JobPositionInPhase` and `System.TotalJobsInPhase` are added to each job.

YAML

```yaml
jobs:
- job: ParallelTesting
  strategy:
    parallel: 2
```

Tip

You can specify as many as 99 agents to scale up testing for large test suites.

Slicing the test suite

To run tests in parallel you must first slice (or partition) the test suite so that each slice can be run independently. For example, instead of running a large suite of 1000 tests on a single agent, you can use two agents and run 500 tests in parallel on each agent. Or you can reduce the amount of time taken to run the tests even further by using 8 agents and running 125 tests in parallel on each agent.

The step that runs the tests in a job needs to know which test slice should be run. The variables `System.JobPositionInPhase` and `System.TotalJobsInPhase` can be used for this purpose:

- `System.TotalJobsInPhase` indicates the total number of slices (you can think of this as "totalSlices")
- `System.JobPositionInPhase` identifies a particular slice (you can think of this as "sliceNum")

If you represent all test files as a single dimensional array, each job can run a test file indexed at [sliceNum + totalSlices], until all the test files are run. For example, if you have six test files and two parallel jobs, the first job (slice0) will run test files numbered 0, 2, and 4, and second job (slice1) will run test files numbered 1, 3, and 5.

If you use three parallel jobs instead, the first job (slice0) will run test files numbered 0 and 3, the second job (slice1) will run test files numbered 1 and 4, and the third job (slice2) will run test files numbered 2 and 5.

Sample code

This .NET Core sample uses `--list-tests` and `--filter` parameters of `dotnet test` to slice the tests. The tests are run using NUnit. Test results created by `DotNetCoreCLI@2`

test task are then published to the server. Import (into Azure Repos or Azure DevOps Server) or fork (into GitHub) this repo:

```
https://github.com/idubnori/ParallelTestingSample-dotnet-core
```

This Python sample uses a PowerShell script to slice the tests. The tests are run using pytest. JUnit-style test results created by pytest are then published to the server. Import (into Azure Repos or Azure DevOps Server) or fork (into GitHub) this repo:

```
https://github.com/PBoraMSFT/ParallelTestingSample-Python
```

This JavaScript sample uses a bash script to slice the tests. The tests are run using the mocha runner. JUnit-style test results created by mocha are then published to the server. Import (into Azure Repos or Azure DevOps Server) or fork (into GitHub) this repo:

```
https://github.com/PBoraMSFT/ParallelTestingSample-Mocha
```

The sample code includes a file `azure-pipelines.yml` at the root of the repository that you can use to create a pipeline. Follow all the instructions in Create your first pipeline to create a pipeline and see test slicing in action.

Combine parallelism for massively parallel testing

When parallel jobs are used in a pipeline, the pipeline employs multiple machines to run each job in parallel. Most test runners provide the capability to run tests in parallel on a single machine (typically by creating multiple processes or threads that are run in parallel). The two types of parallelism can be combined for massively parallel testing, which makes testing in pipelines extremely efficient.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

Speed up testing by using Test Impact Analysis (TIA)

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Continuous Integration (CI) is a key practice in the industry. Integrations are frequent, and verified with an automated build that runs regression tests to detect integration errors as soon as possible. However, as the codebase grows and matures, its regression test suite tends to grow as well - to the extent that running a full regression test might require hours. This slows down the frequency of integrations, and ultimately defeats the purpose of continuous integration. In order to have a CI pipeline that completes quickly, some teams defer the execution of their longer running tests to a separate stage in the pipeline. However, this only serves to further defeat continuous integration.

Instead, enable Test Impact Analysis (TIA) when using the Visual Studio Test task in a build pipeline. TIA performs incremental validation by automatic test selection. It will automatically select only the subset of tests required to validate the code being committed. For a given code commit entering the CI/CD pipeline, TIA will select and run only the relevant tests required to validate that commit. Therefore, that test run will complete more quickly, if there is a failure you will get to know about it sooner, and because it is all scoped by relevance, analysis will be faster as well.

Test Impact Analysis has:

- **A robust test selection mechanism**. It includes existing impacted tests, previously failing tests, and newly added tests.
- **Safe fallback**. For commits and scenarios that TIA cannot understand, it will fall back to running all tests. TIA is currently scoped to only managed code, and single machine topology. So, for example, if the code commit contains changes to HTML or CSS files, it cannot reason about them and will fall back to running all tests.
- **Configurable overrides**. You can run all tests at a configured periodicity.

However, be aware of the following caveats when using TIA with Visual Studio 2015:

- **Running tests in parallel**. In this case, tests will run serially.
- **Running tests with code coverage enabled**. In this case, code coverage data will not get collected.

Test Impact Analysis supported scenarios

At present, TIA is supported for:

- TFS 2017 Update 1 onwards, and Azure Pipelines
- Version 2.* of the Visual Studio Test task in the build pipeline
- Build vNext, with multiple VSTest Tasks
- VS2015 Update 3 onwards on the build agent
- Local and hosted build agents
- CI and in PR workflows
- Git, GitHub, Other Git, TFVC repos (including partially mapped TFVC repositories with a workaround)
- IIS interactions (over REST, SOAP APIs), using HTTP/HTTPS protocols
- Automated Tests
- Single machine topology. Tests and app (SUT) must be running on the same machine.
- Managed code (any .NET Framework app, any .NET service)

At present, TIA is **not** supported for:

- Multi-machine topology (where the test is exercising an app deployed to a different machine)
- Data driven tests
- Test Adapter-specific parallel test execution
- .NET Core
- UWP

More information about TIA scope and applications

Enable Test Impact Analysis

TIA is supported through Version 2.* of the Visual Studio Test task. If your app is a single tier application, all you need to do is to check **Run only impacted tests** in the task UI. The Test Impact data collector is automatically configured. No additional steps are required.

If your application interacts with a service in the context of IIS, you must also configure the Test Impact data collector to run in the context of IIS by using a **.runsettings** file. Here is a sample that creates this configuration:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<RunSettings>
```

```
  <DataCollectionRunSettings>
    <DataCollectors>
      <!-- This is the TestImpact data collector.-->
      <DataCollector uri="datacollector://microsoft/TestImpact/1.0"
assemblyQualifiedName="Microsoft.VisualStudio.TraceCollector.TestImpactDataC
ollector, Microsoft.VisualStudio.TraceCollector, Version=15.0.0.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" friendlyName="Test
Impact">
        <Configuration>
          <!-- enable IIS data collection-->
          <InstrumentIIS>True</InstrumentIIS>
          <!-- file level data collection -->
          <ImpactLevel>file</ImpactLevel>
          <!-- any job agent related executable or any other service that
the test is using needs to be profiled. -->
          <ServicesToInstrument>
            <Name>TeamFoundationSshService</Name>
          </ServicesToInstrument>
        </Configuration>
      </DataCollector>
    </DataCollectors>
  </DataCollectionRunSettings>
</RunSettings>
```

View Test Impact Analysis outcome

TIA is integrated into existing test reporting at both the summary and details levels, including notification emails.

More information about TIA and Azure Pipelines integration

Manage Test Impact Analysis behavior

You can influence the way that tests are either included or ignored during a test run:

- **Through the VSTest task UI**. TIA can be conditioned to run all tests at a configured periodicity. Setting this option is recommended, and is the means to regulate test selection.
- **By setting a build variable**. Even after TIA has been enabled in the VSTest task, it can be disabled for a specific build by setting the variable **DisableTestImpactAnalysis** to **true**. This override will force TIA to run all tests for that build. In subsequent builds, TIA will go back to optimized test selection.

When TIA opens a commit and sees an unknown file type, it falls back to running all tests. While this is good from a safety perspective, tuning this behavior might be useful in some cases. For example:

- Set the **TI_IncludePathFilters** variable to specific paths to include only these paths in a repository for which you want TIA to apply. This is useful when teams use a shared repository. Setting this variable disables TIA for all other paths not included in the setting.
- Set the **TIA_IncludePathFilters** variable to specify file types that do not influence the outcome of tests and for which changes should be ignored. For example, to ignore changes to .csproj files set the variable to the value **!***.csproj**.

> Use the minimatch pattern when setting variables, and separate multiple items with a semicolon.

To evaluate whether TIA is selecting the appropriate tests:

- Manually validate the selection. A developer who knows how the SUT and tests are architected could manually validate the test selection using the TIA reporting capabilities.
- Run TIA selected tests and then all tests in sequence. In a build pipeline, use two test tasks - one that runs only impacted Tests (T1) and one that runs all tests (T2). If T1 passes, check that T2 passes as well. If there was a failing test in T1, check that T2 reports the same set of failures.

More information about TIA advanced configuration

Provide custom dependency mappings

TIA uses dependency maps of the following form.

```
map
```

```
TestMethod1
  dependency1
  dependency2
TestMethod2
  dependency1
  dependency3
```

TIA can generate such a dependencies map for managed code execution. Where such dependencies reside in **.cs** and **.vb** files, TIA can automatically watch for commits into such files and then run tests that had these source files in their list of dependencies.

You can extend the scope of TIA by explicitly providing the dependencies map as an XML file. For example, you might want to support code in other languages such as JavaScript or C++, or support the scenario where tests and product code are running on different machines. The mapping can even be approximate, and the set of tests you want to run can be specified in terms of a test case filter such as you would typically provide in the VSTest task parameters.

The XML file should be checked into your repository, typically at the root level. Then set the build variable **TIA.UserMapFile** to point to it. For example, if the file is named **TIAmap.xml**, set the variable to **$(System.DefaultWorkingDirectory)/TIAmap.xml**.

For an example of the XML file format, see TIA custom dependency mapping.

See Also

- TIA overview and VSTS integration
- TIA scope and applications
- TIA advanced configuration
- TIA custom dependency mapping

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

Manage flaky tests

Article • 02/24/2023

Azure DevOps Services

Productivity for developers relies on the ability of tests to find real problems with the code under development or update in a timely and reliable fashion. Flaky tests present a barrier to finding real problems, since the failures often don't relate to the changes being tested. A flaky test is a test that provides different outcomes, such as pass or fail, even when there are no changes in the source code or execution environment. Flaky tests also impact the quality of shipped code.

> **Note**
>
> This feature is only available on Azure DevOps Services. Typically, new features are introduced in the cloud service first, and then made available on-premises in the next major version or update of Azure DevOps Server. To learn more, see **Azure DevOps Feature Timeline**.

The goal of bringing flaky test management in-product is to reduce developer pain cause by flaky tests and cater to the whole workflow. Flaky test management provides the following benefits.

- **Detection** - Auto detection of flaky test with rerun or extensibility to plug in your own custom detection method
- **Management of flakiness** - Once a test is marked as flaky, the data is available for all pipelines for that branch
- **Report on flaky tests** - Ability to choose if you want to prevent build failures caused by flaky tests, or use the flaky tag only for troubleshooting
- **Resolution** - Manual bug-creation or manual marking and unmarking test as flaky based on your analysis
- **Close the loop** - Reset flaky test as a result of bug resolution / manual input

Enable flaky test management

To configure flaky test management, choose **Project settings**, and select **Test management** in the **Pipelines** section.

Slide the On/Off button to **On**.

The default setting for all projects is to use flaky tests for troubleshooting.

Flaky test detection

Flaky test management supports system and custom detection.

- **System detection**: The in-product flaky detection uses test rerun data. The detection is via **VSTest task** rerunning of failed tests capability or retry of stage in the pipeline. You can select specific pipelines in the project for which you would like to detect flaky tests.

> ⓘ Note

> Once a test is marked as flaky, the data is available for all pipelines for that branch to assist with troubleshooting in every pipeline.

- **Custom detection**: You can integrate your own flaky detection mechanism with Azure Pipelines and use the reporting capability. With custom detection, you need to update the test results metadata for flaky tests. For details, see Test Results, Result Meta Data - Update REST API.

Flaky test options

The **Flaky test options** specify how flaky tests are available in test reporting as well as resolution capabilities, as described in the following sections.

Flaky test management and reporting

On the Test management page under **Flaky test options**, you can set options for how flaky tests are included in the Test Summary report. Flaky test data for both passed and failed test is available in Test results. The **Flaky** tag helps you identify flaky tests. By

default, flaky tests are included in the Test Summary. However, if you want to ensure flaky test failures don't fail your pipeline, you can choose to not include them in your test summary and suppress the test failure. This option ensures flaky tests (both passed and failed) are removed from the pass percentage and shown in **Tests not reported**, as shown in the following screenshot.

> ⓘ **Note**
>
> The Test summary report is updated only for **Visual Studio Test task** and **Publish Test Results task**. You may need to add a custom script to suppress flaky test failure for other scenarios.

Tests marked as flaky

You can mark or unmark a test as flaky based on analysis or context, by choosing **Flaky** (or **UnFlaky**, depending on whether the test is already marked as flaky.)

When a test is marked flaky or unflaky in a pipeline, no changes are made in the current pipeline. Only on future executions of that test is the changed flaky setting evaluated. Tests marked as flaky have the *Marked flaky* tag in the user interface.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

Related articles

- Review test results
- Visual Studio Test task
- Publish Test Results task
- Test Results, Result Meta Data - Update REST API

UI testing considerations

Article • 05/10/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

When running automated tests in the CI/CD pipeline, you may need a special configuration in order to run UI tests such as Selenium, Appium or Coded UI tests. This topic describes the typical considerations for running UI tests.

Prerequisites

Familiarize yourself with agents and deploying an agent on Windows.

Headless mode or visible UI mode?

When running Selenium tests for a web app, you can launch the browser in two ways:

1. **Headless mode**. In this mode, the browser runs as normal but without any UI components being visible. While this mode is obviously not useful for browsing the web, it is useful for running automated tests in an unattended manner in a CI/CD pipeline. Chrome and Firefox browsers can be run in headless mode.

   This mode generally consumes less resources on the machine because the UI is not rendered and tests run faster. As a result, potentially more tests can be run in parallel on the same machine to reduce the total test execution time.

   Screenshots can be captured in this mode and used for troubleshooting failures.

   > **Note**
   >
   > Microsoft Edge browser currently cannot be run in the headless mode.

2. **Visible UI mode**. In this mode, the browser runs normally and the UI components are visible. When running tests in this mode on Windows, special configuration of the agents is required.

If you are running UI tests for a desktop application, such as Appium tests using WinAppDriver or Coded UI tests, a special configuration of the agents is required.

> **Tip**
>
> End-to-end UI tests generally tend to be long-running. When using the visible UI mode, depending on the test framework, you may not be able to run tests in parallel on the same machine because the app must be in focus to receive keyboard and mouse events. In this scenario, you can speed up testing cycles by running tests in parallel on *different* machines. See **run tests in parallel for any test runner** and **run tests in parallel using Visual Studio Test task**.

UI testing in visible UI mode

A special configuration is required for agents to run UI tests in visible UI mode.

Visible UI testing using Microsoft-hosted agents

Microsoft-hosted agents are pre-configured for UI testing and UI tests for both web apps and desktop apps. Microsoft-hosted agents are also pre-configured with popular browsers and matching web-driver versions that can be used for running Selenium tests. The browsers and corresponding web-drivers are updated on a periodic basis. To learn more about running Selenium tests, see UI test with Selenium

Visible UI testing using self-hosted Windows agents

Agents that are configured to run as service can run Selenium tests only with headless browsers. If you are not using a headless browser, or if you are running UI tests for desktop apps, Windows agents *must* be configured to run as an interactive process with auto-logon enabled.

When configuring agents, select 'No' when prompted to run as a service. Subsequent steps then allow you to configure the agent with auto-logon. When your UI tests run, applications and browsers are launched in the context of the user specified in the auto-logon settings.

If you use Remote Desktop to access the computer on which an agent is running with auto-logon, simply disconnecting the Remote Desktop causes the computer to be locked and any UI tests that run on this agent may fail. To avoid this, use the tscon command on the remote computer to disconnect from Remote Desktop. For example:

```
%windir%\System32\tscon.exe 1 /dest:console
```

In this example, the number '1' is the ID of the remote desktop session. This number may change between remote sessions, but can be viewed in Task Manager. Alternatively, to automate finding the current session ID, create a batch file containing the following code:

```batch
for /f "skip=1 tokens=3" %%s in ('query user %USERNAME%') do (
  %windir%\System32\tscon.exe %%s /dest:console
)
```

Save the batch file and create a desktop shortcut to it, then change the shortcut properties to 'Run as administrator'. Running the batch file from this shortcut disconnects from the remote desktop but preserves the UI session and allows UI tests to run.

Provisioning agents in Azure VMs for UI testing

If you are provisioning virtual machines (VMs) on Azure, agent configuration for UI testing is available through the Agent artifact for DevTest Labs.

Setting screen resolution

Before running UI tests you may need to adjust the screen resolution so that apps render correctly. For this, a screen resolution utility task is available from Marketplace. Use this task in your pipeline to set the screen resolution to a value that is supported by the agent machine. By default, this utility sets the resolution to the optimal value supported by the agent machine.

If you encounter failures using the screen resolution task, ensure that the agent is configured to run with auto-logon enabled and that all remote desktop sessions are safely disconnected using the **tscon** command as described above.

> **Note**
>
> The screen resolution utility task runs on the unified build/release/test agent, and cannot be used with the deprecated **Run Functional Tests task**.

Troubleshooting failures in UI tests

When you run UI tests in an unattended manner, capturing diagnostic data such as screenshots or video is useful for discovering the state of the application when the failure was encountered.

Capture screenshots

Most UI testing frameworks provide the ability to capture screenshots. The screenshots collected are available as an attachment to the test results when these results are published to the server.

If you use the Visual Studio test task to run tests, captured screenshots must be added as a result file in order to be available in the test report. For this, use the following code:

MSTest

First, ensure that TestContext is defined in your test class. For example: `public TestContext TestContext { get; set; }`

Add the screenshot file using `TestContext.AddResultFile(fileName); //Where fileName is the name of the file.`

If you use the Publish Test Results task to publish results, test result attachments can only be published if you are using the VSTest (TRX) results format or the NUnit 3.0 results format.

Result attachments cannot be published if you use JUnit or xUnit test results. This is because these test result formats do not have a formal definition for attachments in the results schema. You can use one of the below approaches to publish test attachments instead.

- If you are running tests in the build (CI) pipeline, you can use the Copy and Publish Build Artifacts task to publish any additional files created in your tests. These will appear in the **Artifacts** page of your build summary.

- Use the REST APIs to publish the necessary attachments. Code samples can be found in this GitHub repository.

Capture video

If you use the Visual Studio test task to run tests, video of the test can be captured and is automatically available as an attachment to the test result. For this, you must configure the video data collector in a **.runsettings** file and this file must be specified in the task settings.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

UI test with Selenium

Article • 04/05/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Performing user interface (UI) testing as part of the release pipeline is a great way of detecting unexpected changes, and need not be difficult. This topic describes using Selenium to test your website during a continuous deployment release and test automation. Special considerations that apply when running UI tests are discussed in UI testing considerations.

> Typically you will run unit tests in your build workflow, and functional (UI) tests in your release workflow after your app is deployed (usually to a QA environment).

For more information about Selenium browser automation, see:

- Selenium
- Selenium documentation

Create your test project

As there is no template for Selenium testing, the easiest way to get started is to use the Unit Test template. This automatically adds the test framework references and enables you run and view the results from Visual Studio Test Explorer.

1. In Visual Studio, open the **File** menu and choose **New Project**, then choose **Test** and select **Unit Test Project**. Alternatively, open the shortcut menu for the solution and choose **Add** then **New Project** and then **Unit Test Project**.

2. After the project is created, add the Selenium and browser driver references used by the browser to execute the tests. Open the shortcut menu for the Unit Test project and choose **Manage NuGet Packages**. Add the following packages to your project:

    - Selenium.WebDriver
    - Selenium.Firefox.WebDriver
    - Selenium.WebDriver.ChromeDriver
    - Selenium.WebDriver.IEDriver

3. Create your tests. For example, the following code creates a default class named **MySeleniumTests** that performs a simple test on the Bing.com website. Replace the contents of the **TheBingSearchTest** function with the Selenium code required to test your web app or website. Change the **browser** assignment in the **SetupTest** function to the browser you want to use for the test.

C#

```
using System;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;
using OpenQA.Selenium.Chrome;
using OpenQA.Selenium.IE;

namespace SeleniumBingTests
{
  /// <summary>
  /// Summary description for MySeleniumTests
  /// </summary>
  [TestClass]
  public class MySeleniumTests
  {
    private TestContext testContextInstance;
    private IWebDriver driver;
    private string appURL;

    public MySeleniumTests()
    {
    }

    [TestMethod]
    [TestCategory("Chrome")]
```

```
    public void TheBingSearchTest()
    {
      driver.Navigate().GoToUrl(appURL + "/");
      driver.FindElement(By.Id("sb_form_q")).SendKeys("Azure
Pipelines");
      driver.FindElement(By.Id("sb_form_go")).Click();

driver.FindElement(By.XPath("//ol[@id='b_results']/li/h2/a/strong[3]"))
.Click();
      Assert.IsTrue(driver.Title.Contains("Azure Pipelines"), "Verified
title of the page");
    }

    /// <summary>
    ///Gets or sets the test context which provides
    ///information about and functionality for the current test run.
    ///</summary>
    public TestContext TestContext
    {
      get
      {
        return testContextInstance;
      }
      set
      {
        testContextInstance = value;
      }
    }

    [TestInitialize()]
    public void SetupTest()
    {
      appURL = "http://www.bing.com/";

      string browser = "Chrome";
      switch(browser)
      {
        case "Chrome":
          driver = new ChromeDriver();
          break;
        case "Firefox":
          driver = new FirefoxDriver();
          break;
        case "IE":
          driver = new InternetExplorerDriver();
          break;
        default:
          driver = new ChromeDriver();
          break;
      }

    }

    [TestCleanup()]
    public void MyTestCleanup()
```

```
      {
        driver.Quit();
      }
    }
  }
```

4. Run the Selenium test locally using Test Explorer and check that it works.

Define your build pipeline

You'll need a continuous integration (CI) build pipeline that builds your Selenium tests. For more details, see Build your .NET desktop app for Windows.

Create your web app

You'll need a web app to test. You can use an existing app, or deploy one in your continuous deployment (CD) release pipeline. The example code above runs tests against Bing.com. For details of how to set up your own release pipeline to deploy a web app, see Deploy to Azure Web Apps.

Decide how you will deploy and test your app

You can deploy and test your app using either the Microsoft-hosted agent in Azure, or a self-hosted agent that you install on the target servers.

- When using the **Microsoft-hosted agent**, you should use the Selenium web drivers that are pre-installed on the Windows agents (agents named **Hosted VS 20xx**) because they are compatible with the browser versions installed on the Microsoft-hosted agent images. The paths to the folders containing these drivers can be obtained from the environment variables named `IEWebDriver` (Internet Explorer), `ChromeWebDriver` (Google Chrome), and `GeckoWebDriver` (Firefox). The drivers are **not** pre-installed on other agents such as Linux, Ubuntu, and macOS agents. Also see UI testing considerations.
- When using a **self-hosted agent** that you deploy on your target servers, agents must be configured to run interactively with auto-logon enabled. See Build and release agents and UI testing considerations.

Include the test in a release

1. If you don't have an existing release pipeline that deploys your web app:

- Open the **Releases** page in the **Azure Pipelines** section in Azure DevOps or the **Build & Release** hub in TFS (see Web portal navigation) and choose the + icon, then choose **Create release pipeline**.

- Select the **Azure App Service Deployment** template and choose **Apply**.

- In the **Artifacts** section of the **Pipeline** tab, choose + **Add**. Select your build artifacts and choose **Add**.

- Choose the **Continuous deployment trigger** icon in the **Artifacts** section of the **Pipeline** tab. In the Continuous deployment trigger pane, enable the trigger so that a new release is created from every build. Add a filter for the default branch.

- Open the **Tasks** tab, select the **Stage 1** section, and enter your subscription information and the name of the web app where you want to deploy the app and tests. These settings are applied to the **Deploy Azure App Service** task.

2. If you are deploying your app and tests to environments where the target machines that host the agents do not have Visual Studio installed:

   - In the **Tasks** tab of the release pipeline, choose the + icon in the **Run on agent** section. Select the **Visual Studio Test Platform Installer** task and choose **Add**. Leave all the settings at the default values.

You can find a task more easily by using the search textbox.

3. In the **Tasks** tab of the release pipeline, choose the + icon in the **Run on agent** section. Select the **Visual Studio Test** task and choose **Add**.

4. If you added the **Visual Studio Test Platform Installer** task to your pipeline, change the **Test platform version** setting in the **Execution options** section of the **Visual Studio Test** task to **Installed by Tools Installer**.

How do I pass parameters to my test code from a build pipeline?

5. Save the release pipeline and start a new release. You can do this by queuing a new CI build, or by choosing **Create release** from the **Release** drop-down list in the release pipeline.

6. To view the test results, open the release summary from the **Releases** page and choose the **Tests** link.

Next steps

Review your test results

Requirements traceability

Article • 04/29/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Requirements traceability is the ability to relate and document two or more phases of a development process, which can then be traced both forward or backward from its origin. Requirements traceability help teams to get insights into indicators such as **quality of requirements** or **readiness to ship the requirement**. A fundamental aspect of requirements traceability is association of the requirements to test cases, bugs and code changes.

> Read the glossary to understand test report terminology.

Agile teams running automated tests

Agile teams have characteristics including, but not limited to the following

- Faster release cycles
- Continuous testing in a pipeline
- Negligible manual testing footprint; limited to exploratory testing
- High degree of automation

The following sections explore traceability from **Quality**, **Bug** and **Source** standpoints for Agile teams.

Quality traceability

To ensure user requirements meet the quality goals, the requirements in a project can be linked to test results, which can then be viewed on the team's dashboard. This enables end-to-end traceability with a simple way to monitor test results. To link automated tests with requirements, visit test report in build or release.

1. In the results section under **Tests** tab of a build or release summary, select the test(s) to be linked to requirements and choose **Link**.

2. Choose a work item to be linked to the selected test(s) in one of the specified way:

- Choose an applicable work item from the list of suggested work items. The list is based on the most recently viewed and updated work items.
- Specify a work item ID.
- Search for a work item based on the title text.

> The list shows only work items belonging to the Requirements category.

3. Teams often want to pin the summarized view of requirements traceability to a dashboard. Use the Requirements quality widget for this.

4. Configure the **Requirements quality** widget with the required options and save it.

   - **Requirements query**: Select a work item query that captures the requirements, such as the user stories in the current iteration.
   - **Quality data**: Specify the stage of the pipeline for which the requirements quality should be traced.

5. View the widget in the team's dashboard. It lists all the **Requirements** in scope, along with the **Pass Rate** for the tests and count of Failed tests. Selecting a **Failed**

test count opens the **Tests** tab for the selected build or release. The widget also helps to track the requirements without any associated test(s).

Bug traceability

Testing gives a measure of the confidence to ship a change to users. A test failure signals an issues with the change. Failures can happen for many reasons such as errors in the source under test, bad test code, environmental issues, flaky tests, and more. Bugs provide a robust way to track test failures and drive accountability in the team to take the required remedial actions. To associate bugs with test results, visit test report in build or release.

1. In the results section of the **Tests** tab select the tests against which the bug should be created and choose **Bug**. Multiple test results can be mapped to a single bug. This is typically done when the reason for the failures is attributable to a single cause such as the unavailability of a dependent service, a database connection failure, or similar issues.

2. Open the work item to see the bug. It captures the complete context of the test results including key information such as the error message, stack trace, comments, and more.

3. View the bug with the test result, directly in context, within the **Tests** tab. The **Work Items** tab also lists any linked requirements for the test result.

4. From a work item, navigate directly to the associated test results. Both the test case and the specific test result are linked to the bug.

5. In the work item, select **Test case** or **Test result** to go directly to the **Tests** page for the selected build or release. You can troubleshoot the failure, update your analysis in the bug, and make the changes required to fix the issue as applicable. While both the links take you to the **Tests tab**, the default section shown are **History** and **Debug** respectively.

Source traceability

When troubleshooting test failures that occur consistently over a period of time, it is important to trace back to the initial set of changes - where the failure originated. This can help significantly to narrow down the scope for identifying the problematic test or source under test. To discover the first instance of test failures and trace it back to the associated code changes, visit Tests tab in build or release.

1. In the **Tests** tab, select a test failure to be analyzed. Based on whether it's a build or release, choose the **Failing build** or **Failing release** column for the test.

2. This opens another instance of the **Tests** tab in a new window, showing the first instance of consecutive failures for the test.

3. Based on the build or release pipeline, you can choose the timeline or pipeline view to see what code changes were committed. You can analyze the code changes to identify the potential root cause of the test failure.

Traditional teams using planned testing

Teams that are moving from manual testing to continuous (automated) testing, and have a subset of tests already automated, can execute them as part of the pipeline or on demand (see test report). Referred to as **Planned testing**, automated tests can be associated to the test cases in a test plan and executed from **Azure Test Plans**. Once associated, these tests contribute towards the quality metrics of the corresponding requirements.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

Review test results

Article • 12/20/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Automated tests can be configured to run as part of a build or release for various languages. Test reports provide an effective and consistent way to view the tests results executed using different test frameworks, in order to measure pipeline quality, review traceability, troubleshoot failures and drive failure ownership. In addition, it provides many advanced reporting capabilities explored in the following sections.

You can also perform deeper analysis of test results by using the Analytics Service. For an example of using this with your build and deploy pipelines, see Analyze test results.

Read the glossary to understand test report terminology.

Published test results can be viewed in the **Tests** tab in a build or release summary.

Surface test results in the Tests tab

Test results can be surfaced in the **Tests** tab using one of the following options:

- **Automatically inferred test results**. By default, your pipeline can automatically infer the test output for a few popular test runners. This is done by parsing the error logs generated during the build operation and then checking for signatures of test failures. Currently, Azure DevOps supports the following languages and test runners for automatically inferring the test results:
  - JavaScript - Mocha, Jest and Jasmine
  - Python- Unittest

    > **Note**
    >
    > This inferred test report is a limited experience. Some features available in fully-formed test reports are not present here **(more details)**. We recommend that you publish a fully-formed test report to get the full Test and Insights experience in Pipelines. Also see:

  - Publishing fully-formed test reports for JavaScript test runners

    - Publishing fully-formed test reports for Python test runners

- **Test execution tasks**. Built-in test execution tasks such as Visual Studio Test that automatically publish test results to the pipeline, or others such as Ant, Maven, Gulp, Grunt, and Xcode that provide this capability as an option within the task.

- **Publish Test Results task**. Task that publishes test results to Azure Pipelines or TFS when tests are executed using your choice of runner, and results are available in any of the supported test result formats.

- **API(s)**. Test results published directly by using the Test Management API(s).

Surface test information beyond the Tests tab

The **Tests** tab provides a detailed summary of the test execution. This is helpful in tracking the quality of the pipeline, as well as for troubleshooting failures. Azure DevOps also provides other ways to surface the test information:

- The Dashboard provides visibility of your team's progress. Add one or more widgets that surface test related information:
    - Requirements quality
    - Test results trend
    - Deployment status

- Test analytics provides rich insights into test results measured over a period of time. It can help identify problematic areas in your test by providing data such as the top failing tests, and more.

View test results in build

The build summary provides a timeline view of the key steps executed in the build. If tests were executed and reported as part of the build, a test milestone appears in the timeline view. The test milestone provides a summary of the test results as a measure of **pass percentage** along with indicators for **failures** and **aborts** if these exist.

View test results in release

In the pipeline view you can see all the stages and associated tests. The view provides a summary of the test results as a measure of **pass percentage** along with indicators for **failures** and **aborts** if these exist. These indicators are same as in the build timeline view, giving a consistent experience across build and release.

Tests tab

Both the build and release summaries provide details of test execution. Choose **Test summary** to view the details in the **Tests** tab. This page has the following sections

- **Summary**: provides key quantitative metrics for the test execution such as the total test count, failed tests, pass percentage, and more. It also provides differential indicators of change compared to the previous execution.
- **Results**: lists all tests executed and reported as part of the current build or release. The default view shows only the failed and aborted tests in order to focus on tests that require attention. However, you can choose other outcomes using the filters provided.
- **Details**: A list of tests that you can sort, group, search, and filter to find the test results you need.

Select any test run or result to view the details pane that displays additional information required for troubleshooting such as the error message, stack trace, attachments, work items, historical trend, and more.

> **Tip**
>
> If you use the Visual Studio Test task to run tests, diagnostic output logged from tests (using any of Console.WriteLine, Trace.WriteLine or TestContext.WriteLine methods), will appear as an attachment for a failed test.

The following capabilities of the **Tests** tab help to improve productivity and troubleshooting experience.

Filter large test results

Over time, tests accrue and, for large applications, can easily grow to tens of thousands of tests. For these applications with very many tests, it can be hard to navigate through the results to identify test failures, associate root causes, or get ownership of issues. Filters make it easy to quickly navigate to the test results of your interest. You can filter on **Test Name**, **Outcome** (failed, passed, and more), **Test Files** (files holding tests) and **Owner** (for test files). All of the filter criteria are cumulative in nature.

Additionally, with multiple **Grouping** options such as **Test run**, **Test file**, **Priority**, **Requirement**, and more, you can organize the **Results** view exactly as you require.

Test debt management with bugs

To manage your test debt for failing or long running tests you can create a bug or add data to existing bug and all view all associated work items in the work item tab.

Immersive troubleshooting experience

Error messages and stack traces are lengthy in nature and need enough real estate to view the details during troubleshooting. To provide an immersive troubleshooting experience, the **Details** view can be expanded to full page view while still being able to perform the required operations in context, such as bug creation or requirement association for the selected test result.

Troubleshooting data for Test failure

For the test failures, the error messages and stack traces are available for troubleshooting. You can also view all attachments associated with the test failure in the *Attachments* tab.

Test debt management

You can create or add to an existing bug to manage test debt for failures or long running tests. The *Work Items* tab details all bugs and requirements associated with a Test to help you analyze the requirement impact as well know status and who is working on the bug.

Test trends with historical data

History of test execution can provide meaningful insights into reliability or performance of tests. When troubleshooting a failure, it is valuable to know how a test has performed in the past. The **Tests** tab provides test history in context with the test results. The test history information is exposed in a progressive manner starting with the current build pipeline to other branches, or the current stage to other stages, for build and release respectively.

View execution of in-progress tests

Tests, such as integration and functional tests, can run for a long time. Therefore, it is important to see the current or near real-time status of test execution at any given time. Even for cases where tests run quickly, it's useful to know the status of the relevant test result(s) as early as possible; especially when failures occur. The **in-progress** view eliminates the need to wait for test execution to finish. Results are available in near real-time as execution progresses, helping you to take actions faster. You can debug a failure, file a bug, or abort the pipeline.

> **Note**
>
> The feature is currently available for both build and release, using **Visual Studio Test** task in a Multi Agent job. It will be available for Single Agent jobs in a future release.

The view below shows the **in-progress** test summary in a release, reporting the total test count and the number of test failures at a given point in time. The test failures are available for troubleshooting, creating bug(s), or to take any other appropriate action.

View summarized test results

During test execution, a test might spawn multiple instances or tests that contribute to the overall outcome. Some examples are, tests that are rerun, tests composed of an ordered combination of other tests (ordered tests) or tests having different instances based on an input parameter (data driven tests).

As these tests are related, they must be reported together with the overall outcome derived from the individual instances or tests. These test results are reported as a summarized test result in the **Tests** tab:

- **Rerun failed tests**: The ability to rerun failed tests is available in the latest version of the Visual Studio Test task. During a rerun, multiple attempts can be made for a failed test, and each failure could have a different root cause due to the non-deterministic behavior of the test. Test reports provide a combined view for all the attempts of a rerun, along with the overall test outcome as a summarized unit. Additionally the Test Management API(s) now support the ability to publish and query summarized test results.

- **Data driven tests**: Similar to the rerun of failed tests, all iterations of data driven tests are reported under that test in a summarized view. The summarized view is also available for ordered tests (**.orderedtest** in Visual Studio).

> **Note**
>
> Metrics in the test summary section, such as the total number of tests, passed, failed, or other are computed using the root level of the summarized test result.

View aborted tests

Test execution can abort due to several reasons such as bad test code, errors in the source under test, or environmental issues. Irrespective of the reason for the abort, it is important to be able to diagnose the behavior and identify the root cause. The aborted tests and test runs can be viewed alongside the completed runs in the **Tests** tab.

> **Note**
>
> The feature is currently available for both build and release, using the **Visual Studio Test** task in a Multi Agent job or publishing test results using the **Test Management API(s)**. It will be available for Single Agent jobs in a future release.

Automatically inferred test results

Azure DevOps can automatically infer the output of tests that are running in your pipelines for a few supported test frameworks. These automatically inferred test reports require no specific configuration of your pipelines, and are a zero-effort way to get started using Test Reporting.

See the list of runners for which test results are automatically inferred.

As only limited test metadata is present in such inferred reports, they are limited in features and capabilities. The following features are not available for inferred test reports:

- Group the test results by test file, owner, priority, and other fields
- Search and filter the test results
- Check details of passed tests
- Preview any attachments generated during the tests within the web UI itself
- Associate a test failure with a new bug, or see list of associated work items for this failure
- See build-on-build analytics for testing in Pipelines

> **Note**
>
> Some runners such as Mocha have multiple built-in console reporters such as **dot-matrix** and **progress-bar**. If you have configured a non-default console output for your test runner, or you are using a custom reporter, Azure DevOps will not be able to infer the test results. It can only infer the results from the **default** reporter.

Related articles

- Analyze test results
- Trace test requirements
- Review code coverage results

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

Test Analytics

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Tracking test quality over time and improving test collateral is key to maintaining a healthy DevOps pipeline. Test analytics provides near real-time visibility into your test data for builds and releases. It helps improve the efficiency of your pipeline by identifying repetitive, high impact quality issues.

> **Note**
>
> Test analytics is currently available only with Azure Pipelines.

Read the glossary to understand test reports terminology.

Install the Analytics extension if required

For more information, see The Analytics Marketplace extension.

View test analytics for builds

To help teams find and fix tests that fail frequently or intermittently, use the **top failing tests** report. The build summary includes the **Analytics** page that hosts this report. The top-level view provides a summary of the test pass rate and results for the selected build pipeline, for the specified period. The default range is 14 days.

View test analytics for releases

For tests executing as part of release, access test analytics from the **Analytics** link at the top right corner. As with build, the summary provides an aggregated view of the test pass rate and results for the specified period.

Test Failures

Open a build or release summary to view the top failing tests report. This report provides a granular view of the top failing tests in the pipeline, along with the failure details.

The detailed view contains two sections:

- **Summary**: Provides key quantitative metrics for the tests executed in build or release over the specified period. The default view shows data for 14 days.

- Pass rate and results: Shows the pass percentage, along with the distribution of tests across various outcomes.

- Failing tests: Provides a distinct count of tests that failed during the specified period. In the example above, 986 test failures originated from 124 tests.

- Chart view: A trend of the total test failures and average pass rate on each day of the specified period.

- **Results**: List of top failed tests based on the total number of failures. Helps to identify problematic tests and lets you drill into a detailed summary of results.

Test	Failed	Pass rate	Total count	Average duration
ValidateRetentionTabExperienceForTfvcProject	368	75.46%	1500	42.35s
CachingGms_RemoteCacheInvalidation	87	94.2%	1500	20.31s
RemoteSecurityNamespacesServiceBus	25	98.33%	1500	10.32s
ValidateOptionsTabExperience	15	99%	1500	62.9s
ValidatePreCreateAuditNotifications	9	99.28%	1282	15.83s
ProjectRenameTest	10	99.33%	1500	7.86s

Group test failures

The report view can be organized in several different ways using the **group by** option. Grouping test results can provide deep insights into various aspects of the top failing tests. In the example below, the test results are grouped based on the test files they belong to. It shows the test files and their respective contribution towards the total of test failures, during the specified period to help you easily identify and prioritize your next steps. Additionally, for each test file, it shows the tests that contribute to these failures.

Drill down to individual tests

After you have identified one or more tests in the **Details** section, select the individual test you want to analyze. This provides a drill-down view of the selected test with a stacked chart of various outcomes such as passed or failed instances of the test, for each day in the specified period. This view helps you infer hidden patterns and take actions accordingly.

The corresponding grid view lists all instances of execution of the selected test during that period.

Release pipelines > Tfs.SelfHost.CodeDev > Test Failures Report

CachingGms_RemoteCacheInvalidation | Back

14 Days

Outcome	Date	Duration	Branch	Stage
Passed	8/29/2018 8:58 PM	25.73s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 8:50 PM	29.04s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 8:42 PM	12.47s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 8:22 PM	10.41s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 8:10 PM	27.72s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 8:09 PM	12.4s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 8:06 PM	34.26s	refs/heads/master	Tfs.SelfHost.CodeDev
Failed	8/29/2018 8:02 PM	103.95s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 7:58 PM	31.2s	refs/heads/master	Tfs.SelfHost.CodeDev
Failed	8/29/2018 7:54 PM	104.53s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 7:51 PM	25.99s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 7:44 PM	12.42s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 7:32 PM	12.43s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 7:28 PM	29.61s	refs/heads/master	Tfs.SelfHost.CodeDev
Failed	8/29/2018 7:04 PM	109.01s	refs/heads/master	Tfs.SelfHost.CodeDev
Failed	8/29/2018 7:01 PM	111.89s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 6:54 PM	28.01s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 6:54 PM	14.48s	refs/heads/master	Tfs.SelfHost.CodeDev
Failed	8/29/2018 6:50 PM	111.31s	refs/heads/master	Tfs.SelfHost.CodeDev
Failed	8/29/2018 6:48 PM	110.51s	refs/heads/master	Tfs.SelfHost.CodeDev
Passed	8/29/2018 6:47 PM	34.44s	refs/heads/master	Tfs.SelfHost.CodeDev

Failure analysis

To perform failure analysis for root causes, choose one or more instances of test execution in the drill-down view to see failure details in context.

Infer hidden patterns

When looking at the test failures for a single instance of execution, it is often difficult to infer any pattern. In the example below, the test failures occurred during a specific period, and knowing this can help narrow down the scope of investigation.

Another example is tests that exhibit non-deterministic behavior (often referred to as flaky tests). Looking at an individual instance of test execution may not provide any meaningful insights into the behavior. However, observing test execution trends for a period can help infer hidden patterns, and help you resolve the failures.

Report information source

The source of information for test analytics is the set of published test results for the build or release pipeline. These result are accrued over a period of time, and form the basis of the rich insights that test analytics provides.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

Review code coverage results

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Code coverage helps you determine the proportion of your project's code that is actually being tested by tests such as unit tests. To increase your confidence of the code changes, and guard effectively against bugs, your tests should exercise - or cover - a large proportion of your code.

Reviewing the code coverage result helps to identify code path(s) that are not covered by the tests. This information is important to improve the test collateral over time by reducing the test debt.

Example

To view an example of publishing code coverage results for your choice of language, see the **Ecosystems** section of the Pipelines topics. For example, collect and publish code coverage for JavaScript using Istanbul.

View results

The code coverage summary can be viewed on the **Summary** tab on the pipeline run summary.

The results can be viewed and downloaded on the **Code coverage** tab.

> **Note**
>
> In a multi-stage YAML pipeline, the code coverage results are only available after the completion of the entire pipeline. This means that you may have to separate the build stage into a pipeline of its own if you want to review the code coverage results prior to deploying to production.

> **Note**
>
> Merging code coverage results from multiple **test runs** is limited to .NET and .NET Core at present. This will be supported for other formats in a future release.

Artifacts

The code coverage artifacts published during the build can be viewed under the **Summary** tab on the pipeline run summary.

- If you use the Visual Studio Test task to collect coverage for .NET and .NET Core apps, the artifact contains **.coverage** files that can be downloaded and used for further analysis in Visual Studio.

- If you publish code coverage using Cobertura or JaCoCo coverage formats, the code coverage artifact contains an HTML file that can be viewed offline for further analysis.

> **Note**
>
> For .NET and .NET Core, the link to download the artifact is available by choosing the code coverage milestone in the build summary.

Tasks

- Publish Code Coverage Results publishes code coverage results to Azure Pipelines or TFS, which were produced by a build in Cobertura or JaCoCo format.
- Built-in tasks such as Visual Studio Test, .NET Core, Ant, Maven, Gulp, Grunt, and Gradle provide the option to publish code coverage data to the pipeline.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

Code coverage for pull requests

Article • 09/14/2023

Azure DevOps Services

Code coverage is an important quality metric and helps you measure the percentage of your project's code that is being tested. To ensure that quality for your project improves over time (or at the least, does not regress), new code being brought into the system must be well tested. When developers raise pull requests, knowing whether their changes are covered by tests helps plug any testing holes before the changes are merged into the target branch. Repo owners may also want to set policies to prevent merging large untested changes.

Full coverage, diff coverage

Full coverage is when coverage gets measured for the entire codebase of a project. However, in the context of pull requests, developers are focused on the changes they are making and want to know whether the specific lines of code they have added or changed are covered. This type of coverage is **diff coverage**.

Prerequisites

In order to get coverage metrics for a pull request, first configure a pipeline that validates pull requests. In this pipeline, configure the test tool you are using to collect code coverage metrics. Coverage results must then be published to the server for reporting.

To learn more about collecting and publishing code coverage results for the language of your choice, see the Ecosystems section. For example, collect and publish code coverage for .NET core apps.

> **Note**
>
> While you can collect and publish code coverage results for many different languages using Azure Pipelines, the **code coverage for pull requests** feature discussed in this document is currently available only for .NET and .NET core projects using the Visual Studio code coverage results format (file extension .coverage). Support for other languages and coverage formats will be added in future milestones.

Coverage status, details and indicators

Once you have configured a pipeline that collects and publishes code coverage, it posts a code coverage status when a pull request is raised. By default, the server checks for atleast 70% of changed lines being covered by tests. The diff coverage threshold target can be changed to a value of your choice. See the settings configuration section below to learn more about this.

The status check evaluates the diff coverage value for all the code files in the pull request. If you would like to view the % diff coverage value for each of the files, you can turn on details as mentioned in the configuration section. Turning on details posts details as a comment in the pull request.

In the changed files view of a pull request, lines that are changed are also annotated with coverage indicators to show whether those lines are covered.

> ⓘ **Note**
>
> While you can build code from a wide variety of version control systems that Azure Pipelines supports, the **code coverage for pull requests** feature discussed in this document is currently available only for Azure Repos.

Configuring coverage settings

If you would like to change the default settings of the code coverage experience for pull requests, you must include a configuration YAML file named azurepipelines-coverage.yml at the root of your repo. Set the desired values in this file and it will be used automatically the next time the pipeline runs.

The settings that can be changed are:

Setting	Description	Default	Permissible values
status	Indicates whether code coverage status check should be posted on pull requests. Turning this off will not post any coverage checks and coverage annotations will not appear in the changed files view.	on	on, off
target	Target threshold value for diff coverage must be met for a successful coverage status to be posted.	70%	Desired % number
comments	Indicates whether a comment containing coverage details for each code file should be posted in the pull request	off	on, off

Example configuration:

```
YAML
```

```
coverage:
  status:           # Code coverage status will be posted to pull requests
based on targets defined below.
    comments: on    # Off by default. When on, details about coverage for
each file changed will be posted as a pull request comment.
    diff:           # Diff coverage is code coverage only for the lines
changed in a pull request.
      target: 60%   # Set this to a desired percentage. Default is 70
percent
```

More examples with details can be found in the code coverage YAML samples repo.

> **Note**
>
> Coverage indicators light up in the changed files view regardless of whether the pull request comment details are turned on.

> **Tip**
>
> The coverage settings YAML is different from a YAML pipeline. This is because the coverage settings apply to your repo and will be used regardless of which pipeline builds your code. This separation also means that if you are using the classic designer-based build pipelines, you will get the code coverage status check for pull requests.

Protect a branch using a code coverage policy

Code coverage status check for pull requests is only a suggestion for developers and it does not prevent pull requests with low code coverage from being merged into the target branch. If you maintain a repo where you would like to prevent developers from merging changes that do not meet a coverage threshold, you must configure a branch policy using the coverage status check.

> **Tip**
>
> Code coverage status posted from a pipeline follows the naming convention `{name-of-your-pipeline/codecoverage}`.

> **Note**

Branch policies in Azure Repos (even optional policies) prevent pull requests from completing automatically if they fail. This behavior is not specific to code coverage policy.

Starting September 2023, the code coverage policy won't be overridden to Failed if the build fails. This feature will be enabled for all customers.

FAQ

Which coverage tools and result formats can be used for validating code coverage in pull requests?

Code coverage for pull requests capability is currently only available for Visual Studio code coverage (.coverage) formats. This can be used if you publish code coverage using the Visual Studio Test task, the test verb of dotnet core task and the TRX option of the publish test results task. Support for other coverage tools and result formats will be added in future milestones.

If multiple pipelines are triggered when a pull request is raised, will coverage be merged across the pipelines?

If multiple pipelines are triggered when a pull request is raised, code coverage will not be merged. The capability is currently designed for a single pipeline that collects and publishes code coverage for pull requests. If you need the ability to merge coverage data across pipelines, please file a feature request on developer community.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and get support via the Developer Community

Tutorial: Create a multistage pipeline with Azure DevOps

Article • 07/26/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

You can use an Azure DevOps multistage pipeline to divide your CI/CD process into stages that represent different parts of your development cycle. Using a multistage pipeline gives you more visibility into your deployment process and makes it easier to integrate approvals and checks.

In this article, you'll create two App Service instances and build a YAML pipeline with three stages:

- ✓ Build: build the source code and produce a package
- ✓ Dev: deploy your package to a development site for testing
- ✓ Staging: deploy to a staging Azure App Service instance with a manual approval check

In a real-world scenario, you may have another stage for deploying to production depending on your DevOps process.

The example code in this exercise is for a .NET web application for a pretend space game that includes a leaderboard to show high scores. You'll deploy to both development and staging instances of Azure Web App for Linux.

Prerequisites

- A GitHub account where you can create a repository. Create one for free.
- An Azure account with an active subscription. Create an account for free.
- An Azure DevOps organization and project. Create one for free.
- An ability to run pipelines on Microsoft-hosted agents. You can either purchase a parallel job or you can request a free tier.

Fork the project

Fork the following sample repository at GitHub.

```
https://github.com/MicrosoftDocs/mslearn-tailspin-spacegame-web-deploy
```

Create the App Service instances

Before you can deploy your pipeline, you need to first create an App Service instance to deploy to. You'll use Azure CLI to create the instance.

1. Sign in to the Azure portal.

2. From the menu, select **Cloud Shell** and the **Bash** experience.

3. Generate a random number that makes your web app's domain name unique. The advantage of having a unique value is that your App Service instance won't have a name conflict with other learners completing this tutorial.

   code

   ```
   webappsuffix=$RANDOM
   ```

4. Open a command prompt and use a `az group create` command to create a resource group named *tailspin-space-game-rg* that contains all of your App Service instances. Update the `location` value to use your closest region.

   Azure CLI

   ```
   az group create --location eastus --name tailspin-space-game-rg
   ```

5. Use the command prompt to create an App Service plan.

   Azure CLI

   ```
   az appservice plan create \
     --name tailspin-space-game-asp \
     --resource-group tailspin-space-game-rg \
     --sku B1 \
     --is-linux
   ```

6. In the command prompt, create two App Service instances, one for each instance (Dev and Staging) with the `az webapp create` command.

   Azure CLI

```
az webapp create \
  --name tailspin-space-game-web-dev-$webappsuffix \
  --resource-group tailspin-space-game-rg \
  --plan tailspin-space-game-asp \
  --runtime "DOTNET|6.0"

az webapp create \
  --name tailspin-space-game-web-staging-$webappsuffix \
  --resource-group tailspin-space-game-rg \
  --plan tailspin-space-game-asp \
  --runtime "DOTNET|6.0"
```

7. With the command prompt, list both App Service instances to verify that they're running with the `az webapp list` command.

Azure CLI

```
az webapp list \
  --resource-group tailspin-space-game-rg \
  --query "[].{hostName: defaultHostName, state: state}" \
  --output table
```

8. Copy the names of the App Service instances to use as variables in the next section.

Create your Azure DevOps project and variables

Set up your Azure DevOps project and a build pipeline. You'll also add variables for your development and staging instances.

Your build pipeline:

- Includes a trigger that runs when there's a code change to branch
- Defines two variables, `buildConfiguration` and `releaseBranchName`
- Includes a stage named Build that builds the web application
- Publishes an artifact you'll use in a later stage

Add the Build stage

1. Sign-in to your Azure DevOps organization and go to your project.

2. Go to **Pipelines**, and then select **New pipeline**.

3. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.

4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.

5. When you see the list of repositories, select your repository.

6. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.

7. When the **Configure** tab appears, select **Starter pipeline**.

8. Replace the contents of *azure-pipelines.yml* with this code.

yml

```yml
trigger:
- '*'

variables:
  buildConfiguration: 'Release'
  releaseBranchName: 'release'

stages:
- stage: 'Build'
  displayName: 'Build the web application'
  jobs:
  - job: 'Build'
    displayName: 'Build job'
    pool:
      vmImage: 'ubuntu-20.04'
      demands:
      - npm

    variables:
      wwwrootDir: 'Tailspin.SpaceGame.Web/wwwroot'
      dotnetSdkVersion: '6.x'

    steps:
    - task: UseDotNet@2
      displayName: 'Use .NET SDK $(dotnetSdkVersion)'
      inputs:
        version: '$(dotnetSdkVersion)'

    - task: Npm@1
      displayName: 'Run npm install'
      inputs:
        verbose: false

    - script: './node_modules/.bin/node-sass $(wwwrootDir) --output
$(wwwrootDir)'
      displayName: 'Compile Sass assets'
```

```
    - task: gulp@1
      displayName: 'Run gulp tasks'

    - script: 'echo "$(Build.DefinitionName), $(Build.BuildId),
$(Build.BuildNumber)" > buildinfo.txt'
      displayName: 'Write build info'
      workingDirectory: $(wwwrootDir)

    - task: DotNetCoreCLI@2
      displayName: 'Restore project dependencies'
      inputs:
        command: 'restore'
        projects: '**/*.csproj'

    - task: DotNetCoreCLI@2
      displayName: 'Build the project - $(buildConfiguration)'
      inputs:
        command: 'build'
        arguments: '--no-restore --configuration $(buildConfiguration)'
        projects: '**/*.csproj'

    - 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

    - publish: '$(Build.ArtifactStagingDirectory)'
      artifact: drop
```

9. When you're ready, select **Save and run**.

Add instance variables

1. In Azure DevOps, go to **Pipelines** > **Library**.

2. Select + **Variable group**.

3. Under **Properties**, add *Release* for the variable group name.

4. Create a two variables to refer to your development and staging host names. Replace the value `1234` with the correct value for your instance.

Variable name	Example value
WebAppNameDev	tailspin-space-game-web-dev-1234

Variable name	Example value
WebAppNameStaging	tailspin-space-game-web-staging-1234

5. Select **Save** to save your variables.

Add the Dev stage

Next, you'll update your pipeline to promote your build to the *Dev* stage.

1. In Azure Pipelines, go to **Pipelines** > **Pipelines**.

2. Select **Edit** in the contextual menu to edit your pipeline.

3. Update *azure-pipelines.yml* to include a Dev stage. In the Dev stage, your pipeline will:

   - Run when the Build stage succeeds because of a condition
   - Download an artifact from `drop`
   - Deploy to Azure App Service with an Azure Resource Manager service connection

```yml
trigger:
- '*'

variables:
  buildConfiguration: 'Release'
  releaseBranchName: 'release'

stages:
- stage: 'Build'
  displayName: 'Build the web application'
  jobs:
  - job: 'Build'
    displayName: 'Build job'
    pool:
      vmImage: 'ubuntu-20.04'
```

```yaml
      demands:
      - npm

    variables:
      wwwrootDir: 'Tailspin.SpaceGame.Web/wwwroot'
      dotnetSdkVersion: '6.x'

    steps:
    - task: UseDotNet@2
      displayName: 'Use .NET SDK $(dotnetSdkVersion)'
      inputs:
        version: '$(dotnetSdkVersion)'

    - task: Npm@1
      displayName: 'Run npm install'
      inputs:
        verbose: false

    - script: './node_modules/.bin/node-sass $(wwwrootDir) --
output $(wwwrootDir)'
      displayName: 'Compile Sass assets'

    - task: gulp@1
      displayName: 'Run gulp tasks'

    - script: 'echo "$(Build.DefinitionName), $(Build.BuildId),
$(Build.BuildNumber)" > buildinfo.txt'
      displayName: 'Write build info'
      workingDirectory: $(wwwrootDir)

    - task: DotNetCoreCLI@2
      displayName: 'Restore project dependencies'
      inputs:
        command: 'restore'
        projects: '**/*.csproj'

    - task: DotNetCoreCLI@2
      displayName: 'Build the project - $(buildConfiguration)'
      inputs:
        command: 'build'
        arguments: '--no-restore --configuration
$(buildConfiguration)'
        projects: '**/*.csproj'

    - 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
```

```
      - publish: '$(Build.ArtifactStagingDirectory)'
        artifact: drop

  - stage: 'Dev'
    displayName: 'Deploy to the dev environment'
    dependsOn: Build
    condition:  succeeded()
    jobs:
    - deployment: Deploy
      pool:
        vmImage: 'ubuntu-20.04'
      environment: dev
      variables:
      - group: Release
      strategy:
        runOnce:
          deploy:
            steps:
            - download: current
              artifact: drop
            - task: AzureWebApp@1
              displayName: 'Azure App Service Deploy: website'
              inputs:
                azureSubscription: 'your-subscription'
                appType: 'webAppLinux'
                appName: '$(WebAppNameDev)'
                package:
'$(Pipeline.Workspace)/drop/$(buildConfiguration)/*.zip'
```

4. Change the `AzureWebApp@1` task to use your subscription.

   a. Select **Settings** for the task.

   b. Update the `your-subscription` value for **Azure Subscription** to use your own subscription. You may need to authorize access as part of this process. If you run into a problem authorizing your resource within the YAML editor, an alternate approach is to create a service connection.

c. Set the **App type** to Web App on Linux.

d. Select **Add** to update the task.

5. Save and run your pipeline.

Add the Staging stage

Last, you'll promote the Dev stage to Staging. Unlike the Dev environment, you want to have more control in the staging environment you'll add a manual approval.

Create staging environment

1. From Azure Pipelines, select **Environments**.

2. Select **New environment**.

3. Create a new environment with the name *staging* and **Resource** set to *None*.

4. On the staging environment page, select **Approvals and checks**.

5. Select **Approvals**.

6. In **Approvers**, select **Add users and groups**, and then select your account.

7. In **Instructions to approvers**, write *Approve this change when it's ready for staging*.

8. Select **Save**.

Add new stage to pipeline

You'll add new stage, `Staging` to the pipeline that includes a manual approval.

1. Edit your pipeline file and add the `Staging` section.

yml

```yml
trigger:
- '*'

variables:
  buildConfiguration: 'Release'
  releaseBranchName: 'release'

stages:
- stage: 'Build'
  displayName: 'Build the web application'
  jobs:
  - job: 'Build'
    displayName: 'Build job'
    pool:
      vmImage: 'ubuntu-20.04'
      demands:
      - npm

    variables:
      wwwrootDir: 'Tailspin.SpaceGame.Web/wwwroot'
      dotnetSdkVersion: '6.x'

    steps:
    - task: UseDotNet@2
      displayName: 'Use .NET SDK $(dotnetSdkVersion)'
      inputs:
        version: '$(dotnetSdkVersion)'

    - task: Npm@1
      displayName: 'Run npm install'
      inputs:
        verbose: false

    - script: './node_modules/.bin/node-sass $(wwwrootDir) --output
$(wwwrootDir)'
      displayName: 'Compile Sass assets'

    - task: gulp@1
      displayName: 'Run gulp tasks'
```

```yaml
    - script: 'echo "$(Build.DefinitionName), $(Build.BuildId),
$(Build.BuildNumber)" > buildinfo.txt'
      displayName: 'Write build info'
      workingDirectory: $(wwwrootDir)

    - task: DotNetCoreCLI@2
      displayName: 'Restore project dependencies'
      inputs:
        command: 'restore'
        projects: '**/*.csproj'

    - task: DotNetCoreCLI@2
      displayName: 'Build the project - $(buildConfiguration)'
      inputs:
        command: 'build'
        arguments: '--no-restore --configuration $(buildConfiguration)'
        projects: '**/*.csproj'

    - 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

    - publish: '$(Build.ArtifactStagingDirectory)'
      artifact: drop

- stage: 'Dev'
  displayName: 'Deploy to the dev environment'
  dependsOn: Build
  condition:  succeeded()
  jobs:
  - deployment: Deploy
    pool:
      vmImage: 'ubuntu-20.04'
    environment: dev
    variables:
    - group: Release
    strategy:
      runOnce:
        deploy:
          steps:
          - download: current
            artifact: drop
          - task: AzureWebApp@1
            displayName: 'Azure App Service Deploy: website'
            inputs:
              azureSubscription: 'your-subscription'
              appType: 'webAppLinux'
              appName: '$(WebAppNameDev)'
              package:
```

```
'$(Pipeline.Workspace)/drop/$(buildConfiguration)/*.zip'

- stage: 'Staging'
  displayName: 'Deploy to the staging environment'
  dependsOn: Dev
  jobs:
  - deployment: Deploy
    pool:
      vmImage: 'ubuntu-20.04'
    environment: staging
    variables:
    - group: 'Release'
    strategy:
      runOnce:
        deploy:
          steps:
          - download: current
            artifact: drop
          - task: AzureWebApp@1
            displayName: 'Azure App Service Deploy: website'
            inputs:
              azureSubscription: 'your-subscription'
              appType: 'webAppLinux'
              appName: '$(WebAppNameDev)'
              package:
'$(Pipeline.Workspace)/drop/$(buildConfiguration)/*.zip'
```

2. Change the `AzureWebApp@1` task in the Staging stage to use your subscription.

   a. Select **Settings** for the task.

   b. Update the `your-subscription` value for **Azure Subscription** to use your own subscription. You may need to authorize access as part of this process.

   c. Set the **App type** to Web App on Linux.

   d. Select **Add** to update the task.

3. Go to the pipeline run. Watch the build as it runs. When it reaches `Staging`, the pipeline waits for manual release approval. You'll also receive an email that you have a pipeline pending approval.

4. Review the approval and allow the pipeline to run.

Clean up resources

If you're not going to continue to use this application, delete the resource group in Azure portal and the project in Azure DevOps with the following steps:

To clean up your resource group:

1. Go to the Azure portal and sign in.

2. From the menu bar, select Cloud Shell. When prompted, select the **Bash** experience.

3. Run the following az group delete command to delete the resource group that you used, `tailspin-space-game-rg`.

Azure CLI

```
az group delete --name tailspin-space-game-rg
```

To delete your Azure DevOps project, including the build pipeline:

1. In Azure DevOps, navigate to your project.
2. Select **Project settings**.
3. In the **Project details**, select **Delete**.

Related content

- Key concepts for new Azure Pipelines users
- Deploy to App Service using Azure Pipelines

Create and target an environment

Article • 07/17/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

An environment is a collection of resources that you can target with deployments from a pipeline. Typical examples of environment names are Dev, Test, QA, Staging, and Production. An Azure DevOps environment represents a logical target where your pipeline deploys software.

Azure DevOps environments aren't available in classic pipelines. For classic pipelines, deployment groups offer similar functionality.

Environments provide the following benefits.

Benefit	Description
Deployment history	Pipeline name and run details get recorded for deployments to an environment and its resources. In the context of multiple pipelines targeting the same environment or resource, deployment history of an environment is useful to identify the source of changes.
Traceability of commits and work items	View jobs within the pipeline run that target an environment. You can also view the commits and work items that were newly deployed to the environment. Traceability also allows one to track whether a code change (commit) or feature/bug-fix (work items) reached an environment.
Diagnostic resource health	Validate whether the application is functioning at its wanted state.
Security	Secure environments by specifying which users and pipelines are allowed to target an environment.

While an environment is a grouping of resources, the resources themselves represent actual deployment targets. The Kubernetes resource and virtual machine resource types are currently supported.

When you author a YAML pipeline and refer to an environment that doesn't exist, Azure Pipelines automatically creates the environment when the user performing the operation is known and permissions can be assigned. When Azure Pipelines doesn't have information about the user creating the environment (example: a YAML update from an external code editor), your pipeline fails if the environment doesn't already exist.

Prerequisites

- You need to have the **Creator** role for Environments to add an Environment.

Create an environment

1. Sign in to your organization: `https://dev.azure.com/{yourorganization}` and select your project.

2. Select **Pipelines** > **Environments** > **Create environment**.

3. Enter information for the environment, and then select **Create**. Resources can be added to an existing environment later.

Use a Pipeline to create and deploy to environments, too. For more information, see the how-to guide.

> **Tip**
>
> You can create an empty environment and reference it from deployment jobs. This lets you record the deployment history against the environment.

Target an environment from a deployment job

A deployment job is a collection of steps to be run sequentially. A deployment job can be used to target an entire environment (group of resources) as shown in the following YAML snippet. The pipeline will run on the myVM machine because the resource name is specified.

YAML

```yaml
- stage: deploy
  jobs:
  - deployment: DeployWeb
    displayName: deploy Web App
    pool:
      vmImage: 'Ubuntu-latest'
    # creates an environment if it doesn't exist
    environment:
      name: 'smarthotel-dev'
      resourceName: myVM
      resourceType: virtualMachine
    strategy:
      runOnce:
        deploy:
          steps:
          - script: echo Hello world
```

Target a specific resource within an environment from deployment job

You can scope the target of deployment to a particular resource within the environment. Then, you can record deployment history on a specific resource within the environment. The steps of the deployment job *automatically inherit* the service connection details from the resource targeted by the deployment job.

YAML

```yaml
environment:
  name: 'smarthotel-dev.bookings'
strategy:
 runOnce:
   deploy:
     steps:
     - task: KubernetesManifest@0
       displayName: Deploy to Kubernetes cluster
       inputs:
         action: deploy
         namespace: $(k8sNamespace)
         manifests: $(System.ArtifactsDirectory)/manifests/*
         imagePullSecrets: $(imagePullSecret)
         containers: $(containerRegistry)/$(imageRepository):$(tag)
         # value for kubernetesServiceConnection input automatically passed
down to task by environment.resource input
```

Environment in run details

All environments that get targeted by deployment jobs of a specific run of a pipeline can be found under the *Environments* tab of pipeline run details.

If you're using an AKS private cluster, the **Environments** tab isn't available.

Approvals

Manually control when a stage should run using approval checks. Use approval checks to control deployments to production environments. Checks are available to the resource Owner to control when a stage in a pipeline consumes a resource. As the owner of a resource, such as an environment, you can define approvals and checks that must be satisfied before a stage consuming that resource starts.

We support manual approval checks on environments. For more information, see Approvals.

The Creator, Administrator, and user roles can manage approvals and checks. The Reader role can't manage approvals and checks.

Deployment history

The deployment history view within environments provides the following advantages.

- View jobs from all pipelines that target a specific environment. For example, two micro-services, each having its own pipeline, are deploying to the same environment. The deployment history listing helps identify all pipelines that affect this environment and also helps visualize the sequence of deployments by each pipeline.

- Drill down into the job details to see the list of commits and work items that were deployed to the environment. The list of commits and work items are the new items between deployments. Your first listing includes all of the commits and the following listings will just include changes. If multiple commits are tied to the same pull request, you'll see multiple results on the work items and changes tabs.

- If multiple work items are tied to the same pull request, you'll see multiple results on the work items tab.

Security

User permissions

Control who can create, view, use, and manage the environments with user permissions. There are four roles - Creator (scope: all environments), Reader, User, and Administrator. In the specific environment's **user permissions** panel, you can set the permissions that are inherited and you can override the roles for each environment.

1. Go to the specific **Environment** that you want to authorize.

2. Select ⋮ > **Security** to view the settings.
3. Select **User permissions** > **+Add** > **User or group**, and then select a suitable role.

Role	Description
Creator	Global role, available from environments hub security option. Members of this role can create the environment in the project. Contributors are added as members by default. Required to trigger a YAML pipeline when the environment does not already exist.
Reader	Members of this role can view the environment.
User	Members of this role can use the environment when creating or editing YAML pipelines.
Administrator	Members of this role can administer permissions, create, manage, view and use environments. For a particular environment, its creator is added as Admininistrator by default.

Pipeline permissions

Use pipeline permissions to authorize all or selected pipelines for deployment to the environment.

- To remove **Open access** on the environment or resource, select **Restrict permission** in **Pipeline permissions**.
- To allow specific pipelines to deploy to an environment or specific resource, select + and choose from the list of pipelines.

Next steps

Define approvals and checks

FAQ

Q: Why do I get an error message when I try to create an environment?

A: If you see the message "Access denied: {User} needs Create permissions to do the action", check your organization-level permissions. Go to **Organization Settings** > **Users** and check if you have the stakeholder role. The stakeholder role can't create

environments. Change your access level and then check to see if you can create environments. For more information, see User and permissions management FAQ.

Q: Why am I getting error "Job XXXX: Environment XXXX could not be found. The environment does not exist or has not been authorized for use"?

A: These are some of the possible reasons of the failure:

- When you author a YAML pipeline and refer to an environment that doesn't exist in the YAML file, Azure Pipelines automatically creates the environment in some cases:
  - You use the YAML pipeline creation wizard in the Azure Pipelines web experience and refer to an environment that hasn't been created yet.
  - You update the YAML file using the Azure Pipelines web editor and save the pipeline after adding a reference to an environment that does not exist.
- In the following flows, Azure Pipelines doesn't have information about the user creating the environment: you update the YAML file using another external code editor, add a reference to an environment that doesn't exist, and then cause a manual or continuous integration pipeline to be triggered. In this case, Azure Pipelines doesn't know about the user. Previously, we handled this case by adding all the project contributors to the administrator role of the environment. Any member of the project could then change these permissions and prevent others from accessing the environment.
- You can use variables to create the environment or use templateContext to pass properties to templates. Runtime parameters won't work when creating the environment because they are expanded at run time.
- A user with stakeholder access level can't create the environment as stakeholders don't have access to the repository.

Related articles

- Define variables
- Define resources in YAML

Environment - Kubernetes resource

Article • 11/06/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

The Kubernetes resource view shows the status of objects within the namespace that are mapped to the resource. The resource view also overlays pipeline traceability so you can trace back from a Kubernetes object to the pipeline, and then back to the commit.

Use Kubernetes resources to target Kubernetes clusters in an environment for deployment. Use pipelines to deploy to Azure Kubernetes Service (AKS) and clusters from any other cloud provider.

You can use Kubernetes resources with public or private clusters. To learn more about how resources work, see resources in YAML and security with resources.

Overview

See the following advantages of using Kubernetes resource views within environments:

- **Pipeline traceability** - The Kubernetes manifest task, used for deployments, adds more annotations to show pipeline traceability in resource views. Pipeline traceability helps to identify the originating Azure DevOps organization, project, and pipeline responsible for updates that were made to an object within the namespace.

- **Diagnose resource health** - Workload status can be useful for quickly debugging mistakes or regressions that were introduced by a new deployment. For example, for unconfigured *imagePullSecrets* resulting in ImagePullBackOff errors, pod status information can help you identify the root cause for the issue.

- **Review App** - Review App works by deploying every pull request from your Git repository to a dynamic Kubernetes resource under the environment. Reviewers can see how those changes look and work with other dependent services before they're merged into the target branch and deployed to production.

Use Azure Kubernetes Service

A ServiceAccount gets created in your chosen cluster and namespace when you use Azure Kubernetes Service (AKS). For a Kubernetes RBAC-enabled cluster, RoleBinding also gets created to limit the scope of the created service account to the chosen namespace. For a Kubernetes RBAC-disabled cluster, the ServiceAccount created has cluster-wide privileges (across namespaces).

Add an AKS Kubernetes resource

1. In the environment details page, select **Add resource** and choose **Kubernetes**.
2. Select **Azure Kubernetes Service** in the Provider dropdown.
3. Choose the Azure subscription, cluster, and namespace (new/existing).
4. Select **Validate and create** to create the Kubernetes resource.
5. Verify that you see a cluster for your environment. You'll see the text "Never deployed" if you haven't yet deployed code to your cluster.

Use an existing service account

The Azure Kubernetes Service maps a Kubernetes resource within your environment to a namespace.

For more information about setting up a Kubernetes service connection outside of an environment, see the Kubernetes service connection section in Service connections.

> **Tip**
>
> Use the generic provider (existing service account) to map a Kubernetes resource to a namespace from a non-AKS cluster.

Add a non-AKS Kubernetes resource

1. In the environment details page, select **Add resource** and choose **Kubernetes**.
2. Select **Generic provider (existing service account)** for your provider.
3. Add the cluster name and namespace values.
4. Add the server URL. You can get the URL with the following command:

```
kubectl config view --minify -o 'jsonpath=
{.clusters[0].cluster.server}'
```

5. To get the secret object.

Kubernetes 1.22+

Replace `service-account-name` with your account name.

```
kubectl get secret -n <namespace>  -o jsonpath='{.items[?(@.metadata.annotations.kubernetes\.io/service-account\.name==\"service-account-name\")]}'
```

If you get nothing, see Manually create a long-lived API token for a ServiceAccount.

Kubernetes 1.22 and below:

a. Find the service account secret name

```
kubectl get serviceAccounts <service-account-name> -n <namespace> -o 'jsonpath={.secrets[*].name}'
```

b. replace `<service-account-secret-name>` with the value in previous command in this command

```
kubectl get secret <service-account-secret-name> -n <namespace> -o json
```

6. Get the secret object using the output of the previous step.

```
kubectl get secret <service-account-secret-name> -n <namespace> -o json
```

7. Copy and paste the Secret object fetched in JSON form into the Secret field.

8. Select **Validate and create** to create the Kubernetes resource.

Reference your Kubernetes resources in a pipeline

If you're using Azure Kubernetes Service and building a YAML pipeline, the easiest way to configure your pipeline is to use a template. Connect to your repository and select one of the following two Kubernetes Service options:

- Deploy to Azure Kubernetes Services template
- Deploy to Kubernetes - Review App with Azure DevSpaces

The templates let you set up Review App without needing to write YAML code from scratch or manually create explicit role bindings.

Set up Review App

In the following example, the first deployment job is run for non-PR branches and does deployments against a regular Kubernetes resource under environments. The second job runs only for PR branches and deploys against Review App resources (namespaces inside Kubernetes cluster) generated on demand. Resources get labeled with "Review" in the resource listing view of the environment. Define variables to use in the pipeline. If you use the Deploy to Azure Kubernetes Services template, these variables get defined for you.

YAML

```yaml
# Build and push image to Azure Container Registry; Deploy to Azure
Kubernetes Service
trigger:
- main

resources:
- repo: self

variables:

  # Container registry service connection established during pipeline
creation
  dockerRegistryServiceConnection: '12345' # Docker service connection
identifier
  envName: 'myEnv' # name of your environment
  imageRepository: 'name-of-image-repository' # name of image repository
  containerRegistry: 'mycontainer.azurecr.io' # path to container registry
  dockerfilePath: '**/Dockerfile'
  tag: '$(Build.BuildId)'
  imagePullSecret: 'my-app-secret' # image pull secret

  # Agent VM image name
  vmImageName: 'ubuntu-latest'

  # Name of the new namespace being created to deploy the PR changes.
```

```yaml
  k8sNamespaceForPR: 'review-app-$(System.PullRequest.PullRequestId)'

stages:
- stage: Build
  displayName: Build stage
  jobs:
  - job: Build
    displayName: Build
    pool:
      vmImage: $(vmImageName)
    steps:
    - task: Docker@2
      displayName: Build and push an image to container registry
      inputs:
        command: buildAndPush
        repository: $(imageRepository)
        dockerfile: $(dockerfilePath)
        containerRegistry: $(dockerRegistryServiceConnection)
        tags: |
          $(tag)

    - upload: manifests
      artifact: manifests

- stage: Production
  displayName: Deploy stage
  dependsOn: Build

  jobs:
  - deployment: Production
    condition: and(succeeded(),
not(startsWith(variables['Build.SourceBranch'], 'refs/pull/')))
    displayName: Production
    pool:
      vmImage: $(vmImageName)
    environment:
      name: $(envName).$(resourceName)
      resourceType: Kubernetes
    strategy:
      runOnce:
        deploy:
          steps:
          - task: KubernetesManifest@0
            displayName: Create imagePullSecret
            inputs:
              action: createSecret
              secretName: $(imagePullSecret)
              dockerRegistryEndpoint: $(dockerRegistryServiceConnection)

          - task: KubernetesManifest@0
            displayName: Deploy to Kubernetes cluster
            inputs:
              action: deploy
              manifests: |
                $(Pipeline.Workspace)/manifests/deployment.yml
```

```
                $(Pipeline.Workspace)/manifests/service.yml
              imagePullSecrets: |
                $(imagePullSecret)
              containers: |
                $(containerRegistry)/$(imageRepository):$(tag)

  - deployment: DeployPullRequest
    displayName: Deploy Pull request
    condition: and(succeeded(), startsWith(variables['Build.SourceBranch'],
'refs/pull/'))
    pool:
      vmImage: $(vmImageName)

    environment:
      name: $(envName).$(resourceName)
      resourceType: Kubernetes
    strategy:
      runOnce:
        deploy:
          steps:
          - reviewApp: default

          - task: Kubernetes@1
            displayName: 'Create a new namespace for the pull request'
            inputs:
              command: apply
              useConfigurationFile: true
              inline: '{ "kind": "Namespace", "apiVersion": "v1",
"metadata": { "name": "$(k8sNamespaceForPR)" }}'

          - task: KubernetesManifest@0
            displayName: Create imagePullSecret
            inputs:
              action: createSecret
              secretName: $(imagePullSecret)
              namespace: $(k8sNamespaceForPR)
              dockerRegistryEndpoint: $(dockerRegistryServiceConnection)

          - task: KubernetesManifest@0
            displayName: Deploy to the new namespace in the Kubernetes
cluster
            inputs:
              action: deploy
              namespace: $(k8sNamespaceForPR)
              manifests: |
                $(Pipeline.Workspace)/manifests/deployment.yml
                $(Pipeline.Workspace)/manifests/service.yml
              imagePullSecrets: |
                $(imagePullSecret)
              containers: |
                $(containerRegistry)/$(imageRepository):$(tag)

          - task: Kubernetes@1
            name: get
            displayName: 'Get services in the new namespace'
```

```
            continueOnError: true
            inputs:
              command: get
              namespace: $(k8sNamespaceForPR)
              arguments: svc
              outputFormat:
jsonpath='http://{.items[0].status.loadBalancer.ingress[0].ip}:
{.items[0].spec.ports[0].port}'

          # Getting the IP of the deployed service and writing it to a
variable for posting comment
          - script: |
              url="$(get.KubectlOutput)"
              message="Your review app has been deployed"
              if [ ! -z "$url" -a "$url" != "http://:" ]
              then
                message="${message} and is available at $url.<br><br>[Learn
More](https://aka.ms/testwithreviewapps) about how to test and provide
feedback for the app."
              fi
              echo "##vso[task.setvariable variable=GITHUB_COMMENT]$message"
```

To use this job in an **existing** pipeline, the service connection backing the regular Kubernetes environment resource must be modified to "Use cluster admin credentials". Otherwise, role bindings must be created for the underlying service account to the Review App namespace.

Next steps

Build and deploy to Azure Kubernetes Service

Related articles

- Deploy
- Deploy ASP.NET Core apps to Azure Kubernetes Service with Azure DevOps Starter
- REST API: Kubernetes with Azure DevOps

Environment - virtual machine resource

Article • 03/20/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Use virtual machine (VM) resources to manage deployments across multiple machines with YAML pipelines. VM resources let you install agents on your own servers for rolling deployments.

VM resources connect to environments. After you define an environment, you can add VMs to target with deployments. The deployment history view in an environment provides traceability from your VM to your pipeline.

Prerequisites

You must have at least a Basic license and access to the following areas:

- the repository connected to your pipeline
- the VM you want to connect to the environment

For more information about security for Azure Pipelines, see Pipeline security resources.

To add a VM to an environment, you must have the Administrator role for the corresponding deployment pool. A deployment pool is a set of target servers available to the organization. Learn more about deployment pool and environment permissions.

> ⓘ **Note**
>
> If you are configuring a deployment group agent, or if you see an error when registering a VM environment resource, you must set the PAT scope to **All accessible organizations**.

Create a VM resource

> ⓘ **Note**
>
> You can use this same process to set up physical machines with a registration script.

The first step in adding a VM resource is to define an environment.

Define an environment

1. Select **Create environment** or **New environment**, depending on whether it's your first environment.
2. Add a **Name** (required) for the environment and a **Description**.
3. Save the new environment.

Add a resource

1. Select your environment and choose **Add resource**.

2. Select **Virtual machines** for your **Resource** type. Then select **Next**.

3. Choose Windows or Linux for the **Operating System**.

4. Copy the registration script. Your script will be a PowerShell script if you've selected Windows and a Linux script if you've selected Linux.

5. Run the copied script on each of the target virtual machines that you want to register with this environment.

   - If you're installing on Windows, you'll need to run the script an PowerShell administrator.
   - If you're installing on Linux, you'll need to have permission to download and run executable scripts.

   > **Note**
   >
   > - The Personal Access Token (PAT) for the signed-in user gets included in the script. The PAT expires on the day you generate the script.
   > - If your VM already has any other agent running on it, provide a unique name for **agent** to register with the environment.
   > - To learn more about installing the agent script, see **Self-hosted Linux agents** and **Self-hosted Windows agents**. The agent scripts for VM resources are like the scripts for self-hosted agents and you can use the same commands.

6. Once your VM is registered, it appears as an environment resource under the **Resources** tab of the environment.

7. To add more VMs, copy the script again. Select **Add resource** > **Virtual machines**. The Windows and Linux scripts are the same for all the VMs added to the environment.

8. When the VM script is successfully installed, your VM appears in the list of resources for the environment.

Use VM in pipelines

Target VMs in your pipeline by referencing the environment. By default, the pipeline job runs for all of the VMs defined for an environment with a `resourceName`.

YAML

```yaml
trigger:
- main

pool:
   vmImage: ubuntu-latest

jobs:
- deployment: VMDeploy
  displayName: Deploy to VM
  environment:
   name: VMenv
   resourceName: VMenv
   resourceType: virtualMachine
  strategy:
     runOnce:
        deploy:
          steps:
            - script: echo "Hello world"
```

> **Note**
>
> The `resourceType` values are case sensitive. Specifying the incorrect casing will result in no matching resources found in the environment. See the **YAML schema** for more information.

You can select a specific virtual machine from the environment to only receive the deployment by specifying it by its `resourceName`. For example, to target deploying only

to the Virtual Machine resource named `USHAN-PC` in the `VMenv` environment, add the `resourceName` parameter and give it the value of `USHAN-PC`.

YAML

```yaml
trigger:
- main

pool:
   vmImage: ubuntu-latest

jobs:
- deployment: VMDeploy
  displayName: Deploy to VM
  environment:
    name: VMenv
    resourceType: virtualMachine
    resourceName: USHAN-PC # only deploy to the VM resource named USHAN-PC
  strategy:
    runOnce:
      deploy:
          steps:
          - script: echo "Hello world"
```

To learn more about deployment jobs, see the YAML schema.

Add and manage tags

Tags give you a way to target a set of specific VMs in an environment for deployment. You can add tags to the VM as part of the interactive registration script or through the UI. Tags are each limited to 256 characters. There's no limit to the number of tags that you can use.

Add or remove tags in the UI from the resource view by selecting **More actions** ⋮ for a VM resource.

When you select multiple tags, VMs that include all the tags get used in your pipeline. For example, this pipeline targets VMs with both the `windows` and `prod` tags. If a VM only has one of these tags, it's not targeted.

YAML

```yaml
trigger:
- main

pool:
   vmImage: ubuntu-latest

jobs:
- deployment: VMDeploy
  displayName: Deploy to VM
  environment:
    name: VMenv
    resourceType: virtualMachine
    tags: windows,prod # only deploy to virtual machines with both windows
and prod tags
  strategy:
    runOnce:
      deploy:
          steps:
          - script: echo "Hello world"
```

Apply deployment strategy

Apply a deployment strategy to define how your application gets rolled out. The `runOnce` strategy and the `rolling` strategy for VMs are both supported. For more information about deployment strategies and life-cycle hooks, see Deployment jobs/Deployment strategies.

View deployment history

Select the **Deployments** tab for complete traceability of commits and work items, and a cross-pipeline deployment history per environment and resource.

Remove a VM from an environment

Windows environment

To remove VMs from a Windows environment, run the following command. Ensure you do the following tasks:

- Run the command from an administrator PowerShell command prompt
- Run the command on each machine
- Run the command in the same folder path as the environment registration command was run

```
./config.cmd remove
```

Linux environment

To remove a VM from a Linux environment, run the following command on each machine.

```
./config.sh remove
```

Known limitations

When you retry a stage, it reruns the deployment on all VMs and not just failed targets.

Related articles

- About environments
- Learn about deployment jobs
- YAML schema reference

Deploy to a Linux Virtual Machine

Article • 01/24/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Learn how to set up an Azure DevOps pipeline for multi-virtual machine deployments that uses an environment and virtual machine resources.

Use the instructions in this article for any app that publishes a web deployment package.

> **Note**
>
> If you want to deploy your application to a Linux virtual machine using the classic editor, see **Deploy web apps to Linux virtual machines (VMs)**.

Prerequisites

- An Azure account with an active subscription. Create an account for free.
- An active Azure DevOps organization. Sign up for Azure Pipelines.
- A Linux virtual machine (VM) hosted in Azure.
  - To install a JavaScript or Node.js app, set up a Linux VM with Nginx in Azure, see Create a Linux VM with Azure CLI.
  - To deploy a Java Spring Boot and Spring Cloud based apps, create a Linux VM in Azure using Java 13 on Ubuntu 20.04 template, which provides a fully supported OpenJDK-based runtime.

Get your sample code

JavaScript

If you already have an app in GitHub that you want to deploy, you can create a pipeline for that code.

If you are a new user, fork this repo in GitHub:

```
https://github.com/MicrosoftDocs/pipelines-javascript
```

Create an environment with virtual machines

You can add virtual machines as resources within environments and target them for multi-VM deployments. The deployment history view provides traceability from the VM to the commit.

1. Sign into your Azure DevOps organization and navigate to your project.
2. Go to the Pipelines page and select **Environments** > **Create Environment**.
3. Specify a **Name** (required) for the environment and a **Description**.
4. Choose **Virtual Machines** as a **Resource** to be added to the environment, and then select **Next**.
5. Choose Linux for the **Operating System** and copy the registration script.
6. Run the registration script on each of the target VMs registered with the environment.

   > **Note**
   >
   > - The Personal Access Token (PAT) of the signed in user gets pre-inserted in the script and expires after three hours.
   > - If your VM already has any agent running on it, provide a unique name to register with the environment.

7. Once VM is registered, it will start appearing as an environment resource under **Resources**.

8. To add more VMs, copy the script again. Select **Add resource** and choose **Virtual Machines**. This script is the same for all the VMs you want to add to the same environment.

   Each machine interacts with Azure Pipelines to coordinate deployment of your app.

9. You can add or remove tags for the VM. Select on the dots at the end of each VM resource in **Resources**.

   Tags limit deployment to specific VMs when the environment's used in a deployment job. Each tag is limited to 256 characters. There's no limit to the number of tags you can create.

Define your CI build pipeline

You need a CI build pipeline that publishes your web app. You also need a deployment script that can be run locally on the Ubuntu server. Set up a CI build pipeline based on the runtime you want to use.

1. Sign in to your Azure DevOps organization and go to your project.
2. In your project, go to the **Pipelines** page, and then choose the action to create a new pipeline.
3. Select **GitHub** as the location of your source code.

   You may be redirected to GitHub to sign in. If so, enter your GitHub credentials.
4. When the list of repositories appears, select the sample app repository that you want.
5. Azure Pipelines analyzes your repository and recommends a suitable pipeline template.

   JavaScript

   Select the **starter** template and copy this YAML snippet to build a general Node.js project with npm. You'll add to this YAML in future steps.

   YAML

```yaml
trigger:
- main

pool:
  vmImage: ubuntu-latest

stages:
- stage: Build
  displayName: Build stage
  jobs:
  - job: Build
    displayName: Build
    steps:
    - task: NodeTool@0
      inputs:
        versionSpec: '16.x'
      displayName: 'Install Node.js'
    - script: |
        npm install
        npm run build --if-present
        npm run test --if-present
      displayName: 'npm install, build and test'
    - task: ArchiveFiles@2
      displayName: 'Archive files'
      inputs:
        rootFolderOrFile: '$(System.DefaultWorkingDirectory)'
        includeRootFolder: false
        archiveType: zip
        archiveFile:
$(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip
        replaceExistingArchive: true
    - upload:
$(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip
      artifact: drop
```

For more guidance, review the steps mentioned in Build your Node.js app with gulp for creating a build.

- Select **Save and run** > **Commit directly to the main branch** > **Save and run**.

  A new run starts. Wait for the run to complete.

Define CD steps to deploy to the Linux VM

1. Edit your pipeline and add a deployment job by referencing the environment and the VM resources you created earlier. Update `tags` to refer to tags for your virtual

machine.

```yaml
jobs:
- deployment: VMDeploy
  displayName: Web deploy
  environment:
    name:  <environment name>
    resourceType: VirtualMachine
    tags: web1 # Update or remove value to match your tag
  strategy:
```

To learn more about the `environment` keyword and resources targeted by a deployment job, see the YAML schema.

2. Select specific sets of VMs from the environment to receive the deployment by specifying the **tags** that you've defined for each VM in the environment.

   For more information, see the complete YAML schema for deployment job.

3. Specify either `runOnce` or `rolling` as a deployment strategy.

   `runOnce` is the simplest deployment strategy. All the life-cycle hooks, namely `preDeploy` `deploy`, `routeTraffic`, and `postRouteTraffic`, get executed once. Then, either `on:` `success` or `on:` `failure` gets executed.

   See the following example of a deployment job for `runOnce`:

```yaml
jobs:
- deployment: VMDeploy
  displayName: Web deploy
  environment:
    name: <environment name>
    resourceType: VirtualMachine
  strategy:
    runOnce:
      deploy:
        steps:
        - script: echo my first deployment
```

4. See the following example of a YAML snippet for the rolling strategy with a Java pipeline. You can update up to five targets gets in each iteration. `maxParallel` determines the number of targets that can be deployed to, in parallel. The selection accounts for absolute number or percentage of targets that must remain

available at any time, excluding the targets being deployed to. It's also used to determine the success and failure conditions during deployment.

YAML

```yaml
jobs:
- deployment: VMDeploy
  displayName: web
  environment:
    name: <environment name>
    resourceType: VirtualMachine
  strategy:
      rolling:
        maxParallel: 2  #for percentages, mention as x%
        preDeploy:
          steps:
          - download: current
            artifact: drop
          - script: echo initialize, cleanup, backup, install certs
        deploy:
          steps:
          - task: Bash@3
            inputs:
              targetType: 'inline'
              script: |
                # Modify deployment script based on the app type
                echo "Starting deployment script run"
                sudo java -jar
'$(Pipeline.Workspace)/drop/**/target/*.jar'
        routeTraffic:
          steps:
          - script: echo routing traffic
        postRouteTraffic:
          steps:
          - script: echo health check post-route traffic
        on:
          failure:
            steps:
            - script: echo Restore from backup! This is on failure
          success:
            steps:
            - script: echo Notify! This is on success
```

With each run of this job, deployment history gets recorded against the `<environment name>` environment that you've created and registered the VMs.

Pipeline traceability views in environment

The **Deployments** view provides complete traceability of commits and work items, and a cross-pipeline deployment history per environment.

Next steps

Learn more about Jobs

Related articles

- Tasks
- Catalog of Tasks
- Variables
- Triggers
- Troubleshooting.
- YAML schema reference.

Sign your mobile app

Article • 01/25/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

When developing an app for Android or Apple operating systems, you'll eventually need to manage signing certificates, and in the case of Apple apps, provisioning profiles. This article describes how to securely manage them for signing and provisioning your app.

> **Tip**
>
> Use a Microsoft-hosted Linux, macOS, or Windows build agent, or set up your own agent. See **Build and release agents**.

This article covers:

- Sign your Android app
- Sign your Apple iOS, macOS, tvOS, or watchOS app

Sign your Android app

Follow these steps to sign your Android app while keeping your signing certificate secure:

1. First, obtain a keystore file that contains your signing certificate. The Android documentation describes the process of generating a keystore file and its corresponding key.

2. Create your build pipeline from the Android or Xamarin.Android build template. Or, if you already have a build pipeline, add the Android Signing task after the task that builds your APK.

3. Find the Android Signing task's **Sign the APK** checkbox and enable it.

4. Next to the **Keystore file** field, select the settings icon and upload your keystore file to the Secure Files library. During upload, your keystore will be encrypted and securely stored.

5. Once your keystore has been uploaded to the Secure Files library, select it in the **Keystore file** dropdown.

6. Go to the **Variables** tab and add the following variables. In their **Value** column, enter your **Keystore password**, **Key alias**, and **Key password**.

    - **keystore-password**: Password to the unencrypted keystore file. *Be sure to select the **lock** icon.* This will secure your password and obscure it in logs.
    - **key-alias**: The key alias for the signing certificate you generated.
    - **key-password**: The password for the key associated with the specified alias. *Again, be sure to select the **lock** icon.*

7. Go back to the **Tasks** tab and reference the names of your newly created variables in the signing options.

Save your build pipeline, and you're all set! Any build agent will now be able to securely sign your app without any certificate management on the build machine itself.

Sign your Apple iOS, macOS, tvOS, or watchOS app

For your Xcode or Xamarin.iOS build to sign and provision your app, it needs access to your P12 signing certificate and one or more provisioning profiles. The following sections explain how to obtain these files.

Obtain your P12 signing certificate

After creating your development or distribution signing certificate, export it to a `.p12` file using either Xcode or the Keychain Access app on macOS.

1. To export using Xcode 8 or lower, go to **Xcode** > **Preferences...** > **Accounts** and select your Apple Developer account.

2. Select **View Details...**, right-click on the signing identity you wish to export, and select **Export...**.

3. Enter a filename and password. Take note of the password as you'll need it later.

4. Alternatively, follow a similar process using the **Keychain Access** app on macOS or generate a signing certificate on Windows. Use the procedure described in this article if you prefer this method.

Obtain your provisioning profile

You can download your app provisioning profile from the Apple Developer portal, unless your app uses automatic signing. Learn how to download a provisioning profile in the Apple Developer portal.

You can also use Xcode to access those that are installed on your Mac.

1. Using Xcode 8 or lower, go to **Xcode** > **Preferences...** > **Accounts** and select your Apple Developer account.

2. Right-click the provisioning profile you want to use and select **Show in Finder**.

3. Copy the highlighted file from Finder to another location and give it a descriptive filename.

Configure your build

There are two recommended ways for your build to access signing certificates and provisioning profiles for signing and provisioning your app:

1. Installing them during the build
2. Preinstalling them on a macOS build agent

Choose either of the tabs below for details.

Install them during the build

Use this method when you don't have enduring access to the build agent, such as the hosted macOS agents. The P12 certificate and provisioning profile are installed at the beginning of the build and removed when the build completes.

Install the P12 certificate during your build

Visual Editor

1. Add the Install Apple Certificate task to your build before the Xcode or Xamarin.iOS task.
2. Next to the **Certificate (P12)** field, select the settings icon and upload your P12 file to the Secure Files library. During upload, your certificate will be encrypted and securely stored.
3. Once your certificate has been uploaded to the Secure Files library, select it in the **Certificate (P12)** dropdown.
4. Go to the **Variables** tab and add a variable named `P12password`. Set its value to the password of your certificate. *Be sure to select the **lock** icon.* This will secure your password and obscure it in logs.
5. Go back to the **Tasks** tab. In the **Install Apple Certificate** task's settings, reference your newly created variable in the **Certificate (P12) password** field as: `$(P12password)`

Sample YAML

1. Upload your P12 file to the Secure Files library. During upload, your certificate will be encrypted and securely stored.

2. Go to the **Variables** tab and add a variable named `P12password`. Set its value to the password of your certificate. *Be sure to select the **lock** icon.* This will secure your password and obscure it in logs.

3. Add the Install Apple Certificate task to your YAML before the Xcode or Xamarin.iOS task:

YAML

```yaml
- task: InstallAppleCertificate@2
    inputs:
      certSecureFile: 'my-secure-file.p12' # replace my-secure-file.p12 with the name of your P12 file.
      certPwd: '$(P12password)'
```

Install the provisioning profile during your build

Visual Editor

1. Add the Install Apple Provisioning Profile task to your build before the Xcode or Xamarin.iOS task.
2. For the **Provisioning profile location** option, choose **Secure Files** (in YAML, `secureFiles`).

3. Next to the **Provisioning profile** field, select the settings icon and upload your provisioning profile file to the Secure Files library. During upload, your certificate will be encrypted and securely stored.
4. Once your certificate has been uploaded to the Secure Files library, select it in the **Provisioning profile** dropdown.
5. Enable the checkbox labeled **Remove profile after build**. This will ensure that the provisioning profile isn't left on the agent machine.

Sample YAML

1. Upload your provisioning profile to the Secure Files library. During upload, your certificate will be encrypted and securely stored.

2. Add the Install Apple Provisioning Profile task to your YAML before the Xcode or Xamarin.iOS task:

   YAML

   ```yaml
   - task: InstallAppleProvisioningProfile@1
       inputs:
         provProfileSecureFile: 'my-provisioning-
   profile.mobileprovision' # replace my-provisioning-
   profile.mobileprovision with the name of your provisioning profile
   file.
   ```

   > [NOTE] **Remove profile after build** defaults to *true*.

Reference the files in your Xcode task

Visual Editor

1. Select the **Xcode** task.
2. For the **Signing style** option, choose **Manual signing**.
3. In the **Signing identity** field, enter `$(APPLE_CERTIFICATE_SIGNING_IDENTITY)`. This variable is automatically set by the **Install Apple Certificate** task for the certificate you selected.
4. In the **Provisioning profile UUID** field, enter `$(APPLE_PROV_PROFILE_UUID)`. This variable is automatically set by the **Install Apple Provisioning Profile** task for the provisioning profile you selected.

Sample YAML

YAML

```yaml
- task: Xcode@5
  inputs:
    signingOption: 'manual'
    signingIdentity: '$(APPLE_CERTIFICATE_SIGNING_IDENTITY)'
    provisioningProfileUuid: '$(APPLE_PROV_PROFILE_UUID)'
```

Reference the files in your Xamarin.iOS task

Visual Editor

1. Select the **Xamarin.iOS** task.
2. For the **Override using** option, choose **Identifiers**.
3. In the **Signing identity** field, enter `$(APPLE_CERTIFICATE_SIGNING_IDENTITY)`. This variable is automatically set by the **Install Apple Certificate** task for the certificate you selected.
4. In the **Provisioning profile UUID** field, enter `$(APPLE_PROV_PROFILE_UUID)`. This variable is automatically set by the **Install Apple Provisioning Profile** task for the provisioning profile you selected.

Sample YAML

YAML

```yaml
- task: XamariniOS@2
    inputs:
      solutionFile: '**/*.iOS.csproj'
      signingIdentity: '$(APPLE_CERTIFICATE_SIGNING_IDENTITY)'
      signingProvisioningProfileID: '$(APPLE_PROV_PROFILE_UUID)'
```

Save your build pipeline, and you're all set! The build agent will now be able to securely sign and provision your app.

FAQ

Do I need an agent?

You need at least one agent to run your build or release.

I'm having problems. How can I troubleshoot them?

See Troubleshoot Build and Release.

I can't select a default agent pool and I can't queue my build or release. How do I fix this?

See Agent pools.

My NuGet push task is failing with the following error: "Error: unable to get local issuer certificate". How can I fix this?

This can be fixed by adding a trusted root certificate. You can either add the `NODE_EXTRA_CA_CERTS=file` environment variable to your build agent, or you can add the `NODE.EXTRA.CA.CERTS=file` task variable in your pipeline. See Node.js documentation for more details about this variable. See Set variables in a pipeline for instructions on setting a variable in your pipeline.

Classic release pipelines

Article • 06/05/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Classic release pipelines provide developers with a framework for deploying applications to multiple environments efficiently and securely. Using classic release pipelines, you can automate testing and deployment processes, set up flexible deployment strategies, incorporate approval workflows, and ensure smooth application transitions across various stages.

How do release pipelines work

As part of every deployment, Azure Pipelines executes the following steps:

1. **Pre-deployment approval**:

   When a new deployment request is triggered, Azure Pipelines verifies if a predeployment approval is necessary before deploying a release to a stage. If approval is required, it sends email notifications to the relevant approvers.

2. **Queue deployment job**:

   Azure Pipelines schedules the deployment job on an available Agent.

3. **Agent selection**:

   An available agent picks up the deployment job. A release pipeline can be configured to dynamically select a suitable agent during runtime.

4. **Download artifacts**:

   The agent retrieves and downloads all the artifacts specified in the release.

5. **Run the deployment tasks**:

   The agent executes all the tasks in the deployment job.

6. **Generate progress logs**:

   The agent generates comprehensive logs for each deployment step and sends them back to Azure Pipelines.

7. **Post-deployment approval**:

   After the deployment to a stage is finished, Azure Pipelines verifies if a post-deployment approval is necessary for that particular stage. If no approval is needed, or once a required approval is obtained, it proceeds to initiate the deployment to the next stage.

Deployment model

Azure release pipelines support a wide range of artifact sources including Jenkins, Azure Artifacts, and Team City. The following example illustrates a deployment model using Azure release pipelines:

In the following example, the pipeline consists of two build artifacts originating from separate build pipelines. The application is initially deployed to the *Dev* stage and then to two separate *QA* stages. If the deployment is successful in both QA stages, the application will be deployed to *Prod ring 1* and then to *Prod ring 2*. Each production ring represents multiple instances of the same web app, deployed to different locations across the world.

FAQ

Q: How can I edit variables at release time?

A: In the **Variables** tab of your release pipeline, Select the **Settable at release time** checkbox for the variables that you wish to modify when a release is queued.

Subsequently, when generating a new release, you have the ability to modify the values of those variables.

Q: When would it be more appropriate to modify a release instead of the pipeline that defines it?

A: You can edit the approvals, tasks, and variables of a release instance. However, these edits will only apply to that instance. If you want your changes to apply to all future releases, edit the release pipeline instead.

Q: What are the scenarios where the "abandon a release" feature is useful?

A: If you don't plan to reuse the release, or want to prevent it from being used, you can abandon the release as follows **Pipelines** > (...) > **Abandon**. You can't abandon a release when a deployment is in progress, you must cancel the deployment first.

Q: How do I manage the naming of new releases?

A: The default naming convention for release pipelines is sequential numbering, where the releases are named **Release-1**, **Release-2**, and so on. However, you have the flexibility to customize the naming scheme by modifying the release name format mask. In the **Options** tab of your release pipeline, navigate to the **General** page and adjust the **Release name format** property to suit your preferences.

When specifying the format mask, you can use the following predefined variables. Example: The following release name format: *Release $(Rev:rrr) for build $(Build.BuildNumber) $(Build.DefinitionName)* will create the following release: *Release 002 for build 20170213.2 MySampleAppBuild*.

Variable	Description
Rev: rr	An autoincremented number with at least the specified number of digits.
Date / Date: MMddyy	The current date, with the default format **MMddyy**. Any combinations of M/MM/MMM/MMMM, d/dd/ddd/dddd, y/yy/yyyy/yyyy, h/hh/H/HH, m/mm, s/ss are supported.
System.TeamProject	The name of the project to which this build belongs.
Release.ReleaseId	The ID of the release, which is unique across all releases in the project.
Release.DefinitionName	The name of the release pipeline to which the current release belongs.

Variable	Description
Build.BuildNumber	The number of the build contained in the release. If a release has multiple builds, it's the number of the primary build.
Build.DefinitionName	The pipeline name of the build contained in the release. If a release has multiple builds, it's the pipeline name of the primary build.
Artifact.ArtifactType	The type of the artifact source linked with the release. For example, this can be **Azure Pipelines** or **Jenkins**.
Build.SourceBranch	The branch of the primary artifact source. For Git, this is of the form **main** if the branch is **refs/heads/main**. For Team Foundation Version Control, this is of the form **branch** if the root server path for the workspace is **$/teamproject/branch**. This variable is not set for Jenkins or other artifact sources.
Custom variable	The value of a global configuration property defined in the release pipeline. You can update the release name with custom variables using the Release logging commands

Q: How can I define the retention period for my releases?

A: See retention policies to learn how to set up retention policies for your release pipelines.

Related articles

- Deploy pull request Artifacts
- Deploy from multiple branches
- Set up a multi-stage release pipeline

Deploy from multiple branches using Azure Pipelines

Article • 02/11/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Artifact filters can be used with release triggers to deploy from multiple branches. Applying the artifact filter to a specific branch enables deployment to a specific stage when all the conditions are met.

Prerequisites

- A Git repository to create the pipeline. If you don't have one, use the pipelines-dotnet-core sample app.
- A working build for your repository.

Set up a release pipeline

1. Select **Releases** under **Pipelines**. If this is your first release pipeline, select **New Pipeline**, otherwise select **Create a release**.
2. Select **Start with an empty job** when prompted to select a template.
3. Select **Add an artifact** and specify the project, the build pipeline, and the default version. Select **Add** when you are done.

4. Select the **Continuous deployment trigger** icon and enable the **Continuous deployment trigger** to create a release every time a new build is available.

5. Under **Stages**, select the stage and rename it to **Dev**. This stage will be triggered when a build artifact is published from the dev branch.

6. Select the **Pre-deployment conditions** icon in the *Dev* stage and set the deployment trigger to **After release** to trigger a deployment to this stage every time a new release is created.

7. Enable the **Artifact filters**. Select Add and specify your artifact and build branch.

8. Under **Stage**, select **Add** then **New stage** to add a new stage. Select **Start with an empty job** when prompted to select a template, and rename the stage to **Prod**. This stage will be triggered when a build artifact is published from the main branch. Repeat the steps 6-8 and replace the **Build branch** for this stage to main.

9. Add to each stage all the relevant deployment tasks to your environment.

Now the next time you have a successful build, the pipeline will detect which branch triggered that build and trigger deployment to the appropriate stage only.

Related articles

- Release triggers
- Build Artifacts
- Release artifacts and artifact sources
- Publish and download artifacts

Deploy pull request Artifacts with classic release pipelines

Article • 02/15/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Pull requests provide an effective way to review code changes before it is merged into the codebase. However, these changes can introduce issues that can be tricky to find without building and deploying your application to a specific environment. Pull request triggers enable you to set up a set of criteria that must be met before deploying your code. In this article, you will learn how to set up pull request triggers with Azure Repos and GitHub to deploy your build artifact.

Prerequisites

- Source code hosted on Azure Repos or GitHub. Use the pipelines-dotnet-core sample app and create your repository if you don't have one already.
- A working build pipeline for your repository.
- A classic release pipeline. Set up a release pipeline if you don't have one already.

Pull request deployment

With pull request triggers, anytime you raise a new pull request for the designated branch, a release is triggered automatically to start the deployment to the designated environments. The deployment status will then be displayed on the pull request page. Pull request triggers can help you maintain better code quality, release with higher confidence, and discover any issues early on in the development cycle.

Setting up pull request deployments is a two step process, first we must set up a pull request trigger and then set up branch policies (Azure Repos) or status checks (GitHub) for our release pipelines.

Create a pull request trigger

A pull request trigger creates a release every time a new build Artifact is available.

1. Navigate to your Azure DevOps project, select **Pipelines** > **Releases** and then select your release pipeline.

2. Select the **Continuous deployment trigger** icon in the **Artifacts** section.

3. Select the toggle button to enable the **Pull request trigger**.

4. Select your **Target Branch** from the dropdown menu.

5. To deploy your application to a specific stage you need to explicitly opt in that stage. The **Stages** section shows the stages that are enabled for pull request deployments.

To opt-in a stage for pull request deployment, select the **Pre-deployment conditions** icon for your specific stage, and then select **Triggers** > **After release**. Finally select the **Pull request deployment** toggle button to enable it.

> **Important**
>
> Enabling automatic pull request deployments for production stages is not recommended.

Set up branch policies for Azure Repos

You can use branch policies to implement a list of criteria that must be met for a pull request to be merged.

1. Navigate to your project, and then select **Repos** > **Branches** to access the list of branches for your repository.

2. Select the context menu for your appropriate branch `...`, then select **Branch policies**.

3. Select **Add status policy**, then select a **Status to check** from the dropdown menu. Select the status corresponding to your release definition and then select **Save**.

> ⓘ **Note**
>
> The release definition should have run at least once with the pull request trigger enabled in order to get the list of statuses. See **Configure a branch policy for an external service** for more details.

4. With the new status policy added, users won't be able to merge any changes to the target branch without a "succeeded" status is posted to the pull request.

5. You can view the status of your policies from the pull request Overview page. Depending on your policy settings, you can view the posted release status under the **Required**, **Optional**, or **Status** sections. The release status gets updated every time the pipeline is triggered.

Set up status checks for GitHub repositories

Enabling status checks for a GitHub repository allow an administrator to choose which criteria must be met before a pull request is merged into the target branch.

> **Note**
>
> The status checks will be posted on your pull request only after your release pipeline has run at least once with the pull request deployment condition Enabled. See **Branch protection rules** for more details.

You can view your status checks in your pull request under the **Conversation** tab.

Related articles

- Release triggers
- Deploy from multiple branches
- Supported source repositories

Define your Classic pipeline

Article • 04/05/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Azure Pipelines provide a highly configurable and manageable pipeline for releases to multiple stages such as development, staging, QA, and production. it also offers the opportunity to implement gates and approvals at each specific stage.

In this tutorial, you will learn about:

- ✓ Continuous deployment triggers
- ✓ Adding stages
- ✓ Adding pre-deployment approvals
- ✓ Creating releases and monitoring deployments

Prerequisites

You'll need:

- A release pipeline that contains at least one stage. If you don't already have one, you can create it by working through any of the following quickstarts and tutorials:
  - Deploy to an Azure Web App
  - Azure DevOps Project
  - Deploy to IIS web server on Windows
- Two separate targets where you will deploy the app. These could be virtual machines, web servers, on-premises physical deployment groups, or other types of deployment target. In this example, we are using Azure App Service website instances. If you decide to do the same, you will have to choose names that are unique, but it's a good idea to include "QA" in the name of one, and "Production" in the name of the other so that you can easily identify them. Use the Azure portal to create a new web app.

Continuous deployment (CD) triggers

Enabling continuous deployment trigger will instruct the pipeline to automatically create a new release every time a new build is available.

1. In **Azure Pipelines**, open the **Releases** tab. Select your release pipeline select **Edit**.

2. Select the **Continuous deployment trigger** icon in the **Artifacts** section to open the trigger panel. Make sure this is enabled so that a new release is created after every new successful build is completed.

3. Select the **Pre-deployment conditions** icon in the **Stages** section to open the conditions panel. Make sure that the trigger for deployment to this stage is set to **After release**. This means that a deployment will be initiated automatically when a new release is created from this release pipeline.

You can also set up Release triggers, Stage triggers or schedule deployments.

Add stages

In this section, we will add two new stages to our release pipeline: QA and production (Two Azure App Services websites in this example). This is a typical scenario where you would deploy initially to a test or staging server, and then to a live or production server. Each stage represents one deployment target.

1. Select the **Pipeline** tab in your release pipeline and select the existing stage. Change the name of your stage to **Production**.

2. Select the + **Add** drop-down list and choose **Clone stage** (the clone option is available only when an existing stage is selected).

Typically, you want to use the same deployment methods with a test and a production stage so that you can be sure your deployed apps will behave the same way. Cloning an existing stage is a good way to ensure you have the same settings for both. You then just need to change the deployment targets.

3. Your cloned stage will have the name **Copy of Production**. Select it and change the name to **QA**.

4. To reorganize the stages in the pipeline, select the **Pre-deployment conditions** icon in your **QA** stage and set the trigger to **After release**. The pipeline diagram will then show the two stages in parallel.

5. Select the **Pre-deployment conditions** icon in your **Production** stage and set the trigger to **After stage**, then select **QA** in the **Stages** drop-down list. The pipeline diagram will now indicate that the two stages will execute in the correct order.

> ⓘ **Note**
>
> You can set up your deployment to start when a deployment to the previous stage is *partially* successful. This means that the deployment will continue even if a specific non-critical task have failed. This is usually used in a fork and join deployments that deploy to different stages in parallel.

6. Select the **Tasks** drop-down list and select the **QA** stage.

7. Depending on the tasks that you are using, change the settings so that this stage deploys to your "QA" target. In our example, we will be using **Deploy Azure App Service** task as shown below.

Add Pre-deployment approvals

The release pipeline we previously modified deploys to QA and production. If the deployment to QA fails, then deployment to production won't trigger. It is recommended to always verify if your app is working properly in QA or test stage before deploying to production. Adding approvals will ensure all the criteria are met before deploying to the next stage. To add approvals to your pipeline follow the steps below:

1. Select the **Pipeline** tab, **Pre-deployment conditions** icon then **Pre-deployment approvers**.

2. In the **Approvers** text box, enter the user(s) that will be responsible for approving the deployment. It is also recommended to uncheck the **The user requesting a**

release or deployment should not approve it check box.

You can add as many approvers as you need, both individual users and organization groups. It's also possible to set up post-deployment approvals by selecting the "user" icon at the right side of the stage in the pipeline diagram. For more information, see Releases gates and approvals.

3. Select **Save**.

Create a release

Now that the release pipeline setup is complete, it's time to start the deployment. To do this, we will manually create a new release. Usually a release is created automatically when a new build artifact is available. However, in this scenario we will create it manually.

1. Select the **Release** drop-down list and choose **Create release**.

2. Enter a description for your release, check that the correct artifacts are selected, and then select **Create**.

3. A banner will appear indicating that a new release has been create. Select the release link to see more details.

4. The release summary page will show the status of the deployment to each stage.

Other views, such as the list of releases, also display an icon that indicates approval is pending. The icon shows a pop-up containing the stage name and more details when you point to it. This makes it easy for an administrator to see which releases are awaiting approval, as well as the overall progress of all releases.

5. Select the *pending_approval* icon to open the approval window panel. Enter a brief comment, and select **Approve**.

> **Note**
>
> You can schedule deployment at a later date, for example during non-peak hours. You can also reassign approval to a different user. Release administrators can access and override all approval decisions.

Monitor and track deployments

Deployment logs help you monitor and debug the release of your application. To check the logs of our deployment follow the steps below:

1. In the release summary, hover over a stage and select **Logs**.

During deployment, you can still access the logs page to see the live logs of every task.

2. Select any task to see the logs for that specific task. This makes it easier to trace and debug deployment issues. You can also download individual task logs, or a zip of all the log files.

3. If you need additional information to debug your deployment, you can run the release in debug mode.

Next step

Use approvals and gates to control your deployment

Stage templates

Article • 04/01/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Azure Pipelines provide a list of stage templates you can choose from when creating a new release pipeline or adding a stage to your existing one. The templates are predefined with the appropriate tasks and settings to help you save time and effort when creating your release pipeline.

Aside from the predefined templates, you can also create your own custom stage templates based on your specific needs.

When a stage is created from a template, the tasks in the template are copied over to the stage. Any further updates to the template have no impact on existing stages. If you are trying to add multiple stages to your release pipeline and update them all in one operation, you should use task groups instead.

> **Note**
>
> Templates cannot be restrict to specific users or groups. All templates, predefined or custom, are available to all users who have the permission to create release pipelines.

Save a stage template

You can save a stage template from within your classic release pipeline.

1. Select your release pipeline, and then select **Edit**.
2. Select the stage you want to export.
3. Select the three dots button, and then select **Save as template**.

4. Name your template, and then select **Ok** when you are done.

Use a stage template

1. From within your release pipeline definition, select **Add** to add a stage.

2. Select **New stage**.

3. Use the search bar to search for your custom template. Select **Add** to use your custom template.

FAQs

Q: Can I export templates or share them with other subscriptions, enterprises, or projects?

Custom templates are scoped to the project that hosts them. Templates cannot be exported or shared with other projects, collections, servers, or organizations. You can, however, export a release pipeline into another project, collection, server, or subscription and then re-create the template and use it in that new location.

Q: How do I delete a custom stage template?

Existing custom templates can be deleted from the *Select a Template* window panel. From within your release pipeline definition, select **Add** > **New Stage** to access the list of templates.

Q: How do I update a custom stage template?

To update a stage template, delete the existing one from the list of templates, and then save the new one with the same name.

Related articles

- Deploy pull request Artifacts .
- Deploy from multiple branches.
- View release progress and test summary.

Deploy apps to a Windows Virtual Machine

Article • 09/29/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Learn how to use Azure Pipelines to build and deploy your ASP.NET, ASP.NET Core, or Node.js web app to an IIS web server running on a Windows Virtual Machine.

Prerequisites

- An Azure DevOps Organization. Create an organization, if you don't have one already.
- Build pipeline
- Configure IIS web server

Build Pipeline

Set up a build pipeline if you don't have one already.

.NET

- Build ASP.NET apps

Configure IIS web server

Configure your IIS server if you haven't done so already

.NET

In an elevated PowerShell command prompt window, run the following command to install IIS:

PowerShell

```
Install-WindowsFeature Web-Server,Web-Asp-Net45,NET-Framework-Features
```

Create a deployment group

Deployment groups make it easier to organize the servers that you want to use to host your app. A deployment group is a collection of machines with an Azure Pipelines agent on each of them. Each machine interacts with Azure Pipelines to coordinate the deployment of your app.

1. Select **Pipelines**, and then select **Deployment groups**.
2. Select **Add a deployment group** (or **New** if there are already deployment groups in place).
3. Enter a name for your group, and then select **Create**.
4. In the machine registration section, make sure that **Windows** is selected from the dropdown menu, and that the **Use a personal access token in the script for authentication** checkbox is also selected. Select **Copy script to clipboard** when you are done. The script that you've copied to your clipboard will download and configure an agent on the VM so that it can receive new web deployment packages and apply them to IIS.
5. Log in to your VM, open an elevated PowerShell command prompt window and run the script.
6. When you're prompted to configure tags for the agent, press Enter to skip. (tags are optional)
7. When you're prompted for the user account, press **Enter** to accept the defaults.

   > **Note**
   >
   > The agent running the pipeline must have access to the *C:\Windows\system32\inetsrv* directory. See **Security groups, service accounts, and permissions** for more details.

8. You should see the following message when the script is done *Service vstsagent.account.computername started successfully*.
9. Navigate to **Deployment groups**, and then select your deployment group. Select the **Targets** tab and make sure your VM is listed.

Create a release pipeline

Using release pipelines, you can deploy your build artifacts to your IIS servers.

1. Select **Pipelines**, and then select **Releases**. Select **New pipeline**.

2. Use the search bar to find the **IIS Website Deployment** template, and then select **Apply**.

3. Select **Add an artifact** to add your build artifact.

4. Select **Build**, and then select your **Project** and your **Source (build pipeline)**. Select **Add** when you are done.

5. Select the **Continuous deployment trigger** icon in the **Artifacts** section. Enable the **Continuous deployment trigger**, and add the **main** branch as a filter.

6. Select **Tasks**, and then select **IIS Deployment**. Select the deployment group you created earlier from the dropdown menu.

7. Select **Save** when you are done.

Deploy your app

1. Select **Pipelines** > **Releases**, and then select **Create release**.

2. Check that the artifact version you want to use is selected and then select **Create**.

3. Select the release link in the information bar message. For example: "Release **Release-1** has been queued".

4. Navigate to your pipeline **Logs** to see the logs and agent output.

Related articles

- Deploy from multiple branches
- Deploy pull request Artifacts

Deploy to Azure

Article • 05/09/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Azure Pipelines combines continuous integration (CI) and continuous delivery (CD) to test and build your code and ship it to any target. While you don't have to use Azure services with Pipelines, Pipelines can help you take advantage of Azure. You can use Pipelines to integrate your CI/CD process with most Azure services.

To learn more about selecting an Azure service for hosting your application code, see Choose an Azure compute service for your application.

If you're just getting started, we recommend you review and get started with the following resources.

Azure service	Integration points
Azure DevOps Projects/Azure DevOps Starter	Start using Azure Pipelines to automate the setup of a CI/CD of your application to Azure. Choose where to deploy your application such as Virtual Machines, Azure App Service, Azure Kubernetes Services (AKS), Azure SQL Database, or Azure Service Fabric. To learn more, see Overview of DevOps Starter.
Azure portal	The Azure portal is a web-based, unified console from which you can build, manage, and monitor everything from simple web apps to complex cloud deployments. Also, you can create custom dashboards for an organized view of resources and configure accessibility options. If you have an Azure DevOps Services organization, you have access to the Azure portal. Sign in to your Azure portal.
DevOps solutions on Azure	Use end-to-end solutions on Azure to implement DevOps practices throughout application planning, development, delivery, and operations. Apply the right combination of DevOps technologies, culture, and processes to enable continual

software delivery and better value for customers. Get started with the following Learn modules:

- Deploy applications with Azure DevOps
- Build applications with Azure DevOps
- Deploy and maintain cloud-native apps with GitHub actions and Azure Pipelines
- Load test Azure web apps by using Azure DevOps

Follow the links provided in the following table to learn more about the Azure services that support continuous integration (CI) and continuous delivery (CD) using Azure Pipelines. For a complete list of Azure Pipelines tasks, see Build and release tasks.

Azure service	Integration points
Azure App Service	An HTTP-based service for hosting web applications, REST APIs, and mobile back ends; the Azure App Service employs Azure Pipelines to deliver CI/CD. To learn more, see: • App Service overview • Deploy an Azure Web App • Use CI/CD to deploy a Python web app to Azure App Service on Linux • Continuously deploy from a Jenkins build • Azure App Service Deploy task • Azure App Service Manage task • Azure App Service Settings task
Azure App Configuration	Service to centrally manage application settings and feature flags. To learn more, see the following articles: • Push settings to App Configuration with Azure Pipelines • Pull settings to App Configuration with Azure Pipelines.
Azure Blob Storage Azure Storage	Store and access unstructured data at scale using Azure Pipelines and Azure Blob Storage.
Azure Static Web Apps	Use Azure Static Web Apps to automatically build and deploy a full stack web app to

Azure from a code repository.

- Tutorial: Publish Azure Static Web Apps with Azure DevOps

Azure Container Registry

Build, store, secure, scan, replicate, and manage container images and artifacts. For example, build and publish a private Docker registry service. To learn more, see Build and push Docker images to Azure Container Registry.

Azure Databases

Azure SQL Database
Azure Database for MySQL
Azure Cosmos DB

Use Azure Pipelines to deploy to Azure SQL Database, Azure Database for MySQL, or Azure Cosmos DB. To learn more, see the following articles:

- Deploy to Azure SQL Database
- Azure SQL Database Deployment task
- Azure Database for MySQL Deployment task
- Quickstart: Deploy to Azure MySQL
- Set up a CI/CD pipeline with the Azure Cosmos DB Emulator build task in Azure DevOps

---

Azure Databricks
Azure Data Factory Azure Machine Learning

Configure a pipeline to integrate with a fully managed, serverless data integration service and unlock insights from all your data. Create an Azure Pipeline that builds and deploys a machine learning model as a web service and to automate the machine learning lifecycle. To learn more, see the following resources:

- Build a data pipeline by using Azure Data Factory, DevOps, and machine learning; Configure Azure Databricks and Azure Data Factory
- DevOps for Azure Databricks
- Prepare data, train, deploy, and monitor machine learning models with Azure Pipelines.

Azure DevTest Labs

Quickly provision development and test stages using reusable templates. To learn more, see Manage a virtual machine in Azure DevTest Labs.

Azure Functions

Provides a fully managed Platform as a service (PaaS) to implement serverless architecture. To learn more, see:

- Deploy an Azure Function
- Azure Function App task
- Azure Function App for Containers task

Azure Government

Use Azure Pipelines to set up CI/CD of your web app running in Azure Government.To learn more, see Deploy an app in Azure Government with Azure Pipelines.

Azure IoT Edge

Use Azure Pipelines to managed services built on Azure IoT Hub. To learn more, see Continuous integration and continuous deployment to Azure IoT Edge devices and Create a CI/CD pipeline for IoT Edge with Azure DevOps Starter.

Azure Key Vault

Use Azure Pipelines to managed services for storing secret data. To learn more, see Use Azure Key Vault secrets in Azure Pipelines and Azure Key Vault task.

Azure Kubernetes Services (AKS)

Deploy and manage containerized applications with a fully managed Kubernetes service. To learn more, see Build and deploy to Azure Kubernetes Service.

Azure Monitor

Configure alerts on available metrics for an Azure resource. Observe the configured Azure monitor rules for active alerts in a release pipeline. Define pre or post-deployment gates based on Query Azure Monitor Alerts. For details, see the following articles:

- Define approvals and checks, Query Azure Monitor Alerts
- Release deployment control using gates
- Azure Monitor Alerts task
- Query Azure Monitor Alerts task.

Azure Policy

Manage and prevent IT issues by using policy definitions that enforce rules and effects

for your resources. To learn how, see Check policy compliance with gates.

Azure Resource Manager (ARM)
ARM Templates

Use ARM templates to define the infrastructure and dependencies and streamline authentication to deploy your app using Azure Pipelines. Specifically, you can:

- Create an Azure Resource Manager service connection using automated security
- Create an Azure Resource Manager service connection with an existing service principal
- Create an Azure Resource Manager service connection to a VM with a managed service identity
- Connect to an Azure Government Cloud
- Connect to Azure Stack
  To learn more, see Connect to Microsoft Azure.

Azure Service Bus

In a release pipeline, send a message to an Azure Service Bus using a service connection. To learn more, see Publish To Azure Service Bus task and Manage service connections, Azure Service Bus service connection.

Azure Service Fabric

Distributed systems platform that can run in many environments, including Azure or on-premises. To learn more, see the following articles: Tutorial: Deploy an application with CI/CD to a Service Fabric cluster and Service Fabric Application Deployment task.

Azure Stack

Build, deploy, and run hybrid and edge computing apps consistently across your ecosystems. To learn more, see Deploy to Azure Stack Hub App Service using Azure Pipelines.

Azure Virtual Machines
Azure Virtual Machine Scale Sets

Simplify continuous delivery to Azure VMs using Azure Pipelines. To learn more, see these articles:

- Build an Azure virtual machine using an Azure RM template
- Deploy to Azure VMs using deployment groups in Azure Pipelines
- Tutorial: Deploy a Java app to a Virtual Machine Scale Set

Azure WebApps

Use publish profile to deploy Azure WebApps for Windows from the Deployment Center. To learn more, see the following articles:

- Deploy an Azure Web App
- Deploy an Azure Web App Container
- Azure App Service Deploy task
- Azure App Service Manage task
- Azure App Service Settings task

Connect to Microsoft Azure with an ARM service connection

Article • 10/16/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

You can use an Azure Resource Manager (ARM) service connection to connect to Azure resources with Service Principal Authentication (SPA) or an Azure-Managed Service Identity. With an Azure Resource Manager service connection, you can use a pipeline to deploy to Azure resources like an App Service app without needing to reauthenticate each time.

There are multiple options for connecting to Azure with Azure Resource Manager service connections:

- service principal or managed identity with workload identity federation
- service principal with secret
- agent-assigned managed identity

> For other types of connection, and general information about creating and using connections, see Service connections for builds and releases.

Create an Azure Resource Manager service connection using workload identity federation

> **Important**
>
> Workload identity federation for Azure Resource Manager is currently in public preview. See the **Supplemental Terms of Use for Microsoft Azure Previews** for legal terms that apply to Azure features that are in beta, preview, or otherwise not yet released into general availability.
>
> This feature is in public preview and may not be available in your organization. To request early access, go to **https://aka.ms/azdo-rm-workload-identity-preview**.

Workload identity federation uses OpenID Connect to authenticate with Microsoft Entra protected resources without needing to manage secrets.

We recommend this approach if:

- You have the Owner role for your Azure subscription.
- You're not connecting to Azure Stack or an Azure Government Cloud.
- You're not connecting from Azure DevOps Server 2019 or earlier versions of TFS.
- Any Marketplace extensions tasks used have been updated to support workload identity federation.

Create a new workload identity federation service connection

1. In Azure DevOps, open the **Service connections** page from the project settings page.

2. Choose + **New service connection** and select **Azure Resource Manager**.

3. Select **Workload identity federation (automatic)**.

4. Specify the following parameters.

Parameter	Description
Subscription	Select an existing Azure subscription. If you don't see any Azure subscriptions or instances, see Troubleshoot Azure Resource Manager service connections.
Resource Group	Leave empty to allow users to access all resources defined within the subscription, or select a resource group to which you want to restrict users' access (users will be able to access only the resources defined within that group).
Service connection name	Required. The name you will use to refer to this service connection in task properties. This is not the name of your Azure subscription.

5. After the new service connection is created, copy the connection name into your code as the `azureSubscription` value.

6. To deploy to a specific Azure resource, the task will need additional data about that resource. Go to the resource in the Azure portal, and then copy the data into your code. For example, to deploy a web app, you would copy the name of the App Service into the `WebAppName` value.

Convert an existing ARM service connection to use workload identity federation

1. In Azure DevOps, open the **Service connections** page from the project settings page.

2. Go to **Pipelines** > **Service connections** and open an existing service connection.

3. Select the service connection you want to convert to use workload identity.

4. Select **Convert**.

5. Select **Convert** again to confirm that you want to create a new service connection. You will have seven days to revert the connection. The conversion may take a few

minutes to process. Once the process completes, you'll be able to use the new service connection.

Revert an existing ARM service connection to use a service principal secret

You can revert a converted service connection with its secret for seven days. After seven days, you'll need to manually create a new secret.

To revert a connection:

1. In Azure DevOps, open the **Service connections** page from the project settings page.
2. Go to **Pipelines** > **Service connections** and open an existing service connection.
3. Select the service connection you want to revert.
4. Select **Revert conversion to the original scheme**.

5. Select **Revert** again to confirm your choice.

Create an Azure Resource Manager service connection using a service principal secret

We recommend this simple approach if:

- You're signed in as the owner of the Azure Pipelines organization and the Azure subscription.
- You don't need to further limit the permissions for Azure resources accessed through the service connection.
- You're not connecting to Azure Stack or an Azure Government Cloud.

- You're not connecting from Azure DevOps Server 2019 or earlier versions of TFS

1. In Azure DevOps, open the **Service connections** page from the project settings page. In TFS, open the **Services** page from the "settings" icon in the top menu bar.

2. Choose + **New service connection** and select **Azure Resource Manager**.

3. Specify the following parameters.

Parameter	Description
Connection Name	Required. The name you will use to refer to this service connection in task properties. This is not the name of your Azure subscription.
Scope level	Select Subscription or Management Group. Management groups are containers that help you manage access, policy, and compliance across multiple subscriptions.
Subscription	If you selected Subscription for the scope, select an existing Azure subscription. If you don't see any Azure subscriptions or instances, see Troubleshoot Azure Resource Manager service connections.
Management Group	If you selected Management Group for the scope, select an existing Azure management group. See Create management groups.
Resource Group	Leave empty to allow users to access all resources defined within the subscription, or select a resource group to which you want to restrict

Parameter	Description
	users' access (users will be able to access only the resources defined within that group).

4. After the new service connection is created:

   - If you're using the classic editor, select the connection name you assigned in the **Azure subscription** setting of your pipeline.
   - If you're using YAML, copy the connection name into your code as the `azureSubscription` value.

5. To deploy to a specific Azure resource, the task will need additional data about that resource.

   - If you're using the classic editor, select data you need. For example, the App Service name.
   - If you're using YAML, then go to the resource in the Azure portal, and then copy the data into your code. For example, to deploy a web app, you would copy the name of the App Service into the `WebAppName` value.

> **Note**
>
> When you follow this approach, Azure DevOps *connects with Microsoft Entra ID and creates an app registration with a secret that's valid for two years*. When the service connection is close to two years old, Microsoft Entra ID displays this prompt: **A certificate or secret is expiring soon. Create a new one**. In this scenario, you must refresh the service connection.
>
> To refresh a service connection, in the Azure DevOps portal, edit the connection and select **Verify**. After you save the edit, the service connection is valid for another two years.

See also: Troubleshoot Azure Resource Manager service connection.

If you have problems using this approach (such as no subscriptions being shown in the drop-down list), or if you want to further limit users' permissions, you can instead use a service principal or a VM with a managed service identity.

Create an Azure Resource Manager service connection with an existing service principal

1. If you want to use a predefined set of access permissions, and you don't already have a suitable service principal defined, follow one of these tutorials to create a new service principal:

    - Use the portal to create a Microsoft Entra application and a service principal that can access resources
    - Use Azure PowerShell to create an Azure service principal with a certificate

2. In Azure DevOps, open the **Service connections** page from the project settings page. In TFS, open the **Services** page from the "settings" icon in the top menu bar.

3. Choose + **New service connection** and select **Azure Resource Manager**.

4. Choose Service Principal (manual) option and enter the Service Principal details.

5. Enter a user-friendly **Connection name** to use when referring to this service connection.

6. Select the **Environment** name (such as Azure Cloud, Azure Stack, or an Azure Government Cloud).

7. If you *do not* select **Azure Cloud**, enter the Environment URL. For Azure Stack, this will be something like `https://management.local.azurestack.external`

8. Select the **Scope level** you require:

    - If you choose **Subscription**, select an existing Azure subscription. If you don't see any Azure subscriptions or instances, see Troubleshoot Azure Resource Manager service connections. |
    - If you choose **Management Group**, select an existing Azure management group. See Create management groups. |

9. Enter the information about your service principal into the Azure subscription dialog textboxes:

    - Subscription ID

- Subscription name
- Service principal ID
- Either the service principal client key or, if you have selected **Certificate**, enter the contents of both the certificate and private key sections of the *.pem file.
- Tenant ID

You can obtain this information if you don't have it to hand by downloading and running this PowerShell script in an Azure PowerShell window. When prompted, enter your subscription name, password, role (optional), and the type of cloud such as Azure Cloud (the default), Azure Stack, or an Azure Government Cloud.

10. Choose **Verify connection** to validate the settings you entered.

11. After the new service connection is created:

    - If you are using it in the UI, select the connection name you assigned in the **Azure subscription** setting of your pipeline.
    - If you are using it in YAML, copy the connection name into your code as the **azureSubscription** value.

12. If required, modify the service principal to expose the appropriate permissions. For more details, see Use Role-Based Access Control to manage access to your Azure subscription resources. This blog post also contains more information about using service principal authentication.

See also: Troubleshoot Azure Resource Manager service connections.

Create an Azure Resource Manager service connection to a VM with a managed service identity

> **Note**
>
> You are required to use a self-hosted agent on an Azure VM in order to use managed service identity

You can configure Azure Virtual Machines (VM)-based agents with an Azure Managed Service Identity in Microsoft Entra ID. This lets you use the system assigned identity (Service Principal) to grant the Azure VM-based agents access to any Azure resource that supports Microsoft Entra ID, such as Key Vault, instead of persisting credentials in Azure DevOps for the connection.

1. In Azure DevOps, open the **Service connections** page from the project settings page. In TFS, open the **Services** page from the "settings" icon in the top menu bar.

2. Choose + **New service connection** and select **Azure Resource Manager**.

3. Select the **Managed Identity Authentication** option.

4. Enter a user-friendly **Connection name** to use when referring to this service connection.

5. Select the **Environment** name (such as Azure Cloud, Azure Stack, or an Azure Government Cloud).

6. Enter the values for your subscription into these fields of the connection dialog:

   - Subscription ID
   - Subscription name
   - Tenant ID

7. After the new service connection is created:

   - If you are using it in the UI, select the connection name you assigned in the **Azure subscription** setting of your pipeline.
   - If you are using it in YAML, copy the connection name into your code as the **azureSubscription** value.

8. Ensure that the VM (agent) has the appropriate permissions. For example, if your code needs to call Azure Resource Manager, assign the VM the appropriate role using Role-Based Access Control (RBAC) in Microsoft Entra ID. For more details, see How can I use managed identities for Azure resources? and Use Role-Based Access Control to manage access to your Azure subscription resources.

See also: Troubleshoot Azure Resource Manager service connections.

Connect to an Azure Government Cloud

For information about connecting to an Azure Government Cloud, see:

- Connecting from Azure Pipelines (Azure Government Cloud)

Connect to Azure Stack

For information about connecting to Azure Stack, see:

- Connect to Azure Stack
- Connect Azure Stack to Azure using VPN
- Connect Azure Stack to Azure using ExpressRoute

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and feel free to post your questions, search for answers, or suggest a feature on our Azure DevOps Developer Community. Support page.

Troubleshoot Azure Resource Manager workload identity service connections

Article • 12/08/2023

Get help debugging common issues with workload identity service connections and learn how to manually create a service connection if necessary.

> **Important**
>
> Workload identity federation for Azure Resource Manager is currently in public preview. See the **Supplemental Terms of Use for Microsoft Azure Previews** for legal terms that apply to Azure features that are in beta, preview, or otherwise not yet released into general availability.
>
> This feature is in public preview and may not be available in your organization. To request early access, go to **https://aka.ms/azdo-rm-workload-identity-preview**.

Troubleshooting checklist

Review pipeline tasks

Not all pipelines tasks support workload identity. During the preview, no Azure Marketplace tasks support workload identity service connections. The following tasks don't currently support with workload identity federation.

- `AzureCloudPowerShellDeploymentV1`
- `AzCopy` (`AzureFileCopyV1`, `AzureFileCopyV2`, `AzureFileCopyV3`, `AzureFileCopyV4`, `AzureFileCopyV5`)
- `AzureIoTEdgeV2`
- `AzureMonitorAlertsV0`
- `AzureNLBManagementV1`
- `PackerBuild` (`PackerBuildV0`, `PackerBuildV1`)
- `ServiceFabricComposeDeployV0`
- `ServiceFabricDeployV1`

Verify workload identity federation is active

If you see error message `AADSTS700223` or `AADSTS700238`, workload identity federation was disabled in your Microsoft Entra tenant.

Verify that there are no Microsoft Entra policies in place that block federated credentials.

Check the issuer URL for accuracy

If you see a message that there's `no matching federated identity record found`, either the issuer URL or federation subject doesn't match. The correct issuer URL show start with `https://vstoken.dev.azure.com`.

You can fix the issuer URL by editing and saving the service connection to update the issuer URL. The issuer needs to be updated manually if Azure DevOps didn't create the identity. For Azure identities, the issuer URL automatically updates.

Common issues and causes

I don't have permission to create a service principal in the Azure Microsoft Entra tenant

You can't use the service connection configuration tool when you don't have the correct permissions. Your permission level is inadequate if you don't have permission to create service principals or you're using a different Azure Microsoft Entra tenant than your Azure DevOps user.

You need to either have permission in Microsoft Entra ID to create app registrations or you need to have an appropriate role (example: Application Developer).

You have two options:

- Solution 1: Manually configure workload identity with managed identity
- Solution 2: Manually configure workload identity with a service principal

Error messages

Expand table

Message	Plausible issue
cannot request token: Get `?audience=api://AzureADTokenExchange:`	Task doesn't support workload identity federation.

Message	Plausible issue
`unsupported protocol scheme`	
Identity not found	Task doesn't support workload identity federation.
Could not fetch access token for Azure	Task doesn't support workload identity federation.
<Task> *only support(s) service principal authorization*	Task doesn't support workload identity federation.
*AADSTS700016: Application with identifier '**' was not found*	The identity used for the service connection no longer exists, it may have been removed independent from the service connection. You will need to create a new service connection.
AADSTS7000215: Invalid client secret provided.	You are using a service connection with a secret that expired. Convert the service connection to Workload identity federation to replace the expired secret with a federation.
AADSTS700024: Client assertion is not within its valid time range	You're using the AzureCLI task with `addSpnToEnvironment: true` to consume the `idToken` environment variable. The `idToken` has expired after 10 minutes.
AADSTS70021: No matching federated identity record found for presented assertion. Assertion Issuer: `https://app.vstoken.visualstudio.com`.	The issuer URL isn't correct. The correct issuer URL is `https://vstoken.dev.azure.com/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX`. You can fix the issuer URL by editing and resaving a service connection. You need to manually update the issuer if Azure DevOps didn't create your identity. You can find the correct issuer in the edit dialog of the service connection, or in the response (under authorization parameters) when using the REST API.
AADSTS70021: No matching federated identity record found for presented assertion. Assertion Issuer: `https://vstoken.dev.azure.com/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX`. *Assertion Subject:* `sc://\<org\>/\<project\>/\<service-connection\>.`	Either the issuer url or federation subject doesn't match. The Azure DevOps organization or project has been renamed, or a manually created service connection was renamed without updating the federation subject on the identity.
AADSTS700223	Workload identity federation has been constrained or disabled on the Microsoft Entra tenant. In these cases it is often possible to use a

Message	Plausible issue
	Managed Identity for the federation instead, see Workload identity with managed identity.
Microsoft Entra rejected the token issued by Azure DevOps with error code AADSTS700238	Workload identity federation has been constrained on the Microsoft Entra tenant. The issuer for your organization (`https://vstoken.dev.azure.com/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX`) is not allowed to use Workload identity federation. Please contact your Entra tenant administrator(s) to allow Workload identity federation for your Azure DevOps organization.
Failed to obtain the Json Web Token(JWT) using service principal client ID	Your federation identity credential is misconfigured or the Microsoft Entra tenant blocks OIDC.
Script failed with error: UnrecognizedArgumentError: unrecognized arguments: --federated-token	You're using the AzureCLI task on an agent that has an old version of the Azure CLI installed. Workload identity federation requires Azure CLI 2.30 or higher.
Failed to create an app in Microsoft Entra ID. Error: Insufficient privileges to complete the operation in Microsoft Graph. Ensure that the user has permissions to create a Microsoft Entra Application.	The ability to create App Registrations has been disabled in the Microsoft Entra tenant. Assign the user creating the service connection to the Application Developer Microsoft Entra role. Alternatively, create the service connection manually with a Managed Identity. See Workload identity with managed identity.

Manually configure Azure Resource Manager workload identity service connections

Article • 09/13/2023

When you troubleshoot an Azure Resource Manager workload identity service connection, you many need to manually configure the connection instead of using the automated tool. You should try the automated approach first before starting a manual configuration.

There are two options for authentication: managed identity and using a service principal. The advantage of the managed identity option is that you can use it when you don't have permission to create service principals or you are using a different Azure Microsoft Entra tenant than your Azure DevOps user.

Workload identity with managed identity

> **Important**
>
> Workload identity federation for Azure Resource Manager is currently in public preview. See the **Supplemental Terms of Use for Microsoft Azure Previews** for legal terms that apply to Azure features that are in beta, preview, or otherwise not yet released into general availability.
>
> This feature is in public preview and may not be available in your organization. To request early access, go to **https://aka.ms/azdo-rm-workload-identity-preview**.

You may need to manually create a managed identity with federated credentials and grant appropriate permissions. You can also use the REST API for this process.

Create a managed identity

1. Sign in to the Azure portal.
2. In the search box, enter **Managed Identities**.
3. Select **Create**, and enter values in the following boxes in the **Create User Assigned Managed Identity** pane:

- **Subscription**: Choose the subscription to create the user-assigned managed identity under.
- **Resource group**: Choose a resource group to create the user-assigned managed identity in, or select **Create new** to create a new resource group.
- **Region**: Choose a region to deploy the user-assigned managed identity, for example, **East US**.
- **Name**: Enter the name for your user-assigned managed identity, for example, UADEVOPS.

4. Copy the **Subscription ID** and **Client ID** for your managed identity. You'll use these values later.

5. Go to **Settings** > **Properties**.

6. Copy the **Tenant Id** value to use later.

7. Go to **Settings** > **Federated credentials**.

8. Select **Add credentials**.

9. Select the **Other issuer** scenario.

10. Enter values for **Issuer** and **Subject identifier**. You'll replace these values later.

Field	Description
Issuer	Enter `https://app.vstoken.visualstudio.com/<unique-identifier>`.
Subject identifier	Specify `sc://<Azure DevOps organization>/<Project name>/<Service Connection name>`. You do not need to have created the service connection.

11. Select **Save**.

12. Keep this window open. You'll return and update your app registration federated credential later.

Grant permissions to the managed identity

1. In Azure portal, go to the Azure resource that you want to target (example: resource group).

2. Select **Access control (IAM)**.

3. Select **Add role assignment**. Assign the desired role to your managed identity (example: Contributor).

4. Select **Review and assign**.

Create service connection

1. In Azure DevOps, open your project and go to ⚙ > **Pipelines** > **Service connections**.

2. Select **New service connection**.

3. Select **Azure Resource Manager**.

4. Select **Workload Identity federation (manual)**.

5. Enter the **Service connection name**. The name should match the name you used in the **Subject identifier** field when creating your federated credential.

6. Set the **Subscription ID** to the subscription ID and the **Subscription Name** to the subscription name for your Azure portal account.

7. In the authentication section, set the **Service Principal Id** to the **Client Id** value of your managed identity, and set the **Tenant ID** to the **Tenant Id** value of your managed identity.

8. In Azure DevOps, copy the generated values for **Issuer** and **Subject identifier**.

9. In Azure portal, return to your app registration federated credential.

10. Enter the values for **Issuer** and **Subject identifier** from your Azure DevOps project and enter those values in Azure portal for your federated credential.

11. In Azure portal, select **Update** to save the updated credential.

12. In Azure DevOps, select **Verify and save**.

Workload identity with a service principal

> **Important**
>
> Workload identity federation for Azure Resource Manager is currently in public preview. See the **Supplemental Terms of Use for Microsoft Azure Previews** for legal terms that apply to Azure features that are in beta, preview, or otherwise not yet released into general availability.
>
> This feature is in public preview and may not be available in your organization. To request early access, go to **https://aka.ms/azdo-rm-workload-identity-preview**.

You may need to manually create a service principal with federated credentials and grant appropriate permissions. You can also use the REST API for this process.

Create app registration and federated credential

1. In Azure portal, go to app registrations.

2. Select **New registration**.

3. Enter a **Name** for your app registration and select **Who can use this application or access this API**.

4. Copy the **Application (client) ID** and **Directory (tenant) ID** for your app registration. You'll use those values later.

5. Go to **Manage** > **Certificates & secrets**.

6. Select **Federated credentials**.

7. Select **Add credentials**.

8. Select the **Other issuer** scenario.

9. Enter values for **Issuer** and **Subject identifier**.

Field	Description
Issuer	Enter `https://app.vstoken.visualstudio.com/<unique-identifier>`.
Subject identifier	Specify `sc://<Azure DevOps organization>/<Project name>/<Service Connection name>`. You do not need to have created the service connection.

10. Select **Save**.

11. Keep this window open. You'll return and update your app registration federated credential later.

Grant permissions to the app registration

1. In Azure portal, go to the Azure resource that you want to target (example: resource group).

2. Select **Access control (IAM)**.

3. Select **Add role assignment**. Assign the desired role to your app registration (example: Contributor).

4. Select **Review and assign**.

Create service connection

1. In Azure DevOps, open your project and go to ⚙ > **Pipelines** > **Service connections**.

2. Select **New service connection**.

3. Select **Azure Resource Manager**.

4. Select **Workload Identity federation (manual)**.

5. Enter the **Service connection name**. The name should match the name you used in the **Subject identifier** field when creating your federated credential.

6. Set the **Subscription Id** to the subscription ID and the **Subscription Name** to the subscription name for your Azure portal account.

7. In the authentication section, set the **Service Principal Id** to the **Application (client) ID** value of your app registration.

8. In the authentication section, set the **Tenant Id** to the **Directory (tenant) ID** value of your app registration.

9. Copy the generated values for **Issuer** and **Subject identifier**.

10. In Azure portal, return to your app registration federated credential.

11. Copy the values for **Issuer** and **Subject identifier** from your Azure DevOps project and enter those values in Azure portal for your federated credential.

12. In Azure portal, select **Update** to save the updated credential.

13. In Azure DevOps, select **Verify and save**.

Deploy to App Service using Azure Pipelines

Article • 12/13/2023

Azure DevOps Services | Azure DevOps Server 2020 | Azure DevOps Server 2019

Use Azure Pipelines to automatically deploy your web app to Azure App Service on every successful build. Azure Pipelines lets you build, test, and deploy with continuous integration (CI) and continuous delivery (CD) using Azure DevOps.

YAML pipelines are defined using a YAML file in your repository. A step is the smallest building block of a pipeline and can be a script or task (prepackaged script). Learn about the key concepts and components that make up a pipeline.

You'll use the Azure Web App task (AzureWebApp) to deploy to Azure App Service in your pipeline. For more complicated scenarios such as needing to use XML parameters in your deploy, you can use the Azure App Service deploy task (AzureRmWebAppDeployment).

Prerequisites

- An Azure account with an active subscription. Create an account for free.
- An Azure DevOps organization. Create one for free.
- An ability to run pipelines on Microsoft-hosted agents. You can either purchase a parallel job or you can request a free tier.
- A working Azure App Service app with code hosted on GitHub or Azure Repos.
  - .NET: Create an ASP.NET Core web app in Azure
  - ASP.NET: Create an ASP.NET Framework web app in Azure
  - JavaScript: Create a Node.js web app in Azure App Service
  - Java: Create a Java app on Azure App Service
  - Python: Create a Python app in Azure App Service

1. Create a pipeline for your stack

The code examples in this section assume you're deploying an ASP.NET web app. You can adapt the instructions for other frameworks.

Learn more about Azure Pipelines ecosystem support.

YAML

1. Sign in to your Azure DevOps organization and navigate to your project.
2. Go to **Pipelines**, and then select **New Pipeline**.
3. When prompted, select the location of your source code: either **Azure Repos Git** or **GitHub**.

   You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.
4. When the list of repositories appears, select your repository.
5. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.
6. When the **Configure** tab appears, select **ASP.NET Core**.
7. When your new pipeline appears, take a look at the YAML to see what it does. When you're ready, select **Save and run**.

2. Add the deployment task

YAML

1. Click the end of the YAML file, then select **Show assistant**.'
2. Use the Task assistant to add the Azure Web App task.

Alternatively, you can add the Azure App Service deploy (AzureRmWebAppDeployment) task.

3. Choose your **Azure subscription**. Make sure to **Authorize** your connection. The authorization creates the required service connection.

4. Select the **App type**, **App name**, and **Runtime stack** based on your App Service app. Your complete YAML should look similar to the following code.

YAML

```yaml
variables:
  buildConfiguration: 'Release'

steps:
- script: dotnet build --configuration $(buildConfiguration)
  displayName: 'dotnet build $(buildConfiguration)'
- task: DotNetCoreCLI@2
  inputs:
    command: 'publish'
    publishWebProjects: true
- task: AzureWebApp@1
  inputs:
    azureSubscription: '<service-connection-name>'
    appType: 'webAppLinux'
    appName: '<app-name>'
    package: '$(System.DefaultWorkingDirectory)/**/*.zip'
```

- **azureSubscription**: Name of the authorized service connection to your Azure subscription.
- **appName**: Name of your existing app.
- **package**: Fike path to the package or a folder containing your app service contents. Wildcards are supported.

Example: Deploy a .NET app

YAML

To deploy a .zip web package (for example, from an ASP.NET web app) to an Azure Web App, use the following snippet to deploy the build to an app.

YAML

```yaml
variables:
  buildConfiguration: 'Release'
```

```yaml
steps:
- script: dotnet build --configuration $(buildConfiguration)
  displayName: 'dotnet build $(buildConfiguration)'
- task: DotNetCoreCLI@2
  inputs:
    command: 'publish'
    publishWebProjects: true
- task: AzureWebApp@1
  inputs:
    azureSubscription: '<service-connection-name>'
    appType: 'webAppLinux'
    appName: '<app-name>'
    package: '$(System.DefaultWorkingDirectory)/**/*.zip'
```

- **azureSubscription**: your Azure subscription.
- **appType**: your Web App type.
- **appName**: the name of your existing app service.
- **package**: the file path to the package or a folder containing your app service contents. Wildcards are supported.

Example: deploy to a virtual application

YAML

By default, your deployment happens to the root application in the Azure Web App. You can deploy to a specific virtual application by using the `VirtualApplication` property of the Azure App Service deploy (`AzureRmWebAppDeployment`) task:

YAML

```yaml
- task: AzureRmWebAppDeployment@4
  inputs:
    VirtualApplication: '<name of virtual application>'
```

- **VirtualApplication**: the name of the Virtual Application that's configured in the Azure portal. For more information, see Configure an App Service app in the Azure portal .

Example: Deploy to a slot

YAML

The following example shows how to deploy to a staging slot, and then swap to a production slot:

YAML

```yaml
- task: AzureWebApp@1
  inputs:
    azureSubscription: '<service-connection-name>'
    appType: webAppLinux
    appName: '<app-name>'
    deployToSlotOrASE: true
    resourceGroupName: '<name of resource group>'
    slotName: staging
    package: '$(Build.ArtifactStagingDirectory)/**/*.zip'

- task: AzureAppServiceManage@0
  inputs:
    azureSubscription: '<service-connection-name>'
    appType: webAppLinux
    WebAppName: '<app-name>'
    ResourceGroupName: '<name of resource group>'
    SourceSlot: staging
    SwapWithProduction: true
```

- **azureSubscription**: your Azure subscription.
- **appType**: (optional) Use `webAppLinux` to deploy to a Web App on Linux.
- **appName**: the name of your existing app service.
- **deployToSlotOrASE**: Boolean. Deploy to an existing deployment slot or Azure App Service Environment.
- **resourceGroupName**: Name of the resource group. Required if `deployToSlotOrASE` is true.
- **slotName**: Name of the slot, which defaults to `production`. Required if `deployToSlotOrASE` is true.
- **package**: the file path to the package or a folder containing your app service contents. Wildcards are supported.
- **SourceSlot**: Slot sent to production when `SwapWithProduction` is true.
- **SwapWithProduction**: Boolean. Swap the traffic of source slot with production.

Example: Deploy to multiple web apps

YAML

You can use jobs in your YAML file to set up a pipeline of deployments. By using jobs, you can control the order of deployment to multiple web apps.

YAML

```yaml
jobs:
- job: buildandtest
  pool:
    vmImage: ubuntu-latest

  steps:
  # publish an artifact called drop
  - task: PublishPipelineArtifact@1
    inputs:
      targetPath: '$(Build.ArtifactStagingDirectory)'
      artifactName: drop

  # deploy to Azure Web App staging
  - task: AzureWebApp@1
    inputs:
      azureSubscription: '<service-connection-name>'
      appType: <app type>
      appName: '<staging-app-name>'
      deployToSlotOrASE: true
      resourceGroupName: <group-name>
      slotName: 'staging'
      package: '$(Build.ArtifactStagingDirectory)/**/*.zip'

- job: deploy
  dependsOn: buildandtest
  condition: succeeded()

  pool:
    vmImage: ubuntu-latest

  steps:
    # download the artifact drop from the previous job
  - task: DownloadPipelineArtifact@2
    inputs:
      source: 'current'
      artifact: 'drop'
      path: '$(Pipeline.Workspace)'

  - task: AzureWebApp@1
    inputs:
      azureSubscription: '<service-connection-name>'
      appType: <app type>
      appName: '<production-app-name>'
```

```
      resourceGroupName: <group-name>
      package: '$(Pipeline.Workspace)/**/*.zip'
```

Example: Make variable substitutions

For most language stacks, app settings and connection strings can be set as environment variables at runtime.

But there are other reasons you would want to make variable substitutions to your *Web.config*. In this example, your Web.config file contains a connection string named `connectionString`. You can change its value before deploying to each web app. You can do this either by applying a Web.config transformation or by substituting variables in your Web.config file.

YAML

The following snippet shows an example of variable substitution by using the Azure App Service Deploy (`AzureRmWebAppDeployment`) task:

YAML

```yaml
jobs:
- job: test
  variables:
    connectionString: <test-stage connection string>
  steps:
  - task: AzureRmWebAppDeployment@4
    inputs:
      azureSubscription: '<Test stage Azure service connection>'
      WebAppName: '<name of test stage web app>'
      enableXmlVariableSubstitution: true

- job: prod
  dependsOn: test
  variables:
    connectionString: <prod-stage connection string>
  steps:
  - task: AzureRmWebAppDeployment@4
    inputs:
      azureSubscription: '<Prod stage Azure service connection>'
      WebAppName: '<name of prod stage web app>'
      enableXmlVariableSubstitution: true
```

Example: Deploy conditionally

YAML

To do this in YAML, you can use one of the following techniques:

- Isolate the deployment steps into a separate job, and add a condition to that job.
- Add a condition to the step.

The following example shows how to use step conditions to deploy only builds that originate from the main branch:

YAML

```yaml
- task: AzureWebApp@1
  condition: and(succeeded(), eq(variables['Build.SourceBranch'], 'refs/heads/main'))
  inputs:
    azureSubscription: '<service-connection-name>'
    appName: '<app-name>'
```

To learn more about conditions, see Specify conditions.

Example: deploy using Web Deploy

The Azure App Service deploy (`AzureRmWebAppDeployment`) task can deploy to App Service using Web Deploy.

YAML

yml

```yaml
trigger:
- main

pool:
  vmImage: windows-latest

variables:
  buildConfiguration: 'Release'

steps:
- script: dotnet build --configuration $(buildConfiguration)
  displayName: 'dotnet build $(buildConfiguration)'
- task: DotNetCoreCLI@2
  inputs:
    command: 'publish'
```

```
    publishWebProjects: true
    arguments: '--configuration $(buildConfiguration)'
    zipAfterPublish: true
  - task: AzureRmWebAppDeployment@4
    inputs:
      ConnectionType: 'AzureRM'
      azureSubscription: '<service-connection-name>'
      appType: 'webApp'
      WebAppName: '<app-name>'
      packageForLinux: '$(System.DefaultWorkingDirectory)/**/*.zip'
      enableCustomDeployment: true
      DeploymentType: 'webDeploy'
```

Frequently asked questions

What's the difference between the `AzureWebApp` and `AzureRmWebAppDeployment` tasks?

The Azure Web App task (`AzureWebApp`) is the simplest way to deploy to an Azure Web App. By default, your deployment happens to the root application in the Azure Web App.

The Azure App Service Deploy task (AzureRmWebAppDeployment) can handle more custom scenarios, such as:

- Modify configuration settings inside web packages and XML parameters files.
- Deploy with Web Deploy, if you're used to the IIS deployment process.
- Deploy to virtual applications.
- Deploy to other app types, like Container apps, Function apps, WebJobs, or API and Mobile apps.

> **Note**
>
> File transforms and variable substitution are also supported by the separate **File Transform task** for use in Azure Pipelines. You can use the File Transform task to apply file transformations and variable substitutions on any configuration and parameters files.

I get the message "Invalid App Service package or folder path provided."

In YAML pipelines, depending on your pipeline, there may be a mismatch between where your built web package is saved and where the deploy task is looking for it. For example, the `AzureWebApp` task picks up the web package for deployment. For example, the AzureWebApp task looks in `$(System.DefaultWorkingDirectory)/**/*.zip`. If the web package is deposited elsewhere, modify the value of `package`.

I get the message "Publish using webdeploy options are supported only when using Windows agent."

This error occurs in the **AzureRmWebAppDeployment** task when you configure the task to deploy using Web Deploy, but your agent isn't running Windows. Verify that your YAML has something similar to the following code:

```yml
pool:
  vmImage: windows-latest
```

Web Deploy doesn't work when I disable basic authentication

For troubleshooting information on getting Microsoft Entra ID authentication to work with the `AzureRmWebAppDeployment` task, see I can't Web Deploy to my Azure App Service using Microsoft Entra ID authentication from my Windows agent

Next steps

- Customize your Azure DevOps pipeline.

Deploy a custom container to Azure App Service with Azure Pipelines

Article • 11/08/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Using Azure Pipelines, you can build, test, and automatically deploy your web app to an Azure App Service Web App container on Linux. In this article, you will learn how to use YAML or Classic pipelines to:

- ✓ Build and publish a Docker image to Azure Container Registry
- ✓ Create an Azure Web App
- ✓ Deploy a container to Azure App Service
- ✓ Deploy to deployment slots

Prerequisites

- An Azure account with an active subscription. Create an account for free.
- A GitHub account. Create a free GitHub account if you don't have one already.
- An Azure DevOps organization. Create an organization, if you don't have one already.
- An Azure Container Registry. Create an Azure container registry if you don't have one already.

Get the code

Fork the following sample app at GitHub.

Java

```
https://github.com/spring-guides/gs-spring-boot-docker.git
```

Build and publish a Docker image to Azure Container Registry

To complete this section successfully, you must have an Azure Container Registry. Refer to the prerequisites section for details.

1. Sign in to your Azure DevOps organization and navigate to your project.

2. Select **Pipelines**, and then **New Pipeline**.

3. Select **GitHub** when prompted for the location of your source code, and then select your repository.

4. Select the **Docker: build and push an image to Azure Container Registry** pipeline template.

5. Select your Azure subscription, and then select **Continue**.

6. Select your **Container registry** from the drop-down menu, and then select **Validate and configure**.

7. Review the pipeline YAML template, and then select **Save and run** to build and publish the Docker image to your Azure Container Registry.

YAML

```yaml
trigger:
- main

resources:
- repo: self

variables:
    # Container registry service connection established during pipeline creation
    dockerRegistryServiceConnection: '{{ containerRegistryConnection.Id }}'
    imageRepository: 'javascriptdocker'
    containerRegistry: 'sampleappcontainerregistry.azurecr.io'
    dockerfilePath: '$(Build.SourcesDirectory)/app/Dockerfile'
    tag: '$(Build.BuildId)'

    # Agent VM image name
    vmImageName: 'ubuntu-latest'

stages:
- stage: Build
    displayName: Build and push stage
    jobs:
    - job: Build
    displayName: Build
```

```
    pool:
        vmImage: $(vmImageName)
    steps:
    - task: Docker@2
        displayName: Build and push an image to container registry
        inputs:
        command: buildAndPush
        repository: $(imageRepository)
        dockerfile: $(dockerfilePath)
        containerRegistry: $(dockerRegistryServiceConnection)
        tags: |
            $(tag)
```

8. To view the published Docker image after your pipeline run has been completed, navigate to your container registry in Azure portal, and then select **Repositories**.

9. To deploy your image from the container registry, you must enable the admin user account. Navigate to your container registry in Azure portal, and select **Access keys**. Next, select the toggle button to **Enable Admin user**.

Create a Web App

1. Navigate to Azure portal.

2. Select **Create a resource** > **Containers**, and then choose **Web App for Containers**.

3. Enter a name for your new web app, and create a new Resource Group. Select **Linux** for the **Operating System**.

4. In the **Pricing plans** section, choose the **F1 Free plan**.

5. Select **Review and create**. Review your configuration, and select **Create** when you are done.

Deploy to Web App for Containers

YAML

In this YAML, you build and push a Docker image to a container registry and then deploy it to Azure Web App for Containers. In the Build stage, you build and push a Docker image to an Azure Container Registry with the Docker@2 task. The AzureWebAppContainer@1 task deploys the image to Web App for Containers.

YAML

```yaml
trigger:
- main

resources:
- repo: self

variables:
  ## Add this under variables section in the pipeline
  azureSubscription: <Name of the Azure subscription>
  appName: <Name of the Web App>
  containerRegistry: <Name of the Azure container registry>
  dockerRegistryServiceConnection: '4fa4efbc-59af-4c0b-8637-
1d5bf7f268fc'
  imageRepository: <Name of image repository>
  dockerfilePath: '$(Build.SourcesDirectory)/Dockerfile'
  tag: '$(Build.BuildId)'

  vmImageName: 'ubuntu-latest'

stages:
- stage: Build
  displayName: Build and push stage
  jobs:
  - job: Build
    displayName: Build
    pool:
      vmImage: $(vmImageName)
    steps:
    - task: Docker@2
      displayName: Build and push an image to container registry
      inputs:
        command: buildAndPush
        repository: $(imageRepository)
        dockerfile: $(dockerfilePath)
        containerRegistry: $(dockerRegistryServiceConnection)
        tags: |
          $(tag)


    ## Add the below snippet at the end of your pipeline
    - task: AzureWebAppContainer@1
      displayName: 'Azure Web App on Container Deploy'
      inputs:
        azureSubscription: $(azureSubscription)
```

```
        appName: $(appName)
        containers: $(containerRegistry)/$(imageRepository):$(tag)
```

Deploy to a deployment slot

YAML

You can configure the Azure Web App container to have multiple slots. Slots allow you to safely deploy your app and test it before making it available to your customers. See Create staging environments for more details.

The following YAML snippet shows how to deploy to a staging slot, and then swap to a production slot:

YAML

```yaml
- task: AzureWebAppContainer@1
  inputs:
    azureSubscription: '<Azure service connection>'
    appName: '<Name of the web app>'
    containers: $(containerRegistry)/$(imageRepository):$(tag)
    deployToSlotOrASE: true
    resourceGroupName: '<Name of the resource group>'
    slotName: staging

- task: AzureAppServiceManage@0
  inputs:
    azureSubscription: '<Azure service connection>'
    WebAppName: '<name of web app>'
    ResourceGroupName: '<name of resource group>'
    SourceSlot: staging
    SwapWithProduction: true
```

FAQ

Q: How can I find my Docker registry credentials?

A: Navigate to Azure portal, and then select your Web App for Containers. Select **Configuration** > **Application settings** and then click to show the value.

Related articles

- Deploy to Azure
- Use ARM templates
- Define and target environments

Continuous delivery with Azure Pipelines

Article • 05/25/2023

Use Azure Pipelines to automatically deploy to Azure Functions. Azure Pipelines lets you build, test, and deploy with continuous integration (CI) and continuous delivery (CD) using Azure DevOps.

YAML pipelines are defined using a YAML file in your repository. A step is the smallest building block of a pipeline and can be a script or task (prepackaged script). Learn about the key concepts and components that make up a pipeline.

You'll use the AzureFunctionApp task to deploy to Azure Functions. There are now two versions of the AzureFunctionApp task (AzureFunctionApp@1, AzureFunctionApp@2). AzureFunctionApp@2 includes enhanced validation support that makes pipelines less likely to fail because of errors.

Choose your task version at the top of the article. YAML pipelines aren't available for Azure DevOps 2019 and earlier.

Prerequisites

- A GitHub account, where you can create a repository. If you don't have one, you can create one for free.
- An Azure DevOps organization. If you don't have one, you can create one for free. If your team already has one, then make sure you're an administrator of the Azure DevOps project that you want to use.
- An ability to run pipelines on Microsoft-hosted agents. You can either purchase a parallel job or you can request a free tier.
- A function app with its code in a GitHub repository. If you don't yet have an Azure Functions code project, you can create one by completing the following language-specific article:

  C#

  Quickstart: Create a C# function in Azure using Visual Studio Code

Build your app

YAML

1. Sign in to your Azure DevOps organization and navigate to your project.
2. In your project, navigate to the **Pipelines** page. Then choose the action to create a new pipeline.
3. Walk through the steps of the wizard by first selecting **GitHub** as the location of your source code.
4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.
5. When the list of repositories appears, select your sample app repository.
6. Azure Pipelines will analyze your repository and recommend a template. Select **Save and run**, then select **Commit directly to the main branch**, and then choose **Save and run** again.
7. A new run is started. Wait for the run to finish.

Example YAML build pipelines

The following language-specific pipelines can be used for building apps.

C#

You can use the following sample to create a YAML file to build a .NET app.

If you see errors when building your app, verify that the version of .NET that you use matches your Azure Functions version. For more information, see Azure Functions runtime versions overview.

YAML

```yaml
pool:
  vmImage: 'windows-latest'
steps:
- script: |
    dotnet restore
    dotnet build --configuration Release
- task: DotNetCoreCLI@2
  inputs:
    command: publish
    arguments: '--configuration Release --output publish_output'
    projects: '*.csproj'
    publishWebProjects: false
```

```
    modifyOutputPath: false
    zipAfterPublish: false
- task: ArchiveFiles@2
  displayName: "Archive files"
  inputs:
    rootFolderOrFile: "$(System.DefaultWorkingDirectory)/publish_output"
    includeRootFolder: false
    archiveFile:
"$(System.DefaultWorkingDirectory)/build$(Build.BuildId).zip"
- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish:
'$(System.DefaultWorkingDirectory)/build$(Build.BuildId).zip'
    artifactName: 'drop'
```

Deploy your app

You'll deploy with the Azure Function App Deploy task. This task requires an Azure service connection as an input. An Azure service connection stores the credentials to connect from Azure Pipelines to Azure.

YAML

To deploy to Azure Functions, add the following snippet at the end of your `azure-pipelines.yml` file. The default `appType` is Windows. You can specify Linux by setting the `appType` to `functionAppLinux`.

YAML

```
trigger:
- main

variables:
  # Azure service connection established during pipeline creation
  azureSubscription: <Name of your Azure subscription>
  appName: <Name of the function app>
  # Agent VM image name
  vmImageName: 'ubuntu-latest'

- task: AzureFunctionApp@1 # Add this at the end of your file
  inputs:
    azureSubscription: <Azure service connection>
    appType: functionAppLinux # default is functionApp
    appName: $(appName)
    package: $(System.ArtifactsDirectory)/**/*.zip
    #Uncomment the next lines to deploy to a deployment slot
    #Note that deployment slots is not supported for Linux Dynamic SKU
    #deployToSlotOrASE: true
```

```
    #resourceGroupName: '<Resource Group Name>'
    #slotName: '<Slot name>'
```

The snippet assumes that the build steps in your YAML file produce the zip archive in the `$(System.ArtifactsDirectory)` folder on your agent.

Deploy a container

You can automatically deploy your code to Azure Functions as a custom container after every successful build. To learn more about containers, see Create a function on Linux using a custom container.

Deploy with the Azure Function App for Container task

YAML

The simplest way to deploy to a container is to use the Azure Function App on Container Deploy task.

To deploy, add the following snippet at the end of your YAML file:

YAML

```yaml
trigger:
- main

variables:
  # Container registry service connection established during pipeline
creation
  dockerRegistryServiceConnection: <Docker registry service connection>
  imageRepository: <Name of your image repository>
  containerRegistry: <Name of the Azure container registry>
  dockerfilePath: '$(Build.SourcesDirectory)/Dockerfile'
  tag: '$(Build.BuildId)'

  # Agent VM image name
  vmImageName: 'ubuntu-latest'

- task: AzureFunctionAppContainer@1 # Add this at the end of your file
  inputs:
    azureSubscription: '<Azure service connection>'
    appName: '<Name of the function app>'
    imageName: $(containerRegistry)/$(imageRepository):$(tag)
```

The snippet pushes the Docker image to your Azure Container Registry. The **Azure Function App on Container Deploy** task pulls the appropriate Docker image corresponding to the `BuildId` from the repository specified, and then deploys the image.

Deploy to a slot

YAML

You can configure your function app to have multiple slots. Slots allow you to safely deploy your app and test it before making it available to your customers.

The following YAML snippet shows how to deploy to a staging slot, and then swap to a production slot:

YAML

```yaml
- task: AzureFunctionApp@1
  inputs:
    azureSubscription: <Azure service connection>
    appType: functionAppLinux
    appName: <Name of the Function app>
    package: $(System.ArtifactsDirectory)/**/*.zip
    deployToSlotOrASE: true
    resourceGroupName: <Name of the resource group>
    slotName: staging

- task: AzureAppServiceManage@0
  inputs:
    azureSubscription: <Azure service connection>
    WebAppName: <name of the Function app>
    ResourceGroupName: <name of resource group>
    SourceSlot: staging
    SwapWithProduction: true
```

Create a pipeline with Azure CLI

To create a build pipeline in Azure, use the `az functionapp devops-pipeline create` command. The build pipeline is created to build and release any code changes that are made in your repo. The command generates a new YAML file that defines the build and release pipeline and then commits it to your repo. The prerequisites for this command depend on the location of your code.

- If your code is in GitHub:
  - You must have **write** permissions for your subscription.
  - You must be the project administrator in Azure DevOps.
  - You must have permissions to create a GitHub personal access token (PAT) that has sufficient permissions. For more information, see GitHub PAT permission requirements.
  - You must have permissions to commit to the main branch in your GitHub repository so you can commit the autogenerated YAML file.
- If your code is in Azure Repos:
  - You must have **write** permissions for your subscription.
  - You must be the project administrator in Azure DevOps.

Next steps

- Review the Azure Functions overview.
- Review the Azure DevOps overview.

Quickstart: Use an ARM template to deploy a Linux web app to Azure

Article • 03/30/2023

Azure DevOps Services

Get started with Azure Resource Manager templates (ARM templates) by deploying a Linux web app with MySQL. ARM templates give you a way to save your configuration in code. Using an ARM template is an example of infrastructure as code and a good DevOps practice.

An ARM template is a JavaScript Object Notation (JSON) file that defines the infrastructure and configuration for your project. The template uses declarative syntax. In declarative syntax, you describe your intended deployment without writing the sequence of programming commands to create the deployment.

You can use either JSON or Bicep syntax to deploy Azure resources. Learn more about the difference between JSON and Bicep for templates.

Prerequisites

Before you begin, you need:

- An Azure account with an active subscription. Create an account for free.
- An active Azure DevOps organization. Sign up for Azure Pipelines.
- (For Bicep deployments) An existing resource group. Create a resource group with Azure portal, Azure CLI, or Azure PowerShell.

Get the code

Fork this repository on GitHub:

```
https://github.com/Azure/azure-quickstart-
templates/tree/master/quickstarts/microsoft.web/webapp-linux-managed-mysql
```

Review the template

JSON

The template used in this quickstart is from Azure Quickstart Templates.

The template defines several resources:

- Microsoft.Web/serverfarms
- Microsoft.Web/sites
- Microsoft.DBforMySQL/servers
- Microsoft.DBforMySQL/servers/firewallrules
- Microsoft.DBforMySQL/servers/databases

Create your pipeline and deploy your template

1. Sign in to your Azure DevOps organization and navigate to your project. Create a project if you do not already have one.

2. Go to **Pipelines**, and then select **Create Pipeline**.

3. Select **GitHub** as the location of your source code.

   > **Note**
   >
   > You may be redirected to GitHub to sign in. If so, enter your GitHub credentials.

4. When the list of repositories appears, select `yourname/azure-quickstart-templates/`.

   > **Note**
   >
   > You may be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve and install**.

5. When the Configure tab appears, select `Starter pipeline`.

6. Replace the content of your pipeline with this code:

   ```yml
   ```

```yaml
trigger:
- none

pool:
  vmImage: 'ubuntu-latest'
```

7. Create three variables: `siteName`, `administratorLogin`, and `adminPass`. `adminPass` needs to be a secret variable.

   - Select **Variables**.
   - Use the `+` sign to add three variables. When you create `adminPass`, select **Keep this value secret**.
   - Click **Save** when you're done.

Variable	Value	Secret?
siteName	`mytestsite`	No
adminUser	`fabrikam`	No
adminPass	`Fqdn:5362!`	Yes

8. Map the secret variable `$(adminPass)` so that it is available in your Azure Resource Group Deployment task. At the top of your YAML file, map `$(adminPass)` to `$(ARM_PASS)`.

```yml
variables:
  ARM_PASS: $(adminPass)

trigger:
- none

pool:
  vmImage: 'ubuntu-latest'
```

9. Add the Copy Files task to the YAML file. You will use the `101-webapp-linux-managed-mysql` project. For more information, see Build a Web app on Linux with Azure database for MySQL repo for more details.

```yml
variables:
  ARM_PASS: $(adminPass)
```

```yaml
trigger:
- none

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: CopyFiles@2
  inputs:
    SourceFolder: 'quickstarts/microsoft.web/webapp-linux-managed-mysql/'
    Contents: '**'
    TargetFolder: '$(Build.ArtifactStagingDirectory)'
```

10. Add and configure the **Azure Resource Group Deployment** task.

    The task references both the artifact you built with the Copy Files task and your pipeline variables. Set these values when configuring your task.

    - **Deployment scope (deploymentScope)**: Set the deployment scope to `Resource Group`. You can target your deployment to a management group, an Azure subscription, or a resource group.
    - **Azure Resource Manager connection (azureResourceManagerConnection)**: Select your Azure Resource Manager service connection. To configure new service connection, select the Azure subscription from the list and click **Authorize**. See Connect to Microsoft Azure for more details
    - **Subscription (subscriptionId)**: Select the subscription where the deployment should go.
    - **Action (action)**: Set to `Create or update resource group` to create a new resource group or to update an existing one.
    - **Resource group**: Set to `ARMPipelinesLAMP-rg` to name your new resource group. If this is an existing resource group, it will be updated.
    - **Location(location)**: Location for deploying the resource group. Set to your closest location (for example, West US). If the resource group already exists in your subscription, this value will be ignored.
    - **Template location (templateLocation)**: Set to `Linked artifact`. This is location of your template and the parameters files.
    - **Template (csmFile)**: Set to `$(Build.ArtifactStagingDirectory)/azuredeploy.json`. This is the path to the ARM template.
    - **Template parameters (csmParametersFile)**: Set to `$(Build.ArtifactStagingDirectory)/azuredeploy.parameters.json`. This is the path to the parameters file for your ARM template.

- **Override template parameters (overrideParameters)**: Set to `-siteName $(siteName) -administratorLogin $(adminUser) -administratorLoginPassword $(ARM_PASS)` to use the variables you created earlier. These values will replace the parameters set in your template parameters file.
- **Deployment mode (deploymentMode)**: The way resources should be deployed. Set to `Incremental`. Incremental keeps resources that are not in the ARM template and is faster than `Complete`. `Validate` mode lets you find problems with the template before deploying.

yml

```yml
variables:
  ARM_PASS: $(adminPass)

trigger:
- none

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: CopyFiles@2
  inputs:
    SourceFolder: 'quickstarts/microsoft.web/webapp-linux-managed-mysql/'
    Contents: '**'
    TargetFolder: '$(Build.ArtifactStagingDirectory)'

- task: AzureResourceManagerTemplateDeployment@3
  inputs:
    deploymentScope: 'Resource Group'
    azureResourceManagerConnection: '<your-resource-manager-connection>'
    subscriptionId: '<your-subscription-id>'
    action: 'Create Or Update Resource Group'
    resourceGroupName: 'ARMPipelinesLAMP-rg'
    location: '<your-closest-location>'
    templateLocation: 'Linked artifact'
    csmFile: '$(Build.ArtifactStagingDirectory)/azuredeploy.json'
    csmParametersFile: '$(Build.ArtifactStagingDirectory)/azuredeploy.parameters.json'
    overrideParameters: '-siteName $(siteName) -administratorLogin $(adminUser) -administratorLoginPassword $(ARM_PASS)'
    deploymentMode: 'Incremental'
```

11. Click **Save and run** to deploy your template. The pipeline job will be launched and after few minutes, depending on your agent, the job status should indicate `Success`.

Review deployed resources

JSON

1. Verify that the resources deployed. Go to the `ARMPipelinesLAMP-rg` resource group in the Azure portal and verify that you see App Service, App Service Plan, and Azure Database for MySQL server resources.

   You can also verify the resources using Azure CLI.

   Azure CLI

   ```
   az resource list --resource-group ARMPipelinesLAMP-rg --output table
   ```

2. Go to your new site. If you set `siteName` to `armpipelinetestsite`, the site is located at `https://armpipelinetestsite.azurewebsites.net/`.

Clean up resources

JSON

You can also use an ARM template to delete resources. Change the `action` value in your **Azure Resource Group Deployment** task to `DeleteRG`. You can also remove the inputs for `templateLocation`, `csmFile`, `csmParametersFile`, `overrideParameters`, and `deploymentMode`.

yml

```yml
variables:
  ARM_PASS: $(adminPass)
```

```yaml
trigger:
- none

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: CopyFiles@2
  inputs:
    SourceFolder: 'quickstarts/microsoft.web/webapp-linux-managed-
mysql/'
    Contents: '**'
    TargetFolder: '$(Build.ArtifactStagingDirectory)'

- task: AzureResourceManagerTemplateDeployment@3
  inputs:
    deploymentScope: 'Resource Group'
    azureResourceManagerConnection: '<your-resource-manager-connection>'
    subscriptionId: '<your-subscription-id>'
    action: 'DeleteRG'
    resourceGroupName: 'ARMPipelinesLAMP-rg'
    location: ''<your-closest-location>'
```

Next steps

Create your first ARM template

Why data pipelines?

Article • 01/27/2023

Azure DevOps Services

You can use data pipelines to:

- Ingest data from various data sources
- Process and transform the data
- Save the processed data to a staging location for others to consume

Data pipelines in the enterprise can evolve into more complicated scenarios with multiple source systems and supporting various downstream applications.

Data pipelines provide:

- Consistency: Data pipelines transform data into a consistent format for users to consume
- Error reduction: Automated data pipelines eliminate human errors when manipulating data
- Efficiency: Data professionals save time spent on data processing transformation. Saving time allows then to focus on their core job function - getting the insight out of the data and helping business makes better decisions

What is CI/CD?

Continuous integration and continuous delivery (CI/CD) is a software development approach where all developers work together on a shared repository of code – and as changes are made, there are automated build process for detecting code issues. The outcome is a faster development life cycle and a lower error rate.

What is a CI/CD data pipeline and why does it matter for data science?

The building of machine learning models is similar to traditional software development in the sense that the data scientist needs to write code to train and score machine learning models.

Unlike traditional software development where the product is based on code, data science machine learning models are based on both the code (algorithm, hyper parameters) and the data used to train the model. That's why most data scientists will tell you that they spend 80% of the time doing data preparation, cleaning and feature engineering.

To complicate the matter even further – to ensure the quality of the machine learning models, techniques such as A/B testing are used. With A/B testing, there could be multiple machine learning models being used concurrently. There's usually one control model and one or more treatment models for comparison – so that the model performance can be compared and maintained. Having multiple models adds another layer of complexity for the CI/CD of machine learning models.

Having a CI/CD data pipeline is crucial for the data science team to deliver the machine learning models to the business in a timely and quality manner.

Next steps

Build a data pipeline with Azure

Build a data pipeline by using Azure Data Factory, DevOps, and machine learning

Article • 06/20/2023

Azure DevOps Services

Get started building a data pipeline with data ingestion, data transformation, and model training.

Learn how to grab data from a CSV (comma-separated values) file and save the data to Azure Blob Storage. Transform the data and save it to a staging area. Then train a machine learning model by using the transformed data. Write the model to blob storage as a Python pickle file.

Prerequisites

Before you begin, you need:

- An Azure account that has an active subscription. Create an account for free.
- An active Azure DevOps organization. Sign up for Azure Pipelines.
  - The **Administrator** role for service connections in your Azure DevOps project. Learn how to add the Administrator role.
- Data from sample.csv.
- Access to the data pipeline solution in GitHub.
- DevOps for Azure Databricks.

Provision Azure resources

1. Sign in to the Azure portal.

2. From the menu, select the **Cloud Shell** button. When you're prompted, select the **Bash** experience.

> **Note**

You'll need an Azure Storage resource to persist any files that you create in Azure Cloud Shell. When you first open Cloud Shell, you're prompted to create a resource group, storage account, and Azure Files share. This setup is automatically used for all future Cloud Shell sessions.

Select an Azure region

A *region* is one or more Azure datacenters within a geographic location. East US, West US, and North Europe are examples of regions. Every Azure resource, including an App Service instance, is assigned a region.

To make commands easier to run, start by selecting a default region. After you specify the default region, later commands use that region unless you specify a different region.

1. In Cloud Shell, run the following `az account list-locations` command to list the regions that are available from your Azure subscription.

   Azure CLI

   ```
   az account list-locations \
     --query "[].{Name: name, DisplayName: displayName}" \
     --output table
   ```

2. From the `Name` column in the output, choose a region that's close to you. For example, choose `asiapacific` or `westus2`.

3. Run `az config` to set your default region. In the following example, replace `<REGION>` with the name of the region you chose.

   Azure CLI

   ```
   az config set defaults.location=<REGION>
   ```

   The following example sets `westus2` as the default region.

   Azure CLI

   ```
   az config set defaults.location=westus2
   ```

Create Bash variables

1. In Cloud Shell, generate a random number. You'll use this number to create globally unique names for certain services in the next step.

```bash
resourceSuffix=$RANDOM
```

2. Create globally unique names for your storage account and key vault. The following commands use double quotation marks, which instruct Bash to interpolate the variables by using the inline syntax.

```bash
storageName="datacicd${resourceSuffix}"
keyVault="keyvault${resourceSuffix}"
```

3. Create one more Bash variable to store the names and the region of your resource group. In the following example, replace `<REGION>` with the region that you chose for the default region.

```bash
rgName='data-pipeline-cicd-rg'
region='<REGION>'
```

4. Create variable names for your Azure Data Factory and Azure Databricks instances.

```bash
datafactorydev='data-factory-cicd-dev'
datafactorytest='data-factory-cicd-test'
databricksname='databricks-cicd-ws'
```

Create Azure resources

1. Run the following `az group create` command to create a resource group by using `rgName`.

```azure cli
az group create --name $rgName
```

2. Run the following `az storage account create` command to create a new storage account.

Azure CLI

```
az storage account create \
    --name $storageName \
    --resource-group $rgName \
    --sku Standard_RAGRS \
    --kind StorageV2
```

3. Run the following `az storage container create` command to create two containers, `rawdata` and `prepareddata`.

Azure CLI

```
az storage container create -n rawdata --account-name $storageName
az storage container create -n prepareddata --account-name $storageName
```

4. Run the following `az keyvault create` command to create a new key vault.

Azure CLI

```
az keyvault create \
    --name $keyVault \
    --resource-group $rgName
```

5. Create a new data factory by using the portal UI or Azure CLI:

   - Name: `data-factory-cicd-dev`
   - Version: `V2`
   - Resource group: `data-pipeline-cicd-rg`
   - Location: Your closest location
   - Clear the selection for **Enable Git**.

   a. Add the Azure Data Factory extension.

   Azure CLI

   ```
   az extension add --name datafactory
   ```

   b. Run the following `az datafactory create` command to create a new data factory.

Azure CLI

```
az datafactory create \
    --name data-factory-cicd-dev \
    --resource-group $rgName
```

c. Copy the subscription ID. Your data factory will use this ID later.

6. Create a second data factory by using the portal UI or the Azure CLI. You'll use this data factory for testing.

   - Name: `data-factory-cicd-test`
   - Version: `V2`
   - Resource group: `data-pipeline-cicd-rg`
   - Location: Your closest location
   - Clear the selection for **Enable GIT**.

   a. Run the following `az datafactory create` command to create a new data factory for testing.

   Azure CLI

   ```
   az datafactory create \
       --name data-factory-cicd-test \
       --resource-group $rgName
   ```

   b. Copy the subscription ID. Your data factory will use this ID later.

7. Add a new Azure Databricks service:

   - Resource group: `data-pipeline-cicd-rg`
   - Workspace name: `databricks-cicd-ws`
   - Location: Your closest location

   a. Add the Azure Databricks extension if it's not already installed.

   Azure CLI

   ```
   az extension add --name databricks
   ```

   b. Run the following `az databricks workspace create` command to create a new workspace.

   Azure CLI

```
az databricks workspace create \
    --resource-group $rgName \
    --name databricks-cicd-ws  \
    --location eastus2  \
    --sku trial
```

c. Copy the subscription ID. Your Databricks service will use this ID later.

Upload data to your storage container

1. In the Azure portal, open your storage account in the `data-pipeline-cicd-rg` resource group.
2. Go to **Blob Service** > **Containers**.
3. Open the `prepareddata` container.
4. Upload the sample.csv file.

Set up Key Vault

You'll use Azure Key Vault to store all connection information for your Azure services.

Create a Databricks personal access token

1. In the Azure portal, go Databricks and then open your workspace.
2. In the Azure Databricks UI, create and copy a personal access token.

Copy the account key and connection string for your storage account

1. Go to your storage account.
2. Open **Access keys**.
3. Copy the first key and connection string.

Save values to Key Vault

1. Create three secrets:

    - databricks-token: `your-databricks-pat`
    - StorageKey: `your-storage-key`
    - StorageConnectString: `your-storage-connection`

2. Run the following `az keyvault secret set` command to add secrets to your key vault.

Azure CLI

```
az keyvault secret set --vault-name "$keyVault" --name "databricks-
token" --value "your-databricks-pat"
az keyvault secret set --vault-name "$keyVault" --name "StorageKey" --
value "your-storage-key"
az keyvault secret set --vault-name "$keyVault" --name
"StorageConnectString" --value "your-storage-connection"
```

Import the data pipeline solution

1. Sign in to your Azure DevOps organization and then go to your project.
2. Go to **Repos** and then import your forked version of the GitHub repository. For more information, see Import a Git repo into your project.

Add an Azure Resource Manager service connection

1. Create an Azure Resource Manager service connection.
2. Select **Service Principal (automatic)**.
3. Choose the **data-pipeline-cicd-rg** resource group.
4. Name the service connection `azure_rm_connection`.
5. Select **Grant access permission to all pipelines**. You'll need to have the Service Connections Administrator role to select this option.

Add pipeline variables

1. Create a new variable group named `datapipeline-vg`.

2. Add the Azure DevOps extension if it isn't already installed.

Azure CLI

```
az extension add --name azure-devops
```

3. Sign in to your Azure DevOps organization.

Azure CLI

```
az devops login --org https://dev.azure.com/<yourorganizationname>
```

```
az pipelines variable-group create --name datapipeline-vg -p
<yourazuredevopsprojectname> --variables \
                                    "LOCATION=$region" \
                                    "RESOURCE_GROUP=$rgName" \
                                    "DATA_FACTORY_NAME=$datafactorydev"
\

"DATA_FACTORY_DEV_NAME=$datafactorydev" \

"DATA_FACTORY_TEST_NAME=$datafactorytest" \
                                    "ADF_PIPELINE_NAME=DataPipeline" \
                                    "DATABRICKS_NAME=$databricksname" \

"AZURE_RM_CONNECTION=azure_rm_connection" \
                                    "DATABRICKS_URL=<URL copied from
Databricks in Azure portal>" \
                                    "STORAGE_ACCOUNT_NAME=$storageName"
\
                                    "STORAGE_CONTAINER_NAME=rawdata"
```

4. Create a second variable group named `keys-vg`. This group will pull data variables from Key Vault.

5. Select **Link secrets from an Azure key vault as variables**. For more information, see Link secrets from an Azure key vault.

6. Authorize the Azure subscription.

7. Choose all of the available secrets to add as variables (`databricks-token`, `StorageConnectString`, `StorageKey`).

Configure Azure Databricks and Azure Data Factory

Follow the steps in the next sections to set up Azure Databricks and Azure Data Factory.

Create testscope in Azure Databricks

1. In the Azure portal, go to **Key vault** > **Properties**.
2. Copy the **DNS Name** and **Resource ID**.
3. In your Azure Databricks workspace, create a secret scope named `testscope`.

Add a new cluster in Azure Databricks

1. In the Azure Databricks workspace, go to **Clusters**.
2. Select **Create Cluster**.
3. Name and save your new cluster.
4. Select your new cluster name.
5. In the URL string, copy the content between `/clusters/` and `/configuration`. For example, in the string `clusters/0306-152107-daft561/configuration`, you would copy `0306-152107-daft561`.
6. Save this string to use later.

Set up your code repository in Azure Data Factory

1. In Azure Data Factory, go to **Author & Monitor**. For more information, see Create a data factory.
2. Select **Set up code repository** and then connect your repo.

    - Repository type: Azure DevOps Git
    - Azure DevOps organization: Your active account
    - Project name: Your Azure DevOps data pipeline project
    - Git repository name: **Use existing**.
        - Select the **main** branch for collaboration.
        - Set **/azure-data-pipeline/factorydata** as the root folder.
    - Branch to import resource into: Select **Use existing** and **main**.

Link Azure Data Factory to your key vault

1. In the Azure portal UI, open the key vault.
2. Select **Access policies**.
3. Select **Add Access Policy**.
4. For **Configure from template**, select **Key & Secret Management**.
5. In **Select principal**, search for the name of your development data factory and add it.
6. Select **Add** to add your access policies.
7. Repeat these steps to add an access policy for the test data factory.

Update the key vault linked service in Azure Data Factory

1. Go to **Manage** > **Linked services**.
2. Update the Azure key vault to connect to your subscription.

Update the storage linked service in Azure Data Factory

1. Go to **Manage** > **Linked services**.
2. Update the Azure Blob Storage value to connect to your subscription.

Update the Azure Databricks linked service in Azure Data Factory

1. Go to **Manage** > **Linked services**.
2. Update the Azure Databricks value to connect to your subscription.
3. For the **Existing Cluster ID**, enter the cluster value you saved earlier.

Test and publish the data factory

1. In Azure Data Factory, go to **Edit**.
2. Open `DataPipeline`.
3. Select **Variables**.
4. Verify that the `storage_account_name` refers to your storage account in the Azure portal. Update the default value if necessary. Save your changes.
5. Select **Validate** to verify `DataPipeline`.
6. Select **Publish** to publish data-factory assets to the `adf_publish` branch of your repository.

Run the CI/CD pipeline

Follow these steps to run the continuous integration and continuous delivery (CI/CD) pipeline:

1. Go to the **Pipelines** page. Then choose the action to create a new pipeline.
2. Select **Azure Repos Git** as the location of your source code.
3. When the list of repositories appears, select your repository.
4. As you set up your pipeline, select **Existing Azure Pipelines YAML file**. Choose the YAML file: **/azure-data-pipeline/data_pipeline_ci_cd.yml**.
5. Run the pipeline. If your pipeline hasn't been run before, you might need to give permission to access a resource during the run.

Clean up resources

If you're not going to continue to use this application, delete your data pipeline by following these steps:

1. Delete the `data-pipeline-cicd-rg` resource group.
2. Delete your Azure DevOps project.

Next steps

Learn more about data in Azure Data Factory

Use Azure Pipelines with Azure Machine Learning

Article • 09/29/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

You can use an Azure DevOps pipeline to automate the machine learning lifecycle. Some of the operations you can automate are:

- Data preparation (extract, transform, load operations)
- Training machine learning models with on-demand scale-out and scale-up
- Deployment of machine learning models as public or private web services
- Monitoring deployed machine learning models (such as for performance or data-drift analysis)

This article teaches you how to create an Azure Pipeline that builds and deploys a machine learning model to Azure Machine Learning.

This tutorial uses Azure Machine Learning Python SDK v2 and Azure CLI ML extension v2.

Prerequisites

- Complete the Create resources to get started to:
  - Create a workspace
- Create a cloud-based compute cluster to use for training your model
- Azure Machine Learning extension for Azure Pipelines. This extension can be installed from the Visual Studio marketplace at https://marketplace.visualstudio.com/items?itemName=ms-air-aiagility.azureml-v2 .

Step 1: Get the code

Fork the following repo at GitHub:

```
https://github.com/azure/azureml-examples
```

Step 2: Sign in to Azure Pipelines

Sign-in to Azure Pipelines. After you sign in, your browser goes to `https://dev.azure.com/my-organization-name` and displays your Azure DevOps dashboard.

Within your selected organization, create a *project*. If you don't have any projects in your organization, you see a **Create a project to get started** screen. Otherwise, select the **New Project** button in the upper-right corner of the dashboard.

Step 3: Create a service connection

You can use an existing service connection.

Azure Resource Manager

You need an Azure Resource Manager connection to authenticate with Azure portal.

1. In Azure DevOps, select **Project Settings** and open the **Service connections** page.
2. Choose **+ New service connection** and select **Azure Resource Manager**.
3. Select the default authentication method, **Service principal (automatic)**.
4. Create your service connection. Set your preferred scope level, subscription, resource group, and connection name.

Step 4: Create a pipeline

1. Go to **Pipelines**, and then select **New pipeline**.
2. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.
3. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.
4. When you see the list of repositories, select your repository.
5. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.
6. Select the **Starter pipeline**. You'll update the starter pipeline template.

Step 5: Build your YAML pipeline to submit the Azure Machine Learning job

Delete the starter pipeline and replace it with the following YAML code. In this pipeline, you'll:

- Use the Python version task to set up Python 3.8 and install the SDK requirements.
- Use the Bash task to run bash scripts for the Azure Machine Learning SDK and CLI.
- Use the Azure CLI task to submit an Azure Machine Learning job.

Select the following tabs depending on whether you're using an Azure Resource Manager service connection or a generic service connection. In the pipeline YAML, replace the value of variables with your resources.

Using Azure Resource Manager service connection

YAML

```yaml
name: submit-azure-machine-learning-job

trigger:
- none

variables:
  service-connection: 'machine-learning-connection' # replace with your
service connection name
  resource-group: 'machinelearning-rg' # replace with your resource
group name
  workspace: 'docs-ws' # replace with your workspace name

jobs:
- job: SubmitAzureMLJob
  displayName: Submit AzureML Job
  timeoutInMinutes: 300
  pool:
    vmImage: ubuntu-latest
  steps:
  - checkout: none
  - task: UsePythonVersion@0
    displayName: Use Python >=3.8
    inputs:
      versionSpec: '>=3.8'

  - bash: |
      set -ex

      az version
      az extension add -n ml
    displayName: 'Add AzureML Extension'

  - task: AzureCLI@2
    name: submit_azureml_job_task
    displayName: Submit AzureML Job Task
    inputs:
      azureSubscription: $(service-connection)
      workingDirectory: 'cli/jobs/pipelines-with-
components/nyc_taxi_data_regression'
```

```
      scriptLocation: inlineScript
      scriptType: bash
      inlineScript: |

      # submit component job and get the run name
      job_name=$(az ml job create --file single-job-pipeline.yml -g
$(resource-group) -w $(workspace) --query name --output tsv)

      # Set output variable for next task
      echo "##vso[task.setvariable
variable=JOB_NAME;isOutput=true;]$job_name"
```

Step 6: Wait for Azure Machine Learning job to complete

In step 5, you added a job to submit an Azure Machine Learning job. In this step, you add another job that waits for the Azure Machine Learning job to complete.

Using Azure Resource Manager service connection

If you're using an Azure Resource Manager service connection, you can use the "Machine Learning" extension. You can search this extension in the Azure DevOps extensions Marketplace or go directly to the extension. Install the "Machine Learning" extension.

> **Important**
>
> Don't install the **Machine Learning (classic)** extension by mistake; it's an older extension that doesn't provide the same functionality.

In the Pipeline review window, add a Server Job. In the steps part of the job, select **Show assistant** and search for **AzureML**. Select the **AzureML Job Wait** task and fill in the information for the job.

The task has four inputs: `Service Connection`, `Azure Resource Group Name`, `AzureML Workspace Name` and `AzureML Job Name`. Fill these inputs. The resulting YAML for these steps is similar to the following example:

> **Note**

- The Azure Machine Learning job wait task runs on a **server job**, which doesn't use up expensive agent pool resources and requires no additional charges. Server jobs (indicated by `pool: server`) run on the same machine as your pipeline. For more information, see **Server jobs**.
- One Azure Machine Learning job wait task can only wait on one job. You'll need to set up a separate task for each job that you want to wait on.
- The Azure Machine Learning job wait task can wait for a maximum of 2 days. This is a hard limit set by Azure DevOps Pipelines.

yml

```yml
- job: WaitForAzureMLJobCompletion
  displayName: Wait for AzureML Job Completion
  pool: server
  timeoutInMinutes: 0
  dependsOn: SubmitAzureMLJob
  variables:
    # We are saving the name of azureml job submitted in previous step
to a variable and it will be used as an inut to the AzureML Job Wait
task
    azureml_job_name_from_submit_job: $[
dependencies.SubmitAzureMLJob.outputs['submit_azureml_job_task.AZUREML_J
OB_NAME'] ]
  steps:
  - task: AzureMLJobWaitTask@1
    inputs:
      serviceConnection: $(service-connection)
      resourceGroupName: $(resource-group)
      azureMLWorkspaceName: $(workspace)
      azureMLJobName: $(azureml_job_name_from_submit_job)
```

Step 7: Submit pipeline and verify your pipeline run

Select **Save and run**. The pipeline will wait for the Azure Machine Learning job to complete, and end the task under `WaitForJobCompletion` with the same status as the Azure Machine Learning job. For example: Azure Machine Learning job `Succeeded` == Azure DevOps Task under `WaitForJobCompletion` job `Succeeded` Azure Machine Learning job `Failed` == Azure DevOps Task under `WaitForJobCompletion` job `Failed` Azure

Machine Learning job `Cancelled` == Azure DevOps Task under `WaitForJobCompletion` job `Cancelled`

> **Tip**
>
> You can view the complete Azure Machine Learning job in **Azure Machine Learning studio**.

Clean up resources

If you're not going to continue to use your pipeline, delete your Azure DevOps project. In Azure portal, delete your resource group and Azure Machine Learning instance.

Azure SQL database deployment

Article • 03/21/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

You can automatically deploy your database updates to Azure SQL database after every successful build.

DACPAC

The simplest way to deploy a database is to create data-tier package or DACPAC. DACPACs can be used to package and deploy schema changes and data. You can create a DACPAC using the **SQL database project** in Visual Studio.

YAML

To deploy a DACPAC to an Azure SQL database, add the following snippet to your azure-pipelines.yml file.

YAML

```yaml
- task: SqlAzureDacpacDeployment@1
  displayName: Execute Azure SQL : DacpacTask
  inputs:
    azureSubscription: '<Azure service connection>'
    ServerName: '<Database server name>'
    DatabaseName: '<Database name>'
    SqlUsername: '<SQL user name>'
    SqlPassword: '<SQL user password>'
    DacpacFile: '<Location of Dacpac file in $(Build.SourcesDirectory)
after compilation>'
```

See also authentication information when using the Azure SQL Database Deployment task.

SQL scripts

Instead of using a DACPAC, you can also use SQL scripts to deploy your database. Here's a simple example of a SQL script that creates an empty database.

SQL

```
  USE [main]
  GO
  IF NOT EXISTS (SELECT name FROM main.sys.databases WHERE name =
N'DatabaseExample')
  CREATE DATABASE [DatabaseExample]
  GO
```

To run SQL scripts as part of a pipeline, you'll need Azure PowerShell scripts to create and remove firewall rules in Azure. Without the firewall rules, the Azure Pipelines agent can't communicate with Azure SQL Database.

The following PowerShell script creates firewall rules. You can check in this script as `SetAzureFirewallRule.ps1` into your repository.

ARM

PowerShell

```
[CmdletBinding(DefaultParameterSetName = 'None')]
param
(
  [String] [Parameter(Mandatory = $true)] $ServerName,
  [String] [Parameter(Mandatory = $true)] $ResourceGroupName,
  [String] $FirewallRuleName = "AzureWebAppFirewall"
)
$agentIP = (New-Object
net.webclient).downloadstring("https://api.ipify.org")
New-AzSqlServerFirewallRule -ResourceGroupName $ResourceGroupName -
ServerName $ServerName -FirewallRuleName $FirewallRuleName -StartIPAddress
$agentIp -EndIPAddress $agentIP
```

Classic

PowerShell

```
[CmdletBinding(DefaultParameterSetName = 'None')]
param
(
  [String] [Parameter(Mandatory = $true)] $ServerName,
  [String] [Parameter(Mandatory = $true)] $ResourceGroupName,
  [String] $FirewallRuleName = "AzureWebAppFirewall"
)

$ErrorActionPreference = 'Stop'
```

```
function New-AzureSQLServerFirewallRule {
  $agentIP = (New-Object
net.webclient).downloadstring("https://api.ipify.org")
  New-AzureSqlDatabaseServerFirewallRule -StartIPAddress $agentIp -
EndIPAddress $agentIp -RuleName $FirewallRuleName -ServerName $ServerName
}

function Update-AzureSQLServerFirewallRule{
  $agentIP= (New-Object
net.webclient).downloadstring("https://api.ipify.org")
  Set-AzureSqlDatabaseServerFirewallRule -StartIPAddress $agentIp -
EndIPAddress $agentIp -RuleName $FirewallRuleName -ServerName $ServerName
}

if ((Get-AzureSqlDatabaseServerFirewallRule -ServerName $ServerName -
RuleName $FirewallRuleName -ErrorAction SilentlyContinue) -eq $null)
{
  New-AzureSQLServerFirewallRule
}
else
{
  Update-AzureSQLServerFirewallRule
}
```

The following PowerShell script removes firewall rules. You can check in this script as `RemoveAzureFirewallRule.ps1` into your repository.

ARM

PowerShell

```
[CmdletBinding(DefaultParameterSetName = 'None')]
param
(
  [String] [Parameter(Mandatory = $true)] $ServerName,
  [String] [Parameter(Mandatory = $true)] $ResourceGroupName,
  [String] $FirewallRuleName = "AzureWebAppFirewall"
)
Remove-AzSqlServerFirewallRule -ServerName $ServerName -FirewallRuleName
$FirewallRuleName -ResourceGroupName $ResourceGroupName
```

Classic

PowerShell

```
[CmdletBinding(DefaultParameterSetName = 'None')]
param
(
  [String] [Parameter(Mandatory = $true)] $ServerName,
```

```
  [String] [Parameter(Mandatory = $true)] $ResourceGroupName,
  [String] $FirewallRuleName = "AzureWebAppFirewall"
)

$ErrorActionPreference = 'Stop'

if ((Get-AzureSqlDatabaseServerFirewallRule -ServerName $ServerName -
RuleName $FirewallRuleName -ErrorAction SilentlyContinue))
{
  Remove-AzureSqlDatabaseServerFirewallRule -RuleName $FirewallRuleName -
ServerName $ServerName
}
```

YAML

Add the following to your azure-pipelines.yml file to run a SQL script.

YAML

```
variables:
  AzureSubscription: '<SERVICE_CONNECTION_NAME>'
  ResourceGroupName: '<RESOURCE_GROUP_NAME>'
  ServerName: '<DATABASE_SERVER_NAME>'
  ServerFqdn: '<DATABASE_FQDN>'
  DatabaseName: '<DATABASE_NAME>'
  AdminUser: '<DATABASE_USERNAME>'
  AdminPassword: '<DATABASE_PASSWORD>'
  SQLFile: '<LOCATION_OF_SQL_FILE_IN_$(Build.SourcesDirectory)>'

steps:
- task: AzurePowerShell@5
  displayName: 'Azure PowerShell script'
  inputs:
    azureSubscription: '$(AzureSubscription)'
    ScriptType: filePath
    ScriptPath:
'$(Build.SourcesDirectory)\scripts\SetAzureFirewallRule.ps1'
    ScriptArguments: '-ServerName $(ServerName) -ResourceGroupName
$(ResourceGroupName)'
    azurePowerShellVersion: LatestVersion

- task: CmdLine@2
  displayName: Run Sqlcmd
  inputs:
    filename: Sqlcmd
    arguments: '-S $(ServerFqdn) -U $(AdminUser) -P $(AdminPassword) -d
$(DatabaseName) -i $(SQLFile)'

- task: AzurePowerShell@5
  displayName: 'Azure PowerShell script'
  inputs:
```

```
    azureSubscription: '$(AzureSubscription)'
    ScriptType: filePath
    ScriptPath:
'$(Build.SourcesDirectory)\scripts\RemoveAzureFirewallRule.ps1'
    ScriptArguments: '-ServerName $(ServerName) -ResourceGroupName
$(ResourceGroupName)'
    azurePowerShellVersion: LatestVersion
```

Azure service connection

The **Azure SQL Database Deployment** task is the primary mechanism to deploy a database to Azure. This task, as with other built-in Azure tasks, requires an Azure service connection as an input. The Azure service connection stores the credentials to connect from Azure Pipelines or TFS to Azure.

The easiest way to get started with this task is to be signed in as a user that owns both the Azure DevOps organization and the Azure subscription. In this case, you won't have to manually create the service connection. Otherwise, to learn how to create an Azure service connection, see Create an Azure service connection.

Deploying conditionally

You may choose to deploy only certain builds to your Azure database.

YAML

To do this in YAML, you can use one of these techniques:

- Isolate the deployment steps into a separate job, and add a condition to that job.
- Add a condition to the step.

The following example shows how to use step conditions to deploy only those builds that originate from main branch.

YAML

```
- task: SqlAzureDacpacDeployment@1
  condition: and(succeeded(), eq(variables['Build.SourceBranch'],
'refs/heads/main'))
  inputs:
    azureSubscription: '<Azure service connection>'
    ServerName: '<Database server name>'
```

```
    DatabaseName: '<Database name>'
    SqlUsername: '<SQL user name>'
    SqlPassword: '<SQL user password>'
    DacpacFile: '<Location of Dacpac file in $(Build.SourcesDirectory)
after compilation>'
```

To learn more about conditions, see Specify conditions.

More SQL actions

SQL Azure Dacpac Deployment may not support all SQL server actions that you want to perform. In these cases, you can simply use PowerShell or command-line scripts to run the commands you need. This section shows some of the common use cases for invoking the SqlPackage.exe tool. As a prerequisite to running this tool, you must use a self-hosted agent and have the tool installed on your agent.

> ⓘ **Note**
>
> If you execute **SQLPackage** from the folder where it is installed, you must prefix the path with `&` and wrap it in double-quotes.

Basic Syntax

`<Path of SQLPackage.exe> <Arguments to SQLPackage.exe>`

You can use any of the following SQL scripts depending on the action that you want to perform

Extract

Creates a database snapshot (.dacpac) file from a live SQL server or Microsoft Azure SQL Database.

Command Syntax:

command

```
SqlPackage.exe /TargetFile:"<Target location of dacpac file>"
/Action:Extract
/SourceServerName:"<ServerName>.database.windows.net"
/SourceDatabaseName:"<DatabaseName>" /SourceUser:"<Username>"
/SourcePassword:"<Password>"
```

or

```command
SqlPackage.exe /action:Extract /tf:"<Target location of dacpac file>"
/SourceConnectionString:"Data Source=ServerName;Initial
Catalog=DatabaseName;Integrated Security=SSPI;Persist Security Info=False;"
```

Example:

```command
SqlPackage.exe /TargetFile:"C:\temp\test.dacpac" /Action:Extract
/SourceServerName:"DemoSqlServer.database.windows.net"
 /SourceDatabaseName:"Testdb" /SourceUser:"ajay"
/SourcePassword:"SQLPassword"
```

Help:

```command
sqlpackage.exe /Action:Extract /?
```

Publish

Incrementally updates a database schema to match the schema of a source .dacpac file. If the database doesn't exist on the server, the publish operation will create it. Otherwise, an existing database will be updated.

Command Syntax:

```command
SqlPackage.exe /SourceFile:"<Dacpac file location>" /Action:Publish
/TargetServerName:"<ServerName>.database.windows.net"
/TargetDatabaseName:"<DatabaseName>" /TargetUser:"<Username>"
/TargetPassword:"<Password> "
```

Example:

```command
SqlPackage.exe /SourceFile:"E:\dacpac\ajyadb.dacpac" /Action:Publish
/TargetServerName:"DemoSqlServer.database.windows.net"
```

```
/TargetDatabaseName:"Testdb4" /TargetUser:"ajay"
/TargetPassword:"SQLPassword"
```

Help:

```command
sqlpackage.exe /Action:Publish /?
```

Export

Exports a live database, including database schema and user data, from SQL Server or Microsoft Azure SQL Database to a BACPAC package (.bacpac file).

Command Syntax:

```command
SqlPackage.exe /TargetFile:"<Target location for bacpac file>"
/Action:Export /SourceServerName:"<ServerName>.database.windows.net"
/SourceDatabaseName:"<DatabaseName>" /SourceUser:"<Username>"
/SourcePassword:"<Password>"
```

Example:

```command
SqlPackage.exe /TargetFile:"C:\temp\test.bacpac" /Action:Export
/SourceServerName:"DemoSqlServer.database.windows.net"
/SourceDatabaseName:"Testdb" /SourceUser:"ajay"
/SourcePassword:"SQLPassword"
```

Help:

```command
sqlpackage.exe /Action:Export /?
```

Import

Imports the schema and table data from a BACPAC package into a new user database in an instance of SQL Server or Microsoft Azure SQL Database.

Command Syntax:

command

```
SqlPackage.exe /SourceFile:"<Bacpac file location>" /Action:Import
/TargetServerName:"<ServerName>.database.windows.net"
/TargetDatabaseName:"<DatabaseName>" /TargetUser:"<Username>"
/TargetPassword:"<Password>"
```

Example:

command

```
SqlPackage.exe /SourceFile:"C:\temp\test.bacpac" /Action:Import
/TargetServerName:"DemoSqlServer.database.windows.net"
/TargetDatabaseName:"Testdb" /TargetUser:"ajay"
/TargetPassword:"SQLPassword"
```

Help:

command

```
sqlpackage.exe /Action:Import /?
```

DeployReport

Creates an XML report of the changes that would be made by a publish action.

Command Syntax:

command

```
SqlPackage.exe /SourceFile:"<Dacpac file location>" /Action:DeployReport
/TargetServerName:"<ServerName>.database.windows.net"
/TargetDatabaseName:"<DatabaseName>" /TargetUser:"<Username>"
/TargetPassword:"<Password>" /OutputPath:"<Output XML file path for deploy
report>"
```

Example:

command

```
SqlPackage.exe /SourceFile:"E: \dacpac\ajyadb.dacpac" /Action:DeployReport
/TargetServerName:"DemoSqlServer.database.windows.net"
/TargetDatabaseName:"Testdb" /TargetUser:"ajay"
/TargetPassword:"SQLPassword" /OutputPath:"C:\temp\deployReport.xml"
```

Help:

```command
sqlpackage.exe /Action:DeployReport /?
```

DriftReport

Creates an XML report of the changes that have been made to a registered database since it was last registered.

Command Syntax:

```command
SqlPackage.exe /Action:DriftReport /TargetServerName:"
<ServerName>.database.windows.net" /TargetDatabaseName:"<DatabaseName>"
/TargetUser:"<Username>" /TargetPassword:"<Password>" /OutputPath:"<Output
XML file path for drift report>"
```

Example:

```command
SqlPackage.exe /Action:DriftReport
/TargetServerName:"DemoSqlServer.database.windows.net"
/TargetDatabaseName:"Testdb"
/TargetUser:"ajay" /TargetPassword:"SQLPassword"
/OutputPath:"C:\temp\driftReport.xml"
```

Help:

```command
sqlpackage.exe /Action:DriftReport /?
```

Script

Creates a Transact-SQL incremental update script that updates the schema of a target to match the schema of a source.

Command Syntax:

```command
```

```
SqlPackage.exe /SourceFile:"<Dacpac file location>" /Action:Script
/TargetServerName:"<ServerName>.database.windows.net"
/TargetDatabaseName:"<DatabaseName>" /TargetUser:"<Username>"
/TargetPassword:"<Password>" /OutputPath:"<Output SQL script file path>"
```

Example:

command

```
SqlPackage.exe /Action:Script /SourceFile:"E:\dacpac\ajyadb.dacpac"
/TargetServerName:"DemoSqlServer.database.windows.net"
/TargetDatabaseName:"Testdb" /TargetUser:"ajay"
/TargetPassword:"SQLPassword" /OutputPath:"C:\temp\test.sql"
/Variables:StagingDatabase="Staging DB Variable value"
```

Help:

command

```
sqlpackage.exe /Action:Script /?
```

Azure Pipelines for Azure Database for MySQL single server

Article • 03/28/2023

Get started with Azure Database for MySQL by deploying a database update with Azure Pipelines. Azure Pipelines lets you build, test, and deploy with continuous integration (CI) and continuous delivery (CD) using Azure DevOps.

You'll use the Azure Database for MySQL Deployment task. The Azure Database for MySQL Deployment task only works with Azure Database for MySQL single server.

Prerequisites

Before you begin, you need:

- An Azure account with an active subscription. Create an account for free.
- An active Azure DevOps organization. Sign up for Azure Pipelines.
- A GitHub repository that you can use for your pipeline. If you don't have an existing repository, see Create your first pipeline.

This quickstart uses the resources created in either of these guides as a starting point:

- Create an Azure Database for MySQL server using Azure portal
- Create an Azure Database for MySQL server using Azure CLI

Create your pipeline

You'll use the basic starter pipeline as a basis for your pipeline.

1. Sign in to your Azure DevOps organization and go to your project.

2. In your project, navigate to the **Pipelines** page. Then choose the action to create a new pipeline.

3. Walk through the steps of the wizard by first selecting GitHub as the location of your source code.

4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.

5. When the list of repositories appears, select your desired repository.

6. Azure Pipelines will analyze your repository and offer configuration options. Select **Starter pipeline**.

Create a secret

You'll need to know your database server name, SQL username, and SQL password to use with the Azure Database for MySQL Deployment task.

For security, you'll want to save your SQL password as a secret variable in the pipeline settings UI for your pipeline.

1. Go to the **Pipelines** page, select the appropriate pipeline, and then select **Edit**.

2. Select **Variables**.

3. Add a new variable named `SQLpass` and select **Keep this value secret** to encrypt and save the variable.

4. Select **Ok** and **Save** to add the variable.

Verify permissions for your database

To access your MySQL database with Azure Pipelines, you need to set your database to accept connections from all Azure resources.

1. In the Azure portal, open your database resource.
2. Select **Connection security**.
3. Toggle **Allow access to Azure services** to **Yes**.

Add the Azure Database for MySQL Deployment task

In this example, we'll create a new databases named `quickstartdb` and add an inventory table. The inline SQL script will:

- Delete `quickstartdb` if it exists and create a new `quickstartdb` database.
- Delete the table `inventory` if it exists and creates a new `inventory` table.
- Insert three rows into `inventory`.
- Show all the rows.
- Update the value of the first row in `inventory`.
- Delete the second row in `inventory`.

You'll need to replace the following values in your deployment task.

Input	Description	Example
`azureSubscription`	Authenticate with your Azure Subscription with a service connection.	`My Subscription`
`ServerName`	The name of your Azure Database for MySQL server.	`fabrikam.mysql.database.azure.com`
`SqlUsername`	The user name of your Azure Database for MySQL.	`mysqladmin@fabrikam`
`SqlPassword`	The password for the username. This should be defined as a secret variable.	`$(SQLpass)`

YAML

```yaml
trigger:
- main
```

```yaml
pool:
  vmImage: ubuntu-latest

steps:
- task: AzureMysqlDeployment@1
  inputs:
    azureSubscription: '<your-subscription>
    ServerName: '<db>.mysql.database.azure.com'
    SqlUsername: '<username>@<db>'
    SqlPassword: '$(SQLpass)'
    TaskNameSelector: 'InlineSqlTask'
    SqlInline: |
      DROP DATABASE IF EXISTS quickstartdb;
      CREATE DATABASE quickstartdb;
      USE quickstartdb;

      -- Create a table and insert rows
      DROP TABLE IF EXISTS inventory;
      CREATE TABLE inventory (id serial PRIMARY KEY, name VARCHAR(50),
quantity INTEGER);
      INSERT INTO inventory (name, quantity) VALUES ('banana', 150);
      INSERT INTO inventory (name, quantity) VALUES ('orange', 154);
      INSERT INTO inventory (name, quantity) VALUES ('apple', 100);

      -- Read
      SELECT * FROM inventory;

      -- Update
      UPDATE inventory SET quantity = 200 WHERE id = 1;
      SELECT * FROM inventory;

      -- Delete
      DELETE FROM inventory WHERE id = 2;
      SELECT * FROM inventory;
    IpDetectionMethod: 'AutoDetect'
```

Deploy and verify resources

Select **Save and run** to deploy your pipeline. The pipeline job will be launched and after few minutes, the job status should indicate `Success`.

You can verify that your pipeline ran successfully within the `AzureMysqlDeployment` task in the pipeline run.

Open the task and verify that the last two entries show two rows in `inventory`. There are two rows because the second row has been deleted.

```
33  id      name    quantity
34  1       banana  150
35  2       orange  154
36  3       apple   100
37  id      name    quantity
38  1       banana  200
39  2       orange  154
40  3       apple   100
41  id      name    quantity
42  1       banana  200
43  3       apple   100
44  Finishing: AzureMysqlDeployment
```

Clean up resources

When you're done working with your pipeline, delete `quickstartdb` in your Azure Database for MySQL. You can also delete the deployment pipeline you created.

Next steps

Tutorial: Build an ASP.NET Core and Azure SQL Database app in Azure App Service

Tutorial: Deploy a Java app to a virtual machine scale set

Article • 05/30/2023

Azure DevOps Services

A virtual machine scale set lets you deploy and manage identical, autoscaling virtual machines.

VMs are created as needed in a scale set. You define rules to control how and when VMs are added or removed from the scale set. These rules can be triggered based on metrics such as CPU load, memory usage, or network traffic.

In this tutorial, you build a Java app and deploy it to a virtual machine scale set. You learn how to:

- ✓ Create a virtual machine scale set
- ✓ Build a machine image
- ✓ Deploy a custom image to a virtual machine scale set

Prerequisites

Before you begin, you need:

- An Azure account with an active subscription. Create an account for free.
- An active Azure DevOps organization. Sign up for Azure Pipelines.
- The Azure VM Image Builder DevOps task installed for your DevOps organization.
- A forked GitHub repo with the example project. Fork the pipelines-vmss repository.

Set up your Java pipeline

1. Sign-in to your Azure DevOps organization and go to your project.

2. Go to **Pipelines**, and then select **New pipeline**.

3. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.

4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.

5. When you see the list of repositories, select your repository.

6. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.

> When the **Configure** tab appears, select **Maven**.

Customize the pipeline

1. When your new pipeline appears, take a look at the YAML to see what it does. When you're ready, select **Save and run**.

2. You are prompted to commit a new *azure-pipelines.yml* file to your repository. After you're happy with the message, select **Save and run** again.

   If you want to watch your pipeline in action, select the build job.

   > You just created and ran a pipeline that we automatically created for you, because your code appeared to be a good match for the Maven template.

   You now have a working YAML pipeline (`azure-pipelines.yml`) in your repository that's ready for you to customize!

3. When you're ready to make changes to your pipeline, select it in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file.

Add the Copy Files and Publish Build Artifact tasks

1. Update your pipeline to include the `CopyFiles@2` task. This will create an artifact that you can deploy to your virtual machine scale set.

```
YAML
```

```
  trigger: none

  pool:
    vmImage: 'ubuntu-latest'

  steps:
- task: Maven@4
  inputs:
    mavenPomFile: 'pom.xml'
    mavenOptions: '-Xmx3072m'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.8'
    jdkArchitectureOption: 'x64'
    publishJUnitResults: true
    testResultsFiles: '**/surefire-reports/TEST-*.xml'
    goals: 'package'

- task: CopyFiles@2
  displayName: 'Copy File to: $(TargetFolder)'
  inputs:
    SourceFolder: '$(Build.SourcesDirectory)'
    Contents: |
      **/*.sh
      **/*.war
      **/*jar-with-dependencies.jar
    TargetFolder: '$(System.DefaultWorkingDirectory)/pipeline-
artifacts/'
    flattenFolders: true
```

Create a custom image and upload it to Azure

You'll need a resource group, storage account, and shared image gallery for your custom image.

1. Create a resource group with az group create. This example creates a resource group named *myVMSSResourceGroup* in the *eastus2* location:

   Azure CLI

   ```
   az group create --name myVMSSResourceGroup --location eastus2
   ```

2. Create a new storage account. This example creates a storage account, `vmssstorageaccount`.

   Azure CLI

   ```
   az storage account create \
     --name vmssstorageaccount \
   ```

```
  --resource-group myVMSSResourceGroup \
  --location eastus2 \
  --sku Standard_LRS
```

3. Create a shared image gallery.

Azure CLI

```
az sig create --resource-group myVMSSResourceGroup --gallery-name
myVMSSGallery
```

4. Create a new image gallery in the `myVMSSGallery` resource. See Create an Azure Shared Image Gallery using the portal to learn more about working with image galleries.

Azure CLI

```
az sig create --resource-group myVMSSResourceGroup --gallery-name
myVMSSGallery
```

5. Create an image definition. Copy the `id` of the new image that looks like `/subscriptions/<SUBSCRIPTION ID>/resourceGroups/<RESOURCE GROUP>/providers/Microsoft.Compute/galleries/myVMSSGallery/images/MyImage`.

Azure CLI

```
az sig image-definition create -g myVMSSResourceGroup --gallery-name
myVMSSGallery --gallery-image-definition MyImage --publisher
GreatPublisher --offer GreatOffer --sku GreatSku --os-type linux
```

Create a managed identity

1. Create a managed identity in your resources group.

Azure CLI

```
az identity create -g myVMSSResourceGroup -n myVMSSIdentity
```

2. From the output, copy the `id`. The `id` will look like `/subscriptions/<SUBSCRIPTION ID>/resourcegroups/<RESOURCE GROUP>/providers/Microsoft.ManagedIdentity/userAssignedIdentities/<USER ASSIGNED IDENTITY NAME>`.

3. Open your image gallery in the gallery and assign `myVMSSIdentity` the Contributor role. Follow these steps to add a role assignment.

Create the custom image

To create a custom image, you can use the Azure VM Image Builder DevOps Task.

1. Add the `AzureImageBuilderTask@1` task to your YAML file. Replace the values for `<SUBSCRIPTION ID>`, `<RESOURCE GROUP>`, `<USER ASSIGNED IDENTITY NAME>` with your own. Make sure to verify that the `galleryImageId`, `managedIdentity` and `storageAccountName` values are accurate.

   YAML

   ```yaml
   - task: AzureImageBuilderTask@1
     displayName: 'Azure VM Image Builder Task'
     inputs:
       managedIdentity: '/subscriptions/<SUBSCRIPTION
   ID>/resourcegroups/<RESOURCE
   GROUP>/providers/Microsoft.ManagedIdentity/userAssignedIdentities/<USER
   ASSIGNED IDENTITY NAME>'
       imageSource: 'marketplace'
       packagePath: '$(System.DefaultWorkingDirectory)/pipeline-artifacts'
       inlineScript: |
         sudo mkdir /lib/buildArtifacts
         sudo cp  "/tmp/pipeline-artifacts.tar.gz" /lib/buildArtifacts/.
         cd /lib/buildArtifacts/.
         sudo tar -zxvf pipeline-artifacts.tar.gz
         sudo sh install.sh
       storageAccountName: 'vmssstorageaccount2'
       distributeType: 'sig'
       galleryImageId: '/subscriptions/<SUBSCRIPTION
   ID>/resourceGroups/<RESOURCE
   GROUP>/providers/Microsoft.Compute/galleries/myVMSSGallery/images/MyIma
   ge/versions/0.0.$(Build.BuildId)'
       replicationRegions: 'eastus2'
       ibSubscription: '<SUBSCRIPTION ID>'
       ibAzureResourceGroup: 'myVMSSResourceGroup'
       ibLocation: 'eastus2'
   ```

2. Run the pipeline to generate your first image. You may need to authorize resources during the pipeline run.

3. Go to the new image in the Azure portal and open **Overview**. Select **Create VMSS** to create a new virtual machine scale set from the new image. Set **Virtual machine scale set name** to `vmssScaleSet`. See Create a virtual machine scale set in the Azure portal to learn more about creating virtual machine scale sets in the Azure portal.

Deploy updates to the virtual machine scale set

Add an Azure CLI task to your pipeline to deploy updates to the scale set. Add the task at the end of the pipeline. Replace `<SUBSCRIPTION ID>` with your subscription ID.

yml

```yml
- task: AzureCLI@2
  inputs:
    azureSubscription: '`YOUR_SUBSCRIPTION_ID`' #Authorize and in the task editor
    ScriptType: 'pscore'
    scriptLocation: 'inlineScript'
    Inline: 'az vmss update --resource-group myVMSSResourceGroup --name vmssScaleSet --set virtualMachineProfile.storageProfile.imageReference.id=/subscriptions/<SUBSCRIPTION ID>/resourceGroups/myVMSSResourceGroup/providers/Microsoft.Compute/galleries/myVMSSGallery/images/MyImage/versions/0.0.$(Build.BuildId)'
```

Clean up resources

Go to the Azure portal and delete your resource group, `myVMSSResourceGroup`.

Next steps

Learn more about virtual machine scale sets

Deploy to Azure Stack Hub App Service using Azure Pipelines

Article • 04/05/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

This article walks you through setting up a CI/CD pipeline for deploying an application to app services in an Azure Stack Hub instance using Azure Pipelines.

In this article you can learn to create or validate:

- Azure Stack Hub service principal (SPN) credentials for the pipeline.
- A web app in your Azure Stack Hub instance.
- A service connection to your Azure Stack Hub instance.
- A repo with your app code to deploy to your app

Prerequisites

- Access to Azure Stack Hub instance with the App Service RP enabled.
- An Azure DevOps solution associated with your Azure Stack Hub tenant.

Create or validate your SPN

An SPN provides role-based credentials so that processes outside of Azure can connect to and interact with resources. You'll need an SPN with contributor access and the attributes specified in these instructions to use with your Azure DevOps pipeline.

As a user of Azure Stack Hub you don't have the permission to create the SPN. You'll need to request this principal from your cloud operator. The instructions are being provided here so you can create the SPN if you're a cloud operator, or you can validate the SPN if you're a developer using an SPN in your workflow provided by a cloud operator.

The cloud operator will need to create the SPN using Azure CLI.

The following code snippets are written for a Windows machine using the PowerShell prompt with Azure CLI for Azure Stack Hub. If you're using CLI on a Linux machine and bash, either remove the line extension or replace them with a `\`.

1. Prepare the values of the following parameters used to create the SPN:

Parameter	Example	Description

Parameter	Example	Description
endpoint-resource-manager	"https://management.orlando.azurestack.corp.microsoft.com"	The resource management endpoint.
suffix-storage-endpoint	"orlando.azurestack.corp.microsoft.com"	The endpoint suffix for storage accounts.
suffix-keyvault-dns	".vault.orlando.azurestack.corp.microsoft.com"	The Key Vault service dns suffix.
endpoint-active-directory-graph-resource-id	"https://graph.windows.net/"	The Active Directory resource ID.
endpoint-sql-management	https://notsupported	The sql server management endpoint. Set this to `https://notsupported`
profile	2019-03-01-hybrid	Profile to use for this cloud.

2. Open your command-line tool such as Windows PowerShell or Bash and sign in. Use the following command:

Azure CLI

```
az login
```

3. Use the `register` command for a new environment or the `update` command if you're using an existing environment. Use the following command.

Azure CLI

```
az cloud register `
    -n "AzureStackUser" `
    --endpoint-resource-manager "https://management.<local>.<FQDN>" `
    --suffix-storage-endpoint ".<local>.<FQDN>" `
    --suffix-keyvault-dns ".vault.<local>.<FQDN>" `
    --endpoint-active-directory-graph-resource-id
"https://graph.windows.net/" `
    --endpoint-sql-management https://notsupported  `
    --profile 2019-03-01-hybrid
```

4. Get your subscription ID and resource group that you want to use for the SPN.

5. Create the SPN with the following command with the subscription ID and resource group:

Azure CLI

```
az ad sp create-for-rbac --name "myApp" --role contributor `
    --scopes /subscriptions/{subscription-id}/resourceGroups/{resource-
group} `
    --sdk-auth
```

If you don't have cloud operator privileges, you can also sign in with the SPN provided to you by your cloud operator. You'll need the client ID, the secret, and your tenant ID. With these values, you can use the following Azure CLI commands to create the JSON object that contains the values you'll need to create your service connection.

Azure CLI

```
az login --service-principal -u "<client-id>" -p "<secret>" --tenant "
<tenant-ID>" --allow-no-subscriptions
az account show
```

6. Check the resulting JSON object. You'll use the JSON object to create your service connection. The JSON object should have the following attributes:

JSON

```
{
  "environmentName": "<Environment name>",
  "homeTenantId": "<Tenant ID for the SPN>",
  "id": "<Application ID for the SPN>",
  "isDefault": true,
  "managedByTenants": [],
  "name": "<Tenant name>",
  "state": "Enabled",
  "tenantId": "<Tenant ID for the SPN>",
  "user": {
    "name": "<User email address>",
    "type": "user"
  }
}
```

Create the web app target

1. Sign in to your Azure Stack Hub portal.

2. Select **Create a resource** > **Web + Mobile** > **Web App**.
3. Select your **Subscription**.
4. Create or select a **Resource Group**.
5. Type the **Name** of your app. The name of the app will appear in the URL for your app, for example, `yourappname.appservice.<region>.<FQDN>`
6. Select the **Runtime stack** for your app. The runtime must match the code you plan to use for your web app.
7. Select the **Operating System** (OS) that will host your runtime and app.
8. Select or type the **Region** for your Azure Stack Hub instance.
9. Select the plan based on your Azure Stack Hub instance, region, and app OS.
10. Select **Review + Create**.
11. Review your web app. Select **Create**.
12. Select **Go to resource**.
13. Make note of your app name. You'll add the name to the yml document that defines your pipeline in your repository.

Create a service connection

Create a service connection. You'll need the values from your SPN and the name of your Azure Stack Hub subscription.

1. Sign in to your Azure DevOps organization, and then navigate to your project.
2. Select **Project settings**, and then select **Service connections**.
3. Select **Service connections** > **New service connection**.
4. Select **Azure Resource Manager**, and then select **Next**.
5. Select **Service principal (manual)**.
6. Select **Azure Stack** from **Environment**.
7. Fill out the form, and then select **Verify and save**.
8. Give your service connection a name. (You will need the service connection name to create your yaml pipeline).

Create your repository and add pipeline

1. If you haven't added your web app code to the repository, add it now.
2. Open the repository. Select the repo and select **Browse**.
3. Select **Pipelines**
4. Select **New pipeline**.
5. Select **Azure Repos Git**.
6. Select your repository.
7. Select **Starter pipeline**.
8. Navigate back to the repo and open the `azure-pipelines.yml`.
9. Add the following yaml:

YAML

```yaml
# Starter pipeline
# Start with a minimal pipeline that you can customize to build and
deploy your code.
# Add steps that build, run tests, deploy, and more:
# https://aka.ms/yaml
trigger:
- main
variables:
  azureSubscription: '<your connection name>'
  VSTS_ARM_REST_IGNORE_SSL_ERRORS: true
steps:
- task: AzureWebApp@1
  displayName: Azure Web App Deploy
  inputs:
    azureSubscription: $(azureSubscription)
    appName: <your-app-name>
    package: '$(System.DefaultWorkingDirectory)'
```

> **Note**
>
> To ignore SSL errors, set a variable named `VSTS_ARM_REST_IGNORE_SSL_ERRORS` to the value `true` in the build or release pipeline, as in this example.

10. Update the `azureSubscription` value with the name of your service connection.
11. Update the `appName` with your app name. You're now ready to deploy.

Notes about using Azure tasks with Azure Stack Hub

The following Azure tasks are validated with Azure Stack Hub:

- Azure PowerShell
- Azure File Copy
- Azure Resource Group Deployment
- Azure App Service Deploy
- Azure App Service Manage
- Azure SQL Database Deployment

Next steps

- Deploy an Azure Web App
- Troubleshoot Azure Resource Manager service connections
- Azure Stack Hub User Documentation

Build and push Docker images to Azure Container Registry using Docker templates

Article • 01/30/2023

Azure DevOps Services

In this step-by-step tutorial, you'll learn how to set up a continuous integration pipeline to build a containerized application. New pull requests trigger the pipeline to build and publish Docker images to Azure Container Registry.

Prerequisites

- A GitHub account. Create a free GitHub account, if you don't already have one.
- An Azure DevOps organization and a project. Create a new organization and/or a new project, if you don't already have one.
- An Azure account. Sign up for a free Azure account, if you don't already have one.

Get the code

Fork or clone the *pipeline-javascript-docker* sample application:

```
https://github.com/MicrosoftDocs/pipelines-javascript-docker
```

Create a container registry

1. Sign in to Azure, and then select the **Azure Cloud Shell** button in the upper-right corner.

2. Run the following commands to create a resource group and an Azure Container Registry using the Azure CLI.

Azure CLI

```
# Create a resource group
az group create --name myapp-rg --location eastus

# Create a container registry
az acr create --resource-group myapp-rg --name mycontainerregistry --
sku Basic
```

> **Note**
>
> A Container Registry name must be lowercase.

You can also use the Azure portal web UI to create your Azure Container Registry. See the Create a container registry for details.

> **Important**
>
> You must enable the admin user account in order for you to deploy a Docker image from an Azure Container Registry. See **Container registry authentication** for more details.

Create the pipeline

1. Sign in to your Azure DevOps organization and navigate to your project.

2. Select **Pipelines**, and then select **New Pipeline** to create a new pipeline.

3. Select **GitHub YAML**, and then select **Authorize Azure Pipelines** to provide the appropriate permissions to access your repository.

4. You might be asked to sign in to GitHub. If so, enter your GitHub credentials, and then select your repository from the list of repositories.

5. From the **Configure** tab, select the **Docker - Build and push an image to Azure Container Registry** task.

6. Select your **Azure Subscription**, and then select **Continue**.

7. Select your **Container registry** from the dropdown menu, and then provide an **Image Name** to your container image.

8. Select **Validate and configure** when you are done.

As Azure Pipelines creates your pipeline, it will:

- Create a *Docker registry service connection* to enable your pipeline to push images to your container registry.
- Generate an *azure-pipelines.yml* file, which defines your pipeline.

9. Review your pipeline YAML, and then select **Save and run** when you are ready.

10. Add a **Commit message**, and then select **Save and run** to commit your changes and run your pipeline.

11. As your pipeline runs, select the build job to watch your pipeline in action.

How we build your pipeline

The pipeline that we just created in the previous section was generated from the *Docker container template* YAML. The build stage uses the Docker task `Docker@2` to build and push your Docker image to the container registry.

YAML

```yaml
- stage: Build
  displayName: Build and push stage
  jobs:
  - job: Build
    displayName: Build job
    pool:
      vmImage: $(vmImageName)
    steps:
    - task: Docker@2
      displayName: Build and push an image to container registry
      inputs:
        command: buildAndPush
        repository: $(imageRepository)
        dockerfile: $(dockerfilePath)
        containerRegistry: $(dockerRegistryServiceConnection)
```

```
        tags: |
          $(tag)
```

Clean up resources

If you are not going to continue to use this application, you can delete the resources you created in this tutorial to avoid incurring ongoing charges. Run the following to delete your resource group.

Azure CLI

```
az group delete --name myapp-rg
```

Related articles

- Deploy to Azure Web App for Containers (Classic)
- Docker Content Trust

Build and publish Docker images to Azure Container Registry

Article • 06/14/2023

Using Azure Pipelines, you can set up a pipeline workflow to build and publish your Docker images to Azure Container Registry. In this article, you will learn how to:

- ✓ Create an Azure Container Registry
- ✓ Set up a self-hosted agent on an Azure VM
- ✓ Set up the managed service identity
- ✓ Create a Docker Registry service connection
- ✓ Build and publish your image to Azure Container Registry

Prerequisites

- A GitHub account. sign up for free, if you don't have one already.
- An Azure DevOps organization.
- An Azure DevOps project.
  - The **Administrator** role for service connections in your Azure DevOps project.
- An Azure account with an active subscription. Sign up for free if you don't have one already.

Get the code

Fork or clone the sample app to follow along with this tutorial.

JavaScript

```
https://github.com/MicrosoftDocs/pipelines-javascript-docker
```

Create an Azure Container Registry

Portal

1. Navigate to Azure portal.
2. Select **Create a resource** from the left navigation panel, and then select **Containers** then **Container Registry**.
3. Select your **Subscription** and then select your **Resource group** or create a new one.
4. Enter a **Registry name** for your container registry. The registry name must be unique within Azure and must contain at least 5 characters.
5. Select your preferred **Location** and **SKU** and then select **Review + create**.
6. Review your settings and then select **Create** when you're done.

Set up a self-hosted agent VM

To use Managed Service Identity with Azure Pipelines to publish Docker images to Azure Container Registry, we must set up our own self-hosted agent on an Azure VM.

Create a VM

1. Navigate to Azure portal.
2. Select **Create a resource** from the left navigation panel, and then select **Virtual machine** -> **Create**.
3. Select your **Subscription** and then select the **Resource group** you used to create your container registry.
4. Give your virtual machine a name and choose an **Image**.
5. Enter your **Username** and **Password**, and then select **Review + create**.
6. Review your settings, and then select **Create** when you're done.
7. Select **Go to resource** when the deployment is complete.

Create an agent pool

1. From your Azure DevOps project, select the gear icon to navigate to your **Project settings**.

2. Select **Agent pools**, and then select **Add pool**.

3. Select **New**, and then select *Self-hosted* from the **Pool type** dropdown menu.

4. Give your pool a name, and then check **Grant access permission to all pipelines** checkbox.

5. Select **Create** when you're done.

6. Now select the pool you just created, and then select **New agent**.

7. We will be using the instructions in this window to set up your agent in the VM you created earlier. Select the **Copy** button to copy the download link to your clipboard.

Create a personal access token

1. From your Azure DevOps project, select **User Settings**, and then select **Personal Access Tokens**.
2. Select **New Token** to create a new Personal Access Token.
3. Enter a name for your PAT, and then choose an expiration date.
4. Select **Custom defined** in **Scopes**, and then select **Show all scopes**.
5. Select **Agent Pools** -> **Read & manage**, and **Deployment Groups** -> **Read & manage**.
6. Select **Create** when you're done, and save your PAT in a safe location.

Set up a self-hosted agent

1. In Azure portal, connect to your VM.
2. In an elevated PowerShell command prompt, run the following command to download the agent.

```PowerShell
Invoke-WebRequest -URI <YOUR_AGENT_DOWNLOAD_LINK> -UseBasicParsing  -
OutFile <FILE_PATH>
##Example: Invoke-WebRequest -URI
https://vstsagentpackage.azureedge.net/agent/2.213.2/vsts-agent-win-
x64-2.213.2.zip -OutFile C:\vsts-agent-win-x64-2.213.2.zip
```

3. Run the following command to extract and create your agent.

```PowerShell
mkdir agent ; cd agent
Add-Type -AssemblyName System.IO.Compression.FileSystem ;
[System.IO.Compression.ZipFile]::ExtractToDirectory("<FILE_PATH>",
"$PWD")
```

4. Run the following command to start configuring your agent.

```PowerShell
.\config.cmd
```

5. Enter your server URL when asked for input. Example: https://dev.azure.com/fabrikamFiber

6. Press **Enter** when prompted for the authentication type to choose **PAT** authentication.

7. Paste your personal access token that you created earlier and then press enter.

8. Enter the name of your agent pool, and then enter your agent name.

9. Leave the default value for *work folder*, and then enter *Y* if you want to run your agent as a service.

10. Now that the agent is ready to start listening for jobs, let's install Docker on our VM. Run the following command to download Docker.

PowerShell

```
Invoke-WebRequest -URI
https://desktop.docker.com/win/main/amd64/Docker%20Desktop%20Installer.
exe -OutFile <DOWNLOAD_PATH>
```

11. Navigate to your download path, and then run the following command to install and start Docker.

PowerShell

```
Start-Process 'Docker%20Desktop%20Installer.exe' -Wait install
```

12. Now navigate back to your agent folder, and run the cmd file to run the agent on your Azure VM.

PowerShell

```
.\run.cmd
```

13. Your agent should be listed now in your Agent pool -> **Agents** in the Azure DevOps portal.

Set up the managed identity

1. In Azure portal, navigate to the VM you created earlier.

2. Select **Identity** from the left navigation panel, and then enable the **System assigned** identity.

3. Select **Save** when you're done and then confirm your choice.

4. Select **Azure role assignments**, and then select **Add role assignment**.

5. Select **Resource group** from the **Scope** dropdown menu.

6. Select your **Subscription** and your **Resource group**, and then select the **AcrPush** role.

7. Repeat the steps 5 & 6 to add the **AcrPull** role.

Create a Docker registry service connection

Managed Service Identity

1. From your Azure DevOps project, select the gear icon ⚙ to navigate to your **Project settings**.
2. Select **Service connections** from the left pane.
3. Select **New service connection**, and then select **Docker Registry** then **Next**.
4. Select **Azure Container Registry**, and then select *Managed Service Identity* as your **Authentication Type**.
5. Enter your Subscription ID **Subscription name**, and your Azure container registry login server. Paste your VM's system assigned Object ID that you created in the previous step in the **Tenant ID** text field.
6. Enter a name for your service connection, and then check the **Grant access permission to all pipelines** checkbox. To select this option, you'll need the service connection **Administrator** role.
7. Select **Save** when you are done.

Build and publish to Azure Container Registry

1. From your Azure DevOps project, select **Pipelines** and then select **Create Pipeline**.

2. Select the service hosting your source code (Azure Repos, GitHub etc.).

3. Select your repository, and then select **Starter pipeline**.

4. Delete the default yaml pipeline and use the following snippet:

```yml
trigger:
- main

variables:
  dockerRegistryServiceConnection: '<SERVICE_CONNECTION_NAME>'
  imageRepository: '<IMAGE_NAME>'
  dockerfilePath: '$(Build.SourcesDirectory)/app/Dockerfile'
  tag: '$(Build.BuildId)'

stages:
- stage: Build
  displayName: Build and publish stage
  jobs:
  - job: Build
    displayName: Build job
    pool:
      name: '<YOUR_AGENT_POOL_NAME>'
    steps:
    - task: DockerInstaller@0
      inputs:
        dockerVersion: '17.09.0-ce'

    - task: Docker@2
      displayName: Build and publish image to Azure Container Registry
      inputs:
        command: buildAndPush
        containerRegistry: $(dockerRegistryServiceConnection)
        repository: $(imageRepository)
        dockerfile: $(dockerfilePath)
        tags: |
          $(tag)
```

5. Once the pipeline run is complete, you can verify your image in Azure. Navigate to your Azure Container Registry in Azure portal, and then select **Repositories**.

Clean up resources

If you don't plan to continue using this application, delete the resource group to avoid incurring ongoing charges.

Azure CLI

```
az group delete --name myapp-rg
```

Related articles

- Build and push images to Azure Container Registry with Docker templates
- Build and deploy to Azure Kubernetes Service

Build and deploy to Azure Kubernetes Service with Azure Pipelines

Article • 10/11/2023

Azure DevOps Services

Use Azure Pipelines to automatically deploy to Azure Kubernetes Service (AKS). Azure Pipelines lets you build, test, and deploy with continuous integration (CI) and continuous delivery (CD) using Azure DevOps.

In this article, you'll learn how to create a pipeline that continuously builds and deploys your app. Every time you change your code in a repository that contains a Dockerfile, the images are pushed to your Azure Container Registry, and the manifests are then deployed to your AKS cluster.

Prerequisites

- An Azure account with an active subscription. Create an account for free.
- An Azure Resource Manager service connection. Create an Azure Resource Manager service connection.
- A GitHub account. Create a free GitHub account if you don't have one already.

Get the code

Fork the following repository containing a sample application and a Dockerfile:

```
https://github.com/MicrosoftDocs/pipelines-javascript-docker
```

Create the Azure resources

Sign in to the Azure portal, and then select the Cloud Shell button in the upper-right corner. Use Azure CLI or PowerShell to create an AKS cluster.

Create a container registry

Azure CLI

Azure CLI

```
# Create a resource group
az group create --name myapp-rg --location eastus

# Create a container registry
az acr create --resource-group myapp-rg --name mycontainerregistry --sku
Basic

# Create a Kubernetes cluster
az aks create \
    --resource-group myapp-rg \
    --name myapp \
    --node-count 1 \
    --enable-addons monitoring \
    --generate-ssh-keys
```

Sign in to Azure Pipelines

Sign in to Azure Pipelines. After you sign in, your browser goes to `https://dev.azure.com/my-organization-name` and displays your Azure DevOps dashboard.

Within your selected organization, create a *project*. If you don't have any projects in your organization, you see a **Create a project to get started** screen. Otherwise, select the **Create Project** button in the upper-right corner of the dashboard.

Create the pipeline

Connect and select your repository

1. Sign in to your Azure DevOps organization and go to your project.
2. Go to **Pipelines**, and then select **New pipeline**.
3. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.
4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.
5. When you see the list of repositories, select your repository.

6. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.
7. Select **Deploy to Azure Kubernetes Service**.
8. If you're prompted, select the subscription in which you created your registry and cluster.
9. Select the `myapp` cluster.
10. For **Namespace**, select **Existing**, and then select **default**.
11. Select the name of your container registry.
12. You can leave the image name set to the default.
13. Set the service port to 8080.
14. Set the **Enable Review App for Pull Requests** checkbox for review app related configuration to be included in the pipeline YAML autogenerated in subsequent steps.
15. Select **Validate and configure**.

    As Azure Pipelines creates your pipeline, the process will:

    - Create a *Docker registry service connection* to enable your pipeline to push images into your container registry.
    - Create an *environment* and a Kubernetes resource within the environment. For an RBAC-enabled cluster, the created Kubernetes resource implicitly creates ServiceAccount and RoleBinding objects in the cluster so that the created ServiceAccount can't perform operations outside the chosen namespace.
    - Generate an *azure-pipelines.yml* file, which defines your pipeline.
    - Generate Kubernetes manifest files. These files are generated by hydrating the deployment.yml and service.yml templates based on selections you made. When you're ready, select **Save and run**.
16. Select **Save and run**.
17. You can change the **Commit message** to something like *Add pipeline to our repository*. When you're ready, select **Save and run** to commit the new pipeline into your repo, and then begin the first run of your new pipeline!

See your app deploy

As your pipeline runs, watch as your build stage, and then your deployment stage, go from blue (running) to green (completed). You can select the stages and jobs to watch your pipeline in action.

> **Note**
>
> If you're using a Microsoft-hosted agent, you must add the IP range of the Microsoft-hosted agent to your firewall. Get the weekly list of IP ranges from the **weekly JSON file**, which is published every Wednesday. The new IP ranges become effective the following Monday. For more information, see **Microsoft-hosted agents**. To find the IP ranges that are required for your Azure DevOps organization, learn how to **identify the possible IP ranges for Microsoft-hosted agents**.

After the pipeline run is finished, explore what happened and then go see your app deployed. From the pipeline summary:

1. Select the **Environments** tab.
2. Select **View environment**.
3. Select the instance of your app for the namespace you deployed to. If you used the defaults, then it is the **myapp** app in the **default** namespace.
4. Select the **Services** tab.
5. Select and copy the external IP address to your clipboard.
6. Open a new browser tab or window and enter <IP address>:8080.

If you're building our sample app, then *Hello world* appears in your browser.

How the pipeline builds

When you finished selecting options and then proceeded to validate and configure the pipeline Azure Pipelines created a pipeline for you, using the *Deploy to Azure Kubernetes Service* template.

The build stage uses the Docker task to build and push the image to the Azure Container Registry.

YAML

```yaml
- stage: Build
  displayName: Build stage
  jobs:
  - job: Build
    displayName: Build job
    pool:
      vmImage: $(vmImageName)
    steps:
    - task: Docker@2
      displayName: Build and push an image to container registry
      inputs:
        command: buildAndPush
        repository: $(imageRepository)
        dockerfile: $(dockerfilePath)
        containerRegistry: $(dockerRegistryServiceConnection)
        tags: |
          $(tag)

    - task: PublishPipelineArtifact@1
      inputs:
        artifactName: 'manifests'
        path: 'manifests'
```

The deployment job uses the *Kubernetes manifest task* to create the `imagePullSecret` required by Kubernetes cluster nodes to pull from the Azure Container Registry resource. Manifest files are then used by the Kubernetes manifest task to deploy to the Kubernetes cluster. The manifest files, `service.yml` and `deployment.yml`, were generated when you used the **Deploy to Azure Kubernetes Service** template.

YAML

```yaml
- stage: Deploy
  displayName: Deploy stage
  dependsOn: Build
  jobs:
  - deployment: Deploy
    displayName: Deploy job
    pool:
      vmImage: $(vmImageName)
    environment: 'myenv.aksnamespace' #customize with your environment
    strategy:
      runOnce:
        deploy:
          steps:
          - task: DownloadPipelineArtifact@2
            inputs:
              artifactName: 'manifests'
              downloadPath: '$(System.ArtifactsDirectory)/manifests'
```

```yaml
            - task: KubernetesManifest@0
              displayName: Create imagePullSecret
              inputs:
                action: createSecret
                secretName: $(imagePullSecret)
                namespace: $(k8sNamespace)
                dockerRegistryEndpoint: $(dockerRegistryServiceConnection)

            - task: KubernetesManifest@0
              displayName: Deploy to Kubernetes cluster
              inputs:
                action: deploy
                namespace: $(k8sNamespace)
                manifests: |
                  $(System.ArtifactsDirectory)/manifests/deployment.yml
                  $(System.ArtifactsDirectory)/manifests/service.yml
                imagePullSecrets: |
                  $(imagePullSecret)
                containers: |
                  $(containerRegistry)/$(imageRepository):$(tag)
```

Clean up resources

Whenever you're done with the resources you created, you can use the following command to delete them:

Azure CLI

```cli
az group delete --name myapp-rg
```

Enter `y` when you're prompted.

Implement Azure Policy with Azure DevOps release pipelines

Article • 03/02/2023

Azure DevOps Services

Learn how to enforce compliance policies on your Azure resources before and after deployment with Azure Pipelines. Azure Pipelines lets you build, test, and deploy with continuous integration (CI) and continuous delivery (CD) using Azure DevOps. One scenario for adding Azure Policy to a pipeline is when you want to ensure that resources are deployed only to authorized regions and are configured to send diagnostics logs to Azure Log Analytics.

You can use either the classic pipeline or YAML pipeline processes to implement Azure Policy in your CI/CD pipelines.

For more information, see What is Azure Pipelines? and Create your first pipeline.

Prepare

1. Create an Azure Policy in the Azure portal. There are several predefined sample policies that can be applied to a management group, subscription, and resource group.

2. In Azure DevOps, create a release pipeline that contains at least one stage, or open an existing release pipeline.

3. Add a pre- or post-deployment condition that includes the **Check Azure Policy compliance** task as a gate. More details.

If you're using a YAML pipeline definition, then use the AzurePolicyCheckGate@0 Azure Pipelines task.

Validate for any violation(s) during a release

> **Note**
>
> Use the **AzurePolicyCheckGate** task to check for policy compliance in YAML. This task can only be used as a gate and not in a build or a release pipeline.

1. Navigate to your team project in Azure DevOps.
2. In the **Pipelines** section, open the **Releases** page and create a new release.
3. Choose the **In progress** link in the release view to open the live logs page.
4. When the release is in progress and attempts to perform an action disallowed by the defined policy, the deployment is marked as **Failed**. The error message contains a link to view the policy violations.

5. An error message is written to the logs and displayed in the stage status panel in the releases page of Azure Pipelines.

6. When the policy compliance gate passes the release, a **Succeeded** status is displayed.

7. Choose the successful deployment to view the detailed logs.

Next steps

To learn more about the structures of policy definitions, look at this article:

Azure Policy definition structure

Quickstart: Build a container image to deploy apps using Azure Pipelines

Article • 11/28/2022

Azure DevOps Services

This quickstart shows how to build a container image for app deployment using Azure Pipelines. To build this image, all you need is a Dockerfile in your repository. You can build Linux or Windows containers, based on the agent that you use in your pipeline.

Prerequisites

- An Azure account with an active subscription. Create an account for free.
- A GitHub account. If you don't have one, sign up for free.
- A GitHub repository with a Dockerfile. If you don't have a repository to use, fork the following repository, which contains a sample application and a Dockerfile:

```
https://github.com/MicrosoftDocs/pipelines-javascript-docker
```

Build a Linux or Windows image

1. Sign in to your Azure DevOps organization, and go to your project.
2. Go to **Pipelines**, and select **New Pipeline**.
3. Select **GitHub** as the location for your source code.
4. Select your repository, and then select **Starter pipeline**.
    - If you're redirected to GitHub to sign in, enter your GitHub credentials.
    - If you're redirected to GitHub to install the Azure Pipelines app, select **Approve and install**.
5. Replace the contents of **azure-pipelines.yml** with the following code. Based on whether you're deploying a Linux or Windows app, make sure to respectively set `vmImage` to either `ubuntu-latest` or `windows-latest`.

```yaml
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

variables:
  imageName: 'pipelines-javascript-docker'

steps:
- task: Docker@2
  displayName: Build an image
  inputs:
    repository: $(imageName)
    command: build
    Dockerfile: app/Dockerfile
```

6. When you're done, select **Save and run**.

   When you add the **azure-pipelines.yml** file to your repository, you're prompted to add a commit message.

For more information, see the Docker task used by this sample application. You can also directly invoke Docker commands using a command line task.

Clean up resources

If you don't plan to continue using this application, delete your pipeline and code repository.

FAQ

What agents can I use to build container images?

- You can build Linux container images using Microsoft-hosted Ubuntu agents or Linux platform-based self-hosted agents.
- You can build Windows container images using Microsoft-hosted Windows agents or Windows platform based self-hosted agents. All Microsoft-hosted Windows platform-based agents are shipped with the Moby engine and client needed for Docker builds.

- You currently can't use Microsoft-hosted macOS agents to build container images because the Moby engine needed for building the images isn't pre-installed on these agents.

For more information, see the Windows and Linux agent options available with Microsoft-hosted agents.

What pre-cached images are available on hosted agents?

To avoid spending long intervals pulling these images for every job from the container registry, some commonly used images are pre-cached on Microsoft-hosted agents. For the list of available pre-cached images, see the release notes in the **azure-pipelines-image-generation** repository.

How do I set the BuildKit variable for my Docker builds?

BuildKit introduces build improvements around performance, storage management, feature functionality, and security. BuildKit currently isn't supported on Windows hosts.

To enable Docker builds using BuildKit, set the **DOCKER_BUILDKIT** variable.

YAML

```yaml
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'


variables:
  imageName: 'pipelines-javascript-docker'
  DOCKER_BUILDKIT: 1

steps:
- task: Docker@2
  displayName: Build an image
  inputs:
    repository: $(imageName)
    command: build
    Dockerfile: app/Dockerfile
```

How can I use a self-hosted agent?

1. Before you create your container image, make sure to install Docker on your self-hosted machine.
2. In your pipeline, *prior* to the Docker task that builds your image, add the Docker installer task.

How can I create a script-based Docker build instead of using the Docker task?

You can use the `build` command or any other Docker command.

```
docker build -f Dockerfile -t foobar.azurecr.io/hello:world .
```

This command creates an image equivalent to one built with the Docker task. Internally, the Docker task calls the Docker binary on a script and stitches together a few more commands to provide a few more benefits. Learn more about Docker task.

Can I reuse layer caching during builds on Azure Pipelines?

If you're using Microsoft-hosted agents, every job is dispatched to a newly provisioned virtual machine, based on the image generated from azure-pipelines-image-generation repository templates. These virtual machines are cleaned up after the job completes. This ephemeral lifespan prevents reusing these virtual machines for subsequent jobs and the reuse of cached Docker layers. As a workaround, you can set up a multi-stage build that produces two images and pushes them to an image registry at an early stage. You can then tell Docker to use these images as a cache source with the `--cache-from` argument.

If you're using self-hosted agents, you can cache Docker layers without any workarounds because the ephemeral lifespan problem doesn't apply to these agents.

How do I build Linux container images for architectures other than x64?

When you use Microsoft-hosted Linux agents, you create Linux container images for the x64 architecture. To create images for other architectures, such as x86 or ARM processor, you can use a machine emulator such as QEMU.

The following steps show how to create an ARM processor container image by using QEMU:

1. Author your Dockerfile with a base image that matches the target architecture:

```
FROM arm64v8/alpine:latest
```

2. Run the following script in your job before you build the image:

```
# register QEMU binary - this can be done by running the following
image
docker run --rm --privileged multiarch/qemu-user-static --reset -p yes
# build your image
```

For more information, see qemu-user-static on GitHub.

How do I run tests and publish test results for containerized applications?

For different options on testing containerized applications and publishing the resulting test results, see Publish Test Results task.

Next steps

After you build your container image, push the image to Azure Container Registry, Docker Hub, or Google Container registry. To learn how to push an image to a container registry, continue to either of the following articles:

Push an image to Azure Container Registry

Push an image to Docker Hub or Google Container Registry

Push an image

Article • 10/31/2022

Azure DevOps Services

Use Azure Pipelines to push your image to a container registry such as Azure Container Registry, Docker Hub, or Google Container Registry. Azure Container Registry is a managed registry service based on the open-source Docker Registry 2.0.

For a tutorial on building and pushing images to a container registry, see Build and push Docker images to Azure Container Registry.

To learn how to build a container image to deploy with Azure Pipelines, see Build container images to deploy apps.

About the Docker task

You'll use the Docker@2 task to build or push Docker images, login or logout, start or stop containers, or run a Docker command.

The task uses a Docker registry service connection to log in and push to a container registry. The process for creating a Docker registry service connection differs depending on your registry.

The Docker registry service connection stores credentials to the container registry before pushing the image. You can also directly reference service connections in Docker without an additional script task.

Create a Docker service connection

You'll need to follow a different process to create a service connection for Azure Container Registry, Docker Hub, and Google Container Registry.

Azure Container Registry

With the Azure Container Registry option, the subscription (associated with the Azure Active Directory identity of the user signed into Azure DevOps) and container registry within the subscription are used to create the service connection.

When you create a new pipeline for a repository that contains a Dockerfile, Azure Pipelines will detect Dockerfile in the repository. To start this process, create a new

pipeline and select the repository with your Dockerfile.

1. From the **Configure** tab, select the **Docker - Build and push an image to Azure Container Registry** task.

2. Select your **Azure Subscription**, and then select **Continue**.

3. Select your **Container registry** from the dropdown menu, and then provide an **Image Name** to your container image.

4. Select **Validate and configure** when you are done.

As Azure Pipelines creates your pipeline, it will:

- Create a *Docker registry service connection* to enable your pipeline to push images to your container registry.
- Generate an *azure-pipelines.yml* file, which defines your pipeline.

For a more detailed overview, see Build and Push to Azure Container Registry document.

Docker Content Trust

Article • 08/03/2022

Azure DevOps Services

Docker Content Trust (DCT) lets you use digital signatures for data sent to and received from remote Docker registries. These signatures allow client-side or runtime verification of the integrity and publisher of specific image tags.

> **Note**
>
> A prerequisite for signing an image is a Docker Registry with a Notary server attached (examples include **Docker Hub** or **Azure Container Registry**)

Signing images in Azure Pipelines

Prerequisites on development machine

1. Use Docker trust's built-in generator or manually generate delegation key pair. If the built-in generator is used, the delegation private key is imported into the local Docker trust store. Else, the private key will need to be manually imported into the local Docker trust store. See Manually Generating Keys for details.
2. Using the delegation key generated from the step above, upload the first key to a delegation and initiate the repository

> **Tip**
>
> To view the list of local Delegation keys, use the Notary CLI to run the following command: `$ notary key list`.

Set up pipeline for signing images

1. Grab the delegation private key, which is in the local Docker trust store of your development machine used earlier, and add the same as a secure file in Pipelines.

2. Authorize this secure file for use in all pipelines.

3. The service principal associated with `containerRegistryServiceConnection` must have the AcrImageSigner role in the target container registry.

4. Create a pipeline based on the following YAML snippet:

YAML

```yaml
pool:
  vmImage: 'Ubuntu 16.04'

variables:
  system.debug: true
  containerRegistryServiceConnection: serviceConnectionName
  imageRepository: foobar/content-trust
  tag: test

steps:
- task: Docker@2
  inputs:
    command: login
    containerRegistry: $(containerRegistryServiceConnection)

- task: DownloadSecureFile@1
  name: privateKey
  inputs:
    secureFile:
cc8f3c6f998bee63fefaaabc5a2202eab06867b83f491813326481f56a95466f.key
- script: |
    mkdir -p $(DOCKER_CONFIG)/trust/private
    cp $(privateKey.secureFilePath) $(DOCKER_CONFIG)/trust/private

- task: Docker@2
  inputs:
    command: build
    Dockerfile: '**/Dockerfile'
    containerRegistry: $(containerRegistryServiceConnection)
    repository: $(imageRepository)
    tags: |
      $(tag)
    arguments: '--disable-content-trust=false'

- task: Docker@2
  inputs:
    command: push
    containerRegistry: $(containerRegistryServiceConnection)
    repository: $(imageRepository)
    tags: |
      $(tag)
    arguments: '--disable-content-trust=false'
  env:
    DOCKER_CONTENT_TRUST_REPOSITORY_PASSPHRASE:
$(DOCKER_CONTENT_TRUST_REPOSITORY_PASSPHRASE)
```

In the previous example, the `DOCKER_CONFIG` variable is set by the `login` command in the Docker task. We recommend that you set up `DOCKER_CONTENT_TRUST_REPOSITORY_PASSPHRASE` as a secret variable for your pipeline. The alternative approach of using a pipeline variable in YAML exposes the passphrase in plain text. `DOCKER_CONTENT_TRUST_REPOSITORY_PASSPHRASE` in this example refers to the private key's passphrase (not the repository passphrase). We only need the private key's passphrase in this example because the repository has been initiated already (prerequisites).

Deploy to Kubernetes

Article • 12/01/2023

Azure DevOps Services | Azure DevOps Server 2022

You can use Azure Pipelines to deploy to Azure Kubernetes Service and Kubernetes clusters offered by other cloud providers. Azure Pipelines has two tasks for working with Kubernetes:

- KubernetesManifest task: bake and deploy manifests to Kubernetes clusters with Helm, Kompose, or Kustomize
- Kubectl task: deploy, configure, and update a Kubernetes cluster in Azure Container Service by running kubectl commands

If you're using Azure Kubernetes Service with either task, the Azure Resource Manager service connection type is the best way to connect to a private cluster, or a cluster that has local accounts disabled.

For added deployment traceability, use a Kubernetes resource in environments with a Kubernetes task.

To get started with Azure Pipelines and Azure Kubernetes service, see Build and deploy to Azure Kubernetes Service with Azure Pipelines. To get started with Azure Pipelines, Kubernetes, and the canary deployment strategy specifically, see Use a canary deployment strategy for Kubernetes deployments with Azure Pipelines.

KubernetesManifest task

The KubernetesManifest task checks for object stability before marking a task as success/failure. The task can also perform artifact substitution, add pipeline traceability-related annotations, simplify creation and referencing of imagePullSecrets, bake manifests, and aid in deployment strategy roll outs.

> **Note**
>
> While YAML-based pipeline support triggers on a single Git repository, if you need a trigger for a manifest file stored in another Git repository or if triggers are needed for Azure Container Registry or Docker Hub, you should use a classic pipeline instead of a YAML-based pipeline.

You can use the bake action in the Kubernetes manifest task to bake templates into Kubernetes manifest files. The action lets you use tools such as Helm, Kustomize, and Kompose. The bake action of the Kubernetes manifest task provides visibility into the transformation between input templates and the end manifest files that are used in deployments. You can consume baked manifest files downstream (in tasks) as inputs for the deploy action of the Kubernetes manifest task.

You can target Kubernetes resources that are part of environments with deployment jobs. Using environments and resources deployment gives you access to better pipeline traceability so that you can diagnose deployment issues. You can also deploy to Kubernetes clusters with regular jobs without the same health features.

The following YAML code is an example of baking manifest files from Helm charts

YAML

```yaml
steps:
- task: KubernetesManifest@0
  name: bake
  displayName: Bake K8s manifests from Helm chart
  inputs:
    action: bake
    helmChart: charts/sample
    overrides: 'image.repository:nginx'

- task: KubernetesManifest@0
  displayName: Deploy K8s manifests
  inputs:
    kubernetesServiceConnection: someK8sSC
    namespace: default
    manifests: $(bake.manifestsBundle)
    containers: |
      nginx: 1.7.9
```

Kubectl task

As an alternative to the KubernetesManifest KubernetesManifest task, you can use the Kubectl task to deploy, configure, and update a Kubernetes cluster in Azure Container Service by running kubectl commands.

The following example shows how a service connection is used to refer to the Kubernetes cluster.

YAML

```yaml
- task: Kubernetes@1
  displayName: kubectl apply
  inputs:
    connectionType: Kubernetes Service Connection
    kubernetesServiceEndpoint: Contoso
```

Script task

You can also use `kubectl` with a script task.

The following example uses a script to run `kubectl`.

YAML

```yaml
- script: |
    kubectl apply -f manifest.yml
```

Kubernetes deployment strategies

The Kubernetes manifest task currently supports canary deployment strategy. Use canary deployment strategy to partially deploy new changes so that the new changes coexist with the current deployments before a full rollout.

For more information on canary deployments with pipelines, see Use a canary deployment strategy for Kubernetes deployments with Azure Pipelines.

Multicloud Kubernetes deployments

Kubernetes runs the same way on all cloud providers. Azure Pipelines can be used for deploying to Azure Kubernetes Service (AKS), Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), or clusters from any other cloud providers.

To set up multicloud deployment, create an environment and then add your Kubernetes resources associated with namespaces of Kubernetes clusters.

- Azure Kubernetes Service
- Generic provider using existing service account

> The generic provider approach based on existing service account works with clusters from any cloud provider, including Azure. The benefit of using the Azure Kubernetes Service option instead is that it creates new ServiceAccount and RoleBinding

> objects (instead of reusing an existing ServiceAccount) so that the newly created RoleBinding object can limit the operations of the ServiceAccount to the chosen namespace only.

When you use the generic provider approach, make sure that a RoleBinding exists, which grants permissions in the `edit` `ClusterRole` to the desired service account. You need to grant permissions to the right services account so that the service account can be used by Azure Pipelines to create objects in the chosen namespace.

Parallel deployments to multiple clouds

The following example shows how to do parallel deployments to clusters in multiple clouds. In this example, there are deployments to the AKS, GKE, EKS, and OpenShift clusters. These four namespaces are associated with Kubernetes resources under the `contoso` environment.

YAML

```yaml
trigger:
- main

jobs:
- deployment:
  displayName: Deploy to AKS
  pool:
    vmImage: ubuntu-latest
  environment: contoso.aksnamespace
  strategy:
    runOnce:
      deploy:
        steps:
        - checkout: self
        - task: KubernetesManifest@0
          displayName: Deploy to Kubernetes cluster
          inputs:
            action: deploy
            kubernetesServiceConnection: serviceConnection #replace with
your service connection
            namespace: aksnamespace
            manifests: manifests/*
- deployment:
  displayName: Deploy to GKE
  pool:
    vmImage: ubuntu-latest
  environment: contoso.gkenamespace
  strategy:
    runOnce:
      deploy:
```

```
        steps:
        - checkout: self
        - task: KubernetesManifest@0
          displayName: Deploy to Kubernetes cluster
          inputs:
            action: deploy
            kubernetesServiceConnection: serviceConnection #replace with
your service connection
            namespace: gkenamespace
            manifests: manifests/*
- deployment:
  displayName: Deploy to EKS
  pool:
    vmImage: ubuntu-latest
  environment: contoso.eksnamespace
  strategy:
    runOnce:
      deploy:
        steps:
        - checkout: self
        - task: KubernetesManifest@0
          displayName: Deploy to Kubernetes cluster
          inputs:
            action: deploy
            kubernetesServiceConnection: serviceConnection #replace with
your service connection
            namespace: eksnamespace
            manifests: manifests/*
- deployment:
  displayName: Deploy to OpenShift
  pool:
    vmImage: ubuntu-latest
  environment: contoso.openshiftnamespace
  strategy:
    runOnce:
      deploy:
        steps:
        - checkout: self
        - task: KubernetesManifest@0
          displayName: Deploy to Kubernetes cluster
          inputs:
            action: deploy
            kubernetesServiceConnection: serviceConnection #replace with
your service connection
            namespace: openshiftnamespace
            manifests: manifests/*
- deployment:
  displayName: Deploy to DigitalOcean
  pool:
    vmImage: ubuntu-latest
  environment: contoso.digitaloceannamespace
  strategy:
    runOnce:
      deploy:
        steps:
```

```
        - checkout: self
        - task: KubernetesManifest@0
          displayName: Deploy to Kubernetes cluster
          inputs:
            action: deploy
            kubernetesServiceConnection: serviceConnection #replace with
your service connection
            namespace: digitaloceannamespace
            manifests: manifests/*
```

Tutorial: Use a canary deployment strategy for Kubernetes deployments

Article • 12/01/2023

Azure DevOps Services | Azure DevOps Server 2022

A *canary* deployment strategy means deploying new versions of an application next to stable, production versions. You can then see how the canary version compares to the baseline, before promoting or rejecting the deployment.

This step-by-step guide covers how to use the Kubernetes manifest task's canary strategy. Specifically, you'll learn how to set up canary deployments for Kubernetes, and the associated workflow to evaluate code. You then use that code to compare baseline and canary app deployments, so you can decide whether to promote or reject the canary deployment.

If you're using Azure Kubernetes Service, the Azure Resource Manager service connection type is the best way to connect to a private cluster, or a cluster that has local accounts disabled.

Prerequisites

- An Azure account with an active subscription. Create an account for free .
- A GitHub account. Create a free GitHub account if you don't have one already.
- An Azure Container Registrywith push privileges. Create an Azure Container Registry if you don't have one already.
- A Kubernetes cluster. Deploy an Azure Kubernetes Service (AKS) cluster.

Sample code

Fork the following repository on GitHub.

```
https://github.com/MicrosoftDocs/azure-pipelines-canary-k8s
```

Here's a brief overview of the files in the repository that are used during this guide:

- *./app*:

    - *app.py* - A simple, Flask-based web server that is instrumented by using the Prometheus instrumentation library for Python applications. A custom counter is set up for the number of good and bad responses given out, based on the value of the `success_rate` variable.
    - *Dockerfile* - Used for building the image with each change made to *app.py*. With each change, the build pipeline is triggered and the image gets built and pushed to the container registry.
- *./manifests*:
    - *deployment.yml* - Contains the specification of the `sampleapp` deployment workload corresponding to the image published earlier. You use this manifest file not just for the stable version of deployment object, but also for deriving the baseline and canary variants of the workloads.
    - *service.yml* - Creates the `sampleapp` service. This service routes requests to the pods spun up by the deployments (stable, baseline, and canary) mentioned previously.
- *./misc*
    - *service-monitor.yml* - Used to set up a ServiceMonitor object. This object sets up Prometheus metric scraping.
    - *fortio-deploy.yml* - Used to set up a fortio deployment. This deployment is later used as a load-testing tool, to send a stream of requests to the `sampleapp` service deployed earlier. The stream of requests sent to `sampleapp` are routed to pods under all three deployments (stable, baseline, and canary).

> **Note**
>
> In this guide, you use **Prometheus** for code instrumentation and monitoring. Any equivalent solution, like **Azure Application Insights**, can be used as an alternative.

Install prometheus-operator

To install Prometheus on your cluster, use the following command from your development machine. You must have kubectl and Helm installed, and you must set the context to the cluster you want to deploy against. Grafana, which you use later to visualize the baseline and canary metrics on dashboards, is installed as part of this Helm chart.

You'll first add the Prometheus Community Kubernetes Helm Charts repository to your Helm installation. Then you'll install the kube-prometheus stack, a collection of Kubernetes manifests, Grafana dashboards, and Prometheus rules.

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update # update local cache
helm install --name sampleapp  prometheus-community/kube-prometheus-stack
```

Create service connections

1. Go to **Project settings** > **Pipelines** > **Service connections** in the Azure DevOps menu.
2. Create a Docker registry service connection associated with your container registry. Name it **azure-pipelines-canary-k8s**.
3. Create a Kubernetes service connection for the Kubernetes cluster and namespace you want to deploy to. Name it **azure-pipelines-canary-k8s**.

> **Note**
>
> If you're using Azure Kubernetes Service, the **Azure Resource Manager service connection type** is the best way to connect to a private cluster, or a cluster that has local accounts disabled.

Set up continuous integration

1. Go to **Pipelines** > **Create Pipeline**, and select your repository.

2. On the **Configure** tab, choose **Starter pipeline**.

3. On the **Review** tab, replace the pipeline YAML with this code.

YAML

```yaml
trigger:
- main

pool:
  vmImage: ubuntu-latest

variables:
  imageName: azure-pipelines-canary-k8s

steps:
- task: Docker@2
  displayName: Build and push image
  inputs:
```

```
    containerRegistry: azure-pipelines-canary-k8s #replace with name of
your Docker registry service connection
    repository: $(imageName)
    command: buildAndPush
    Dockerfile: app/Dockerfile
    tags: |
      $(Build.BuildId)
```

If the Docker registry service connection that you created is associated with `example.azurecr.io`, then the image is to `example.azurecr.io/azure-pipelines-canary-k8s:$(Build.BuildId)`, based on the preceding configuration.

Edit manifest file

In *manifests/deployment.yml*, replace `<example>` with your container registry's URL. For example, after replacement, the image field should look something like `contosodemo.azurecr.io/azure-pipelines-canary-k8s`.

Set up continuous deployment

The following sections provide steps for setting up continuous deployment, including how to deploy the canary stage, and how to promote or reject the canary through manual intervention.

Deploy canary stage

You can deploy with YAML or Classic.

YAML

1. Go to **Pipelines** > **Environments** > **Create environment**.

2. Create a new environment.

   - **Name**: *akscanary*
   - **Resource**: Choose Kubernetes.

3. Select **Next**, and configure your Kubernetes resource as follows:

   - **Provider**: Azure Kubernetes Service
   - **Azure subscription**: Choose the subscription that holds your Kubernetes cluster.

- **Cluster**: Choose your cluster.
- **Namespace**: Create a new namespace, with the name *canarydemo*.

4. Select **Validate and Create**.

5. Go to **Pipelines**. Select the pipeline you created, and select **Edit**.

6. Change the step you created previously to now use a stage. Add two more steps to copy the manifests and *misc* directories as artifacts for use by consecutive stages. You might also want to move a couple of values to variables, for easier usage later in your pipeline. Your complete YAML should now look like this.

YAML

```yaml
trigger:
- main

pool:
  vmImage: ubuntu-latest

variables:
  imageName: azure-pipelines-canary-k8s
  dockerRegistryServiceConnection:
dockerRegistryServiceConnectionName #replace with name of your
Docker registry service connection
  imageRepository: 'azure-pipelines-canary-k8s'
  containerRegistry: example.azurecr.io #replace with the name of
your container registry, Should be in the format example.azurecr.io
  tag: '$(Build.BuildId)'

stages:
- stage: Build
  displayName: Build stage
  jobs:
  - job: Build
    displayName: Build
    pool:
      vmImage: ubuntu-latest
    steps:
    - task: Docker@2
      displayName: Build and push image
      inputs:
        containerRegistry: $(dockerRegistryServiceConnection)
        repository: $(imageName)
        command: buildAndPush
        Dockerfile: app/Dockerfile
        tags: |
          $(tag)

    - publish: manifests
      artifact: manifests
```

```
    - publish: misc
      artifact: misc
```

7. Add a stage at the end of your YAML file to deploy the canary version.

YAML

```yaml
- stage: DeployCanary
  displayName: Deploy canary
  dependsOn: Build
  condition: succeeded()

  jobs:
  - deployment: Deploycanary
    displayName: Deploy canary
    pool:
      vmImage: ubuntu-latest
    environment: 'akscanary.canarydemo'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: KubernetesManifest@0
            displayName: Create imagePullSecret
            inputs:
              action: createSecret
              secretName: azure-pipelines-canary-k8s
              dockerRegistryEndpoint: azure-pipelines-canary-k8s

          - task: KubernetesManifest@0
            displayName: Deploy to Kubernetes cluster
            inputs:
              action: 'deploy'
              strategy: 'canary'
              percentage: '25'
              manifests: |
                $(Pipeline.Workspace)/manifests/deployment.yml
                $(Pipeline.Workspace)/manifests/service.yml
              containers:
'$(containerRegistry)/$(imageRepository):$(tag)'
              imagePullSecrets: azure-pipelines-canary-k8s

          - task: KubernetesManifest@0
            displayName: Deploy Forbio and ServiceMonitor
            inputs:
              action: 'deploy'
              manifests: |
                $(Pipeline.Workspace)/misc/*
```

8. Save your pipeline by committing directly to the main branch. This commit should already run your pipeline successfully.

Manual intervention for promoting or rejecting canary

You can intervene manually with YAML or Classic.

YAML

1. Go to **Pipelines** > **Environments** > **New environment**.
2. Configure the new environment.
   - **Name**: *akspromote*
   - **Resource**: Choose Kubernetes.
3. Select **Next**, and configure your Kubernetes resource as follows:
   - **Provider**: Azure Kubernetes Service
   - **Azure subscription**: Choose the subscription that holds your Kubernetes cluster.
   - **Cluster**: Choose your cluster.
   - **Namespace**: Choose the namespace, *canarydemo*, that you created earlier.
4. Select **Validate and Create**.
5. Select your new `akspromote` environment from the list of environments.
6. Select **Approvals and checks** > **Approvals**. Then select the ellipsis icon (the three dots).
7. Configure your approval as follows:
   - **Approvers**: Add your own user account.
   - **Advanced**: Make sure that the **Allow approvers to approve their own runs** box is selected.
8. Select **Create**.
9. Go to **Pipelines**, and select the pipeline that you created. Then select **Edit**.
10. Add another stage, `PromoteRejectCanary`, at the end of your YAML file, to promote the changes.

    YAML

```yaml
- stage: PromoteRejectCanary
  displayName: Promote or Reject canary
  dependsOn: DeployCanary
  condition: succeeded()

  jobs:
  - deployment: PromoteCanary
    displayName: Promote Canary
    pool:
      vmImage: ubuntu-latest
    environment: 'akspromote.canarydemo'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: KubernetesManifest@0
            displayName: promote canary
            inputs:
              action: 'promote'
              strategy: 'canary'
              manifests: '$(Pipeline.Workspace)/manifests/*'
              containers:
'$(containerRegistry)/$(imageRepository):$(tag)'
              imagePullSecrets: '$(imagePullSecret)'
```

11. Add another stage, `RejectCanary`, at the end of your YAML file, to roll back the changes.

YAML

```yaml
- stage: RejectCanary
  displayName: Reject canary
  dependsOn: PromoteRejectCanary
  condition: failed()

  jobs:
  - deployment: RejectCanary
    displayName: Reject Canary
    pool:
      vmImage: ubuntu-latest
    environment: 'akscanary.canarydemo'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: KubernetesManifest@0
            displayName: reject canary
            inputs:
              action: 'reject'
              strategy: 'canary'
              manifests: '$(Pipeline.Workspace)/manifests/*'
```

12. Save your YAML pipeline by selecting **Save**, and then commit it directly to the main branch.

Deploy a stable version

You can deploy a stable version with YAML or Classic.

YAML

For the first run of the pipeline the stable version of the workloads, and their baseline or canary versions don't exist in the cluster. To deploy the stable version:

1. In *app/app.py*, change `success_rate = 5` to `success_rate = 10`. This change triggers the pipeline, leading to a build and push of the image to the container registry. It will also trigger the `DeployCanary` stage.
2. Because you configured an approval on the `akspromote` environment, the release will wait before running that stage.
3. In the summary of the run, select **Review** > **Approve**. This deploys the stable version of the workloads (the `sampleapp` deployment in *manifests/deployment.yml*) to the namespace.

Initiate canary workflow

The stable version of the workload `sampleapp` now exists in the cluster. Next, make the following change to the simulation application:

In *app/app.py*, change `success_rate = 10` to `success_rate = 20`.

This change triggers the build pipeline, resulting in the build and push of the image to the container registry. This process in turn triggers the release pipeline and begins the **Deploy canary** stage.

Simulate requests

On your development machine, run the following commands, and keep it running to send a constant stream of requests at the `sampleapp` service. `sampleapp` routes the requests to the pods spun by the stable `sampleapp` deployment, and to the pods spun

up by the `sampleapp-baseline` and `sampleapp-canary` deployments. The selector specified for `sampleapp` is applicable to all of these pods.

```
FORTIO_POD=$(kubectl get pod | grep fortio | awk '{ print $1 }')
kubectl exec -it $FORTIO_POD -c fortio /usr/bin/fortio -- load -allow-
initial-errors -t 0 http://sampleapp:8080/
```

Set up Grafana dashboard

1. Run the following port-forwarding command on your local development machine to be able to access Grafana.

   ```
   kubectl port-forward svc/sampleapp-grafana 3000:80
   ```

2. In a browser, open the following URL.

   ```
   http://localhost:3000/login
   ```

3. When you're prompted for credentials, unless the `adminPassword` value was overridden during the `prometheus-operator` Helm chart installation, you can use the following values:

   - username: admin
   - password: prom-operator

4. From the menu on the left, choose + > **Dashboard** > **Graph**.

5. Select anywhere on the newly added panel, and type `e` to edit the panel.

6. On the **Metrics** tab, enter the following query:

   ```
   rate(requests_total{pod=~"sampleapp-.*", custom_status="good"}[1m])
   ```

7. On the **General** tab, change the name of this panel to **All sampleapp pods**.

8. In the overview bar at the top of the page, change the duration range to **Last 5 minutes** or **Last 15 minutes**.

9. To save this panel, select the save icon in the overview bar.

10. The preceding panel visualizes success rate metrics from all the variants. These include stable (from the `sampleapp` deployment), baseline (from the `sampleapp-baseline` deployment), and canary (from the `sampleapp-canary` deployment). You can visualize just the baseline and canary metrics by adding another panel, with the following configuration:

    - On the **General** tab, for **Title**, select **sampleapp baseline and canary**.
    - On the **Metrics** tab, use the following query:

    ```
    rate(requests_total{pod=~"sampleapp-baseline-.*|sampleapp-canary-.*",
    custom_status="good"}[1m])
    ```

    > **Note**
    >
    > The panel for baseline and canary metrics will only have metrics available for comparison under certain conditions. These conditions are when the **Deploy canary** stage has successfully completed, and the **Promote/reject canary** stage is waiting on manual intervention.

    > **Tip**
    >
    > Set up **annotations for Grafana dashboards** to visually depict stage completion events for **Deploy canary** and **Promote/reject canary**. This is helpful so that you know when to start comparing the baseline with the canary, and when the promotion or rejection of the canary has completed, respectively.

Compare baseline and canary

1. At this point, the **Deploy canary** stage has successfully completed (based on the change of `success_rate` from `10` to `20`). The **Promote/reject canary** stage is waiting on manual intervention. You can now compare the success rate (as

determined by `custom_status=good`) of the baseline and canary variants in the Grafana dashboard. It should look similar to the following:

2. Based on the observation that the success rate is higher for canary, promote the canary. Select **Resume** in the manual intervention task.

Artifacts in Azure Pipelines - overview

Article • 12/06/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Azure Artifacts allow developers to publish and consume various types of packages from feeds and public registries like PyPI, Maven Central, and NuGet.org. You can combine Azure Artifacts with Azure Pipelines to publish build and pipeline artifacts, deploy packages, or integrate files across different stages of your pipeline for building, testing, or deploying your application.

Supported artifacts

Expand table

Artifact type	Description
Build artifacts	The files generated by a build. Example: .dll, .exe, and .PDB files.
Pipeline artifacts	Recommended for faster performance if you are using Azure DevOps Services. Not supported in release pipelines.
NuGet	Publish NuGet packages to Azure Artifacts feeds or public registries such as nuget.org.
npm	Publish npm packages to Azure Artifacts feeds or public registries such as nmpjs.com.
Maven	Publish Maven packages to Azure Artifacts feeds or public registries such as Maven Central, Google Maven Repository, Gradle Plugins, and JitPack.
Python	Publish Python packages to Azure Artifacts feeds or public registries such as PyPI.org.
Cargo	Publish Cargo packages to Azure Artifacts feeds or public registries such as Crates.io.
Universal Packages	Publish Universal Packages to Azure Artifacts feeds or Universal Packages upstream sources.
Symbols	Publish symbols to Azure Artifacts symbol server or to a file share.

Publish artifacts

NuGet

Publish NuGet packages with Azure Pipelines (YAML/Classic)

Tip

If your organization is using a firewall or a proxy server, make sure you allow **Azure Artifacts Domain URLs and IP addresses**.

Related articles

- Publish and download pipeline artifacts
- Publish and download build artifacts
- Artifact sources

Publish and download pipeline Artifacts

Article • 10/25/2023

Azure DevOps Services

Using Azure Pipelines, you can download artifacts from earlier stages in your pipeline or from another pipeline. You can also publish your artifact to a file share or make it available as a pipeline artifact.

Publish artifacts

You can publish your artifacts using YAML, the classic editor, or Azure CLI:

> **Note**
>
> Publishing pipeline artifacts is not supported in release pipelines.

YAML

YAML

```yaml
steps:
- publish: $(System.DefaultWorkingDirectory)/bin/WebApp
  artifact: WebApp
```

> **Note**
>
> The `publish` keyword is a shortcut for the **Publish Pipeline Artifact task** .

Although the artifact's name is optional, it is a good practice to specify a name that accurately reflects the contents of your artifact. If you plan to consume the artifact from a job running on a different OS, you must ensure all the file paths are valid for the target environment. For example, a file name containing the character `\` or `*` will fail to download on Windows.

The path of the file/folder that you want to publish is required. This can be an absolute or a relative path to `$(System.DefaultWorkingDirectory)`.

Packages in Azure Artifacts are immutable. Once you publish a package, its version will be permanently reserved. rerunning failed jobs will fail if the package has been published. A good way to approach this if you want to be able to rerun failed jobs without facing an error *package already exists*, is to use Conditions to only run if the previous job succeeded.

yml

```yml
jobs:
- job: Job1
  steps:
    - script: echo Hello Job1!

- job: Job2
  steps:
    - script: echo Hello Job2!
  dependsOn: Job1
```

> ⓘ **Note**
>
> You will not be billed for storing Pipeline Artifacts. Pipeline Caching is also exempt from storage billing. See **Which artifacts count toward my total billed storage**.

> ⊗ **Caution**
>
> Deleting a pipeline run will result in the deletion of all Artifacts associated with that run.

Use .artifactignore

`.artifactignore` uses a similar syntax to `.gitignore` (with few limitations) to specify which files should be ignored when publishing artifacts. Make sure that the *.artifactignore* file is located within the directory specified by the *targetPath* argument of your Publish Pipeline Artifacts task.

> ⓘ **Note**
>
> The plus sign character `+` is not supported in URL paths and some builds metadata for package types such as Maven.

Example: ignore all files except **.exe** files:

```
**/*
!*.exe
```

> **Important**
>
> Azure Artifacts automatically ignore the *.git* folder path when you don't have a *.artifactignore* file. You can bypass this by creating an empty *.artifactignore* file.

Download artifacts

You can download artifacts using YAML, the classic editor, or Azure CLI.

YAML

YAML

```yaml
steps:
- download: current
  artifact: WebApp
```

- **current**: download artifacts produced by the current pipeline run. Options: current, specific.

> **Note**
>
> List of published artifacts will be available only in following dependant jobs. Therefore, use `current` option only in separate jobs, that has dependency on jobs with publish artifacts tasks.

> **Tip**
>
> You can use **Pipeline resources** to define your source in one place and use it anywhere in your pipeline.

> **Note**

The `download` keyword downloads artifacts. For more information, see **steps.download**.

To download a pipeline artifact from a different project within your organization, make sure that you have the appropriate permissions configured for both your downstream project and downstream pipeline. By default, files are downloaded to **$(Pipeline.Workspace)**. If an artifact name was not specified, a subdirectory will be created for each downloaded artifact. You can use matching patterns to limit which files get downloaded. See File matching patterns for more details.

yml

```yml
steps:
- download: current
  artifact: WebApp
  patterns: |
    **/*.js
    **/*.zip
```

Artifacts selection

A single download step can download one or more artifacts. To download multiple artifacts, leave the *artifact name* field empty and use file matching patterns to limit which files will be downloaded. `**` is the default file matching pattern (all files in all artifacts).

Single artifact

When an artifact name is specified:

1. Only files for that specific artifact are downloaded. If the artifact does not exist, the task will fail.
2. File matching patterns are evaluated relative to the root of the artifact. For example, the pattern `*.jar` matches all files with a `.jar` extension at the root of the artifact.

The following example illustrates how to download all `*.js` from an artifact `WebApp`:

YAML

YAML

```yaml
steps:
- download: current
  artifact: WebApp
  patterns: '**/*.js'
```

Multiple artifacts

When no artifact name is specified:

1. Multiple artifacts can be downloaded and the task does not fail if no files are found.
2. A subdirectory is created for each artifact.
3. File matching patterns should assume the first segment of the pattern is (or matches) an artifact name. For example, `WebApp/**` matches all files from the `WebApp` artifact. The pattern `*/*.dll` matches all files with a `.dll` extension at the root of each artifact.

The following example illustrates how to download all `.zip` files from all artifacts:

YAML

YAML

```yaml
steps:
- download: current
  patterns: '**/*.zip'
```

Artifacts in release and deployment jobs

Artifacts are only downloaded automatically in deployment jobs. By default, artifacts are downloaded to `$(Pipeline.Workspace)`. The download artifact task will be auto injected only when using the `deploy` lifecycle hook in your deployment. To stop artifacts from being downloaded automatically, add a `download` step and set its value to none. In a regular build job, you need to explicitly use the `download` step keyword or the Download Pipeline Artifact task. See lifecycle hooks to learn more about the other types of hooks.

YAML

```
steps:
- download: none
```

Use Artifacts across stages

If you want to be able to access your artifact across different stages in your pipeline, you can now publish your artifact in one stage and then download it in the next stage leveraging dependencies. See Stage to stage dependencies for more details.

Example

In the following example, we will copy and publish a script folder from our repo to the `$(Build.ArtifactStagingDirectory)`. In the second stage, we will download and run our script.

YAML

```yaml
trigger:
- main
stages:
- stage: build
  jobs:
  - job: run_build
    pool:
      vmImage: 'windows-latest'
    steps:
    - task: VSBuild@1
      inputs:
        solution: '**/*.sln'
        msbuildArgs: '/p:DeployOnBuild=true /p:WebPublishMethod=Package
/p:PackageAsSingleFile=true /p:SkipInvalidConfigurations=true
/p:DesktopBuildPackageLocation="$(build.artifactStagingDirectory)\WebApp.zip
" /p:DeployIisAppPath="Default Web Site"'
        platform: 'Any CPU'
        configuration: 'Release'

    - task: CopyFiles@2
      displayName: 'Copy scripts'
      inputs:
        contents: 'scripts/**'
        targetFolder: '$(Build.ArtifactStagingDirectory)'

    - publish: '$(Build.ArtifactStagingDirectory)/scripts'
      displayName: 'Publish script'
      artifact: drop

- stage: test
  dependsOn: build
```

```yaml
jobs:
- job: run_test
  pool:
    vmImage: 'windows-latest'
  steps:
  - download: current
    artifact: drop
  - task: PowerShell@2
    inputs:
      filePath: '$(Pipeline.Workspace)\drop\test.ps1'
```

Migrate from build artifacts

Pipeline artifacts are the next generation of build artifacts and are the recommended way to work with artifacts. Artifacts published using the Publish Build Artifacts task can still be downloaded using Download Build Artifacts, but we recommend using the latest Download Pipeline Artifact task instead.

When migrating from build artifacts to pipeline artifacts:

1. By default, the **Download Pipeline Artifact** task downloads files to `$(Pipeline.Workspace)`. This is the default and recommended path for all types of artifacts.

2. File matching patterns for the **Download Build Artifacts** task are expected to start with (or match) the artifact name, regardless if a specific artifact was specified or

not. In the **Download Pipeline Artifact** task, patterns should not include the artifact name when an artifact name has already been specified. For more information, see single artifact selection.

Example

YAML

```yaml
- task: PublishPipelineArtifact@1
  displayName: 'Publish pipeline artifact'
  inputs:
    targetPath: '$(Pipeline.Workspace)'
    ${{ if eq(variables['Build.SourceBranchName'], 'main') }}:
        artifact: 'prod'
    ${{ else }}:
        artifact: 'dev'
    publishLocation: 'pipeline'
```

- **targetPath**: (Required) The path of the file or directory to publish. Can be absolute or relative to the default working directory. Can include variables, but wildcards are not supported. Default: $(Pipeline.Workspace).
- **publishLocation**: (Required) Artifacts publish location. Choose whether to store the artifact in Azure Pipelines, or to copy it to a file share that must be accessible from the pipeline agent. Options: `pipeline`, `filepath`. Default: pipeline.
- **artifact**: (Optional) Name of the artifact to publish. If not set, defaults to a unique ID scoped to the job.

FAQ

Q: What are build artifacts?

A: Build artifacts are the files generated by your build. See Build Artifacts to learn more about how to publish and consume your build artifacts.

Q: Can I delete pipeline artifacts when re-running failed jobs?

A: Pipeline artifacts are not deletable or overwritable. If you want to regenerate artifacts when you re-run a failed job, you can include the job ID in the artifact name. `$(system.JobId)` is the appropriate variable for this purpose. See System variables to learn more about predefined variables.

Q: How can I access Artifacts feeds behind a firewall?

A: If your organization is using a firewall or a proxy server, make sure you allow Azure Artifacts Domain URLs and IP addresses.

Related articles

- Build artifacts
- Releases in Azure Pipelines
- Release artifacts and artifact sources
- How to mitigate risk when using private package feeds

Publish and download build artifacts

Article • 12/06/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> **Note**
>
> We recommend using **Download Pipeline Artifacts** and **Publish Pipeline Artifacts** for faster performance.

Azure Artifacts enables teams to use feeds and upstream sources to manage their dependencies. You can use Azure Pipelines to publish and download different types of artifacts as part of your CI/CD workflow.

Publish artifacts

Artifacts can be published at any stage of your pipeline. You can use YAML or the classic Azure DevOps editor to publish your packages.

YAML

YAML

```yaml
- powershell: gci env:* | sort-object name | Format-Table -AutoSize |
Out-File $env:BUILD_ARTIFACTSTAGINGDIRECTORY/environment-variables.txt

- task: PublishBuildArtifacts@1
  inputs:
    pathToPublish: '$(Build.ArtifactStagingDirectory)'
    artifactName: drop
```

- **pathToPublish**: the path of your artifact. This can be an absolute or a relative path. Wildcards aren't supported.
- **artifactName**: the name of your artifact.

> **Note**

Make sure you aren't using one of the reserved folder names when publishing your artifact. See **Application Folders** for more details.

Example: Use multiple tasks

YAML

YAML

```yaml
- powershell: gci env:* | sort-object name | Format-Table -AutoSize |
Out-File $env:BUILD_ARTIFACTSTAGINGDIRECTORY/environment-variables.txt

- task: PublishBuildArtifacts@1
  inputs:
    pathToPublish: '$(Build.ArtifactStagingDirectory)'
    artifactName: drop1
- task: PublishBuildArtifacts@1
  inputs:
    pathToPublish: '$(Build.ArtifactStagingDirectory)'
    artifactName: drop2
```

- **pathToPublish**: the path of your artifact. This can be an absolute or a relative path. Wildcards aren't supported.
- **artifactName**: the name of your artifact.

Example: Copy and publish binaries

YAML

YAML

```yaml
- powershell: gci env:* | sort-object name | Format-Table -AutoSize |
Out-File $env:BUILD_ARTIFACTSTAGINGDIRECTORY/environment-variables.txt

- task: CopyFiles@2
  inputs:
    sourceFolder: '$(Build.SourcesDirectory)'
    contents: '**/$(BuildConfiguration)/**/?(*.exe|*.dll|*.pdb)'
    targetFolder: '$(Build.ArtifactStagingDirectory)'
- task: PublishBuildArtifacts@1
  inputs:
```

```
    pathToPublish: '$(Build.ArtifactStagingDirectory)'
    artifactName: drop
```

- **sourceFolder**: the folder that contains the files you want to copy. If you leave this empty, copying will be done from **$(Build.SourcesDirectory)**.
- **contents**: File paths to include as part of the copy.
- **targetFolder**: destination folder.
- **pathToPublish**: the folder or file path to publish. It can be an absolute or a relative path. Wildcards aren't supported.
- **artifactName**: the name of the artifact that you want to create.

> **Note**
>
> Make sure not to use reserved name for *artifactName* such as *Bin* or *App_Data*. See **ASP.NET Web Project Folder Structure** for more details.

Download artifacts

YAML

YAML

```
- powershell: gci env:* | sort-object name | Format-Table -AutoSize |
Out-File $env:BUILD_ARTIFACTSTAGINGDIRECTORY/environment-variables.txt

- task: DownloadBuildArtifacts@0
  inputs:
    buildType: 'current'
    downloadType: 'single'
    artifactName: 'drop'
    downloadPath: '$(System.ArtifactsDirectory)'
```

- **buildType**: specify which build artifacts will be downloaded: `current` (the default value) or from a specific build.
- **downloadType**: choose whether to download a single artifact or all artifacts of a specific build.
- **artifactName**: the name of the artifact that will be downloaded.
- **downloadPath**: path on the agent machine where the artifacts will be downloaded.

> **Note**
>
> If you're using a deployment task, you can reference your build artifacts using **$(Agent.BuildDirectory)**. See Agent variables for more details.

When your pipeline run is completed, navigate to **Summary** to explore or download your artifact.

Tips

- Disable IIS Basic Authentication if you're using Azure DevOps Server to allow authentication with your Personal Access Token. See IIS Basic Authentication and PATs for more details.
- Use forward slashes in file path arguments. Backslashes don't work in macOS/Linux agents.
- Build artifacts are stored on a Windows filesystem, which causes all UNIX permissions to be lost, including the execution bit. You might need to restore the correct UNIX permissions after downloading your artifacts from Azure Pipelines or TFS.
- `Build.ArtifactStagingDirectory` and `Build.StagingDirectory` are interchangeable.
- `Build.ArtifactStagingDirectory` path is cleaned up after each build.
- Deleting a build associated with packages published to a file share will result in the deletion of all Artifacts in that UNC path.
- If you're publishing your packages to a file share, make sure you provide access to the build agent.
- Make sure you allow Azure Artifacts Domain URLs and IP addresses if your organization is using a firewall.

Related articles

- Publish and download artifacts in Azure Pipelines
- Define your multi-stage classic pipeline
- How to mitigate risk when using private package feeds

Releases in Azure Pipelines

Article • 08/23/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> **Note**
>
> This topic covers classic release pipelines. If you want to create your pipelines using YAML, see **Customize your pipeline**.

A **release** is a construct that holds a versioned set of artifacts specified in a CI/CD pipeline. It includes a snapshot of all the information required to carry out all the tasks and actions in the release pipeline, such as stages, tasks, policies such as triggers and approvers, and deployment options. There can be multiple releases from one release pipeline, and information about each one is stored and displayed in Azure Pipelines for the specified retention period.

A **deployment** is the action of running the tasks for one stage, which can include running automated tests, deploying build artifacts, and whatever other actions are specified for that stage. Initiating a release starts each deployment based on the settings and policies defined in the original release pipeline. There can be multiple deployments of each release even for one stage. When a deployment of a release fails for a stage, you can redeploy the same release to that stage. To redeploy a release, simply navigate to the release you want to deploy and select deploy.

The following diagram shows the relationship between release, release pipelines, and deployments.

Create release pipelines

Releases can be created in several ways:

1. By using a deployment trigger to create a release every time a new build artifact is available.

2. By using the **Create release** button from within your **Pipelines** > **Releases** to manually create a release pipeline.

3. By using the REST API to create a release definition.

> ⓘ **Note**
>
> If your organization is using a firewall or a proxy server, make sure you allow **Azure Artifacts Domain URLs and IP addresses**.

Q&A

Q: Why my deployment did not get triggered?

A: Creating a release pipeline does not necessarily mean that it will automatically/immediately start a deployment. Below are few reasons why this might happen:

- Defined deployment triggers forcing the deployment to pause.This can happen with scheduled triggers or when a delay is imposed until deployment to another stage is complete.
- Defined queuing policies dictating the order of execution and when releases are queued for deployment.
- Pre-deployment approvals or gates for a specific stage preventing deployment until all the defined conditions are met.

Related articles

- Release deployment control using approvals.

- Release deployment control using gates.
- Release triggers.
- Release artifacts and artifact sources.
- Add stages, dependencies, & conditions.

Release pipelines and Artifact sources

Article • 05/30/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

With Azure Pipelines, you can deploy your artifacts from a wide range of artifact sources and integrate your workflow with different types of artifact repositories. Releases can be linked to multiple artifact sources, where one is designated as the primary source.

Artifact sources

Azure Pipelines supports a wide range of repositories, source control tools, and continuous integration systems.

When creating a release, you can specify the version of your artifact source. By default, releases use the latest version of the source artifact. You can also choose to use the latest build from a specific branch by specifying the tags, a specific version, or allow the user to specify the version at the time of release creation.

If you link more than one artifact, you can specify which one is the primary source (default). The primary artifact source is used to set a number of predefined variables. It can also be used in naming releases.

> **Note**
>
> The `Default version` dropdown items depend on the source type of the linked build definition.

- The following options are supported by all the repository types: `Specify at the time of release creation`, `Specific version`, and `Latest`.
- `Latest from a specific branch with tags` and `Latest from the build pipeline default branch with tags` options are supported by the following repository types: `TfsGit`, `GitHub`, `Bitbucket`, and `GitHubEnterprise`.
- `Latest from the build pipeline default branch with tags` is not supported by `XAML` build definitions.

The following sections describe how to work with the different types of artifact sources.

- Azure Pipelines
- Version control
- Jenkins
- Azure Container Registry, Docker, and Kubernetes
- Azure Artifacts
- TFS server
- TeamCity

Artifact sources - Azure Pipelines

You can link a release pipeline to any Azure Pipelines build. You can also link multiple build pipelines and specify their default values and set up deployment triggers on multiple build sources. When any of the builds completes, it will trigger the creation of a release.

The following features are available when using Azure Pipelines as an artifact source:

Feature	Description
Auto-trigger releases	New releases can be created automatically when a new build artifact is available (including XAML builds). See Release triggers for more details.
Artifact variables	A number of artifact variables are supported fo Azure Pipelines sources.
Work items and commits	You can link Azure Pipelines work items and it will be displayed in the releases details. Commits are displayed when you use Git or TFVC source controls.
Artifact download	By default, build artifacts are downloaded to the agent running the pipeline. You can also configure a step in your stage to skip downloading your artifact.
Deployment stages	The build summary list all the deployment stages where the artifact was deployed to.

> **Note**
>
> You must include a **Publish Artifacts** task in your build pipeline. For YAML build pipelines, an artifact with the name **drop** is published implicitly.

By default, releases run with a collection level job authorization scope. This means that releases can access resources in all projects in the organization (or collection for Azure DevOps Server). This is useful when linking build artifacts from other projects. You can enable **Limit job authorization scope to current project for release pipelines** in project settings to restrict access to a project's artifact.

To set job authorization scope for the organization:

- Navigate to your **Organization settings**.
- Select **Settings** under Pipelines.
- Turn on the toggle **Limit job authorization scope to current project for release pipelines** to limit the scope to current project. This is the recommended setting for a good security measures.

To set job authorization scope for a specific project:

- Navigate to your **Project settings**.
- Select **Settings** under Pipelines.
- Turn on the toggle **Limit job authorization scope to current project for release pipelines** to limit the scope to the current project. This is the recommended setting, as it enhances security for your pipelines.

> ⓘ **Note**
>
> If the scope is set to project at the organization level, you cannot change the scope in each project.

Artifact sources - version control

There are some scenarios in which you may want to consume artifacts from different source controls directly without passing them through a build pipeline. For example:

- Developing a PHP or a JavaScript application that doesn't require an explicit build pipeline.
- You manage configurations for various stages in different version control repositories, and you want to consume these configuration files directly from version control as part of the deployment pipeline.
- You manage your infrastructure and configuration as code and you want to manage these files in a version control repository.

Because you can configure multiple artifact sources in a single release pipeline, you can link both a build pipeline that produces the binaries of your application as well as a version control repository that stores the configuration files into the same pipeline, and use the two sets of artifacts together while deploying.

Azure Pipelines supports Team Foundation Version Control (TFVC) repositories, Git repositories, and GitHub repositories.

You can link a release pipeline to any of the Git or TFVC repositories in any project in your collection (you'll need read access to these repositories). No additional setup is required when deploying version control artifacts within the same collection.

When you link a GitHub repository and select a branch, you can edit the default properties of the artifact types after the artifact has been saved. This is particularly useful in scenarios where the branch for the stable version of the artifact changes, and

continuous delivery releases should use this branch to obtain newer versions of the artifact. You can also specify details of the checkout, such as whether checkout submodules and LFS-tracked files, and the shallow fetch depth.

When you link a TFVC branch, you can specify the changeset to be deployed when creating a release.

The following features are available when using TFVC, Git, and GitHub as an artifact source:

Feature	Description
Auto-trigger releases	New releases can be created automatically when a new build artifact is available (including XAML builds). See Release triggers for more details.
Artifact variables	A number of artifact variables are supported fo Azure Pipelines sources.
Work items and commits	You can link Azure Pipelines work items are it will be displayed in the releases details. Commits are displayed when you use Git or TFVC source controls.
Artifact download	By default, build artifacts are downloaded to the agent running the pipeline. You can also configure a step in your stage to skip downloading your artifact.

By default, releases run with a collection level job authorization scope. This means that releases can access resources in all projects in the organization (or collection for Azure DevOps Server). This is useful when linking build artifacts from other projects. You can enable **Limit job authorization scope to current project for release pipelines** in project settings to restrict access to a project's artifact.

Artifact sources - Jenkins

To consume Jenkins artifacts, you must create a service connection to authenticate with your Jenkins server. See manage service connections and Jenkins service connection for more information. The Jenkins project must be configured with a post build action to publish your artifacts.

The following features are available when using Jenkins as an artifact source:

Feature	Description

Feature	Description
Auto-trigger releases	New releases can be created automatically when a new build artifact is available (including XAML builds). See Release triggers for more details.
Artifact variables	A number of artifact variables are supported fo Azure Pipelines sources.
Work items and commits	You can link Azure Pipelines work items and it will be displayed in the releases details. Commits are displayed when you use Git or TFVC source controls.
Artifact download	By default, build artifacts are downloaded to the agent running the pipeline. You can also configure a step in your stage to skip downloading your artifact.

Artifacts generated by Jenkins builds are typically propagated to storage repositories for archiving and sharing. Azure blob storage is one of the supported repositories, allowing you to consume Jenkins projects that publish to Azure storage as artifact sources in a release pipeline. Azure Pipelines download the artifacts automatically from Azure to the agent running the pipeline. In this scenario, connectivity between the agent and the Jenkins server is not required. Microsoft-hosted agents can be used without exposing the server to internet.

> **Note**
>
> Azure Pipelines may not be able to contact your Jenkins server if, for example, it is within your enterprise network. If this is the case, you can integrate Azure Pipelines with Jenkins by setting up an on-premises agent that can access the Jenkins server. You will not be able to see the name of your Jenkins projects when linking to a build, but you can enter the name in the URL text field.

Artifact sources - containers

When deploying containerized apps, the container image is first pushed to a container registry. You can then deploy your container image to Azure Web App for Containers or a Docker/Kubernetes cluster. You must create a service connection to authenticate with Azure. See manage service connections for more details.

The following features are available when using Azure Container as an artifact source:

Feature	Description

Feature	Description
Auto-trigger releases	New releases can be created automatically when a new build artifact is available (including XAML builds). See Release triggers for more details.
Artifact variables	A number of artifact variables are supported fo Azure Pipelines sources.
Work items and commits	You can link Azure Pipelines work items and it will be displayed in the releases details. Commits are displayed when you use Git or TFVC source controls.
Artifact download	By default, build artifacts are downloaded to the agent running the pipeline. You can also configure a step in your stage to skip downloading your artifact.

> **Note**
>
> When using multiple artifact sources, mapping an artifact source to trigger a particular stage is not supported. A release will be created anytime there is a push to any of the artifact sources. If you wish to do so, Azure Pipelines recommend splitting your release pipeline into multiple releases.

Artifact sources - Azure Artifacts

Below are some of the scenarios where you can use Azure Artifacts as an artifact source:

1. Your application binary is published to Azure Artifacts and you want to consume the package in a release pipeline.
2. You need additional packages stored in Azure Artifacts as part of your deployment workflow.

Using Azure Artifacts in your release pipeline, you must select the *Feed*, *Package*, and the *Default version* for your package. You can choose to pick up the *latest version of the package*, use a *specific version*, or *select the version at the time of release creation*. During deployment, the package gets downloaded/extracted to the agent running your pipeline.

The following features are available when using Azure Artifacts as an artifact source:

Feature	Description

Feature	Description
Auto-trigger releases	New releases can be created automatically when a new build artifact is available (including XAML builds). See Release triggers for more details.
Artifact variables	A number of artifact variables are supported fo Azure Pipelines sources.
Work items and commits	You can link Azure Pipelines work items and it will be displayed in the releases details. Commits are displayed when you use Git or TFVC source controls.
Artifact download	By default, build artifacts are downloaded to the agent running the pipeline. You can also configure a step in your stage to skip downloading your artifact.

Handling Maven snapshots

When using Maven snapshots, multiple versions can be downloaded at once (example `myApplication-2.1.0.BUILD-20190920.220048-3.jar`, `myApplication-2.1.0.BUILD-20190820.221046-2.jar`, `myApplication-2.1.0.BUILD-20190820.220331-1.jar`). You might need to remove the old version and only keep the latest Artifact before deployment. Run the following PowerShell command in an elevated command prompt to remove all copies except the one with the highest lexicographical value:

PowerShell

```
Get-Item "myApplication*.jar" | Sort-Object -Descending Name | Select-Object
-SkipIndex 0 | Remove-Item
```

> **Note**
>
> You can store up to 30 Maven snapshots in your feed. Once you reach the maximum limit, Azure Artifacts will automatically delete snapshots down to 25. This process will be triggered automatically every time 30+ snapshots are published to your feed.

Artifact sources - TFS server

You can use Azure Pipelines to deploy artifacts from TFS servers without having to make your server discoverable on the Internet by setting up an on-premises automation agent. Artifacts are downloaded to the on-premises agent and then deployed to the

specified target servers without leaving your enterprise network. This is ideal for customers to leverage their investments of their on-premises infrastructure while taking advantage of Azure Pipelines releases.

To use TFS servers as an artifact source, you must install the TFS artifacts for Azure Pipelines extension from the Visual Studio Marketplace, and then create a service connection to authenticate with Azure Pipelines. Once authenticated, you can then link a TFS build pipeline to your release pipeline and choose **External TFS Build** from the *Type* dropdown menu.

The following features are available when using TFS servers as an artifact source:

Feature	Description
Auto-trigger releases	New releases can be created automatically when a new build artifact is available (including XAML builds). See Release triggers for more details.
Artifact variables	A number of artifact variables are supported fo Azure Pipelines sources.
Work items and commits	You can link Azure Pipelines work items and it will be displayed in the releases details. Commits are displayed when you use Git or TFVC source controls.
Artifact download	By default, build artifacts are downloaded to the agent running the pipeline. You can also configure a step in your stage to skip downloading your artifact.

Azure Pipelines may not be able to contact an on-premises TFS server in case it's within your enterprise network. In that case you can integrate Azure Pipelines with TFS by setting up an on-premises agent that can access the TFS server. You will not be able to see the name of your TFS projects or build pipelines when linking to a build, but you can include those variables in the URL text fields. In addition, when you create a release, Azure Pipelines may not be able to query the TFS server for the build numbers. Instead, enter the *Build ID* (not the build number) of the desired build in the appropriate field, or select the *Latest* build.

Artifact sources - TeamCity

To use TeamCity as an Artifact source, you must first install the TeamCity artifacts for Azure Pipelines extension from Visual Studio Marketplace.

Once completed, create a service connection to authenticate with your TeamCity server. You can then link your build artifact to a release pipeline. The TeamCity build configuration must be set up with an action to publish artifacts.

The following features are available when using TeamCity as an artifact source:

Feature	Description
Auto-trigger releases	New releases can be created automatically when a new build artifact is available (including XAML builds). See Release triggers for more details.
Artifact variables	A number of artifact variables are supported fo Azure Pipelines sources.
Work items and commits	You can link Azure Pipelines work items and it will be displayed in the releases details. Commits are displayed when you use Git or TFVC source controls.
Artifact download	By default, build artifacts are downloaded to the agent running the pipeline. You can also configure a step in your stage to skip downloading your artifact.

Azure Pipelines may not be able to contact your TeamCity server if, for example, it is within your enterprise network. In this case you can integrate Azure Pipelines with TeamCity by setting up an on-premises agent that can access the TeamCity server. You will not be able to see the name of your TeamCity projects when linking to a build, but you can type this into the URL text field.

Artifact source alias

To ensure the uniqueness of every artifact download, each artifact source linked to a release pipeline is automatically provided with a specific download location known as the *source alias*. This location can be accessed by using the variable: `$(System.DefaultWorkingDirectory)\[source alias]`

Using a source aliases ensures that renaming a linked artifact source doesn't require editing the task properties because the download location defined in the agent does not change.

By default, the source alias is the name of the artifact source prefixed with an underscore. Depending on the type of the artifact source, this will be the name of the build pipeline, job name, project name, or the repository name. You can edit the source alias from the artifacts tab of your release pipeline.

Artifact download

When a deployment is completed to a stage, the versioned artifacts from each of the sources are downloaded to the pipeline agent so that tasks running within that stage

can access those artifacts. The downloaded artifacts do not get deleted when a release is completed. However, when you initiate the next release, the downloaded artifacts are deleted and replaced with the new set of artifacts.

A new unique folder in the agent is created for every release pipeline when a release is initiated, and the artifacts are downloaded to the following folder: `$(System.DefaultWorkingDirectory)`.

Azure Pipelines does not perform any optimization to avoid downloading the unchanged artifacts if the same release is deployed again. In addition, because the previously downloaded contents are always deleted when you initiate a new release, Azure Pipelines cannot perform incremental downloads to the agent.

Related articles

- Classic release and artifacts variables
- Classic release pipelines
- Publish and download pipeline Artifacts
- Add stages, dependencies, & conditions

Publish NuGet packages with Azure Pipelines (YAML/Classic)

Article • 05/24/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

In Azure Pipelines, you can use the classic editor or the YAML tasks to publish your NuGet packages within your pipeline, to your Azure Artifacts feed, or to public registries such as *nuget.org*.

Create a NuGet package

There are various ways to create your NuGet packages such as using Visual Studio to pack your NuGet packages. If you're already using MSBuild or some other task to create your packages, skip this section and jump to the publish NuGet packages section.

YAML

To create a NuGet package, add the following snippet to your pipeline YAML file. See NuGet task for more details.

YAML

```yaml
- task: NuGetCommand@2
  inputs:
    command: pack
    packagesToPack: '**/*.csproj'
    packDestination: '$(Build.ArtifactStagingDirectory)'
```

- **packagesToPack**: pattern to search for csproj directories to pack
- **packDestination**: directory where packages will be created

Package versioning

NuGet packages are distinguished by their names and version numbers. Employing Semantic Versioning is a recommended strategy for effectively managing package

versions. Semantic versions consist of three numeric components: Major, Minor, and Patch.

The *Patch* is usually incremented after fixing a bug. When you release a new backward-compatible feature, you increment the *Minor* version and reset the *Patch* version to 0, and when you make a backward-incompatible change, you increment the *Major* version and reset the *Minor* and *Patch* versions to 0.

With Semantic Versioning, you can also use prerelease labels to tag your packages. To do so, enter a hyphen followed by your prerelease tag: E.g.**1.0.0-beta**. Semantic Versioning is supported in Azure Pipelines and can be configured in your NuGet task as follows:

- **Use the date and time** (Classic): **byPrereleaseNumber** (YAML). Your package version will be in the format: *Major.Minor.Patch-ci-datetime* where you have the flexibility to choose the values of your Major, Minor, and Patch.
- **Use an environment variable** (Classic): **byEnvVar** (YAML). Your package version will be set to the value of the environment variable you specify.
- **Use the build number** (Classic): **byBuildNumber** (YAML). Your package version will be set to the build number. Make sure you set your build number format under your pipeline **Options** to `$(BuildDefinitionName)_$(Year:yyyy).$(Month).$(DayOfMonth)$(Rev:.r)`. To do this in YAML, add a property `name:` at the root of your pipeline and add your format.

The following example shows how to use the date and time versioning option. This will generate a SemVer compliant version formatted as: `Major.Minor.Patch-ci-datetime`.

YAML

YAML

```yaml
variables:
  Major: '1'
  Minor: '0'
  Patch: '0'

steps:
- task: NuGetCommand@2
  inputs:
    command: pack
    versioningScheme: byPrereleaseNumber
    majorVersion: '$(Major)'
```

```
    minorVersion: '$(Minor)'
    patchVersion: '$(Patch)'
```

> **Note**
>
> `DotNetCore` and `DotNetStandard` packages should be packaged with the `DotNetCoreCLI@2` task to avoid System.InvalidCastExceptions. See the **.NET Core CLI task** for more details.

YAML

```yaml
task: DotNetCoreCLI@2
inputs:
    command: pack
    versioningScheme: byPrereleaseNumber
    majorVersion: '$(Major)'
    minorVersion: '$(Minor)'
    patchVersion: '$(Patch)'
```

Publish NuGet packages

To publish packages to an Azure Artifacts feed from your pipeline, you must set the **Project Collection Build Service** identity to be a **Contributor** on your feed. See Configure feed settings for more details.

YAML

YAML

```yaml
steps:
- task: NuGetAuthenticate@0
  displayName: 'NuGet Authenticate'
- task: NuGetCommand@2
  displayName: 'NuGet push'
  inputs:
    command: push
    publishVstsFeed: '<projectName>/<feed>'
    allowPackageConflicts: true
```

To publish a package to an external NuGet feed, you must first create a service connection to connect to that feed. You can do this by going to **Project settings** > **Service connections** > **New service connection**. Select **NuGet**, and then select

Next. Fill out the form and then select **Save** when you're done. See Manage service connections for more details.

To publish a package to an external NuGet feed, add the following snippet to your YAML pipeline.

Using the Command line task (with NuGet.exe):

YAML

```yaml
  - task: NuGetAuthenticate@1
    inputs:
      nuGetServiceConnections: <NAME_OF_YOUR_SERVICE_CONNECTION>

  - script: |
      nuget push <PACKAGE_PATH> -src
https://pkgs.dev.azure.com/<ORGANIZATION_NAME>/<PROJECT_NAME>/_packaging
/<FEED_NAME>/nuget/v3/index.json -ApiKey <ANY_STRING>
    displayName: "Push"
```

Using the Command line task (with dotnet):

YAML

```yaml
  - task: NuGetAuthenticate@1
    inputs:
      nuGetServiceConnections: <NAME_OF_YOUR_SERVICE_CONNECTION>

  - script: |
      dotnet build <CSPROJ_PATH> --configuration <CONFIGURATION>
      dotnet pack <CSPROJ_PATH> -p:PackageVersion=<YOUR_PACKAGE_VERSION>
--output <OUTPUT_DIRECTORY> --configuration <CONFIGURATION>
      dotnet nuget push <PACKAGE_PATH> --source
https://pkgs.dev.azure.com/<ORGANIZATION_NAME>/<PROJECT_NAME>/_packaging
/<FEED_NAME>/nuget/v3/index.json --api-key <ANY_STRING>
    displayName: "Build, pack and push"
```

> **Note**
>
> The `ApiKey` is required, but you can use any arbitrary value when pushing to Azure Artifacts feeds.

Publish to NuGet.Org

1. Generate an API key

2. Navigate to your Azure DevOps project and then select **Project settings**.

3. Select **Service Connections**, and then select **New service connection**.

4. Select **NuGet**, and then select **Next**.

5. Select **ApiKey** as your authentication method. Use the following url for your **Feed URL**: *https://api.nuget.org/v3/index.json*.

6. Enter the **ApiKey** you generated earlier, and then enter a **Service connection name**.

7. Select **Grant access permission to all pipelines**, and then select **Save** when you're done. To select this option, you'll need the service connection **Administrator** role.

YAML

Add the following YAML snippet to your pipeline definition:

yml

```yml
steps:
- task: NuGetCommand@2
  displayName: 'NuGet push'
  inputs:
    command: push
    nuGetFeedType: external
    publishFeedCredentials: nuget.org
```

Related articles

- Publish npm packages with Azure Pipelines
- Publish and download Universal Packages in Azure Pipelines
- Releases in Azure Pipelines
- Release artifacts and artifact sources

Publish npm packages (YAML/Classic)

Article • 01/06/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Using Azure Pipelines, you can publish your npm packages to Azure Artifacts feeds or to public registries such as *npmjs.com*. In this article, you will learn how to publish your npm packages using YAML and Classic pipelines.

Publish to Azure Artifacts feeds

YAML

> **Note**
>
> The **Project Collection Build Service** and your project's **Build Service** identity must be set to **Contributor** to publish your packages to a feed using Azure Pipelines. See **Add new users/groups** for more details.

YAML

```yaml
- task: Npm@1
  inputs:
    command: publish
    publishRegistry: useFeed
    publishFeed: <FEED_NAME>        ## For project-scoped feeds, use:
<PROJECT_NAME>/<FEED_NAME>
```

- **publishRegistry**: Options: *useExternalRegistry*, *useFeed*. Select *useFeed* to use a feed within your organization.
- **publishFeed**: Required when publishRegistry = useFeed. The feed you want to publish to.

> **Tip**
>
> Using the YAML editor to add the npm publish task will generate the project and feed IDs for your `publishFeed`.

Publish to a public registry

To publish your packages to a public npm registry such as *npmjs.com*, you must first create a service connection to connect to the desired external service.

1. Select **Project settings**, and then select **Service connections**.

2. Select **Create service connection** to create a new service connection.

3. Select **npm** and then select **Next**. Fill out the required fields, and then select **Save** when you are done.

YAML

```yaml
- task: Npm@1
  inputs:
    command: publish
    publishRegistry: useExternalRegistry
    publishEndpoint: '<NAME_OF_YOUR_SERVICE_CONNECTION>'
```

- **publishRegistry**: Select *useExternalRegistry* to publish to a public registry. Options: useExternalRegistry | useFeed.
- **publishEndpoint**: required when `publishRegistry == useExternalRegistry`. Replace the placeholder with the name of the service connection you created earlier.

Related articles

- Publish and download Artifacts in Azure Pipelines.
- Publish npm packages from the command line.
- Use packages from npmjs.com.

Publish Maven artifacts with Azure Pipelines

Article • 02/01/2023

Using Azure Pipelines, you can publish your Maven packages to Azure Artifacts feeds, public registries, or as a pipeline artifact.

Set up your project

1. Add the following snippet to the `repositories` and `distributionManagement` sections in your *pom.xml* file. Replace the placeholders with your organization name, project name, and your feed name.

   XML

   ```xml
   <repository>
     <id>MavenDemo</id>
     <url>https://pkgs.dev.azure.com/ORGANIZATION-NAME/PROJECT-
   NAME/_packaging/FEED-NAME/maven/v1</url>
     <releases>
       <enabled>true</enabled>
     </releases>
     <snapshots>
       <enabled>true</enabled>
     </snapshots>
   </repository>
   ```

2. Configure your *settings.xml* file as follows. Replace the placeholders with your organization name, your project name, and your personal access token.

   XML

   ```xml
   <server>
     <id>PROJECT-NAME</id>
     <username>ORGANIZATION-NAME</username>
     <password>PERSONAL-ACCESS-TOKEN</password>
   </server>
   ```

3. Create a Personal Access Token with **Packaging read & write** scope and paste it into the `password` tag in your *settings.xml* file.

Build your code

In this example, we're using the Maven task to build the project with Azure Pipelines.

yml

```yml
- task: Maven@3
  inputs:
    mavenPomFile: 'my-app/pom.xml'    // Path to your pom file
    mavenOptions: '-Xmx3072m'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.8'
    jdkArchitectureOption: 'x64'
    publishJUnitResults: true
    testResultsFiles: '**/surefire-reports/TEST-*.xml'
    goals: 'package'
```

Publish packages to your pipeline

The following example illustrates how to publish your artifact to *drop* in your pipeline. Use the Copy files task to copy your packages to a target folder, then use Publish Build Artifacts to publish your build artifacts to Azure Pipelines.

yml

```yml
- task: CopyFiles@2
  inputs:
    Contents: '**'
    TargetFolder: '$(build.artifactstagingdirectory)'
- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory)'
    ArtifactName: 'drop'
    publishLocation: 'Container'
```

Publish packages to your feed

To publish your package to an Azure Artifacts feed, use the Maven task to deploy your artifact to your feed.

yml

```yml
- task: Maven@3
  inputs:
    mavenPomFile: 'my-app/pom.xml'
    mavenOptions: '-Xmx3072m'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.8'
    jdkArchitectureOption: 'x64'
    mavenAuthenticateFeed: true
    publishJUnitResults: false
    testResultsFiles: '**/surefire-reports/TEST-*.xml'
    goals: 'deploy'
```

Q&A

Q: How to authenticate with MavenAuthenticate?

A: You can use the MavenAuthenticate task to authenticate with Maven feeds inside and outside your organization. See the examples below for more details:

- Authenticate with Maven feeds inside your organization
- Authenticate with Maven feeds outside your organization

Related articles

- Publish npm packages with Azure Pipelines
- Release artifacts and artifact sources
- Publish NuGet packages with Azure Pipelines

Build and publish artifacts with Gradle and Azure Pipelines

Article • 10/04/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Gradle is a popular build tool for Java applications and the primary build tool for Android. Using Azure Pipelines, we can add the gradle task to our build definition and build and publish our build artifacts.

Prerequisites

- Install Java.
- Install Gradle.

To make sure you have all the prerequisites set up, run the following command in an elevated command prompt to check which Java version is installed on your machine.

Command

```
java -version
```

If the above command doesn't return a java version, make sure you go back and install the Java JDK or JRE first.

To confirm the installation of Gradle, run the following command in an elevated command prompt:

Command

```
gradle -v
```

Set up authentication

1. Select **User settings**, and then select **Personal access tokens**

2. Select **New Token**, and then fill out the required fields. Make sure you select the **Packaging** > **Read & write** scope.

3. Select **Create** when you're done.

4. Copy your token and save it in a secure location.

5. Create a new file in your `.gradle` folder and name it **gradle.properties**. The path to your gradle folder is usually in `%INSTALLPATH%/gradle/user/home/.gradle/`.

6. Open the **gradle.properties** file with a text editor and add the following snippet:

```
vstsMavenAccessToken=<PASTE_YOUR_PERSONAL_ACCESS_TOKEN_HERE>
```

7. Save your file when you're done.

Build projects with Gradle CLI

1. Open your *build.gradle* file and make sure it starts with the following:

```groovy
apply plugin: 'java'
```

2. Add the following snippet to your *build.gradle* file to download your artifact during the build. Replace the placeholders with your groupID, artifactID, and versionNumber. For example: `compile(group: 'siteOps', name: 'odata-wrappers', version: '1.0.0.0')

```groovy
dependencies {
    compile(group: '<YOUR_GROUP_ID>', name: '<ARTIFACT_ID>', version:
'<VERSION_NUMBER>')
}
```

To test this, we can create a sample Java console app and build it with Gradle.

```Java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Run the following command to build your project. Your build output should return: `BUILD SUCCESSFUL`

```Command
gradle build
```

Use Gradle in Azure Pipelines

1. Run the following command to create the Gradle wrapper **gradlew**.

```cli
gradle wrapper
```

2. Push your changes to your remote branch. We will need this file later when we add the **Gradle** task.

3. Navigate to your pipeline definition. If you don't have one, create a new pipeline, select **Use the classic editor** and then select the **Gradle** template.

4. You can use the default settings with the **gradlew build** task.

5. The **Publish build artifacts** task will publish our artifact to Azure Pipelines.

6. Select **Save & queue** when you're done.

7. You can view your published artifact in your pipeline **Summary** once the run is complete.

Related articles

- Publish and download pipeline Artifacts
- Restore NuGet packages in Azure Pipelines
- Artifacts in Azure Pipelines

Publish Python packages with Azure Pipelines

Article • 11/07/2023

Azure DevOps Services

Using Azure Pipelines, developers can publish Python packages to Azure Artifacts feeds, public registries, or store them as pipeline artifacts. This article will guide you through how to:

- ✓ Install the prerequisites
- ✓ Connect to an Azure Artifacts feed
- ✓ Publish Python packages to an Azure Artifacts feed

Prerequisites

- An Azure DevOps organization and a project. Create an organization or a project if you haven't already.
- An Azure Artifacts feed. Create a feed if you don't have one already.

Authenticate with Azure Artifacts

To use `twine` for publishing your Python packages, you must first authenticate with your Azure Artifacts feed. The TwineAuthenticate task provides twine credentials to a `PYPIRC_PATH` environment variable. This variable is then used by `twine` to facilitate the publishing of your packages directly from your pipeline.

YAML

YAML

```yaml
- task: TwineAuthenticate@1
  inputs:
    artifactFeed: <PROJECT_NAME/FEED_NAME>     ## For an organization-scoped feed, artifactFeed: <FEED_NAME>
```

ⓘ **Important**

The credentials stored in the `PYPIRC_PATH` environment variable supersede those in your `.ini` and `.conf` files.

If you add multiple TwineAuthenticate tasks at different stages in your pipeline, each additional task execution will extend **(not override)** the existing `PYPIRC_PATH` environment variable.

Publish Python packages to an Azure Artifacts feed

YAML

YAML

```yaml
- script: |
    pip install build
    pip install twine
  displayName: 'Install build and twine'

- script: |
    python -m build -w
  displayName: 'Python build'

- task: TwineAuthenticate@1
  inputs:
    artifactFeed: <PROJECT_NAME/FEED_NAME>
  displayName: 'Twine Authenticate'

- script: |
    python -m twine upload -r <FEED_NAME> --config-file $(PYPIRC_PATH)
dist/*.whl
  displayName: 'Upload to feed'
```

Note

To publish your packages to a feed using Azure Pipelines, both the **Project Collection Build Service** and your project's **Build Service** identities must have the **Contributor** role assigned in your feed settings. See Manage permissions for details.

Related articles

- Publish and download Python packages CLI
- Search for packages in upstream sources
- Configure permissions

Publish Cargo packages with Azure Pipelines

Article • 12/18/2023

Azure Pipelines enables developers to publish their Cargo packages to Azure Artifacts feeds and public registries such as Crates.io. In this article, you will learn how to publish your Cargo packages to an Azure Artifacts feed using both YAML and Classic pipelines.

This article will guide you through how to:

- ✓ Create an Azure Artifacts feed
- ✓ Authenticate with Azure Artifacts
- ✓ Publish Cargo packages

Prerequisites

- An Azure DevOps organization. Create an organization if you haven't already.
- An Azure DevOps project. Create a project if you don't have one yet.

Create a feed

Azure Artifacts recommends having a dedicated feed for consuming crates from crates.io and a separate feed exclusively for publishing internal crates. If you already have a feed, you can proceed to the next section.

1. Sign in to your Azure DevOps organization, and then navigate to your project.
2. Select **Artifacts**, and then select **Create Feed**.
3. Provide a **Name** for your feed, specify its **Visibility**, and then choose a **Scope** for your feed.
4. Select **Create** when you're done.

Authenticate with Azure Artifacts

1. Sign in to your Azure DevOps organization, and then navigate to your project.

2. Select **Artifacts**, and then select your feed.

3. Select **Connect to feed**, and then select **Cargo** from the left pane.

4. Follow the instructions in the **Project setup** section by adding the provided snippet to your *config.toml* file in your source repository:

    - Project-scoped feed:

```
[registries]
<FEED_NAME> = { index =
"sparse+https://pkgs.dev.azure.com/<ORGANIZATION_NAME>/<PROJECT_NA
ME>/_packaging/<FEED_NAME>/Cargo/index/" }

[source.crates-io]
replace-with = "<FEED_NAME>"
```

- Organization-scoped feed:

```
[registries]
<FEED_NAME> = { index =
"sparse+https://pkgs.dev.azure.com/<ORGANIZATION_NAME>/_packaging/
<FEED_NAME>/Cargo/index/" }

[source.crates-io]
replace-with = "<FEED_NAME>"
```

5. Create a Personal access token with **Packaging** > **Read & write** scopes to authenticate with your feed.

6. Use the CargoAuthenticate task to authenticate from your pipeline:

Classic

a. Sign in to your Azure DevOps organization, and then navigate to your project.

b. Select **Pipelines**, select your pipeline definition, and then select **Edit**.

c. Select the `+` sign to add a new task. Search for the **Cargo Authenticate** task, and then select **Add** to add it to your pipeline.

d. Select the ellipsis icon to open a new window displaying your repository contents, and then choose your *config.toml* file.

Publish crates to your feed

Classic

1. From your Azure DevOps project, select **Pipelines**, select your pipeline definition, and then select **Edit**.

2. Select the `+` sign on your agent job to add a new task. Find the **PowerShell** task through the search function, and then select **Add** to add it to your pipeline.

3. Give your task a name, e.g., Publish, and then select **Inline** as the type. Paste your publish command inline, replacing the placeholder with your feed name:

PowerShell

```
cargo publish --registry <FEED_NAME>
```

Example

In this example, we will install rustup on the agent, set up the PATH environment variable, build our project, authenticate with CargoAuthenticate, and finally publish our crate to our Azure Artifacts feed:

Windows

YAML

```yaml
trigger:
- main

pool:
  vmImage: windows-latest

steps:
- powershell: |
   Invoke-WebRequest -Uri https://sh.rustup.rs -OutFile rustup-init.sh
```

```
    bash .\rustup-init.sh -y
    echo "##vso[task.prependpath]$env:USERPROFILE\.cargo\bin"
  displayName: Install

- task: CargoAuthenticate@0
  displayName: 'cargo Authenticate'
  inputs:
    configFile: '.cargo/config.toml'

- script: |
    cargo build --all
  displayName: Build

- powershell: |
    cargo publish --registry CargoInternalFeed
  displayName: Publish
```

After your pipeline run is completed, your crate should be available in your feed, as shown below:

Related articles

- Delete/Yank Cargo packages
- Promote a package to a view
- Feed permissions

Publish and download Universal Packages with Azure Pipelines

Article • 05/05/2023

Azure DevOps Services

Universal Packages allow you to package any number of files of any type and share them with your team. Using the Universal Package task in Azure Pipelines, you can pack, publish, and download packages of various sizes, up to 4 TB. Each package is uniquely identified with a name and a version number. You can use Azure CLI or Azure Pipelines to publish and consume packages from your Artifacts feeds.

> **Note**
>
> Universal Packages are only available in Azure DevOps Services.

Copy files

The Universal Packages task in Azure Pipelines is set to use `$(Build.ArtifactStagingDirectory)` as the default publish directory. To ready your Universal Package for publishing, move the files you wish to publish to that directory. You can also use the Copy Files utility task to copy those files to the publish directory.

Publish a Universal Package

YAML

To publish a Universal Package to your Azure Artifacts feed, add the following task to your pipeline's YAML file.

YAML

```yaml
- task: UniversalPackages@0
  displayName: Publish a Universal Package
  inputs:
    command: publish
    publishDirectory: '$(Build.ArtifactStagingDirectory)'
    vstsFeedPublish: '<projectName>/<feedName>'
    vstsFeedPackagePublish: '<Package name>'
    packagePublishDescription: '<Package description>'
```

Argument	Description
publishDirectory	Location of the files you wish to publish.
vstsFeedPublish	The project and feed name to publish to. If you're working with an organization-scoped feed, specify only the feed name.
vstsFeedPackagePublish	The package name. Must be lower case. Use only letters, numbers, and dashes.
packagePublishDescription	Description of the package content.

To publish packages to an Azure Artifacts feed from your pipeline, you must add the **Project Collection Build Service** identity as a **Contributor** from your feed's settings. See Adding users/groups permissions to a feed for more details.

To publish to an external feed, you must first create a service connection to authenticate with your feed. see Manage service connection for more details.

Package versioning

Universal Packages follow the semantic versioning specification and can be identified by their names and version numbers. Semantic version numbers are composed of three numeric components, Major, Minor, and Patch, in the format: `Major.Minor.Patch`.

The minor version number is incremented when new features are added that are backward compatible with previous versions, in this case, you increment the minor version and reset the patch version to 0 (`1.4.17` to `1.5.0`). The major version number is incremented when there are significant changes that could break compatibility with previous versions. In this case, you increment the major version and reset the minor and patch versions to 0 (`2.6.5` to `3.0.0`). The patch version number should be incremented when only bug fixes or other small changes are made that do not affect compatibility with previous versions (`1.0.0` to `1.0.1`).

When publishing a new package, the Universal Packages task will automatically select the next major, minor, or patch version for you.

YAML

To enable versioning for your package, add a `versionOption` input to your YAML file. The options for publishing a new package version are: `major`, `minor`, `patch`, or `custom`.

Selecting `custom` enables you to manually specify your package version. The other options will get the latest package version from your feed and increment the chosen version segment by 1. So if you have a *testPackage 1.0.0*, and select the *major* option, your new package will be *testPackage 2.0.0*. If you select the *minor* option, your package version will be *1.1.0*, and if you select the *patch* option, your package version will be *1.0.1*.

Note that if you choose the `custom` option, you must also specify a `versionPublish` value as follows:

YAML

```yaml
- task: UniversalPackages@0
  displayName: Publish a Universal Package
  inputs:
    command: publish
    publishDirectory: '$(Build.ArtifactStagingDirectory)'
    vstsFeedPublish: '<projectName>/<feedName>'
    vstsFeedPackagePublish: '<Package name>'
    versionOption: custom
    versionPublish: '<Package version>'
    packagePublishDescription: '<Package description>'
```

Argument	Description
publishDirectory	Location of the files you wish to publish.
vstsFeedPublish	The project and feed name to publish to. If you're working with an organization-scoped feed, specify only the feed name.
vstsFeedPackagePublish	The package name. Must be lower case. Use only letters, numbers, and dashes.
versionOption	Select a versioning strategy. Options: `major`, `minor`, `patch`, `custom`.
versionPublish	The custom package version.
packagePublishDescription	Description of the package content.

Download a Universal Package

YAML

To download a Universal Package from a feed in your organization, use the Universal Package task with the `download` command as follows:

YAML

```yaml
steps:
- task: UniversalPackages@0
  displayName: Download a Universal Package
  inputs:
    command: download
    vstsFeed: '<projectName>/<feedName>'
    vstsFeedPackage: '<packageName>'
    vstsPackageVersion: '<packageVersion>'
    downloadDirectory: '$(Build.SourcesDirectory)\someFolder'
```

Argument	Description
vstsFeed	The Artifacts feed hosting the package to be downloaded.
vstsFeedPackage	Name of the package to be downloaded.
vstsPackageVersion	Version of the package to be downloaded.
downloadDirectory	The package destination folder. Default value: $(System.DefaultWorkingDirectory).

To download a Universal Package from an external source, use the following snippet:

YAML

```yaml
steps:
- task: UniversalPackages@0
  displayName: Download a Universal Package
  inputs:
    command: download
    feedsToUse: external
    externalFeedCredentials: 'MSENG2'
    feedDownloadExternal: 'fabrikamFeedExternal'
    packageDownloadExternal: 'fabrikam-package'
    versionDownloadExternal: 1.0.0
```

Argument	Description
feedsToUse	Set the value to `external` when downloading from an external source.

Argument	Description
externalFeedCredentials	Name of the service connection to the external feed. See manage service connections for more details.
feedDownloadExternal	Name of the external feed.
packageDownloadExternal	The package name you wish to download.
versionDownloadExternal	The version of the package you wish to download.

Tip

You can use wildcards to download the latest version of a Universal Package. See **Download the latest version** for more details.

Related articles

- Universal Packages upstream sources
- Search for packages in upstream sources
- Feed permissions

Publish symbols with Azure Pipelines

Article • 09/25/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

With Azure Pipelines, you can publish your symbols to Azure Artifacts symbol server using the *Index sources and publish symbols* task. You can use the debugger to connect and automatically retrieve the correct symbol files without knowing product names, build numbers, or package names. Using Azure Pipelines, you can also publish your symbols to files shares and portable PDBs.

> ⓘ **Note**
>
> The **Index sources and publish symbols** task is not supported in release pipelines.

Publish symbols to Azure Artifacts symbol server

Classic

1. Sign in to your Azure DevOps organization, and then navigate to your project.
2. Select **Pipelines**, select your pipeline, and then select **Edit** to modify your pipeline.
3. From your pipeline definition, select `+` to add a new task.
4. Search for the **Index sources and publish symbols** task. Select **Add** to add it to your pipeline.
5. Fill out the required fields as follows:
    - **Task version**: 2.*.
    - **Display name**: task display name.
    - **Path to symbols folder**: path to the folder hosting the symbol files.
    - **Search pattern**: the pattern used to locate the *.pdb* files in the folder you've designated under *Path to symbols folder*. Single-folder wildcard

(`*`) and recursive wildcards (`**`) are both supported. Example: **bin***.pdb*: will search for all *.pdb* files within all subdirectories named *bin*.

- **Index sources**: indicates whether to inject source server information into the PDB files.
- **Publish symbols**: indicates whether to publish the symbol files.
  - **Symbol server type**: select **Symbol Server in this organization/collection (requires Azure Artifacts)** to publish your symbols to Azure Artifacts symbol server.
- **Verbose logging**: include more information in your logs.

Publish symbols to a file share

Aside from Azure Artifacts symbol server, you can also publish your symbols to a file share using the *Index Sources and Publish Symbols* task.

Classic

1. Sign in to your Azure DevOps organization, and then navigate to your project.
2. Select **Pipelines**, select your pipeline, and then select **Edit** to modify your pipeline.

3. From your pipeline definition, select `+` to add a new task.

4. Search for the **Index sources and publish symbols** task. Select **Add** to add it to your pipeline.

5. Fill out the required fields as follows:

    - **Task version**: 2.*.
    - **Display name**: task display name.
    - **Path to symbols folder**: path to the folder hosting the symbol files.
    - **Search pattern**: the pattern used to locate the *.pdb* files in the folder you've designated under *Path to symbols folder*.
    - **Index sources**: indicates whether to inject source server information into the PDB files.
    - **Publish symbols**: indicates whether to publish the symbol files.
        - **Symbol server type**: select **File share** to publish your symbols to a file share.
        - **Path to publish symbols**: the file share that will host your symbols.
    - **Verbose logging**: check to include more information in your logs.

Publish portable PDBs to Azure Artifacts symbol server

Portable PDBs are symbol files that can be created and used on all platforms unlike the traditional PDBs which are used on Windows only. For portable PDBs, the build does the indexing, but you still need to use the **Index Sources and Publish Symbols** task to publish your symbols.

Use Source Link in .NET projects

Source Link is a set of tools that allow developers to debug their source code by mapping from the .NET assemblies back to the source code. Check out the dotnet/sourcelink GitHub repository to learn about the different packages included.

- For projects hosted on GitHub, add the `Microsoft.SourceLink.GitHub` package reference to your project file.

  XML

  ```xml
  <ItemGroup>
    <PackageReference Include="Microsoft.SourceLink.GitHub"
  Version="1.1.1" PrivateAssets="All"/>
  </ItemGroup>
  ```

- For projects hosted on Azure Repos (former Visual Studio Team Services), add the `Microsoft.SourceLink.AzureRepos.Git` package reference to your project file.

  XML

  ```xml
  <ItemGroup>
    <PackageReference Include="Microsoft.SourceLink.AzureRepos.Git"
  Version="1.1.1" PrivateAssets="All"/>
  </ItemGroup>
  ```

- For projects hosted on Azure DevOps Server (former Team Foundation Server), add the `Microsoft.SourceLink.AzureDevOpsServer.Git` package reference to your project file.

  XML

  ```xml
  <ItemGroup>
    <PackageReference
  Include="Microsoft.SourceLink.AzureDevOpsServer.Git" Version="1.1.1"
  ```

```
PrivateAssets="All"/>
</ItemGroup>
```

Set up the publish task

The Index Sources & Publish Symbols task is used to index your source code and publish your symbols to Azure Artifacts symbols server and file shares. Because we're using **Source Link**, we'll have to **disable indexing** in the publish task.

Classic

1. Sign in to your Azure DevOps organization, and then navigate to your project.
2. Select **Pipelines**, select your pipeline, and then select **Edit** to modify your pipeline.
3. From your pipeline definition, select `+` to add a new task.
4. Search for the **Index sources and publish symbols** task. Select **Add** to add it to your pipeline.
5. Fill out the required fields and select *Symbol Server* for the **Symbol server type**. Make sure you uncheck **Index sources** to disable indexing.

> **Important**

To delete symbols published via the *Index Sources & Publish Symbols* task, you must first delete the build that generated those symbols. This can be accomplished by using **retention policies** or by manually **deleting the run**.

Set up Visual Studio

> **Note**
>
> Visual Studio for Mac does not support debugging using symbol servers.

Before starting to consume our symbols from Azure Artifacts symbol server, let's make sure that Visual Studio is set up properly:

1. In Visual Studio, select **Tools** then **Options**.
2. Select **Symbols** from the **Debugging** menu.
3. Select the `+` sign to add a new symbol server location.

4. A new dialog box will appear, select your account from the dropdown menu, and then select the organization that you wish to connect to. Select **Connect** when you're done.
5. Select **General** from the same **Debugging** section. Scroll down and check **Enable Source Link support** to enable support for portable PDBs.

> ⓘ **Note**
>
> Checking the **Enable source server support** option allows you to use Source Server in cases where the source code isn't available locally or the symbol file does not match the source code. If you want to enable debugging for third-party source code, deselect the **Enable Just My Code** checkbox.

FAQs

Q: What is the duration for which symbols are retained?

A: A symbol file has the same retention period as the build that generated it. When you delete a build either manually or using retention policies, the symbols that were generated by that build will be deleted as well.

Q: Can I use source indexing on a portable PDB generated from a .NET Core assembly?

A: This is not possible at the moment. Source indexing is not currently supported for portable PDBs. The recommended approach is to configure your build to do the indexing.

Related articles

- Debug with Visual Studio.
- Debug with WinDbg.
- Configure retention policies.

Restore NuGet packages with Azure Pipelines

Article • 03/10/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

With NuGet Package Restore you can install all your project's dependency without having to store them in source control. This allows for a cleaner development environment and a smaller repository size. You can restore your NuGet packages using the NuGet restore task, the NuGet CLI, or the .NET Core CLI. This article will show you how to restore your NuGet packages using both Classic and YAML Pipelines.

Prerequisites

- An Azure DevOps organization. Create an organization, if you don't have one already.
- An Azure DevOps project. If you don't have one yet, you can create a new project.
- An Azure Artifacts feed. Create a new feed if you don't have one already.
- Connect to Azure Artifacts feed: NuGet.exe, dotnet.
- Set up your pipeline permissions.

Restore NuGet packages from a feed

Classic

1. Navigate to your classic pipeline definition, and then select **Edit**.
2. Select + to add a new task. Search for **NuGet**, and then select **Add** to add the task to your pipeline.
3. Name your task and select **Restore** from the **Command**.
4. Select **Feed(s) I select here**, and select your feed from the dropdown menu. If you want to use your own config file, select **Feeds in my NuGet.config** and enter the path to your NuGet.config file and the service connection if you want to authenticate with feeds outside your organization.

5. If you want to include packages from NuGet.org, check the **Use packages from NuGet.org** checkbox.

6. Select **Save & queue** when you're done.

> ⓘ **Note**
>
> Classic NuGet restore uses the **NuGetCommand@2** task. By default, this version uses NuGet 4.1.0. Use the **NuGet Tool Installer task** if you want to use a different NuGet version.

Restore NuGet packages locally

Place your `nuget.config` in the same folder as your `.csproj` or `.sln` file. Your config file should look similar to the following example:

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
```

```
  <packageSources>
    <!-- remove inherited connection strings -->
    <clear />
    <!-- add an Azure Artifacts feed -->
    <add key="FabrikamFiber"
value="https://pkgs.dev.azure.com/microsoftLearnModule/_packaging/FabrikamFi
ber/nuget/v3/index.json" />
    <!-- Get packages from NuGet.org -->
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json"
protocolVersion="3" />
  </packageSources>
</configuration>
```

To restore your NuGet packages, run the following command in your project directory:

Command

```
nuget.exe restore
```

Restore NuGet packages from a feed in a different organization

To restore NuGet packages from a feed in a different Azure DevOps organization, you must use a personal access token to authenticate.

Create a personal access token

1. Navigate to your Azure DevOps organization, and then select **User settings** > **Personal Access Tokens**.

2. Create a personal access token with **Packaging (read)** scope and copy your PAT to the clipboard.

Restore NuGet packages

Classic

1. Navigate to your pipeline definition and select the **NuGet restore** task. Make sure you're using version 2 of the task.

2. Select **Feeds and authentication**, and then select **Feeds in my NuGet.config**.

3. Select the path of your *NuGet.config* file.

4. Select **New** to add **Credentials for feeds outside this organization/collection**.

5. Select **External Azure DevOps Server**, and then enter your feed URL (make sure it matches what's in your NuGet.config), your service connection name, and the personal access token you created earlier. Select **Save** when you're done.

6. Select **Save & queue** when you're done.

FAQ

Q: My pipeline is failing to restore my NuGet packages?

A: The NuGet restore task can fail for several reasons. The most common scenario is when you add a new project that requires a target framework that is not supported by the NuGet version your pipeline is using. This failure doesn't occur generally in the local

development environment because Visual Studio takes care of updating your solution accordingly. Make sure you upgrade your NuGet task to the latest version.

Related articles

- Publish to NuGet feeds (YAML/Classic)
- Publish and consume build artifacts
- How to mitigate risk when using private package feeds

Publish NuGet packages to NuGet.org with Azure Pipelines

Article • 05/17/2023

Using Azure Pipelines, developers can streamline the process of publishing their NuGet packages to feeds and public registries. In this tutorial, we'll explore how to leverage YAML and classic pipelines to publish NuGet packages to NuGet.org. In this article, you'll learn how to:

- ✓ Authenticate with NuGet.org.
- ✓ Create service connections.
- ✓ Publish packages to NuGet.Org.

Prerequisites

- An Azure DevOps organization and a project. Create one for free, if you don't have one already.
  - The **Administrator** role for service connections in your Azure DevOps project.
- A nuget.org account.

Create an API key

1. Navigate to NuGet.org and sign in to your account.
2. Select your user name icon, and then select **API Keys**.
3. Select **Create**, and then provide a name for your key. Assign the **Push new packages and package version** scope to your key, and enter `*` in the **Glob Pattern** field to include all packages.
4. Select **Create** when you're done.
5. Select **Copy** and save your API key in a safe location.

Create a service connection

1. Sign in to your Azure DevOps organization `https://dev.azure.com/<Your_Organization>` and then navigate to your project.

2. Select ⚙ **Project settings** located at the bottom left corner of the page.

3. Select **NuGet**, and then select **Next**.

4. Select **ApiKey** as your authentication method and set the **Feed URL** to the following: `https://api.nuget.org/v3/index.json`.

5. Enter the ApiKey you generated earlier in the **ApiKey** field, and then provide a name for your service connection.

6. Check the **Grant access permission to all pipelines** checkbox, and then select **Save** when you're done. To select this option, you'll need the service connection **Administrator** role.

Publish packages

1. Sign in to your Azure DevOps organization `https://dev.azure.com/<Your_Organization>` and then navigate to your project.

2. Select **Pipelines**, and then select your pipeline. Select **Edit** to edit your pipeline.

Classic

3. Select + to add a new task, and then search for the **.NET Core** task. select **Add** to add it to your pipeline.

4. Select the **pack** command from the command's dropdown menu, and then select the **Path to csproj or nuspec file(s) to pack**. You can keep the default values for the other fields depending on your scenario.

5. Select + to add a new task, and then search for the **NuGet** task. select **Add** to add it to your pipeline.

6. Select the **push** command from the command's dropdown menu, and then select the **Path to NuGet package(s) to publish**.

7. Select **External NuGet server** for your **Target feed location**. Then, in the **NuGet server** field, select the service connection you created earlier.

Once completed, you can visit the packages page on nuget.org, where you can find your recently published package listed at the top.

Related articles

- Release triggers
- Deploy from multiple branches
- Pipeline caching

About resources for Azure Pipelines

Article • 11/06/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

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

Resources offer the following benefits:

- Ways to share something such as a secure file or password across pipelines.
  - Examples of using resources for sharing are variable groups, secure files, and service connections. In all cases, you're using a resource as a way for a pipeline to access and consume something.
- A tool for enhancing security through access checks and other restrictions.
  - For example, you can limit a service connection to only run on one pipeline. You could also make sure that a repository can only be accessed from a pipeline after a manual approval check.
- Ways to improve traceability for your pipeline and make it easier to troubleshoot environments.
  - For example, you can see the number of the last run that was deployed to an environment.

Share resources across pipelines

Share resources across pipelines by configuring them within the pipelines UI. Then, reference those resources in a task. You can also access some shared resources with the resources YAML pipeline syntax.

Examples of sharing resources with the pipelines UI include secure files, variable groups, and service connections. With the `resources` syntax, examples include accessing pipelines themselves, repositories, and packages.

How a resource gets used in a pipeline depends on the type of pipeline and type of resource.

YAML

For YAML pipelines:

- Service connections and secure files are directly used as inputs to tasks and don't need to be predeclared.

- Variable groups use the `group` syntax.
- Pipelines and repositories use the `resources` syntax.

For example, to use variable groups in a pipeline, add your variables at **Pipelines** > **Library**. Then, you can reference the variable group in your YAML pipeline with the `variables` syntax.

```yml
variables:
- group: my-variable-group
```

To call a second pipeline from your pipeline with the `resources` syntax, reference `pipelines`.

```yml
resources:
  pipelines:
  - pipeline: SmartHotel-resource # identifier for the resource (used in pipeline resource variables)
    source: SmartHotel-CI # name of the pipeline that produces an artifact
```

Use resources to enhance security

You can enhance your pipeline's security with resources by identifying how the resource gets consumed, and how to prevent unauthorized access.

For YAML pipelines only, set resources as protected or open. When a resource is protected, you can apply approvals and checks to limit access to specific users and YAML pipelines. Protected resources include service connections, agent pools, environments, repositories, variable groups, and secure files.

Resource	How is it consumed?	How do you prevent an unintended pipeline from using this?
service connections	Consumed by tasks in a YAML file that use the service connection as an input.	Protected with checks and pipeline permissions. Checks and pipeline permissions are controlled by service connection users. A resource owner can control which pipelines can access a service connection. You can also use pipeline permissions to

Resource	How is it consumed?	How do you prevent an unintended pipeline from using this?
		restrict access to particular YAML pipelines and all classic pipelines.
secret variables in variable groups	A special syntax exists for using variable groups in a pipeline or in a job. A variable group gets added like a service connection.	Protected with checks and pipeline permissions. Checks and pipeline permissions are controlled by variable group users. A resource owner can control which pipelines can access a variable group. You can also use pipeline permissions to restrict access to particular YAML pipelines and all classic pipelines.
secure files	Secure files are consumed by tasks (example: Download Secure File task).	Protected with checks and pipeline permissions. Checks and pipeline permissions are controlled by secure files users. A resource owner can control which pipelines can access secure files. You can also use pipeline permissions to restrict access to particular YAML pipelines and all classic pipelines.
agent pools	There's a special syntax to use an agent pool to run a job.	Protected with checks and pipeline permissions. Checks and pipeline permissions are controlled by agent pool users. A resource owner can control which pipelines can access an agent pool. You can also use pipeline permissions to restrict access to particular YAML pipelines and all classic pipelines.
environments	There's a special syntax to use an environment in a YAML.	Protected with checks and pipeline permissions that are controlled by environment users. You can also use pipeline permissions to restrict access to a particular environment.
repositories	A script can clone a repository if the job access token has access to the repo.	Protected with checks and pipeline permissions controlled by repository contributors. A repository owner can restrict ownership.
artifacts, work items, pipelines	Pipeline artifacts are resources, but Azure Artifacts aren't. A script can download artifacts if the job access token has access to the feed. A pipeline	Artifacts and work items have their own permissions controls. Checks and pipeline permissions for feeds aren't supported.

Resource	How is it consumed?	How do you prevent an unintended pipeline from using this?
	artifact can be declared as a resource in the resources section – primarily for the intent of triggering the pipeline when a new artifact is available, or to consume that artifact in the pipeline.	
containers, packages, webhooks	These live outside the Azure DevOps ecosystem and access is controlled with service connections. There's a special syntax for using all three types in YAML pipelines.	Protected with checks and pipeline permissions controlled by service connection users.

Use resources for traceability

YAML

Environments support the following resource types:

- Kubernetes
- Virtual machines

Next steps

Add resources to a pipeline

Related articles

- Define variables
- Add and use variable groups
- Use secure files
- Library for Azure Pipelines

Library of assets

Article • 04/05/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

A *library* is a collection of build and release assets for an Azure DevOps project. Assets defined in a library can be used in multiple build and release pipelines of the project. The **Library** tab can be accessed directly in Azure Pipelines.

The library contains two types of assets: variable groups and secure files.

Variable groups are only available to release pipelines in TFS 2017 and earlier. They're available to build and release pipelines in TFS 2018 and in Azure Pipelines. Task groups and service connections are available to build and release pipelines in TFS 2015 and newer, and in Azure Pipelines.

Library security

All assets defined in the **Library** share a common security model. You can control who can define new items in a library, and who can use an existing item. **Roles** are defined for library items, and membership of these roles governs the operations you can perform on those items.

Role for library item	Description
Reader	Can view the item.
User	Can use the item when authoring build or release pipelines. For example, you must be a 'User' for a variable group to use it in a release pipeline.
Administrator	Can also manage membership of all other roles for the item. The user who created an item gets automatically added to the Administrator role for that item. By default, the following groups get added to the Administrator role of the library: Build Administrators, Release Administrators, and Project Administrators.
Creator	Can create new items in the library, but this role doesn't include Reader or User permissions. The Creator role can't manage permissions for other users.

The security settings for the **Library** tab control access for *all* items in the library. Role memberships for individual items get automatically inherited from the roles of the Library node.

For more information on pipeline security roles, see About pipeline security roles.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and feel free to post your questions, search for answers, or suggest a feature on our Azure DevOps Developer Community. Support page.

Related articles

- Create and target an environment
- Manage service connections
- Add and use variable groups
- Resources in YAML
- Agents and agent pools

Add an admin role to a protected resource

Article • 08/30/2023

Azure Pipelines requires specific permissions to a protected resource and to open access to all pipelines. For all protected resource types other than Environments, you'll need the administrator role to create a protected resource or edit its **Security** configuration. For Environments, you'll need the **Creator** role. When you don't appropriate access, the **Grant access permission to all pipelines** option is disabled.

Protected resources include:

- Agent pools
- Secret variables in variable groups
- Secure files
- Service connections
- Environments
- Repositories

For repository resources, see protect a repository resource.

Prerequisites

You must be a member of the Project Administrators group to update resource permissions.

Agent pools

You can add the **Administrator** role for a specific agent pool and for all agent pools.

To add the **Administrator** role for all agent pools:

1. Go to **Project Settings** > **Pipelines** > **Agent pools**.

2. Select **Security**.

3. Assign the **Administrator** role in the **Role** column.

To add the **Administrator** role to a specific agent pool:

1. Go to **Project Settings** > **Pipelines** > **Agent pools**.

2. Select a specific agent pool.

3. Select **Security**.

4. In **User permissions**, assign the **Administrator** role in the **Role** column.

Library resources (variable groups and secure files)

1. Go to **Pipelines** > **Library**.

2. Select **Security**.

3. Assign the **Administrator** role in the **Role** column.

Service connections

1. Go to **Project Settings** > **Service connections**.

2. Select a service connection.

3. Go to ⋮ and select **Security**.

4. Assign the **Administrator** role.

Environments

1. Go to **Pipelines** > **Environments**.

2. Select an environment.

3. Go to ⋮ and select **Security**.

4. Assign the **Administrator** role in the **Role** column.

Next steps

Learn more about permissions in Azure DevOps.

Protect a repository resource

Article • 06/16/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

You can add protection to your repository resource with checks and pipeline permissions. When you add protection, you're better able to restrict repository ownership and editing privileges.

Prerequisites

You must be a member of the **Project Administrators** group or have your **Manage permissions** set to **Allow** for Git repositories.

Add a repository resource check

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`) and choose your project.
2. Select **Project settings** > **Repos**.

3. Choose the repository that you want to modify.

4. Select ⋮ > **Approvals and checks**.

5. Choose a check to set how your repository resource can be used, and then select **Next**. In the following example, we choose to add Approvals, so a manual approver for each time a pipeline requests the repository. For more information, see Approvals and checks.

6. Configure the check in the resulting screen, and then select **Create**.

Your repository has a resource check.

Add pipeline permissions to a repository resource

You can also set a repository to only be used on specific YAML pipelines. Restricting a repository to specific pipelines prevents an unauthorized YAML pipeline in your project from using your repository. This setting only applies to YAML pipelines.

> **Important**
>
> Access to all pipelines is turned *off* for **protected resources** by default. To grant access to all pipelines, enter a check in the security box next to "Grant access permission to all pipelines" for the resource. You can do so when you're creating or editing a resource. You'll need to have the repository **Administrator** role to have this option available.

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`) and choose your project.

2. Select **Project settings** > **Repositories**.

3. Choose the repository that you want to modify.

4. Select **Security**.

5. Go to **Pipeline permissions**.

6. Select + .

7. Choose the repository to add.

You can see the added repository listed.

Next steps

Add and use variable groups

Related articles

- Set Git repository permissions
- Git repository settings and policies
- Azure Pipelines resources in YAML

Manage service connections

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

You can create a connection from Azure Pipelines to external and remote services for executing tasks in a job. Once you establish a connection, you can view, edit, and add security to the service connection.

For example, you might want to connect to one of the following categories and their services.

- Your Microsoft Azure subscription: Create a service connection with your Microsoft Azure subscription and use the name of the service connection in an Azure Web Site Deployment task in a release pipeline.
- A different build server or file server: Create a standard GitHub Enterprise Server service connection to a GitHub repository.
- An online continuous integration environment: Create a Jenkins service connection for continuous integration of Git repositories.
- Services installed on remote computers: Create an Azure Resource Manager service connection to a VM with a managed service identity.

> **Tip**
>
> The Azure Resource Manager service connection is a single example of a service connection. For more information, see **Common service connection types**.

Prerequisites

You can create, view, use, and manage a service connection based on your assigned user roles. For more information, see User permissions.

Create a service connection

Complete the following steps to create a service connection for Azure Pipelines.

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`) and select your project.

2. Select **Project settings** > **Service connections**.

3. Select + **New service connection**, select the type of service connection that you need, and then select **Next**.

4. Choose an authentication method, and then select **Next**.

5. Enter the parameters for the service connection. The list of parameters differs for each type of service connection. For more information, see the list of service connection types and associated parameters.

6. Select **Save** to create the connection.

7. Validate the connection, once it's created and parameters are entered. The validation link uses a REST call to the external service with the information that you

entered, and indicates whether the call succeeded.

> **Note**
>
> The new service connection window may appear different for the various types of service connections and have different parameters. See the list of parameters in **Common service connection types** for each service connection type.

Edit service connection

Complete the following steps to edit a service connection.

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`) and select your project.
2. Select **Project settings** > **Service connections**.
3. Select the service connection that you want to edit.
4. See the **Overview** tab of the service connection where you can see the details of the service connection. For example, you can see details like type, creator, and authentication type. For instance, token, username/password, or OAuth, and so on.

5. Next to the overview tab, you can see **Usage history**, which shows the list of pipelines that are using the service connection.

6. To update the service connection, select **Edit**. **Approvals and checks**, **Security**, and **Delete** are part of the more options at the top-right corner.

Secure a service connection

Complete the following steps to manage security for a service connection.

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`) and select your project.
2. Select **Project settings** > **Service connections**.
3. Highlight the service connection that you want to edit.
4. Go to the more options at the top-right corner and choose **Security**.

Based on usage patterns, service connection security is divided into the following categories. Edit permissions as desired.

- User permissions
- Pipeline permissions
- Project permissions

User permissions

Control who can create, view, use, and manage a service connection with user roles. In **Project settings** > **Service connections**, you can set the hub-level permissions, which are inherited. You can also override the roles for each service connection.

Role on a service connection	Purpose
Creator	Members of this role can create the service connection in the project. Contributors are added as members by default
Reader	Members of this role can view the service connection.
User	Members of this role can use the service connection when authoring build or release pipelines or authorize yaml pipelines.

Role on a service connection	Purpose
Administrator	Members of this role can use the service connection and manage membership of all other roles for the project's service connection. Project Administrators are added as members by default.

> **Note**
>
> Previously, Endpoint Creator and Endpoint Administrator groups were used to control who could create and manage service connections. We've moved to the pure RBAC model, which uses roles. For backward compatibility in existing projects, the Endpoint Administrator group is now the Administrator role and the Endpoint Creator group is the Creator role, which ensures there's no change in the behavior for existing service connections.
>
> We've also introduced **Sharing of service connections across projects**. With this feature, service connections now become an organization-level object, however scoped to your current project by default. In **User permissions**, you can see **Project**- and **Organization**- level permissions. The functionality of the Administrator role is split between the two permission levels.

Project-level permissions

The project-level permissions are the user permissions with reader, user, creator and administrator roles, as explained above, within the project scope. You have inheritance and you can set the roles at the hub level and for each service connection.

The project-level Administrator can do the following tasks:

- Manage other users and roles at the project-level
- Rename a service connection and update the description
- Delete a service connection, which removes it from the project

The user who created the service connection is automatically added to the Administrator role for that service connection. Users and groups assigned the Administrator role at the hub-level are inherited if the inheritance is turned on.

Organization-level permissions

Any permissions set at the organization-level reflect across all the projects where the service connection is shared. There's no inheritance for organization-level permissions.

The organization-level Administrator can do the following administrative tasks:

- Manage organization-level users
- Edit all the fields of a service connection
- Share and unshare a service connection with other projects

The user who created the service connection is automatically added as an organization-level Administrator role for that service connection. In all existing service connections, the connection Administrators are made organization-level Administrators.

Pipeline permissions

Pipeline permissions control which YAML pipelines are authorized to use the service connection. Pipeline permissions do not restrict access from Classic pipelines.

You can choose from the following options:

- Open access for all pipelines to consume the service connection from the more options at top-right corner of the **Pipeline permissions** section in security tab of a service connection.
- Lock down the service connection and only allow selected YAML pipelines to consume the service connection. If any other YAML pipeline refers to the service connection, an authorization request gets raised, which must be approved by a connection Administrator. This does not limit access from Classic pipelines.

Project permissions - Cross project sharing of service connections

Project permissions control which projects can use the service connection. By default, service connections aren't shared with any other projects.

- Only the organization-level administrators from **User permissions** can share the service connection with other projects.
- The user who's sharing the service connection with a project should have at least Create service connection permission in the target project.
- The user who shares the service connection with a project becomes the project-level Administrator for that service connection. The project-level inheritance is set to *on* in the target project.
- The service connection name is appended with the project name and it can be renamed in the target project scope.
- Organization-level administrator can unshare a service connection from any shared project.

> **Note**
>
> The project permissions feature is dependent on the new service connections UI. When we enable this feature, the old service connections UI will no longer be usable.

Use a service connection

Once you've created your service connection, complete the following steps to use it.

YAML

1. Copy the connection name into your code as the **azureSubscription** (or the equivalent connection name) value.

1. Authorize the service connection using one of the following techniques:

- To authorize any pipeline to use the service connection, go to Azure Pipelines, open the Settings page, select Service connections, and enable the setting **Allow all pipelines to use this connection** option for the connection.
- To authorize a service connection for a specific pipeline, open the pipeline by selecting **Edit** and queue a build manually. You see a resource authorization error and an "Authorize resources" action on the error. Choose this action to explicitly add the pipeline as an authorized user of the service connection.

You can also create your own custom service connections.

> **Note**
>
> Service connections can't be specified by variables.

Common service connection types

Azure Pipelines supports the following service connection types by default:

Azure Classic | Azure Repos/TFS | Azure Resource Manager | Azure Service Bus | Bitbucket | Chef | Docker hub or others | Other Git | Generic | GitHub | GitHub Enterprise Server | Jenkins | Kubernetes | Maven | npm | NuGet | Python package download | Python package upload | Service Fabric | SSH | Subversion | Visual Studio App Center |

Azure Classic service connection

Use the following parameters to define and secure a connection to a Microsoft Azure subscription, using Azure credentials or an Azure management certificate.

How do I create a new service connection?

Parameter	Description
[authentication type]	Required. Select **Credentials** or **Certificate based**.
Connection name	Required. The name you use to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Environment	Required. Select **Azure Cloud**, **Azure Stack**, or one of the predefined Azure Government Clouds where your subscription is defined.

Parameter	Description
Subscription ID	Required. The GUID-like identifier for your Azure subscription (not the subscription name). You can copy the subscription ID from the Azure portal.
Subscription name	Required. The name of your Microsoft Azure subscription.
User name	Required for Credentials authentication. User name of a work or school account (for example @fabrikam.com). Microsoft accounts (for example @live or @hotmail) are't supported.
Password	Required for Credentials authentication. Password for the user specified above.
Management certificate	Required for Certificate-based authentication. Copy the value of the management certificate key from your publish settings XML file or the Azure portal.

If your subscription is defined in an Azure Government Cloud, ensure your application meets the relevant compliance requirements before you configure a service connection.

Azure Repos

Use the following parameters to define and secure a connection to another Azure DevOps organization.

Parameter	Description
(authentication)	Select **Basic** or **Token Based** authentication.
Connection name	Required. The name you use to refer to the service connection in task properties. This isn't the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Connection URL	Required. The URL of the TFS or the other Azure DevOps organization.
User name	Required for Basic authentication. The username to connect to the service.
Password	Required for Basic authentication. The password for the specified username.
Personal Access Token	Required for Token Based authentication (TFS 2017 and newer and Azure Pipelines only). The token to use to authenticate with the service. Learn more.

Use the **Verify connection** link to validate your connection information.

For more information, see Authenticate access with personal access tokens for Azure DevOps.

Azure Resource Manager service connection

Use the following parameters to define and secure a connection to a Microsoft Azure subscription using Service Principal Authentication (SPA) or an Azure managed Service Identity. The dialog offers two main modes:

- **Automated subscription detection**. In this mode, Azure Pipelines queries Azure for all of the subscriptions and instances to which you have access. They use the credentials you're currently signed in with in Azure Pipelines (including Microsoft accounts and School or Work accounts).

If you don't see the subscription you want to use, sign out of Azure Pipelines and sign in again using the appropriate account credentials.

- **Manual subscription pipeline**. In this mode, you must specify the service principal you want to use to connect to Azure. The service principal specifies the resources and the access levels that are available over the connection.

Use this approach when you need to connect to an Azure account using different credentials from the credentials you're currently signed in with in Azure Pipelines. It's useful way to maximize security and limit access. Service principals are valid for two years.

For more information, see Connect to Microsoft Azure

> ⓘ **Note**
>
> If you don't see any Azure subscriptions or instances, or you have problems validating the connection, see **Troubleshoot Azure Resource Manager service connections**.

Azure Service Bus service connection

Use the following parameters to define and secure a connection to a Microsoft Azure Service Bus queue.

Parameter	Description

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task properties. This name isn't the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Service Bus ConnectionString	The URL of your Azure Service Bus instance. More information.
Service Bus Queue Name	The name of an existing Azure Service Bus queue.

Bitbucket Cloud service connection

Use OAuth with **Grant authorization** or a username and password with **Basic Authentication** to define a connection to Bitbucket Cloud. For pipelines to keep working, your repository access must remain active.

Grant authorization

Parameter	Description
OAuth configuration	Required. OAuth connection to Bitbucket.

Basic authentication

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
User name	Required. The username to connect to the service.
Password	Required. The password for the specified username.

Chef service connection

Use the following parameters to define and secure a connection to a Chef automation server.

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Server URL	Required. The URL of the Chef automation server.
Node Name (Username)	Required. The name of the node to connect to. Typically this is your username.
Client Key	Required. The key specified in the Chef .pem file.

Docker Host service connection

Use the following parameters to define and secure a connection to a Docker host.

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Server URL	Required. The URL of the Docker host.
CA certificate	Required. A trusted certificate authority certificate to use to authenticate with the host.
Certificate	Required. A client certificate to use to authenticate with the host.
Key	Required. The key specified in the Docker key.pem file.

For more information about protecting your connection to the Docker host, see Protect the Docker daemon socket.

Docker Registry service connection

Use the following parameters to define a connection to a container registry for either Azure Container Registry or Docker Hub or others.

Azure Container Registry

Parameter	Description

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task inputs.
Azure subscription	Required. The Azure subscription containing the container registry to be used for service connection creation.
Azure Container Registry	Required. The Azure Container Registry to be used for creation of service connection.

Docker Hub or others

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task inputs.
Docker Registry	Required. The URL of the Docker registry.
Docker ID	Required. The identifier of the Docker account user.
Password	Required. The password for the account user identified above. (Docker Hub requires a PAT instead of a password.)
Email	Optional. An email address to receive notifications.

Other Git service connection

Use the following parameters to define and secure a connection to an external Git repository server. There's a specific service connection for GitHub and GitHub Enterprise Server.

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Server URL	Required. The URL of the Git repository server.

Parameter	Description
Attempt accessing this Git server from Azure Pipelines	When checked, Azure Pipelines attempts to connect to the repository before queuing a pipeline run. You can disable this setting to improve performance when working with repositories that are not publicly accessible. Note that CI triggers will not work in when an **Other Git** repository is not publicly accessible. You can only start manual or scheduled pipeline runs.
User name	Required. The username to connect to the Git repository server.
Password/Token key	Required. The password or access token for the specified username.

For more information, see Artifact sources.

Generic service connection

Use the following parameters to define and secure a connection to any generic type of service or application.

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Server URL	Required. The URL of the service.
User name	Optional. The username to connect to the service.
Password/Token key	Optional. The password or access token for the specified username.

GitHub service connection

Use the following parameters to define a connection to a GitHub repository.

> **Tip**
>
> There's a specific service connection for **Other Git servers** and **GitHub Enterprise Server connections**.

Parameter	Description

Parameter	Description
Choose authorization	Required. Either **Grant authorization** or **Personal access token**. See notes below.
Token	Required for Personal access token authorization. See notes below.
Connection name	Required. The name you use to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.

> **Note**
>
> If you select **Grant authorization** for the **Choose authorization** option, the dialog shows an **Authorize** button that opens the GitHub signing page. If you select **Personal access token**, paste it into the **Token** textbox. The dialog shows the recommended scopes for the token: **repo, user, admin:repo_hook**. For more information, see **Create an access token for command line use** Then, complete the following steps to register your GitHub account in your profile.

1. Open your **User settings** from your account name at the right of the Azure Pipelines page heading.
2. Choose **Personal access tokens**.
3. Select **Add** and enter the information required to create the token.

For more information, see Artifact sources - version control.

GitHub Enterprise Server service connection

Use the following parameters to define a connection to a GitHub repository.

> **Tip**
>
> There's a specific service connection for **Other Git servers** and **standard GitHub service connections**.

Parameter	Description
Choose authorization	Required. Either **Personal access token**, **Username and Password**, or **OAuth2**. See notes below.

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task properties. This isn't the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Server URL	Required. The URL of the service.
Accept untrusted TLS/SSL certificates	Set this option to allow clients to accept a self-signed certificate instead of installing the certificate in the TFS service role or the computers hosting the agent.
Token	Required for Personal access token authorization. See notes below.
User name	Required for Username and Password authentication. The username to connect to the service.
Password	Required for Username and Password authentication. The password for the specified username.
OAuth configuration	Required for OAuth2 authorization. The OAuth configuration specified in your account.
GitHub Enterprise Server configuration URL	The URL is fetched from OAuth configuration.

> **Note**
>
> If you select **Personal access token** (PAT) you must paste the PAT into the **Token** textbox. The dialog shows the recommended scopes for the token: **repo, user, admin:repo_hook**. For more information, see **Create an access token for command line use** Then, complete the following steps to register your GitHub account in your profile.

1. Open your **User settings** from your account name at the right of the Azure Pipelines page heading.
2. Choose **Personal access tokens**.
3. Select **Add** and enter the information required to create the token.

Jenkins service connection

Use the following parameters to define a connection to the Jenkins service.

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Server URL	Required. The URL of the service.
Accept untrusted TLS/SSL certificates	Set this option to allow clients to accept a self-signed certificate instead of installing the certificate in the TFS service role or the computers hosting the agent.
User name	Required. The username to connect to the service.
Password	Required. The password for the specified username.

For more information, see Azure Pipelines Integration with Jenkins and Artifact sources - Jenkins.

Kubernetes service connection

Use the following parameters when you define a connection to a Kubernetes cluster. Choose from the following service connection options:

- Azure subscription
- Service account
- Kubeconfig

Azure subscription option

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task inputs.
Azure subscription	Required. The Azure subscription containing the cluster to be used for service connection creation.
Cluster	Name of the Azure Kubernetes Service cluster.
Namespace	Namespace within the cluster.

For an Azure RBAC enabled cluster, a ServiceAccount gets created in the chosen namespace along with RoleBinding object, so that the created ServiceAccount can do actions only on the chosen namespace.

For an Azure RBAC disabled cluster, a ServiceAccount gets created in the chosen namespace, but, the created ServiceAccount has cluster-wide privileges (across namespaces).

> **Note**
>
> This option lists all the subscriptions the service connection creator has access to *across different Azure tenants*. If you can't see subscriptions from other Azure tenants, check your Azure AD permissions in those tenants.

Service account option

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task inputs.
Server URL	Required. Cluster's API server URL.
Secret	Secret associated with the service account to be used for deployment

Use the following command to fetch the Server URL.

```
kubectl config view --minify -o 'jsonpath={.clusters[0].cluster.server}'
```

Use the following sequence of commands to fetch the Secret object that's required to connect and authenticate with the cluster.

```
kubectl get serviceAccounts <service-account-name> -n <namespace> -o
'jsonpath={.secrets[*].name}'
```

In the following command, replace the `service-account-secret-name` with the output of the previous command.

```
kubectl get secret <service-account-secret-name> -n <namespace> -o json
```

Copy and paste the Secret object fetched in YAML form into the Secret text-field.

> **Note**
>
> When using the service account option, **ensure that a RoleBinding exists**, which grants permissions in the `edit` `ClusterRole` to the desired service account. This is needed so that the service account can be used by Azure Pipelines for creating objects in the chosen namespace.

Kubeconfig option

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task inputs.
Kubeconfig	Required. Contents of the kubeconfig file.
Context	Context within the kubeconfig file that is to be used for identifying the cluster.

Maven service connection

Use the following parameters when you define and secure a connection to a Maven repository.

Parameter	Description
Connection name	Required. The name you use to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Registry URL	Required. The URL of the Maven repository.
Registry ID	Required. This is the ID of the server that matches the ID element of the repository/mirror that Maven tries to connect to.
Username	Required when connection type is **Username and Password**. The username for authentication.

Parameter	Description
Password	Required when connection type is **Username and Password**. The password for the username.
Personal Access Token	Required when connection type is **Authentication Token**. The token to use to authenticate with the service. Learn more.

npm service connection

Use the following parameters when you define and secure a connection to an npm server.

Parameter	Description
Connection name	Required. The name used to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Registry URL	Required. The URL of the npm server.
Username	Required when connection type is **Username and Password**. The username for authentication.
Password	Required when connection type is **Username and Password**. The password for the username.
Personal Access Token	Required when connection type is **External Azure Pipelines**. The token to use to authenticate with the service. Learn more.

NuGet service connection

Use the following parameters when you define and secure a connection to a NuGet server.

Parameter	Description
Connection name	Required. The name used to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Feed URL	Required. The URL of the NuGet server.

Parameter	Description
ApiKey	Required when connection type is **ApiKey**. The authentication key.
Personal Access Token	Required when connection type is **External Azure Pipelines**. The token to use to authenticate with the service. Learn more.
Username	Required when connection type is **Basic authentication**. The username for authentication.
Password	Required when connection type is **Basic authentication**. The password for the username.

To configure NuGet to authenticate with Azure Artifacts and other NuGet repositories, see NuGet Authenticate.

Python package download service connection

Use the following parameters when you define and secure a connection to a Python repository for downloading Python packages.

Parameter	Description
Connection name	Required. The name used to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Python repository url for download	Required. The URL of the Python repository.
Personal Access Token	Required when connection type is **Authentication Token**. The token to use to authenticate with the service. Learn more.
Username	Required when connection type is **Username and Password**. The username for authentication.
Password	Required when connection type is **Username and Password**. The password for the username.

Python package upload service connection

Use the following parameters when you define and secure a connection to a Python repository for uploading Python packages.

Parameter	Description
Connection name	Required. The name used to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Python repository url for upload	Required. The URL of the Python repository.
EndpointName	Required. Unique repository name used for twine upload. Spaces and special characters aren't allowed.
Personal Access Token	Required when connection type is **Authentication Token**. The token to use to authenticate with the service. Learn more.
Username	Required when connection type is **Username and Password**. The username for authentication.
Password	Required when connection type is **Username and Password**. The password for the username.

Service Fabric service connection

Use the following parameters when you define and secure a connection to a Service Fabric cluster.

Parameter	Description
Connection name	Required. The name used to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Cluster Endpoint	Required. The TCP endpoint of the cluster.
Server Certificate Thumbprint	Required when connection type is **Certificate based** or **Azure Active Directory**.
Client Certificate	Required when connection type is **Certificate based**.
Password	Required when connection type is **Certificate based**. The certificate password.

Parameter	Description
Username	Required when connection type is **Azure Active Directory**. The username for authentication.
Password	Required when connection type is **Azure Active Directory**. The password for the username.
Use Windows security	Required when connection type is **Others**.
Cluster SPN	Required when connection type is **Others** and using Windows security.

SSH service connection

Use the following parameters when you define and secure a connection to a remote host using Secure Shell (SSH).

Parameter	Description
Connection name	Required. The name used to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Host name	Required. The name of the remote host machine or the IP address.
Port number	Required. The port number of the remote host machine to which you want to connect. The default is port 22.
User name	Required. The username to use when connecting to the remote host machine.
Password or passphrase	The password or passphrase for the specified username if using a keypair as credentials.
Private key	The entire contents of the private key file if using this type of authentication.

For more information, see SSH task and Copy files over SSH.

Subversion service connection

Use the following parameters when you define and secure a connection to the Subversion repository.

Parameter	Description

Parameter	Description
Connection name	Required. The name used to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
Server repository URL	Required. The URL of the repository.
Accept untrusted TLS/SSL certificates	Set this option to allow the client to accept self-signed certificates installed on the agent computer(s).
Realm name	Optional. If you use multiple credentials in a build or release pipeline, use this parameter to specify the realm containing the credentials specified for the service connection.
User name	Required. The username to connect to the service.
Password	Required. The password for the specified username.

Visual Studio App Center service connection

Use the following parameters when you define and secure a connection to Visual Studio App Center.

Parameter	Description
Connection name	Required. The name used to refer to the service connection in task properties. It's not the name of your Azure account or subscription. If you're using YAML, use the name as the **azureSubscription** or the equivalent subscription name value in the script.
API token	Required. The token to use to authenticate with the service. For more information, see the API docs.

Extensions for other service connections

Other service connection types and tasks can be installed as extensions. See the following examples of service connections available through extensions:

- TFS artifacts for Azure Pipelines. Deploy on-premises TFS builds with Azure Pipelines through a TFS service connection and the **Team Build (external)** artifact,

even when the TFS machine isn't reachable directly from Azure Pipelines. For more information, see External TFS and this blog post.

- TeamCity artifacts for Azure Pipelines. This extension provides integration with TeamCity through a TeamCity service connection, enabling artifacts produced in TeamCity to be deployed by using Azure Pipelines. For more information, see TeamCity.

- System Center Virtual Machine Manager (SCVMM) Integration. Connect to an SCVMM server to provision virtual machines and do actions on them such as managing checkpoints, starting and stopping virtual machines (VMs), and running PowerShell scripts.

- VMware Resource Deployment. Connect to a VMware vCenter Server from Visual Studio Team Services or TFS to provision. Start, stop, or snapshot VMware virtual machines.

- Power Platform Build Tools. Use Microsoft Power Platform Build Tools to automate common build and deployment tasks related to apps built on Microsoft Power Platform. After installing the extension, the **Power Platform** service connection type has the following properties.

Parameter	Description
Connection Name	Required. The name you will use to refer to this service connection in task properties.
Server URL	Required. The URL of the Power Platform instance. Example: `https://contoso.crm4.dynamics.com`
Tenant ID	Required. Tenant ID (also called directory ID in Azure portal) to authenticate to. Refer to https://aka.ms/buildtools-spn for a script that shows Tenant ID and configures Application ID and associated Client Secret. The application user must also be created in CDS
Application ID	Required. Azure Application ID to authenticate with.
Client secret of Application ID	Required. Client secret of the Service Principal associated to above Application ID used to prove identity.

You can also create your own custom service connections.

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and feel free to post your questions, search for answers, or suggest a feature on our Azure DevOps Developer Community. Support page.

Define resources in YAML

Article • 04/14/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

Resources in YAML represent sources of pipelines, builds, repositories, containers, packages, and webhooks. Resources also provide you the full traceability of the services used in your pipeline including the version, artifacts, associated commits, and work items. When you define a resource, it can be consumed anywhere in your pipeline. And, you can fully automate your DevOps workflow by subscribing to trigger events on your resources.

For more information, see About resources.

Schema

YAML

```yaml
resources:
  pipelines: [ pipeline ]
  builds: [ build ]
  repositories: [ repository ]
  containers: [ container ]
  packages: [ package ]
  webhooks: [ webhook ]
```

Variables

When a resource triggers a pipeline, the following variables get set:

YAML

```yaml
resources.triggeringAlias
resources.triggeringCategory
```

These values are empty if a resource doesn't trigger a pipeline run. The variable `Build.Reason` must be `ResourceTrigger` for these values to get set.

Define a `pipelines` resource

If you have a pipeline that produces artifacts, you can consume the artifacts by defining a `pipelines` resource. `pipelines` is a dedicated resource only for Azure Pipelines. You can also set triggers on a pipeline resource for your CD workflows.

In your resource definition, `pipeline` is a unique value that you can use to reference the pipeline resource later on. `source` is the name of the pipeline that produces an artifact. Use the label defined by `pipeline` to reference the pipeline resource from other parts of the pipeline, such as when using pipeline resource variables or downloading artifacts.

For an alternative way to download pipelines, see the tasks in Pipeline Artifacts.

Schema

YAML

```yaml
resources:        # types: pipelines | builds | repositories |
containers | packages
  pipelines:
  - pipeline: string  # identifier for the resource used in pipeline
resource variables
    project: string # project for the source; optional for current
project
    source: string  # name of the pipeline that produces an artifact
    version: string  # the pipeline run number to pick the artifact,
defaults to latest pipeline successful across all stages; Used only for
manual or scheduled triggers
    branch: string  # branch to pick the artifact, optional; defaults to
all branches; Used only for manual or scheduled triggers
    tags: [ string ] # list of tags required on the pipeline to pickup
default artifacts, optional; Used only for manual or scheduled triggers
    trigger:     # triggers aren't enabled by default unless you add
trigger section to the resource
      branches:  # branch conditions to filter the events, optional;
Defaults to all branches.
        include: [ string ]  # branches to consider the trigger events,
optional; Defaults to all branches.
        exclude: [ string ]  # branches to discard the trigger events,
optional; Defaults to none.
      tags: [ string ]  # list of tags to evaluate for trigger event,
optional
      stages: [ string ] # list of stages to evaluate for trigger event,
optional
```

Important

When you define a resource trigger, if its pipeline resource is from the same repository (say self) as the current pipeline, triggering follows the same branch and

commit on which the event is raised. But, if the pipeline resource is from a different repository, the current pipeline triggers on the default branch of the self repository.

Evaluation of artifact version

The version of the resource pipeline's artifacts depends on how your pipeline is triggered.

If your pipeline runs because you manually triggered it or due to a scheduled run, the version of artifacts's version is defined by the values of the `version`, `branch`, and `tags` properties.

Specified properties	Artifact version
`version`	The artifacts from the build having the specified run number
`branch`	The artifacts from the latest build performed on the specified branch
`tags` list	The artifacts from the latest build that has all the specified tags
`branch` and `tags` list	The artifacts from the latest build performed on the specified branch *and* that has all the specified tags
None	The artifacts from the latest build across all the branches

Let's look at an example. Say your pipeline contains the following resource definition.

yml

```yml
resources:
  pipelines:
  - pipeline: MyCIAlias
    project: Fabrikam
    source: Farbrikam-CI
    branch: main      ### This branch input cannot have wild cards. It is
used for evaluating default version when pipeline is triggered manually or
scheduled.
    tags:               ### These tags are used for resolving default
version when the pipeline is triggered manually or scheduled
    - Production        ### Tags are AND'ed
    - PreProduction
```

When you manually trigger your pipeline to run, the version of the artifacts of the `MyCIAlias` pipeline is the one of the latest build done on the `main` branch and that has the `Production` and `PrepProduction` tags.

When your pipeline gets triggered because one of its resource pipelines completes, the version of the artifacts is the one of the triggering pipeline. The values of the `version`, `branch`, and `tags` properties are ignored.

Specified triggers	Outcome
`branches`	A new run of the current pipeline is triggered whenever the resource pipeline successfully completes a run on the `include` branches
`tags`	A new run of the current pipeline is triggered whenever the resource pipeline successfully completes a run that is tagged with *all* the specified tags
`stages`	A new run of the current pipeline is triggered whenever the resource pipeline successfully executed the specified `stages`
`branches`, `tags`, and `stages`	A new run of the current pipeline is triggered whenever the resource pipeline run satisfies *all* branch, tags, and stages conditions
`trigger: true`	A new run of the current pipeline is triggered whenever the resource pipeline successfully completes a run
Nothing	No new run of the current pipeline is triggered when the resource pipeline successfully completes a run

Let's look at an example. Say your pipeline contains the following resource definition.

YAML

```yaml
resources:
  pipelines:
  - pipeline: SmartHotel
    project: DevOpsProject
    source: SmartHotel-CI
    trigger:
      branches:
        include:
        - releases/*
        - main
        exclude:
        - topic/*
      tags:
      - Verified
      - Signed
      stages:
      - Production
      - PreProduction
```

Your pipeline will run whenever the `SmartHotel-CI` pipelines runs on one of the `releases` branches or on the `main` branch, is tagged with both `Verified` and `Signed`, and it completed both the `Production` and `PreProduction` stages.

`download` for pipelines

All artifacts from the current pipeline and from all `pipeline` resources are automatically downloaded and made available at the beginning of each `deployment` job. You can override this behavior. For more information, see Pipeline Artifacts. Regular 'job' artifacts aren't automatically downloaded. Use `download` explicitly when needed.

Schema

YAML

```yaml
steps:
- download: [ current | pipeline resource identifier | none ] # disable
automatic download if "none"
  artifact: string ## artifact name, optional; downloads all the
available artifacts if not specified
  patterns: string # patterns representing files to include; optional
```

Artifacts from the `pipeline` resource get downloaded to `$(PIPELINE.WORKSPACE)/<pipeline-identifier>/<artifact-identifier>` folder.

Pipeline resource variables

In each run, the metadata for a pipeline resource is available to all jobs in the form of predefined variables. The `<Alias>` is the identifier that you gave for your pipeline resource. Pipeline resources variables are only available at runtime.

Schema

YAML

```yaml
resources.pipeline.<Alias>.projectID
resources.pipeline.<Alias>.pipelineName
resources.pipeline.<Alias>.pipelineID
resources.pipeline.<Alias>.runName
resources.pipeline.<Alias>.runID
resources.pipeline.<Alias>.runURI
resources.pipeline.<Alias>.sourceBranch
resources.pipeline.<Alias>.sourceCommit
```

```
resources.pipeline.<Alias>.sourceProvider
resources.pipeline.<Alias>.requestedFor
resources.pipeline.<Alias>.requestedForID
```

For more information, see Pipeline resource metadata as predefined variables.

Define a `builds` resource

If you have an external CI build system that produces artifacts, you can consume artifacts with a `builds` resource. A `builds` resource can be any external CI systems like Jenkins, TeamCity, CircleCI, and so on.

Schema

YAML

```yaml
resources:        # types: pipelines | builds | repositories |
containers | packages
  builds:
  - build: string   # identifier for the build resource
    type: string   # the type of your build service like Jenkins,
circleCI etc.
    connection: string   # service connection for your build service.
    source: string   # source definition of the build
    version: string   # the build number to pick the artifact, defaults
to Latest successful build
    trigger: boolean    # Triggers aren't enabled by default and should
be explicitly set
```

`builds` is an extensible category. You can write an extension to consume artifacts from your builds service and introduce a new type of service as part of `builds`. Jenkins is a type of resource in `builds`.

> **Important**
>
> Triggers are only supported for hosted Jenkins where Azure DevOps has line of sight with Jenkins server.

`downloadBuild` task for builds

You can consume artifacts from the `build` resource as part of your jobs using the `downloadBuild` task. Based on the type of `build` resource defined, this task automatically resolves to the corresponding download task for the service during the run time.

Artifacts from the `build` resource get downloaded to `$(PIPELINE.WORKSPACE)/<build-identifier>/` folder.

> **Important**
>
> `build` resource artifacts aren't automatically downloaded in your jobs/deploy-jobs. You need to explicitly add the `downloadBuild` task for consuming the artifacts.

Schema

YAML

```yaml
- downloadBuild: string # identifier for the resource from which to download artifacts
  artifact: string # artifact to download; if left blank, downloads all artifacts associated with the resource provided
  patterns: string | [ string ] # a minimatch path or list of [minimatch paths](tasks/file-matching-patterns.md) to download; if blank, the entire artifact is downloaded
```

Define a `repositories` resource

If your pipeline has templates in another repository, or if you want to use multi-repo checkout with a repository that requires a service connection, you must let the system know about that repository. The `repository` keyword lets you specify an external repository.

Schema

YAML

```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.
```

Type

Pipelines support the following values for the repository type: `git`, `github`, `githubenterprise`, and `bitbucket`. The `git` type refers to Azure Repos Git repos.

Type specified	Outcome	Example
`type: git`	The `name` value refers to another repository in the same project.	`name: otherRepo` To refer to a repository in another project within the same organization, prefix the name with that project's name. An example is `name: OtherProject/otherRepo`.
`type: github`	The `name` value is the full name of the GitHub repository and includes the user or organization.	`name: Microsoft/vscode`
`type: githubenterprise`	the `name` value is the full name of the GitHub Enterprise repository and includes the user or organization.	`name: Microsoft/vscode`
`type: bitbucket`	The `name` value is the full name of the Bitbucket Cloud repository and includes the user or organization.	`name: MyBitbucket/vscode`

GitHub Enterprise repos require a GitHub Enterprise service connection for authorization.

Bitbucket Cloud repos require a Bitbucket Cloud service connection for authorization.

Use `checkout` to consume repository

Use `checkout` keyword to consume your repos defined as part of `repository` resource.

Schema

YAML

```yaml
steps:
- checkout: string # Required as first property. Configures checkout for the
specified repository.
  clean: string # If true, run git clean -ffdx followed by git reset --hard
HEAD before fetching.
  fetchDepth: string # Depth of Git graph to fetch.
  fetchTags: string # Set to 'true' to sync tags when fetching the repo, or
'false' to not sync tags. See remarks for the default behavior.
  lfs: string # Set to 'true' to download Git-LFS files. Default is not to
download them.
  persistCredentials: string # Set to 'true' to leave the OAuth token in the
Git config after the initial fetch. The default is not to leave it.
  submodules: string # Set to 'true' for a single level of submodules or
'recursive' to get submodules of submodules. Default is not to fetch
submodules.
  path: string # Where to put the repository. The root directory is
$(Pipeline.Workspace).
  condition: string # Evaluate this condition expression to determine
whether to run this task.
  continueOnError: boolean # Continue running even on failure?
  displayName: string # Human-readable name for the task.
  target: string | target # Environment in which to run this task.
  enabled: boolean # Run this task when the job runs?
  env: # Variables to map into the process's environment.
    string: string # Name/value pairs
  name: string # ID of the step.
  timeoutInMinutes: string # Time to wait for this task to complete before
the server kills it.
  retryCountOnTaskFailure: string # Number of retries if the task fails.
```

Repos from the `repository` resource aren't automatically synced in your jobs. Use `checkout` to fetch your repos as part of your jobs.

For more information, see Check out multiple repositories in your pipeline.

Define a `containers` resource

If you need to consume a container image as part of your continuous integration/continuous delivery (CI/CD) pipeline, you can achieve it using `containers`. A container resource can be a public or private Docker Registry, or Azure Container Registry.

If you need to consume images from Docker registry as part of your pipeline, you can define a generic container resource (not `type` keyword required).

Schema

YAML

```yaml
resources:
  containers:
  - container: string  # identifier (A-Z, a-z, 0-9, and underscore)
    image: string  # container image name
    options: string  # arguments to pass to container at startup
    endpoint: string  # reference to a service connection for the
private registry
    env: { string: string }  # list of environment variables to add
    ports: [ string ] # ports to expose on the container
    volumes: [ string ] # volumes to mount on the container
    mapDockerSocket: bool # whether to map in the Docker daemon socket;
defaults to true
    mountReadOnly:  # volumes to mount read-only - all default to false
      externals: boolean  # components required to talk to the agent
      tasks: boolean  # tasks required by the job
      tools: boolean  # installable tools like Python and Ruby
      work: boolean # the work directory
```

You can use a generic container resource as an image consumed as part of your job, or it can also be used for Container jobs. If your pipeline requires the support of one or more services, you'll want to create and connect to service containers. You can use volumes to share data between services.

You can use a first class container resource type for Azure Container Registry (ACR) to consume your ACR images. This resources type can be used as part of your jobs and also to enable automatic pipeline triggers. You need to have **Contributor** or **Owner** permissions for ACR to use automatic pipeline triggers. For more information, see Azure Container Registry roles and permissions.

Schema

YAML

```yaml
resources:          # types: pipelines | repositories | containers |
builds | packages
  containers:
  - container: string # identifier for the container resource
    type: string # type of the registry like ACR, GCR etc.
    azureSubscription: string # Azure subscription (ARM service
connection) for container registry;
    resourceGroup: string # resource group for your ACR
    registry: string # registry for container images
    repository: string # name of the container image repository in ACR
```

```
    trigger: # Triggers aren't enabled by default and need to be set
explicitly
      enabled: boolean # set to 'true' to trigger on all image tags if
'tags' is unset.
      tags:
        include: [ string ]  # image tags to consider the trigger
events, optional; defaults to any new tag
        exclude: [ string ]  # image tags on discard the trigger events,
optional; defaults to none
```

> **Note**
>
> The syntax that's used to enable container triggers for all image tags (`enabled: 'true'`) is different from the syntax that's used for other resource triggers. Pay close attention to use the correct syntax for a specific resource.

Container resource variables

Once you define a container as a resource, container image metadata gets passed to the pipeline in the form of variables. Information like image, registry, and connection details are accessible across all the jobs to be used in your container deploy tasks.

Schema

YAML

```
resources.container.<Alias>.type
resources.container.<Alias>.registry
resources.container.<Alias>.repository
resources.container.<Alias>.tag
resources.container.<Alias>.digest
resources.container.<Alias>.URI
resources.container.<Alias>.location
```

Location variable is only applicable for `ACR` type of container resources.

Define a `packages` resource

You can consume NuGet and npm GitHub packages as a resource in YAML pipelines.

When you're specifying `package` resources, set the package as NuGet or npm. You can also enable automated pipeline triggers when a new package version gets released.

To use GitHub packages, use personal access token (PAT)-based authentication and create a GitHub service connection that uses PATs.

By default, packages aren't automatically downloaded into jobs. To download, use `getPackage`.

Schema

YAML

```yaml
resources:
  packages:
    - package: myPackageAlias # alias for the package resource
      type: Npm # type of the package NuGet/npm
      connection: GitHubConnectionName # GitHub service connection with the PAT type
      name: nugetTest/nodeapp # <Repository>/<Name of the package>
      version: 1.0.1 # Version of the package to consume; Optional; Defaults to latest
      trigger: true # To enable automated triggers (true/false); Optional; Defaults to no triggers
```

Define a `webhooks` resource

> **Note**
>
> Webhooks were released in Azure DevOps Server 2020.1.

With other resources (such as pipelines, containers, build, and packages) you can consume artifacts and enable automated triggers. However, you can't automate your deployment process based on other external events or services. The `webhooks` resource enables you to integrate your pipeline with any external service and automate the workflow. You can subscribe to any external events through its webhooks (GitHub, GitHub Enterprise, Nexus, Artifactory, and so on) and trigger your pipelines.

Do the following steps to configure the webhook triggers.

1. Set up a webhook on your external service. When you're creating your webhook, you need to provide the following info:
    - Request Url

```
https://dev.azure.com/<ADO
Organization>/_apis/public/distributedtask/webhooks/<WebHook
Name>?api-version=6.0-preview
```

- Secret - Optional. If you need to secure your JSON payload, provide the **Secret** value.

2. Create a new "Incoming Webhook" service connection. This connection is a newly introduced Service Connection Type that allows you to define the following important information:

   - **Webhook Name**: The name of the webhook should match webhook created in your external service.
   - **HTTP Header** - The name of the HTTP header in the request that contains the payload's HMAC-SHA1 hash value for request verification. For example, for GitHub, the request header is "**X-Hub-Signature**".
   - **Secret** - The secret is used to verify the payload's HMAC-SHA1 hash used for verification of the incoming request (optional). If you used a secret when creating your webhook, you must provide the same secret key.

3. A new resource type called `webhooks` is introduced in YAML pipelines. To subscribe to a webhook event, define a webhook resource in your pipeline and point it to the Incoming webhook service connection. You can also define more filters on the webhook resource, based on the JSON payload data, to customize the triggers for each pipeline. Consume the payload data in the form of variables in your jobs.

4. Whenever the Incoming Webhook service connection receives a webhook event, a new run gets triggered for all the pipelines subscribed to the webhook event. You can consume the JSON payload data in your jobs using the format `${{ parameters.<WebhookAlias>.<JSONPath>}}`

Schema

```yml
resources:
  webhooks:
    - webhook: MyWebhookTriggerAlias           ### Webhook alias
      connection: IncomingWebhookConnection    ### Incoming webhook
service connection
      filters:                                 ### List of JSON
parameters to filter; Parameters are AND'ed
        - path: JSONParameterPath              ### JSON path in the
payload
          value: JSONParameterExpectedValue    ### Expected value in the
path provided
```

Webhooks automate your workflow based on any external webhook event that isn't supported by first class resources. Resources like pipelines, builds, containers, and packages. Also, for on-premise services where Azure DevOps doesn't have visibility into the process, you can configure webhooks on the service and to trigger your pipelines automatically.

Manual version picker for resources in the create run dialogue

When you manually trigger a CD YAML pipeline, we automatically evaluate the default version for the resources defined in the pipeline, based on the inputs provided. However, you can choose to pick a different version from resource version picker when you create a run.

1. In the **Create run** pane, select **Resources**. You see a list of resources consumed in this pipeline.

2. Select a resource and pick a specific version from the list of versions available. Resource version picker is supported for pipeline, build, repository, container, and package resources.

For pipeline resources, you can see all the available runs across all branches. Search them based on the pipeline number or branch. And, pick a run that's successful, failed, or in-progress. This flexibility ensures that you can run your CD pipeline if you're sure it produced all the artifacts that you need. You don't need to wait for the CI run to complete or rerun because of an unrelated stage failure in the CI run. However, we only consider successfully completed CI runs when we evaluate the default version for scheduled triggers, or if you don't use manual version picker.

For resources where you can't fetch available versions, like GitHub packages, we show a text box as part of version picker so you can provide the version for the run to pick.

Authorize a YAML pipeline

Resources must be authorized before they can be used. A resource owner controls the users and pipelines that can access that resource. The pipeline must be authorized to use the resource. See the following ways you can authorize a YAML pipeline.

- Go 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 access that resource. This authorization is convenient if you don't have a need to restrict access to a resource - for example, test resources.

- When you create a pipeline for the first time, all the resources that are referenced in the YAML file get automatically authorized for use by the pipeline, if you're a member of the **User** role for that resource. So, resources that are referenced in the YAML file when you create a pipeline get automatically authorized.

- When you make changes to the YAML file and add resources, then the build fails with an error similar to the following error: `Could not find a <resource> with name`

`<resource-name>. The <resource> does not exist or has not been authorized for use.`

In this case, you see an option to authorize the resources on the failed build. If you're 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.

- Verify that the agent pool security roles for your project are correct.

Set approval checks for resources

You can manually control when a resource runs with approval checks and templates. With the required template approval check, you can require any pipeline using a resource or environment also extends from a specific YAML template. Setting a required template approval enhances security. Make sure your resource only gets used under specific conditions with a template. Learn more about how to enhance pipeline security with templates and resources.

Traceability

We provide full traceability for any resource consumed at a pipeline- or deployment job-level.

Pipeline traceability

For every pipeline run, we show the following information.

- The resource that has triggered the pipeline, if it's triggered by a resource.

- Version of the resource and the artifacts consumed.

- Commits associated with each resource.

- Work items associated with each resource.

Environment traceability

Whenever a pipeline deploys to an environment, you can see a list of resources that are consumed. The following view includes resources consumed as part of the deployment jobs and their associated commits and work items.

Show associated CD pipelines information in CI pipelines

To provide end-to-end traceability, you can track which CD pipelines are consuming a giving CI pipeline. You can see the list of CD YAML pipelines runs where a CI pipeline run is consumed through the `pipeline` resource. If other pipelines consume your CI pipeline, you see an "Associated pipelines" tab in the run view. Here you can find all the pipeline runs that consume your pipeline and artifacts from it.

YAML resource trigger issues support and traceability

It can be confusing when pipeline triggers fail to execute. We've added a new menu item in the pipeline definition page called **Trigger Issues**, where you can learn why triggers aren't executing. To access this page, open your pipeline history. **Trigger Issues** is only available for non-repository resources.

Resource triggers can fail to execute for the following reasons.

- If the source of the service connection that's provided is invalid, or if there are any syntax errors in the trigger, the trigger isn't configured, resulting in an error.
- If trigger conditions aren't matched, the trigger won't execute. A warning is surfaced so you can understand why the conditions weren't matched.

Next steps

Add and use variable groups

FAQ

Why should I use pipelines `resources` instead of the `download` shortcut?

Using a `pipelines` resource is a way to consume artifacts from a CI pipeline and also configure automated triggers. A resource gives you full visibility into the process by displaying the version consumed, artifacts, commits, and work items. When you define a pipeline resource, the associated artifacts get automatically downloaded in deployment jobs.

You can choose to download the artifacts in build jobs or to override the download behavior in deployment jobs with `download`. The `download` task internally uses the Download Pipeline Artifacts task.

Why should I use `resources` instead of the Download Pipeline Artifacts task?

When you use the Download Pipeline Artifacts task directly, you miss traceability and triggers. Sometimes it makes sense to use the Download Pipeline Artifacts task directly. For example, you might have a script task stored in a different template and the script task requires artifacts from a build to be downloaded. Or, you may not know if someone using a template wants to add a pipeline resource. To avoid dependencies, you can use the Download Pipeline Artifacts task to pass all the build information to a task.

How can I trigger a pipeline run when my Docker Hub image gets updated?

You'll need to set up a classic release pipeline because the containers resource trigger is not available for Docker Hub for YAML pipelines.

1. Create a new Docker Hub service connection.
2. Create a classic release pipeline and add a Docker Hub artifact. Set your service connection. Select the namespace, repository, version, and source alias.

3. Select the trigger and toggle the continuous deployment trigger to **Enable**. You'll create a release every time a Docker push occurs to the selected repository.

4. Create a new stage and job. Add two tasks, Docker login and Bash:

- The Docker task has the `login` action and logs you into Docker Hub.

- The Bash task runs `docker pull <hub-user>/<repo-name>[:<tag>]`. Replace `hub-user`, `repo-name`, and `tag` with your values.

How can I validate and troubleshoot webhooks?

1. Create a service connection.

2. Reference your service connection and name your webhook in the `webhooks` section.

```yml
resources:
  webhooks:
    - webhook: MyWebhookTriggerAlias
      connection: MyServiceConnection
```

3. Run your pipeline. When you run your pipeline, the webhook will be created in Azure as a distributed task for your organization.

4. Perform a `POST` API call with valid JSON in the body to `https://dev.azure.com/{organization}/_apis/public/distributedtask/webhooks/{webhook-name}?api-version={apiversion}`. If you receive a 200 status code response, your webhook is ready for consumption by your pipeline. If you receive a 500 status code response with the error `Cannot find webhook for the given webHookId ...`, your code may be in a branch that is not your default branch.
   a. Open your pipeline.
   b. Select **Edit**.
   c. Select the more actions menu ⋮ .
   d. Select **Triggers** > **YAML** > **Get Sources**.
   e. Go to **Default branch for manual and scheduled builds** to update your feature branch.
   f. Select **Save & queue**.
   g. After this pipeline runs successfully, perform a `POST` API call with valid JSON in the body to `https://dev.azure.com/{organization}/_apis/public/distributedtask/webhooks/{webhook-name}?api-version={apiversion}`. You should now receive a 200 status code response.

Related articles

- Define variables
- Create and target an environment
- Use YAML pipeline editor

- YAML schema reference

Add & use variable groups

Article • 10/05/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Variable groups store values and secrets that you might want to be passed into a YAML pipeline or make available across multiple pipelines. You can share and use variable groups in multiple pipelines in the same project.

Secret variables in Variables groups are protected resources. You can add combinations of approvals, checks, and pipeline permissions to limit access to secret variables in a variable group. Access to non-secret variables is not limited by approvals, checks, and pipeline permissions.

Create a variable group

YAML

You can't create variable groups in YAML, but they can be used as described in Use a variable group.

Use a variable group

YAML

To use a variable from a variable group, add a reference to the group in your YAML file:

YAML

```yaml
variables:
- group: my-variable-group
```

Then, variables from the variable group can be used in your YAML file.

If you use both variables and variable groups, use the `name`/`value` syntax for the individual non-grouped variables:

YAML

```yaml
variables:
- group: my-variable-group
- name: my-bare-variable
  value: 'value of my-bare-variable'
```

To reference a variable group, use macro syntax or a runtime expression. In the following example, the group `my-variable-group` has a variable named `myhello`.

YAML

```yaml
variables:
- group: my-variable-group
- name: my-passed-variable
  value: $[variables.myhello] # uses runtime expression

steps:
- script: echo $(myhello) # uses macro syntax
- script: echo $(my-passed-variable)
```

You can reference multiple variable groups in the same pipeline. If multiple variable groups include the same variable, the variable group included last in your YAML file sets the variable's value.

YAML

```yaml
variables:
- group: my-first-variable-group
- group: my-second-variable-group
```

You can also reference a variable group in a template. In the template `variables.yml`, the group `my-variable-group` is referenced. The variable group includes a variable named `myhello`.

YAML

```yaml
# variables.yml
variables:
- group: my-variable-group
```

In this pipeline, the variable `$(myhello)` from the variable group `my-variable-group` is included and `variables.yml` is referenced.

YAML

```yaml
# azure-pipeline.yml
stages:
- stage: MyStage
  variables:
  - template: variables.yml
  jobs:
  - job: Test
    steps:
    - script: echo $(myhello)
```

Authorize a variable group

To work with a variable group, you must authorize the group. If you only name the variable group in YAML, then anyone who can push code to your repository could extract the contents of secrets in the variable group. To authorize the group, use one of the following techniques:

- To authorize any pipeline to use the variable group, go to Azure Pipelines. This might be a good option if you don't have any secrets in the group. Select **Library** > **Variable groups**, and then select the variable group in question and enable the setting **Allow access to all pipelines**.
- To authorize a variable group for a specific pipeline, open the pipeline, select **Edit**, and then queue a build manually. You see a resource authorization error and an "Authorize resources" action on the error. Choose this action to explicitly add the pipeline as an authorized user of the variable group.

> ⓘ **Note**
>
> If you add a variable group to a pipeline and don't get a resource authorization error in your build when you expected one, turn off the **Allow access to all pipelines** setting.

Access the variable values in a linked variable group the same way as variables you define within the pipeline itself. For example, to access the value of a variable named **customer** in a variable group linked to the pipeline, use `$(customer)` in a task parameter or a script. But, you can't access secret variables (encrypted variables and key vault variables) directly in scripts - instead, they must be passed as arguments to a task. For more information, see secrets

Changes that you make to a variable group are automatically available to all the definitions or stages to which the variable group gets linked.

List variable groups

Use the CLI to list the variable groups for pipeline runs with the az pipelines variable-group list command. If the Azure DevOps extension for CLI is new to you, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines variable-group list [--action {manage, none, use}]
                                 [--continuation-token]
                                 [--group-name]
                                 [--org]
                                 [--project]
                                 [--query-order {Asc, Desc}]
                                 [--top]
```

Optional parameters

- **action**: Specifies the action that can be performed on the variable groups. Accepted values are *manage*, *none*, and *use*.
- **continuation-token**: Lists the variable groups after a continuation token is provided.
- **group-name**: Name of the variable group. Wildcards are accepted, such as `new-var*`.
- **org**: Azure DevOps organization URL. Configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. Configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.
- **query-order**: Lists the results in either ascending or descending (the default) order. Accepted values are *Asc* and *Desc*.
- **top**: Number of variable groups to list.

Example

The following command lists the top three variable groups in ascending order and returns the results in table format.

Azure CLI

```
az pipelines variable-group list --top 3 --query-order Asc --output table

ID    Name                Type    Number of Variables
----  ------------------  ------  ---------------------
1     myvariables         Vsts    2
2     newvariables        Vsts    4
3     new-app-variables   Vsts    3
```

Show details for a variable group

Display the details of a variable group in your project with the az pipelines variable-group show command. If the Azure DevOps extension for CLI is new to you, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines variable-group show --group-id
                                 [--org]
                                 [--project]
```

Parameters

- **group-id**: Required. ID of the variable group. To find the variable group ID, see List variable groups.
- **org**: Azure DevOps organization URL. Configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. Configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.

Example

The following command shows details for the variable group with the ID **4** and returns the results in YAML format.

Azure CLI

```
az pipelines variable-group show --group-id 4 --output yaml

authorized: false
```

```
description: Variables for my new app
id: 4
name: MyNewAppVariables
providerData: null
type: Vsts
variables:
  app-location:
    isSecret: null
    value: Head_Office
  app-name:
    isSecret: null
    value: Fabrikam
```

Delete a variable group

Delete a variable group in your project with the az pipelines variable-group delete command. If the Azure DevOps extension for CLI is new to you, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines variable-group delete --group-id
                                   [--org]
                                   [--project]
                                   [--yes]
```

Parameters

- **group-id**: Required. ID of the variable group. To find the variable group ID, see List variable groups.
- **org**: Azure DevOps organization URL. Configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. Configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.
- **yes**: Optional. Doesn't prompt for confirmation.

Example

The following command deletes the variable group with the ID **1** and doesn't prompt for confirmation.

Azure CLI

```
az pipelines variable-group delete --group-id 1 --yes

Deleted variable group successfully.
```

Add variables to a variable group

With the Azure DevOps CLI, you can add a variable to a variable group in a pipeline with the az pipelines variable-group variable create command. If the Azure DevOps extension for CLI is new to you, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines variable-group variable create --group-id
                                            --name
                                            [--org]
                                            [--project]
                                            [--secret {false, true}]
                                            [--value]
```

Parameters

- **group-id**: Required. ID of the variable group. To find the variable group ID, see List variable groups.
- **name**: Required. Name of the variable you're adding.
- **org**: Azure DevOps organization URL. Configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. Configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.
- **secret**: Optional. Indicates whether the variable's value is a secret. Accepted values are *false* and *true*.
- **value**: Required for non-secret variable. Value of the variable. For secret variables, if **value** parameter isn't provided, it's picked from environment variable prefixed with `AZURE_DEVOPS_EXT_PIPELINE_VAR_` or user is prompted to enter it via standard input. For example, a variable named **MySecret** can be input using the environment variable `AZURE_DEVOPS_EXT_PIPELINE_VAR_MySecret`.

Example

The following command creates a variable in the variable group with ID of **4**. The new variable is named **requires-login** and has a value of **True**, and the result is shown in table format.

Azure CLI

```
az pipelines variable-group variable create --group-id 4 --name requires-
login --value True --output table

Name            Is Secret    Value
--------------  -----------  -------
requires-login  False        True
```

List variables in a variable group

You can list the variables in a variable group with the az pipelines variable-group variable list command. If the Azure DevOps extension for CLI is new to you, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines variable-group variable list --group-id
                                          [--org]
                                          [--project]
```

Parameters

- **group-id**: Required. ID of the variable group. To find the variable group ID, see List variable groups.
- **org**: Azure DevOps organization URL. Configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. Configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.

Example

The following command lists all of the variables in the variable group with ID of **4** and shows the result in table format.

Azure CLI

```
az pipelines variable-group variable list --group-id 4 --output table

Name            Is Secret    Value
--------------  -----------  -----------
app-location    False        Head_Office
app-name        False        Fabrikam
requires-login  False        True
```

Update variables in a variable group

Update a variable in a variable group with the az pipelines variable-group variable update command. If the Azure DevOps extension for CLI is new to you, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines variable-group variable update --group-id
                                            --name
                                            [--new-name]
                                            [--org]
                                            [--project]
                                            [--prompt-value {false, true}]
                                            [--secret {false, true}]
                                            [--value]
```

Parameters

- **group-id**: Required. ID of the variable group. To find the variable group ID, see List variable groups.
- **name**: Required. Name of the variable you're adding.
- **new-name**: Optional. Specify to change the name of the variable.
- **org**: Azure DevOps organization URL. Configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. Configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.

- **prompt-value**: Set to **true** to update the value of a secret variable using environment variable or prompt via standard input. Accepted values are *false* and *true*.
- **secret**: Optional. Indicates whether the variable's value is kept secret. Accepted values are *false* and *true*.
- **value**: Updates the value of the variable. For secret variables, use the **prompt-value** parameter to be prompted to enter it via standard input. For non-interactive consoles, it can be picked from environment variable prefixed with `AZURE_DEVOPS_EXT_PIPELINE_VAR_`. For example, a variable named **MySecret** can be input using the environment variable `AZURE_DEVOPS_EXT_PIPELINE_VAR_MySecret`.

Example

The following command updates the **requires-login** variable with the new value **False** in the variable group with ID of **4**. It specifies that the variable is a **secret** and shows the result in YAML format. Notice that the output shows the value as **null** instead of **False** since it's a secret hidden value.

Azure CLI

```
az pipelines variable-group variable update --group-id 4 --name requires-
login --value False --secret true --output yaml

requires-login:
  isSecret: true
  value: null
```

Delete variables from a variable group

Delete a variable from a variable group with the az pipelines variable-group variable delete command. If the Azure DevOps extension for CLI is new to you, see Get started with Azure DevOps CLI.

Azure CLI

```
az pipelines variable-group variable delete --group-id
                                            --name
                                            [--org]
                                            [--project]
                                            [--yes]
```

Parameters

- **group-id**: Required. ID of the variable group. To find the variable group ID, see List variable groups.
- **name**: Required. Name of the variable you're deleting.
- **org**: Azure DevOps organization URL. Configure the default organization using `az devops configure -d organization=ORG_URL`. Required if not configured as default or picked up using `git config`. Example: `--org https://dev.azure.com/MyOrganizationName/`.
- **project**: Name or ID of the project. Configure the default project using `az devops configure -d project=NAME_OR_ID`. Required if not configured as default or picked up using `git config`.
- **yes**: Optional. Doesn't prompt for confirmation.

Example

The following command deletes the **requires-login** variable from the variable group with ID of **4** and prompts for confirmation.

Azure CLI

```
az pipelines variable-group variable delete --group-id 4 --name requires-
login

Are you sure you want to delete this variable? (y/n): y
Deleted variable 'requires-login' successfully.
```

Link secrets from an Azure key vault

Link an existing Azure key vault to a variable group and map selective vault secrets to the variable group.

1. In the **Variable groups** page, enable **Link secrets from an Azure key vault as variables**. You'll need an existing key vault containing your secrets. Create a key vault using the Azure portal.

2. Specify your Azure subscription end point and the name of the vault containing your secrets.

   Ensure the Azure service connection has at least **Get** and **List** management permissions on the vault for secrets. Enable Azure Pipelines to set these permissions by choosing **Authorize** next to the vault name. Or, set the permissions manually in the Azure portal:
   a. Open **Settings** for the vault, and then choose **Access policies** > **Add new**.
   b. Select **Select principal** and then choose the service principal for your client account.
   c. Select **Secret permissions** and ensure that **Get** and **List** have check marks.
   d. Select **OK** to save the changes.

3. On the **Variable groups** page, select **+ Add** to select specific secrets from your vault for mapping to this variable group.

Manage key vault secrets

See the following list of helpful tips for managing secrets.

- Only the secret *names* get mapped to the variable group, not the secret values. The latest secret value, fetched from the vault, is used in the pipeline run that's linked to the variable group.

- Any change made to *existing* secrets in the key vault is automatically available to all the pipelines the variable group's used in.

- When new secrets get added to or deleted from the vault, the associated variable groups aren't automatically updated. The secrets included in the variable group must be explicitly updated so the pipelines that are using the variable group get executed correctly.

- Azure Key Vault supports storing and managing cryptographic keys and secrets in Azure. Currently, Azure Pipelines variable group integration supports mapping only

secrets from the Azure key vault. Cryptographic keys and certificates aren't supported.

Expand variables in a group

YAML

When you set a variable in a group and use it in a YAML file, it's equal to other defined variables in the YAML file. For more information about precedence of variables, see Variables.

Related articles

- Use Azure Key Vault secrets in Azure Pipelines
- Define variables
- Use secret and nonsecret variables in variable groups
- Add approvals and checks

Use secure files

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Secure files give you a way to store files that you can share across pipelines. Use the secure files library to store files such as:

- signing certificates
- Apple Provisioning Profiles
- Android Keystore files
- SSH keys

These files can be stored on the server without having to commit them to your repository.

The contents of the secure files are encrypted and can only be used when you consume them from a task. Secure files are a protected resource. You can add approvals and checks to them and set pipeline permissions. Secure files also can use the Library security model.

The size limit for each secure file is 10 MB.

Add a secure file

1. Go to **Pipelines** > **Library** > **Secure files**.

2. Select + **Secure file** to upload a new secure file. Browse to upload or drag and drop your file. You can delete this file, but you can't replace it.

3. Add permissions to your file.
   a. Apply security role restrictions for all files from the **Security** tab at **Pipelines** > **Library**.
   b. To add permissions for an individual file, in the file's edit view, select **Pipeline permissions** to set per-pipeline permissions. Or, select **Security** to set security roles.

      - You can also set **Approvals and Checks** for the file. For more information, see Approvals and checks.

Consume a secure file in a pipeline

Use the Download Secure File utility task to consume secure files in a pipeline.

The following YAML pipeline example downloads a secure certificate file and installs it in a Linux environment.

```
YAML
```

```yaml
- task: DownloadSecureFile@1
  name: caCertificate
  displayName: 'Download CA certificate'
  inputs:
    secureFile: 'myCACertificate.pem'

- 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)
```

FAQ

Q: How can I create a custom task using secure files?

A: Build your own tasks that use secure files by using inputs with type `secureFile` in the `task.json`. Learn how to build a custom task.

The Install Apple Provisioning Profile task is a simple example of a task using a secure file. See the reference documentation and source code.

To handle secure files during build or release, you can refer to the common module available here.

Q: My task can't access the secure files. What do I do?

A: Make sure your agent is running version of 2.116.0 or higher. See Agent version and upgrades.

Q: How do I authorize a secure file for use in a specific pipeline?

A:

1. In **Azure Pipelines**, select the **Library** tab.
2. Select the **Secure files** tab at the top.
3. Select the secure file you want to authorize.
4. Select the **Pipeline permissions** button.
5. Review and modify the access for each available pipeline.

Q: Why do I see an `Invalid Resource` error when downloading a secure file with Azure DevOps Server/TFS on-premises?

A: Make sure IIS Basic Authentication is disabled on the TFS or Azure DevOps Server.

Q: How are secure files secured?

A: Secure files, variable groups, and service connections are all secured in the same way in Azure DevOps. They're also all protected resources.

Secrets are encrypted and stored in the database. The keys to decrypt secrets are stored in Azure Key Vault. The keys are specific to each scale unit. So, two regions don't share the same keys. The keys are also rotated with every deployment of Azure DevOps.

The rights to retrieve secure keys are only given to the Azure DevOps service principals and (on special occasions) on-demand to diagnose problems. The secure storage doesn't have any certifications.

Azure Key Vault is another, more secure option for securing sensitive information. If you decide to use Azure Key Vault, you can use it with variable groups.

Provision deployment groups

Article • 04/17/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

A deployment group is a logical set of deployment target machines that have agents installed on each one. Deployment groups represent the physical environments; for example, "Dev", "Test", or "Production" environment. In effect, a deployment group is just another grouping of agents, much like an agent pool.

Deployment groups are only available with Classic release pipelines and are different from deployment jobs. A deployment job is a collection of deployment-related steps defined in a YAML file to accomplish a specific task.

With deployment groups you can:

- Specify the security context and runtime targets for the agents. As you create a deployment group, you add users and give them appropriate permissions to administer, manage, view, and use the group.
- Let you view live logs for each server as a deployment takes place, and download logs for all servers to track your deployments down to individual machines.
- Enable you to use machine tags to limit deployment to specific sets of target servers.

Create a deployment group

A deployment group is a set of virtual machines with deployment agents. Every VM of the deployment group interacts with Azure Pipelines to coordinate the deployment tasks.

1. From within your project, select **Pipelines** > **Deployment groups**.

2. Select **Add a deployment group**.

3. Enter a **Deployment group name** and then select **Create**. A registration script will be generated. Select the **Type of target to register** and then select **Use a personal access token in the script for authentication**. Finally, select **Copy script to the clipboard**.

4. Log onto each of your target machines and run the script from an elevated PowerShell command prompt to register it as a target server. When prompted to enter tags for your agent, press *Y* and enter the tag(s) you will use to filter subsets of the servers.

```
d-----         6/18/2021   11:28 AM                azagent


    Directory: C:\azagent


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----         6/18/2021   11:28 AM                A1

  ___                      _____ _                 _ _
 / _ \                    | ___ (_)               | (_)
/ /_\ \_____   _ _ __ ___ | |_/ /_ _ __   ___ | |_ _ __   ___ ___
|  _  |_  / | | | '__/ _ \ |  __/| | '_ \ / _ \ | | '_ \ / _ \/ __|
| | | |/ /| |_| | | |  __/ | |   | | |_) |  __/ | | | | |  __/\__ \
\_| |_/___|\__,_|_|  \___| \_|   |_| .__/ \___|_|_|_| |_|\___||___/
                                   | |
        agent v2.188.1             |_|          (commit b6472da)


>> Connect:
```

After setting up your target servers, the script should return the following message: `Service vstsagent.{organization-name}.{computer-name} started successfully`.

The tags you assign to your target servers allow you to limit deployment to specific servers in a Deployment group job. A tag is limited to 256 characters, but there is no limit to the number of tags you can use.

> ⓘ **Note**

A deployment pool is a set of target servers available to the organization (org-scoped). When you create a new deployment pool for projects in your organization, a corresponding deployment group is automatically provisioned for each project. The deployment groups will have the same target servers as the deployment pool. You can manually trigger an agent version upgrade for your target servers by hovering over the ellipsis (...) in **Deployment Pools** and selecting **Update targets**. See **Agent versions and upgrades** for more details.

Set up agents on deployment groups

Every target server in the deployment group requires a deployment agent. You can install an agent on your target servers in three different ways:

- By running the generated script from the **Deployment Groups** tab.
- If the target servers are Azure VMs, you can easily set up your servers by installing the Azure Pipelines Agent extension on each of the VMs.
- By using the ARM template deployment task in your release pipeline to create a deployment group dynamically.

You can force the agents on the target servers to be upgraded to the latest version without needing to redeploy them by selecting **Update targets** from your deployment groups page.

Monitor release status for deployment groups

When a release pipeline is executing, you can view the live logs for each target server in your deployment group. When the deployment is completed, you can download the log files for every server to examine the deployments and debug any issues.

Share a deployment group with other projects

Deployment groups can be shared with other projects in the same organization. Follow the steps below to provision your deployment group for other projects:

1. From within your project, select **Pipelines** > **Deployment groups**.

2. Select your deployment group and then select **Manage**.

3. Select a project from the list and then select **Save**.

4. You will now notice that there is a new Deployment Group in the project you just included.

Automatically deploy to new target servers

When new target servers get added to a deployment group, you can configure the environment to automatically deploy the last successful release to the new targets.

From your release pipeline definition, select the post deployment icon, and then enable the **Auto redeploy trigger**. Select the events and action as shown below.

Related articles

- Deployment group jobs
- Deploy to Azure VMs using deployment groups
- Provision agents for deployment groups
- Self-hosted Windows agents
- Self-hosted macOS agents
- Self-hosted Linux agents

Create and target an environment

Article • 07/17/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

An environment is a collection of resources that you can target with deployments from a pipeline. Typical examples of environment names are Dev, Test, QA, Staging, and Production. An Azure DevOps environment represents a logical target where your pipeline deploys software.

Azure DevOps environments aren't available in classic pipelines. For classic pipelines, deployment groups offer similar functionality.

Environments provide the following benefits.

Benefit	Description
Deployment history	Pipeline name and run details get recorded for deployments to an environment and its resources. In the context of multiple pipelines targeting the same environment or resource, deployment history of an environment is useful to identify the source of changes.
Traceability of commits and work items	View jobs within the pipeline run that target an environment. You can also view the commits and work items that were newly deployed to the environment. Traceability also allows one to track whether a code change (commit) or feature/bug-fix (work items) reached an environment.
Diagnostic resource health	Validate whether the application is functioning at its wanted state.
Security	Secure environments by specifying which users and pipelines are allowed to target an environment.

While an environment is a grouping of resources, the resources themselves represent actual deployment targets. The Kubernetes resource and virtual machine resource types are currently supported.

When you author a YAML pipeline and refer to an environment that doesn't exist, Azure Pipelines automatically creates the environment when the user performing the operation is known and permissions can be assigned. When Azure Pipelines doesn't have information about the user creating the environment (example: a YAML update from an external code editor), your pipeline fails if the environment doesn't already exist.

Prerequisites

- You need to have the **Creator** role for Environments to add an Environment.

Create an environment

1. Sign in to your organization: `https://dev.azure.com/{yourorganization}` and select your project.

2. Select **Pipelines** > **Environments** > **Create environment**.

3. Enter information for the environment, and then select **Create**. Resources can be added to an existing environment later.

Use a Pipeline to create and deploy to environments, too. For more information, see the how-to guide.

> **Tip**
>
> You can create an empty environment and reference it from deployment jobs. This lets you record the deployment history against the environment.

Target an environment from a deployment job

A deployment job is a collection of steps to be run sequentially. A deployment job can be used to target an entire environment (group of resources) as shown in the following YAML snippet. The pipeline will run on the myVM machine because the resource name is specified.

YAML

```yaml
- stage: deploy
  jobs:
  - deployment: DeployWeb
    displayName: deploy Web App
    pool:
      vmImage: 'Ubuntu-latest'
    # creates an environment if it doesn't exist
    environment:
      name: 'smarthotel-dev'
      resourceName: myVM
      resourceType: virtualMachine
    strategy:
      runOnce:
        deploy:
          steps:
          - script: echo Hello world
```

Target a specific resource within an environment from deployment job

You can scope the target of deployment to a particular resource within the environment. Then, you can record deployment history on a specific resource within the environment. The steps of the deployment job *automatically inherit* the service connection details from the resource targeted by the deployment job.

YAML

```yaml
environment:
  name: 'smarthotel-dev.bookings'
strategy:
 runOnce:
   deploy:
     steps:
     - task: KubernetesManifest@0
       displayName: Deploy to Kubernetes cluster
       inputs:
         action: deploy
         namespace: $(k8sNamespace)
         manifests: $(System.ArtifactsDirectory)/manifests/*
         imagePullSecrets: $(imagePullSecret)
         containers: $(containerRegistry)/$(imageRepository):$(tag)
         # value for kubernetesServiceConnection input automatically passed
down to task by environment.resource input
```

Environment in run details

All environments that get targeted by deployment jobs of a specific run of a pipeline can be found under the *Environments* tab of pipeline run details.

If you're using an AKS private cluster, the **Environments** tab isn't available.

Approvals

Manually control when a stage should run using approval checks. Use approval checks to control deployments to production environments. Checks are available to the resource Owner to control when a stage in a pipeline consumes a resource. As the owner of a resource, such as an environment, you can define approvals and checks that must be satisfied before a stage consuming that resource starts.

We support manual approval checks on environments. For more information, see Approvals.

The Creator, Administrator, and user roles can manage approvals and checks. The Reader role can't manage approvals and checks.

Deployment history

The deployment history view within environments provides the following advantages.

- View jobs from all pipelines that target a specific environment. For example, two micro-services, each having its own pipeline, are deploying to the same environment. The deployment history listing helps identify all pipelines that affect this environment and also helps visualize the sequence of deployments by each pipeline.

- Drill down into the job details to see the list of commits and work items that were deployed to the environment. The list of commits and work items are the new items between deployments. Your first listing includes all of the commits and the following listings will just include changes. If multiple commits are tied to the same pull request, you'll see multiple results on the work items and changes tabs.

- If multiple work items are tied to the same pull request, you'll see multiple results on the work items tab.

Security

User permissions

Control who can create, view, use, and manage the environments with user permissions. There are four roles - Creator (scope: all environments), Reader, User, and Administrator. In the specific environment's **user permissions** panel, you can set the permissions that are inherited and you can override the roles for each environment.

1. Go to the specific **Environment** that you want to authorize.

2. Select ⋮ > **Security** to view the settings.
3. Select **User permissions** > **+Add** > **User or group**, and then select a suitable role.

Role	Description
Creator	Global role, available from environments hub security option. Members of this role can create the environment in the project. Contributors are added as members by default. Required to trigger a YAML pipeline when the environment does not already exist.
Reader	Members of this role can view the environment.
User	Members of this role can use the environment when creating or editing YAML pipelines.
Administrator	Members of this role can administer permissions, create, manage, view and use environments. For a particular environment, its creator is added as Admininistrator by default.

Pipeline permissions

Use pipeline permissions to authorize all or selected pipelines for deployment to the environment.

- To remove **Open access** on the environment or resource, select **Restrict permission** in **Pipeline permissions**.
- To allow specific pipelines to deploy to an environment or specific resource, select + and choose from the list of pipelines.

Next steps

Define approvals and checks

FAQ

Q: Why do I get an error message when I try to create an environment?

A: If you see the message "Access denied: {User} needs Create permissions to do the action", check your organization-level permissions. Go to **Organization Settings** > **Users** and check if you have the stakeholder role. The stakeholder role can't create

environments. Change your access level and then check to see if you can create environments. For more information, see User and permissions management FAQ.

Q: Why am I getting error "Job XXXX: Environment XXXX could not be found. The environment does not exist or has not been authorized for use"?

A: These are some of the possible reasons of the failure:

- When you author a YAML pipeline and refer to an environment that doesn't exist in the YAML file, Azure Pipelines automatically creates the environment in some cases:
  - You use the YAML pipeline creation wizard in the Azure Pipelines web experience and refer to an environment that hasn't been created yet.
  - You update the YAML file using the Azure Pipelines web editor and save the pipeline after adding a reference to an environment that does not exist.
- In the following flows, Azure Pipelines doesn't have information about the user creating the environment: you update the YAML file using another external code editor, add a reference to an environment that doesn't exist, and then cause a manual or continuous integration pipeline to be triggered. In this case, Azure Pipelines doesn't know about the user. Previously, we handled this case by adding all the project contributors to the administrator role of the environment. Any member of the project could then change these permissions and prevent others from accessing the environment.
- You can use variables to create the environment or use templateContext to pass properties to templates. Runtime parameters won't work when creating the environment because they are expanded at run time.
- A user with stakeholder access level can't create the environment as stakeholders don't have access to the repository.

Related articles

- Define variables
- Define resources in YAML

Environment - Kubernetes resource

Article • 11/06/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

The Kubernetes resource view shows the status of objects within the namespace that are mapped to the resource. The resource view also overlays pipeline traceability so you can trace back from a Kubernetes object to the pipeline, and then back to the commit.

Use Kubernetes resources to target Kubernetes clusters in an environment for deployment. Use pipelines to deploy to Azure Kubernetes Service (AKS) and clusters from any other cloud provider.

You can use Kubernetes resources with public or private clusters. To learn more about how resources work, see resources in YAML and security with resources.

Overview

See the following advantages of using Kubernetes resource views within environments:

- **Pipeline traceability** - The Kubernetes manifest task, used for deployments, adds more annotations to show pipeline traceability in resource views. Pipeline traceability helps to identify the originating Azure DevOps organization, project, and pipeline responsible for updates that were made to an object within the namespace.

- **Diagnose resource health** - Workload status can be useful for quickly debugging mistakes or regressions that were introduced by a new deployment. For example, for unconfigured *imagePullSecrets* resulting in ImagePullBackOff errors, pod status information can help you identify the root cause for the issue.

- **Review App** - Review App works by deploying every pull request from your Git repository to a dynamic Kubernetes resource under the environment. Reviewers can see how those changes look and work with other dependent services before they're merged into the target branch and deployed to production.

Use Azure Kubernetes Service

A ServiceAccount gets created in your chosen cluster and namespace when you use Azure Kubernetes Service (AKS). For a Kubernetes RBAC-enabled cluster, RoleBinding also gets created to limit the scope of the created service account to the chosen namespace. For a Kubernetes RBAC-disabled cluster, the ServiceAccount created has cluster-wide privileges (across namespaces).

Add an AKS Kubernetes resource

1. In the environment details page, select **Add resource** and choose **Kubernetes**.
2. Select **Azure Kubernetes Service** in the Provider dropdown.
3. Choose the Azure subscription, cluster, and namespace (new/existing).
4. Select **Validate and create** to create the Kubernetes resource.
5. Verify that you see a cluster for your environment. You'll see the text "Never deployed" if you haven't yet deployed code to your cluster.

Use an existing service account

The Azure Kubernetes Service maps a Kubernetes resource within your environment to a namespace.

For more information about setting up a Kubernetes service connection outside of an environment, see the Kubernetes service connection section in Service connections.

> **Tip**
>
> Use the generic provider (existing service account) to map a Kubernetes resource to a namespace from a non-AKS cluster.

Add a non-AKS Kubernetes resource

1. In the environment details page, select **Add resource** and choose **Kubernetes**.
2. Select **Generic provider (existing service account)** for your provider.
3. Add the cluster name and namespace values.
4. Add the server URL. You can get the URL with the following command:

```
kubectl config view --minify -o 'jsonpath=
{.clusters[0].cluster.server}'
```

5. To get the secret object.

Kubernetes 1.22+

Replace `service-account-name` with your account name.

```
kubectl get secret -n <namespace>  -o jsonpath='{.items[?(@.metadata.annotations.kubernetes\.io/service-account\.name==\"service-account-name\")]}'
```

If you get nothing, see Manually create a long-lived API token for a ServiceAccount.

Kubernetes 1.22 and below:

a. Find the service account secret name

```
kubectl get serviceAccounts <service-account-name> -n <namespace> -o 'jsonpath={.secrets[*].name}'
```

b. replace `<service-account-secret-name>` with the value in previous command in this command

```
kubectl get secret <service-account-secret-name> -n <namespace> -o json
```

6. Get the secret object using the output of the previous step.

```
kubectl get secret <service-account-secret-name> -n <namespace> -o json
```

7. Copy and paste the Secret object fetched in JSON form into the Secret field.

8. Select **Validate and create** to create the Kubernetes resource.

Reference your Kubernetes resources in a pipeline

If you're using Azure Kubernetes Service and building a YAML pipeline, the easiest way to configure your pipeline is to use a template. Connect to your repository and select one of the following two Kubernetes Service options:

- Deploy to Azure Kubernetes Services template
- Deploy to Kubernetes - Review App with Azure DevSpaces

The templates let you set up Review App without needing to write YAML code from scratch or manually create explicit role bindings.

Set up Review App

In the following example, the first deployment job is run for non-PR branches and does deployments against a regular Kubernetes resource under environments. The second job runs only for PR branches and deploys against Review App resources (namespaces inside Kubernetes cluster) generated on demand. Resources get labeled with "Review" in the resource listing view of the environment. Define variables to use in the pipeline. If you use the Deploy to Azure Kubernetes Services template, these variables get defined for you.

YAML

```yaml
# Build and push image to Azure Container Registry; Deploy to Azure
Kubernetes Service
trigger:
- main

resources:
- repo: self

variables:

  # Container registry service connection established during pipeline
creation
  dockerRegistryServiceConnection: '12345' # Docker service connection
identifier
  envName: 'myEnv' # name of your environment
  imageRepository: 'name-of-image-repository' # name of image repository
  containerRegistry: 'mycontainer.azurecr.io' # path to container registry
  dockerfilePath: '**/Dockerfile'
  tag: '$(Build.BuildId)'
  imagePullSecret: 'my-app-secret' # image pull secret

  # Agent VM image name
  vmImageName: 'ubuntu-latest'

  # Name of the new namespace being created to deploy the PR changes.
```

```yaml
  k8sNamespaceForPR: 'review-app-$(System.PullRequest.PullRequestId)'

stages:
- stage: Build
  displayName: Build stage
  jobs:
  - job: Build
    displayName: Build
    pool:
      vmImage: $(vmImageName)
    steps:
    - task: Docker@2
      displayName: Build and push an image to container registry
      inputs:
        command: buildAndPush
        repository: $(imageRepository)
        dockerfile: $(dockerfilePath)
        containerRegistry: $(dockerRegistryServiceConnection)
        tags: |
          $(tag)

    - upload: manifests
      artifact: manifests

- stage: Production
  displayName: Deploy stage
  dependsOn: Build

  jobs:
  - deployment: Production
    condition: and(succeeded(),
not(startsWith(variables['Build.SourceBranch'], 'refs/pull/')))
    displayName: Production
    pool:
      vmImage: $(vmImageName)
    environment:
      name: $(envName).$(resourceName)
      resourceType: Kubernetes
    strategy:
      runOnce:
        deploy:
          steps:
          - task: KubernetesManifest@0
            displayName: Create imagePullSecret
            inputs:
              action: createSecret
              secretName: $(imagePullSecret)
              dockerRegistryEndpoint: $(dockerRegistryServiceConnection)

          - task: KubernetesManifest@0
            displayName: Deploy to Kubernetes cluster
            inputs:
              action: deploy
              manifests: |
                $(Pipeline.Workspace)/manifests/deployment.yml
```

```
                $(Pipeline.Workspace)/manifests/service.yml
              imagePullSecrets: |
                $(imagePullSecret)
              containers: |
                $(containerRegistry)/$(imageRepository):$(tag)

  - deployment: DeployPullRequest
    displayName: Deploy Pull request
    condition: and(succeeded(), startsWith(variables['Build.SourceBranch'],
'refs/pull/'))
    pool:
      vmImage: $(vmImageName)

    environment:
      name: $(envName).$(resourceName)
      resourceType: Kubernetes
    strategy:
      runOnce:
        deploy:
          steps:
          - reviewApp: default

          - task: Kubernetes@1
            displayName: 'Create a new namespace for the pull request'
            inputs:
              command: apply
              useConfigurationFile: true
              inline: '{ "kind": "Namespace", "apiVersion": "v1",
"metadata": { "name": "$(k8sNamespaceForPR)" }}'

          - task: KubernetesManifest@0
            displayName: Create imagePullSecret
            inputs:
              action: createSecret
              secretName: $(imagePullSecret)
              namespace: $(k8sNamespaceForPR)
              dockerRegistryEndpoint: $(dockerRegistryServiceConnection)

          - task: KubernetesManifest@0
            displayName: Deploy to the new namespace in the Kubernetes
cluster
            inputs:
              action: deploy
              namespace: $(k8sNamespaceForPR)
              manifests: |
                $(Pipeline.Workspace)/manifests/deployment.yml
                $(Pipeline.Workspace)/manifests/service.yml
              imagePullSecrets: |
                $(imagePullSecret)
              containers: |
                $(containerRegistry)/$(imageRepository):$(tag)

          - task: Kubernetes@1
            name: get
            displayName: 'Get services in the new namespace'
```

```
            continueOnError: true
            inputs:
              command: get
              namespace: $(k8sNamespaceForPR)
              arguments: svc
              outputFormat:
jsonpath='http://{.items[0].status.loadBalancer.ingress[0].ip}:
{.items[0].spec.ports[0].port}'

          # Getting the IP of the deployed service and writing it to a
variable for posting comment
          - script: |
              url="$(get.KubectlOutput)"
              message="Your review app has been deployed"
              if [ ! -z "$url" -a "$url" != "http://:" ]
              then
                message="${message} and is available at $url.<br><br>[Learn
More](https://aka.ms/testwithreviewapps) about how to test and provide
feedback for the app."
              fi
              echo "##vso[task.setvariable variable=GITHUB_COMMENT]$message"
```

To use this job in an **existing** pipeline, the service connection backing the regular Kubernetes environment resource must be modified to "Use cluster admin credentials". Otherwise, role bindings must be created for the underlying service account to the Review App namespace.

Next steps

Build and deploy to Azure Kubernetes Service

Related articles

- Deploy
- Deploy ASP.NET Core apps to Azure Kubernetes Service with Azure DevOps Starter
- REST API: Kubernetes with Azure DevOps

Environment - virtual machine resource

Article • 03/20/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Use virtual machine (VM) resources to manage deployments across multiple machines with YAML pipelines. VM resources let you install agents on your own servers for rolling deployments.

VM resources connect to environments. After you define an environment, you can add VMs to target with deployments. The deployment history view in an environment provides traceability from your VM to your pipeline.

Prerequisites

You must have at least a Basic license and access to the following areas:

- the repository connected to your pipeline
- the VM you want to connect to the environment

For more information about security for Azure Pipelines, see Pipeline security resources.

To add a VM to an environment, you must have the Administrator role for the corresponding deployment pool. A deployment pool is a set of target servers available to the organization. Learn more about deployment pool and environment permissions.

> ⓘ **Note**
>
> If you are configuring a deployment group agent, or if you see an error when registering a VM environment resource, you must set the PAT scope to **All accessible organizations**.

Create a VM resource

> ⓘ **Note**
>
> You can use this same process to set up physical machines with a registration script.

The first step in adding a VM resource is to define an environment.

Define an environment

1. Select **Create environment** or **New environment**, depending on whether it's your first environment.
2. Add a **Name** (required) for the environment and a **Description**.
3. Save the new environment.

Add a resource

1. Select your environment and choose **Add resource**.

2. Select **Virtual machines** for your **Resource** type. Then select **Next**.

3. Choose Windows or Linux for the **Operating System**.

4. Copy the registration script. Your script will be a PowerShell script if you've selected Windows and a Linux script if you've selected Linux.

5. Run the copied script on each of the target virtual machines that you want to register with this environment.

   - If you're installing on Windows, you'll need to run the script an PowerShell administrator.
   - If you're installing on Linux, you'll need to have permission to download and run executable scripts.

   > **Note**
   >
   > - The Personal Access Token (PAT) for the signed-in user gets included in the script. The PAT expires on the day you generate the script.
   > - If your VM already has any other agent running on it, provide a unique name for **agent** to register with the environment.
   > - To learn more about installing the agent script, see **Self-hosted Linux agents** and **Self-hosted Windows agents**. The agent scripts for VM resources are like the scripts for self-hosted agents and you can use the same commands.

6. Once your VM is registered, it appears as an environment resource under the **Resources** tab of the environment.

7. To add more VMs, copy the script again. Select **Add resource** > **Virtual machines**. The Windows and Linux scripts are the same for all the VMs added to the environment.

8. When the VM script is successfully installed, your VM appears in the list of resources for the environment.

Use VM in pipelines

Target VMs in your pipeline by referencing the environment. By default, the pipeline job runs for all of the VMs defined for an environment with a `resourceName`.

YAML

```yaml
trigger:
- main

pool:
   vmImage: ubuntu-latest

jobs:
- deployment: VMDeploy
  displayName: Deploy to VM
  environment:
   name: VMenv
   resourceName: VMenv
   resourceType: virtualMachine
  strategy:
     runOnce:
        deploy:
          steps:
            - script: echo "Hello world"
```

> **Note**
>
> The `resourceType` values are case sensitive. Specifying the incorrect casing will result in no matching resources found in the environment. See the **YAML schema** for more information.

You can select a specific virtual machine from the environment to only receive the deployment by specifying it by its `resourceName`. For example, to target deploying only

to the Virtual Machine resource named `USHAN-PC` in the `VMenv` environment, add the `resourceName` parameter and give it the value of `USHAN-PC`.

YAML

```yaml
trigger:
- main

pool:
   vmImage: ubuntu-latest

jobs:
- deployment: VMDeploy
  displayName: Deploy to VM
  environment:
    name: VMenv
    resourceType: virtualMachine
    resourceName: USHAN-PC # only deploy to the VM resource named USHAN-PC
  strategy:
    runOnce:
      deploy:
          steps:
          - script: echo "Hello world"
```

To learn more about deployment jobs, see the YAML schema.

Add and manage tags

Tags give you a way to target a set of specific VMs in an environment for deployment. You can add tags to the VM as part of the interactive registration script or through the UI. Tags are each limited to 256 characters. There's no limit to the number of tags that you can use.

Add or remove tags in the UI from the resource view by selecting **More actions** ⋮ for a VM resource.

When you select multiple tags, VMs that include all the tags get used in your pipeline. For example, this pipeline targets VMs with both the `windows` and `prod` tags. If a VM only has one of these tags, it's not targeted.

YAML

```yaml
trigger:
- main

pool:
   vmImage: ubuntu-latest

jobs:
- deployment: VMDeploy
  displayName: Deploy to VM
  environment:
    name: VMenv
    resourceType: virtualMachine
    tags: windows,prod # only deploy to virtual machines with both windows
and prod tags
  strategy:
    runOnce:
      deploy:
          steps:
          - script: echo "Hello world"
```

Apply deployment strategy

Apply a deployment strategy to define how your application gets rolled out. The `runOnce` strategy and the `rolling` strategy for VMs are both supported. For more information about deployment strategies and life-cycle hooks, see Deployment jobs/Deployment strategies.

View deployment history

Select the **Deployments** tab for complete traceability of commits and work items, and a cross-pipeline deployment history per environment and resource.

Remove a VM from an environment

Windows environment

To remove VMs from a Windows environment, run the following command. Ensure you do the following tasks:

- Run the command from an administrator PowerShell command prompt
- Run the command on each machine
- Run the command in the same folder path as the environment registration command was run

```
./config.cmd remove
```

Linux environment

To remove a VM from a Linux environment, run the following command on each machine.

```
./config.sh remove
```

Known limitations

When you retry a stage, it reruns the deployment on all VMs and not just failed targets.

Related articles

- About environments
- Learn about deployment jobs
- YAML schema reference

Azure Pipelines agents

Article • 10/09/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

To build your code or deploy your software using Azure Pipelines, you need at least one agent. As you add more code and people, you'll eventually need more.

When your pipeline runs, the system begins one or more jobs. An agent is computing infrastructure with installed agent software that runs one job at a time.

Azure Pipelines provides several different types of agents.

Agent type	Description	Availability
Microsoft-hosted agents	Agents hosted and managed by Microsoft	Azure DevOps Services
Self-hosted agents	Agents that you configure and manage, hosted on your VMs	Azure DevOps Services, Azure DevOps Server, TFS
Azure Virtual Machine Scale Set agents	A form of self-hosted agents, using Azure Virtual Machine Scale Sets, that can be auto-scaled to meet demands	Azure DevOps Services

Jobs can be run directly on the host machine of the agent or in a container.

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 for each job in the pipeline. The virtual machine is discarded after one job (which means any change that a job makes to the virtual machine file system, such as checking out code, will be unavailable to the next job). Microsoft-hosted agents can run jobs directly on the VM or in a container.

Azure Pipelines provides a predefined agent pool named **Azure Pipelines** with Microsoft-hosted agents.

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.

> **Tip**
>
> You can try a Microsoft-hosted agent for no charge.

Learn more about Microsoft-hosted agents.

Self-hosted agents

An agent that you set up and manage on your own to run jobs is a **self-hosted agent**. You can use self-hosted agents in Azure Pipelines or Azure DevOps Server, formerly named Team Foundation Server (TFS). 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.

> **Note**
>
> Although multiple agents can be installed per machine, we strongly suggest to only install one agent per machine. Installing two or more agents might adversely affect performance and the result of your pipelines.

> **Tip**
>
> Before you install a self-hosted agent you might want to see if a Microsoft-hosted agent pool will work for you. In many cases this is the simplest way to get going. **Give it a try**.

You can install the agent on Linux, macOS, or Windows machines. You can also install an agent on a Docker container. For more information about installing a self-hosted agent, see:

- macOS agent
- Linux agent
- Windows agent
- Docker agent

> **Note**
>
> On macOS, you need to clear the special attribute on the download archive to prevent Gatekeeper protection from displaying for each assembly in the tar file

when `./config.sh` is run. The following command clears the extended attribute on the file:

```bash
xattr -c vsts-agent-osx-x64-V.v.v.tar.gz  ## replace V.v.v with the version in the filename downloaded.

# then unpack the gzip tar file normally:

tar xvfz vsts-agent-osx-x64-V.v.v.tar.gz
```

After you've installed the agent on a machine, you can install any other software on that machine as required by your jobs.

> **Note**
>
> Agents are widely backward compatible. Any version of the agent should be compatible with any Azure DevOps version as long as Azure DevOps isn't demanding a higher version of the agent.
>
> We only support the most recent version of the agent since that is the only version guaranteed to have all up-to-date patches and bug fixes.

Node runner versions

The agent ships with several versions of NodeJS libraries to support target tasks that use different Node handlers.

All official Azure DevOps tasks use Node 10 as a universal handler, however, customers might still use custom tasks that use the outdated Node 6 library. To support backward compatibility with Node that has currently reached End-of-Life, we provide the following self-service methods to install the designated Node runner manually.

- Manually install the Node 6 runner. For more information on manually installing the Node 6 runner, see Node 6 support for more details.
- Use the NodeTaskRunnerInstaller@0 task in your pipelines that require the outdated Node 6 library.
- Install an agent package that includes Node 6.

  Azure Pipelines provides two versions of agent packages.

- **vsts-agent-*** packages support Node 6.
- **pipelines-agent-*** packages do not support Node 6. This version of the package will become the default agent package in the future.

If you know that you are not using any Node 6 dependant tasks, and you don't want Node 6 installed on your agent machine, you can install the agent from the **Alternate Agent Downloads** section from https://github.com/microsoft/azure-pipelines-agent/releases .

Azure Virtual Machine Scale Set agents

Azure Virtual Machine Scale Set agents are a form of self-hosted agents that can be auto-scaled to meet your demands. This elasticity reduces your need to run dedicated agents all the time. Unlike Microsoft-hosted agents, you have flexibility over the size and the image of machines on which agents run.

You specify a Virtual Machine Scale Set, a number of agents to keep on standby, a maximum number of virtual machines in the scale set, and Azure Pipelines manages the scaling of your agents for you.

For more information, see Azure Virtual Machine Scale Set agents.

Parallel jobs

Parallel jobs represents the number of jobs you can run at the same time in your organization. If your organization has a single parallel job, you can run a single job at a time in your organization, with any additional concurrent jobs being queued until the first job completes. To run two jobs at the same time, you need two parallel jobs. In Azure Pipelines, you can run parallel jobs on Microsoft-hosted infrastructure or on your own (self-hosted) infrastructure.

Microsoft provides a free tier of service by default in every organization that includes at least one parallel job. Depending on the number of concurrent pipelines you need to run, you might need more parallel jobs to use multiple Microsoft-hosted or self-hosted agents at the same time. For more information on parallel jobs and different free tiers of service, see Parallel jobs in Azure Pipelines.

Capabilities

Every self-hosted agent has a set of capabilities that indicate what it can do. Capabilities are name-value pairs that are either automatically discovered by the agent software, in

which case they are called **system capabilities**, or those that you define, in which case they are called **user capabilities**.

The agent software automatically determines various system capabilities such as the name of the machine, type of operating system, and versions of certain software installed on the machine. Also, environment variables defined in the machine automatically appear in the list of system capabilities.

> **Note**
>
> Storing environment variables as capabilities means that when an agent runs, the stored capability values are used to set the environment variables. Also, any changes to environment variables that are made while the agent is running won't be picked up and used by any task. If you have sensitive environment variables that change and you don't want them to be stored as capabilities, you can have them ignored by setting the `VSO_AGENT_IGNORE` environment variable, with a comma-delimited list of variables to ignore. For example, `PATH` is a critical variable that you might want to ignore if you're installing software.

When you author a pipeline, you specify certain **demands** of the agent. The system sends the job only to agents that have capabilities matching the demands specified in the pipeline. As a result, agent capabilities allow you to direct jobs to specific agents.

> **Note**
>
> Demands and capabilities are designed for use with self-hosted agents so that jobs can be matched with an agent that meets the requirements of the job. When using Microsoft-hosted agents, you select an image for the agent that matches the requirements of the job, so although it is possible to add capabilities to a Microsoft-hosted agent, you don't need to use capabilities with Microsoft-hosted agents.

Configure demands

YAML

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

YAML

```yaml
pool:
  name: Default
  demands: SpecialSoftware # exists check for SpecialSoftware
```

You can check for the existence of a capability, or make a comparison with the value of a capability. For more information, see YAML schema - Demands.

Configure agent capabilities

Browser

You can view the details of an agent, including its version and system capabilities, and manage its user capabilities, by navigating to **Agent pools** and selecting the **Capabilities** tab for the desired agent.

1. In your web browser, navigate to Agent pools:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

   b. Select **Agents** and choose the desired agent.

c. Choose the **Capabilities** tab.

> ⓘ **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

3. To register a new capability with the agent, choose **Add a new capability**.

> **Tip**
>
> After you install new software on a self-hosted agent, you must restart the agent for the new capability to show up. For more information, see **Restart Windows agent**, **Restart Linux agent**, and **Restart Mac agent**.

Communication

Communication with Azure Pipelines

The agent communicates with Azure Pipelines or Azure DevOps Server to determine which job it needs to run, and to report the logs and job status. This communication is always initiated by the agent. All the messages from the agent to Azure Pipelines or Azure DevOps Server happen over HTTP or HTTPS, depending on how you configure the agent. This pull model allows the agent to be configured in different topologies as shown below.

Here is a common communication pattern between the agent and Azure Pipelines or Azure DevOps Server.

1. The user registers an agent with Azure Pipelines or Azure DevOps Server by adding it to an agent pool. You need to be an agent pool administrator to register an agent in that agent pool. The identity of agent pool administrator is needed only at the time of registration and is not persisted on the agent, nor is it used in any further communication between the agent and Azure Pipelines or Azure DevOps Server. Once the registration is complete, the agent downloads a *listener OAuth token* and uses it to listen to the job queue.

2. The agent listens to see if a new job request has been posted for it in the job queue in Azure Pipelines/Azure DevOps Server using an HTTP long poll. When a job is available, the agent downloads the job as well as a *job-specific OAuth token*. This token is generated by Azure Pipelines/Azure DevOps Server for the scoped identity specified in the pipeline. That token is short lived and is used by the agent to access resources (for example, source code) or modify resources (for example, upload test results) on Azure Pipelines or Azure DevOps Server within that job.

3. After the job is completed, the agent discards the job-specific OAuth token and goes back to checking if there is a new job request using the listener OAuth token.

The payload of the messages exchanged between the agent and Azure Pipelines/Azure DevOps Server are secured using asymmetric encryption. Each agent has a public-private key pair, and the public key is exchanged with the server during registration. The server uses the public key to encrypt the payload of the job before sending it to the agent. The agent decrypts the job content using its private key. This is how secrets stored in pipelines or variable groups are secured as they are exchanged with the agent.

> **Note**
>
> The agent provides support for UTF-8 client encoding output. However, if your system has a different encoding from UTF-8, you might encounter some problems with the output of logs. For example, the logs might contain characters that aren't recognized by your system's encoding so they might appear as garbled or missing symbols.

Communication to deploy to target servers

When you use the agent to deploy artifacts to a set of servers, it must have "line of sight" connectivity to those servers. The Microsoft-hosted agent pools, by default, have connectivity to Azure websites and servers running in Azure.

> **Note**
>
> If your Azure resources are running in an Azure Virtual Network, you can get the **Agent IP ranges** where Microsoft-hosted agents are deployed so you can configure the firewall rules for your Azure VNet to allow access by the agent.

If your on-premises environments do not have connectivity to a Microsoft-hosted agent pool (which is typically the case due to intermediate firewalls), you'll need to manually

configure a self-hosted agent on on-premises computer(s). The agents must have connectivity to the target on-premises environments, and access to the Internet to connect to Azure Pipelines or Team Foundation Server, as shown in the following schematic.

Authentication

To register an agent, you need to be a member of the administrator role in the agent pool. The identity of agent pool administrator is needed only at the time of registration and is not persisted on the agent, and is not used in any subsequent communication between the agent and Azure Pipelines or Azure DevOps Server. In addition, you must be a local administrator on the server in order to configure the agent.

When you register an agent, choose from the following authentication types, and agent setup prompts you for the specific additional information required for each authentication type. For more information, see Self-hosted agent authentication options.

- **PAT**
- **AAD** (device code flow authentication)
- **SP** (Service Principal authentication)

The authentication method used for registering the agent is used only during agent registration. To learn more about how agents communicate with Azure Pipelines after registration, see Communication with Azure Pipelines or TFS.

Interactive vs. service

You can run your self-hosted agent as either a service or an interactive process. After you've configured the agent, we recommend you first try it in interactive mode to make sure it works. Then, for production use, we recommend you run the agent in one of the following modes so that it reliably remains in a running state. These modes also ensure that the agent starts automatically if the machine is restarted.

1. **As a service**. You can leverage the service manager of the operating system to manage the lifecycle of the agent. In addition, the experience for auto-upgrading the agent is better when it is run as a service.

2. **As an interactive process with auto-logon enabled**. In some cases, you might need to run the agent interactively for production use - such as to run UI tests. When the agent is configured to run in this mode, the screen saver is also disabled. Some domain policies might prevent you from enabling auto-logon or disabling the screen saver. In such cases, you might need to seek an exemption from the domain policy, or run the agent on a workgroup computer where the domain policies do not apply.

> **Note**
>
> There are security risks when you enable automatic logon or disable the screen saver because you enable other users to walk up to the computer and use the account that automatically logs on. If you configure the agent to run in this way, you must ensure the computer is physically protected; for example, located in a secure facility. If you use Remote Desktop to access the computer on which an agent is running with auto-logon, simply closing the Remote Desktop causes the computer to be locked and any UI tests that run on this agent might fail. To avoid this, use the **tscon** command to disconnect from Remote Desktop. For example:
>
> `%windir%\System32\tscon.exe 1 /dest:console`

Agent account

Whether you run an agent as a service or interactively, you can choose which computer account you use to run the agent. (Note that this is different from the credentials that you use when you register the agent with Azure Pipelines or Azure DevOps Server.) The choice of agent account depends solely on the needs of the tasks running in your build and deployment jobs.

For example, to run tasks that use Windows authentication to access an external service, you must run the agent using an account that has access to that service. However, if you are running UI tests such as Selenium or Coded UI tests that require a browser, the browser is launched in the context of the agent account.

On Windows, you should consider using a service account such as Network Service or Local Service. These accounts have restricted permissions and their passwords don't expire, meaning the agent requires less management over time.

Agent version and upgrades

We update the agent software every few weeks in Azure Pipelines. We indicate the agent version in the format `{major}.{minor}`. For instance, if the agent version is `2.1`, then the major version is 2 and the minor version is 1.

Microsoft-hosted agents are always kept up-to-date. If the newer version of the agent is only different in *minor* version, self-hosted agents can usually be updated automatically (configure this setting in **Agent pools**, select your agent, **Settings** - the default is enabled) by Azure Pipelines. An upgrade is requested when a platform feature or one of the tasks used in the pipeline requires a newer version of the agent.

If you run a self-hosted agent interactively, or if there is a newer *major* version of the agent available, then you might have to manually upgrade the agents. You can do this easily from the **Agent pools** tab under your organization. Your pipelines won't run until they can target a compatible agent.

To update self-hosted agents

1. Navigate to **Project settings**, **Agent pools**.

2. Select your agent pool and choose **Update all agents**.

You can also update agents individually by choosing **Update agent** from the ... menu.

3. Select **Update** to confirm the update.

4. An update request is queued for each agent in the pool, and runs when any currently running jobs complete. Upgrading typically only takes a few moments - long enough to download the latest version of the agent software (approximately

200 MB), unzip it, and restart the agent with the new version. You can monitor the status of your agents on the **Agents** tab.

You can view the version of an agent by navigating to **Agent pools** and selecting the **Capabilities** tab for the desired agent, as described in Configure agent capabilities.

To trigger agent update programmatically you can use Agent update API as described in section How can I trigger agent updates programmatically for specific agent pool?.

> **Note**
>
> For servers with no internet access, manually copy the agent zip file to the following folder to use as a local file. Create the **Agents** folder if it is not present.
>
> - Windows: `%ProgramData%\Microsoft\Azure DevOps\Agents`
> - Linux: `usr/share/Microsoft/Azure DevOps/Agents`
> - macOS: `usr/share/Microsoft/Azure DevOps/Agents`

Create the **Agents** folder if it is not present.

FAQ

How do I make sure I have the latest v2 agent version?

1. Navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

c. Choose **Agent pools**.

2. Click the pool that contains the agent.

3. Make sure the agent is enabled.

4. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

   b. Select **Agents** and choose the desired agent.

c. Choose the **Capabilities** tab.

> ⓘ **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

5. Look for the `Agent.Version` capability. You can check this value against the latest published agent version. See Azure Pipelines Agent and check the page for the

highest version number listed.

6. Each agent automatically updates itself when it runs a task that requires a newer version of the agent. If you want to manually update some agents, right-click the pool, and select **Update all agents**.

Do self-hosted agents have any performance advantages over Microsoft-hosted agents?

In many cases, yes. Specifically:

- If you use a self-hosted agent, you can run incremental builds. For example, if you define a pipeline that does not clean the repo and does not perform a clean build, your builds will typically run faster. When you use a Microsoft-hosted agent, you don't get these benefits (unless you use features such as caching) because the agent is destroyed after the build or release pipeline is completed.
- A Microsoft-hosted agent can take longer to start your build. While it often takes just a few seconds for your job to be assigned to a Microsoft-hosted agent, it can sometimes take several minutes for an agent to be allocated depending on the load on our system.

Can I install multiple self-hosted agents on the same machine?

Yes. This approach can work well for agents that run jobs that don't consume many shared resources. For example, you could try it for agents that run releases that mostly orchestrate deployments and don't do much work on the agent itself.

You might find that in other cases you don't gain much efficiency by running multiple agents on the same machine. For example, it might not be worthwhile for agents that run builds that consume much disk and I/O resources.

You might also run into problems if parallel build jobs are using the same singleton tool deployment, such as npm packages. For example, one build might update a dependency while another build is in the middle of using it, which could cause unreliable results and errors.

What’s the behavior of agents when the pipeline jobs are canceled?

For Microsoft-hosted agents, the agent is torn down and returned to the Azure Pipelines pool.

For self-hosted agents:

When a pipeline is canceled, the agent sends a sequence of commands to the process executing the current step. The first command is sent with a timeout of 7.5 seconds. If the process has not terminated, a second command is sent with a timeout of 2.5 seconds. If the process has not terminated, the agent issues a command to kill the process. If the process does not honor the two initial termination requests, it will be killed. From the initial request to termination takes approximately 10 seconds.

The commands issued to the process to cancel the pipeline differ based on the agent operating system.

- macOS and Linux - The commands sent are SIGINT, followed by SIGTERM, followed by SIGKILL.
- Windows - The commands sent to the process are Ctrl+C, followed by Ctrl+Break, followed by Process.Kill.

How can I trigger agent updates programmatically for specific agent pool?

You can trigger agent updates for the pool by using the following API:

```
POST
https://dev.azure.com/{organization}/_apis/distributedtask/pools/{poolId}/me
ssages?agentId={agentId}&api-version=6.0
```

URI Parameters

Name	In	Required	Type	Description
`agentId`	query	False	string	The agent to update. If not specified - update will be triggered for all agents.
`organization`	path	True	string	The name of the Azure DevOps organization.
`poolId`	path	True	integer int32	The agent pool to use
`api-version`	query	False	string	Version of the API to use. This should be set to

Name	In	Required	Type	Description
				'6.0' to use this version of the api.

To trigger agent update - request body should be empty.

> **Note**
>
> Azure Pipelines Agent is open source on **GitHub**.

Learn more

For more information about agents, see the following modules from the Build applications with Azure DevOps learning path.

- Choose a Microsoft-hosted or self-hosted build agent
- Host your own build agent in Azure Pipelines

Agent software version 3

Article • 11/27/2023

The pipelines team is upgrading the agent software from version 2.x (using .NET Core 3.1) to version 3.x (using .NET 6). The new agent version supports new Apple silicon hardware and newer operating systems like Ubuntu 22.04, or Windows on ARM64.

Upgrade to 3.x agent on supported operating systems

If you're running your self-hosted agents on newer operating systems supported by .NET 6, the upgrade to the new agent version is automatic.

The following operating systems are supported by the 3.x agent.

- **Linux**
  - x64
    - CentOS 7, 8
    - Debian 10+
    - Fedora 36+
    - openSUSE 15+
    - Red Hat Enterprise Linux 7+
      - No longer requires separate package
    - SUSE Enterprise Linux 12 SP2 or later
    - Ubuntu 22.04, 20.04, 18.04, 16.04
    - Azure Linux 2.0
  - ARM64
    - Debian 10+
    - Ubuntu 22.04, 20.04, 18.04
  - Alpine x64
    - Alpine Linux 3.13 and higher (requires agent 3.227 or higher)
- **macOS**
  - x64
    - macOS 10.15 "Catalina"
    - macOS 11.0 "Big Sur"
    - macOS 12.0 "Monterey"
    - macOS 13.0 "Ventura"
  - ARM64
    - macOS 11.0 "Big Sur"

    - macOS 12.0 "Monterey"
    - macOS 13.0 "Ventura"
    - Note: Not all Azure Pipelines tasks have been updated to support ARM64 yet
- **Windows**
  - Client OS
    - Windows 7 SP1 ESU
    - Windows 8.1
    - Windows 10
    - Windows 11
  - Server OS
    - Windows Server 2012 or higher

Upgrade to 3.x agent on unsupported operating systems

If you're running your self-hosted agents on an operating system that isn't supported by .NET 6, you must update your machines to use a newer supported operating system supported by .NET 6.

The following list of operating systems are commonly used for self-hosted 2.x agents. These operating systems aren't supported by .NET 6 and can't be used to run the new .NET 6 based version 3.x agent.

Expand table

System/Distribution	Version not supported by .NET 6
CentOS	< 7
Debian	<= 4.9
Fedora	<= 32
RedHat Enterprise Linux	<= 6
Ubuntu	< 18.04 LTS
macOS	< 10.15

You can use a script to predict whether the agents in your self-hosted pools are able to upgrade from 2.x to 3.x.

When attempting to run pipelines on agent version 2.218 (or 2.214 on RHEL 6), pipelines running on one of the unsupported operating systems listed here will fail with

following error message: `This operating system will stop receiving updates of the Pipelines Agent in the future. To be able to continue to run pipelines please upgrade the operating system or set an environment variable or agent knob "AGENT_ACKNOWLEDGE_NO_UPDATES" to "true". See https://aka.ms/azdo-pipeline-agent-v2-eos for more information.`

To resolve this error you can:

1. Upgrade or move your agent machines to one of the supported operating systems listed previously in this article. This is the preferred solution and allows you to get future agent updates,
2. Set an `AGENT_ACKNOWLEDGE_NO_UPDATES` variable on the agent, either by setting an environment variable or a pipeline variable.

- You can set `AGENT_ACKNOWLEDGE_NO_UPDATES` by configuring an environment variable on the agent, for example in **/etc/environment** or **etc/profile.d**: `AGENT_ACKNOWLEDGE_NO_UPDATES=true`.

- You can set a pipeline variable.

```yml
jobs:
- job: 'agentWithVariables'
  displayName: 'Agent with variables'

  variables:
    AGENT_ACKNOWLEDGE_NO_UPDATES: 'true' # Required to not fail job on operating system that is not supported by .NET 6
```

FAQ

What is the difference between the 2.x and 3.x agents?

The 2.x agents (for example 2.212) are .NET Core 3.1 and the 3.x agents (for example 3.212) are .NET 6. During Phase I and II, both versions are available, with the 3.x versions being in prerelease.

How can I check my agents to see if they can upgrade to 3.x?

You can use a script to predict whether the agents in your self-hosted pools will be able to upgrade from 2.x to 3.x.

How will security issues in the agent be patched going forward?

When the .NET 6 agent becomes generally available for self-hosted pools in Q1 2023, there will be no patches done, in general, for the 2.x agents. The patches will be done only for the 3.x agents. However, we also have Azure DevOps Server customers that will still be relying on 2.x agents. So, we'll review the security issues on a case by case basis to decide.

What do I need to do when I'm on an unsupported OS?

You should migrate to a newer operating system that is supported by .NET 6 now. Otherwise, your agent may attempt to upgrade, and it will fail as .NET 6 can't be installed on your OS. We'll publish some guidance in a follow-up blog post that will prevent auto-upgrades of the agent. However, that is only meant to be a temporary solution to give you some more time to upgrade your agent machines.

Can I stay on 2.x agents if I'm not working on any changes in my project anymore?

No. The pipelines team is regularly adding new features to Azure Pipelines and some of them may require an update to the agent even though your pipeline doesn't explicitly depend on that feature. When you prevent auto-upgrades of the agent using the guidance in a follow-up blog, that agent can't be used to schedule the pipeline. If no agent with the required capabilities can be found, the pipeline execution will fail.

Do I have to install .NET 6 before installing the 3.x agent software?

You don't have to install .NET 6 on your agent machine before installing and configuring the 3.x agent software. All .NET dependencies the 3.x agent requires are part of the agent itself.

Do I have to build my code using .NET 6 if I'm using the 3.x agent?

The version of .NET used to run the 3.x agent is self-contained in the agent installation, and isn't used to build your code. The version of .NET that is used to build your code depends on the pipeline and the version or versions of .NET you have installed onto your agent machine.

I use Azure DevOps Server and not Azure DevOps Service. Does this change impact me?

No. The new agent is only applicable for Azure DevOps Service customers at this time. However, a future version of Azure DevOps Server will include the new agent. The pipelines team recommends that you update your agent machines to newer operating systems that are supported by .NET 6 starting now, if you plan to keep up with the Azure DevOps Server releases in the future.

What is the timeline for agent version 3 deployment?

Agent version 3 was released March 2023.

What will happen when a task requires an agent to be updated to agent version 3?

Normally, when a task requires a newer version of the agent, it will automatically update itself. For now, while agent version 2 continues to be updated, we have disabled auto update from agent version 2 to agent version 3. Once we enable it, for Operating Systems that aren't compatible with agent version 3, agent version 2.217 and newer won't attempt to update itself to the v3 agent. Instead, a warning will be shown informing users they need to upgrade the Operating System first: `The operating system the agent is running on is <OS>, which will not be supported by the .NET 6 based v3 agent. Please upgrade the operating system of this host to ensure compatibility with the v3 agent. See https://aka.ms/azdo-pipeline-agent-version`

Create and manage agent pools

Article • 06/20/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

An agent pool is a collection of agents. Instead of managing each agent individually, you organize agents into agent pools. When you configure an agent, it is registered with a single pool, and when you create a pipeline, you specify the pool in which the pipeline runs. When you run the pipeline, it runs on an agent from that pool that meets the demands of the pipeline.

In Azure Pipelines, pools are scoped to the entire organization; so you can share the agent machines across projects.

> **Note**
>
> Agent pool jobs run a job on a single agent. If you need to run a job on all agents, such as a deployment group for classic release pipelines, see **Provision deployment groups**.

If you are an organization administrator, you create and manage agent pools from the agent pools tab in admin settings.

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).
2. Choose **Azure DevOps**, **Organization settings**.

3. Choose **Agent pools**.

If you are a project team member, you create and manage agent pools from the agent pools tab in project settings.

Navigate to your project and choose **Project settings**, **Agent pools**.

Default agent pools

The following agent pools are provided by default:

- **Default** pool: Use it to register self-hosted agents that you've set up.
- **Azure Pipelines** hosted pool with various Windows, Linux, and macOS images. For a complete list of the available images and their installed software, see Microsoft-hosted agents.

> ⓘ Note

> The Azure Pipelines hosted pool replaces the previous hosted pools that had names that mapped to the corresponding images. Any jobs you had in the previous hosted pools are automatically redirected to the correct image in the new Azure Pipelines hosted pool. In some circumstances, you may still see the old pool names, but behind the scenes the hosted jobs are run using the Azure Pipelines pool. For more information, see the **Single hosted pool** release notes from the **July 1 2019 - Sprint 154 release notes**.

By default, all contributors in a project are members of the **User** role on hosted pools. This allows every contributor in a project to author and run pipelines using Microsoft-hosted agents.

Designate a pool in your pipeline

YAML

To choose a Microsoft-hosted agent from the Azure Pipelines pool in your Azure DevOps Services YAML pipeline, specify the name of the image, using the **YAML VM Image Label** from this table.

YAML

```yaml
pool:
  vmImage: ubuntu-latest # This is the default if you don't specify a pool or vmImage.
```

To use a private pool with no demands:

YAML

```yaml
pool: MyPool
```

For more information, see the YAML schema for pools.

Manage pools and queues

Browser

If you are an organization administrator, you create and manage agent pools from the agent pools tab in admin settings.

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

2. Choose **Azure DevOps**, **Organization settings**.

3. Choose **Agent pools**.

If you are a project team member, you create and manage agent pools from the agent pools tab in project settings.

Navigate to your project and choose **Project settings**, **Agent pools**.

Pools are used to run jobs. Learn about specifying pools for jobs.

If you've got a lot of self-hosted agents intended for different teams or purposes, you might want to create additional pools as explained below.

Create agent pools

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

- 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. First, make sure you've got the permissions to create pools in your project by selecting **Security** on the agent pools page in your **Project settings**. You must have **Administrator** role to be able to create new pools. Next, select **Add pool** and select the option to create a **new** pool. Finally install and configure agents to be part of that agent pool.

- You're a member of the infrastructure team and would like to set up a pool of agents for use in all projects. First, make sure you've got the permissions to create pools in your project by selecting **Security** on the agent pools page in your **Organization settings**. Next create a **New agent pool** and select the option to **Auto-provision this agent pool in all projects** while creating the pool. This setting ensures all projects have access to this agent pool. Finally install and configure agents to be part of that agent pool.

- You want to share a set of agent machines with multiple projects, but not all of them. First, navigate to the settings for one of the projects, add an agent pool, and select the option to create a **new** pool at the organization level. Next, 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**. Finally, install and configure agents to be part of the shared agent pool.

Security of agent pools

Understanding how security works for agent pools helps you control sharing and use of agents.

Roles are defined on each agent pool, and **membership** in these roles governs what operations you can perform on an agent pool.

Organization-level security settings

Role on an agent pool in organization settings	Purpose
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.
Service Account	Members of this role can use the organization agent pool to create a project agent pool in a project. If you follow the guidelines above for creating new project agent pools, you typically do not have to add any members here.
Administrator	In addition to all the above permissions, members of this role can register or unregister agents from the organization agent pool. They can also refer to the organization agent pool when creating a project agent pool in a project. 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.

The **All agent pools** node in the Agent Pools tab is used to control the security of *all* organization agent pools. Role memberships for individual organization agent pools are automatically inherited from those of the 'All agent pools' node. By default, TFS and Azure DevOps Server administrators are also administrators of the 'All agent pools' node when using TFS or Azure DevOps Server.

Project-level security settings

Roles are also defined on each project agent pool, and memberships in these roles govern what operations you can perform on an agent pool at the project level.

Role on an agent pool in project settings	Purpose
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.
User	Members of this role can use the project agent pool when authoring pipelines.
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.

Pipeline permissions

Pipeline permissions control which YAML pipelines are authorized to use an agent pool. Pipeline permissions do not restrict access from Classic pipelines.

You can choose from the following options:

- Open access for all pipelines to use the agent pool from the more options at top-right corner of the **Pipeline permissions** section in security tab of an agent pool.
- Lock down the agent pool and only allow selected YAML pipelines to use it. If any other YAML pipeline refers to the agent pool, an authorization request gets raised, which must be approved by an agent pool Administrator. This does not limit access from Classic pipelines.

Pipeline permissions for the *Azure Pipelines* agent pool cannot be configured, as the pool is accessible, by default, to all pipelines.

The **Security** action in the Agent pools tab is used to control the security of *all* project agent pools in a project. Role memberships for individual project agent pools are

automatically inherited from what you define here. By default, the following groups are added to the Administrator role of 'All agent pools': Build Administrators, Release Administrators, Project Administrators.

FAQ

If I don't schedule a maintenance window, when will the agents run maintenance?

If no window is scheduled, then the agents in that pool will not run the maintenance job.

What is a maintenance job?

You can configure agent pools to periodically clean stale working directories and repositories. This should reduce the potential for the agents to run out of disk space. Maintenance jobs are configured at the project collection or organization level in agent pool settings.

To configure maintenance job settings:

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

2. Choose **Azure DevOps**, **Organization settings**.

3. Choose **Agent pools**.

Choose the desired pool and choose **Settings** to configure maintenance job settings for that agent pool.

> **ⓘ Important**
>
> You must have the **Manage build queues** permission to configure maintenance job settings. If you don't see the **Settings** tab or the **Maintenance History** tab, you don't have that permission, which is granted by default to the **Administrator** role. For more information, see **Security of agent pools**.

Configure your desired settings and choose **Save**.

Select **Maintenance History** to see the maintenance job history for the current agent pool. You can download and review logs to see the cleaning steps and actions taken.

The maintenance is done per agent pool, not per machine; so if you have multiple agent pools on a single machine, you may still run into disk space issues.

The maintenance job of my self-hosted agent pool looks stuck. Why?

Typically, a maintenance job gets "stuck" when it's waiting to run on an agent that is no longer in the agent pool. This happens when, for example, the agent has been purposefully taken offline or when there are issues communicating with it.

Maintenance jobs that have been queued to run will wait seven days to run. Afterward, they'll be automatically set to failed state if not run. This time limit cannot be changed.

The seven-day limit is different from the *maintenance job timeout* setting. The latter controls the maximum number of minutes an agent can spend doing maintenance. The timer starts when the job starts, not when the job is queued on an agent.

I'm trying to create a project agent pool that uses an existing organization agent pool, but the controls are grayed out. Why?

On the 'Create a project agent pool' dialog box, you can't use an existing organization agent pool if it is already referenced by another project agent pool. Each organization agent pool can be referenced by only one project agent pool within a given project collection.

I can't select a Microsoft-hosted pool and I can't queue my build. How do I fix this?

Ask the owner of your Azure DevOps organization to grant you permission to use the pool. See Security of agent pools.

I need more hosted build resources. What can I do?

A: The Azure Pipelines pool provides all Azure DevOps organizations with cloud-hosted build agents and free build minutes each month. If you need more Microsoft-hosted build resources, or need to run more jobs in parallel, then you can either:

- Host your own agents on infrastructure that you manage
- Buy additional parallel jobs

Historical graph for agent pools (Preview)

Article • 01/26/2023

Azure DevOps Services

The pool consumption report enables you to view jobs running in your agent pools graphed with agent pool job concurrency over a span of up to 30 days. You can use this information to help decide whether your jobs aren't running because of concurrency limits. If you have many jobs queued or running jobs at the concurrency or online agents limit, you may wish to purchase additional parallel jobs or provision more self-hosted agents.

Prerequisites

> **Important**
>
> You must be a member of the **Project Collection Administrators** group to view the pool consumption reports for agent pools in an organization, including project level reports in that organization.

Pool consumption report

The pool consumption report is part of the **Analytics** tab for an agent pool and contains the following charts, depending on the agent pool type.

Chart type	Description	Agent pool type
Public hosted concurrency	Displays concurrency, queued jobs, and running jobs for public projects	Microsoft-hosted
Private hosted concurrency	Displays concurrency, queued jobs, and running jobs for private projects	Microsoft-hosted
Agent usage	Displays online agents, queued jobs, and running jobs for self-hosted agents	Scale set agent and self-hosted
Private self-hosted concurrency	Displays concurrency, queued jobs, and running jobs for private self-hosted projects	Scale set agent and self-hosted

The charts in the pool consumption report graph the following data points:

- **Concurrency** - The number of parallel jobs in the organization that apply to the project type (public or private) and agent pool type (Microsoft-hosted or self-hosted). For more information, see Configure and pay for parallel jobs.
- **Online agents** - The number of agents online in a self-hosted agent pool or a scale set agent pool.
- **Queued jobs** - The number of jobs queued and waiting for an agent.
- **Running jobs** - The number of running jobs.

Pool data is aggregated at a granularity of 10 minutes, and the number of running jobs is plotted based on the maximum number of running jobs for the specified interval of time. Because multiple short-running jobs may complete within the 10 minute timeline, the count of running jobs may sometimes be higher than the concurrency or online agents during that same period.

Report scope

The pool consumption report can be displayed at organization scope, or project scope. At the organization level, the chart is plotted using data from pipelines across any project within the organization that have run jobs in that pool. At the project level, the chart is plotted using data from pipelines in that particular project that have run jobs in that pool.

- To view the pool consumption report at the organization level, choose **Organization settings**, **Pipelines**, **Agent pools**.
- To view the pool consumption report at the project level, navigate to the desired project and choose **Project settings**, **Pipelines**, **Agent pools**.

From the **Agent pools** view, choose the desired pool, and view the **Analytics** tab. The following example shows the pool consumption report for a self-hosted agent pool.

This example shows the usage graphs for the **Azure Pipelines** Microsoft-hosted agent pool.

Filtering

To adjust the timeline of the graph, choose **Filter** , select the interval drop-down, and choose the desired interval.

For the 1 day interval, you can view data per hour, and for the other intervals you can view it per day. Pool data is aggregated at a granularity of 10 minutes, and the number of running jobs is plotted based on the maximum number of running jobs for the specified interval of time. In this example there are two online agents, but in some areas there are four running jobs due to the way the pool data is aggregated.

FAQ

Where does the pool consumption report get the data it displays?

The pool consumption report uses the Azure DevOps Analytics service and the `TaskAgentRequestSnapshots` endpoint. You can query this endpoint using the following URL prefix: `https://analytics.dev.azure.com/{organization}/{project_id}/_odata/v4.0-preview/TaskAgentRequestSnapshots`.

You can retrieve the `project_id` for your project by navigating to the following URL: `https://dev.azure.com/{organization}/_apis/projects?api-version=5.0-preview.3`.

The following example shows a sample query and response.

```
{
"@odata.context":
"https://analytics.dev.azure.com/{org}/{project_id}/_odata/v4.0-
preview/$metadata#TaskAgentRequestSnapshots",
"vsts.warnings@odata.type": "#Collection(String)",
"@vsts.warnings": [
"VS403507: The specified query does not include a $select or $apply clause
which is recommended for all queries. Details on recommended query patterns
are available here: https://go.microsoft.com/fwlink/?linkid=861060."
],
"value": [
{
"SamplingDateSK": 20201117,
"SamplingHour": 13,
"SamplingTime": "2020-11-17T13:10:00-08:00",
"QueuedDate": "2020-11-17T13:07:26.22-08:00",
"QueuedDateSK": 20201117,
"StartedDate": "2020-11-17T15:02:23.7398429-08:00",
"StartedDateSK": 20201117,
```

```
"FinishedDate": "2020-11-17T15:13:49.89-08:00",
"FinishedDateSK": 20201117,
"QueueDurationSeconds": 6897.519,
"ProjectSK": "...",
"PipelineSK": 5141,
"RequestId": 6313,
"PoolId": 28,
"PipelineType": "Build",
"IsHosted": true,
"IsRunning": false,
"IsQueued": true
},
...
```


For more information on query options, see Query guidelines for Analytics with OData.

> **Note**
>
> The `TaskAgentRequestSnapshots` endpoint is in preview and not yet documented but you can view information about the data returned by navigating to the endpoint URL: `https://analytics.dev.azure.com/{org}/{project_id}/_odata/v4.0-preview/TaskAgentRequestSnapshots`.

Why are there more running jobs than there are agents or concurrency?

Pool data is aggregated at a granularity of 10 minutes, and the number of running jobs is plotted based on the maximum number of running jobs for the specified interval of time. Each running job is counted separately, and if multiple jobs complete during the 10 minute interval they contribute to the total count of running jobs for that interval.

What is the difference between viewing the graphs in Project settings vs Organization settings?

The scope of the data in the graph is determined based on whether the chart is accessed through **Project settings** or **Organization settings**. At the organization level, the chart is plotted using data from pipelines across any project within the organization that have run jobs in that pool. At the project level, the chart is plotted using data from pipelines in that particular project that have run jobs in that pool.

Microsoft-hosted agents

Article • 11/16/2023

Azure DevOps Services

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. You always get the latest version of the VM image you specify in your pipeline. Each time you run a pipeline, you get a fresh virtual machine for each job in the pipeline. The virtual machine is discarded after one job (which means any change that a job makes to the virtual machine file system, such as checking out code, will be unavailable to the next job). Microsoft-hosted agents can run jobs directly on the VM or in a container.

Azure Pipelines provides a predefined agent pool named **Azure Pipelines** with Microsoft-hosted agents.

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 scale set agents or a self-hosted agent.

> **Tip**
>
> You can try a Microsoft-hosted agent for no charge.

Software

The **Azure Pipelines** agent pool offers several virtual machine images to choose from, each including a broad range of tools and software.

Image	Classic Editor Agent Specification	YAML VM Image Label	Included Software
Windows Server 2022 with Visual Studio 2022	*windows-2022*	`windows-latest` OR `windows-2022`	Link
Windows Server 2019 with Visual Studio 2019	*windows-2019*	`windows-2019`	Link
Ubuntu 22.04	*ubuntu-22.04*	`ubuntu-latest` OR `ubuntu-22.04`	Link
Ubuntu 20.04	*ubuntu-20.04*	`ubuntu-20.04`	Link

Image	Classic Editor Agent Specification	YAML VM Image	Included Software
macOS 13 Ventura	*macOS-13*	`macOS-13`	Link
macOS 12 Monterey	*macOS-12*	`macOS-latest` OR `macOS-12`	Link
macOS 11 Big Sur	*macOS-11*	`macOS-11`	Link

The default agent image for classic build pipelines is *windows-2019*, and the default agent image for YAML build pipelines is `ubuntu-latest`. For more information, see Designate a pool in your pipeline.

You can see the installed software for each hosted agent by choosing the **Included Software** link in the table. When using macOS images, you can manually select from tool versions. Read more.

Recent updates

- The macOS 13 image is available
- The macOS 10.15 image is fully unsupported as of 4/24/2023
- Ubuntu 18.04 has been retired
- ubuntu-latest images use ubuntu-22.04.
- General availability of Ubuntu 22.04 for Azure Pipelines hosted pools.
- The Ubuntu 18.04 image will begin deprecation on 8/8/22 and will be fully unsupported by 4/1/2023.
- The macOS 10.15 image will begin deprecation on 5/31/22 and will be fully unsupported by 12/1/2022.
- windows-latest images use windows-2022.
- macOS-latest images use macOS-11.
- The Ubuntu 16.04 hosted image was removed September 2021.
- The Windows Server 2016 with Visual Studio 2017 image has been deprecated and will be retired June 30 2022. Read this blog post on how to identify pipelines using deprecated images.
- In December 2021, we removed the following Azure Pipelines hosted image:
  - macOS X Mojave 10.14 (`macOS-10.14`)
- In March 2020, we removed the following Azure Pipelines hosted images:
  - Windows Server 2012R2 with Visual Studio 2015 (`vs2015-win2012r2`)
  - macOS X High Sierra 10.13 (`macOS-10.13`)
  - Windows Server Core 1803 (`win1803`)

Customers are encouraged to migrate to newer versions or a self-hosted agent.

For more information and instructions on how to update your pipelines that use those images, see Removing older images in Azure Pipelines hosted pools.

> **Note**
>
> The Azure Pipelines hosted pool replaces the previous hosted pools that had names that mapped to the corresponding images. Any jobs you had in the previous hosted pools are automatically redirected to the correct image in the new Azure Pipelines hosted pool. In some circumstances, you may still see the old pool names, but behind the scenes the hosted jobs are run using the Azure Pipelines pool. For more information about this update, see the **Single hosted pool** release notes from the **July 1 2019 - Sprint 154 release notes**.

> **Important**
>
> To request additional software to be installed on Microsoft-hosted agents, don't create a feedback request on this document or open a support ticket. Instead, open an issue on our **repository**, where we manage the scripts to generate various images.

How to identify pipelines using a deprecated hosted image

To identify pipelines that are using a deprecated image, browse to the following location in your organization: `https://dev.azure.com/{organization}/{project}/_settings/agentqueues`, and filter on the image name to check. The following example checks the `vs2017-win2016` image.

You can also query job history for deprecated images across projects using the script located here, as shown in the following example.

PowerShell

```
./QueryJobHistoryForRetiredImages.ps1 -accountUrl
https://dev.azure.com/{org} -pat {pat}
```

Use a Microsoft-hosted agent

YAML

In YAML pipelines, if you do not specify a pool, pipelines default to the Azure Pipelines agent pool. You simply need to specify which virtual machine image you want to use.

YAML

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

```
    vmImage: 'macOS-latest'
  steps:
  - script: echo hello from macOS
- job: Windows
  pool:
    vmImage: 'windows-latest'
  steps:
  - script: echo hello from Windows
```

> **Note**
>
> The specification of a pool can be done at multiple levels in a YAML file. If you notice that your pipeline is not running on the expected image, make sure that you verify the pool specification at the pipeline, stage, and job levels.

Avoid hard-coded references

When you use a Microsoft-hosted agent, always use variables to refer to the build environment and agent resources. For example, don't hard-code the drive letter or folder that contains the repository. The precise layout of the hosted agents is subject to change without warning.

Hardware

Microsoft-hosted agents that run Windows and Linux images are provisioned on Azure general purpose virtual machines with a 2 core CPU, 7 GB of RAM, and 14 GB of SSD disk space. These virtual machines are co-located in the same geography as your Azure DevOps organization.

Agents that run macOS images are provisioned on Mac pros with a 3 core CPU, 14 GB of RAM, and 14 GB of SSD disk space. These agents always run in the US irrespective of the location of your Azure DevOps organization. If data sovereignty is important to you and if your organization is not in the US, then you should not use macOS images. Learn more.

All of these machines have at least 10 GB of free disk space available for your pipelines to run. This free space is consumed when your pipeline checks out source code, downloads packages, pulls docker images, or generates intermediate files.

> **Important**

We cannot honor requests to increase disk space on Microsoft-hosted agents, or to provision more powerful machines. If the specifications of Microsoft-hosted agents do not meet your needs, then you should consider **self-hosted agents** or **scale set agents**.

Networking

In some setups, you may need to know the range of IP addresses where agents are deployed. For instance, if you need to grant the hosted agents access through a firewall, you may wish to restrict that access by IP address. Because Azure DevOps uses the Azure global network, IP ranges vary over time. Microsoft publishes a weekly JSON file listing IP ranges for Azure datacenters, broken out by region. This file is updated weekly with new planned IP ranges. Only the latest version of the file is available for download. If you need previous versions, you must download and archive them each week as they become available. The new IP ranges become effective the following week. We recommend that you check back frequently (at least once every week) to ensure you keep an up-to-date list. If agent jobs begin to fail, a key first troubleshooting step is to make sure your configuration matches the latest list of IP addresses. The IP address ranges for the hosted agents are listed in the weekly file under `AzureCloud.<region>`, such as `AzureCloud.westus` for the West US region.

Your hosted agents run in the same Azure geography as your organization. Each geography contains one or more regions. While your agent may run in the same region as your organization, it is not guaranteed to do so. To obtain the complete list of possible IP ranges for your agent, you must use the IP ranges from all of the regions that are contained in your geography. For example, if your organization is located in the **United States** geography, you must use the IP ranges for all of the regions in that geography.

To determine your geography, navigate to `https://dev.azure.com/<your_organization>/_settings/organizationOverview`, get your region, and find the associated geography from the Azure geography table. Once you have identified your geography, use the IP ranges from the weekly file for all regions in that geography.

Important

You cannot use private connections such as **ExpressRoute** or VPN to connect Microsoft-hosted agents to your corporate network. The traffic between Microsoft-hosted agents and your servers will be over public network.

To identify the possible IP ranges for Microsoft-hosted agents

1. Identify the region for your organization in **Organization settings**.
2. Identify the Azure Geography for your organization's region.
3. Map the names of the regions in your geography to the format used in the weekly file, following the format of `AzureCloud.<region>`, such as `AzureCloud.westus`. You can map the names of the regions from the Azure Geography list to the format used in the weekly file by reviewing the region names passed to the constructor of the regions defined in the source code for the Region class, from the Azure Management Libraries for .NET.

   > **Note**
   >
   > Since there is no API in the **Azure Management Libraries for .NET** to list the regions for a geography, you must list them manually as shown in the following example.

4. Retrieve the IP addresses for all regions in your geography from the weekly file. If your region is **Brazil South** or **West Europe**, you must include additional IP ranges based on your fallback geography, as described in the following note.

> **Note**
>
> Due to capacity restrictions, some organizations in the **Brazil South** or **West Europe** regions may occasionally see their hosted agents located outside their expected geography. In these cases, in addition to including the IP ranges for all the regions in your geography as described in the previous section, additional IP ranges must be included for the regions in the capacity fallback geography.
>
> If your organization is in the **Brazil South** region, your capacity fallback geography is **United States**.
>
> If your organization is in the **West Europe** region, the capacity fallback geography is **France**.

Our Mac IP ranges are not included in the Azure IPs above, as they are hosted in GitHub's macOS cloud. IP ranges can be retrieved using the **GitHub metadata API** using the instructions provided **here**.

Example

In the following example, the hosted agent IP address ranges for an organization in the West US region are retrieved from the weekly file. Since the West US region is in the United States geography, the IP addresses for all regions in the United States geography are included. In this example, the IP addresses are written to the console.

C#

```
using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;

namespace WeeklyFileIPRanges
{
    class Program
    {
        // Path to the locally saved weekly file
        const string weeklyFilePath =
@"C:\MyPath\ServiceTags_Public_20230904.json";

        static void Main(string[] args)
        {
            // United States geography has the following regions:
            // Central US, East US, East US 2, East US 3, North Central US,
            // South Central US, West Central US, West US, West US 2, West
US 3
            // This list is accurate as of 9/8/2023
            List<string> USGeographyRegions = new List<string>
            {
                "centralus",
                "eastus",
                "eastus2",
                "eastus3",
                "northcentralus",
                "southcentralus",
                "westcentralus",
                "westus",
                "westus2",
                "westus3"
            };

            // Load the weekly file
            JObject weeklyFile =
```

```
JObject.Parse(File.ReadAllText(weeklyFilePath));
            JArray values = (JArray)weeklyFile["values"];

            foreach (string region in USGeographyRegions)
            {
                string tag = $"AzureCloud.{region}";
                Console.WriteLine(tag);

                var ipList =
                    from v in values
                    where tag.Equals((string)v["name"],
StringComparison.OrdinalIgnoreCase)
                    select v["properties"]["addressPrefixes"];

                foreach (var ip in ipList.Children())
                {
                    Console.WriteLine(ip);
                }
            }
        }
    }
}
```

Service tags

Microsoft-hosted agents can't be listed by service tags. If you're trying to grant hosted agents access to your resources, you'll need to follow the IP range allow listing method.

Security

Microsoft-hosted agents run on secure Azure platform. However, you must be aware of the following security considerations.

- Although Microsoft-hosted agents run on Azure public network, they are not assigned public IP addresses. So, external entities cannot target Microsoft-hosted agents.
- Microsoft-hosted agents are run in individual VMs, which are re-imaged after each run. Each agent is dedicated to a single organization, and each VM hosts only a single agent.
- There are several benefits to running your pipeline on Microsoft-hosted agents, from a security perspective. If you run untrusted code in your pipeline, such as contributions from forks, it is safer to run the pipeline on Microsoft-hosted agents than on self-hosted agents that reside in your corporate network.
- When a pipeline needs to access your corporate resources behind a firewall, you have to allow the IP address range for the Azure geography. This may increase

your exposure as the range of IP addresses is rather large and since machines in this range can belong to other customers as well. The best way to prevent this is to avoid the need to access internal resources. For information on deploying artifacts to a set of servers, see Communication to deploy to target servers.
- Hosted images do not conform to CIS hardening benchmarks. To use CIS-hardened images, you must create either self-hosted agents or scale-set agents.

Capabilities and limitations

Microsoft-hosted agents:

- Have the above software. You can also add software during your build or release using tool installer tasks.
  - You get a freshly imaged agent for each job in your pipeline.
- Provide 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). Buy Microsoft-hosted parallel jobs.
  - When you create a new Azure DevOps organization, you are not given these free grants by default. To request the free grant for public or private projects, submit a request.
- Run on Microsoft Azure general purpose virtual machines Standard_DS2_v2.
- Run as an administrator on Windows and a passwordless sudo user on Linux.
- (Linux only) Run steps in a `cgroup` that offers 6 GB of physical memory and 13 GB of total memory.
- Use VM images that are regularly updated (every 3 weeks).

Microsoft-hosted agents do not offer:

- The ability to remotely connect.
- The ability to drop artifacts to a UNC file share.
- The ability to join machines directly to your corporate network.
- The ability to get bigger or more powerful build machines.
- The ability to pre-load custom software. You can install software during a pipeline run, such as through tool installer tasks or in a script.

- Potential performance advantages that you might get by using self-hosted agents that might start and run builds faster. Learn more
- The ability to run XAML builds.
- The ability to roll back to a previous VM image version. You always use the latest version.

If Microsoft-hosted agents don't meet your needs, then you can deploy your own self-hosted agents or use scale set agents.

FAQ

How can I see what software is included in an image?

You can see the installed software for each hosted agent by choosing the **Included Software** link in the Software table.

> **Note**
>
> By default, the Windows agent uses the version of Git that is bundled with the agent software. Microsoft recommends using the version of Git that is bundled with the agent, but you have several options to override this default behavior and use the version of Git that the agent machine has installed in the path.
>
> - Set a pipeline variable named `System.PreferGitFromPath` to `true` in your pipelines.
> - On self-hosted agents, you can create a file named **.env** in the agent root directory and add a `System.PreferGitFromPath=true` line to the file. For more information, see **How do I set different environment variables for each individual agent?**
>
> To see the version of Git used by a pipeline, you can look at the logs for a `checkout` step in your pipeline, as shown in the following example.
>
> ```
> Syncing repository: PathFilter (Git)
> Prepending Path environment variable with directory containing
> 'git.exe'.
> git version
> git version 2.26.2.windows.1
> ```

How does Microsoft choose the software and versions to put on the image?

More information about the versions of software included on the images can be found at Guidelines for what's installed.

When are the images updated?

Images are typically updated weekly. You can check the status badges which are in the format `20200113.x` where the first part indicates the date the image was updated.

What can I do if software I need is removed or replaced with a newer version?

You can let us know by filing a GitHub issue by choosing the **Included Software** links in the Use a Microsoft-hosted agent table.

You can also use a self-hosted agent that includes the exact versions of software that you need. For more information, see Self-hosted agents.

What if I need a bigger machine with more processing power, memory, or disk space?

We can't increase the memory, processing power, or disk space for Microsoft-hosted agents, but you can use self-hosted agents or scale set agents hosted on machines with your desired specifications.

I can't select a Microsoft-hosted agent and I can't queue my build or deployment. What should I do?

Microsoft-hosted agents are only available in Azure Pipelines and not in TFS or Azure DevOps Server.

By default, all project contributors in an organization have access to the Microsoft-hosted agents. But, your organization administrator may limit the access of Microsoft-hosted agents to select users or projects. Ask the owner of your Azure DevOps organization to grant you permission to use a Microsoft-hosted agent. See agent pool security.

My pipelines running on Microsoft-hosted agents take more time to complete. How can I speed them up?

If your pipeline has recently become slower, review our status page for any outages. We could be having issues with our service. Or else, review any changes that you made in your application code or pipeline. Your repository size during check-out might have increased, you may be uploading larger artifacts, or you may be running more tests.

If you are just setting up a pipeline and are comparing the performance of Microsoft-hosted agents to your local machine or a self-hosted agent, then note the specifications of the hardware that we use to run your jobs. We are unable to provide you with bigger or powerful machines. You can consider using self-hosted agents or scale set agents if this performance is not acceptable.

I need more agents. What can I do?

All Azure DevOps organizations are provided with several free parallel jobs for open-source projects, and one free parallel job and limited minutes each month for private projects. If you need additional minutes or parallel jobs for your open-source project, contact support. If you need additional minutes or parallel jobs for your private project, then you can buy more.

My pipeline succeeds on self-hosted agent, but fails on Microsoft-hosted agents. What should I do?

Your self-hosted agent probably has all the right dependencies installed on it, whereas the same dependencies, tools, and software are not installed on Microsoft-hosted agents. First, carefully review the list of software that is installed on Microsoft-hosted agents by following the link to **Included software** in the table above. Then, compare that with the software installed on your self-hosted agent. In some cases, Microsoft-hosted agents may have the tools that you need (for example, Visual Studio), but all of the necessary optional components may not have been installed. If you find differences, then you have two options:

- You can create a new issue on the repository, where we track requests for additional software. Contacting support can't help you set up new software on Microsoft-hosted agents.
- You can use self-hosted agents or scale set agents. With these agents, you are fully in control of the images that are used to run your pipelines.

My build succeeds on my local machine, but fails on Microsoft-hosted agents. What should I do?

Your local machine probably has all the right dependencies installed on it, whereas the same dependencies, tools, and software are not installed on Microsoft-hosted agents. First, carefully review the list of software that is installed on Microsoft-hosted agents by following the link to **Included software** in the table above. Then, compare that with the software installed on your local machine. In some cases, Microsoft-hosted agents may have the tools that you need (e.g., Visual Studio), but all of the necessary optional components may not have been installed. If you find differences, then you have two options:

- You can create a new issue on the repository, where we track requests for additional software. This is your best bet for getting new software installed. Contacting support will not help you with setting up new software on Microsoft-hosted agents.
- You can use self-hosted agents or scale set agents. With these agents, you are fully in control of the images that are used to run your pipelines.

My pipeline fails with the error: "no space left on device".

Microsoft-hosted agents only have 10 GB of disk space available for running your job. This space is consumed when you check out source code, when you download packages, when you download docker images, or when you produce intermediate files. Unfortunately, we cannot increase the free space available on Microsoft-hosted images. You can restructure your pipeline so that it can fit into this space. Or, you can consider using self-hosted agents or scale set agents.

My pipeline running on Microsoft-hosted agents requires access to servers on our corporate network. How do we get a list of IP addresses to allow in our firewall?

See the section Agent IP ranges

Our pipeline running on Microsoft-hosted agents is unable to resolve the name of a server on our corporate network. How can we fix this?

If you refer to the server by its DNS name, then make sure that your server is publicly accessible on the Internet through its DNS name. If you refer to your server by its IP address, make sure that the IP address is publicly accessible on the Internet. In both cases, ensure that any firewall in between the agents and your corporate network has the agent IP ranges allowed.

I'm getting an SAS IP authorization error from an Azure Storage account

If you get an SAS error code, it is most likely because the IP address ranges from the Microsoft-hosted agents aren't permitted due to your Azure Storage rules. There are a few workarounds:

1. Manage the IP network rules for your Azure Storage account and add the IP address ranges for your hosted agents.
2. In your pipeline, use Azure CLI to update the network ruleset for your Azure Storage account right before you access storage, and then restore the previous ruleset.
3. Use self-hosted agents or Scale set agents.

How can I manually select versions of tools on the Hosted macOS agent?

Xamarin

Hosted macOS agent stores Xamarin SDK versions and the associated Mono versions as a set of symlinks to Xamarin SDK locations that are available by a single bundle symlink.

To manually select a Xamarin SDK version to use on the **Hosted macOS** agent, execute the following bash command before your Xamarin build task as a part of your build, specifying the symlink to Xamarin versions bundle that you need.

```
/bin/bash -c "sudo $AGENT_HOMEDIRECTORY/scripts/select-xamarin-sdk.sh <symlink>"
```

The list of all available Xamarin SDK versions and symlinks can be found in the agents documentation:

- macOS 11
- macOS 12

This command does not select the Mono version beyond the Xamarin SDK. To manually select a Mono version, see instructions below.

In case you are using a non-default version of Xcode for building your Xamarin.iOS or Xamarin.Mac apps, you should additionally execute this command line:

`/bin/bash -c "echo '##vso[task.setvariable variable=MD_APPLE_SDK_ROOT;]'$(xcodeRoot);sudo xcode-select --switch $(xcodeRoot)/Contents/Developer"`

where `$(xcodeRoot)` = `/Applications/Xcode_13.2.app`

Xcode versions on the **Hosted macOS** agent pool can be found here for the `macos-11` agent and here for the `macos-12` agent.

Xcode

If you use the Xcode task included with Azure Pipelines and TFS, you can select a version of Xcode in that task's properties. Otherwise, to manually set the Xcode version to use on the **Hosted macOS** agent pool, before your `xcodebuild` build task, execute this command line as part of your build, replacing the Xcode version number 13.2 as needed:

`/bin/bash -c "sudo xcode-select -s /Applications/Xcode_13.2.app/Contents/Developer"`

Xcode versions on the **Hosted macOS** agent pool can be found here for the `macos-11` agent and here for the `macos-12` agent.

This command does not work for Xamarin apps. To manually select an Xcode version for building Xamarin apps, see instructions above.

Mono

To manually select a Mono version to use on the **Hosted macOS** agent pool, execute this script in each job of your build before your Mono build task, specifying the symlink with the required Mono version (list of all available symlinks can be found in the Xamarin section above):

Bash

```
SYMLINK=<symlink>
MONOPREFIX=/Library/Frameworks/Mono.framework/Versions/$SYMLINK
echo "##vso[task.setvariable variable=DYLD_FALLBACK_LIBRARY_PATH;]$MONOPREFIX/lib:/lib:/usr/lib:$DYLD_LIBRARY_FALLBACK_PATH"
echo "##vso[task.setvariable
```

```
variable=PKG_CONFIG_PATH;]$MONOPREFIX/lib/pkgconfig:$MONOPREFIX/share/pkgcon
fig:$PKG_CONFIG_PATH"
echo "##vso[task.setvariable variable=PATH;]$MONOPREFIX/bin:$PATH"
```

Self-hosted Linux agents

Article • 11/27/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> **Important**
>
> This article provides guidance for using the **3.x agent software** with Azure DevOps Services. If you're using Azure DevOps Server or TFS, see **Self-hosted Linux agents (Agent version 2.x)**.

To run your jobs, you'll need at least one agent. A Linux agent can build and deploy different kinds of apps, including Java and Android apps. See Check prerequisites for a list of supported Linux distributions.

> **Note**
>
> This article describes how to configure a **self-hosted agent**. If you're using Azure DevOps Services and a **Microsoft-hosted agent** meets your needs, you can skip setting up a self-hosted Linux agent.

Learn about agents

If you already know what an agent is and how it works, feel free to jump right in to the following sections. But if you'd like some more background about what they do and how they work, see Azure Pipelines agents.

Check prerequisites

The agent is based on .NET 6. You can run this agent on several Linux distributions. We support the following subset of .NET 6 supported distributions:

- Supported distributions
  - x64
    - CentOS 7, 8
    - Debian 10+
    - Fedora 36+

    - openSUSE 15+
    - Red Hat Enterprise Linux 7+
      - No longer requires separate package
    - SUSE Enterprise Linux 12 SP2 or later
    - Ubuntu 22.04, 20.04, 18.04, 16.04
    - Azure Linux 2.0
  - ARM64
    - Debian 10+
    - Ubuntu 22.04, 20.04, 18.04
  - Alpine x64
    - Alpine Linux 3.13 and higher (requires agent 3.227 or higher)
- **Git** - Regardless of your platform, you will need to install Git 2.9.0 or higher. We strongly recommend installing the latest version of Git.
- **.NET** - The agent software runs on .NET 6, but installs its own version of .NET so there is no .NET prerequisite.
- **Subversion** - If you're building from a Subversion repo, you must install the Subversion client on the machine.
- **TFVC** - If you're building from a TFVC repo, see TFVC prerequisites.

> **Note**
>
> The agent installer knows how to check for other dependencies. You can install those dependencies on supported Linux platforms by running `./bin/installdependencies.sh` in the agent directory.
>
> Be aware that some of these dependencies required by .NET are fetched from third party sites, like `packages.efficios.com`. Review the `installdependencies.sh` script and ensure any referenced third party sites are accessible from your Linux machine before running the script.
>
> Please also make sure that all required repositories are connected to the relevant package manager used in `installdependencies.sh` (like `apt` or `zypper`).
>
> For issues with dependencies installation (like 'dependency was not found in repository' or 'problem retrieving the repository index file') - you can reach out to distribution owner for further support.

You should run agent setup manually the first time. After you get a feel for how agents work, or if you want to automate setting up many agents, consider using unattended config.

Prepare permissions

Information security for self-hosted agents

The user configuring the agent needs pool admin permissions, but the user running the agent does not.

The folders controlled by the agent should be restricted to as few users as possible and they contain secrets that could be decrypted or exfiltrated.

The Azure Pipelines agent is a software product designed to execute code it downloads from external sources. It inherently could be a target for Remote Code Execution (RCE) attacks.

Therefore, it is important to consider the threat model surrounding each individual usage of Pipelines Agents to perform work, and decide what are the minimum permissions could be granted to the user running the agent, to the machine where the agent runs, to the users who have write access to the Pipeline definition, the git repos where the yaml is stored, or the group of users who control access to the pool for new pipelines.

It is a best practice to have the identity running the agent be different from the identity with permissions to connect the agent to the pool. The user generating the credentials (and other agent-related files) is different than the user that needs to read them. Therefore, it is safer to carefully consider access granted to the agent machine itself, and the agent folders which contain sensitive files, such as logs and artifacts.

It makes sense to grant access to the agent folder only for DevOps administrators and the user identity running the agent process. Administrators may need to investigate the file system to understand build failures or get log files to be able to report Azure DevOps failures.

Decide which user you'll use

As a one-time step, you must register the agent. Someone with permission to administer the agent queue must complete these steps. The agent will not use this person's credentials in everyday operation, but they're required to complete registration. Learn more about how agents communicate.

Confirm the user has permission

Make sure the user account that you're going to use has permission to register the agent.

Is the user an Azure DevOps organization owner or TFS or Azure DevOps Server administrator? **Stop here**, you have permission.

Otherwise:

1. Open a browser and navigate to the **Agent pools** tab for your Azure Pipelines organization or Azure DevOps Server or TFS server:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Select the pool on the right side of the page and then click **Security**.

3. If the user account you're going to use is not shown, then get an administrator to add it. The administrator can be an agent pool administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   If it's a deployment group agent, the administrator can be a deployment group administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   You can add a user to the deployment group administrator role in the **Security** tab on the **Deployment Groups** page in **Azure Pipelines**.

> ⓘ **Note**
>
> If you see a message like this: **Sorry, we couldn't add the identity. Please try a different identity.**, you probably followed the above steps for an organization owner or TFS or Azure DevOps Server administrator. You don't need to do anything; you already have permission to administer the agent pool.

Download and configure the agent

Azure Pipelines

1. Log on to the machine using the account for which you've prepared permissions as explained above.

2. In your web browser, sign in to Azure Pipelines, and navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

c. Choose **Agent pools**.

3. Select the **Default** pool, select the **Agents** tab, and choose **New agent**.

4. On the **Get the agent** dialog box, click **Linux**.

5. On the left pane, select the specific flavor. We offer x64 or ARM for many Linux distributions.

6. On the right pane, click the **Download** button.

7. Follow the instructions on the page.

8. Unpack the agent into the directory of your choice. `cd` to that directory and run `./config.sh`.

Server URL

Azure Pipelines: `https://dev.azure.com/{your-organization}`

Authentication type

When you register an agent, choose from the following authentication types, and agent setup prompts you for the specific additional information required for each authentication type. For more information, see Self-hosted agent authentication options.

- Personal access token
- Device code flow
- Service principal

Run interactively

For guidance on whether to run the agent in interactive mode or as a service, see Agents: Interactive vs. service.

To run the agent interactively:

1. If you have been running the agent as a service, uninstall the service.

2. Run the agent.

```bash
./run.sh
```

To restart the agent, press Ctrl+C and then run `run.sh` to restart it.

To use your agent, run a job using the agent's pool. If you didn't choose a different pool, your agent will be in the **Default** pool.

Run once

For agents configured to run interactively, you can choose to have the agent accept only one job. To run in this configuration:

```bash
```

```
./run.sh --once
```

Agents in this mode will accept only one job and then spin down gracefully (useful for running in Docker on a service like Azure Container Instances).

Run as a systemd service

If your agent is running on these operating systems you can run the agent as a `systemd` service:

- Ubuntu 16 LTS or newer
- Red Hat 7.1 or newer

We provide an example `./svc.sh` script for you to run and manage your agent as a `systemd` service. This script will be generated after you configure the agent. We encourage you to review, and if needed, update the script before running it.

Some important caveats:

- If you run your agent as a service, you cannot run the agent service as `root` user.
- Users running SELinux have reported difficulties with the provided `svc.sh` script. Refer to this agent issue as a starting point. SELinux is not an officially supported configuration.

> **Note**
>
> If you have a different distribution, or if you prefer other approaches, you can use whatever kind of service mechanism you prefer. See **Service files**.

Commands

Change to the agent directory

For example, if you installed in the `myagent` subfolder of your home directory:

Bash

```bash
cd ~/myagent$
```

Install

Command:

```bash
sudo ./svc.sh install [username]
```

This command creates a service file that points to `./runsvc.sh`. This script sets up the environment (more details below) and starts the agents host. If `username` parameter is not specified then the username is taken from the $SUDO_USER environment variable which is set by sudo command. This variable is always equal to the name of the user who invoked the `sudo` command.

Start

```bash
sudo ./svc.sh start
```

Status

```bash
sudo ./svc.sh status
```

Stop

```bash
sudo ./svc.sh stop
```

Uninstall

> You should stop before you uninstall.

```bash
sudo ./svc.sh uninstall
```

Update environment variables

When you configure the service, it takes a snapshot of some useful environment variables for your current logon user such as PATH, LANG, JAVA_HOME, ANT_HOME, and MYSQL_PATH. If you need to update the variables (for example, after installing some new software):

```bash
./env.sh
sudo ./svc.sh stop
sudo ./svc.sh start
```

The snapshot of the environment variables is stored in `.env` file (`PATH` is stored in `.path`) under agent root directory, you can also change these files directly to apply environment variable changes.

Run instructions before the service starts

You can also run your own instructions and commands to run when the service starts. For example, you could set up the environment or call scripts.

1. Edit `runsvc.sh`.
2. Replace the following line with your instructions:

   ```bash
   # insert anything to setup env when running as a service
   ```

Service files

When you install the service, some service files are put in place.

systemd service file

A systemd service file is created:

`/etc/systemd/system/vsts.agent.{tfs-name}.{agent-name}.service`

For example, you have configured an agent (see above) with the name `our-linux-agent`. The service file will be either:

- **Azure Pipelines**: the name of your organization. For example if you connect to `https://dev.azure.com/fabrikam`, then the service name would be `/etc/systemd/system/vsts.agent.fabrikam.our-linux-agent.service`
- **TFS or Azure DevOps Server**: the name of your on-premises server. For example if you connect to `http://our-server:8080/tfs`, then the service name would be `/etc/systemd/system/vsts.agent.our-server.our-linux-agent.service`

`sudo ./svc.sh install` generates this file from this template: `./bin/vsts.agent.service.template`

.service file

`sudo ./svc.sh start` finds the service by reading the `.service` file, which contains the name of systemd service file described above.

Alternative service mechanisms

We provide the `./svc.sh` script as a convenient way for you to run and manage your agent as a systemd service. But you can use whatever kind of service mechanism you prefer (for example: initd or upstart).

You can use the template described above as to facilitate generating other kinds of service files.

Use a cgroup to avoid agent failure

It's important to avoid situations in which the agent fails or become unusable because otherwise the agent can't stream pipeline logs or report pipeline status back to the server. You can mitigate the risk of this kind of problem being caused by high memory pressure by using cgroups and a lower `oom_score_adj`. After you've done this, Linux reclaims system memory from pipeline job processes before reclaiming memory from the agent process. Learn how to configure cgroups and OOM score.

Replace an agent

To replace an agent, follow the *Download and configure the agent* steps again.

When you configure an agent using the same name as an agent that already exists, you're asked if you want to replace the existing agent. If you answer `Y`, then make sure

you remove the agent (see below) that you're replacing. Otherwise, after a few minutes of conflicts, one of the agents will shut down.

Remove and reconfigure an agent

To remove the agent:

1. Stop and uninstall the service as explained in the previous section.
2. Remove the agent.

   ```bash
   ./config.sh remove
   ```

3. Enter your credentials.

After you've removed the agent, you can configure it again.

Unattended config

The agent can be set up from a script with no human intervention. You must pass `--unattended` and the answers to all questions.

To configure an agent, it must know the URL to your organization or collection and credentials of someone authorized to set up agents. All other responses are optional. Any command-line parameter can be specified using an environment variable instead: put its name in upper case and prepend `VSTS_AGENT_INPUT_`. For example, `VSTS_AGENT_INPUT_PASSWORD` instead of specifying `--password`.

Required options

- `--unattended` - agent setup will not prompt for information, and all settings must be provided on the command line
- `--url <url>` - URL of the server. For example: https://dev.azure.com/myorganization or http://my-azure-devops-server:8080/tfs
- `--auth <type>` - authentication type. Valid values are:
  - `pat` (Personal access token) - PAT is the only scheme that works with Azure DevOps Services.
  - `negotiate` (Kerberos or NTLM)

- `alt` (Basic authentication)
- `integrated` (Windows default credentials)

Authentication options

- If you chose `--auth pat`:
  - `--token <token>` - specifies your personal access token
  - PAT is the only scheme that works with Azure DevOps Services.
- If you chose `--auth negotiate` or `--auth alt`:
  - `--userName <userName>` - specifies a Windows username in the format `domain\userName` or `userName@domain.com`
  - `--password <password>` - specifies a password

Pool and agent names

- `--pool <pool>` - pool name for the agent to join
- `--agent <agent>` - agent name
- `--replace` - replace the agent in a pool. If another agent is listening by the same name, it will start failing with a conflict

Agent setup

- `--work <workDirectory>` - work directory where job data is stored. Defaults to `_work` under the root of the agent directory. The work directory is owned by a given agent and should not be shared between multiple agents.
- `--acceptTeeEula` - accept the Team Explorer Everywhere End User License Agreement (macOS and Linux only)
- `--disableloguploads` - don't stream or send console log output to the server. Instead, you may retrieve them from the agent host's filesystem after the job completes.

Windows-only startup

- `--runAsService` - configure the agent to run as a Windows service (requires administrator permission)
- `--runAsAutoLogon` - configure auto-logon and run the agent on startup (requires administrator permission)
- `--windowsLogonAccount <account>` - used with `--runAsService` or `--runAsAutoLogon` to specify the Windows user name in the format `domain\userName` or

`userName@domain.com`

- `--windowsLogonPassword <password>` - used with `--runAsService` or `--runAsAutoLogon` to specify Windows logon password (not required for Group Managed Service Accounts and Windows built in accounts such as 'NT AUTHORITY\NETWORK SERVICE')
- `--overwriteAutoLogon` - used with `--runAsAutoLogon` to overwrite the existing auto logon on the machine
- `--noRestart` - used with `--runAsAutoLogon` to stop the host from restarting after agent configuration completes

Troubleshooting configuring the agent with the `runAsAutoLogon` option

Configuring the agent with the `runAsAutoLogon` option runs the agent each time after restarting the machine. Perform next steps if the agent is not run after restarting the machine.

If the agent was already configured on the machine

Before reconfiguring the agent, it is necessary to remove the old agent configuration, so try to run this command from the agent folder:

```
.\config.cmd remove --auth 'PAT' --token '<token>'
```

Check if the agent was removed from your agent pool after executing the command:

```
<Azure DevOps organization> / <Project> / Settings / Agent pools / <Agent
Pool> / Agents
```

Remove the agent from your agent pool manually if it was not removed by running the command.

Then try to reconfigure the agent by running this command from the agent folder:

```
.\config.cmd --unattended --agent '<agent-name>' --pool '<agent-pool-name>'
--url '<azure-dev-ops-organization-url>' --auth 'PAT' --token '<token>' --
```

```
runAsAutoLogon --windowsLogonAccount '<domain\user-name>' --
windowsLogonPassword '<windows-password>'
```

Specify the agent name (any specific unique name) and check if this agent appeared in your agent pool after reconfiguring.

It will be much better to unpack an agent archive (which can be downloaded here) and run this command from the new unpacked agent folder.

Check if the Windows registry key is recorded and saved correctly

Run the `whoami /user` command to get the `<sid>`. Open `Registry Editor` and follow the path:

```
Computer\HKEY_USERS\<sid>\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

Check if there is the `VSTSAgent` key. Delete this key if it exists, then close `Registry Editor` and configure the agent by running the `.\config.cmd` command (without args) from the agent folder. Before answering the question `Enter Restart the machine at a later time?`, open `Registry Editor` again and check if the `VSTSAgent` key has appeared. Press `Enter` to answer the question, and check if the `VSTSAgent` key remains in its place after restarting the machine.

Check if Windows registry keys work fine on your machine

Create a `autorun.cmd` file that contains the following line: `echo "Hello from AutoRun!"`. Open `Registry Editor` and create in the path above a new key-value pair with the key `AutoRun` and the value

```
C:\windows\system32\cmd.exe /D /S /C start "AutoRun"
"D:\path\to\autorun.cmd"
```

Restart your machine. You have an issue with Windows registry keys if you do not see a console window with the `Hello from AutoRun!` message.

Deployment group only

- `--deploymentGroup` - configure the agent as a deployment group agent
- `--deploymentGroupName <name>` - used with `--deploymentGroup` to specify the deployment group for the agent to join
- `--projectName <name>` - used with `--deploymentGroup` to set the project name
- `--addDeploymentGroupTags` - used with `--deploymentGroup` to indicate that deployment group tags should be added
- `--deploymentGroupTags <tags>` - used with `--addDeploymentGroupTags` to specify the comma separated list of tags for the deployment group agent - for example "web, db"

Environments only

- `--addvirtualmachineresourcetags` - used to indicate that environment resource tags should be added
- `--virtualmachineresourcetags <tags>` - used with `--addvirtualmachineresourcetags` to specify the comma separated list of tags for the environment resource agent - for example "web, db"

`./config.sh --help` always lists the latest required and optional responses.

Diagnostics

If you're having trouble with your self-hosted agent, you can try running diagnostics. After configuring the agent:

```bash
./run.sh --diagnostics
```

This will run through a diagnostic suite that may help you troubleshoot the problem. The diagnostics feature is available starting with agent version 2.165.0.

Network diagnostics for self-hosted agents

Set the value of `Agent.Diagnostic` to `true` to collect additional logs that can be used for troubleshooting network issues for self-hosted agents. For more information, see Network diagnostics for self-hosted agents.

Help on other options

To learn about other options:

```bash
./config.sh --help
```

The help provides information on authentication alternatives and unattended configuration.

Capabilities

Your agent's capabilities are cataloged and advertised in the pool so that only the builds and releases it can handle are assigned to it. See Build and release agent capabilities.

In many cases, after you deploy an agent, you'll need to install software or utilities. Generally you should install on your agents whatever software and tools you use on your development machine.

For example, if your build includes the npm task, then the build won't run unless there's a build agent in the pool that has npm installed.

> **Important**
>
> Capabilities include all environment variables and the values that are set when the agent runs. If any of these values change while the agent is running, the agent must be restarted to pick up the new values. After you install new software on an agent, you must restart the agent for the new capability to show up in the pool, so that the build can run.
>
> If you want to exclude environment variables as capabilities, you can designate them by setting an environment variable `VSO_AGENT_IGNORE` with a comma-delimited list of variables to ignore.

FAQ

Where can I learn more about the new v3 agent software?

For information and FAQs about the v3 agent software, see Agent software version 3.

How do I make sure I have the latest agent version?

1. Navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Click the pool that contains the agent.

3. Make sure the agent is enabled.

4. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

b. Select **Agents** and choose the desired agent.

c. Choose the **Capabilities** tab.

> **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

5. Look for the `Agent.Version` capability. You can check this value against the latest published agent version. See Azure Pipelines Agent and check the page for the highest version number listed.

6. Each agent automatically updates itself when it runs a task that requires a newer version of the agent. If you want to manually update some agents, right-click the pool, and select **Update all agents**.

Why is sudo needed to run the service commands?

`./svc.sh` uses `systemctl`, which requires `sudo`.

Source code: systemd.svc.sh.template on GitHub

I'm running a firewall and my code is in Azure Repos. What URLs does the agent need to communicate with?

If you're running an agent in a secure network behind a firewall, make sure the agent can initiate communication with the following URLs and IP addresses.

Domain URL	Description
`https://{organization_name}.pkgs.visualstudio.com`	Azure DevOps Packaging API for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsrm.visualstudio.com`	Release Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://*.blob.core.windows.net`	Azure Artifacts
`https://*.dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://*.vsassets.io`	Azure Artifacts via CDN
`https://*.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `dev.azure.com` domain
`https://*.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `dev.azure.com` domain
`https://*.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the

Domain URL	Description
	`dev.azure.com` domain
`https://app.vssps.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://login.microsoftonline.com`	Microsoft Entra sign-in
`https://management.core.windows.net`	Azure Management APIs
`https://vstsagentpackage.azureedge.net`	Agent package

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> For more information about allowed addresses, see **Allowed address lists and network connections**.

How do I run the agent with self-signed certificate?

Run the agent with self-signed certificate

How do I run the agent behind a web proxy?

Run the agent behind a web proxy

How do I restart the agent

If you are running the agent interactively, see the restart instructions in Run interactively. If you are running the agent as a systemd service, follow the steps to Stop and then Start the agent.

How do I configure the agent to bypass a web proxy and connect to Azure Pipelines?

If you want the agent to bypass your proxy and connect to Azure Pipelines directly, then you should configure your web proxy to enable the agent to access the following URLs.

For organizations using the `*.visualstudio.com` domain:

```
https://login.microsoftonline.com
https://app.vssps.visualstudio.com
https://{organization_name}.visualstudio.com
https://{organization_name}.vsrm.visualstudio.com
https://{organization_name}.vstmr.visualstudio.com
https://{organization_name}.pkgs.visualstudio.com
https://{organization_name}.vssps.visualstudio.com
```

For organizations using the `dev.azure.com` domain:

```
https://dev.azure.com
https://*.dev.azure.com
https://login.microsoftonline.com
https://management.core.windows.net
https://vstsagentpackage.azureedge.net
https://vssps.dev.azure.com
```

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> This procedure enables the agent to bypass a web proxy. Your build pipeline and scripts must still handle bypassing your web proxy for each task and tool you run in your build.
>
> For example, if you are using a NuGet task, you must configure your web proxy to support bypassing the URL for the server that hosts the NuGet feed you're using.

TFVC prerequisites

If you'll be using TFVC, you'll also need the Oracle Java JDK 1.6 or higher. (The Oracle JRE and OpenJDK aren't sufficient for this purpose.)

TEE plugin is used for TFVC functionality. It has an EULA, which you'll need to accept during configuration if you plan to work with TFVC.

Since the TEE plugin is no longer maintained and contains some out-of-date Java dependencies, starting from Agent 2.198.0 it's no longer included in the agent distribution. However, the TEE plugin will be downloaded during checkout task execution if you're checking out a TFVC repo. The TEE plugin will be removed after the job execution.

> **Note**
>
> Note: You may notice your checkout task taking a long time to start working because of this download mechanism.

If the agent is running behind a proxy or a firewall, you'll need to ensure access to the following site: `https://vstsagenttools.blob.core.windows.net/`. The TEE plugin will be downloaded from this address.

If you're using a self-hosted agent and facing issues with TEE downloading, you may install TEE manually:

1. Set `DISABLE_TEE_PLUGIN_REMOVAL` environment or pipeline variable to `true`. This variable prevents the agent from removing the TEE plugin after TFVC repository checkout.
2. Download TEE-CLC version 14.135.0 manually from Team Explorer Everywhere GitHub releases.
3. Extract the contents of `TEE-CLC-14.135.0` folder to `<agent_directory>/externals/tee`.

Self-hosted macOS agents

Article • 10/10/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> **Important**
>
> This article provides guidance for using the **3.x agent software** with Azure DevOps Services. If you're using Azure DevOps Server or TFS, see **Self-hosted macOS agents (Agent version 2.x)**.

To build and deploy Xcode apps or Xamarin.iOS projects, you need at least one macOS agent. This agent can also build and deploy Java and Android apps.

> **Note**
>
> This article describes how to configure a **self-hosted agent**. If you're using Azure DevOps Services and a **Microsoft-hosted agent** meets your needs, you can skip setting up a self-hosted macOS agent.

Learn about agents

If you already know what an agent is and how it works, feel free to jump right in to the following sections. But if you'd like some more background about what they do and how they work, see Azure Pipelines agents.

Check prerequisites

- Supported operating systems
  - x64
    - macOS 10.15 "Catalina"
    - macOS 11.0 "Big Sur"
    - macOS 12.0 "Monterey"
    - macOS 13.0 "Ventura"
  - ARM64
    - macOS 11.0 "Big Sur"
    - macOS 12.0 "Monterey"

    - macOS 13.0 "Ventura"
    - Note: Not all Azure Pipelines tasks have been updated to support ARM64 yet

- **Git** - Git 2.9.0 or higher (latest version recommended - you can easily install with Homebrew)

- **.NET** - The agent software runs on .NET 6, but installs its own version of .NET so there is no .NET prerequisite.
- **TFVC** - If you're building from a TFVC repo, see TFVC prerequisites.

Prepare permissions

If you're building from a Subversion repo, you must install the Subversion client on the machine.

You should run agent setup manually the first time. After you get a feel for how agents work, or if you want to automate setting up many agents, consider using unattended config.

Information security for self-hosted agents

The user configuring the agent needs pool admin permissions, but the user running the agent does not.

The folders controlled by the agent should be restricted to as few users as possible and they contain secrets that could be decrypted or exfiltrated.

The Azure Pipelines agent is a software product designed to execute code it downloads from external sources. It inherently could be a target for Remote Code Execution (RCE) attacks.

Therefore, it is important to consider the threat model surrounding each individual usage of Pipelines Agents to perform work, and decide what are the minimum permissions could be granted to the user running the agent, to the machine where the agent runs, to the users who have write access to the Pipeline definition, the git repos where the yaml is stored, or the group of users who control access to the pool for new pipelines.

It is a best practice to have the identity running the agent be different from the identity with permissions to connect the agent to the pool. The user generating the credentials (and other agent-related files) is different than the user that needs to read them. Therefore, it is safer to carefully consider access granted to the agent machine itself, and the agent folders which contain sensitive files, such as logs and artifacts.

It makes sense to grant access to the agent folder only for DevOps administrators and the user identity running the agent process. Administrators may need to investigate the file system to understand build failures or get log files to be able to report Azure DevOps failures.

Decide which user you'll use

As a one-time step, you must register the agent. Someone with permission to administer the agent queue must complete these steps. The agent will not use this person's credentials in everyday operation, but they're required to complete registration. Learn more about how agents communicate.

Confirm the user has permission

Make sure the user account that you're going to use has permission to register the agent.

Is the user an Azure DevOps organization owner or TFS or Azure DevOps Server administrator? **Stop here**, you have permission.

Otherwise:

1. Open a browser and navigate to the **Agent pools** tab for your Azure Pipelines organization or Azure DevOps Server or TFS server:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Select the pool on the right side of the page and then click **Security**.

3. If the user account you're going to use is not shown, then get an administrator to add it. The administrator can be an agent pool administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   If it's a deployment group agent, the administrator can be a deployment group administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   You can add a user to the deployment group administrator role in the **Security** tab on the **Deployment Groups** page in **Azure Pipelines**.

> ⓘ **Note**
>
> If you see a message like this: **Sorry, we couldn't add the identity. Please try a different identity.**, you probably followed the above steps for an organization owner or TFS or Azure DevOps Server administrator. You don't need to do anything; you already have permission to administer the agent pool.

Download and configure the agent

Azure Pipelines

1. Log on to the machine using the account for which you've prepared permissions as explained in the previous section.

2. In your web browser, sign in to Azure Pipelines, and navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

3. Select the **Default** pool, select the **Agents** tab, and choose **New agent**.

4. On the **Get the agent** dialog box, click **macOS**.

5. Click the **Download** button.

6. Follow the instructions on the page.

7. Clear the extended attribute on the tar file: `xattr -c vsts-agent-osx-x64-V.v.v.tar.gz`.

8. Unpack the agent into the directory of your choice. `cd` to that directory and run `./config.sh`. Make sure that the path to the directory contains no spaces because tools and scripts don't always properly escape spaces.

Server URL

Azure Pipelines: `https://dev.azure.com/{your-organization}`

Authentication type

When you register an agent, choose from the following authentication types, and agent setup prompts you for the specific additional information required for each authentication type. For more information, see Self-hosted agent authentication options.

- Personal access token
- Device code flow
- Service principal

Run interactively

For guidance on whether to run the agent in interactive mode or as a service, see Agents: Interactive vs. service.

To run the agent interactively:

1. If you have been running the agent as a service, uninstall the service.

2. Run the agent.

```bash
./run.sh
```

To restart the agent, press Ctrl+C and then run `run.sh` to restart it.

To use your agent, run a job using the agent's pool. If you didn't choose a different pool, your agent will be in the **Default** pool.

Run once

For agents configured to run interactively, you can choose to have the agent accept only one job. To run in this configuration:

```bash
./run.sh --once
```

Agents in this mode accept only one job and then spin down gracefully (useful for running on a service like Azure Container Instances).

Run as a launchd service

We provide the `./svc.sh` script for you to run and manage your agent as a launchd LaunchAgent service. This script will be generated after you configure the agent. The service has access to the UI to run your UI tests.

> **Note**
>
> If you prefer other approaches, you can use whatever kind of service mechanism you prefer. See **Service files**.

Tokens

In the section below, these tokens are replaced:

- `{agent-name}`
- `{tfs-name}`

For example, you have configured an agent (see above) with the name `our-osx-agent`. In the following examples, `{tfs-name}` will be either:

- Azure Pipelines: the name of your organization. For example if you connect to `https://dev.azure.com/fabrikam`, then the service name would be `vsts.agent.fabrikam.our-osx-agent`
- TFS: the name of your on-premises TFS AT server. For example if you connect to `http://our-server:8080/tfs`, then the service name would be `vsts.agent.our-server.our-osx-agent`

Commands

Change to the agent directory

For example, if you installed in the `myagent` subfolder of your home directory:

```bash
cd ~/myagent$
```

Install

Command:

```bash
./svc.sh install
```

This command creates a launchd plist that points to `./runsvc.sh`. This script sets up the environment (more details below) and starts the agent's host.

Start

Command:

```bash
./svc.sh start
```

Output:

```bash
starting vsts.agent.{tfs-name}.{agent-name}
status vsts.agent.{tfs-name}.{agent-name}:

/Users/{your-name}/Library/LaunchAgents/vsts.agent.{tfs-name}.{agent-name}.plist

Started:
13472 0 vsts.agent.{tfs-name}.{agent-name}
```

The left number is the pid if the service is running. If second number is not zero, then a problem occurred.

Status

Command:

```bash
./svc.sh status
```

Output:

```bash
status vsts.agent.{tfs-name}.{agent-name}:

/Users/{your-name}/Library/LaunchAgents/vsts.{tfs-name}.{agent-name}.testsvc.plist

Started:
13472 0 vsts.agent.{tfs-name}.{agent-name}
```

The left number is the pid if the service is running. If second number is not zero, then a problem occurred.

Stop

Command:

```bash
./svc.sh stop
```

Output:

```bash
stopping vsts.agent.{tfs-name}.{agent-name}
status vsts.agent.{tfs-name}.{agent-name}:

/Users/{your-name}/Library/LaunchAgents/vsts.{tfs-name}.{agent-name}.testsvc.plist

Stopped
```

Uninstall

> You should stop before you uninstall.

Command:

```bash
./svc.sh uninstall
```

Automatic login and lock

Normally, the agent service runs only after the user logs in. If you want the agent service to automatically start when the machine restarts, you can configure the machine to automatically login and lock on startup. See Set your Mac to automatically login during startup - Apple Support.

> **Note**
>
> For more information, see the **Terminally Geeky: use automatic login more securely** blog. The .plist file mentioned in that blog may no longer be available at the source, but a copy can be found here: **Lifehacker - Make OS X load your desktop before you log in**.

Update environment variables

When you configure the service, it takes a snapshot of some useful environment variables for your current logon user such as PATH, LANG, JAVA_HOME, ANT_HOME, and MYSQL_PATH. If you need to update the variables (for example, after installing some new software):

```bash
./env.sh
./svc.sh stop
./svc.sh start
```

The snapshot of the environment variables is stored in `.env` file under agent root directory, you can also change that file directly to apply environment variable changes.

Run instructions before the service starts

You can also run your own instructions and commands to run when the service starts. For example, you could set up the environment or call scripts.

1. Edit `runsvc.sh`.
2. Replace the following line with your instructions:

```bash
# insert anything to setup env when running as a service
```

Service Files

When you install the service, some service files are put in place.

.plist service file

A .plist service file is created:

```
~/Library/LaunchAgents/vsts.agent.{tfs-name}.{agent-name}.plist
```

For example:

```
~/Library/LaunchAgents/vsts.agent.fabrikam.our-osx-agent.plist
```

`./svc.sh install` generates this file from this template: `./bin/vsts.agent.plist.template`

.service file

`./svc.sh start` finds the service by reading the `.service` file, which contains the path to the plist service file described above.

Alternative service mechanisms

We provide the `./svc.sh` script as a convenient way for you to run and manage your agent as a launchd LaunchAgent service. But you can use whatever kind of service mechanism you prefer.

You can use the template described above as to facilitate generating other kinds of service files. For example, you modify the template to generate a service that runs as a launch daemon if you don't need UI tests and don't want to configure automatic log on and lock. See Apple Developer Library: Creating Launch Daemons and Agents .

Replace an agent

To replace an agent, follow the *Download and configure the agent* steps again.

When you configure an agent using the same name as an agent that already exists, you're asked if you want to replace the existing agent. If you answer `Y`, then make sure you remove the agent (see below) that you're replacing. Otherwise, after a few minutes of conflicts, one of the agents will shut down.

Remove and reconfigure an agent

To remove the agent:

1. Stop and uninstall the service as explained in the previous section.
2. Remove the agent.

   ```bash
   ./config.sh remove
   ```

3. Enter your credentials.

After you've removed the agent, you can configure it again.

Unattended config

The agent can be set up from a script with no human intervention. You must pass `--unattended` and the answers to all questions.

To configure an agent, it must know the URL to your organization or collection and credentials of someone authorized to set up agents. All other responses are optional. Any command-line parameter can be specified using an environment variable instead:

put its name in upper case and prepend `VSTS_AGENT_INPUT_`. For example, `VSTS_AGENT_INPUT_PASSWORD` instead of specifying `--password`.

Required options

- `--unattended` - agent setup will not prompt for information, and all settings must be provided on the command line
- `--url <url>` - URL of the server. For example: https://dev.azure.com/myorganization or http://my-azure-devops-server:8080/tfs
- `--auth <type>` - authentication type. Valid values are:
  - `pat` (Personal access token) - PAT is the only scheme that works with Azure DevOps Services.
  - `negotiate` (Kerberos or NTLM)
  - `alt` (Basic authentication)
  - `integrated` (Windows default credentials)

Authentication options

- If you chose `--auth pat`:
  - `--token <token>` - specifies your personal access token
  - PAT is the only scheme that works with Azure DevOps Services.
- If you chose `--auth negotiate` or `--auth alt`:
  - `--userName <userName>` - specifies a Windows username in the format `domain\userName` or `userName@domain.com`
  - `--password <password>` - specifies a password

Pool and agent names

- `--pool <pool>` - pool name for the agent to join
- `--agent <agent>` - agent name
- `--replace` - replace the agent in a pool. If another agent is listening by the same name, it will start failing with a conflict

Agent setup

- `--work <workDirectory>` - work directory where job data is stored. Defaults to `_work` under the root of the agent directory. The work directory is owned by a given agent and should not be shared between multiple agents.

- `--acceptTeeEula` - accept the Team Explorer Everywhere End User License Agreement (macOS and Linux only)
- `--disableloguploads` - don't stream or send console log output to the server. Instead, you may retrieve them from the agent host's filesystem after the job completes.

Windows-only startup

- `--runAsService` - configure the agent to run as a Windows service (requires administrator permission)
- `--runAsAutoLogon` - configure auto-logon and run the agent on startup (requires administrator permission)
- `--windowsLogonAccount <account>` - used with `--runAsService` or `--runAsAutoLogon` to specify the Windows user name in the format `domain\userName` or `userName@domain.com`
- `--windowsLogonPassword <password>` - used with `--runAsService` or `--runAsAutoLogon` to specify Windows logon password (not required for Group Managed Service Accounts and Windows built in accounts such as 'NT AUTHORITY\NETWORK SERVICE')
- `--overwriteAutoLogon` - used with `--runAsAutoLogon` to overwrite the existing auto logon on the machine
- `--noRestart` - used with `--runAsAutoLogon` to stop the host from restarting after agent configuration completes

Troubleshooting configuring the agent with the `runAsAutoLogon` option

Configuring the agent with the `runAsAutoLogon` option runs the agent each time after restarting the machine. Perform next steps if the agent is not run after restarting the machine.

If the agent was already configured on the machine

Before reconfiguring the agent, it is necessary to remove the old agent configuration, so try to run this command from the agent folder:

```
.\config.cmd remove --auth 'PAT' --token '<token>'
```

Check if the agent was removed from your agent pool after executing the command:

```
<Azure DevOps organization> / <Project> / Settings / Agent pools / <Agent
Pool> / Agents
```

Remove the agent from your agent pool manually if it was not removed by running the command.

Then try to reconfigure the agent by running this command from the agent folder:

```
.\config.cmd --unattended --agent '<agent-name>' --pool '<agent-pool-name>'
--url '<azure-dev-ops-organization-url>' --auth 'PAT' --token '<token>' --
runAsAutoLogon --windowsLogonAccount '<domain\user-name>' --
windowsLogonPassword '<windows-password>'
```

Specify the agent name (any specific unique name) and check if this agent appeared in your agent pool after reconfiguring.

It will be much better to unpack an agent archive (which can be downloaded here) and run this command from the new unpacked agent folder.

Check if the Windows registry key is recorded and saved correctly

Run the `whoami /user` command to get the `<sid>`. Open `Registry Editor` and follow the path:

```
Computer\HKEY_USERS\<sid>\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

Check if there is the `VSTSAgent` key. Delete this key if it exists, then close `Registry Editor` and configure the agent by running the `.\config.cmd` command (without args) from the agent folder. Before answering the question `Enter Restart the machine at a later time?`, open `Registry Editor` again and check if the `VSTSAgent` key has appeared. Press `Enter` to answer the question, and check if the `VSTSAgent` key remains in its place after restarting the machine.

Check if Windows registry keys work fine on your machine

Create a `autorun.cmd` file that contains the following line: `echo "Hello from AutoRun!"`. Open `Registry Editor` and create in the path above a new key-value pair with the key `AutoRun` and the value

```
C:\windows\system32\cmd.exe /D /S /C start "AutoRun"
"D:\path\to\autorun.cmd"
```

Restart your machine. You have an issue with Windows registry keys if you do not see a console window with the `Hello from AutoRun!` message.

Deployment group only

- `--deploymentGroup` - configure the agent as a deployment group agent
- `--deploymentGroupName <name>` - used with `--deploymentGroup` to specify the deployment group for the agent to join
- `--projectName <name>` - used with `--deploymentGroup` to set the project name
- `--addDeploymentGroupTags` - used with `--deploymentGroup` to indicate that deployment group tags should be added
- `--deploymentGroupTags <tags>` - used with `--addDeploymentGroupTags` to specify the comma separated list of tags for the deployment group agent - for example "web, db"

Environments only

- `--addvirtualmachineresourcetags` - used to indicate that environment resource tags should be added
- `--virtualmachineresourcetags <tags>` - used with `--addvirtualmachineresourcetags` to specify the comma separated list of tags for the environment resource agent - for example "web, db"

`./config.sh --help` always lists the latest required and optional responses.

Diagnostics

If you're having trouble with your self-hosted agent, you can try running diagnostics. After configuring the agent:

```
Bash
```

```
./run.sh --diagnostics
```

This will run through a diagnostic suite that may help you troubleshoot the problem. The diagnostics feature is available starting with agent version 2.165.0.

Network diagnostics for self-hosted agents

Set the value of `Agent.Diagnostic` to `true` to collect additional logs that can be used for troubleshooting network issues for self-hosted agents. For more information, see Network diagnostics for self-hosted agents.

Help on other options

To learn about other options:

```bash
./config.sh --help
```

The help provides information on authentication alternatives and unattended configuration.

Capabilities

Your agent's capabilities are cataloged and advertised in the pool so that only the builds and releases it can handle are assigned to it. See Build and release agent capabilities.

In many cases, after you deploy an agent, you'll need to install software or utilities. Generally you should install on your agents whatever software and tools you use on your development machine.

For example, if your build includes the npm task, then the build won't run unless there's a build agent in the pool that has npm installed.

> **Important**
>
> Capabilities include all environment variables and the values that are set when the agent runs. If any of these values change while the agent is running, the agent must be restarted to pick up the new values. After you install new software on an agent,

you must restart the agent for the new capability to show up in the pool, so that the build can run.

If you want to exclude environment variables as capabilities, you can designate them by setting an environment variable `VSO_AGENT_IGNORE` with a comma-delimited list of variables to ignore.

FAQ

How do I make sure I have the latest agent version?

1. Navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Click the pool that contains the agent.

3. Make sure the agent is enabled.

4. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

   b. Select **Agents** and choose the desired agent.

c. Choose the **Capabilities** tab.

> ⓘ **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

5. Look for the `Agent.Version` capability. You can check this value against the latest published agent version. See Azure Pipelines Agent and check the page for the

highest version number listed.

6. Each agent automatically updates itself when it runs a task that requires a newer version of the agent. If you want to manually update some agents, right-click the pool, and select **Update all agents**.

How do I make sure I have the latest agent version?

1. Navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Click the pool that contains the agent.

3. Make sure the agent is enabled.

4. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

   b. Select **Agents** and choose the desired agent.

   c. Choose the **Capabilities** tab.

> **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

5. Look for the `Agent.Version` capability. You can check this value against the latest published agent version. See Azure Pipelines Agent and check the page for the highest version number listed.

6. Each agent automatically updates itself when it runs a task that requires a newer version of the agent. If you want to manually update some agents, right-click the pool, and select **Update all agents**.

Where can I learn more about how the launchd service works?

Apple Developer Library: Creating Launch Daemons and Agents

I'm running a firewall and my code is in Azure Repos. What URLs does the agent need to communicate with?

If you're running an agent in a secure network behind a firewall, make sure the agent can initiate communication with the following URLs and IP addresses.

Expand table

Domain URL	Description
`https://{organization_name}.pkgs.visualstudio.com`	Azure DevOps Packaging API for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsrm.visualstudio.com`	Release Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://*.blob.core.windows.net`	Azure Artifacts
`https://*.dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://*.vsassets.io`	Azure Artifacts via CDN
`https://*.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `dev.azure.com` domain
`https://*.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `dev.azure.com` domain

Domain URL	Description
`https://*.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the `dev.azure.com` domain
`https://app.vssps.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://login.microsoftonline.com`	Microsoft Entra sign-in
`https://management.core.windows.net`	Azure Management APIs
`https://vstsagentpackage.azureedge.net`	Agent package

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> For more information about allowed addresses, see **Allowed address lists and network connections**.

How do I run the agent with self-signed certificate?

Run the agent with self-signed certificate

How do I run the agent behind a web proxy?

Run the agent behind a web proxy

How do I restart the agent

If you are running the agent interactively, see the restart instructions in Run interactively. If you are running the agent as a service, follow the steps to Stop and then Start the agent.

How do I configure the agent to bypass a web proxy and connect to Azure Pipelines?

If you want the agent to bypass your proxy and connect to Azure Pipelines directly, then you should configure your web proxy to enable the agent to access the following URLs.

For organizations using the `*.visualstudio.com` domain:

```
https://login.microsoftonline.com
https://app.vssps.visualstudio.com
https://{organization_name}.visualstudio.com
https://{organization_name}.vsrm.visualstudio.com
https://{organization_name}.vstmr.visualstudio.com
https://{organization_name}.pkgs.visualstudio.com
https://{organization_name}.vssps.visualstudio.com
```

For organizations using the `dev.azure.com` domain:

```
https://dev.azure.com
https://*.dev.azure.com
https://login.microsoftonline.com
https://management.core.windows.net
https://vstsagentpackage.azureedge.net
https://vssps.dev.azure.com
```

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> This procedure enables the agent to bypass a web proxy. Your build pipeline and scripts must still handle bypassing your web proxy for each task and tool you run in your build.
>
> For example, if you are using a NuGet task, you must configure your web proxy to support bypassing the URL for the server that hosts the NuGet feed you're using.

TFVC prerequisites

If you'll be using TFVC, you'll also need the Oracle Java JDK 1.6 or higher. (The Oracle JRE and OpenJDK aren't sufficient for this purpose.)

TEE plugin is used for TFVC functionality. It has an EULA, which you'll need to accept during configuration if you plan to work with TFVC.

Since the TEE plugin is no longer maintained and contains some out-of-date Java dependencies, starting from Agent 2.198.0 it's no longer included in the agent distribution. However, the TEE plugin will be downloaded during checkout task execution if you're checking out a TFVC repo. The TEE plugin will be removed after the job execution.

> **Note**
>
> Note: You may notice your checkout task taking a long time to start working because of this download mechanism.

If the agent is running behind a proxy or a firewall, you'll need to ensure access to the following site: `https://vstsagenttools.blob.core.windows.net/`. The TEE plugin will be downloaded from this address.

If you're using a self-hosted agent and facing issues with TEE downloading, you may install TEE manually:

1. Set `DISABLE_TEE_PLUGIN_REMOVAL` environment or pipeline variable to `true`. This variable prevents the agent from removing the TEE plugin after TFVC repository checkout.
2. Download TEE-CLC version 14.135.0 manually from Team Explorer Everywhere GitHub releases.
3. Extract the contents of `TEE-CLC-14.135.0` folder to `<agent_directory>/externals/tee`.

Self-hosted Windows agents

Article • 10/10/2023

Azure DevOps Services

To build and deploy Windows, Azure, and other Visual Studio solutions you'll need at least one Windows agent. Windows agents can also build Java and Android apps.

> **Important**
>
> This article provides guidance for using the **3.x agent software** with Azure DevOps Services. If you're using Azure DevOps Server or TFS, see **Self-hosted Windows agents (Agent version 2.x)**.

> **Note**
>
> This article describes how to configure a **self-hosted agent**. If you're using Azure DevOps Services and a **Microsoft-hosted agent** meets your needs, you can skip setting up a self-hosted Windows agent.

Learn about agents

If you already know what an agent is and how it works, feel free to jump right in to the following sections. But if you'd like some more background about what they do and how they work, see Azure Pipelines agents.

Check prerequisites

Make sure your machine has these prerequisites:

- Operating system version
  - Client OS
    - Windows 7 SP1 ESU
    - Windows 8.1
    - Windows 10
    - Windows 11
  - Server OS
    - Windows Server 2012 or higher

- The agent software installs its own version of .NET so there's no .NET prerequisite.
- PowerShell 3.0 or higher
- **Subversion** - If you're building from a Subversion repo, you must install the Subversion client on the machine.
- Recommended - Visual Studio build tools (2015 or higher)

You should run agent setup manually the first time. After you get a feel for how agents work, or if you want to automate setting up many agents, consider using unattended config.

Hardware specs

The hardware specs for your agents will vary with your needs, team size, etc. It's not possible to make a general recommendation that will apply to everyone. As a point of reference, the Azure DevOps team builds the hosted agents code using pipelines that utilize hosted agents. On the other hand, the bulk of the Azure DevOps code is built by 24-core server class machines running four self-hosted agents apiece.

Prepare permissions

Information security for self-hosted agents

The user configuring the agent needs pool admin permissions, but the user running the agent does not.

The folders controlled by the agent should be restricted to as few users as possible and they contain secrets that could be decrypted or exfiltrated.

The Azure Pipelines agent is a software product designed to execute code it downloads from external sources. It inherently could be a target for Remote Code Execution (RCE) attacks.

Therefore, it is important to consider the threat model surrounding each individual usage of Pipelines Agents to perform work, and decide what are the minimum permissions could be granted to the user running the agent, to the machine where the agent runs, to the users who have write access to the Pipeline definition, the git repos where the yaml is stored, or the group of users who control access to the pool for new pipelines.

It is a best practice to have the identity running the agent be different from the identity with permissions to connect the agent to the pool. The user generating the credentials (and other agent-related files) is different than the user that needs to read them. Therefore, it is safer to carefully consider access granted to the agent machine itself, and the agent folders which contain sensitive files, such as logs and artifacts.

It makes sense to grant access to the agent folder only for DevOps administrators and the user identity running the agent process. Administrators may need to investigate the file system to understand build failures or get log files to be able to report Azure DevOps failures.

Decide which user you'll use

As a one-time step, you must register the agent. Someone with permission to administer the agent queue must complete these steps. The agent will not use this person's credentials in everyday operation, but they're required to complete registration. Learn more about how agents communicate.

Confirm the user has permission

Make sure the user account that you're going to use has permission to register the agent.

Is the user an Azure DevOps organization owner or TFS or Azure DevOps Server administrator? **Stop here**, you have permission.

Otherwise:

1. Open a browser and navigate to the **Agent pools** tab for your Azure Pipelines organization or Azure DevOps Server or TFS server:

    a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

    b. Choose **Azure DevOps**, **Organization settings**.

c. Choose **Agent pools**.

2. Select the pool on the right side of the page and then click **Security**.

3. If the user account you're going to use is not shown, then get an administrator to add it. The administrator can be an agent pool administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   If it's a deployment group agent, the administrator can be a deployment group administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   You can add a user to the deployment group administrator role in the **Security** tab on the **Deployment Groups** page in **Azure Pipelines**.

> ⓘ **Note**

If you see a message like this: **Sorry, we couldn't add the identity. Please try a different identity.**, you probably followed the above steps for an organization owner or TFS or Azure DevOps Server administrator. You don't need to do anything; you already have permission to administer the agent pool.

Download and configure the agent

Azure Pipelines

1. Log on to the machine using the account for which you've prepared permissions as explained above.

2. In your web browser, sign in to Azure Pipelines, and navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

3. Select the **Default** pool, select the **Agents** tab, and choose **New agent**.

4. On the **Get the agent** dialog box, choose **Windows**.

5. On the left pane, select the processor architecture of the installed Windows OS version on your machine. The x64 agent version is intended for 64-bit Windows, whereas the x86 version is intended for 32-bit Windows. If you aren't sure which version of Windows is installed, follow these instructions to find out.

6. On the right pane, click the **Download** button.

7. Follow the instructions on the page to download the agent.

8. Unpack the agent into the directory of your choice. Make sure that the path to the directory contains no spaces because tools and scripts don't always properly escape spaces. A recommended folder is `C:\agents`. Extracting in the download folder or other user folders may cause permission issues. Then run `config.cmd`. This will ask you a series of questions to configure the agent.

> **Important**
>
> We strongly recommend you configure the agent from an elevated PowerShell window. If you want to configure as a service, this is **required**.
>
> You must not use **Windows PowerShell ISE** to configure the agent.

> **Important**

For security reasons we strongly recommend making sure the agents folder (`C:\agents`) is only editable by admins.

> **Note**
>
> Please avoid using mintty based shells, such as git-bash, for agent configuration. Mintty is not fully compatible with native Input/Output Windows API (**here** is some info about it) and we couldn't guarantee correct work of setup script in this case.

Server URL

When setup asks for your server URL, for Azure DevOps Services, answer `https://dev.azure.com/{your-organization}`.

Agent setup authentication type

When you register an agent, choose from the following authentication types, and agent setup prompts you for the specific additional information required for each authentication type. For more information, see Self-hosted agent authentication options.

- Personal access token
- Device code flow
- Service principal

The authentication method used for registering the agent is used only during agent registration. To learn more about how agents communicate with Azure Pipelines after registration, see Communication with Azure Pipelines or TFS.

Choose interactive or service mode

For guidance on whether to run the agent in interactive mode or as a service, see Agents: Interactive vs. service.

If you choose to run as a service (which we recommend), the username you run as should be 20 characters or fewer.

Run the agent

Run interactively

If you configured the agent to run interactively, to run it:

```ps
.\run.cmd
```

To restart the agent, press Ctrl+C to stop the agent and then run `run.cmd` to restart it.

Run once

For agents configured to run interactively, you can choose to have the agent accept only one job. To run in this configuration:

```ps
.\run.cmd --once
```

Agents in this mode will accept only one job and then spin down gracefully (useful for running in Docker on a service like Azure Container Instances).

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 restart the agent, right-click the entry and choose **Restart**.

> **Note**
>
> If you need to change the agent's logon account, don't do it from the Services snap-in. Instead, see the information below to reconfigure the agent.

To use your agent, run a job using the agent's pool. If you didn't choose a different pool, your agent will be in the **Default** pool.

Replace an agent

To replace an agent, follow the *Download and configure the agent* steps again.

When you configure an agent using the same name as an agent that already exists, you're asked if you want to replace the existing agent. If you answer `Y`, then make sure you remove the agent (see below) that you're replacing. Otherwise, after a few minutes of conflicts, one of the agents will shut down.

Remove and reconfigure an agent

To remove the agent:

```ps
.\config remove
```

After you've removed the agent, you can configure it again.

Unattended config

The agent can be set up from a script with no human intervention. You must pass `--unattended` and the answers to all questions.

To configure an agent, it must know the URL to your organization or collection and credentials of someone authorized to set up agents. All other responses are optional. Any command-line parameter can be specified using an environment variable instead: put its name in upper case and prepend `VSTS_AGENT_INPUT_`. For example, `VSTS_AGENT_INPUT_PASSWORD` instead of specifying `--password`.

Required options

- `--unattended` - agent setup will not prompt for information, and all settings must be provided on the command line
- `--url <url>` - URL of the server. For example: https://dev.azure.com/myorganization or http://my-azure-devops-server:8080/tfs
- `--auth <type>` - authentication type. Valid values are:
  - `pat` (Personal access token) - PAT is the only scheme that works with Azure DevOps Services.

- `negotiate` (Kerberos or NTLM)
- `alt` (Basic authentication)
- `integrated` (Windows default credentials)

Authentication options

- If you chose `--auth pat`:
  - `--token <token>` - specifies your personal access token
  - PAT is the only scheme that works with Azure DevOps Services.
- If you chose `--auth negotiate` or `--auth alt`:
  - `--userName <userName>` - specifies a Windows username in the format `domain\userName` or `userName@domain.com`
  - `--password <password>` - specifies a password

Pool and agent names

- `--pool <pool>` - pool name for the agent to join
- `--agent <agent>` - agent name
- `--replace` - replace the agent in a pool. If another agent is listening by the same name, it will start failing with a conflict

Agent setup

- `--work <workDirectory>` - work directory where job data is stored. Defaults to `_work` under the root of the agent directory. The work directory is owned by a given agent and should not be shared between multiple agents.
- `--acceptTeeEula` - accept the Team Explorer Everywhere End User License Agreement (macOS and Linux only)
- `--disableloguploads` - don't stream or send console log output to the server. Instead, you may retrieve them from the agent host's filesystem after the job completes.

Windows-only startup

- `--runAsService` - configure the agent to run as a Windows service (requires administrator permission)
- `--runAsAutoLogon` - configure auto-logon and run the agent on startup (requires administrator permission)

- `--windowsLogonAccount <account>` - used with `--runAsService` or `--runAsAutoLogon` to specify the Windows user name in the format `domain\userName` or `userName@domain.com`
- `--windowsLogonPassword <password>` - used with `--runAsService` or `--runAsAutoLogon` to specify Windows logon password (not required for Group Managed Service Accounts and Windows built in accounts such as 'NT AUTHORITY\NETWORK SERVICE')
- `--overwriteAutoLogon` - used with `--runAsAutoLogon` to overwrite the existing auto logon on the machine
- `--noRestart` - used with `--runAsAutoLogon` to stop the host from restarting after agent configuration completes

Troubleshooting configuring the agent with the `runAsAutoLogon` option

Configuring the agent with the `runAsAutoLogon` option runs the agent each time after restarting the machine. Perform next steps if the agent is not run after restarting the machine.

If the agent was already configured on the machine

Before reconfiguring the agent, it is necessary to remove the old agent configuration, so try to run this command from the agent folder:

```
.\config.cmd remove --auth 'PAT' --token '<token>'
```

Check if the agent was removed from your agent pool after executing the command:

```
<Azure DevOps organization> / <Project> / Settings / Agent pools / <Agent
Pool> / Agents
```

Remove the agent from your agent pool manually if it was not removed by running the command.

Then try to reconfigure the agent by running this command from the agent folder:

```
.\config.cmd --unattended --agent '<agent-name>' --pool '<agent-pool-name>'
--url '<azure-dev-ops-organization-url>' --auth 'PAT' --token '<token>' --
runAsAutoLogon --windowsLogonAccount '<domain\user-name>' --
windowsLogonPassword '<windows-password>'
```

Specify the agent name (any specific unique name) and check if this agent appeared in your agent pool after reconfiguring.

It will be much better to unpack an agent archive (which can be downloaded here) and run this command from the new unpacked agent folder.

Check if the Windows registry key is recorded and saved correctly

Run the `whoami /user` command to get the `<sid>`. Open `Registry Editor` and follow the path:

```
Computer\HKEY_USERS\<sid>\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

Check if there is the `VSTSAgent` key. Delete this key if it exists, then close `Registry Editor` and configure the agent by running the `.\config.cmd` command (without args) from the agent folder. Before answering the question `Enter Restart the machine at a later time?`, open `Registry Editor` again and check if the `VSTSAgent` key has appeared. Press `Enter` to answer the question, and check if the `VSTSAgent` key remains in its place after restarting the machine.

Check if Windows registry keys work fine on your machine

Create a `autorun.cmd` file that contains the following line: `echo "Hello from AutoRun!"`. Open `Registry Editor` and create in the path above a new key-value pair with the key `AutoRun` and the value

```
C:\windows\system32\cmd.exe /D /S /C start "AutoRun"
"D:\path\to\autorun.cmd"
```

Restart your machine. You have an issue with Windows registry keys if you do not see a console window with the `Hello from AutoRun!` message.

Deployment group only

- `--deploymentGroup` - configure the agent as a deployment group agent
- `--deploymentGroupName <name>` - used with `--deploymentGroup` to specify the deployment group for the agent to join
- `--projectName <name>` - used with `--deploymentGroup` to set the project name
- `--addDeploymentGroupTags` - used with `--deploymentGroup` to indicate that deployment group tags should be added
- `--deploymentGroupTags <tags>` - used with `--addDeploymentGroupTags` to specify the comma separated list of tags for the deployment group agent - for example "web, db"

Environments only

- `--addvirtualmachineresourcetags` - used to indicate that environment resource tags should be added
- `--virtualmachineresourcetags <tags>` - used with `--addvirtualmachineresourcetags` to specify the comma separated list of tags for the environment resource agent - for example "web, db"

`.\config --help` always lists the latest required and optional responses.

Diagnostics

If you're having trouble with your self-hosted agent, you can try running diagnostics. After configuring the agent:

```ps
.\run --diagnostics
```

This will run through a diagnostic suite that may help you troubleshoot the problem. The diagnostics feature is available starting with agent version 2.165.0.

Network diagnostics for self-hosted agents

Set the value of `Agent.Diagnostic` to `true` to collect additional logs that can be used for troubleshooting network issues for self-hosted agents. For more information, see Network diagnostics for self-hosted agents

Help on other options

To learn about other options:

```ps
.\config --help
```

The help provides information on authentication alternatives and unattended configuration.

Capabilities

Your agent's capabilities are cataloged and advertised in the pool so that only the builds and releases it can handle are assigned to it. See Build and release agent capabilities.

In many cases, after you deploy an agent, you'll need to install software or utilities. Generally you should install on your agents whatever software and tools you use on your development machine.

For example, if your build includes the npm task, then the build won't run unless there's a build agent in the pool that has npm installed.

> **Important**
>
> Capabilities include all environment variables and the values that are set when the agent runs. If any of these values change while the agent is running, the agent must be restarted to pick up the new values. After you install new software on an agent, you must restart the agent for the new capability to show up in the pool, so that the build can run.
>
> If you want to exclude environment variables as capabilities, you can designate them by setting an environment variable `VSO_AGENT_IGNORE` with a comma-delimited list of variables to ignore.

FAQ

What version of Git does my agent run?

By default, the Windows agent uses the version of Git that is bundled with the agent software. Microsoft recommends using the version of Git that is bundled with the agent, but you have several options to override this default behavior and use the version of Git that the agent machine has installed in the path.

- Set a pipeline variable named `System.PreferGitFromPath` to `true` in your pipelines.
- On self-hosted agents, you can create a file named **.env** in the agent root directory and add a `System.PreferGitFromPath=true` line to the file. For more information, see How do I set different environment variables for each individual agent?

To see the version of Git used by a pipeline, you can look at the logs for a `checkout` step in your pipeline, as shown in the following example.

```
Syncing repository: PathFilter (Git)
Prepending Path environment variable with directory containing 'git.exe'.
git version
git version 2.26.2.windows.1
```

How do I make sure I have the latest agent version?

1. Navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Click the pool that contains the agent.

3. Make sure the agent is enabled.

4. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

   b. Select **Agents** and choose the desired agent.

c. Choose the **Capabilities** tab.

> ⓘ **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

5. Look for the `Agent.Version` capability. You can check this value against the latest published agent version. See Azure Pipelines Agent and check the page for the

highest version number listed.

6. Each agent automatically updates itself when it runs a task that requires a newer version of the agent. If you want to manually update some agents, right-click the pool, and select **Update all agents**.

I'm running a firewall and my code is in Azure Repos. What URLs does the agent need to communicate with?

If you're running an agent in a secure network behind a firewall, make sure the agent can initiate communication with the following URLs and IP addresses.

Expand table

Domain URL	Description
`https://{organization_name}.pkgs.visualstudio.com`	Azure DevOps Packaging API for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsrm.visualstudio.com`	Release Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://*.blob.core.windows.net`	Azure Artifacts

Domain URL	Description
`https://*.dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://*.vsassets.io`	Azure Artifacts via CDN
`https://*.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `dev.azure.com` domain
`https://*.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `dev.azure.com` domain
`https://*.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the `dev.azure.com` domain
`https://app.vssps.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://login.microsoftonline.com`	Microsoft Entra sign-in
`https://management.core.windows.net`	Azure Management APIs
`https://vstsagentpackage.azureedge.net`	Agent package

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> For more information about allowed addresses, see **Allowed address lists and network connections**.

How do I run the agent with self-signed certificate?

> **Note**
>
> Running the agent with a self-signed certificate only applies to Azure DevOps Server.

Run the agent with self-signed certificate

How do I run the agent behind a web proxy?

Run the agent behind a web proxy

How do I restart the agent

If you're running the agent interactively, see the restart instructions in Run interactively. If you're running the agent as a service, restart the agent by following the steps in Run as a service.

How do I set different environment variables for each individual agent?

Create a `.env` file under agent's root directory and put the environment variables you want to set into the file in the following format, and then restart the agent.

```
MyEnv0=MyEnvValue0
MyEnv1=MyEnvValue1
MyEnv2=MyEnvValue2
```

```
MyEnv3=MyEnvValue3
MyEnv4=MyEnvValue4
```

How do I configure the agent to bypass a web proxy and connect to Azure Pipelines?

If you want the agent to bypass your proxy and connect to Azure Pipelines directly, then you should configure your web proxy to enable the agent to access the following URLs.

For organizations using the `*.visualstudio.com` domain:

```
https://login.microsoftonline.com
https://app.vssps.visualstudio.com
https://{organization_name}.visualstudio.com
https://{organization_name}.vsrm.visualstudio.com
https://{organization_name}.vstmr.visualstudio.com
https://{organization_name}.pkgs.visualstudio.com
https://{organization_name}.vssps.visualstudio.com
```

For organizations using the `dev.azure.com` domain:

```
https://dev.azure.com
https://*.dev.azure.com
https://login.microsoftonline.com
https://management.core.windows.net
https://vstsagentpackage.azureedge.net
https://vssps.dev.azure.com
```

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> This procedure enables the agent to bypass a web proxy. Your build pipeline and scripts must still handle bypassing your web proxy for each task and tool you run in your build.
>
> For example, if you are using a NuGet task, you must configure your web proxy to support bypassing the URL for the server that hosts the NuGet feed you're using.

I'm using TFS and the URLs in the sections above don't work for me. Where can I get help?

Web site settings and security

What is enable SERVICE_SID_TYPE_UNRESTRICTED for agent service?

When configuring the agent software on Windows Server, you can specify the service security identifier from the following prompt.

```
Enter enable SERVICE_SID_TYPE_UNRESTRICTED for agent service (Y/N) (press
enter for N)
```

Previous versions of the agent software set the service security identifier type to `SERVICE_SID_TYPE_NONE`, which is the default value for the current agent versions. To configure the security service identifier type to `SERVICE_SID_TYPE_UNRESTRICTED`, press `Y`.

For more information, see SERVICE_SID_INFO structure and Security identifiers.

Self-hosted agent authentication options

Article • 10/16/2023

Azure Pipelines provides a choice of several authentication options you can use when you are registering an agent. These methods of authentication are used only during agent registration. See Agents communication for details of how agents communicate after registration.

Agent registration method	Azure DevOps Services	Azure DevOps Server & TFS
Personal access token (PAT)	Supported	Supported when server is configured with HTTPS
Service Principal (SP)	Supported	Currently not supported
Device code flow (Microsoft Entra ID)	Supported	Currently not supported
Integrated	Not supported	Windows agents only
Negotiate	Not supported	Windows agents only
Alternate (ALT)	Not supported	Supported when server is configured with HTTPS

Personal access token (PAT)

Specify **PAT** for authentication type during agent configuration to use a personal access token to authenticate during agent registration, then specify a personal access token (PAT) with **Agent Pools (read, manage)** scope (or **Deployment group (read, manage)** scope for a deployment group agent) can be used for agent registration.

For more information, see Register an agent using a personal access token (PAT)

Service Principal

Specify **SP** for authentication type during agent configuration to use a service principal to authenticate during agent registration.

For more information, see Register an agent using a Service Principal.

Device code flow

Specify `AAD` for authentication type during agent configuration to use device code flow to authenticate during agent registration.

For more information, see Register an agent using device code flow.

Integrated

Integrated windows authentication for agent registration is only available for Windows agent registration on Azure DevOps Server and TFS.

Negotiate

The negotiate authentication method for agent registration is only available for Windows agent registration on Azure DevOps Server and TFS.

Alternate (ALT)

The alternate (basic) authentication method for agent registration is only available on Azure DevOps Server and TFS.

Register an agent using a personal access token (PAT)

Article • 10/10/2023

Specify **PAT** for authentication type during agent configuration to use a personal access token to authenticate during agent registration, then specify a personal access token (PAT) with **Agent Pools (read, manage)** scope (or **Deployment group (read, manage)** scope for a deployment group agent) can be used for agent registration.

A single PAT can be used for registering multiple agents, the PAT is used only at the time of registering the agent, and not for subsequent communication.

To use a PAT with Azure DevOps Server, your server must be configured with HTTPS. See Web site settings and security.

Create a personal access token for agent registration

1. Sign in with the user account you plan to use in your Azure DevOps organization (`https://dev.azure.com/{your_organization}`).

2. From your home page, open your user settings, and then select **Personal access tokens**.

3. Create a personal access token.

4. For the scope select **Agent Pools (read, manage)** and make sure all the other boxes are cleared. If it's a deployment group agent, for the scope select **Deployment group (read, manage)** and make sure all the other boxes are cleared.

   Select **Show all scopes** at the bottom of the **Create a new personal access token window** window to see the complete list of scopes.

5. Copy the token. You'll use this token when you configure the agent.

> **Note**
>
> When using PAT as the authentication method, the PAT token is used only for the initial configuration of the agent. Learn more at **Communication with Azure Pipelines or TFS**.

Next steps

- Self-hosted Windows agents
- Self-hosted Linux agents
- Self-hosted macOS agents

Register an agent using a service principal

Article • 10/10/2023

You can register an agent using a Service Principal starting with agent version 3.227.1 by specifying **SP** as the agent authentication option.

Grant the Service Principal access to the agent pool

Before registering an agent using a Service Principal you must have created a Service Principal and granted it permission to access the agent pool.

1. Open a browser and navigate to the **Agent pools** tab for your Azure Pipelines organization.

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Select the desired agent pool on the right side of the page and then choose **Security**. Choose Add, and add the Service Principal with the **Administrator** role.

3. If the Service Principal you're going to use is not shown, then get an administrator to add it, granting the Service Principal the administrator role for the agent pool. The administrator can be an agent pool administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   If it's a deployment group agent, the administrator can be a deployment group administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   You can add a Service Principal to the deployment group administrator role in the **Security** tab on the **Deployment Groups** page in **Azure Pipelines**.

> **Note**
>
> If you see a message like this: **Sorry, we couldn't add the identity. Please try a different identity.** or **Cannot modify the role for self identity. Please try with a different identity.**, you probably followed the above steps for an organization owner or TFS or Azure DevOps Server administrator. You don't need to do anything; you already have permission to administer the agent pool.
>
> If you are adding the Service Principal to the agent pool security group using **Project Settings**, **Agent pools**, you must first add the Service Principal as an organization user with **Basic Access level** (recommended) or higher.

Register the agent using a Service Principal

1. Specify **SP** when prompted for authentication type during agent configuration to use a Service Principal to authenticate during agent registration.

2. When prompted, supply the **Client(App) ID** and the **Tenant ID**.

3. Specify the client secret. The client secret is used only during agent registration.

4. Specify the name of the agent pool for which you granted administrator permission for the Service Principal, and continue the agent registration steps.

Register an agent using device code flow

Article • 10/18/2023

You can register an agent using device code flow starting with agent version 3.227.1 by specifying `AAD` when prompted for the agent authentication type.

Grant the user access to the agent pool

Before registering an agent using device code flow, you must grant the desired user permission to access the agent pool.

1. Open a browser and navigate to the **Agent pools** tab for your Azure Pipelines organization.

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Select the desired agent pool on the right side of the page and then choose **Security**. Choose Add, and add the user with the **Administrator** role.

3. If the user account you're going to use is not shown, then get an administrator to add it, granting the account the administrator role for the agent pool. The administrator can be an agent pool administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   If it's a deployment group agent, the administrator can be a deployment group administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   You can add the desired account to the deployment group administrator role in the **Security** tab on the **Deployment Groups** page in **Azure Pipelines**.

> **Note**
>
> If you see a message like this: **Sorry, we couldn't add the identity. Please try a different identity.** or **Cannot modify the role for self identity. Please try with a different identity.**, you probably followed the above steps for an organization owner or TFS or Azure DevOps Server administrator. You don't need to do anything; you already have permission to administer the agent pool.
>
> If you are adding the user account to the agent pool security group using **Project Settings**, **Agent pools**, you must first add the user account as an organization user with **Basic Access level** (recommended) or higher.

Register the agent using device code flow

1. Specify `AAD` when prompted for authentication type during agent configuration to use a Service Principal to authenticate during agent registration. You'll receive a prompt similar to the following message: **Please finish device code flow in browser (https://microsoft.com/devicelogin), user code: A12WDTGEFD**
2. Go to the specified link, enter the user code, and complete the sign-in process there.
3. Specify the name of the agent pool for which you granted administrator permission for the Service Principal, and continue the agent registration steps.

Self-hosted Linux agents (2.x)

Article • 05/10/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> **Important**
>
> This article provides guidance for using the 2.x version agent software with Azure DevOps Server and TFS. If you're using Azure DevOps Services, see **Self-hosted Linux agents**.

To run your jobs, you'll need at least one agent. A Linux agent can build and deploy different kinds of apps, including Java and Android apps. We support Ubuntu, Red Hat, and CentOS.

> Before you begin:
>
> - If your pipelines are in Azure Pipelines and a Microsoft-hosted agent meets your needs, you can skip setting up a private Linux agent.
> - Otherwise, you've come to the right place to set up an agent on Linux. Continue to the next section.

Learn about agents

If you already know what an agent is and how it works, feel free to jump right in to the following sections. But if you'd like some more background about what they do and how they work, see Azure Pipelines agents.

Check prerequisites

The agent is based on .NET Core 3.1. You can run this agent on several Linux distributions. We support the following subset of .NET Core supported distributions:

- x64
  - CentOS 7, 6 (see note 1)
  - Debian 9
  - Fedora 30, 29
  - Linux Mint 18, 17
  - openSUSE 42.3 or later

    - Oracle Linux 8, 7
    - Red Hat Enterprise Linux 8, 7, 6 (see note 1)
    - SUSE Enterprise Linux 12 SP2 or later
    - Ubuntu 20.04, 18.04, 16.04
    - Azure Linux 1.0 (see note 3)
- ARM32 (see note 2)
    - Debian 9
    - Ubuntu 18.04
- ARM64
    - Debian 9
    - Ubuntu 21.04, 20.04, 18.04

> **Note**
>
> Note 1: RHEL 6 and CentOS 6 require installing the specialized `rhel.6-x64` version of the agent.

> **Important**
>
> As of February 2023, no more agent releases support RHEL 6. For more information, see **Customers using Red Hat Enterprise Linux (RHEL) 6 should upgrade the OS on Self-hosted agents**.

> **Note**
>
> Note 2: ARM instruction set **ARMv7** or above is required. Run `uname -a` to see your Linux distro's instruction set.

> **Note**
>
> Azure Linux OS distribution currently has partial support from the Azure DevOps Agent. We are providing a mechanism for detection of this OS distribution in `installdependencies.sh` script, but due to lack of support from the **.Net Core** side, we couldn't guarantee full operability of all agent functions when running on this OS distribution.

Regardless of your platform, you will need to install Git 2.9.0 or higher. We strongly recommend installing the latest version of Git.

> **Note**
>
> The agent installer knows how to check for other dependencies. You can install those dependencies on supported Linux platforms by running `./bin/installdependencies.sh` in the agent directory.
>
> Be aware that some of these dependencies required by .NET Core are fetched from third party sites, like `packages.efficios.com`. Review the `installdependencies.sh` script and ensure any referenced third party sites are accessible from your Linux machine before running the script.
>
> Please also make sure that all required repositories are connected to the relevant package manager used in `installdependencies.sh` (like `apt` or `zypper`).
>
> For issues with dependencies installation (like 'dependency was not found in repository' or 'problem retrieving the repository index file') - you can reach out to distribution owner for further support.

Subversion

If you're building from a Subversion repo, you must install the Subversion client on the machine.

You should run agent setup manually the first time. After you get a feel for how agents work, or if you want to automate setting up many agents, consider using unattended config.

TFVC

If you'll be using TFVC, you'll also need the Oracle Java JDK 1.6 or higher. (The Oracle JRE and OpenJDK aren't sufficient for this purpose.)

TEE plugin is used for TFVC functionality. It has an EULA, which you'll need to accept during configuration if you plan to work with TFVC.

Since the TEE plugin is no longer maintained and contains some out-of-date Java dependencies, starting from Agent 2.198.0 it's no longer included in the agent distribution. However, the TEE plugin will be downloaded during checkout task execution if you're checking out a TFVC repo. The TEE plugin will be removed after the job execution.

> **Note**
>
> Note: You may notice your checkout task taking a long time to start working because of this download mechanism.

If the agent is running behind a proxy or a firewall, you'll need to ensure access to the following site: `https://vstsagenttools.blob.core.windows.net/`. The TEE plugin will be downloaded from this address.

If you're using a self-hosted agent and facing issues with TEE downloading, you may install TEE manually:

1. Set `DISABLE_TEE_PLUGIN_REMOVAL` environment or pipeline variable to `true`. This variable prevents the agent from removing the TEE plugin after TFVC repository checkout.
2. Download TEE-CLC version 14.135.0 manually from Team Explorer Everywhere GitHub releases.
3. Extract the contents of `TEE-CLC-14.135.0` folder to `<agent_directory>/externals/tee`.

Prepare permissions

Information security for self-hosted agents

The user configuring the agent needs pool admin permissions, but the user running the agent does not.

The folders controlled by the agent should be restricted to as few users as possible and they contain secrets that could be decrypted or exfiltrated.

The Azure Pipelines agent is a software product designed to execute code it downloads from external sources. It inherently could be a target for Remote Code Execution (RCE) attacks.

Therefore, it is important to consider the threat model surrounding each individual usage of Pipelines Agents to perform work, and decide what are the minimum permissions could be granted to the user running the agent, to the machine where the agent runs, to the users who have write access to the Pipeline definition, the git repos where the yaml is stored, or the group of users who control access to the pool for new pipelines.

It is a best practice to have the identity running the agent be different from the identity with permissions to connect the agent to the pool. The user generating the credentials (and other agent-related files) is different than the user that needs to read them. Therefore, it is safer to carefully consider access granted to the agent machine itself, and the agent folders which contain sensitive files, such as logs and artifacts.

It makes sense to grant access to the agent folder only for DevOps administrators and the user identity running the agent process. Administrators may need to investigate the file system to understand build failures or get log files to be able to report Azure DevOps failures.

Decide which user you'll use

As a one-time step, you must register the agent. Someone with permission to administer the agent queue must complete these steps. The agent will not use this person's credentials in everyday operation, but they're required to complete registration. Learn more about how agents communicate.

Authenticate with a personal access token (PAT)

1. Sign in with the user account you plan to use in your Azure DevOps organization (`https://dev.azure.com/{your_organization}`).

2. From your home page, open your user settings, and then select **Personal access tokens**.

3. Create a personal access token.

4. For the scope select **Agent Pools (read, manage)** and make sure all the other boxes are cleared. If it's a deployment group agent, for the scope select **Deployment group (read, manage)** and make sure all the other boxes are cleared.

   Select **Show all scopes** at the bottom of the **Create a new personal access token window** window to see the complete list of scopes.

5. Copy the token. You'll use this token when you configure the agent.

Confirm the user has permission

Make sure the user account that you're going to use has permission to register the agent.

Is the user an Azure DevOps organization owner or TFS or Azure DevOps Server administrator? **Stop here**, you have permission.

Otherwise:

1. Open a browser and navigate to the **Agent pools** tab for your Azure Pipelines organization or Azure DevOps Server or TFS server:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Select the pool on the right side of the page and then click **Security**.

3. If the user account you're going to use is not shown, then get an administrator to add it. The administrator can be an agent pool administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   If it's a deployment group agent, the administrator can be a deployment group administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   You can add a user to the deployment group administrator role in the **Security** tab on the **Deployment Groups** page in **Azure Pipelines**.

> ⓘ **Note**
>
> If you see a message like this: **Sorry, we couldn't add the identity. Please try a different identity.**, you probably followed the above steps for an organization owner or TFS or Azure DevOps Server administrator. You don't need to do anything; you already have permission to administer the agent queue.

Download and configure the agent

Azure Pipelines

1. Log on to the machine using the account for which you've prepared permissions as explained above.

2. In your web browser, sign in to Azure Pipelines, and navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

3. Select the **Default** pool, select the **Agents** tab, and choose **New agent**.

4. On the **Get the agent** dialog box, click **Linux**.

5. On the left pane, select the specific flavor. We offer x64 or ARM for most Linux distributions.

6. On the right pane, click the **Download** button.

7. Follow the instructions on the page.

8. Unpack the agent into the directory of your choice. `cd` to that directory and run `./config.sh`.

Server URL

Azure Pipelines: `https://dev.azure.com/{your-organization}`

Authentication type

Azure Pipelines

Choose **PAT**, and then paste the PAT token you created into the command prompt window.

> **Note**
>
> When using PAT as the authentication method, the PAT token is used only for the initial configuration of the agent. Learn more at **Communication with Azure Pipelines or TFS**.

TFS or Azure DevOps Server

> **Important**
>
> Make sure your server is **configured to support the authentication method** you want to use.

When you configure your agent to connect to TFS, you've got the following options:

- **Alternate** Connect to TFS or Azure DevOps Server using Basic authentication. After you select Alternate you'll be prompted for your credentials.
- **Integrated** Not supported on macOS or Linux.
- **Negotiate** (Default) Connect to TFS or Azure DevOps Server as a user other than the signed-in user via a Windows authentication scheme such as NTLM or Kerberos. After you select Negotiate you'll be prompted for credentials.

- **PAT** Supported only on Azure Pipelines and TFS 2017 and newer. After you choose PAT, paste the PAT token you created into the command prompt window. Use a personal access token (PAT) if your Azure DevOps Server or TFS instance and the agent machine are not in a trusted domain. PAT authentication is handled by your Azure DevOps Server or TFS instance instead of the domain controller.

> **Note**
>
> When using PAT as the authentication method, the PAT token is used only for the initial configuration of the agent on Azure DevOps Server and the newer versions of TFS. Learn more at **Communication with Azure Pipelines or TFS**.

Run interactively

For guidance on whether to run the agent in interactive mode or as a service, see Agents: Interactive vs. service.

To run the agent interactively:

1. If you have been running the agent as a service, uninstall the service.
2. Run the agent.

```bash
./run.sh
```

To restart the agent, press Ctrl+C and then run `run.sh` to restart it.

To use your agent, run a job using the agent's pool. If you didn't choose a different pool, your agent will be in the **Default** pool.

Run once

For agents configured to run interactively, you can choose to have the agent accept only one job. To run in this configuration:

```bash
./run.sh --once
```

Agents in this mode will accept only one job and then spin down gracefully (useful for running in Docker on a service like Azure Container Instances).

Run as a systemd service

If your agent is running on these operating systems you can run the agent as a `systemd` service:

- Ubuntu 16 LTS or newer
- Red Hat 7.1 or newer

We provide an example `./svc.sh` script for you to run and manage your agent as a `systemd` service. This script will be generated after you configure the agent. We encourage you to review, and if needed, update the script before running it.

Some important caveats:

- If you run your agent as a service, you cannot run the agent service as `root` user.
- Users running SELinux have reported difficulties with the provided `svc.sh` script. Refer to this agent issue as a starting point. SELinux is not an officially supported configuration.

> **Note**
>
> If you have a different distribution, or if you prefer other approaches, you can use whatever kind of service mechanism you prefer. See **Service files**.

Commands

Change to the agent directory

For example, if you installed in the `myagent` subfolder of your home directory:

```bash
cd ~/myagent$
```

Install

Command:

```bash
sudo ./svc.sh install [username]
```

This command creates a service file that points to `./runsvc.sh`. This script sets up the environment (more details below) and starts the agents host. If `username` parameter is not specified then the username is taken from the $SUDO_USER environment variable which is set by sudo command. This variable is always equal to the name of the user who invoked the `sudo` command.

Start

```bash
sudo ./svc.sh start
```

Status

```bash
sudo ./svc.sh status
```

Stop

```bash
sudo ./svc.sh stop
```

Uninstall

> You should stop before you uninstall.

```bash
sudo ./svc.sh uninstall
```

Update environment variables

When you configure the service, it takes a snapshot of some useful environment variables for your current logon user such as PATH, LANG, JAVA_HOME, ANT_HOME, and MYSQL_PATH. If you need to update the variables (for example, after installing some new software):

```bash
./env.sh
sudo ./svc.sh stop
sudo ./svc.sh start
```

The snapshot of the environment variables is stored in `.env` file (`PATH` is stored in `.path`) under agent root directory, you can also change these files directly to apply environment variable changes.

Run instructions before the service starts

You can also run your own instructions and commands to run when the service starts. For example, you could set up the environment or call scripts.

1. Edit `runsvc.sh`.
2. Replace the following line with your instructions:

   ```bash
   # insert anything to setup env when running as a service
   ```

Service files

When you install the service, some service files are put in place.

systemd service file

A systemd service file is created:

`/etc/systemd/system/vsts.agent.{tfs-name}.{agent-name}.service`

For example, you have configured an agent (see above) with the name `our-linux-agent`. The service file will be either:

- **Azure Pipelines**: the name of your organization. For example if you connect to `https://dev.azure.com/fabrikam`, then the service name would be

`/etc/systemd/system/vsts.agent.fabrikam.our-linux-agent.service`

- **TFS or Azure DevOps Server**: the name of your on-premises server. For example if you connect to `http://our-server:8080/tfs`, then the service name would be `/etc/systemd/system/vsts.agent.our-server.our-linux-agent.service`

`sudo ./svc.sh install` generates this file from this template: `./bin/vsts.agent.service.template`

.service file

`sudo ./svc.sh start` finds the service by reading the `.service` file, which contains the name of systemd service file described above.

Alternative service mechanisms

We provide the `./svc.sh` script as a convenient way for you to run and manage your agent as a systemd service. But you can use whatever kind of service mechanism you prefer (for example: initd or upstart).

You can use the template described above as to facilitate generating other kinds of service files.

Use a cgroup to avoid agent failure

It's important to avoid situations in which the agent fails or become unusable because otherwise the agent can't stream pipeline logs or report pipeline status back to the server. You can mitigate the risk of this kind of problem being caused by high memory pressure by using cgroups and a lower `oom_score_adj`. After you've done this, Linux reclaims system memory from pipeline job processes before reclaiming memory from the agent process. Learn how to configure cgroups and OOM score.

Replace an agent

To replace an agent, follow the *Download and configure the agent* steps again.

When you configure an agent using the same name as an agent that already exists, you're asked if you want to replace the existing agent. If you answer `Y`, then make sure you remove the agent (see below) that you're replacing. Otherwise, after a few minutes of conflicts, one of the agents will shut down.

Remove and re-configure an agent

To remove the agent:

1. Stop and uninstall the service as explained above.
2. Remove the agent.

   ```bash
   ./config.sh remove
   ```

3. Enter your credentials.

After you've removed the agent, you can configure it again.

Unattended config

The agent can be set up from a script with no human intervention. You must pass `--unattended` and the answers to all questions.

To configure an agent, it must know the URL to your organization or collection and credentials of someone authorized to set up agents. All other responses are optional. Any command-line parameter can be specified using an environment variable instead: put its name in upper case and prepend `VSTS_AGENT_INPUT_`. For example, `VSTS_AGENT_INPUT_PASSWORD` instead of specifying `--password`.

Required options

- `--unattended` - agent setup will not prompt for information, and all settings must be provided on the command line
- `--url <url>` - URL of the server. For example: https://dev.azure.com/myorganization or http://my-azure-devops-server:8080/tfs
- `--auth <type>` - authentication type. Valid values are:
  - `pat` (Personal access token) - PAT is the only scheme that works with Azure DevOps Services.
  - `negotiate` (Kerberos or NTLM)
  - `alt` (Basic authentication)
  - `integrated` (Windows default credentials)

Authentication options

- If you chose `--auth pat`:
  - `--token <token>` - specifies your personal access token
  - PAT is the only scheme that works with Azure DevOps Services.
- If you chose `--auth negotiate` or `--auth alt`:
  - `--userName <userName>` - specifies a Windows username in the format `domain\userName` or `userName@domain.com`
  - `--password <password>` - specifies a password

Pool and agent names

- `--pool <pool>` - pool name for the agent to join
- `--agent <agent>` - agent name
- `--replace` - replace the agent in a pool. If another agent is listening by the same name, it will start failing with a conflict

Agent setup

- `--work <workDirectory>` - work directory where job data is stored. Defaults to `_work` under the root of the agent directory. The work directory is owned by a given agent and should not be shared between multiple agents.
- `--acceptTeeEula` - accept the Team Explorer Everywhere End User License Agreement (macOS and Linux only)
- `--disableloguploads` - don't stream or send console log output to the server. Instead, you may retrieve them from the agent host's filesystem after the job completes.

Windows-only startup

- `--runAsService` - configure the agent to run as a Windows service (requires administrator permission)
- `--runAsAutoLogon` - configure auto-logon and run the agent on startup (requires administrator permission)
- `--windowsLogonAccount <account>` - used with `--runAsService` or `--runAsAutoLogon` to specify the Windows user name in the format `domain\userName` or `userName@domain.com`
- `--windowsLogonPassword <password>` - used with `--runAsService` or `--runAsAutoLogon` to specify Windows logon password (not required for Group

Managed Service Accounts and Windows built in accounts such as 'NT AUTHORITY\NETWORK SERVICE')

- `--overwriteAutoLogon` - used with `--runAsAutoLogon` to overwrite the existing auto logon on the machine
- `--noRestart` - used with `--runAsAutoLogon` to stop the host from restarting after agent configuration completes

Troubleshooting configuring the agent with the `runAsAutoLogon` option

Configuring the agent with the `runAsAutoLogon` option runs the agent each time after restarting the machine. Perform next steps if the agent is not run after restarting the machine.

If the agent was already configured on the machine

Before reconfiguring the agent, it is necessary to remove the old agent configuration, so try to run this command from the agent folder:

```
.\config.cmd remove --auth 'PAT' --token '<token>'
```

Check if the agent was removed from your agent pool after executing the command:

```
<Azure DevOps organization> / <Project> / Settings / Agent pools / <Agent
Pool> / Agents
```

Remove the agent from your agent pool manually if it was not removed by running the command.

Then try to reconfigure the agent by running this command from the agent folder:

```
.\config.cmd --unattended --agent '<agent-name>' --pool '<agent-pool-name>'
--url '<azure-dev-ops-organization-url>' --auth 'PAT' --token '<token>' --
runAsAutoLogon --windowsLogonAccount '<domain\user-name>' --
windowsLogonPassword '<windows-password>'
```

Specify the agent name (any specific unique name) and check if this agent appeared in your agent pool after reconfiguring.

It will be much better to unpack an agent archive (which can be downloaded here) and run this command from the new unpacked agent folder.

Check if the Windows registry key is recorded and saved correctly

Run the `whoami /user` command to get the `<sid>`. Open `Registry Editor` and follow the path:

```
Computer\HKEY_USERS\<sid>\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

Check if there is the `VSTSAgent` key. Delete this key if it exists, then close `Registry Editor` and configure the agent by running the `.\config.cmd` command (without args) from the agent folder. Before answering the question `Enter Restart the machine at a later time?`, open `Registry Editor` again and check if the `VSTSAgent` key has appeared. Press `Enter` to answer the question, and check if the `VSTSAgent` key remains in its place after restarting the machine.

Check if Windows registry keys work fine on your machine

Create a `autorun.cmd` file that contains the following line: `echo "Hello from AutoRun!"`. Open `Registry Editor` and create in the path above a new key-value pair with the key `AutoRun` and the value

```
C:\windows\system32\cmd.exe /D /S /C start "AutoRun"
"D:\path\to\autorun.cmd"
```

Restart your machine. You have an issue with Windows registry keys if you do not see a console window with the `Hello from AutoRun!` message.

Deployment group only

- `--deploymentGroup` - configure the agent as a deployment group agent
- `--deploymentGroupName <name>` - used with `--deploymentGroup` to specify the deployment group for the agent to join

- `--projectName <name>` - used with `--deploymentGroup` to set the project name
- `--addDeploymentGroupTags` - used with `--deploymentGroup` to indicate that deployment group tags should be added
- `--deploymentGroupTags <tags>` - used with `--addDeploymentGroupTags` to specify the comma separated list of tags for the deployment group agent - for example "web, db"

Environments only

- `--addvirtualmachineresourcetags` - used to indicate that environment resource tags should be added
- `--virtualmachineresourcetags <tags>` - used with `--addvirtualmachineresourcetags` to specify the comma separated list of tags for the environment resource agent - for example "web, db"

`./config.sh --help` always lists the latest required and optional responses.

Diagnostics

If you're having trouble with your self-hosted agent, you can try running diagnostics. After configuring the agent:

```bash
./run.sh --diagnostics
```

This will run through a diagnostic suite that may help you troubleshoot the problem. The diagnostics feature is available starting with agent version 2.165.0.

Help on other options

To learn about other options:

```bash
./config.sh --help
```

The help provides information on authentication alternatives and unattended configuration.

Capabilities

Your agent's capabilities are cataloged and advertised in the pool so that only the builds and releases it can handle are assigned to it. See Build and release agent capabilities.

In many cases, after you deploy an agent, you'll need to install software or utilities. Generally you should install on your agents whatever software and tools you use on your development machine.

For example, if your build includes the npm task, then the build won't run unless there's a build agent in the pool that has npm installed.

> **Important**
>
> Capabilities include all environment variables and the values that are set when the agent runs. If any of these values change while the agent is running, the agent must be restarted to pick up the new values. After you install new software on an agent, you must restart the agent for the new capability to show up in the pool, so that the build can run.
>
> If you want to exclude environment variables as capabilities, you can designate them by setting an environment variable `VSO_AGENT_IGNORE` with a comma-delimited list of variables to ignore.

FAQ

How do I make sure I have the latest agent version?

1. Navigate to the **Agent pools** tab:

    a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

    b. Choose **Azure DevOps**, **Organization settings**.

c. Choose **Agent pools**.

2. Click the pool that contains the agent.

3. Make sure the agent is enabled.

4. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

b. Select **Agents** and choose the desired agent.

c. Choose the **Capabilities** tab.

> **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

5. Look for the `Agent.Version` capability. You can check this value against the latest published agent version. See Azure Pipelines Agent and check the page for the highest version number listed.

6. Each agent automatically updates itself when it runs a task that requires a newer version of the agent. If you want to manually update some agents, right-click the pool, and select **Update all agents**.

How do I make sure I have the latest v2 agent version?

1. Navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

c. Choose **Agent pools**.

2. Click the pool that contains the agent.

3. Make sure the agent is enabled.

4. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

b. Select **Agents** and choose the desired agent.

c. Choose the **Capabilities** tab.

> ⓘ **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

5. Look for the `Agent.Version` capability. You can check this value against the latest published agent version. See Azure Pipelines Agent and check the page for the highest version number listed.

6. Each agent automatically updates itself when it runs a task that requires a newer version of the agent. If you want to manually update some agents, right-click the pool, and select **Update all agents**.

Why is sudo needed to run the service commands?

`./svc.sh` uses `systemctl`, which requires `sudo`.

Source code: systemd.svc.sh.template on GitHub

I'm running a firewall and my code is in Azure Repos. What URLs does the agent need to communicate with?

If you're running an agent in a secure network behind a firewall, make sure the agent can initiate communication with the following URLs and IP addresses.

Domain URL	Description
`https://{organization_name}.pkgs.visualstudio.com`	Azure DevOps Packaging API for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsrm.visualstudio.com`	Release Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://*.blob.core.windows.net`	Azure Artifacts
`https://*.dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://*.vsassets.io`	Azure Artifacts via CDN
`https://*.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `dev.azure.com` domain
`https://*.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `dev.azure.com` domain
`https://*.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the

Domain URL	Description
	`dev.azure.com` domain
`https://app.vssps.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://login.microsoftonline.com`	Azure Active Directory sign in
`https://management.core.windows.net`	Azure Management API's
`https://vstsagentpackage.azureedge.net`	Agent package

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> For more information about allowed addresses, see **Allowed address lists and network connections**.

How do I run the agent with self-signed certificate?

Run the agent with self-signed certificate

How do I run the agent behind a web proxy?

Run the agent behind a web proxy

How do I restart the agent

If you are running the agent interactively, see the restart instructions in Run interactively. If you are running the agent as a systemd service, follow the steps to Stop and then Start the agent.

How do I configure the agent to bypass a web proxy and connect to Azure Pipelines?

If you want the agent to bypass your proxy and connect to Azure Pipelines directly, then you should configure your web proxy to enable the agent to access the following URLs.

For organizations using the `*.visualstudio.com` domain:

```
https://login.microsoftonline.com
https://app.vssps.visualstudio.com
https://{organization_name}.visualstudio.com
https://{organization_name}.vsrm.visualstudio.com
https://{organization_name}.vstmr.visualstudio.com
https://{organization_name}.pkgs.visualstudio.com
https://{organization_name}.vssps.visualstudio.com
```

For organizations using the `dev.azure.com` domain:

```
https://dev.azure.com
https://*.dev.azure.com
https://login.microsoftonline.com
https://management.core.windows.net
https://vstsagentpackage.azureedge.net
https://vssps.dev.azure.com
```

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> This procedure enables the agent to bypass a web proxy. Your build pipeline and scripts must still handle bypassing your web proxy for each task and tool you run in your build.
>
> For example, if you are using a NuGet task, you must configure your web proxy to support bypassing the URL for the server that hosts the NuGet feed you're using.

I'm using TFS and the URLs in the sections above don't work for me. Where can I get help?

Web site settings and security

Self-hosted macOS agents (2.x)

Article • 05/10/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> **Important**
>
> This article provides guidance for using the 2.x version agent software with Azure DevOps Server and TFS. If you're using Azure DevOps Services, see **Self-hosted macOS agents**.

To build and deploy Xcode apps or Xamarin.iOS projects, you'll need at least one macOS agent. This agent can also build and deploy Java and Android apps.

> Before you begin:
>
> - If your pipelines are in Azure Pipelines and a Microsoft-hosted agent meets your needs, you can skip setting up a self-hosted macOS agent.
> - Otherwise, you've come to the right place to set up an agent on macOS. Continue to the next section.

Learn about agents

If you already know what an agent is and how it works, feel free to jump right in to the following sections. But if you'd like some more background about what they do and how they work, see Azure Pipelines agents.

Check prerequisites

Make sure your machine has these prerequisites:

- macOS 10.15 "Catalina", macOS 11.0 "Big Sur", or macOS 12.0 "Monterey"
- Git 2.9.0 or higher (latest version strongly recommended - you can easily install with Homebrew)

These prereqs are required for agent version 2.125.0 and higher.

TFVC

If you'll be using TFVC, you'll also need the Oracle Java JDK 1.6 or higher. (The Oracle JRE and OpenJDK aren't sufficient for this purpose.)

TEE plugin is used for TFVC functionality. It has an EULA, which you'll need to accept during configuration if you plan to work with TFVC.

Since the TEE plugin is no longer maintained and contains some out-of-date Java dependencies, starting from Agent 2.198.0 it's no longer included in the agent distribution. However, the TEE plugin will be downloaded during checkout task execution if you're checking out a TFVC repo. The TEE plugin will be removed after the job execution.

> **Note**
>
> Note: You may notice your checkout task taking a long time to start working because of this download mechanism.

If the agent is running behind a proxy or a firewall, you'll need to ensure access to the following site: `https://vstsagenttools.blob.core.windows.net/`. The TEE plugin will be downloaded from this address.

If you're using a self-hosted agent and facing issues with TEE downloading, you may install TEE manually:

1. Set `DISABLE_TEE_PLUGIN_REMOVAL` environment or pipeline variable to `true`. This variable prevents the agent from removing the TEE plugin after TFVC repository checkout.
2. Download TEE-CLC version 14.135.0 manually from Team Explorer Everywhere GitHub releases.
3. Extract the contents of `TEE-CLC-14.135.0` folder to `<agent_directory>/externals/tee`.

Prepare permissions

If you're building from a Subversion repo, you must install the Subversion client on the machine.

You should run agent setup manually the first time. After you get a feel for how agents work, or if you want to automate setting up many agents, consider using unattended config.

Information security for self-hosted agents

The user configuring the agent needs pool admin permissions, but the user running the agent does not.

The folders controlled by the agent should be restricted to as few users as possible and they contain secrets that could be decrypted or exfiltrated.

The Azure Pipelines agent is a software product designed to execute code it downloads from external sources. It inherently could be a target for Remote Code Execution (RCE) attacks.

Therefore, it is important to consider the threat model surrounding each individual usage of Pipelines Agents to perform work, and decide what are the minimum permissions could be granted to the user running the agent, to the machine where the agent runs, to the users who have write access to the Pipeline definition, the git repos where the yaml is stored, or the group of users who control access to the pool for new pipelines.

It is a best practice to have the identity running the agent be different from the identity with permissions to connect the agent to the pool. The user generating the credentials (and other agent-related files) is different than the user that needs to read them. Therefore, it is safer to carefully consider access granted to the agent machine itself, and the agent folders which contain sensitive files, such as logs and artifacts.

It makes sense to grant access to the agent folder only for DevOps administrators and the user identity running the agent process. Administrators may need to investigate the file system to understand build failures or get log files to be able to report Azure DevOps failures.

Decide which user you'll use

As a one-time step, you must register the agent. Someone with permission to administer the agent queue must complete these steps. The agent will not use this person's credentials in everyday operation, but they're required to complete registration. Learn more about how agents communicate.

Authenticate with a personal access token (PAT)

1. Sign in with the user account you plan to use in your Azure DevOps organization (`https://dev.azure.com/{your_organization}`).

2. From your home page, open your user settings, and then select **Personal access tokens**.

3. Create a personal access token.

4. For the scope select **Agent Pools (read, manage)** and make sure all the other boxes are cleared. If it's a deployment group agent, for the scope select **Deployment group (read, manage)** and make sure all the other boxes are cleared.

Select **Show all scopes** at the bottom of the **Create a new personal access token window** window to see the complete list of scopes.

5. Copy the token. You'll use this token when you configure the agent.

Confirm the user has permission

Make sure the user account that you're going to use has permission to register the agent.

Is the user an Azure DevOps organization owner or TFS or Azure DevOps Server administrator? **Stop here**, you have permission.

Otherwise:

1. Open a browser and navigate to the **Agent pools** tab for your Azure Pipelines organization or Azure DevOps Server or TFS server:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Select the pool on the right side of the page and then click **Security**.

3. If the user account you're going to use is not shown, then get an administrator to add it. The administrator can be an agent pool administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   If it's a deployment group agent, the administrator can be a deployment group administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   You can add a user to the deployment group administrator role in the **Security** tab on the **Deployment Groups** page in **Azure Pipelines**.

> ⓘ **Note**
>
> If you see a message like this: **Sorry, we couldn't add the identity. Please try a different identity.**, you probably followed the above steps for an organization owner or TFS or Azure DevOps Server administrator. You don't need to do anything; you already have permission to administer the agent queue.

Download and configure the agent

Azure Pipelines

1. Log on to the machine using the account for which you've prepared permissions as explained above.

2. In your web browser, sign in to Azure Pipelines, and navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

3. Select the **Default** pool, select the **Agents** tab, and choose **New agent**.

4. On the **Get the agent** dialog box, click **macOS**.

5. Click the **Download** button.

6. Follow the instructions on the page.

7. Clear the extended attribute on the tar file: `xattr -c vsts-agent-osx-x64-V.v.v.tar.gz`.

8. Unpack the agent into the directory of your choice. `cd` to that directory and run `./config.sh`. Make sure that the path to the directory contains no spaces because tools and scripts don't always properly escape spaces.

Server URL

Azure Pipelines: `https://dev.azure.com/{your-organization}`

Authentication type

Azure Pipelines

Choose **PAT**, and then paste the PAT token you created into the command prompt window.

> ⓘ **Note**
>
> When using PAT as the authentication method, the PAT token is used only for the initial configuration of the agent. Learn more at **Communication with Azure Pipelines or TFS**.

TFS or Azure DevOps Server

> ⓘ **Important**
>
> Make sure your server is **configured to support the authentication method** you want to use.

When you configure your agent to connect to TFS, you've got the following options:

- **Alternate** Connect to TFS or Azure DevOps Server using Basic authentication. After you select Alternate you'll be prompted for your credentials.
- **Integrated** Not supported on macOS or Linux.
- **Negotiate** (Default) Connect to TFS or Azure DevOps Server as a user other than the signed-in user via a Windows authentication scheme such as NTLM or

Kerberos. After you select Negotiate you'll be prompted for credentials.

- **PAT** Supported only on Azure Pipelines and TFS 2017 and newer. After you choose PAT, paste the PAT token you created into the command prompt window. Use a personal access token (PAT) if your Azure DevOps Server or TFS instance and the agent machine are not in a trusted domain. PAT authentication is handled by your Azure DevOps Server or TFS instance instead of the domain controller.

> **Note**
>
> When using PAT as the authentication method, the PAT token is used only for the initial configuration of the agent on Azure DevOps Server and the newer versions of TFS. Learn more at **Communication with Azure Pipelines or TFS**.

Run interactively

For guidance on whether to run the agent in interactive mode or as a service, see Agents: Interactive vs. service.

To run the agent interactively:

1. If you have been running the agent as a service, uninstall the service.
2. Run the agent.

```bash
./run.sh
```

To restart the agent, press Ctrl+C and then run `run.sh` to restart it.

To use your agent, run a job using the agent's pool. If you didn't choose a different pool, your agent will be in the **Default** pool.

Run once

For agents configured to run interactively, you can choose to have the agent accept only one job. To run in this configuration:

```bash
./run.sh --once
```

Agents in this mode will accept only one job and then spin down gracefully (useful for running on a service like Azure Container Instances).

Run as a launchd service

We provide the `./svc.sh` script for you to run and manage your agent as a launchd LaunchAgent service. This script will be generated after you configure the agent. The service has access to the UI to run your UI tests.

> **Note**
>
> If you prefer other approaches, you can use whatever kind of service mechanism you prefer. See **Service files**.

Tokens

In the section below, these tokens are replaced:

- `{agent-name}`
- `{tfs-name}`

For example, you have configured an agent (see above) with the name `our-osx-agent`. In the following examples, `{tfs-name}` will be either:

- Azure Pipelines: the name of your organization. For example if you connect to `https://dev.azure.com/fabrikam`, then the service name would be `vsts.agent.fabrikam.our-osx-agent`
- TFS: the name of your on-premises TFS AT server. For example if you connect to `http://our-server:8080/tfs`, then the service name would be `vsts.agent.our-server.our-osx-agent`

Commands

Change to the agent directory

For example, if you installed in the `myagent` subfolder of your home directory:

```
Bash
```

```
cd ~/myagent$
```

Install

Command:

```bash
./svc.sh install
```

This command creates a launchd plist that points to `./runsvc.sh`. This script sets up the environment (more details below) and starts the agent's host.

Start

Command:

```bash
./svc.sh start
```

Output:

```bash
starting vsts.agent.{tfs-name}.{agent-name}
status vsts.agent.{tfs-name}.{agent-name}:

/Users/{your-name}/Library/LaunchAgents/vsts.agent.{tfs-name}.{agent-name}.plist

Started:
13472 0 vsts.agent.{tfs-name}.{agent-name}
```

The left number is the pid if the service is running. If second number is not zero, then a problem occurred.

Status

Command:

```bash
```

```
./svc.sh status
```

Output:

```bash
status vsts.agent.{tfs-name}.{agent-name}:

/Users/{your-name}/Library/LaunchAgents/vsts.{tfs-name}.{agent-name}.testsvc.plist

Started:
13472 0 vsts.agent.{tfs-name}.{agent-name}
```

The left number is the pid if the service is running. If second number is not zero, then a problem occurred.

Stop

Command:

```bash
./svc.sh stop
```

Output:

```bash
stopping vsts.agent.{tfs-name}.{agent-name}
status vsts.agent.{tfs-name}.{agent-name}:

/Users/{your-name}/Library/LaunchAgents/vsts.{tfs-name}.{agent-name}.testsvc.plist

Stopped
```

Uninstall

> You should stop before you uninstall.

Command:

```bash
./svc.sh uninstall
```

Automatic login and lock

Normally, the agent service runs only after the user logs in. If you want the agent service to automatically start when the machine restarts, you can configure the machine to automatically login and lock on startup. See Set your Mac to automatically login during startup - Apple Support.

> **Note**
>
> For more information, see the **Terminally Geeky: use automatic login more securely** blog. The .plist file mentioned in that blog may no longer be available at the source, but a copy can be found here: **Lifehacker - Make OS X load your desktop before you log in**.

Update environment variables

When you configure the service, it takes a snapshot of some useful environment variables for your current logon user such as PATH, LANG, JAVA_HOME, ANT_HOME, and MYSQL_PATH. If you need to update the variables (for example, after installing some new software):

```bash
./env.sh
./svc.sh stop
./svc.sh start
```

The snapshot of the environment variables is stored in `.env` file under agent root directory, you can also change that file directly to apply environment variable changes.

Run instructions before the service starts

You can also run your own instructions and commands to run when the service starts. For example, you could set up the environment or call scripts.

1. Edit `runsvc.sh`.

2. Replace the following line with your instructions:

```bash
# insert anything to setup env when running as a service
```

Service Files

When you install the service, some service files are put in place.

.plist service file

A .plist service file is created:

```
~/Library/LaunchAgents/vsts.agent.{tfs-name}.{agent-name}.plist
```

For example:

```
~/Library/LaunchAgents/vsts.agent.fabrikam.our-osx-agent.plist
```

`sudo ./svc.sh install` generates this file from this template: `./bin/vsts.agent.plist.template`

.service file

`./svc.sh start` finds the service by reading the `.service` file, which contains the path to the plist service file described above.

Alternative service mechanisms

We provide the `./svc.sh` script as a convenient way for you to run and manage your agent as a launchd LaunchAgent service. But you can use whatever kind of service mechanism you prefer.

You can use the template described above as to facilitate generating other kinds of service files. For example, you modify the template to generate a service that runs as a

launch daemon if you don't need UI tests and don't want to configure automatic log on and lock. See Apple Developer Library: Creating Launch Daemons and Agents.

Replace an agent

To replace an agent, follow the *Download and configure the agent* steps again.

When you configure an agent using the same name as an agent that already exists, you're asked if you want to replace the existing agent. If you answer `Y`, then make sure you remove the agent (see below) that you're replacing. Otherwise, after a few minutes of conflicts, one of the agents will shut down.

Remove and re-configure an agent

To remove the agent:

1. Stop and uninstall the service as explained above.
2. Remove the agent.

   ```bash
   ./config.sh remove
   ```

3. Enter your credentials.

After you've removed the agent, you can configure it again.

Unattended config

The agent can be set up from a script with no human intervention. You must pass `--unattended` and the answers to all questions.

To configure an agent, it must know the URL to your organization or collection and credentials of someone authorized to set up agents. All other responses are optional. Any command-line parameter can be specified using an environment variable instead: put its name in upper case and prepend `VSTS_AGENT_INPUT_`. For example, `VSTS_AGENT_INPUT_PASSWORD` instead of specifying `--password`.

Required options

- `--unattended` - agent setup will not prompt for information, and all settings must be provided on the command line
- `--url <url>` - URL of the server. For example: https://dev.azure.com/myorganization or http://my-azure-devops-server:8080/tfs
- `--auth <type>` - authentication type. Valid values are:
  - `pat` (Personal access token) - PAT is the only scheme that works with Azure DevOps Services.
  - `negotiate` (Kerberos or NTLM)
  - `alt` (Basic authentication)
  - `integrated` (Windows default credentials)

Authentication options

- If you chose `--auth pat`:
  - `--token <token>` - specifies your personal access token
  - PAT is the only scheme that works with Azure DevOps Services.
- If you chose `--auth negotiate` or `--auth alt`:
  - `--userName <userName>` - specifies a Windows username in the format `domain\userName` or `userName@domain.com`
  - `--password <password>` - specifies a password

Pool and agent names

- `--pool <pool>` - pool name for the agent to join
- `--agent <agent>` - agent name
- `--replace` - replace the agent in a pool. If another agent is listening by the same name, it will start failing with a conflict

Agent setup

- `--work <workDirectory>` - work directory where job data is stored. Defaults to `_work` under the root of the agent directory. The work directory is owned by a given agent and should not be shared between multiple agents.
- `--acceptTeeEula` - accept the Team Explorer Everywhere End User License Agreement (macOS and Linux only)
- `--disableloguploads` - don't stream or send console log output to the server. Instead, you may retrieve them from the agent host's filesystem after the job completes.

Windows-only startup

- `--runAsService` - configure the agent to run as a Windows service (requires administrator permission)
- `--runAsAutoLogon` - configure auto-logon and run the agent on startup (requires administrator permission)
- `--windowsLogonAccount <account>` - used with `--runAsService` or `--runAsAutoLogon` to specify the Windows user name in the format `domain\userName` or `userName@domain.com`
- `--windowsLogonPassword <password>` - used with `--runAsService` or `--runAsAutoLogon` to specify Windows logon password (not required for Group Managed Service Accounts and Windows built in accounts such as 'NT AUTHORITY\NETWORK SERVICE')
- `--overwriteAutoLogon` - used with `--runAsAutoLogon` to overwrite the existing auto logon on the machine
- `--noRestart` - used with `--runAsAutoLogon` to stop the host from restarting after agent configuration completes

Troubleshooting configuring the agent with the `runAsAutoLogon` option

Configuring the agent with the `runAsAutoLogon` option runs the agent each time after restarting the machine. Perform next steps if the agent is not run after restarting the machine.

If the agent was already configured on the machine

Before reconfiguring the agent, it is necessary to remove the old agent configuration, so try to run this command from the agent folder:

```
.\config.cmd remove --auth 'PAT' --token '<token>'
```

Check if the agent was removed from your agent pool after executing the command:

```
<Azure DevOps organization> / <Project> / Settings / Agent pools / <Agent
Pool> / Agents
```

Remove the agent from your agent pool manually if it was not removed by running the command.

Then try to reconfigure the agent by running this command from the agent folder:

```
.\config.cmd --unattended --agent '<agent-name>' --pool '<agent-pool-name>'
--url '<azure-dev-ops-organization-url>' --auth 'PAT' --token '<token>' --
runAsAutoLogon --windowsLogonAccount '<domain\user-name>' --
windowsLogonPassword '<windows-password>'
```

Specify the agent name (any specific unique name) and check if this agent appeared in your agent pool after reconfiguring.

It will be much better to unpack an agent archive (which can be downloaded here) and run this command from the new unpacked agent folder.

Check if the Windows registry key is recorded and saved correctly

Run the `whoami /user` command to get the `<sid>`. Open `Registry Editor` and follow the path:

```
Computer\HKEY_USERS\<sid>\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

Check if there is the `VSTSAgent` key. Delete this key if it exists, then close `Registry Editor` and configure the agent by running the `.\config.cmd` command (without args) from the agent folder. Before answering the question `Enter Restart the machine at a later time?`, open `Registry Editor` again and check if the `VSTSAgent` key has appeared. Press `Enter` to answer the question, and check if the `VSTSAgent` key remains in its place after restarting the machine.

Check if Windows registry keys work fine on your machine

Create a `autorun.cmd` file that contains the following line: `echo "Hello from AutoRun!"`. Open `Registry Editor` and create in the path above a new key-value pair with the key `AutoRun` and the value

```
C:\windows\system32\cmd.exe /D /S /C start "AutoRun"
"D:\path\to\autorun.cmd"
```

Restart your machine. You have an issue with Windows registry keys if you do not see a console window with the `Hello from AutoRun!` message.

Deployment group only

- `--deploymentGroup` - configure the agent as a deployment group agent
- `--deploymentGroupName <name>` - used with `--deploymentGroup` to specify the deployment group for the agent to join
- `--projectName <name>` - used with `--deploymentGroup` to set the project name
- `--addDeploymentGroupTags` - used with `--deploymentGroup` to indicate that deployment group tags should be added
- `--deploymentGroupTags <tags>` - used with `--addDeploymentGroupTags` to specify the comma separated list of tags for the deployment group agent - for example "web, db"

Environments only

- `--addvirtualmachineresourcetags` - used to indicate that environment resource tags should be added
- `--virtualmachineresourcetags <tags>` - used with `--addvirtualmachineresourcetags` to specify the comma separated list of tags for the environment resource agent - for example "web, db"

`./config.sh --help` always lists the latest required and optional responses.

Diagnostics

If you're having trouble with your self-hosted agent, you can try running diagnostics. After configuring the agent:

```bash
./run.sh --diagnostics
```

This will run through a diagnostic suite that may help you troubleshoot the problem. The diagnostics feature is available starting with agent version 2.165.0.

Help on other options

To learn about other options:

```bash
./config.sh --help
```

The help provides information on authentication alternatives and unattended configuration.

Capabilities

Your agent's capabilities are cataloged and advertised in the pool so that only the builds and releases it can handle are assigned to it. See Build and release agent capabilities.

In many cases, after you deploy an agent, you'll need to install software or utilities. Generally you should install on your agents whatever software and tools you use on your development machine.

For example, if your build includes the npm task, then the build won't run unless there's a build agent in the pool that has npm installed.

> **Important**
>
> Capabilities include all environment variables and the values that are set when the agent runs. If any of these values change while the agent is running, the agent must be restarted to pick up the new values. After you install new software on an agent, you must restart the agent for the new capability to show up in the pool, so that the build can run.
>
> If you want to exclude environment variables as capabilities, you can designate them by setting an environment variable `VSO_AGENT_IGNORE` with a comma-delimited list of variables to ignore.

FAQ

How do I make sure I have the latest agent version?

1. Navigate to the **Agent pools** tab:

a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

b. Choose **Azure DevOps**, **Organization settings**.

c. Choose **Agent pools**.

2. Click the pool that contains the agent.

3. Make sure the agent is enabled.

4. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

b. Select **Agents** and choose the desired agent.

c. Choose the **Capabilities** tab.

> ⓘ **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

5. Look for the `Agent.Version` capability. You can check this value against the latest published agent version. See Azure Pipelines Agent and check the page for the highest version number listed.

6. Each agent automatically updates itself when it runs a task that requires a newer version of the agent. If you want to manually update some agents, right-click the pool, and select **Update all agents**.

How do I make sure I have the latest v2 agent version?

1. Navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

c. Choose **Agent pools**.

2. Click the pool that contains the agent.

3. Make sure the agent is enabled.

4. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

b. Select **Agents** and choose the desired agent.

c. Choose the **Capabilities** tab.

> **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

5. Look for the `Agent.Version` capability. You can check this value against the latest published agent version. See Azure Pipelines Agent and check the page for the highest version number listed.

6. Each agent automatically updates itself when it runs a task that requires a newer version of the agent. If you want to manually update some agents, right-click the pool, and select **Update all agents**.

Where can I learn more about how the launchd service works?

Apple Developer Library: Creating Launch Daemons and Agents

I'm running a firewall and my code is in Azure Repos. What URLs does the agent need to communicate with?

If you're running an agent in a secure network behind a firewall, make sure the agent can initiate communication with the following URLs and IP addresses.

Expand table

Domain URL	Description
`https://{organization_name}.pkgs.visualstudio.com`	Azure DevOps Packaging API for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsrm.visualstudio.com`	Release Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://*.blob.core.windows.net`	Azure Artifacts
`https://*.dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://*.vsassets.io`	Azure Artifacts via CDN
`https://*.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `dev.azure.com` domain
`https://*.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `dev.azure.com` domain

Domain URL	Description
`https://*.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the `dev.azure.com` domain
`https://app.vssps.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://login.microsoftonline.com`	Microsoft Entra sign-in
`https://management.core.windows.net`	Azure Management API's
`https://vstsagentpackage.azureedge.net`	Agent package

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> For more information about allowed addresses, see **Allowed address lists and network connections**.

How do I run the agent with self-signed certificate?

Run the agent with self-signed certificate

How do I run the agent behind a web proxy?

Run the agent behind a web proxy

How do I restart the agent

If you are running the agent interactively, see the restart instructions in Run interactively. If you are running the agent as a service, follow the steps to Stop and then Start the agent.

How do I configure the agent to bypass a web proxy and connect to Azure Pipelines?

If you want the agent to bypass your proxy and connect to Azure Pipelines directly, then you should configure your web proxy to enable the agent to access the following URLs.

For organizations using the `*.visualstudio.com` domain:

```
https://login.microsoftonline.com
https://app.vssps.visualstudio.com
https://{organization_name}.visualstudio.com
https://{organization_name}.vsrm.visualstudio.com
https://{organization_name}.vstmr.visualstudio.com
https://{organization_name}.pkgs.visualstudio.com
https://{organization_name}.vssps.visualstudio.com
```

For organizations using the `dev.azure.com` domain:

```
https://dev.azure.com
https://*.dev.azure.com
https://login.microsoftonline.com
https://management.core.windows.net
https://vstsagentpackage.azureedge.net
https://vssps.dev.azure.com
```

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> This procedure enables the agent to bypass a web proxy. Your build pipeline and scripts must still handle bypassing your web proxy for each task and tool you run in your build.
>
> For example, if you are using a NuGet task, you must configure your web proxy to support bypassing the URL for the server that hosts the NuGet feed you're using.

I'm using TFS and the URLs in the sections above don't work for me. Where can I get help?

Web site settings and security

Self-hosted Windows agents (2.x)

Article • 05/10/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

> **Important**
>
> This article provides guidance for using the 2.x version agent software with Azure DevOps Server and TFS. If you're using Azure DevOps Services, see **Self-hosted Windows agents**.

To build and deploy Windows, Azure, and other Visual Studio solutions you'll need at least one Windows agent. Windows agents can also build Java and Android apps.

> Before you begin:
>
> - If your code is in Azure Pipelines and a Microsoft-hosted agent meets your needs, you can skip setting up a self-hosted Windows agent.
> - Otherwise, you've come to the right place to set up an agent on Windows. Continue to the next section.

Learn about agents

If you already know what an agent is and how it works, feel free to jump right in to the following sections. But if you'd like some more background about what they do and how they work, see Azure Pipelines agents.

Check prerequisites

Make sure your machine has these prerequisites:

- Windows 7 SP1 ESU, 8.1, 10, or 11 (if using a client OS)
- Windows 2012 or higher (if using a server OS)
- PowerShell 3.0 or higher
- .NET Framework 4.6.2 or higher

> **Important**

> Starting December 2019, the minimum required .NET version for build agents is 4.6.2 or higher.

Recommended:

- Visual Studio build tools (2015 or higher)

If you're building from a Subversion repo, you must install the Subversion client on the machine.

You should run agent setup manually the first time. After you get a feel for how agents work, or if you want to automate setting up many agents, consider using unattended config.

Hardware specs

The hardware specs for your agents will vary with your needs, team size, etc. It's not possible to make a general recommendation that will apply to everyone. As a point of reference, the Azure DevOps team builds the hosted agents code using pipelines that utilize hosted agents. On the other hand, the bulk of the Azure DevOps code is built by 24-core server class machines running 4 self-hosted agents apiece.

Prepare permissions

Information security for self-hosted agents

The user configuring the agent needs pool admin permissions, but the user running the agent does not.

The folders controlled by the agent should be restricted to as few users as possible and they contain secrets that could be decrypted or exfiltrated.

The Azure Pipelines agent is a software product designed to execute code it downloads from external sources. It inherently could be a target for Remote Code Execution (RCE) attacks.

Therefore, it is important to consider the threat model surrounding each individual usage of Pipelines Agents to perform work, and decide what are the minimum permissions could be granted to the user running the agent, to the machine where the agent runs, to the users who have write access to the Pipeline definition, the git repos where the yaml is stored, or the group of users who control access to the pool for new pipelines.

It is a best practice to have the identity running the agent be different from the identity with permissions to connect the agent to the pool. The user generating the credentials (and other agent-related files) is different than the user that needs to read them. Therefore, it is safer to carefully consider access granted to the agent machine itself, and the agent folders which contain sensitive files, such as logs and artifacts.

It makes sense to grant access to the agent folder only for DevOps administrators and the user identity running the agent process. Administrators may need to investigate the file system to understand build failures or get log files to be able to report Azure DevOps failures.

Decide which user you'll use

As a one-time step, you must register the agent. Someone with permission to administer the agent queue must complete these steps. The agent will not use this person's credentials in everyday operation, but they're required to complete registration. Learn more about how agents communicate.

Authenticate with a personal access token (PAT)

1. Sign in with the user account you plan to use in your Azure DevOps organization (`https://dev.azure.com/{your_organization}`).

2. From your home page, open your user settings, and then select **Personal access tokens**.

3. Create a personal access token.

4. For the scope select **Agent Pools (read, manage)** and make sure all the other boxes are cleared. If it's a deployment group agent, for the scope select **Deployment group (read, manage)** and make sure all the other boxes are cleared.

   Select **Show all scopes** at the bottom of the **Create a new personal access token window** window to see the complete list of scopes.

5. Copy the token. You'll use this token when you configure the agent.

Confirm the user has permission

Make sure the user account that you're going to use has permission to register the agent.

Is the user an Azure DevOps organization owner or TFS or Azure DevOps Server administrator? **Stop here**, you have permission.

Otherwise:

1. Open a browser and navigate to the **Agent pools** tab for your Azure Pipelines organization or Azure DevOps Server or TFS server:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Select the pool on the right side of the page and then click **Security**.

3. If the user account you're going to use is not shown, then get an administrator to add it. The administrator can be an agent pool administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   If it's a deployment group agent, the administrator can be a deployment group administrator, an Azure DevOps organization owner, or a TFS or Azure DevOps Server administrator.

   You can add a user to the deployment group administrator role in the **Security** tab on the **Deployment Groups** page in **Azure Pipelines**.

> ⓘ **Note**
>
> If you see a message like this: **Sorry, we couldn't add the identity. Please try a different identity.**, you probably followed the above steps for an organization owner or TFS or Azure DevOps Server administrator. You don't need to do anything; you already have permission to administer the agent queue.

Download and configure the agent

Azure Pipelines

1. Log on to the machine using the account for which you've prepared permissions as explained above.

2. In your web browser, sign in to Azure Pipelines, and navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

3. Select the **Default** pool, select the **Agents** tab, and choose **New agent**.

4. On the **Get the agent** dialog box, choose **Windows**.

5. On the left pane, select the processor architecture of the installed Windows OS version on your machine. The x64 version is intended for 64-bit Windows, whereas

the x86 version is intended for 32-bit Windows. If you aren't sure which version of Windows is installed, follow these instructions to find out.

6. On the right pane, click the **Download** button.

7. Follow the instructions on the page to download the agent.

8. Unpack the agent into the directory of your choice. Make sure that the path to the directory contains no spaces because tools and scripts don't always properly escape spaces. A recommended folder is `C:\agents`. Extracting in the download folder or other user folders may cause permission issues. Then run `config.cmd`. This will ask you a series of questions to configure the agent.

> **Important**
>
> We strongly recommend you configure the agent from an elevated PowerShell window. If you want to configure as a service, this is **required**.
>
> You must not use **Windows PowerShell ISE** to configure the agent.

> **Important**
>
> For security reasons we strongly recommend making sure the agents folder (`C:\agents`) is only editable by admins.

> **Note**
>
> Please avoid using mintty based shells, such as git-bash, for agent configuration. Mintty is not fully compatible with native Input/Output Windows API (**here** is some info about it) and we couldn't guarantee correct work of setup script in this case.

Server URL and authentication

When setup asks for your server URL, for Azure DevOps Services, answer `https://dev.azure.com/{your-organization}`.

When setup asks for your authentication type, choose **PAT**. Then paste the PAT token you created into the command prompt window.

> **Note**
>
> When using PAT as the authentication method, the PAT token is only used during the initial configuration of the agent. Later, if the PAT expires or needs to be renewed, no further changes are required by the agent.

Choose interactive or service mode

For guidance on whether to run the agent in interactive mode or as a service, see Agents: Interactive vs. service.

If you choose to run as a service (which we recommend), the username you run as should be 20 characters or fewer.

Run the agent

Run interactively

If you configured the agent to run interactively, to run it:

```ps
.\run.cmd
```

To restart the agent, press Ctrl+C to stop the agent and then run `run.cmd` to restart it.

Run once

For agents configured to run interactively, you can choose to have the agent accept only one job. To run in this configuration:

```ps
.\run.cmd --once
```

Agents in this mode will accept only one job and then spin down gracefully (useful for running in Docker on a service like Azure Container Instances).

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 restart the agent, right-click the entry and choose **Restart**.

> **Note**
>
> If you need to change the agent's logon account, don't do it from the Services snap-in. Instead, see the information below to re-configure the agent.

To use your agent, run a job using the agent's pool. If you didn't choose a different pool, your agent will be in the **Default** pool.

Replace an agent

To replace an agent, follow the *Download and configure the agent* steps again.

When you configure an agent using the same name as an agent that already exists, you're asked if you want to replace the existing agent. If you answer `Y`, then make sure you remove the agent (see below) that you're replacing. Otherwise, after a few minutes of conflicts, one of the agents will shut down.

Remove and re-configure an agent

To remove the agent:

```ps
.\config remove
```

After you've removed the agent, you can configure it again.

Unattended config

The agent can be set up from a script with no human intervention. You must pass `--unattended` and then answers to all questions.

To configure an agent, it must know the URL to your organization or collection and credentials of someone authorized to set up agents. All other responses are optional. Any command-line parameter can be specified using an environment variable instead: put its name in upper case and prepend `VSTS_AGENT_INPUT_`. For example, `VSTS_AGENT_INPUT_PASSWORD` instead of specifying `--password`.

Required options

- `--unattended` - agent setup will not prompt for information, and all settings must be provided on the command line
- `--url <url>` - URL of the server. For example: https://dev.azure.com/myorganization or http://my-azure-devops-server:8080/tfs
- `--auth <type>` - authentication type. Valid values are:
  - `pat` (Personal access token) - PAT is the only scheme that works with Azure DevOps Services.
  - `negotiate` (Kerberos or NTLM)
  - `alt` (Basic authentication)
  - `integrated` (Windows default credentials)

Authentication options

- If you chose `--auth pat`:
  - `--token <token>` - specifies your personal access token
  - PAT is the only scheme that works with Azure DevOps Services.
- If you chose `--auth negotiate` or `--auth alt`:
  - `--userName <userName>` - specifies a Windows username in the format `domain\userName` or `userName@domain.com`
  - `--password <password>` - specifies a password

Pool and agent names

- `--pool <pool>` - pool name for the agent to join
- `--agent <agent>` - agent name
- `--replace` - replace the agent in a pool. If another agent is listening by the same name, it will start failing with a conflict

Agent setup

- `--work <workDirectory>` - work directory where job data is stored. Defaults to `_work` under the root of the agent directory. The work directory is owned by a given agent and should not be shared between multiple agents.
- `--acceptTeeEula` - accept the Team Explorer Everywhere End User License Agreement (macOS and Linux only)
- `--disableloguploads` - don't stream or send console log output to the server. Instead, you may retrieve them from the agent host's filesystem after the job completes.

Windows-only startup

- `--runAsService` - configure the agent to run as a Windows service (requires administrator permission)
- `--runAsAutoLogon` - configure auto-logon and run the agent on startup (requires administrator permission)
- `--windowsLogonAccount <account>` - used with `--runAsService` or `--runAsAutoLogon` to specify the Windows user name in the format `domain\userName` or `userName@domain.com`
- `--windowsLogonPassword <password>` - used with `--runAsService` or `--runAsAutoLogon` to specify Windows logon password (not required for Group Managed Service Accounts and Windows built in accounts such as 'NT AUTHORITY\NETWORK SERVICE')
- `--overwriteAutoLogon` - used with `--runAsAutoLogon` to overwrite the existing auto logon on the machine
- `--noRestart` - used with `--runAsAutoLogon` to stop the host from restarting after agent configuration completes

Troubleshooting configuring the agent with the `runAsAutoLogon` option

Configuring the agent with the `runAsAutoLogon` option runs the agent each time after restarting the machine. Perform next steps if the agent is not run after restarting the machine.

If the agent was already configured on the machine

Before reconfiguring the agent, it is necessary to remove the old agent configuration, so try to run this command from the agent folder:

```
.\config.cmd remove --auth 'PAT' --token '<token>'
```

Check if the agent was removed from your agent pool after executing the command:

```
<Azure DevOps organization> / <Project> / Settings / Agent pools / <Agent
Pool> / Agents
```

Remove the agent from your agent pool manually if it was not removed by running the command.

Then try to reconfigure the agent by running this command from the agent folder:

```
.\config.cmd --unattended --agent '<agent-name>' --pool '<agent-pool-name>'
--url '<azure-dev-ops-organization-url>' --auth 'PAT' --token '<token>' --
runAsAutoLogon --windowsLogonAccount '<domain\user-name>' --
windowsLogonPassword '<windows-password>'
```

Specify the agent name (any specific unique name) and check if this agent appeared in your agent pool after reconfiguring.

It will be much better to unpack an agent archive (which can be downloaded here) and run this command from the new unpacked agent folder.

Check if the Windows registry key is recorded and saved correctly

Run the `whoami /user` command to get the `<sid>`. Open `Registry Editor` and follow the path:

```
Computer\HKEY_USERS\<sid>\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

Check if there is the `VSTSAgent` key. Delete this key if it exists, then close `Registry Editor` and configure the agent by running the `.\config.cmd` command (without args)

from the agent folder. Before answering the question `Enter Restart the machine at a later time?`, open `Registry Editor` again and check if the `VSTSAgent` key has appeared. Press `Enter` to answer the question, and check if the `VSTSAgent` key remains in its place after restarting the machine.

Check if Windows registry keys work fine on your machine

Create a `autorun.cmd` file that contains the following line: `echo "Hello from AutoRun!"`. Open `Registry Editor` and create in the path above a new key-value pair with the key `AutoRun` and the value

```
C:\windows\system32\cmd.exe /D /S /C start "AutoRun"
"D:\path\to\autorun.cmd"
```

Restart your machine. You have an issue with Windows registry keys if you do not see a console window with the `Hello from AutoRun!` message.

Deployment group only

- `--deploymentGroup` - configure the agent as a deployment group agent
- `--deploymentGroupName <name>` - used with `--deploymentGroup` to specify the deployment group for the agent to join
- `--projectName <name>` - used with `--deploymentGroup` to set the project name
- `--addDeploymentGroupTags` - used with `--deploymentGroup` to indicate that deployment group tags should be added
- `--deploymentGroupTags <tags>` - used with `--addDeploymentGroupTags` to specify the comma separated list of tags for the deployment group agent - for example "web, db"

Environments only

- `--addvirtualmachineresourcetags` - used to indicate that environment resource tags should be added
- `--virtualmachineresourcetags <tags>` - used with `--addvirtualmachineresourcetags` to specify the comma separated list of tags for the environment resource agent - for example "web, db"

`.\config --help` always lists the latest required and optional responses.

Diagnostics

If you're having trouble with your self-hosted agent, you can try running diagnostics. After configuring the agent:

```ps
.\run --diagnostics
```

This will run through a diagnostic suite that may help you troubleshoot the problem. The diagnostics feature is available starting with version 2.165.0.

Help on other options

To learn about other options:

```ps
.\config --help
```

The help provides information on authentication alternatives and unattended configuration.

Capabilities

Your agent's capabilities are cataloged and advertised in the pool so that only the builds and releases it can handle are assigned to it. See Build and release agent capabilities.

In many cases, after you deploy an agent, you'll need to install software or utilities. Generally you should install on your agents whatever software and tools you use on your development machine.

For example, if your build includes the npm task, then the build won't run unless there's a build agent in the pool that has npm installed.

> **Important**
>
> Capabilities include all environment variables and the values that are set when the agent runs. If any of these values change while the agent is running, the agent must be restarted to pick up the new values. After you install new software on an agent,

you must restart the agent for the new capability to show up in the pool, so that the build can run.

If you want to exclude environment variables as capabilities, you can designate them by setting an environment variable `VSO_AGENT_IGNORE` with a comma-delimited list of variables to ignore.

FAQ

How do I make sure I have the latest agent version?

1. Navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Click the pool that contains the agent.

3. Make sure the agent is enabled.

4. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

   b. Select **Agents** and choose the desired agent.

c. Choose the **Capabilities** tab.

> ⓘ **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

5. Look for the `Agent.Version` capability. You can check this value against the latest published agent version. See Azure Pipelines Agent and check the page for the

highest version number listed.

6. Each agent automatically updates itself when it runs a task that requires a newer version of the agent. If you want to manually update some agents, right-click the pool, and select **Update all agents**.

How do I make sure I have the latest v2 agent version?

1. Navigate to the **Agent pools** tab:

   a. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

   b. Choose **Azure DevOps**, **Organization settings**.

   c. Choose **Agent pools**.

2. Click the pool that contains the agent.

3. Make sure the agent is enabled.

4. Navigate to the capabilities tab:

   a. From the **Agent pools** tab, select the desired agent pool.

   b. Select **Agents** and choose the desired agent.

   c. Choose the **Capabilities** tab.

> **Note**
>
> Microsoft-hosted agents don't display system capabilities. For a list of software installed on Microsoft-hosted agents, see **Use a Microsoft-hosted agent**.

5. Look for the `Agent.Version` capability. You can check this value against the latest published agent version. See Azure Pipelines Agent and check the page for the highest version number listed.

6. Each agent automatically updates itself when it runs a task that requires a newer version of the agent. If you want to manually update some agents, right-click the pool, and select **Update all agents**.

I'm running a firewall and my code is in Azure Repos. What URLs does the agent need to communicate with?

If you're running an agent in a secure network behind a firewall, make sure the agent can initiate communication with the following URLs and IP addresses.

Domain URL	Description
`https://{organization_name}.pkgs.visualstudio.com`	Azure DevOps Packaging API for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vsrm.visualstudio.com`	Release Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://{organization_name}.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the `{organization_name}.visualstudio.com` domain
`https://*.blob.core.windows.net`	Azure Artifacts
`https://*.dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://*.vsassets.io`	Azure Artifacts via CDN
`https://*.vsblob.visualstudio.com`	Azure DevOps Telemetry for organizations using the `dev.azure.com` domain
`https://*.vssps.visualstudio.com`	Azure DevOps Platform Services for organizations using the `dev.azure.com` domain
`https://*.vstmr.visualstudio.com`	Azure DevOps Test Management Services for organizations using the `dev.azure.com` domain

Domain URL	Description
`https://app.vssps.visualstudio.com`	For organizations using the `{organization_name}.visualstudio.com` domain
`https://dev.azure.com`	For organizations using the `dev.azure.com` domain
`https://login.microsoftonline.com`	Azure Active Directory sign in
`https://management.core.windows.net`	Azure Management API's
`https://vstsagentpackage.azureedge.net`	Agent package

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> For more information about allowed addresses, see **Allowed address lists and network connections**.

How do I run the agent with self-signed certificate?

> **Note**

Running the agent with a self-signed certificate only applies to Azure DevOps Server.

Run the agent with self-signed certificate

How do I run the agent behind a web proxy?

Run the agent behind a web proxy

How do I restart the agent

If you are running the agent interactively, see the restart instructions in Run interactively. If you are running the agent as a service, restart the agent by following the steps in Run as a service.

How do I set different environment variables for each individual agent?

Create a `.env` file under agent's root directory and put environment variables you want to set into the file as following format:

```
MyEnv0=MyEnvValue0
MyEnv1=MyEnvValue1
MyEnv2=MyEnvValue2
MyEnv3=MyEnvValue3
MyEnv4=MyEnvValue4
```

How do I configure the agent to bypass a web proxy and connect to Azure Pipelines?

If you want the agent to bypass your proxy and connect to Azure Pipelines directly, then you should configure your web proxy to enable the agent to access the following URLs.

For organizations using the `*.visualstudio.com` domain:

```
https://login.microsoftonline.com
https://app.vssps.visualstudio.com
```

```
https://{organization_name}.visualstudio.com
https://{organization_name}.vsrm.visualstudio.com
https://{organization_name}.vstmr.visualstudio.com
https://{organization_name}.pkgs.visualstudio.com
https://{organization_name}.vssps.visualstudio.com
```

For organizations using the `dev.azure.com` domain:

```
https://dev.azure.com
https://*.dev.azure.com
https://login.microsoftonline.com
https://management.core.windows.net
https://vstsagentpackage.azureedge.net
https://vssps.dev.azure.com
```

To ensure your organization works with any existing firewall or IP restrictions, ensure that `dev.azure.com` and `*dev.azure.com` are open and update your allow-listed IPs to include the following IP addresses, based on your IP version. If you're currently allow-listing the `13.107.6.183` and `13.107.9.183` IP addresses, leave them in place, as you don't need to remove them.

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`

> **Note**
>
> This procedure enables the agent to bypass a web proxy. Your build pipeline and scripts must still handle bypassing your web proxy for each task and tool you run in your build.

For example, if you are using a NuGet task, you must configure your web proxy to support bypassing the URL for the server that hosts the NuGet feed you're using.

I'm using TFS and the URLs in the sections above don't work for me. Where can I get help?

Web site settings and security

What is enable SERVICE_SID_TYPE_UNRESTRICTED for agent service?

When configuring the agent software on Windows Server, you can specify the service security identifier from the following prompt.

```
Enter enable SERVICE_SID_TYPE_UNRESTRICTED for agent service (Y/N) (press
enter for N)
```

Previous versions of the agent software set the service security identifier type to `SERVICE_SID_TYPE_NONE`, which is the default value for the current agent versions. To configure the security service identifier type to `SERVICE_SID_TYPE_UNRESTRICTED`, press `Y`.

For more information, see SERVICE_SID_INFO structure and Security identifiers.

Azure Virtual Machine Scale Set agents

Article • 05/10/2023

Azure DevOps Services

Azure Virtual Machine Scale Set agents, hereafter referred to as scale set agents, are a form of self-hosted agents that can be autoscaled to meet your demands. This elasticity reduces your need to run dedicated agents all the time. Unlike Microsoft-hosted agents, you have flexibility over the size and the image of machines on which agents run.

If you like Microsoft-hosted agents but are limited by what they offer, you should consider scale set agents. Here are some examples:

- You need more memory, more processor, more storage, or more IO than what we offer in native Microsoft-hosted agents.
- You need NCv2 VM with particular instruction sets for machine learning.
- You need to deploy to a private Azure App Service in a private VNET with no inbound connectivity.
- You need to open corporate firewall to specific IP addresses so that Microsoft-hosted agents can communicate with your servers.
- You need to restrict network connectivity of agent machines and allow them to reach only approved sites.
- You can't get enough agents from Microsoft to meet your needs.
- Your jobs exceed the Microsoft-hosted agent timeout.
- You can't partition Microsoft-hosted parallel jobs to individual projects or teams in your organization.
- You want to run several consecutive jobs on an agent to take advantage of incremental source and machine-level package caches.
- You want to run configuration or cache warmup before an agent begins accepting jobs.

If you like self-hosted agents but wish that you could simplify managing them, you should consider scale set agents. Here are some examples:

- You don't want to run dedicated agents around the clock. You want to de-provision agent machines that aren't being used to run jobs.
- You run untrusted code in your pipeline and want to reimage agent machines after each job.
- You want to simplify periodically updating the base image for your agents.

Note

- You cannot run Mac agents using scale sets. You can only run Windows or Linux agents this way.
- Using VMSS agent pools for Azure DevOps Services is only supported for Azure Public (global service) cloud. Currently, VMSS agent pools does not support any other **national cloud offerings**.
- You should not associate a VMSS to multiple pools.

Create the scale set

In preparation for creating scale set agents, you must first create a Virtual Machine Scale Set in the Azure portal. You must create the Virtual Machine Scale Set in a certain way so that Azure Pipelines can manage it. In particular, you must **disable** autoscaling so that Azure Pipelines can determine how to perform scaling based on number of incoming pipeline jobs. We recommend that you use the following steps to create the scale set.

In the following example, a new resource group and Virtual Machine Scale Set are created with Azure Cloud Shell using the UbuntuLTS VM image.

> **Note**
>
> In this example, the UbuntuLTS VM image is used for the scale set. If you require a customized VM image as the basis for your agent, create the customized image before creating the scale set, by following the steps in **Create a scale set with custom image, software, or disk size**.

1. Browse to Azure Cloud Shell at `https://shell.azure.com/`.
2. Run the following command to verify your default Azure subscription.

   Azure CLI

   ```
   az account list -o table
   ```

   If your desired subscription isn't listed as the default, select your desired subscription.

   Azure CLI

   ```
   az account set -s <your subscription ID>
   ```

3. Create a resource group for your Virtual Machine Scale Set.

Azure CLI

```
az group create \
--location westus \
--name vmssagents
```

4. Create a Virtual Machine Scale Set in your resource group. In this example, the UbuntuLTS VM image is specified.

Azure CLI

```
az vmss create \
--name vmssagentspool \
--resource-group vmssagents \
--image UbuntuLTS \
--vm-sku Standard_D2_v3 \
--storage-sku StandardSSD_LRS \
--authentication-type SSH \
--generate-ssh-keys \
--instance-count 2 \
--disable-overprovision \
--upgrade-policy-mode manual \
--single-placement-group false \
--platform-fault-domain-count 1 \
--load-balancer ""
```

> **Note**
>
> Azure Pipelines does not support scale set **overprovisioning** and **autoscaling**. Make sure both features are disabled for your scale set.

Because Azure Pipelines manages the scale set, the following settings are required or recommended:

- `--disable-overprovision` - required
- `--upgrade-policy-mode manual` - required
- `--load-balancer ""` - Azure Pipelines doesn't require a load balancer to route jobs to the agents in the scale set agent pool, but configuring a load balancer is one way to get an IP address for your scale set agents that you could use for firewall rules. Another option for getting an IP address for your scale set agents is to create your scale set using the `--public-ip-address` options. For more information about configuring your scale set with a load balancer or

public IP address, see the Virtual Machine Scale Sets documentation and az vmss create.

- `--instance-count 2` - this setting isn't required, but it gives you an opportunity to verify that the scale set is fully functional before you create an agent pool. Creation of the two VMs can take several minutes. Later, when you create the agent pool, Azure Pipelines deletes these two VMs and create new ones.

> **Important**
>
> If you run this script using Azure CLI on Windows, you must enclose the `""` in `--load-balancer ""` with single quotes like this: `--load-balancer '""'`

If your VM size supports Ephemeral OS disks, the following parameters to enable Ephemeral OS disks are optional but recommended to improve virtual machine reimage times.

- `--ephemeral-os-disk true`
- `--os-disk-caching readonly`

> **Important**
>
> Ephemeral OS disks are not supported on all VM sizes. For list of supported VM sizes, see **Ephemeral OS disks for Azure VMs**.

Select any Linux or Windows image - either from Azure Marketplace or your own custom image - to create the scale set. Don't pre-install Azure Pipelines agent in the image. Azure Pipelines automatically installs the agent as it provisions new virtual machines. In the above example, we used a plain `UbuntuLTS` image. For instructions on creating and using a custom image, see FAQ.

Select any VM SKU and storage SKU.

> **Note**
>
> Licensing considerations limit us from distributing Microsoft-hosted images. We are unable to provide these images for you to use in your scale set agents. But, the **scripts** that we use to generate these images are open source. You are free to use these scripts and create your own custom images.

5. After creating your scale set, navigate to your scale set in the Azure portal and verify the following settings:

   - **Upgrade policy - Manual**

   You can also verify this setting by running the following Azure CLI command.

   Azure CLI

```
az vmss show --resource-group vmssagents --name vmssagentspool --
output table

Name            ResourceGroup    Location    Zones    Capacity
Overprovision    UpgradePolicy
--------------  ---------------  ----------  -------  ----------
---------------  ---------------
vmssagentspool  vmssagents       westus               0
False            Manual
```

   - **Scaling - Manual scale**

> **Important**
>
> Azure Pipelines does not support **instance protection**. Make sure you have the *scale-in* and *scale set actions* instance protections disabled.

Create the scale set agent pool

1. Navigate to your Azure DevOps **Project settings**, select **Agent pools** under **Pipelines**, and select **Add pool** to create a new agent pool.

> **ⓘ Important**
>
> You may create your scale set pool in **Project settings** or **Organization settings**, but when you delete a scale set pool, you must delete it from **Organization settings**, and not **Project settings**.

2. Select **Azure Virtual Machine Scale Set** for the pool type. Select the **Azure subscription** that contains the scale set, choose **Authorize**, and choose the desired Virtual Machine Scale Set from that subscription. If you have an existing service connection, you can choose that from the list instead of the subscription.

> **ⓘ Important**

- To configure a scale set agent pool, you must have either **Owner** or **User Access Administrator** permissions on the selected subscription. If you have one of these permissions but get an error when you choose **Authorize**, see **troubleshooting**.

- The only service connection currently supported is an Azure Resource Manager (ARM) service connection based on a service principal key. ARM service connections based on a certificate credential or a Managed Identity will fail. When you attempt to list the existing scale sets in your subscription, you'll see an error like this:

  ```
  Invalid Service Endpoint with Id <guid> and Scope <guid>
  ```

3. Choose the desired Virtual Machine Scale Set from that subscription.

4. Specify a name for your agent pool.

5. Configure the following options:

   - **Automatically tear down virtual machines after every use** - A new VM instance is used for every job. The VM goes offline after running a job and is reimaged before picking up another job.
   - **Save an unhealthy agent for investigation** - Whether to save unhealthy agent VMs for troubleshooting instead of deleting them.
   - **Maximum number of virtual machines in the scale set** - Azure Pipelines will automatically scale out the number of agents, but won't exceed this limit.
   - **Number of agents to keep on standby** - Azure Pipelines will automatically scale in the number of agents, but will ensure that there are always this many agents available to run new jobs. If you set **Number of agents to keep on standby** to **0**, for example to conserve cost for a low volume of jobs, Azure Pipelines will start a VM only when it has a job.
   - **Delay in minutes before deleting excess idle agents** - To account for the variability in build load throughout the day, Azure Pipelines will wait for the specified duration before deleting an excess idle agent.
   - **Configure VMs to run interactive tests** (Windows Server OS Only) - Windows agents can either be configured to run unelevated with autologon and with interactive UI, or they can be configured to run with elevated permissions. Check this box to run unelevated with interactive UI. In either case, the agent user is a member of the Administrators group.

6. When your settings are configured, choose **Create** to create the agent pool.

Use scale set agent pool

Using a scale set agent pool is similar to any other agent pool. You can use it in classic build, release, or YAML pipelines. User permissions, pipeline permissions, approvals, and other checks work the same way as in any other agent pool. For more information, see Agent pools.

> **Important**
>
> Caution must be exercised when making changes directly to the scale set in the Azure portal.
>
> - You may not change many of the the scale set configuration settings in the Azure portal. Azure Pipelines updates the configuration of the scale set. Any manual changes you make to the scale set may interfere with the operation of Azure Pipelines.
> - You may not rename or delete a scale set without first deleting the scale set pool in Azure Pipelines.

How Azure Pipelines manages the scale set

Once the scale set agent pool is created, Azure Pipelines automatically scales the agent machines.

Azure Pipelines samples the state of the agents in the pool and virtual machines in the scale set every 5 minutes. The decision to scale in or out is based on the number of idle agents at that time. An agent is considered idle if it's online and isn't running a pipeline job. Azure Pipelines performs a scale-out operation if either of the following conditions is satisfied:

- The number of idle agents falls below the number of standby agents you specify
- There are no idle agents to service pipeline jobs waiting in the queue

If one of these conditions is met, Azure Pipelines grows the number of VMs. Scaling out is done in increments of a certain percentage of the maximum pool size. Allow 20 minutes for machines to be created for each step.

Azure Pipelines scales in the agents when the number of idle agents exceeds the standby count for more than 30 minutes (configurable using **Delay in minutes before deleting excess idle agents**).

To put all of this into an example, consider a scale set agent pool that is configured with two standby agents and four maximum agents. Let us say that you want to tear down the VM after each use. Also, let us assume that there are no VMs to start with in the scale set.

- Since the number of idle agents is 0, and since the number of idle agents is below the standby count of 2, Azure Pipelines scales out and adds two VMs to the scale set. Once these agents come online, there will be two idle agents.
- Let us say that one pipeline job arrives and is allocated to one of the agents.
- At this time, the number of idle agents is 1, and that is less than the standby count of 2. So, Azure Pipelines scales out and adds 2 more VMs (the increment size used in this example). At this time, the pool has three idle agents and one busy agent.
- Let us say that the job on the first agent completes. Azure Pipelines takes that agent offline to reimage that machine. After a few minutes, it comes back with a fresh image. At this time, we'll have four idle agents.
- If no other jobs arrive for 30 minutes (configurable using **Delay in minutes before deleting excess idle agents**), Azure Pipelines determines that there are more idle agents than are necessary. So, it scales in the pool to two agents.

Throughout this operation, the goal for Azure Pipelines is to reach the desired number of idle agents on standby. Pools scale out and in slowly. Over the course of a day, the pool will scale out as requests are queued in the morning and scale in as the load subsides in the evening. You may observe more idle agents than you desire at various times, which is expected as Azure Pipelines converges gradually to the constraints that you specify.

> **Note**
>
> It can take an hour or more for Azure Pipelines to scale out or scale in the virtual machines. Azure Pipelines will scale out in steps, monitor the operations for errors, and react by deleting unusable machines and by creating new ones in the course of time. This corrective operation can take over an hour.

To achieve maximum stability, scale set operations are done sequentially. For example if the pool needs to scale out and there are also unhealthy machines to delete, Azure Pipelines will first scale out the pool. Once the pool has scaled out to reach the desired number of idle agents on standby, the unhealthy machines will be deleted, depending on the **Save an unhealthy agent for investigation** setting. For more information, see Unhealthy agents.

Due to the sampling size of 5 minutes, it's possible that all agents can be running pipelines for a short period of time and no scaling out will occur.

Customizing Pipeline Agent Configuration

You can customize the configuration of the Azure Pipelines Agent by defining environment variables in your operating system custom image for your scale set. For example, the scale set agent working directory defaults to C:\a for Windows and /agent/_work for Linux. If you want to change the working directory, set an environment variable named VSTS_AGENT_INPUT_WORK with the desired working directory. More information can be found in the Pipelines Agent Unattended Configuration documentation. Some examples include:

- `VSTS_AGENT_INPUT_WORK`
- `VSTS_AGENT_INPUT_PROXYURL`
- `VSTS_AGENT_INPUT_PROXYUSERNAME`
- `VSTS_AGENT_INPUT_PROXYPASSWORD`

> **Important**
>
> Caution must be exercised when customizing the Pipelines agent. Some settings will conflict with other required settings, causing the agent to fail to register, and the VM to be deleted. These settings should not be set or altered:
>
> - `VSTS_AGENT_INPUT_URL`
> - `VSTS_AGENT_INPUT_AUTH`
> - `VSTS_AGENT_INPUT_TOKEN`
> - `VSTS_AGENT_INPUT_USERNAME`
> - `VSTS_AGENT_INPUT_PASSWORD`
> - `VSTS_AGENT_INPUT_POOL`
> - `VSTS_AGENT_INPUT_AGENT`
> - `VSTS_AGENT_INPUT_RUNASSERVICE`
> - ... and anything related to Deployment Groups.

Customizing Virtual Machine Startup via the Custom Script Extension

Users may want to execute startup scripts on their scaleset agent machines before those machines start running pipeline jobs. Some common use cases for startup scripts include installing software, warming caches, or fetching repos. You can execute startup scripts by installing the Custom Script Extension for Windows or Custom Script Extension for Linux.

This extension will be executed on every virtual machine in the scaleset immediately after it's created or reimaged. The custom script extension will be executed before the Azure Pipelines agent extension is executed.

Here's an example to create a custom script extension for Linux.

Azure CLI

```
az vmss extension set \
--vmss-name <scaleset name> \
--resource-group <resource group> \
--name CustomScript \
--version 2.0 \
--publisher Microsoft.Azure.Extensions \
--settings '{ \"fileUris\":[\"https://<myGitHubRepoUrl>/myScript.sh\"],
\"commandToExecute\": \"bash ./myScript.sh /myArgs \" }'
```

Here's an example to create a custom script extension for Windows.

Azure CLI

```
az vmss extension set \
--vmss-name <scaleset name> \
--resource-group <resource group> \
--name CustomScriptExtension \
--version 1.9 \
--publisher Microsoft.Compute \
--settings '{ \"FileUris\":[\"https://<myGitHubRepoUrl>/myscript.ps1\"],
\"commandToExecute\": \"Powershell.exe -ExecutionPolicy Unrestricted -File
myscript.ps1 -myargs 0 \" }'
```

Important

The scripts executed in the Custom Script Extension must return with exit code 0 in order for the VM to finish the VM creation process. If the custom script extension throws an exception or returns a non-zero exit code, the Azure Pipeline extension will not be executed and the VM will not register with Azure DevOps agent pool.

It might happen that your extension runs before all VM resources are provisioned, in which case you'll see an error similar to "failed installing basic prerequisites." You can fix this by adding a `sleep` command at the beginning of your script, for example, `sleep 30`.

Lifecycle of a Scale Set Agent

Here's the flow of operations for an Azure Pipelines Virtual Machine Scale Set Agent

1. The Azure DevOps Scale Set Agent Pool sizing job determines the pool has too few idle agents and needs to scale out. Azure Pipelines makes a call to Azure Scale Sets to increase the scale set capacity.

2. The Azure Scale Set begins creating the new virtual machines. Once the virtual machines are running, Azure Scale Sets sequentially executes any installed VM extensions.

3. If the Custom Script Extension is installed, it's executed before the Azure Pipelines Agent extension. If the Custom Script Extension returns a non-zero exit code, the VM creation process is aborted and will be deleted.

4. The Azure Pipelines Agent extension is executed. This extension downloads the latest version of the Azure Pipelines Agent along with the latest version of configuration script. The configuration scripts can be found at URLs with the following formats:

   - Linux: `https://vstsagenttools.blob.core.windows.net/tools/ElasticPools/Linux/<script_version>/enableagent.sh`, for example, version 14
   - Windows: `https://vstsagenttools.blob.core.windows.net/tools/ElasticPools/Windows/<script_version>/enableagent.ps1`, for example, version 16

5. The configuration script creates a local user named `AzDevOps` if the operating system is Windows Server or Linux. For Windows 10 Client OS, the agent runs as LocalSystem. The script then unzips, installs, and configures the Azure Pipelines Agent. As part of configuration, the agent registers with the Azure DevOps agent pool and appears in the agent pool list in the Offline state.

6. For most scenarios, the configuration script then immediately starts the agent to run as the local user `AzDevOps`. The agent goes Online and is ready to run pipeline jobs.

If the pool is configured for interactive UI, the virtual machine reboots after the agent is configured. After reboot, the local user automatically logs in and pipelines agent starts. The agent then goes online and is ready to run pipeline jobs.

Create a scale set with custom image, software, or disk size

If you just want to create a scale set with the default 128-GB OS disk using a publicly available Azure image, then skip straight to step 10 and use the public image name (UbuntuLTS, Win2019DataCenter, etc.) to create the scale set. Otherwise follow these steps to customize your VM image.

1. Create a VM with your desired OS image and optionally expand the OS disk size from 128 GB to `<myDiskSizeGb>`.

   - If starting with an available Azure Image, for example <myBaseImage> = (Win2019DataCenter, UbuntuLTS):

     Azure CLI

     ```
     az vm create --resource-group <myResourceGroup> --name <MyVM> --image <myBaseImage> --os-disk-size-gb <myDiskSize>  --admin-username myUserName --admin-password myPassword
     ```

   - If starting with a generalized VHD:

     a. First create the VM with an unmanaged disk of the desired size and then convert to a managed disk:

     Azure CLI

     ```
     az vm create --resource-group <myResourceGroup> --name <MyVM> --image <myVhdUrl> --os-type windows --os-disk-size-gb <myDiskSizeGb> --use-unmanaged-disk --admin-username <myUserName> --admin-password <myPassword> --storage-account <myVhdStorageAccount>
     ```

     b. Shut down the VM

     Azure CLI

     ```
     az vm stop --resource-group <myResourceGroup> --name <MyVM>
     ```

c. Deallocate the VM

Azure CLI

```
az vm deallocate --resource-group <myResourceGroup> --name
<MyVM>
```

d. Convert to a managed disk

Azure CLI

```
az vm convert --resource-group <myResourceGroup> --name <MyVM>
```

e. Restart the VM

Azure CLI

```
az vm start --resource-group <myResourceGroup> --name <MyVM>
```

2. Remote Desktop (or SSH) to the VM's public IP address to customize the image. You may need to open ports in the firewall to unblock the RDP (3389) or SSH (22) ports.

   a. **Windows** - If `<MyDiskSizeGb>` is greater than 128 GB, extend the OS disk size to fill the disk size you specified by `<MyDiskSizeGb>`.

   Open DiskPart tool as administrator and run these DiskPart commands:
   i. `list volume` (to see the volumes)
   ii. `select volume 2` (depends on which volume is the OS drive)
   iii. `extend size 72000` (to extend the drive by 72 GB, from 128 GB to 200 GB)

3. Install any desired additional software on the VM.

4. To customize the permissions of the pipeline agent user, you can create a user named `AzDevOps`, and grant that user the permissions you require. This user will be created by the scaleset agent startup script if it doesn't already exist.

5. Reboot the VM when finished with customizations

6. Generalize the VM.

   - **Windows** - From an admin console window:

Console

```
C:\Windows\System32\sysprep\sysprep.exe /generalize /oobe
/shutdown
```

- **Linux**:

```bash
sudo waagent -deprovision+user -force
```

> **Important**
>
> Wait for the VM to finish generalization and shutdown. Do not proceed until the VM has stopped. Allow 60 minutes.

7. Deallocate the VM

```azure cli
az vm deallocate --resource-group <myResourceGroup> --name <MyVM>
```

8. Mark the VM as Generalized

```azure cli
az vm generalize --resource-group <myResourceGroup> --name <MyVM>
```

9. Create a VM Image based on the generalized image. When performing these steps to update an existing scaleset image, make note of the image ID url in the output.

```azure cli
az image create  --resource-group <myResourceGroup> --name <MyImage> --
source <MyVM>
```

10. Create the scale set based on the custom VM image

```azure cli
az vmss create --resource-group <myResourceGroup> --name <myScaleSet> -
-image <MyImage> --admin-username <myUsername> --admin-password
<myPassword> --instance-count 2 --disable-overprovision --upgrade-
policy-mode manual --load-balancer '""'
```

11. Verify that both VMs created in the scale set come online, have different names, and reach the Succeeded state

You're now ready to create an agent pool using this scale set.

Update an existing scale set with a new custom image

To update the image on an existing scaleset, follow the steps in the previous Create a scale set with custom image, software, or disk size section up through the `az image create` step to generate the custom OS image. Make note of the ID property URL that is output from the `az image create` command. Then update the scaleset with the new image as shown in the following example. After the scaleset image has been updated, all future VMs in the scaleset will be created with the new image.

Azure CLI

```
az vmss update --resource-group <myResourceGroup> --name <myScaleSet> --set
virtualMachineProfile.storageProfile.imageReference.id=<id url>
```

Supported Operating Systems

Scale set agents currently support Ubuntu Linux, Windows Server/DataCenter 2016/2019, and Windows 10 client.

Known issues

- Debian or RedHat Linux distributions aren't supported. Only Ubuntu is.
- Windows 10 client doesn't support running the pipeline agent as a local user and therefore the agent can't interact with the UI. The agent will run as Local Service instead.

Troubleshooting issues

Navigate to your Azure DevOps **Project settings**, select **Agent pools** under **Pipelines**, and select your agent pool. Select the tab labeled **Diagnostics**.

The Diagnostic tab shows all actions executed by Azure DevOps to Create, Delete, or Reimage VMs in your Azure Scale Set. Diagnostics also logs any errors encountered while trying to perform these actions. Review the errors to make sure your scaleset has

sufficient resources to scale out. If your Azure subscription has reached the resource limit in VMs, CPU cores, disks, or IP Addresses, those errors will show up here.

Unhealthy Agents

When agents or virtual machines are failing to start, not connecting to Azure DevOps, or going offline unexpectedly, Azure DevOps logs the failures to the Agent Pool's **Diagnostics** tab and tries to delete the associated virtual machine. Networking configuration, image customization, and pending reboots may cause these issues. Connecting to the VM to debug and gather logs can help with the investigation.

If you would like Azure DevOps to save an unhealthy agent VM for investigation and not automatically delete it when it detects the unhealthy state, navigate to your Azure DevOps **Project settings**, select **Agent pools** under **Pipelines**, and select your agent pool. Choose **Settings**, select the option **Save an unhealthy agent for investigation**, and choose **Save**.

Now, when an unhealthy agent is detected in the scale set, Azure DevOps saves that agent and associated virtual machine. The saved agent will be visible on the **Diagnostics** tab of the Agent pool UI. Navigate to your Azure DevOps **Project settings**, select **Agent pools** under **Pipelines**, select your agent pool, choose **Diagnostics**, and make note of the agent name.

Find the associated virtual machine in your Azure Virtual Machine Scale Set via the Azure portal, in the **Instances** list.

Select the instance, choose **Connect**, and perform your investigation.

To delete the saved agent when you're done with your investigation, navigate to your Azure DevOps **Project settings**, select **Agent pools** under **Pipelines**, and select your agent pool. Choose the tab labeled **Diagnostics**. Find the agent on the **Agents saved for**

investigation card, and choose **Delete**. This removes the agent from the pool and deletes the associated virtual machine.

FAQ

- Where can I find the images used for Microsoft-hosted agents?
- How do I configure scale set agents to run UI tests?
- How can I delete agents?
- Can I configure the scale set agent pool to have zero agents on standby?
- How much do scale set agents cost?
- What are some common issues and their solutions?
  - You observe more idle agents than desired at various times
  - VMSS scale up isn't happening in the expected five-minute interval
  - Azure DevOps Linux VM Scale Set frequently fails to start the pipeline
  - You check the option to automatically tear down virtual machines after every use for the agent pool, but you see that the VMs aren't re-imaging as they should and just pick up new jobs as they're queued
  - VMSS shows the agent as offline if the VM restarts
  - You can see multiple tags like _AzureDevOpsElasticPoolTimeStamp for VMSS in cost management
  - You can't create a new scale set agent pool and get an error message that a pool with the same name already exists
  - VMSS maintenance job isn't running on agents or getting logs
  - If you specify AzDevOps as the primary administrator in your script for VMSS, you may observe issues with the agent configurations on scale set instances
  - Agent extension installation fails on scale set instances due to network security and firewall configurations
  - I want to increase my pool size. What should I take into consideration?

Where can I find the images used for Microsoft-hosted agents?

Licensing considerations limit us from distributing Microsoft-hosted images. We're unable to provide these images for you to use in your scale set agents. But, the scripts that we use to generate these images are open source. You're free to use these scripts and create your own custom images.

How do I configure scale set agents to run UI tests?

Create a Scale Set with a Windows Server OS and when creating the Agent Pool select the "Configure VMs to run interactive tests" option.

How can I delete agents?

` Navigate to your Azure DevOps **Project settings**, select **Agent pools** under **Pipelines**, and select your agent pool. Select the tab labeled **Agents**. Click the 'Enabled' toggle button to disable the agent. The disabled agent will complete the pipeline it's currently running and won't pick up additional work. Within a few minutes after completing its current pipeline job, the agent will be deleted.

Can I configure the scale set agent pool to have zero agents on standby?

Yes, if you set **Number of agents to keep on standby** to zero, for example to conserve cost for a low volume of jobs, Azure Pipelines starts a VM only when it has a job.

How much do scale set agents cost?

Pricing for scale set agents is similar to other self-hosted agents. You provide the infrastructure on which to run the agent software and the jobs, and you pay for the desired number of jobs that can run concurrently by purchasing parallel jobs.

For scale set agents, the infrastructure to run the agent software and jobs is Azure Virtual Machine Scale Sets, and the pricing is described in Virtual Machine Scale Sets pricing.

For information on purchasing parallel jobs, see Configure and pay for parallel jobs.

What are some common issues and their solutions?

You observe more idle agents than desired at various times

To better understand why this happens, see How Azure Pipelines manages the scale set. Throughout the scaling operation, the goal for Azure Pipelines is to reach the desired number of idle agents on standby. Pools scale out and in slowly. Over a day, the pool will scale out as requests are queued in the morning and scale in as the load subsides in the evening. This is an expected behavior as Azure Pipelines converges gradually to the constraints that you specify.

VMSS scale up isn't happening in the expected five-minute interval

The scaling job runs every five minutes, but if only one operation is processed, you may observe that scale up isn’t happening within five minutes; this is currently by design.

Azure DevOps Linux VM Scale Set frequently fails to start the pipeline

The first place to look when experiencing issues with scale set agents is the Diagnostics tab in the agent pool.

Also, consider saving the unhealthy VM for debugging purposes. For more information, see Unhealthy Agents.

Saved agents are there unless you delete them. If the agent doesn't come online in 10 minutes, it's marked as unhealthy and saved if possible. Only one VM is kept in a saved state. If the agent goes offline unexpectedly (due to a VM reboot or something happening to the image), it isn't saved for investigation.

Only VMs for which agents fail to start are saved. If a VM has a failed state during creation, it isn't saved. In this case, the message in the Diagnostics tab is "deleting unhealthy machine" instead of "failed to start".

You check the option to automatically tear down virtual machines after every use for the agent pool, but you see that the VMs aren't re-imaging as they should and just pick up new jobs as they're queued

The option to tear down the VM after each build will only work for Windows Server and supported Linux images. It isn’t supported for Windows client images.

VMSS shows the agent as offline if the VM restarts

Showing the agents as offline if the VM restarts is the expected behavior. The agent service runs in the systemd context only. However, if the machine restarts for some reason, it's considered an unhealthy VM and deleted. For more information, see Unhealthy Agents.

When agents or virtual machines fail to start, can't connect to Azure DevOps, or go offline unexpectedly, Azure DevOps logs the failures to the Agent Pool's **Diagnostics** tab and tries to delete the associated virtual machine. Networking configuration, image customization, and pending reboots may cause these issues. To avoid the issue, disable the software update on the image. You can also connect to the VM to debug and gather logs to help investigate the issue.

You can see multiple tags like _AzureDevOpsElasticPoolTimeStamp for VMSS in cost management

When the pool is created, a tag is added to the scale set to mark the scale set as in use (to avoid two pools using the same scale set), and another tag is added for the timestamp that updates each time the configuration job runs (every two hours).

You can't create a new scale set agent pool and get an error message that a pool with the same name already exists

You may get an error message like `This virtual machine scale set is already in use by pool <pool name>` because the tag still exists on the scale set even after it's deleted. When an agent pool is deleted, you attempt to delete the tag from the scale set, but this is a best-effort attempt, and you give up after three retries. Also, there can be a maximum of a two-hour gap, in which a Virtual Machine Scale Set that isn't used by any agent pool can't be assigned to a new one. The fix for this is to wait for that time interval to pass, or manually delete the tag for the scale set from the Azure portal. When viewing the scale set in the Azure portal, select the **Tags** link on the left and delete the tag labeled **_AzureDevOpsElasticPool**.

VMSS maintenance job isn't running on agents or getting logs

The maintenance job runs once every 24 hours. It's possible that VMs are getting filled up before this time. Consider increasing the disk size on the VM and adding a script in the pipeline to delete the contents.

If you specify `AzDevOps` as the primary administrator in your script for VMSS, you may observe issues with the agent configurations on

scale set instances

If you specify `AzDevOps` as the primary administrator in your script for Virtual Machine Scale Set, you may observe issues with the agent configurations on scale set instances (the password for the user is changed if it already exists).

This issue occurs because agent extension scripts attempt to create the user `AzDevOps` and change its password.

> **Note**
>
> It's OK to create the user and grant it extra permissions, but it should not be the primary administrator, and nothing should depend on the password, as the password will be changed. To avoid the issue, pick a different user as the primary administrator when creating the scale set, instead of `AzDevOps`.

Agent extension installation fails on scale set instances due to network security and firewall configurations

The extension needs to be able to download the build agent files from `https://vstsagentpackage.azureedge.net/agent`, and the build agent needs to be able to register with Azure DevOps Services. Make sure that this URL and Azure DevOps Services-related IPs and URLs are open on the instance. For IPs and URLs that need to be unblocked on your firewall, see Allowed IP addresses and domain URLs.

I want to increase my pool size. What should I take into consideration?

Before you increase the size of your pool, make sure the Azure Virtual Network configured for your Virtual Machine Scale Sets pool has a large enough *Address space* range to accommodate all your new agents. If not, you may get an error similar to *Failed to increase capacity. Subnet azure-devops-agent-pool-fabrikam-fiber with address prefix 12.123.45.224/28 does not have enough capacity for 5 IP addresses.*

Run a self-hosted agent behind a web proxy

Article • 01/26/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

When your self-hosted agent requires a web proxy, you can inform the agent about the proxy during configuration. This allows your agent to connect to Azure Pipelines or TFS through the proxy. This in turn allows the agent to get sources and download artifacts. Finally, it passes the proxy details through to tasks which also need proxy settings in order to reach the web.

Azure Pipelines, TFS 2018 RTM and newer

(Applies to agent version 2.122 and newer.)

To enable the agent to run behind a web proxy, pass `--proxyurl`, `--proxyusername` and `--proxypassword` during agent configuration.

For example:

Windows

```
./config.cmd --proxyurl http://127.0.0.1:8888 --proxyusername "myuser" -
-proxypassword "mypass"
```

We store your proxy credential responsibly on each platform to prevent accidental leakage. On Linux, the credential is encrypted with a symmetric key based on the machine ID. On macOS, we use the Keychain. On Windows, we use the Credential Store.

> **Note**
>
> Agent version 122.0, which shipped with TFS 2018 RTM, has a known issue configuring as a service on Windows. Because the Windows Credential Store is per user, you must configure the agent using the same user the service is going to run as. For example, in order to configure the agent service run as

`mydomain\buildadmin`, you must launch `config.cmd` as `mydomain\buildadmin`. You can do that by logging into the machine with that user or using `Run as a different user` in the Windows shell.

How the agent handles the proxy within a build or release job

The agent will talk to Azure DevOps/TFS service through the web proxy specified in the `.proxy` file.

Since the code for the `Get Source` task in builds and `Download Artifact` task in releases are also baked into the agent, those tasks will follow the agent proxy configuration from the `.proxy` file.

The agent exposes proxy configuration via environment variables for every task execution. Task authors need to use azure-pipelines-task-lib methods to retrieve proxy configuration and handle the proxy within their task.

Note that many tools do not automatically use the agent configured proxy settings. For example, tools such as `curl` and `dotnet` may require proxy environment variables such as `http_proxy` to also be set on the machine.

Specify proxy bypass URLs

Create a `.proxybypass` file in the agent's root directory that specifies regular expressions (in ECMAScript syntax) to match URLs that should bypass the proxy. For example:

```
github\.com
bitbucket\.com
```

Run a self-hosted agent in Docker

Article • 10/05/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

This article provides instructions for running your Azure Pipelines agent in Docker. You can set up a self-hosted agent in Azure Pipelines to run inside a Windows Server Core (for Windows hosts), or Ubuntu container (for Linux hosts) with Docker. This is useful when you want to run agents with outer orchestration, such as Azure Container Instances. In this article, you'll walk through a complete container example, including handling agent self-update.

Both Windows and Linux are supported as container hosts. Windows containers should run on a Windows `vmImage`. To run your agent in Docker, you'll pass a few environment variables to `docker run`, which configures the agent to connect to Azure Pipelines or Azure DevOps Server. Finally, you customize the container to suit your needs. Tasks and scripts might depend on specific tools being available on the container's `PATH`, and it's your responsibility to ensure that these tools are available.

Windows

Enable Hyper-V

Hyper-V isn't enabled by default on Windows. If you want to provide isolation between containers, you must enable Hyper-V. Otherwise, Docker for Windows won't start.

- Enable Hyper-V on Windows 10
- Enable Hyper-V on Windows Server 2016

> **Note**
>
> You must enable virtualization on your machine. It's typically enabled by default. However, if Hyper-V installation fails, refer to your system documentation for how to enable virtualization.

Install Docker for Windows

If you're using Windows 10, you can install the Docker Community Edition. For Windows Server 2016, install the Docker Enterprise Edition.

Switch Docker to use Windows containers

By default, Docker for Windows is configured to use Linux containers. To allow running the Windows container, confirm that Docker for Windows is running the Windows daemon.

Create and build the Dockerfile

Next, create the Dockerfile.

1. Open a command prompt.
2. Create a new directory:

```
PowerShell
mkdir "C:\azp-agent-in-docker\"
```

3. Go to this new directory:

```
PowerShell
cd "C:\azp-agent-in-docker\"
```

4. Save the following content to a file called `C:\azp-agent-in-docker\azp-agent-windows.dockerfile`:

```
Dockerfile
FROM mcr.microsoft.com/windows/servercore:ltsc2022

WORKDIR /azp/

COPY ./start.ps1 ./

CMD powershell .\start.ps1
```

5. Save the following content to `C:\azp-agent-in-docker\start.ps1`:

```
PowerShell
function Print-Header ($header) {
  Write-Host "`n${header}`n" -ForegroundColor Cyan
}
```

```
if (-not (Test-Path Env:AZP_URL)) {
  Write-Error "error: missing AZP_URL environment variable"
  exit 1
}

if (-not (Test-Path Env:AZP_TOKEN_FILE)) {
  if (-not (Test-Path Env:AZP_TOKEN)) {
    Write-Error "error: missing AZP_TOKEN environment variable"
    exit 1
  }

  $Env:AZP_TOKEN_FILE = "\azp\.token"
  $Env:AZP_TOKEN | Out-File -FilePath $Env:AZP_TOKEN_FILE
}

Remove-Item Env:AZP_TOKEN

if ((Test-Path Env:AZP_WORK) -and -not (Test-Path $Env:AZP_WORK)) {
  New-Item $Env:AZP_WORK -ItemType directory | Out-Null
}

New-Item "\azp\agent" -ItemType directory | Out-Null

# Let the agent ignore the token env variables
$Env:VSO_AGENT_IGNORE = "AZP_TOKEN,AZP_TOKEN_FILE"

Set-Location agent

Print-Header "1. Determining matching Azure Pipelines agent..."

$base64AuthInfo =
[Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes(":$(Get-
Content ${Env:AZP_TOKEN_FILE})"))
$package = Invoke-RestMethod -Headers @{Authorization=("Basic
$base64AuthInfo")}
"$(${Env:AZP_URL})/_apis/distributedtask/packages/agent?platform=win-
x64&`$top=1"
$packageUrl = $package[0].Value.downloadUrl

Write-Host $packageUrl

Print-Header "2. Downloading and installing Azure Pipelines agent..."

$wc = New-Object System.Net.WebClient
$wc.DownloadFile($packageUrl, "$(Get-Location)\agent.zip")

Expand-Archive -Path "agent.zip" -DestinationPath "\azp\agent"

try {
  Print-Header "3. Configuring Azure Pipelines agent..."

  .\config.cmd --unattended `
    --agent "$(if (Test-Path Env:AZP_AGENT_NAME) {
${Env:AZP_AGENT_NAME} } else { hostname })" `
    --url "$(${Env:AZP_URL})" `
```

```
    --auth PAT `
    --token "$(Get-Content ${Env:AZP_TOKEN_FILE})" `
    --pool "$(if (Test-Path Env:AZP_POOL) { ${Env:AZP_POOL} } else {
'Default' })" `
    --work "$(if (Test-Path Env:AZP_WORK) { ${Env:AZP_WORK} } else {
'_work' })" `
    --replace

  Print-Header "4. Running Azure Pipelines agent..."

  .\run.cmd
} finally {
  Print-Header "Cleanup. Removing Azure Pipelines agent..."

  .\config.cmd remove --unattended `
    --auth PAT `
    --token "$(Get-Content ${Env:AZP_TOKEN_FILE})"
}
```

6. Run the following command within that directory:

```
PowerShell

docker build --tag "azp-agent:windows" --file "./azp-agent-
windows.dockerfile" .
```

The final image is tagged `azp-agent:windows`.

Start the image

Now that you have created an image, you can run a container. This installs the latest version of the agent, configures it, and runs the agent. It targets the specified agent pool (the `Default` agent pool by default) of a specified Azure DevOps or Azure DevOps Server instance of your choice:

```
PowerShell

docker run -e AZP_URL="<Azure DevOps instance>" -e AZP_TOKEN="<Personal
Access Token>" -e AZP_POOL="<Agent Pool Name>" -e AZP_AGENT_NAME="Docker
Agent - Windows" --name "azp-agent-windows" azp-agent:windows
```

You might need to specify the `--network` parameter if you run into network issues.

```
PowerShell

docker run --network "Default Switch" < . . . >
```

You might need to specify `--interactive` and `--tty` flags (or simply `-it`) if you want to be able to stop the container and remove the agent with `Ctrl` + `C`.

PowerShell

```
docker run --interactive --tty < . . . >
```

If you want a fresh agent container for every pipeline job, pass the --once flag to the `run` command.

PowerShell

```
docker run < . . . > --once
```

With the `--once` flag, you might want to use a container orchestration system, like Kubernetes or Azure Container Instances, to start a new copy of the container when the job completes.

You can control the agent name, the agent pool, and the agent work directory by using optional environment variables.

Linux

Install Docker

Depending on your Linux Distribution, you can either install Docker Community Edition or Docker Enterprise Edition.

Create and build the Dockerfile

Next, create the Dockerfile.

1. Open a terminal.
2. Create a new directory (recommended):

   Bash

   ```
   mkdir ~/azp-agent-in-docker/
   ```

3. Go to this new directory:

```bash
cd ~/azp-agent-in-docker/
```

4. Save the following content to `~/azp-agent-in-docker/azp-agent-linux.dockerfile`:

   - For Alpine:

```dockerfile
FROM alpine

RUN apk update
RUN apk upgrade
RUN apk add bash curl git icu-libs jq

ENV TARGETARCH="linux-musl-x64"

WORKDIR /azp/

COPY ./start.sh ./
RUN chmod +x ./start.sh

ENTRYPOINT ./start.sh
```

   - For Ubuntu 22.04:

```dockerfile
FROM ubuntu:22.04

RUN apt update
RUN apt upgrade -y
RUN apt install -y curl git jq libicu70

# Also can be "linux-arm", "linux-arm64".
ENV TARGETARCH="linux-x64"

WORKDIR /azp/

COPY ./start.sh ./
RUN chmod +x ./start.sh

ENTRYPOINT ./start.sh
```

ⓘ **Note**

Tasks might depend on executables that your container is expected to provide. For instance, you must add the `zip` and `unzip` packages to the `RUN apt install -y` command in order to run the `ArchiveFiles` and `ExtractFiles` tasks. Also, as this is a Linux Ubuntu image for the agent to use, you can customize the image as you need. E.g.: if you need to build .NET applications you can follow the document **Install the .NET SDK or the .NET Runtime on Ubuntu** and add that to your image.

5. Save the following content to `~/azp-agent-in-docker/start.sh`, making sure to use Unix-style (LF) line endings:

Bash

```bash
#!/bin/bash
set -e

if [ -z "${AZP_URL}" ]; then
  echo 1>&2 "error: missing AZP_URL environment variable"
  exit 1
fi

if [ -z "${AZP_TOKEN_FILE}" ]; then
  if [ -z "${AZP_TOKEN}" ]; then
    echo 1>&2 "error: missing AZP_TOKEN environment variable"
    exit 1
  fi

  AZP_TOKEN_FILE="/azp/.token"
  echo -n "${AZP_TOKEN}" > "${AZP_TOKEN_FILE}"
fi

unset AZP_TOKEN

if [ -n "${AZP_WORK}" ]; then
  mkdir -p "${AZP_WORK}"
fi

export AGENT_ALLOW_RUNASROOT="1"

cleanup() {
  trap "" EXIT

  if [ -e ./config.sh ]; then
    print_header "Cleanup. Removing Azure Pipelines agent..."

    # If the agent has some running jobs, the configuration removal
process will fail.
    # So, give it some time to finish the job.
    while true; do
      ./config.sh remove --unattended --auth "PAT" --token $(cat
```

```
"${AZP_TOKEN_FILE}") && break

      echo "Retrying in 30 seconds..."
      sleep 30
    done
  fi
}

print_header() {
  lightcyan="\033[1;36m"
  nocolor="\033[0m"
  echo -e "\n${lightcyan}$1${nocolor}\n"
}

# Let the agent ignore the token env variables
export VSO_AGENT_IGNORE="AZP_TOKEN,AZP_TOKEN_FILE"

print_header "1. Determining matching Azure Pipelines agent..."

AZP_AGENT_PACKAGES=$(curl -LsS \
    -u user:$(cat "${AZP_TOKEN_FILE}") \
    -H "Accept:application/json;" \
    "${AZP_URL}/_apis/distributedtask/packages/agent?
platform=${TARGETARCH}&top=1")

AZP_AGENT_PACKAGE_LATEST_URL=$(echo "${AZP_AGENT_PACKAGES}" | jq -r
".value[0].downloadUrl")

if [ -z "${AZP_AGENT_PACKAGE_LATEST_URL}" -o
"${AZP_AGENT_PACKAGE_LATEST_URL}" == "null" ]; then
  echo 1>&2 "error: could not determine a matching Azure Pipelines
agent"
  echo 1>&2 "check that account "${AZP_URL}" is correct and the token
is valid for that account"
  exit 1
fi

print_header "2. Downloading and extracting Azure Pipelines agent..."

curl -LsS "${AZP_AGENT_PACKAGE_LATEST_URL}" | tar -xz & wait $!

source ./env.sh

trap "cleanup; exit 0" EXIT
trap "cleanup; exit 130" INT
trap "cleanup; exit 143" TERM

print_header "3. Configuring Azure Pipelines agent..."

./config.sh --unattended \
  --agent "${AZP_AGENT_NAME:-$(hostname)}" \
  --url "${AZP_URL}" \
  --auth "PAT" \
  --token $(cat "${AZP_TOKEN_FILE}") \
  --pool "${AZP_POOL:-Default}" \
```

```
  --work "${AZP_WORK:-_work}" \
  --replace \
  --acceptTeeEula & wait $!

print_header "4. Running Azure Pipelines agent..."

chmod +x ./run.sh

# To be aware of TERM and INT signals call ./run.sh
# Running it with the --once flag at the end will shut down the agent
after the build is executed
./run.sh "$@" & wait $!
```

> **Note**
>
> You must also use a container orchestration system, like Kubernetes or **Azure Container Instances**, to start new copies of the container when the work completes.

6. Run the following command within that directory:

```bash
docker build --tag "azp-agent:linux" --file "./azp-agent-linux.dockerfile" .
```

The final image is tagged `azp-agent:linux`.

Start the image

Now that you have created an image, you can run a container. This installs the latest version of the agent, configures it, and runs the agent. It targets the specified agent pool (the `Default` agent pool by default) of a specified Azure DevOps or Azure DevOps Server instance of your choice:

```bash
docker run -e AZP_URL="<Azure DevOps instance>" -e AZP_TOKEN="<Personal Access Token>" -e AZP_POOL="<Agent Pool Name>" -e AZP_AGENT_NAME="Docker Agent - Linux" --name "azp-agent-linux" azp-agent:linux
```

You might need to specify `--interactive` and `--tty` flags (or simply `-it`) if you want to be able to stop the container and remove the agent with `Ctrl` + `C`.

```bash
docker run --interactive --tty < . . . >
```

If you want a fresh agent container for every pipeline job, pass the --once flag to the `run` command.

```bash
docker run < . . . > --once
```

With the `--once` flag, you might want to use a container orchestration system, like Kubernetes or Azure Container Instances, to start a new copy of the container when the job completes.

You can control the agent name, the agent pool, and the agent work directory by using optional environment variables.

Environment variables

Environment variable	Description
AZP_URL	The URL of the Azure DevOps or Azure DevOps Server instance.
AZP_TOKEN	Personal Access Token (PAT) with **Agent Pools (read, manage)** scope, created by a user who has permission to configure agents, at `AZP_URL`.
AZP_AGENT_NAME	Agent name (default value: the container hostname).
AZP_POOL	Agent pool name (default value: `Default`).
AZP_WORK	Work directory (default value: `_work`).

Add tools and customize the container

You have created a basic build agent. You can extend the Dockerfile to include additional tools and their dependencies, or build your own container by using this one as a base layer. Just make sure that the following are left untouched:

- The `start.sh` script is called by the Dockerfile.
- The `start.sh` script is the last command in the Dockerfile.

- Ensure that derivative containers don't remove any of the dependencies stated by the Dockerfile.

Use Docker within a Docker container

In order to use Docker from within a Docker container, you bind-mount the Docker socket.

> ⊗ **Caution**
>
> Doing this has serious security implications. The code inside the container can now run as root on your Docker host.

If you're sure you want to do this, see the bind mount documentation on Docker.com.

Use Azure Kubernetes Service cluster

> ⊗ **Caution**
>
> Please, consider that any docker based tasks will not work on AKS 1.19 or later due to docker in docker restriction. **Docker was replaced with containerd** in Kubernetes 1.19, and Docker-in-Docker became unavailable.

Deploy and configure Azure Kubernetes Service

Follow the steps in Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster by using the Azure portal. After this, your PowerShell or Shell console can use the `kubectl` command line.

Deploy and configure Azure Container Registry

Follow the steps in Quickstart: Create an Azure container registry by using the Azure portal. After this, you can push and pull containers from Azure Container Registry.

Configure secrets and deploy a replica set

1. Create the secrets on the AKS cluster.

```
Bash
```

```
kubectl create secret generic azdevops \
  --from-literal=AZP_URL=https://dev.azure.com/yourOrg \
  --from-literal=AZP_TOKEN=YourPAT \
  --from-literal=AZP_POOL=NameOfYourPool
```

2. Run this command to push your container to Container Registry:

Bash

```bash
docker push "<acr-server>/azp-agent:<tag>"
```

3. Configure Container Registry integration for existing AKS clusters.

> **Note**
>
> If you have multiple subscriptions on the Azure Portal, please, use this command first to select a subscription
>
> Azure CLI
>
> ```azure-cli
> az account set --subscription "<subscription id or subscription name>"
> ```

Azure CLI

```azure-cli
az aks update -n "<myAKSCluster>" -g "<myResourceGroup>" --attach-acr "<acr-name>"
```

4. Save the following content to `~/AKS/ReplicationController.yml`:

yml

```yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azdevops-deployment
  labels:
    app: azdevops-agent
spec:
  replicas: 1 # here is the configuration for the actual agent always running
  selector:
    matchLabels:
      app: azdevops-agent
  template:
```

```yaml
    metadata:
      labels:
        app: azdevops-agent
    spec:
      containers:
      - name: kubepodcreation
        image: <acr-server>/azp-agent:<tag>
        env:
          - name: AZP_URL
            valueFrom:
              secretKeyRef:
                name: azdevops
                key: AZP_URL
          - name: AZP_TOKEN
            valueFrom:
              secretKeyRef:
                name: azdevops
                key: AZP_TOKEN
          - name: AZP_POOL
            valueFrom:
              secretKeyRef:
                name: azdevops
                key: AZP_POOL
        volumeMounts:
        - mountPath: /var/run/docker.sock
          name: docker-volume
      volumes:
      - name: docker-volume
        hostPath:
          path: /var/run/docker.sock
```

This Kubernetes YAML creates a replica set and a deployment, where `replicas: 1` indicates the number or the agents that are running on the cluster.

5. Run this command:

```bash
kubectl apply -f ReplicationController.yml
```

Now your agents will run the AKS cluster.

Set custom MTU parameter

Allow specifying MTU value for networks used by container jobs (useful for docker-in-docker scenarios in k8s cluster).

You need to set the environment variable AGENT_DOCKER_MTU_VALUE to set the MTU value, and then restart the self-hosted agent. You can find more about agent restart

here and about setting different environment variables for each individual agent here.

This allows you to set up a network parameter for the job container, the use of this command is similar to the use of the next command while container network configuration:

```bash
-o com.docker.network.driver.mtu=AGENT_DOCKER_MTU_VALUE
```

Mounting volumes using Docker within a Docker container

If a Docker container runs inside another Docker container, they both use host's daemon, so all mount paths reference the host, not the container.

For example, if we want to mount path from host into outer Docker container, we can use this command:

```bash
docker run ... -v "<path-on-host>:<path-on-outer-container>" ...
```

And if we want to mount path from host into inner Docker container, we can use this command:

```bash
docker run ... -v "<path-on-host>:<path-on-inner-container>" ...
```

But we can't mount paths from outer container into the inner one; to work around that, we have to declare an ENV variable:

```bash
docker run ... --env DIND_USER_HOME=$HOME ...
```

After this, we can start the inner container from the outer one using this command:

```bash
docker run ... -v "${DIND_USER_HOME}:<path-on-inner-container>" ...
```

Common errors

If you're using Windows, and you get the following error:

```
standard_init_linux.go:178: exec user process caused "no such file or
directory"
```

Install Git Bash by downloading and installing git-scm .

Run this command:

```bash
dos2unix ~/azp-agent-in-docker/Dockerfile
dos2unix ~/azp-agent-in-docker/start.sh
git add .
git commit -m "Fixed CR"
git push
```

Try again. You no longer get the error.

Related articles

- Self-hosted Windows agents
- Self-hosted Linux agents
- Self-hosted macOS agents
- Microsoft-hosted agents

Run the agent with a self-signed certificate

Article • 06/22/2023

Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

This topic explains how to run a self-hosted agent with a self-signed certificate.

> **Note**
>
> This article applies to agent versions 2.x and newer.

Work with SSL server certificate

```
Enter server URL > https://corp.tfs.com/tfs
Enter authentication type (press enter for Integrated) >
Connecting to server ...
An error occurred while sending the request.
```

Agent diagnostic log shows:

```
[2017-11-06 20:55:33Z ERR  AgentServer]
System.Net.Http.HttpRequestException: An error occurred while sending the
request. ---> System.Net.Http.WinHttpException: A security error occurred
```

This error may indicate the server certificate you used on your TFS server is not trusted by the build machine. Make sure you install your self-signed ssl server certificate into the OS certificate store.

```
Windows: Windows certificate store
Linux: OpenSSL certificate store
macOS: OpenSSL certificate store for agent version 2.124.0 or below
       Keychain for agent version 2.125.0 or above
```

You can easily verify whether the certificate has been installed correctly by running few commands. You should be good as long as SSL handshake finished correctly even you get a 401 for the request.

```
Windows: PowerShell Invoke-WebRequest -Uri https://corp.tfs.com/tfs -
UseDefaultCredentials
Linux: curl -v https://corp.tfs.com/tfs
macOS: curl -v https://corp.tfs.com/tfs (agent version 2.124.0 or below,
curl needs to be built for OpenSSL)
       curl -v https://corp.tfs.com/tfs (agent version 2.125.0 or above,
curl needs to be built for Secure Transport)
```

If somehow you can't successfully install certificate into your machine's certificate store due to various reasons, like: you don't have permission or you are on a customized Linux machine. The agent version 2.125.0 or above has the ability to ignore SSL server certificate validation error.

> **Important**
>
> This is not secure and not recommended, we highly suggest you to install the certificate into your machine certificate store.

Pass `--sslskipcertvalidation` during agent configuration

```
./config.cmd/sh --sslskipcertvalidation
```

> **Note**
>
> There is limitation of using this flag on Linux and macOS
> The libcurl library on your Linux or macOS machine needs to built with OpenSSL, More Detail

Git get sources fails with SSL certificate problem (Windows agent only)

We ship command-line Git as part of the Windows agent. We use this copy of Git for all Git related operation. When you have a self-signed SSL certificate for your on-premises

TFS server, make sure to configure the Git we shipped to allow that self-signed SSL certificate. There are 2 approaches to solve the problem.

1. Set the following git config in global level by the agent's run as user.

   ```bash
   git config --global http."https://tfs.com/".sslCAInfo certificate.pem
   ```

   > **Note**
   >
   > Setting system level Git config is not reliable on Windows. The system .gitconfig file is stored with the copy of Git we packaged, which will get replaced whenever the agent is upgraded to a new version.

2. Enable git to use `SChannel` during configure with 2.129.0 or higher version agent Pass `--gituseschannel` during agent configuration

   ```
   ./config.cmd --gituseschannel
   ```

   > **Note**
   >
   > Git SChannel has more restrict requirement for your self-signed certificate. Self-singed certificate that generated by IIS or PowerShell command may not be capable with SChanel.

Work with SSL client certificate

IIS has an SSL setting that requires all incoming requests to Azure DevOps Server or TFS must present client certificate in addition to the regular credential.

When that IIS SSL setting enabled, you need to use `2.125.0` or above version agent and follow these extra steps in order to configure the build machine against your Azure DevOps or TFS server.

- Prepare all required certificate information
  - CA certificate(s) in `.pem` format (This should contain the public key and signature of the CA certificate, you need put the root ca certificate and all your

intermediate ca certificates into one `.pem` file)
    - Client certificate in `.pem` format (This should contain the public key and signature of the Client certificate)
    - Client certificate private key in `.pem` format (This should contain only the private key of the Client certificate)
    - Client certificate archive package in `.pfx` format (This should contain the signature, public key and private key of the Client certificate)
    - Use `SAME` password to protect Client certificate private key and Client certificate archive package, since they both have client certificate's private key

- Install CA certificate(s) into machine certificate store
    - Linux: OpenSSL certificate store
    - macOS: System or User Keychain
    - Windows: Windows certificate store

- Pass `--sslcacert`, `--sslclientcert`, `--sslclientcertkey`. `--sslclientcertarchive` and `--sslclientcertpassword` during agent configuration.

```
.\config.cmd/sh --sslcacert ca.pem --sslclientcert clientcert.pem --
sslclientcertkey clientcert-key-pass.pem --sslclientcertarchive
clientcert-archive.pfx --sslclientcertpassword "mypassword"
```

Your client certificate private key password is securely stored on each platform.

```
Linux: Encrypted with a symmetric key based on the machine ID
macOS: macOS Keychain
Windows: Windows Credential Store
```

Learn more about agent client certificate support.

Provision deployment groups

Article • 04/17/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

A deployment group is a logical set of deployment target machines that have agents installed on each one. Deployment groups represent the physical environments; for example, "Dev", "Test", or "Production" environment. In effect, a deployment group is just another grouping of agents, much like an agent pool.

Deployment groups are only available with Classic release pipelines and are different from deployment jobs. A deployment job is a collection of deployment-related steps defined in a YAML file to accomplish a specific task.

With deployment groups you can:

- Specify the security context and runtime targets for the agents. As you create a deployment group, you add users and give them appropriate permissions to administer, manage, view, and use the group.
- Let you view live logs for each server as a deployment takes place, and download logs for all servers to track your deployments down to individual machines.
- Enable you to use machine tags to limit deployment to specific sets of target servers.

Create a deployment group

A deployment group is a set of virtual machines with deployment agents. Every VM of the deployment group interacts with Azure Pipelines to coordinate the deployment tasks.

1. From within your project, select **Pipelines** > **Deployment groups**.

2. Select **Add a deployment group**.

3. Enter a **Deployment group name** and then select **Create**. A registration script will be generated. Select the **Type of target to register** and then select **Use a personal access token in the script for authentication**. Finally, select **Copy script to the clipboard**.

4. Log onto each of your target machines and run the script from an elevated PowerShell command prompt to register it as a target server. When prompted to enter tags for your agent, press *Y* and enter the tag(s) you will use to filter subsets of the servers.

```
d-----         6/18/2021   11:28 AM                azagent


    Directory: C:\azagent


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----         6/18/2021   11:28 AM                A1

  ___                      _____ _                 _ _
 / _ \                    | ___ (_)               | (_)
/ /_\ \_____   _ _ __ ___ | |_/ /_ _ __   ___ | |_ _ __   ___ ___
|  _  |_  / | | | '__/ _ \ |  __/| | '_ \ / _ \ | | '_ \ / _ \/ __|
| | | |/ /| |_| | | |  __/ | |   | | |_) |  __/ | | | | |  __/\__ \
\_| |_/___|\__,_|_|  \___| \_|   |_| .__/ \___|_|_|_| |_|\___||___/
                                   | |
        agent v2.188.1             |_|          (commit b6472da)


>> Connect:
```

After setting up your target servers, the script should return the following message: `Service vstsagent.{organization-name}.{computer-name} started successfully`.

The tags you assign to your target servers allow you to limit deployment to specific servers in a Deployment group job. A tag is limited to 256 characters, but there is no limit to the number of tags you can use.

> ⓘ **Note**

A deployment pool is a set of target servers available to the organization (org-scoped). When you create a new deployment pool for projects in your organization, a corresponding deployment group is automatically provisioned for each project. The deployment groups will have the same target servers as the deployment pool. You can manually trigger an agent version upgrade for your target servers by hovering over the ellipsis (...) in **Deployment Pools** and selecting **Update targets**. See **Agent versions and upgrades** for more details.

Set up agents on deployment groups

Every target server in the deployment group requires a deployment agent. You can install an agent on your target servers in three different ways:

- By running the generated script from the **Deployment Groups** tab.
- If the target servers are Azure VMs, you can easily set up your servers by installing the Azure Pipelines Agent extension on each of the VMs.
- By using the ARM template deployment task in your release pipeline to create a deployment group dynamically.

You can force the agents on the target servers to be upgraded to the latest version without needing to redeploy them by selecting **Update targets** from your deployment groups page.

Monitor release status for deployment groups

When a release pipeline is executing, you can view the live logs for each target server in your deployment group. When the deployment is completed, you can download the log files for every server to examine the deployments and debug any issues.

Share a deployment group with other projects

Deployment groups can be shared with other projects in the same organization. Follow the steps below to provision your deployment group for other projects:

1. From within your project, select **Pipelines** > **Deployment groups**.

2. Select your deployment group and then select **Manage**.

3. Select a project from the list and then select **Save**.

4. You will now notice that there is a new Deployment Group in the project you just included.

Automatically deploy to new target servers

When new target servers get added to a deployment group, you can configure the environment to automatically deploy the last successful release to the new targets.

From your release pipeline definition, select the post deployment icon, and then enable the **Auto redeploy trigger**. Select the events and action as shown below.

Related articles

- Deployment group jobs
- Deploy to Azure VMs using deployment groups
- Provision agents for deployment groups
- Self-hosted Windows agents
- Self-hosted macOS agents
- Self-hosted Linux agents

Provision agents for deployment groups

Article • 03/30/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Deployment groups make it easy to define logical groups of target machines for deployment, and install the required agent on each machine. This topic explains how to create a deployment group, and how to install and provision the agent on each virtual or physical machine in your deployment group.

You can install the agent in any one of these ways:

- Run the script that is generated automatically when you create a deployment group.
- Install the **Azure Pipelines Agent** Azure VM extension on each of the VMs.
- Use the **ARM Template deployment** task in your release pipeline.

For information about agents and pipelines, see:

- Parallel jobs in Team Foundation Server.
- Parallel jobs in Azure Pipelines.
- Pricing for Azure Pipelines features

Run the installation script on the target servers

1. In the **Deployment groups** tab of **Azure Pipelines**, choose **+New** to create a new group.

2. Enter a name for the group, and optionally a description, then choose **Create**.

3. In the **Register machines using command line** section of the next page, select the target machine operating system.

4. Choose **Use a personal access token in the script for authentication**. Learn more.

5. Choose **Copy the script to clipboard**.

6. Log onto each target machine in turn using the account with the appropriate permissions and:

    - Open an Administrator PowerShell command prompt, paste in the script you copied, then execute it to register the machine with this group.

- If you get an error when running the script that a secure channel could not be created, execute this command at the Administrator PowerShell prompt:

  `[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12`

- When prompted to configure tags for the agent, press `Y` and enter any tags you will use to identify subsets of the machines in the group for partial deployments.

  > Tags you assign allow you to limit deployment to specific servers when the deployment group is used in a **Run on machine group** job.

- When prompted for the user account, press *Return* to accept the defaults.
- Wait for the script to finish with the message `Service vstsagent.{organization-name}.{computer-name} started successfully`.

7. In the **Deployment groups** page of **Azure Pipelines**, open the **Machines** tab and verify that the agents are running. If the tags you configured are not visible, refresh the page.

Install the Azure Pipelines Agent Azure VM extension

1. In the **Deployment groups** tab of **Azure Pipelines**, choose **+New** to create a new group.
2. Enter a name for the group, and optionally a description, then choose **Create**.
3. In the Azure portal, for each VM that will be included in the deployment group open the **Extension** blade, choose **+ Add** to open the **New resource** list, and select **Azure Pipelines Agent**.

4. In the **Install extension** blade, specify the name of the Azure Pipelines subscription to use. For example, if the URL is `https://dev.azure.com/contoso`, just specify **contoso**.

5. Specify the project name and the deployment group name.

6. Optionally, specify a name for the agent. If not specified, it uses the VM name appended with `-DG`.

7. Enter the Personal Access Token (PAT) to use for authentication against Azure Pipelines.

8. Optionally, specify a comma-separated list of tags that will be configured on the agent. Tags are not case-sensitive, and each must be no more than 256 characters.

9. Choose **OK** to begin installation of the agent on this VM.

10. Add the extension to any other VMs you want to include in this deployment group.

Use the ARM Template deployment task

> **Important**
>
> These instructions refer to version 2 of the task. Switch your **Task version** from 3 to 2.

You can use the ARM Template deployment task to deploy an Azure Resource Manager (ARM) template that installs the Azure Pipelines Agent Azure VM extension as you create a virtual machine, or to update the resource group to apply the extension after the virtual machine has been created. Alternatively, you can use the advanced deployment options of the ARM Template deployment task to deploy the agent to deployment groups.

Install the "Azure Pipelines Agent" Azure VM extension using an ARM template

An ARM template is a JSON file that declaratively defines a set of Azure resources. The template can be automatically read and the resources provisioned by Azure. In a single template, you can deploy multiple services along with their dependencies.

For a Windows VM, create an ARM template and add a resources element under the `Microsoft.Compute/virtualMachine` resource as shown here:

ARMTemplate

```
"resources": [
  {
    "name": "
[concat(parameters('vmNamePrefix'),copyIndex(),'/TeamServicesAgent')]",
    "type": "Microsoft.Compute/virtualMachines/extensions",
    "location": "[parameters('location')]",
    "apiVersion": "2015-06-15",
    "dependsOn": [
        "[resourceId('Microsoft.Compute/virtualMachines/',
                      concat(parameters('vmNamePrefix'),copyindex()))]"
    ],
    "properties": {
      "publisher": "Microsoft.VisualStudio.Services",
      "type": "TeamServicesAgent",
      "typeHandlerVersion": "1.0",
      "autoUpgradeMinorVersion": true,
      "settings": {
        "VSTSAccountName": "[parameters('VSTSAccountName')]",
        "TeamProject": "[parameters('TeamProject')]",
        "DeploymentGroup": "[parameters('DeploymentGroup')]",
        "AgentName": "[parameters('AgentName')]",
        "AgentMajorVersion": "auto|N",
        "Tags": "[parameters('Tags')]"
```

```
      },
      "protectedSettings": {
      "PATToken": "[parameters('PATToken')]"
     }
    }
   }
]
```

where:

- **VSTSAccountName** is required. The Azure Pipelines subscription to use. Example: If your URL is `https://dev.azure.com/contoso`, just specify `contoso`
- **TeamProject** is required. The project that has the deployment group defined within it
- **DeploymentGroup** is required. The deployment group against which deployment agent will be registered
- **AgentName** is optional. If not specified, the VM name with `-DG` appended will be used
- **Tags** is optional. A comma-separated list of tags that will be set on the agent. Tags are not case sensitive and each must be no more than 256 characters
- **PATToken** is required. The Personal Access Token that will be used to authenticate against Azure Pipelines to download and configure the agent

> **Note**
>
> If you are deploying to a Linux VM, ensure that the `type` parameter in the code is `TeamServicesAgentLinux`.

Troubleshoot the extension

These are some known issues with the extension:

- **Status file getting too big**: This issue occurs on Windows VMs; it has not been observed on Linux VMs. The status file contains a JSON object that describes the current status of the extension. The object is a placeholder to list the operations performed so far. Azure reads this status file and passes the status object as response to API requests. The file has a maximum allowed size; if the size exceeds the threshold, Azure cannot read it completely and gives an error for the status. On each machine reboot, some operations are performed by the extension (even though it might be installed successfully earlier), which append the status file. If the machine is rebooted a large number of times, the status file size exceeds the threshold, which causes this error. The error message reads: `Handler`

`Microsoft.VisualStudio.Services.TeamServicesAgent:1.27.0.2 status file 0.status size xxxxxx bytes is too big. Max Limit allowed: 131072 bytes`. Note that extension installation might have succeeded, but this error hides the actual state of the extension.

We have fixed this issue for machine reboots (version `1.27.0.2` for Windows extension and `1.21.0.1` for Linux extension onward), so on a reboot, nothing will be added to the status file. If you had this issue with your extension before the fix was made (that is, you were having this issue with earlier versions of the extension) and your extension was auto-upadted to the versions with the fix, the issue will still persist. This is because on extension update, the newer version of the extension still works with the earlier status file. Currently, you could still be facing this issue if you are using an earlier version of the extension with the flag to turn off minor version auto-updates, or if a large status file was carried from an earlier extension version to the newer versions that contains the fix, or for any other reason. If that is the case, you can get past this issue by uninstalling and re-installing the extension. Uninstalling the extension cleans up the entire extension directory, so a new status file will be created for fresh install. You need to install latest version of the extension. This solution is a permanent fix, and after following this, you should not face the issue again.

- **Issue with custom data**: This issue is not with the extension, but some customers have reported confusion regarding the custom data location on the VM on switching OS versions. We suggest the following workaround. Python 2 has been deprecated, so we have made the extension to work with Python 3. If you are still using earlier OS versions that don't have Python 3 installed by default, to run the extension, you should either install Python 3 on the VM or switch to OS versions that have Python 3 installed by default. On linux VMs, custom data is copied to the file `/var/lib/waagent/ovf-env.xml` for earlier Microsoft Azure Linux Agent versions, and to `/var/lib/waagent/CustomData` for newer Microsoft Azure Linux Agent versions. It appears that customers who have hardcoded only one of these two paths face issues while switching OS versions because the file does not exist on the new OS version, but the other file is present. So, to avoid breaking the VM provisioning, you should consider both the files in the template so that if one fails, the other should succeed.

For more information about ARM templates, see Define resources in Azure Resource Manager templates.

To use the template:

1. In the **Deployment groups** tab of **Azure Pipelines**, choose **+New** to create a new group.
2. Enter a name for the group, and optionally a description, then choose **Create**.
3. In the **Releases** tab of **Azure Pipelines**, create a release pipeline with a stage that contains the **ARM Template deployment** task.
4. Provide the parameters required for the task such as the Azure subscription, resource group name, location, and template information, then save the release pipeline.
5. Create a release from the release pipeline to install the agents.

Install agents using the advanced deployment options

1. In the **Deployment groups** tab of **Azure Pipelines**, choose **+New** to create a new group.
2. Enter a name for the group, and optionally a description, then choose **Create**.
3. In the **Releases** tab of **Azure Pipelines**, create a release pipeline with a stage that contains the **ARM Template deployment** task.
4. Select the task and expand the **Advanced deployment options for virtual machines** section. Configure the parameters in this section as follows:
    - **Enable Prerequisites**: select **Configure with Deployment Group Agent**.
    - **Azure Pipelines/TFS endpoint**: Select an existing Team Foundation Server/TFS service connection that points to your target. Agent registration for deployment groups requires access to your Visual Studio project. If you do not have an existing service connection, choose **Add** and create one now. Configure it to use a Personal Access Token (PAT) with scope restricted to **Deployment Group**.
    - **Project**: Specify the project containing the deployment group.
    - **Deployment Group**: Specify the name of the deployment group against which the agents will be registered.
    - **Copy Azure VM tags to agents**: When set (ticked), any tags already configured on the Azure VM will be copied to the corresponding deployment group agent. By default, all Azure tags are copied using the format `Key: Value`. For example, `Role: Web`.

5. Provide the other parameters required for the task such as the Azure subscription, resource group name, and location, then save the release pipeline.

6. Create a release from the release pipeline to install the agents.

Related topics

- Run on machine group job
- Deploy an agent on Windows
- Deploy an agent on macOS
- Deploy an agent on Linux

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and feel free to post your questions, search for answers, or suggest a feature on our Azure DevOps Developer Community. Support page.

Deployment group jobs

Article • 01/27/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Deployment groups in Classic pipelines make it easy to define groups of target servers for deployment. Tasks that you define in a deployment group job run on some or all of the target servers, depending on the arguments you specify for the tasks and the job itself.

You can select specific sets of servers from a deployment group to receive the deployment by specifying the machine tags that you've defined for each server in the deployment group. You can also specify the proportion of the target servers that the pipeline should deploy to at the same time. This ensures that the app running on these servers is capable of handling requests while the deployment is taking place.

If you're using a YAML pipeline, you should use Environments with virtual machines instead.

YAML

> **Note**
>
> Deployment group jobs are not supported in YAML. You can use **Virtual machine resources in Environments** to do a rolling deployment to VMs in YAML pipelines.

Rolling deployments can be configured by specifying the keyword `rolling:` under `strategy:` node of a deployment job.

YAML

```
strategy:
  rolling:
    maxParallel: [ number or percentage as x% ]
    preDeploy:
      steps:
      - script: [ script | bash | pwsh | powershell | checkout | task |
templateReference ]
    deploy:
      steps:
```

```
    ...
  routeTraffic:
    steps:
    ...
  postRouteTraffic:
    steps:
    ...
  on:
    failure:
      steps:
      ...
    success:
      steps:
      ...
```

Timeouts

Use the job timeout to specify the timeout in minutes for jobs in this job. A zero value for this option means that the timeout is effectively infinite and so, by default, jobs run until they complete or fail. You can also set the timeout for each task individually - see task control options. Jobs targeting Microsoft-hosted agents have additional restrictions on how long they may run.

Related articles

- Jobs
- Conditions

Deploy to Azure VMs using deployment groups in Azure Pipelines

Article • 04/05/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

In earlier versions of Azure Pipelines, applications that needed to be deployed to multiple servers required a significant amount of planning and maintenance. Windows PowerShell remoting had to be enabled manually, required ports opened, and deployment agents installed on each of the servers. The pipelines then had to be managed manually if a roll-out deployment was required.

All the above challenges have been evolved seamlessly with the introduction of the Deployment Groups.

A deployment group installs a deployment agent on each of the target servers in the configured group and instructs the release pipeline to gradually deploy the application to those servers. Multiple pipelines can be created for the roll-out deployments so that the latest version of an application can be delivered in a phased manner to multiple user groups for validation of newly introduced features.

> **Note**
>
> Deployment groups are a concept used in Classic pipelines. If you are using YAML pipelines, see **Environments**.

In this tutorial, you learn about:

- ✓ Provisioning VM infrastructure to Azure using a template
- ✓ Creating an Azure Pipelines deployment group
- ✓ Creating and running a CI/CD pipeline to deploy the solution with a deployment group

Prerequisites

- A Microsoft Azure account.
- An Azure DevOps organization.

Use the Azure DevOps Demo Generator to provision the tutorial project on your Azure DevOps organization.

Setting up the Azure deployment environment

The following resources are provisioned on the Azure using an ARM template:

- Six Virtual Machines (VM) web servers with IIS configured

- SQL server VM (DB server)
- Azure Network Load Balancer

1. Click the **Deploy to Azure** button below to initiate resource provisioning. Provide all the necessary information and select **Purchase**. You may use any combination of allowed administrative usernames and passwords as they are not used again in this tutorial. The **Env Prefix Name** is prefixed to all of the resource names in order to ensure that those resources are generated with globally unique names. Try to use something personal or random, but if you see a naming conflict error during validation or creation, try changing this parameter and running again.

   Deploy to Azure button.

> ⓘ **Note**
>
> It takes approximately 10-15 minutes to complete the deployment. If you receive any naming conflict errors, try changing the parameter you provide for **Env Prefix Name**.

2. Once the deployment completes, you can review all of the resources generated in the specified resource group using the Azure portal. Select the DB server VM with **sqlSrv** in its name to view its details.

Name	Type	Location
cust1sqlPip	Public IP address	West US 2
cust1sqlSrv14	Virtual machine	West US 2
cust1sqlSrv14Nic	Network interface	West US 2
cust1stgjo5zndv53idtw	Storage account	West US 2
cust1Vnet	Virtual network	West US 2
cust1webSrv0	Virtual machine	West US 2
cust1webSrv1	Virtual machine	West US 2
cust1webSrv2	Virtual machine	West US 2

3. Make a note of the **DNS name**. This value is required in a later step. You can use the copy button to copy it to the clipboard.

Resource group (change)	: DeploymentGroups	Operating system	: Windows (Windows Server 2016 Datacenter)
Status	: Running	Size	: Standard D2 v3 (2 vcpus, 7 GiB memory)
Location	: West US 2	Public IP address	: 52.175.214.35
Subscription (change)	: Contoso	Colocation status	: N/A
Subscription ID	: 01234567-89ab-cdef-0123-456789abcdef	Virtual network/subnet	: cust1Vnet/DBSubnetName
		DNS name	: cust1sqljo5zndv53idtw.westus2.cloudapp.azure.com

Creating and configuring a deployment group

Azure Pipelines makes it easier to organize servers required for deploying applications. A deployment group is a collection of machines with deployment agents. Each of the machines interacts with Azure Pipelines to coordinate the deployment of the app.

Since there is no configuration change required for the build pipeline, the build is triggered automatically after the project is provisioned. When you queue a release later on, this build is used.

1. Navigate to the Azure DevOps project created by the demo generator.

2. From under **Pipelines**, navigate to **Deployment groups**.

3. Select **Add a deployment group**.

4. Enter the **Deployment group name** of **Release** and select **Create**. A registration script is generated. You can register the target servers using the script provided if working on your own. However, in this tutorial, the target servers are automatically registered as part of the release pipeline. The release definition uses stages to deploy the application to the target servers. A stage is a logical grouping of the tasks that defines the runtime target on which the tasks will execute. Each deployment group stage executes tasks on the machines defined in the deployment group.

5. From under **Pipelines**, navigate to **Releases**. Select the release pipeline named **Deployment Groups** and select **Edit**.

6. Select the **Tasks** tab to view the deployment tasks in pipeline. The tasks are organized as three stages called **Agent phase**, **Deployment group phase**, and **IIS Deployment phase**.

7. Select the **Agent phase**. In this stage, the target servers are associated with the deployment group using the Azure Resource Group Deployment task. To run, an agent pool and specification must be defined. Select the **Azure Pipelines** pool and **windows-latest** specification.

8. Select the **Azure Resource Group Deployment** task. Configure a service connection to the Azure subscription used earlier to create infrastructure. After authorizing the connection,

select the resource group created for this tutorial.

9. This task will run on the virtual machines hosted in Azure, and will need to be able to connect back to this pipeline in order to complete the deployment group requirements. To secure the connection, they will need a **personal access token (PAT)**. From the **User settings** dropdown, open **Personal access tokens** in a new tab. Most browsers support opening a link in a new tab via right-click context menu or **Ctrl+Click**.

10. In the new tab, select **New Token**.

11. Enter a name and select the **Full access** scope. Select **Create** to create the token. Once created, copy the token and close the browser tab. You return to the Azure Pipeline editor.

12. Under **Azure Pipelines service connection**, select **New**.

13. Enter the **Connection URL** to the current instance of Azure DevOps. This URL is something like `https://dev.azure.com/[Your account]`. Paste in the **Personal Access Token** created earlier and specify a **Service connection name**. Select **Verify and save**.

> **Note**
>
> To register an agent, you must be a member of the Administrator role in the agent pool. The identity of the agent pool administrator is needed only at the time of registration. The administrator identity isn't persisted on the agent, and it's not used in any subsequent communication between the agent and Azure Pipelines or TFS. After the agent is registered, there's no need to renew the personal access token because it's required only at the time of registration.

14. Select the current **Team project** and the **Deployment group** created earlier.

15. Select the **Deployment group phase** stage. This stage executes tasks on the machines defined in the deployment group. This stage is linked to the **SQL-Svr-DB** tag. Choose the **Deployment Group** from the dropdown.

16. Select the **IIS Deployment phase** stage. This stage deploys the application to the web servers using the specified tasks. This stage is linked to the **WebSrv** tag. Choose the **Deployment Group** from the dropdown.

17. Select the **Disconnect Azure Network Load Balancer** task. As the target machines are connected to the NLB, this task will disconnect the machines from the NLB prior to the deployment and reconnect them back to the NLB after the deployment. Configure the task to use the Azure connection, resource group, and load balancer (there should only be one).

18. Select the **IIS Web App Manage** task. This task runs on the deployment target machines registered with the deployment group configured for the task/stage. It creates a web app and application pool locally with the name **PartsUnlimited** running under the port **80**

19. Select the **IIS Web App Deploy** task. This task runs on the deployment target machines registered with the deployment group configured for the task/stage. It deploys the application to the IIS server using **Web Deploy**.

20. Select the **Connect Azure Network Load Balancer** task. Configure the task to use the Azure connection, resource group, and load balancer (there should only be one).

21. Select the **Variables** tab and enter the variable values as below.

Variable Name	Variable Value

Variable Name	Variable Value
DatabaseName	PartsUnlimited-Dev
DBPassword	P2ssw0rd@123
DBUserName	sqladmin
DefaultConnectionString	Data Source=[YOUR_DNS_NAME];Initial Catalog=PartsUnlimited-Dev;User ID=sqladmin;Password=P2ssw0rd@123;MultipleActiveResultSets=False;Connection Timeout=30;
ServerName	localhost

> **Important**
>
> Make sure to replace your SQL server DNS name (which you noted from Azure portal earlier) in **DefaultConnectionString** variable.

Your DefaultConnectionString should be similar to this string after replacing the SQL DNS:

```
Data Source=cust1sqljo5zndv53idtw.westus2.cloudapp.azure.com;Initial Catalog=PartsUnlimited-Dev;User ID=sqladmin;Password=P2ssw0rd@123;MultipleActiveResultSets=False;Connection Timeout=30;
```

The final variable list should look something like this:

> **Note**
>
> You may receive an error that the `DefaultConnectionString` variable must be saved as a secret. If that happens, select the variable and click the padlock icon that appears next to its value to protect it.

Queuing a release and reviewing the deployment

1. Select **Save** and confirm.

2. Select **Create release** and confirm. Follow the release through to completion. The deployment is then ready for review.

3. In the Azure portal, open one of the web VMs in your resource group. You can select any that have `websrv` in the name.

4. Copy the **DNS** of the VM. The **Azure Load Balancer** will distribute incoming traffic among healthy instances of servers defined in a load-balanced set. As a result, the **DNS** of all web server instances is the same.

5. Open a new browser tab to the DNS of the VM. Confirm the deployed app is running.

Summary

In this tutorial, you deployed a web application to a set of Azure VMs using Azure Pipelines and Deployment Groups. While this scenario covered a handful of machines, you can easily scale the process up to support hundreds, or even thousands, of machines using virtually any configuration.

Cleaning up resources

This tutorial created an Azure DevOps project and some resources in Azure. If you're not going to continue to use these resources, delete them with the following steps:

1. Delete the Azure DevOps project created by the Azure DevOps Demo Generator.
2. All Azure resources created during this tutorial were assigned to the resource group specified during creation. Deleting that group will delete the resources they contain. This deletion can be done via the CLI or portal.

Next steps

Provision agents for deployment groups

Set retention policies for builds, releases, and tests

Article • 05/03/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Retention policies let you set how long to keep runs, releases, and tests stored in the system. To save storage space, you want to delete older runs, tests, and releases.

The following retention policies are available in Azure DevOps in your **Project settings**:

1. **Pipeline** - Set how long to keep artifacts, symbols, attachments, runs, and pull request runs.
2. **Release (classic)** - Set whether to save builds and view the default and maximum retention settings.
3. **Test** - Set how long to keep automated and manual test runs, results, and attachments.

> **Note**
>
> If you are using an on-premises server, you can also specify retention policy defaults for a project and when releases are permanently destroyed. Learn more about **release retention** later in this article.

Prerequisites

By default, members of the Contributors, Build Admins, Project Admins, and Release Admins groups can manage retention policies.

To manage retention policies, you must have one of the following subscriptions:

- Enterprise
- Test Professional
- MSDN Platforms

You can also buy monthly access to Azure Test Plans and assign the Basic + Test Plans access level. See Testing access by user role.

Configure retention policies

1. Sign in to your project.
2. Go to the **Settings** tab of your project's settings.
3. Select **Settings** or **Release retention** under **Pipelines** or **Retention** under **Test**.
    - Select **Settings** to configure retention policies for runs, artifacts, symbols, attachments, and pull request runs.
    - Select **Release retention** to set up your release retention policies and configure when to delete or permanently destroy releases.
    - Select **Retention** to set up how long to keep manual and automated test runs.

Set run retention policies

In most cases, you don't need to retain completed runs longer than a certain number of days. Using retention policies, you can control **how many days** you want to keep each run before deleting it.

1. Go to the ⚙ **Settings** tab of your project's settings.

2. Select **Settings** in the Pipelines section.

- Set the number of days to keep artifacts, symbols, and attachments.
- Set the number of days to keep runs
- Set the number of days to keep pull request runs
- Set the number of recent runs to keep for each pipeline

> ⚠ **Warning**
>
> Azure DevOps no longer supports per-pipeline retention rules. The only way to configure retention policies for YAML and classic pipelines is through the project settings described above. You can no longer configure per-pipeline retention policies.

The setting for number of recent runs to keep for each pipeline requires a little more explanation. The interpretation of this setting varies based on the type of repository you build in your pipeline.

- **Azure Repos:** Azure Pipelines retains the configured number of latest runs for the pipeline's default branch and for each protected branch of the repository. A branch that has any branch policies configured is considered to be a protected branch.

  As an example, consider a repository with two branches, `main` and `release`. Imagine the `pipeline's default branch` is the `main` branch, and the `release` branch has a branch policy, making it a protected branch. In this case, if you configured the policy to retain three runs, then both the latest three runs of `main` and the latest three runs of the `release` branch are retained. In addition, the latest three runs of this pipeline (irrespective of the branch) are also retained.

  To clarify this logic further, let us say the list of runs for this pipeline is as follows, with the most recent run at the top. The table shows which runs will be retained if you have configured to retain the latest three runs (ignoring the effect of the number of days setting):

Run #	Branch	Retained / Not retained	Why?
Run 10	main	Retained	Latest 3 for main and Latest 3 for pipeline
Run 9	branch1	Retained	Latest 3 for pipeline
Run 8	branch2	Retained	Latest 3 for pipeline
Run 7	main	Retained	Latest 3 for main
Run 6	main	Retained	Latest 3 for main

Run #	Branch	Retained / Not retained	Why?
Run 5	main	Not retained	Neither latest 3 for main, nor for pipeline
Run 4	main	Not retained	Neither latest 3 for main, nor for pipeline
Run 3	branch1	Not retained	Neither latest 3 for main, nor for pipeline
Run 2	release	Retained	Latest 3 for release
Run 1	main	Not retained	Neither latest 3 for main, nor for pipeline

- **All other Git repositories:** Azure Pipelines retains the configured number of latest runs for the whole pipeline.
- **TFVC:** Azure Pipelines retains the configured number of latest runs for the whole pipeline, irrespective of the branch.

What parts of the run get deleted

The following information is deleted when a run is deleted:

- Logs
- All pipeline and build artifacts
- All symbols
- Binaries
- Test results
- Run metadata
- Source labels (TFVC) or tags (Git)

Universal packages, NuGet, npm, and other packages are not tied to pipelines retention.

When are runs deleted

Your retention policies are processed once a day. The time that the policies get processed variables because we spread the work throughout the day for load-balancing purposes. There is no option to change this process.

A run is deleted if all of the following conditions are true:

- It exceeds the number of days configured in the retention settings
- It is not one of the recent runs as configured in the retention settings
- It is not marked to be retained indefinitely
- It is not retained by a release

Automatically set retention lease on pipeline runs

Retention leases are used to manage the lifetime of pipeline runs beyond the configured retention periods. Retention leases can be added or deleted on a pipeline run by calling the Lease API. This API can be invoked within the pipeline using a script and using predefined variables for runId and definitionId.

A retention lease can be added on a pipeline run for a specific period. For example, a pipeline run which deploys to a test environment can be retained for a shorter duration while a run deploying to production environment can be retained longer.

Manually set retention lease on pipeline runs

You can manually set a pipeline run to be retained using the More actions menu on the Pipeline run details page.

Delete a run

You can delete runs using the More actions menu on the Pipeline run details page.

> **Note**
>
> If any retention policies currently apply to the run, they must be removed before the run can be deleted. For instructions, see **Pipeline run details - delete a run**.

Set release retention policies

The release retention policies for a classic release pipeline determine how long a release and the run linked to it are retained. Using these policies, you can control **how many days** you want to keep each release after it has been last modified or deployed and the **minimum number of releases** that should be retained for each pipeline.

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. For example, if you specify to retain a minimum of three releases, the most recent three will be retained indefinitely - irrespective of the number of days specified. However, you can manually delete these releases when you no longer require them. See FAQ below for more details about how release retention works.

As an author of a release pipeline, you can customize retention policies for releases of your pipeline on the **Retention** tab.

The retention policy for YAML and build pipelines is the same. You can see your pipeline's retention settings in **Project Settings** for **Pipelines** in the **Settings** section.

Global release retention policy

If you are using an on-premises Team Foundation Server or Azure DevOps Server, you can specify release retention policy defaults and maximums for a project. You can also specify when releases are permanently destroyed (removed from the **Deleted** tab in the build explorer).

If you are using Azure DevOps Services, you can view but not change these settings for your project.

Global release retention policy settings can be reviewed from the **Release retention** settings of your project:

- Azure DevOps Services: `https://dev.azure.com/{organization}/{project}/_settings/release?app=ms.vss-build-web.build-release-hub-group`

- On-premises: `https://{your_server}/tfs/{collection_name}/{project}/_admin/_apps/hub/ms.vss-releaseManagement-web.release-project-admin-hub`

The **maximum retention policy** sets the upper limit for how long releases can be retained for all release pipelines. Authors of release pipelines cannot configure settings for their definitions beyond the values specified here.

The **default retention policy** sets the default retention values for all the release pipelines. Authors of build pipelines can override these values.

The **destruction policy** helps you keep the releases for a certain period of time after they are deleted. This policy cannot be overridden in individual release pipelines.

Set collection-level retention policies

For on-premises servers, you can also set the collection-level retention policies with custom retention rules. These retention policies apply to Classic build pipelines. The page at `https://{your_server}/{collection_name}/_settings/buildqueue` governs your maximum values and default values.

Use the Copy Files task to save data longer

You can use the Copy Files task to save your build and artifact data for longer than what is set in the retention policies. The **Copy Files task** is preferable to the Publish Build Artifacts task because data saved with the **Publish Build Artifacts task** will get periodically cleaned up and deleted.

YAML

YAML

```yaml
- task: CopyFiles@2
  displayName: 'Copy Files to: \\mypath\storage\$(Build.BuildNumber)'
```

```
    inputs:
      SourceFolder: '$(Build.SourcesDirectory)'
      Contents: '_buildOutput/**'
      TargetFolder: '\\mypath\storage\$(Build.BuildNumber)'
```

FAQ

If I mark a run or a release to be retained indefinitely, does the retention policy still apply?

No. Neither the pipeline's retention policy nor the maximum limits set by the administrator are applied when you mark an individual run or release to be retained indefinitely. It will remain until you stop retaining it indefinitely.

How do I specify that runs deployed to production will be retained longer?

If you use classic releases to deploy to production, then customize the retention policy on the release pipeline. Specify the number of days that releases deployed to production must be retained. In addition, indicate that runs associated with that release are to be retained. This will override the run retention policy.

If you use multi-stage YAML pipelines to deploy to production, the only retention policy you can configure is in the project settings. You cannot customize retention based on the environment to which the build is deployed.

I did not mark runs to be retained indefinitely. However, I see a large number of runs being retained. How can I prevent this?

This could be for one of the following reasons:

- The runs are marked by someone in your project to be retained indefinitely.
- The runs are consumed by a release, and the release holds a retention lock on these runs. Customize the release retention policy as explained above.

If you believe that the runs are no longer needed or if the releases have already been deleted, then you can manually delete the runs.

How does 'minimum releases to keep' setting work?

Minimum releases to keep are defined at stage level. It denotes that Azure DevOps will always retain the given number of last deployed releases for a stage even if the releases are out of retention period. A release will be considered under minimum releases to keep for a stage only when the deployment started on that stage. Both successful and failed deployments are considered. Releases pending approval are not considered.

How is retention period decided when release is deployed to multiple stages having different retention period?

Final retention period is decided by considering days to retain settings of all the stages on which release is deployed and taking max days to keep among them. **Minimum releases to keep** is governed at stage level and do not change based on release deployed to multiple stages or not. Retain associated artifacts will be applicable when release is deployed to a stage for which it is set true.

I deleted a stage for which I have some old releases. What retention will be considered for this case?

As the stage is deleted, so the stage level retention settings are not applicable now. Azure DevOps will fall back to project level default retention for such case.

My organization requires us to retain builds and releases longer than what is allowed in the settings. How can I request a longer retention?

The only way to retain a run or a release longer than what is allowed through retention settings is to manually mark it to be retained indefinitely. There is no way to configure a longer retention setting manually. Please reach out to Azure DevOps Support for assistance.

You can also explore the possibility of using the REST APIs in order to download information and artifacts about the runs and upload them to your own storage or artifact repository.

I lost some runs. Is there a way to get them back?

If you believe that you have lost runs due to a bug in the service, create a support ticket immediately to recover the lost information. If a build definition was manually deleted

more than a week earlier, it will not be possible to recover it. If the runs were deleted as expected due to a retention policy, it will not be possible to recover the lost runs.

How do I use the `Build.Cleanup` capability of agents?

Setting a `Build.Cleanup` capability on agents will cause the pool's cleanup jobs to be directed to just those agents, leaving the rest free to do regular work. When a pipeline run is deleted, artifacts stored outside of Azure DevOps are cleaned up through a job run on the agents. When the agent pool gets saturated with cleanup jobs, this can cause a problem. The solution to that is to designate a subset of agents in the pool that are the cleanup agents. If any agents have `Build.Cleanup` set, only those agents will run the cleanup jobs, leaving the rest of the agents free to continue running pipeline jobs. The Cleanup functionality can be enabled by navigating to **Agent** > **Capabilities** and setting `Build.Cleanup` equal to `1`.

What happens to file share Artifacts when the build is deleted

When a build with file share Artifacts is deleted, a new build task is queued on a build agent to clean up those files. An agent is picked to perform this task based on the following criteria: Is there an agent with `Build.Cleanup` capability available? Is the agent that ran the build available? Is an agent from the same pool available? Is an agent from a similar pool available? Is any agent available?

Are automated test results that are published as part of a release retained until the release is deleted?

Test results published within a stage of a release are retained as specified by the retention policy configured for the test results. The test results do not get retained until the release is retained. If you need the test results as long as the release, set the retention settings for automated test runs in the Project settings accordingly to Never delete. This makes sure the test results are deleted only when the release is deleted.

Are manual test results deleted?

No. Manual test results are not deleted.

How do I preserve my version control labels or tags?

> **Caution**
>
> Any version control labels or tags that are applied during a build pipeline that arent automatically created from the Sources task will be preserved, even if the build is deleted. However, any version control labels or tags that are automatically created from the Sources task during a build are considered part of the build artifacts and will be deleted when the build is deleted.

If version control labels or tags need to be preserved, even when the build is deleted, they will need to be either applied as part of a task in the pipeline, manually labeled outside of the pipeline, or the build will need to be retained indefinitely.

What happens to pipelines that are consumed in other pipelines?

Classic releases retain pipelines that they consume automatically.

Related articles

- Control how long to keep test results
- Delete test artifacts

Configure and pay for parallel jobs

Article • 04/27/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Learn how to estimate how many parallel jobs you need and buy more parallel jobs for your organization.

> **Note**
>
> We have temporarily disabled the free grant of parallel jobs for public projects and for certain private projects in new organizations. However, you can request this grant by submitting **a request**. Existing organizations and projects are not affected. Please note that it takes us 2-3 business days to respond to your free tier requests.

What is a parallel job?

When you define a pipeline, you can define it as a collection of jobs. When a pipeline runs, you can run multiple jobs as part of that pipeline. Each running job consumes a *parallel job* that runs on an agent. When there aren't enough parallel jobs available for your organization, the jobs are queued up and run one after the other.

In Azure Pipelines, you can run parallel jobs on Microsoft-hosted infrastructure or your own (self-hosted) infrastructure. Each parallel job allows you to run a single job at a time in your organization. You don't need to pay for parallel jobs if you're using an on-premises server. The concept of parallel jobs only applies to Azure DevOps Services.

Microsoft-hosted vs. self-hosted parallel jobs

If you want to run your jobs on machines that Microsoft manages, use *Microsoft-hosted parallel jobs*. Your jobs will run on Microsoft-hosted agents.

If you want Azure Pipelines to orchestrate your builds and releases, but use your own machines to run them, use *self-hosted parallel jobs*. For self-hosted parallel jobs, you'll start by deploying our self-hosted agents on your machines. You can register any number of these self-hosted agents in your organization.

How much do parallel jobs cost?

We provide a *free tier* of service by default in every organization for both hosted and self-hosted parallel jobs. Parallel jobs are purchased at the organization level, and they're shared by all projects in an organization.

Microsoft-hosted

For Microsoft-hosted parallel jobs, you can get up to 10 free Microsoft-hosted parallel jobs that can run for up to 360 minutes (6 hours) each time for **public projects**. When you create a new Azure DevOps organization, you aren't given this free grant by default.

For **private projects**, you can get one free job that can run for up to 60 minutes each time. When you create a new Azure DevOps organization, you may not always be given this free grant by default.

To request the free grant for public or private projects, submit a request.

> **Note**
>
> It takes us 2-3 business days to respond to your free tier request.

There's no time limit on parallel jobs for public projects and a 30 hour time limit per month for private projects.

	Number of parallel jobs	Time limit
Public project	Up to 10 free Microsoft-hosted parallel jobs that can run for up to 360 minutes (6 hours) each time	No overall time limit per month
Private project	One free job that can run for up to 60 minutes each time	1,800 minutes (30 hours) per month

When the free tier is no longer sufficient, you can pay for additional capacity per parallel job. For pricing cost per parallel job, see the Azure DevOps pricing page. Paid parallel jobs remove the monthly time limit and allow you to run each job for up to 360 minutes (6 hours).

Buy Microsoft-hosted parallel jobs.

New organizations have a maximum limit of 25 parallel jobs for Microsoft-hosted agents. Contact support to request a limit increase, subject to capacity in your

organization's region.

When you purchase your first Microsoft-hosted parallel job, the number of parallel jobs you have in the organization is still one. To be able to run two jobs concurrently, you'll need to purchase two parallel jobs if you're currently on the free tier. The first purchase only removes the time limits on the first job.

> **Tip**
>
> If your pipeline exceeds the maximum job timeout, try splitting your pipeline into multiple jobs. For more information on jobs, see **Specify jobs in your pipeline**.

How many parallel jobs do I need?

As the number of queued builds and releases exceeds the number of parallel jobs you have, your build and release queues grow longer. When you find the queue delays are too long, you can purchase additional parallel jobs as needed. There are several methods you can use to check your parallel job limits and job history.

View job history using the pool consumption report

You can use the **Pool consumption report**, available on the **Analytics** tab of your agent pool, to see a chart of running and queued jobs graphed with your parallel jobs for the previous 30 days. If you have a backlog of queued jobs and your running jobs are at the concurrency limit, you may wish to purchase more parallel jobs. For more information, see Pool consumption report.

Check the parallel jobs setting directly

Figure out how many parallel jobs you need by first seeing how many parallel jobs your organization currently uses:

1. Browse to **Organization settings** > **Pipelines** > **Parallel jobs**.

   URL example: `https://{your_organization}/_admin/_buildQueue?_a=resourceLimits`

2. View the maximum number of parallel jobs that are available in your organization.

3. Select **View in-progress jobs** to display all the builds and releases that are actively consuming an available parallel job or that are queued waiting for a parallel job to be available.

Estimate costs

A simple rule of thumb: Estimate that you'll need one parallel job for every four to five users in your organization.

In the following scenarios, you might need multiple parallel jobs:

- If you have multiple teams, and if each of them require CI, you'll likely need a parallel job for each team.
- If your CI trigger applies to multiple branches, you'll likely need a parallel job for each active branch.
- If you develop multiple applications by using one organization or server, you'll likely need additional parallel jobs: one to deploy each application at the same time.

How do I buy more parallel jobs?

To buy more parallel jobs:

- Billing must be set up for your organization
- You need to be a member of the **Project Collection Administrators** group.

Buy parallel jobs

Buy more parallel jobs within your organization settings:

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

2. Select **Organization settings**.

3. Select **Parallel jobs** under Pipelines, and then select either **Change** for Microsoft-hosted jobs or **Change** for self-hosted jobs.

4. Enter your desired amount, and then **Save**.

5. It may take up to 30 minutes for your additional parallel jobs to become available to use.

For pricing cost per parallel job, see the Azure DevOps pricing page.

How do I change the quantity of parallel jobs for my organization?

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

2. Select **Organization settings**.

3. Select **Parallel jobs** under Pipelines, and then select either **Purchase parallel jobs** or **Change** for Microsoft-hosted jobs or **Change** for self-hosted jobs.

4. Enter a lesser or greater quantity of Microsoft-hosted or self-hosted jobs, and then select **Save**.

> **Important**
>
> Hosted XAML build controller isn't supported. If you have an organization where you need to run **XAML builds**, set up an **on-premises build server** and switch to an **on-premises build controller**. For more information about the hosted XAML model, see **Get started with XAML**.

5. It may take up to 30 minutes for the new number of parallel jobs to become active.

How is a parallel job consumed in DevOps Services?

Consider an organization that has only one Microsoft-hosted parallel job. This job allows users in that organization to collectively run only one job at a time. When additional jobs are triggered, they're queued and will wait for the previous job to finish.

If you use release or YAML pipelines, then a run consumes a parallel job only when it's being actively deployed to a stage. While the release is waiting for an approval or a

manual intervention, it doesn't consume a parallel job.

When you run a server job or deploy to a deployment group using release pipelines, you don't consume any parallel jobs.

1. FabrikamFiber CI Build 102 (main branch) starts first.
2. Deployment of FabrikamFiber Release 11 gets triggered by completion of FabrikamFiber CI Build 102.
3. FabrikamFiber CI Build 101 (feature branch) is triggered. The build can't start yet because Release 11's deployment is active. So the build stays queued.
4. Release 11 waits for approvals. Fabrikam CI Build 101 starts because a release that's waiting for approvals doesn't consume a parallel job.
5. Release 11 is approved. It resumes only after Fabrikam CI Build 101 is completed.

FAQ

How do I qualify for the free tier of public projects?

You qualify for the free tier limits for public projects if you meet both of these conditions:

- Your pipeline is part of an Azure Pipelines public project.
- Your pipeline builds a public repository from GitHub or from the same public project in your Azure DevOps organization.

For information on how to apply for the grant of free parallel jobs, see How much do parallel jobs cost (Microsoft-hosted)?

Can I assign a parallel job to a specific project or agent pool?

Currently, there isn't a way to partition or dedicate parallel job capacity to a specific project or agent pool. For example:

- You purchase two parallel jobs in your organization.
- You start two runs in the first project, and both the parallel jobs are consumed.
- You start a run in the second project. That run won't start until one of the runs in your first project is completed.

Are there limits on who can use Azure Pipelines?

You can have as many users as you want when you're using Azure Pipelines. There's no per-user charge for using Azure Pipelines. Users with both basic and stakeholder access can author as many builds and releases as they want.

Are there any limits on the number of builds and release pipelines that I can create?

No. You can create hundreds or even thousands of pipelines for no charge. You can register any number of self-hosted agents for no charge.

As a Visual Studio Enterprise subscriber, do I get additional parallel jobs for TFS and Azure Pipelines?

Yes. Visual Studio Enterprise subscribers get one parallel job in Team Foundation Server 2017 or later and one self-hosted parallel job in each Azure DevOps Services organization where they are a member.

What about the option to pay for hosted agents by the minute?

Some of our earlier customers are still on a per-minute plan for the hosted agents. In this plan, you pay $0.05/minute for the first 20 hours after the free tier, and $0.01/minute after 20 hours. Because of the following limitations in this plan, you might want to consider moving to the parallel jobs model:

- When you're using the per-minute plan, you can run only one job at a time.

- If you run builds for more than 14 paid hours in a month, the per-minute plan might be less cost-effective than the parallel jobs model.

I use XAML build controllers with my organization. How am I charged for those?

You can register one XAML build controller for each self-hosted parallel job in your organization. Your organization gets at least one free self-hosted parallel job, so you can register one XAML build controller for no additional charge. For each additional XAML build controller, you'll need an additional self-hosted parallel job.

Related articles

- Set up billing
- Manage paid access
- Buy access to test hub
- Add user for billing management
- Azure DevOps billing overview

Set pipeline permissions

Article • 01/04/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Pipeline permissions and roles help you securely manage your pipelines. You can set hierarchical permissions at the organization, project, and object levels for all pipelines in a project or for an individual pipeline. You can update pipeline permissions with security groups or by adding individual users.

Object-level permissions are more granular than organization-level permissions, so you can increase the security of your pipeline. For example, a user could have access to your Azure Repos repository thanks to their organization-level permissions. However, that same user could be prevented from running a pipeline manually because of that object/pipeline's permissions.

In this article, we break down the permissions to the following levels of permissions:

- Project-level permissions
- Object-level permissions:
  - Release
  - Task group
  - Agent pool
  - Library
  - Service connection
  - Deployment pool
  - Environment

For more information, see Get started with permissions, access, and security groups, Securing Azure Pipelines, and Verify permissions for contributors.

Prerequisites

- To manage permissions and add users to Azure Pipelines for project-level groups, you must be a **Project Administrator**. For more information, see Project-level group permissions.
- To manage permissions for collection groups, you must be a **Project Collection Administrator**. For more information, see collection-level group permissions.
- Keep the following information in mind when you're setting pipeline permissions.

- In many cases, you might want to set **Delete build pipeline** to *Allow*. Otherwise, these team members can't delete their own build pipelines.
- Without the **Delete builds** permission, users can't delete their own completed builds. However, they can automatically delete old unneeded builds with retention policies.
- We recommend that you don't grant permissions directly to a user. A better practice is to add the user to the build administrator group or another group, and manage permissions for that group.

For more information and best practices, see Securing Azure Pipelines.

Set project-level pipeline permissions

Do the following steps to set project-level permissions for all pipelines.

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`).

2. From within your project, select **Pipelines** > **Pipelines**.

3. Select **More actions** ⋮ > **Manage security**.

4. Modify the permissions associated with an Azure DevOps group, for example, Build Administrators, or for an individual user.

5. Select **Allow** or **Deny** for the permission for a security group or an individual user, and then exit the screen.

Your project-level pipelines permissions are set.

Set individual pipeline permissions

Do the following steps to set permissions for an individual pipeline.

1. From within your project, select **Pipelines** > **Pipelines**.

2. Select an individual pipeline, and then select **More actions** ⋮ > **Manage security**.

3. Set permissions, and then **Save** your changes.

Pipeline permissions reference

You can set the following permissions for all pipelines in a project or for an individual pipeline. Default values are set for project collections and project groups. For example, Project Collection Administrators, Project Administrators, and Build Administrators have all these permissions by default.

Permission	Description
Administer build permissions	Can change any of the other permissions listed here.
Delete build pipeline	Can delete build pipeline(s).
Delete builds	Can delete builds for a pipeline. Deleted builds are retained in the **Deleted** tab for a period before they're destroyed.
Destroy builds	Can delete builds from the **Deleted** tab.
Edit build pipeline	Can create pipelines and save any changes to a build pipeline, including configuration variables, triggers, repositories, and retention policy.
Edit build quality	Can add tags to a build.

Permission	Description
Manage build qualities	*Only applies to XAML builds*
Manage build queue	*Only applies to XAML builds*
Override check-in validation by build	Applies to TFVC gated check-in builds. Doesn't apply to pull request builds.
Queue builds	Can queue new builds.
Retain indefinitely	Can toggle the retain indefinitely flag on a build.
Stop builds	Can stop builds queued by other team members or by the system.
Update build information	It is recommended to leave this alone. It's intended to enable service accounts, not team members.
View build pipeline	Can view build pipeline(s).
View builds	Can view builds belonging to build pipeline(s).

All team members are members of the **Contributors** group. This group permission allows you to define and manage builds and releases. The most common built-in groups include Readers, Contributors, and Project Administrators.

For more information, see the following articles:

- Default permissions and access quick reference
- About permissions and inheritance
- About security roles.

Set release permissions

Permissions for release pipelines follow a hierarchical model. You can set default permissions at the project-level, and you can override these permissions on an individual release pipeline.

Set all release permissions

Do the following steps to update permissions for all releases.

1. Select the file view.

2. Select the **All pipelines** folder.

3. Select **More actions** and select **Security**.

4. Set permissions, and then **Save** your changes.

Set individual release permissions

Do the following steps to update permissions for an individual release.

1. Select the release you want to modify.

2. Select **More actions** > **Security**.

3. Set permissions, and then **Save** your changes.

Release permissions reference

The following table defines permissions for releases. The scope column explains whether the permission can be set at the project, release pipeline, or stage level.

Permission	Description
Administer release permissions	Can change any of the other permissions listed here. *Scopes*: Project, Release pipeline, Stage
Create releases	Can create new releases. *Scopes*: Project, Release pipeline
Delete release pipeline	Can delete release pipeline(s). *Scopes*: Project, Release pipeline
Delete release stage	Can delete stage(s) in release pipeline(s). *Scopes*: Project, Release pipeline, Stage

Permission	Description
Delete releases	Can delete releases for a pipeline. *Scopes*: Project, Release pipeline
Edit release pipeline	Can save any changes to a release pipeline, including configuration variables, triggers, artifacts, and retention policy as well as configuration within a stage of the release pipeline. To update a specific stage in a release pipeline, the user also needs **Edit release stage** permission. *Scopes*: Project, Release pipeline
Edit release stage	Can edit stage(s) in release pipeline(s). To save the changes to the release pipeline, the user also needs **Edit release pipeline** permission. This permission also controls whether a user can edit the configuration inside the stage of a specific release instance. The user also needs **Manage releases** permission to save the modified release. *Scopes*: Project, Release pipeline, Stage
Manage deployments	Can initiate a deployment of a release to a stage. This permission is only for deployments that are manually initiated by selecting the **Deploy** or **Redeploy** actions in a release. If the condition on a stage is set to any type of automatic deployment, the system automatically initiates deployment without checking the permission of the user that created the release. If the condition is set to start after some stage, manually initiated deployments do not wait for those stages to be successful. *Scopes*: Project, Release pipeline, Stage
Manage release approvers	Can add or edit approvers for stage(s) in release pipeline(s). This permission also controls whether a user can edit the approvers inside the stage of a specific release instance. *Scopes*: Project, Release pipeline, Stage
Manage releases	Can edit the configuration in releases. To edit the configuration of a specific stage in a release instance (including variables marked as `settable at release time`), the user also needs **Edit release stage** permission. *Scopes*: Project, Release pipeline
View release pipeline	Can view release pipeline(s). *Scopes*: Project, Release pipeline
View releases	Can view releases belonging to release pipeline(s). *Scopes*: Project, Release pipeline

Default values for all permissions are set for team project collections and project groups. For example, Project Collection Administrators, Project Administrators, and Release Administrators are given all the previously listed permissions by default. Contributors are given all permissions except *Administer release permissions*. By default, Readers are denied all permissions except *View release pipeline* and *View releases*.

Set task group permissions

Use task groups to combine a sequence of tasks already defined in a pipeline into a single, reusable task.

Task group permissions follow a hierarchical model. You can set default permissions at the project-level, and you can override these permissions on an individual task group pipeline.

Set project-level task group permissions

Do the following steps to update permissions for project-level task groups.

> **Note**
>
> Task groups aren't supported in YAML pipelines, but templates are. For more information, see **YAML schema reference**.

1. From your project, select **Pipelines** > **Task groups**.

2. Select **Security**.

3. Select **Allow** or **Deny** for the permission for a security group or an individual user.

Set pipeline-level task group permissions

Do the following steps to update permissions for pipeline-level task groups.

1. From your project, select **Pipelines** > **Task groups**.

2. Select a task group.

3. Select **More actions** > **Security**.

4. Select **Allow** or **Deny** for the permission for a security group or an individual user.

Task group permissions reference

Permission	Description
Administer task group permissions	Can add and remove users or groups to task group security.
Delete task group	Can delete a task group.
Edit task group	Can create, modify, or delete a task group.

Set agent pool permissions

You can use predefined roles to configure security on agent pools in a hierarchical manner for all pools, or for an individual pool. Do the following steps to set agent pool permissions for all pools.

Set all agent pool permissions

Do the following steps to update permissions for all agent pools.

1. From your project, select **Project settings** ⚙ > **Agent pools**.

2. Select **Security**.

3. Set permissions, and then **Save** your changes.

Set individual agent pool permissions

Do the following steps to set permissions for an individual agent pool.

1. From the agent pool, select the agent.

2. Select **Security**.

3. Set permissions, and then **Save** your changes.

Set library permissions

Use a variable group to store values that you want to make available across multiple build and release pipelines. Define roles to help you configure security on shared library entities. You can also configure the inheritance of roles.

Do the following steps to manage permissions for library artifacts, such as variable groups and secure files.

1. From your project, select **Pipelines** > **Library**.

2. Select **Security**.

3. Set permissions for everything in your library or for an individual variable group or secure file, and then **Save** your changes.

Library permissions reference

Role	Description
Administrator	Can edit/delete and manage security for library items.
Creator	Can create library items.
Reader	Can only read library items.
User	Can consume library items in pipelines.

Set service connection permissions

Configure permissions for all service connections or for an individual connection.

Set all service connection permissions

Do the following steps to configure permissions for all service connections.

1. From within your project, select **Project settings** .
2. Select **Service connections** under Pipelines.
3. Set permissions, and then **Save** your changes.

Set individual service connection permissions

Do the following steps to configure permissions for an individual service connection.

1. From within your project, open a service connection.

2. Select **More actions** > **Security**.

3. Set permissions, and then **Save** your changes.

If you're having trouble with permissions and service connections, see Troubleshoot Azure Resource Manager service connections.

Service connection permissions reference

Role	Description
User	Can use the endpoint when authoring build or release pipelines.
Administrator	Can manage membership of all other roles for the service connection as well as use the endpoint to author build or release pipelines. The system automatically adds the user that created the service connection to the Administrator role for that pool.

Set deployment pool permissions

You can set default security roles for all deployment groups and manage security roles for an individual deployment group.

Set all deployment pool permissions

1. From within your project, select **Project settings** .

2. Select **Deployment groups** under Pipelines.

3. Set permissions, and then **Save** your changes.

Set individual deployment group permissions

1. From within your project, open a deployment pool.

2. Select **More actions** > **Security**.
3. Set permissions, and then **Save** your changes.

Deployment pool permissions reference

Role	Description
Reader	Can only view deployment pools.
Service Account	Can view agents, create sessions, and listen for jobs from the agent pool.
User	Can view and use the deployment pool for creating deployment groups.
Administrator	Can administer, manage, view and use deployment pools.

Set environment permissions

You can use roles and user permissions to control who can create, view, and manage environments. Apply a hierarchy of roles to all environments or one environment.

Set all environment permissions

1. From within your project, select **Project settings** .

2. Select **Environments** under Pipelines.

3. Set permissions, and then **Save** your changes.

Set individual environment permissions

Select **Security** from **More actions** ⋮ to change permissions for all environment.

1. From within your project, open an environment.
2. Select **More actions** ⋮ > **Security**.
3. Set permissions, and then **Save** your changes.

Environment permissions reference

> **Important**
>
> When you create an environment in a YAML, contributors and project administrators have the administrator role. When you create an environment through the UI, only the creator has the administrator role.

Role	Description
Creator	Global role, available from environments hub security option. Members of this role can create the environment in the project. Contributors are added as members by default. Required to trigger a YAML pipeline when the environment does not already exist.
Reader	Members of this role can view the environment.
User	Members of this role can use the environment when creating or editing YAML pipelines.

Role	Description
Administrator	In addition to using the environment, members of this role can manage membership of all other roles for the environment. Creators are added as members by default.

FAQs

See the following frequently asked questions (FAQs) about pipeline permissions.

Q: Why can't I create a new pipeline?

A: You need **Edit build pipeline** permissions to create a new pipeline. To add permission, open the security settings for all pipelines and verify that **Edit build pipeline** is set to *Allow* for your security group.

If you still can't create a pipeline, check to see if your access level is set to **Stakeholder**. If you have stakeholder access, change your access to **Basic**.

Q: Why do I see the message that I need to authorize a resource before the run can continue?

A: You need to authorize resources before you can use them. The exception to this rule is when you create a pipeline for the first time and all the resources referenced in the YAML file are automatically authorized. The resources are authorized for the pipeline as long as the user running the pipeline has access to the resource.

To authorize **All pipelines** to access a resource like an agent pool, do the following steps.

1. From your project, select **Settings** ⚙ > **Pipelines** > **Agent Pools**.
2. Select **Security** for a specific agent pool, and then update permissions to grant access to all pipelines.

For more information, see Resources in YAML.

Related articles

- Set build and release permissions
- Default permissions and access
- Permissions and groups reference
- Troubleshoot Azure Resource Manager service connections

Add users to Azure Pipelines

Article • 01/04/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Permissions for build and release pipelines are primarily set at the object-level for a specific build or release, or for select tasks, at the collection level.

You can manage security for different types of resources such as variable groups, secure files, and deployment groups by adding users or groups to that role. Project administrator can grant or restrict access to project resources. If you want to allow a team member to edit pipelines, you must be a project administrator in order to do so.

Add users to your project

1. Navigate to your project's summary page: `https://dev.azure.com/{your-organization}/{your-project}`

2. Select the **Invite** button to add a user to your project, and then fill out the required fields. Select **Add** when you are done.

3. The new user must accept the invitation before they can start creating or modifying pipelines.

Verify permissions for contributors

> **Note**
>
> A security best practice is to only grant permissions to required users or groups. The **Contributors** group may be too broad in a given project.

To verify the permissions for your project's contributors, make sure you are a member of the **Build Administrators** group or the **Project Administrators** group. See Change project-level permissions for more details.

1. From within your project, select **Pipelines** > **Pipelines**. Select the **All** tab, and then select the more actions menu then **Manage security**.

2. On the permissions dialog box, make sure the following **Contributors** permissions are set to Allow.

Related articles

- Grant version control permissions to the build service
- Set pipelines permissions
- Set retention policies for builds, releases, and tests
- Default permissions and access
- Permissions and groups reference

Run Git commands in a script

Article • 07/03/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

For some workflows, you need your build pipeline to run Git commands. For example, after a CI build on a feature branch is done, the team might want to merge the branch to main.

Git is available on Microsoft-hosted agents and on on-premises agents.

Enable scripts to run Git commands

> **Note**
>
> Before you begin, be sure your account's default identity is set with the following code. This must be done as the very first step after checking out your code.
>
> ```
> git config --global user.email "you@example.com"
> git config --global user.name "Your Name"
> ```

Grant version control permissions to the build service

1. Go to the project settings page for your organization at **Organization Settings** > **General** > **Projects**.

2. Select the project you want to edit.

3. Within **Project Settings**, select **Repositories**. Select the repository you want to run Git commands on.

4. Select **Security** to edit your repository security.

5. Search for **Project Collection Build Service**. Choose the identity **{{your project name}} Build Service ({your organization})** (not the group **Project Collection Build Service Accounts ({your organization})**). By default, this identity can read from the repo but can't push any changes back to it. Grant permissions needed for the Git commands you want to run. Typically you'll want to grant:

   - **Create branch:** Allow
   - **Contribute:** Allow
   - **Read:** Allow
   - **Create tag:** Allow

Allow scripts to access the system token

YAML

Add a `checkout` section with `persistCredentials` set to `true`.

YAML

```yaml
steps:
- checkout: self
  persistCredentials: true
```

Learn more about checkout.

Make sure to clean up the local repo

Certain kinds of changes to the local repository aren't automatically cleaned up by the build pipeline. So make sure to:

- Delete local branches you create.
- Undo git config changes.

If you run into problems using an on-premises agent, make sure the repo is clean:

YAML

Make sure `checkout` has `clean` set to `true`.

YAML

```yaml
steps:
- checkout: self
  clean: true
```

Examples

List the files in your repo

On the build tab, add this task:

Task	Arguments
Utility: Command Line List the files in the Git repo.	Tool: `git` Arguments: `ls-files`

Merge a feature branch to main

You want a CI build to merge to main if the build succeeds.

On the Triggers tab, select **Continuous integration (CI)** and include the branches you want to build.

Create `merge.bat` at the root of your repo:

bat

```bat
@echo off
ECHO SOURCE BRANCH IS %BUILD_SOURCEBRANCH%
IF %BUILD_SOURCEBRANCH% == refs/heads/main (
   ECHO Building main branch so no merge is needed.
   EXIT
)
SET sourceBranch=origin/%BUILD_SOURCEBRANCH:refs/heads/=%
ECHO GIT CHECKOUT MAIN
git checkout main
ECHO GIT STATUS
git status
ECHO GIT MERGE
git merge %sourceBranch% -m "Merge to main"
ECHO GIT STATUS
git status
ECHO GIT PUSH
git push origin
ECHO GIT STATUS
git status
```

On the build tab add this as the last task:

Task	Arguments
Utility: Batch Script Run merge.bat.	Path: `merge.bat`

FAQ

Can I run Git commands if my remote repo is in GitHub or another Git service such as Bitbucket Cloud?

Yes

Which tasks can I use to run Git commands?

Batch Script

Command Line

PowerShell

Shell Script

How do I avoid triggering a CI build when the script pushes?

Add `[skip ci]` to your commit message or description. Here are examples:

- `git commit -m "This is a commit message [skip ci]"`
- `git merge origin/features/hello-world -m "Merge to main [skip ci]"`

You can also use any of these variations for commits to Azure Repos Git, Bitbucket Cloud, GitHub, and GitHub Enterprise Server.

- `[skip ci]` or `[ci skip]`
- `skip-checks: true` or `skip-checks:true`
- `[skip azurepipelines]` or `[azurepipelines skip]`
- `[skip azpipelines]` or `[azpipelines skip]`
- `[skip azp]` or `[azp skip]`
- `***NO_CI***`

Do I need an agent?

You need at least one agent to run your build or release.

I'm having problems. How can I troubleshoot them?

See Troubleshoot Build and Release.

I can't select a default agent pool and I can't queue my build or release. How do I fix this?

See Agent pools.

My NuGet push task is failing with the following error: "Error: unable to get local issuer certificate". How can I fix this?

This can be fixed by adding a trusted root certificate. You can either add the `NODE_EXTRA_CA_CERTS=file` environment variable to your build agent, or you can add the `NODE.EXTRA.CA.CERTS=file` task variable in your pipeline. See Node.js documentation

for more details about this variable. See Set variables in a pipeline for instructions on setting a variable in your pipeline.

Securing Azure Pipelines

Article • 10/31/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Azure Pipelines poses unique security challenges. You can use a pipeline to run scripts or deploy code to production environments. But you want to ensure your CI/CD pipelines don't become avenues to run malicious code. You also want to ensure only code you intend to deploy is deployed. Security must be balanced with giving teams the flexibility and power they need to run their own pipelines.

> **Note**
>
> Azure Pipelines is one among a collection of Azure DevOps Services, all built on the same secure infrastructure in Azure. To understand the main concepts around security for all of Azure DevOps Services, see **Azure DevOps Data Protection Overview** and **Azure DevOps Security and Identity**.

Traditionally, organizations implemented security through draconian lock-downs. Code, pipelines, and production environments had severe restrictions on access and use. In small organizations with a few users and projects, this stance was relatively easy to manage. However, that's not the case in larger organizations. Where many users have contributor access to code, one must "assume breach". Assuming breach means behaving as if an adversary has contributor access to some (if not all) of the repositories.

The goal in this case is to prevent that adversary from running malicious code in the pipeline. Malicious code may steal secrets or corrupt production environments. Another goal is to prevent lateral exposure to other projects, pipelines, and repositories from the compromised pipeline.

YAML pipelines offer the best security for your Azure Pipelines. In contrast to classic build and release pipelines, YAML pipelines:

- *Can be code reviewed*. YAML pipelines are no different from any other piece of code. You can prevent malicious actors from introducing malicious steps in your pipelines by enforcing the use of Pull Requests to merge changes. Branch policies make it easy for you to set this up.
- *Provide resource access management*. Resource owners decide if a YAML pipeline can access a resource or not. This security feature control attacks like stealing another repository. Approvals and checks provide access control for *each* pipeline run.

- *Support runtime parameters*. Runtime parameters help you avoid a host of security issues related to variables, such as Argument Injection.

This series of articles outlines recommendations to help you put together a secure YAML-based CI/CD pipeline. It also covers the places where you can make trade-offs between security and flexibility. The series also assumes familiarity with Azure Pipelines, the core Azure DevOps security constructs, and Git.

Topics covered:

- Incremental approach to improving security
- Repository protection
- Secure access to repositories
- Using secrets
- Pipeline resources
- Project structure
- Security through templates
- Variables and parameters
- Shared infrastructure
- Other security considerations

Plan how to secure your YAML pipelines

Article • 11/27/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

We recommend that you use an incremental approach to secure your pipelines. Ideally, you would implement all of the guidance that we offer. But don't be daunted by the number of recommendations. And don't hold off making *some* improvements just because you can't make all the changes right now.

Security recommendations depend on each other

Security recommendations have complex interdependencies. Your security posture depends heavily on which recommendations you choose to implement. The recommendations that you choose, in turn, depend on the concerns of your DevOps and security teams. They also depend on the policies and practices of your organization.

You might choose to tighten security in one critical area and accept less security but more convenience in another area. For example, if you use extends templates to require all builds to run in containers, then you might not need a separate agent pool for each project.

Begin with a nearly empty template

A good place to start is by enforcing extension from a nearly empty template. This way, as you start to apply security practices, you have a centralized place that already catches every pipeline.

For more information, see Templates.

Disable creation of classic pipelines

If you develop *only* YAML pipelines, disable creation of classic build and release pipelines. Doing so prevents a security concern that stems from YAML and classic pipelines sharing the same resources, for example the same service connections.

You can disable the creation of classic build pipelines and classic release pipelines independently. When you disable both, no classic build pipeline, classic release pipeline,

task groups, and deployment groups can be created using either the user interface or the REST API.

You can disable creation of classic pipelines by turning on two toggles at either organization level or project level. To turn them on, navigate to your *Organization / Project settings*, then under the *Pipelines* section choose *Settings*. In the *General* section, toggle on *Disable creation of classic build pipelines* and *Disable creation of classic release pipelines*.

When you turn them on at organization level, it is on for all projects in that organization. If you leave them off, you can choose for which projects you wish to turn them on.

To improve the security of newly created organizations, starting with Sprint 226, by default we will disable creating classic build and release pipelines for new organizations.

Next steps

After you plan your security approach, consider how your repositories provide protection.

Repository protection

Article • 11/27/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Source code, the pipeline's YAML file, and necessary scripts & tools are all stored in a version control repository. Permissions and branch policies must be employed to ensure changes to the code and pipeline are safe. You can also add pipeline permissions and checks to repositories.

Also, you should review default access control for repositories.

Because of Git's design, protection at a branch level will only carry you so far. Users with push access to a repo can usually create new branches. If you use GitHub open-source projects, anyone with a GitHub account can fork your repository and propose contributions back. Since pipelines are associated with a repository and not with specific branches, you must assume the code and YAML files are untrusted.

Forks

If you build public repositories from GitHub, you must consider your stance on fork builds. Forks are especially dangerous since they come from outside your organization. To protect your products from contributed code, consider the following recommendations.

> **Note**
>
> The following recommendations apply primarily to building public repos from GitHub.

Don't provide secrets to fork builds

By default, your pipelines are configured to build forks, but secrets and protected resources aren't made available to the jobs in those pipelines by default. Don't turn off this latter protection.

> **Note**
>
> When you enable fork builds to access secrets, Azure Pipelines by default restricts the access token used for fork builds. It has more limited access to open resources than a normal access token. To give fork builds the same permissions as regular builds, enable the **Make fork builds have the same permissions as regular builds** setting.

Consider manually triggering fork builds

You can turn off automatic fork builds and instead use pull request comments as a way to manually building these contributions. This setting will give you an opportunity to review the code before triggering a build.

Use Microsoft-hosted agents for fork builds

Don't run builds from forks on self-hosted agents. By doing so, you're effectively providing a path to external organizations to run outside code on machines inside your corporate network. Use Microsoft-hosted agents whenever possible. For your self-hosted agent, use some form of network isolation and ensure agents don't persist their state between jobs.

Review code changes

Before you run your pipeline on a forked pull-request, carefully review the proposed changes, and make sure you're comfortable running it.

The version of the YAML pipeline you'll run is the one from the pull request. Thus, pay special attention to changes to the YAML code and to the code that runs when the pipeline runs, such as command line scripts or unit tests.

GitHub token scope limitation

When you build a GitHub forked pull request, Azure Pipelines ensures the pipeline can't change any GitHub repository content. This restriction applies *only* if you use the Azure Pipelines GitHub app to integrate with GitHub. If you use other forms of GitHub integration, for example, the OAuth app, the restriction isn't enforced.

User branches

Users in your organization with the right permissions can create new branches containing new or updated code. That code can run through the same pipeline as your protected branches. Further, if the YAML file in the new branch is changed, then the updated YAML will be used to run the pipeline. While this design allows for great flexibility and self-service, not all changes are safe (whether made maliciously or not).

If your pipeline consumes source code or is defined in Azure Repos, you must fully understand the Azure Repos permissions model. In particular, a user with **Create Branch** permission at the repository level can introduce code to the repo even if that user lacks **Contribute** permission.

Next steps

Next, learn about the more protection offered by checks on protected resources.

Secure access to Azure Repos from pipelines

Article • 08/04/2022

Your repositories are a critical resource to your business success, because they contain the code that powers your business. Access to repositories shouldn't be granted easily.

This article shows you how to improve the security of your pipelines accessing Azure Repos, to limit the risk of your source code getting into the wrong hands.

The setup for pipelines to securely access Azure repositories is one in which the toggles *Limit job authorization scope to current project for non-release pipelines*, *Limit job authorization scope to current project for release pipelines*, and *Protect access to repositories in YAML pipelines*, are enabled.

We'll cover both build pipelines and classic release pipelines:

- Build pipelines
- Classic release pipelines

Basic process

The steps are similar across all pipelines:

1. Determine the list of Azure Repos repositories your pipeline needs access to that are part of the same organization, but are in different projects.

   You can compile the list of repositories by inspecting your pipeline. Or, you can turn on the *Limit job authorization scope to current project for (non-)release pipelines* toggle and note which repositories your pipeline fails to check out. Submodule repositories may not show up in the first failed run.

2. For each Azure DevOps project that contains a repository your pipeline needs to access, follow the steps to grant the pipeline's build identity access to that project.

3. For each Azure Repos repository your pipeline checks out, follow the steps to grant the pipeline's build identity *Read* access to that repository.

4. For each repository that is used as a submodule by a repository your pipeline checks out and is in the same project, follow the steps to grant the pipeline's build identity *Read* access to that repository.

5. Turn on the *Limit job authorization scope to current project for non-release pipelines*, *Limit job authorization scope to current project for release pipelines*, and *Protect access to repositories in YAML pipelines* toggles.

Build pipelines

To illustrate the steps to take to improve the security of your pipelines when they access Azure Repos, we'll use a running example.

Assume you're working on the `SpaceGameWeb` pipeline hosted in the `fabrikam-tailspin/SpaceGameWeb` project, in the `SpaceGameWeb` Azure Repos repository. Furthermore, let's say your `SpaceGameWeb` pipeline checks out the `SpaceGameWebReact` repository in the same project, and the `FabrikamFiber` and `FabrikamChat` repositories in the `fabrikam-tailspin/FabrikamFiber` project.

Finally, assume the `FabrikamFiber` repository uses the `FabrikamFiberLib` repository as a submodule, hosted in the same project. Read more about how to check out submodules.

The `SpaceGameWeb` project's repository structures look like in the following screenshot.

The `FabrikamFiber` project's repository structures look like in the following screenshot.

Image your project isn't set up to use a project-based build identity or to protect access to repositories in YAML pipelines. Also, assume you've already successfully ran your pipeline.

Use a Project-based build identity for build pipelines

When a pipeline executes, it uses an identity to access various resources, such as repositories, service connections, variable groups. There are two types of identities a pipeline can use: a project-level one and a collection-level one. The former provides better security, the latter provides ease of use. Read more about scoped build identities and job authorization scope.

We recommend you use project-level identities for running your pipelines. By default, project-level identities can only access resources in the project of which they're a member. Using this identity improves security, because it reduces the access gained by a malicious person when hijacking your pipeline.

To make your pipeline use a project-level identity, turn on the *Limit job authorization scope to current project for non-release pipelines* setting.

In our running example, when this toggle is off, the `SpaceGameWeb` pipeline can access all repositories in all projects. When the toggle is on, `SpaceGameWeb` can only access resources in the `fabrikam-tailspin/SpaceGameWeb` project, so only the `SpaceGameWeb` and `SpaceGameWebReact` repositories.

If you run our example pipeline, when you turn on the toggle, the pipeline will fail, and the error logs will tell you `remote: TF401019: The Git repository with name or identifier FabrikamChat does not exist or you do not have permissions for the`

`operation you are attempting.` and `remote: TF401019: The Git repository with name or identifier FabrikamFiber does not exist or you do not have permissions for the operation you are attempting.`

To fix the checkout issues, follow the steps described in Basic process.

Additionally, you need to explicitly check out the submodule repositories, *before* the repositories that use them. In our example, it means the `FabrikamFiberLib` repository.

If you now run our example pipeline, it will succeed.

Further configuration

To further improve security when accessing Azure Repos, consider turning on the *Protect access to repositories in YAML pipelines* setting.

YAML pipelines

Assume the `SpaceGameWeb` pipeline is a YAML pipeline, and its YAML source code looks similar to the following code.

yml

```yml
trigger:
- main

pool:
  vmImage: ubuntu-latest

resources:
  repositories:
    - repository: SpaceGameWebReact
      name: SpaceGameWeb/SpaceGameWebReact
      type: git
    - repository: FabrikamFiber
      name: FabrikamFiber/FabrikamFiber
      type: git
    - repository: FabrikamChat
      name: FabrikamFiber/FabrikamChat
      type: git

steps:
  - script: echo "Building SpaceGameWeb"
  - checkout: SpaceGameWebReact
  - checkout: FabrikamChat
    condition: always()
  - checkout: FabrikamFiber
    submodules: true
```

```
    condition: always()
  - script: |
      cd FabrikamFiber
      git -c http.extraheader="AUTHORIZATION: bearer
$(System.AccessToken)" submodule update --recursive --remote
  - script: cat
$(Build.Repository.LocalPath)/FabrikamFiber/FabrikamFiberLib/README.md
  - ...
```

Protect access to repositories in YAML pipelines

Azure DevOps provides a fine-grained permissions mechanism for Azure Repos repositories, in the form of the *Protect access to repositories in YAML pipelines* setting. This setting makes a YAML pipeline explicitly ask for permission to access *all* Azure Repos repositories, regardless of which project they belong to. Read more about this setting. Checking out other types of repositories, for example, GitHub-hosted ones, isn't affected by this setting.

In our running example, when this toggle is on, the `SpaceGameWeb` pipeline will ask permission to access the `SpaceGameWebReact` repository in the `fabrikam-tailspin/SpaceGameWeb` project, and the `FabrikamFiber` and `FabrikamChat` repositories in the `fabrikam-tailspin/FabrikamFiber` project.

When you run the example pipeline, you'll see a build similar to the following screenshot.

You'll be asked to grant permission to the repositories your pipeline checks out or has defined as resources.

Once you do, your pipeline will run, but it will fail because it will not be able to check out the `FabrikamFiberLib` repository as a submodule of `FabrikamFiber`. To solve this issue, explicitly check out the `FabrikamFiberLib`, for example, add a `-checkout: git://FabrikamFiber/FabrikamFiberLib` step, before the `-checkout: FabrikamFiber` step.

If you now run the example pipeline, it will succeed.

Our final YAML pipeline source code looks like the following code snippet.

yml

```yml
trigger:
- main

pool:
  vmImage: ubuntu-latest

resources:
  repositories:
    - repository: SpaceGameWebReact
      name: SpaceGameWeb/SpaceGameWebReact
      type: git
    - repository: FabrikamFiber
      name: FabrikamFiber/FabrikamFiber
      type: git
    - repository: FabrikamChat
      name: FabrikamFiber/FabrikamChat
      type: git

steps:
  - script: echo "Building SpaceGameWeb"
  - checkout: SpaceGameWebReact
  - checkout: FabrikamChat
    condition: always()
```

```
  - checkout: git://FabrikamFiber/FabrikamFiberLib
  - checkout: FabrikamFiber
    submodules: true
    condition: always()
  - script: |
      cd FabrikamFiber
      git -c http.extraheader="AUTHORIZATION: bearer
$(System.AccessToken)" submodule update --recursive --remote
  - script: cat
$(Build.Repository.LocalPath)/FabrikamFiber/FabrikamFiberLib/README.md
```

Troubleshooting

Here are a couple of problematic situations and how to handle them.

You use git in command line to check out repositories in the same organization

For example, you're using `- script: git clone https://$(System.AccessToken)@dev.azure.com/fabrikam-tailspin/FabrikamFiber/_git/OtherRepo/`. The command will fail when the *Protect access to repositories in YAML pipelines* toggle is on.

To solve the issue, check out the `OtherRepo` repository using the `checkout` command, for example, `- checkout: git://FabrikamFiber/OtherRepo`.

A repository is using another repository as submodule

Say one of the repositories your pipeline checks out uses another repository (in the same project) as submodule, as is the case in our example for the `FabrikamFiber` and `FabrikamFiberLib` repositories. Read more about how to check out submodules.

Furthermore, assume you gave the `SpaceGame` build identity *Read* access to this repo, but the checkout of the `FabrikamFiber` repository still fails when checking out the `FabrikamFiberLib` submodule.

To solve this issue, explicitly check out the `FabrikamFiberLib`, for example, add a `- checkout: git://FabrikamFiber/FabrikamFiberLib` step before the `-checkout: FabrikamFiber` one.

Classic release pipelines

The process for securing access to repositories for release pipelines is similar to the one for build pipelines.

To illustrate the steps you need to take, we'll use a running example. In our example, there's a release pipeline named `FabrikamFiberDocRelease` in the `fabrikam-tailspin/FabrikamFiberDocRelease` project. Assume the pipeline checks out the `FabrikamFiber` repository in the `fabrikam-tailspin/FabrikamFiber` project, runs a command to generate public documentation, and then publishes it to a website. Additionally, imagine the `FabrikamFiber` repository uses the `FabrikamFiberLib` repository (in the same project) as a submodule

Use a Project-based build identity for classic release pipelines

When a pipeline executes, it uses an identity to access various resources, such as repositories, service connections, variable groups. There are two types of identities a pipeline can use: a project-level one and a collection-level one. The former provides better security, the latter provides ease of use. Read more about scoped build identities and job authorization scope.

We recommend you use project-level identities for running your pipelines. By default, project-level identities can only access resources in the project of which they're a member. Using this identity improves security, because it reduces the access gained by a malicious person when hijacking your pipeline.

To make your pipeline use a project-level identity, turn on the *Limit job authorization scope to current project for release pipelines* setting.

In our running example, when this toggle is off, the `FabrikamFiberDocRelease` release pipeline can access all repositories in all projects, including the `FabrikamFiber` repository. When the toggle is on, `FabrikamFiberDocRelease` can only access resources in the `fabrikam-tailspin/FabrikamFiberDocRelease` project, so the `FabrikamFiber` repository becomes inaccessible.

If you run our example pipeline, when you turn on the toggle, the pipeline will fail, and the logs will tell you `remote: TF401019: The Git repository with name or identifier FabrikamFiber does not exist or you do not have permissions for the operation you are attempting.`

To fix these issues, follow the steps in Basic process.

If you now run our example pipeline, it will succeed.

See also

- Scoped build identities
- Job authorization scope
- Grant a pipeline's build identity access to a project
- Grant a pipeline's build identity *Read* access to a repository
- How to check out submodules

Protecting secrets in Azure Pipelines

Article • 11/13/2023

This article provides best practices on protecting secrets in Azure Pipelines.

Azure Pipelines doesn't generate secret values. However, you might need to add a secret to a pipeline to store sensitive data like an API key. To learn more about setting secret variables, see Set secret variables.

Don't use secrets if another method is available

The best method to protect a secret is not to have a secret in the first place. Check to see if your pipeline can use a different method than using a secret to perform a task.

- Use service connections. For example, if you are targeting Azure or another service that uses a service connection, you should use the service connection instead of managing secrets in variables. For more information, see Manage service connections and Connect to Microsoft Azure with an ARM service connection.
- Prefer managed identities instead of using secrets.
  - Azure services that can use managed identities to access other services
  - What are managed identities for Azure resources?
  - If you are using the Azure CLI task, you can use the `addSpnToEnvironment` setting to access service principal details in script.
  - Use service principals & managed identities

Use secret variables

Sensitive values should never be stored as plaintext in an Azure Pipelines **.yml** file.

Secret variables can be used for private information like passwords, IDs, and other identifying data that you wouldn't want exposed in a pipeline. The recommended way to set secret variables is with Azure Key Vault. You can also set secret variables in the UI, in a variable group, and in a script with a logging command. Using a logging command is not recommended since anyone who can access your pipeline will be able to also see the secret.

Secret variables are encrypted and can be used in pipelines without exposing their values. Although their values aren't exposed, never echo secrets as output and don't pass secrets on the command line. Instead, we suggest that you map your secrets into environment variables.

When you create a secret, follow variable naming guidelines and make sure that your secret name does not disclose sensitive information.

Limit access to secret variables

To limit access to secrets in Azure DevOps, you can:

- Store your secrets in Azure Key Vault. With Azure Key Vault, you can then use Azure's role-based access control model to limit access to a secret or group of secrets.
- Set secret variables in the UI for a pipeline. Secret variables set in the pipeline settings UI for a pipeline are scoped to the pipeline where they are set. So, you can have secrets that only visible to users with access to that pipeline.
- Set secrets in a variable group. Variable groups follow the library security model. You can control who can define new items in a library, and who can use an existing item.

Don't write secrets to logs

Azure Pipelines attempts to scrub secrets from logs wherever possible. This filtering is on a best-effort basis and can't catch every way that secrets can be leaked. Avoid echoing secrets to the console, using them in command line parameters, or logging them to files. For example, some Azure CLI commands output information you must protect. If you call Azure CLI from your pipeline, use the None output format, and if you need to retrieve a secret from an Azure CLI call, Use none output format and retrieve security information to a secret variable.

Don't use structured data as secrets

Structured data can cause secret redaction within logs to fail, because redaction largely relies on finding an exact match for the specific secret value. For example, don't use a blob of JSON, XML, or YAML (or similar) to encapsulate a secret value, including control characters such as carriage return (`\r`) and line feed (`\n`), as this significantly reduces the probability the secrets are properly redacted. Instead, create individual secrets for each sensitive value.

Audit how secrets are handled

Audit how secrets are used, to help ensure they’re being handled as expected. You can do this by reviewing the source code of the repository hosting the pipeline, and

checking any tasks used in the pipeline. For example, check that they're not sent to unintended hosts, or explicitly being printed to log output.

View the run logs for your pipeline after testing valid/invalid inputs, and check that secrets are properly redacted, or not shown. It's not always obvious how a command or tool you're invoking will emit errors, and secrets might later end up in error logs. Azure Pipelines attempts to scrub secrets from logs wherever possible. This filtering is on a best-effort basis and can't catch every way that secrets can be leaked. As a result, it's good practice to manually review the pipeline logs after testing valid and invalid inputs.

Audit and rotate secrets

Periodically review the registered secrets used by your pipelines to confirm they're still required, and remove those that are no longer needed.

Periodically rotate secrets to reduce the window of time during which a compromised secret is valid.

Types of secrets used by pipelines can include:

- Personal access tokens
  - Security best practices - Choose the right authentication method
  - Manage personal access tokens (PATs) using REST API
- Secret variables
- Azure Key Vault secrets
- Service connections

Next steps

- Best practices for protecting Azure secrets
- Key and secret management considerations in Azure
- Azure DevOps security best practices

Pipeline resources

Article • 06/16/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Azure Pipelines offers security beyond just protecting the YAML file and source code. When YAML pipelines run, access to resources goes through a system called checks. Checks can suspend or even fail a pipeline run in order to keep resources safe. A pipeline can access two types of resources, protected and open.

Protected resources

Your pipelines often have access to secrets. For instance, to sign your build, you need a signing certificate. To deploy to a production environment, you need a credential to that environment. Azure Pipelines requires the **Administrator** role when opening up access to a resource to all pipelines for all protected resources except for Environments. For Environments, you'll need the **Creator** role. Learn more about resource protection.
In Azure Pipelines, all of the following are considered *protected* resources in YAML pipelines:

- Agent pools
- Secret variables in variable groups
- Secure files
- Service connections
- Environments
- Repositories

"Protected" means:

- They can be made accessible to specific users and specific pipelines within the project. They can't be accessed by users and pipelines outside of a project.
- You can run other manual or automated checks every time a YAML pipeline uses one of these resources. To learn more about protected resources, see About pipeline resources.

Protecting repository resources

Repositories can optionally be protected. At the organization or project level, you may choose to limit the scope of the Azure Pipelines access token to mentioned repositories. When you do this, Azure Pipelines will add two more protections:

- The access token given to the agent for running jobs will only have access to repositories explicitly mentioned in the `resources` section of the pipeline.
- Repositories added to the pipeline will have to be authorized by someone with contribute access to the repository the first time that pipeline uses the repository.

This setting is on by default for all organizations created after May 2020. Organizations created before that should enable it in **Organization settings**.

Open resources

All the other resources in a project are considered *open* resources. Open resources include:

- Artifacts
- Pipelines
- Test plans
- Work items

You'll learn more about which pipelines can access what resources in the section on projects.

User permissions

The first line of defense for protected resources is user permissions. In general, ensure that you only give permissions to users who require them. All protected resources have a similar security model. A member of user role for a resource can:

- Remove approvers and checks configured on that resource
- Grant access to other users or pipelines to use that resource

Pipeline permissions

When you use YAML pipelines, user permissions aren't enough to secure your protected resources. You can easily copy the name of a protected resource (for example, a service connection for your production environment) and include that in a different pipeline. Pipeline permissions protect against such copying. For each of the protected resources, ensure that you've disabled the option to grant access to "all pipelines". Instead, explicitly granted access to specific pipelines that you trust.

Checks

In YAML, a combination of user and pipeline permissions isn't enough to fully secure your protected resources. Pipeline permissions to resources are granted to the whole pipeline. Nothing prevents an adversary from creating another branch in your repository, injecting malicious code, and using the same pipeline to access that resource. Even without malicious intent, most pipelines need a second set of eyes look over changes (especially to the pipeline itself) before deploying to production. **Checks** allow you to pause the pipeline run until certain conditions are met:

- **Manual approval check**. Every run that uses a project protected resource is blocked for your manual approval before proceeding. Manual protection gives you the opportunity to review the code and ensure that it's coming from the right branch.
- **Protected branch check**. If you have manual code review processes in place for some of your branches, you can extend this protection to pipelines. Configure a protected branch check on each of your resources. This will automatically stop your pipeline from running on top of any user branches.
- **Protected resource check** You can add checks to environments, service connections, repositories, variable groups, agent pools, variable groups, and secure

files to specify conditions that must be satisfied before a stage in any pipeline can consume a resource. Learn more about checks and approvals.

Next steps

Next, consider how you group resources into a project structure.

Recommendations to securely structure projects in your pipeline

Article • 01/27/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Beyond the scale of individual resources, you should also consider groups of resources. In Azure DevOps, resources are grouped by team projects. It's important to understand what resources your pipeline can access based on project settings and containment.

Every job in your pipeline receives an access token. This token has permissions to read open resources. In some cases, pipelines might also update those resources. In other words, your user account might not have access to a certain resource, but scripts and tasks that run in your pipeline might have access to that resource. The security model in Azure DevOps also allows access to these resources from other projects in the organization. If you choose to shut off pipeline access to some of these resources, then your decision applies to all pipelines in a project. A specific pipeline can't be granted access to an open resource.

Separate projects

Given the nature of open resources, you should consider managing each product and team in a separate project. This practice ensures that a pipeline from one product can't access open resources from another product. In this way, you prevent lateral exposure. When multiple teams or products share a project, you can't granularly isolate their resources from one another.

If your Azure DevOps organization was created before August 2019, then runs might be able to access open resources in all of your organization's projects. Your organization administrator must review a key security setting in Azure Pipelines that enables project isolation for pipelines. You can find this setting at **Azure DevOps** > **Organization settings** > **Pipelines** > **Settings**. Or go directly to this Azure DevOps location: *https://dev.azure.com/ORG-NAME/_settings/pipelinessettings*.

Next steps

After you've set up the right project structure, enhance runtime security by using templates.

Security through templates

Article • 01/24/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Checks on protected resources are the basic building block of security for Azure Pipelines. Checks work no matter the structure - the stages and jobs - of your pipeline. If several pipelines in your team or organization have the same structure, you can further simplify security using templates.

Azure Pipelines offers two kinds of templates: **includes** and **extends**. Included templates behave like `#include` in C++: it's as if you paste the template's code right into the outer file, which references it. For example, here an includes template (`include-npm-steps.yml`) is inserted into `steps`.

YAML

```yaml
steps:
- template: templates/include-npm-steps.yml
```

To continue the C++ metaphor, `extends` templates are more like inheritance: the template provides the outer structure of the pipeline and a set of places where the template consumer can make targeted alterations.

Use extends templates

For the most secure pipelines, we recommend starting with `extends` templates. By providing the outer structure, a template can prevent malicious code from getting into your pipeline. You can still use `includes`, both in the template and in the final pipeline, to factor out common pieces of configuration. To use an extends template, your pipeline might look like the below example.

YAML

```yaml
# template.yml
parameters:
- name: usersteps
  type: stepList
  default: []
steps:
- ${{ each step in parameters.usersteps }}:
  - ${{ step }}
```

YAML

```yaml
# azure-pipelines.yml
resources:
  repositories:
  - repository: templates
    type: git
    name: MyProject/MyTemplates
    ref: refs/tags/v1

extends:
  template: template.yml@templates
  parameters:
    usersteps:
    - script: echo This is my first step
    - script: echo This is my second step
```

When you set up `extends` templates, consider anchoring them to a particular Git branch or tag. That way, if breaking changes need to be made, existing pipelines won't be affected. The examples above use this feature.

Security features enforced through YAML

There are several protections built into the YAML syntax, and an extends template can enforce the usage of any or all of them.

Step targets

Restrict some steps to run in a container instead of the host. Without access to the agent's host, user steps can't modify agent configuration or leave malicious code for later execution. Run code on the host first to make the container more secure. For instance, we recommend limiting access to network. Without open access to the network, user steps will be unable to access packages from unauthorized sources, or upload code and secrets to a network location.

YAML

```yaml
resources:
  containers:
  - container: builder
    image: mysecurebuildcontainer:latest
steps:
- script: echo This step runs on the agent host, and it could use docker
commands to tear down or limit the container's network
- script: echo This step runs inside the builder container
  target: builder
```

Agent logging command restrictions

Restrict what services the Azure Pipelines agent will provide to user steps. Steps request services using "logging commands" (specially formatted strings printed to stdout). In restricted mode, most of the agent's services such as uploading artifacts and attaching test results are unavailable.

YAML

```yaml
# this task will fail because its `target` property instructs the agent not
to allow publishing artifacts
- task: PublishBuildArtifacts@1
  inputs:
    artifactName: myartifacts
  target:
    commands: restricted
```

One of the commands still allowed in restricted mode is the `setvariable` command. Because pipeline variables are exported as environment variables to subsequent tasks, tasks that output user-provided data (for example, the contents of open issues retrieved from a REST API) can be vulnerable to injection attacks. Such user content can set environment variables that can in turn be used to exploit the agent host. To disallow this, pipeline authors can explicitly declare which variables are settable via the `setvariable` logging command. Specifying an empty list disallows setting all variables.

YAML

```yaml
# this task will fail because the task is only allowed to set the
'expectedVar' variable, or a variable prefixed with "ok"
- task: PowerShell@2
  target:
    commands: restricted
    settableVariables:
    - expectedVar
    - ok*
  inputs:
    targetType: 'inline'
    script: |
      Write-Host "##vso[task.setvariable variable=BadVar]myValue"
```

Conditional insertion of stages or jobs

Restrict stages and jobs to run under specific conditions. Conditions can help, for example, to ensure that you are only building certain branches.

YAML

```yaml
jobs:
- job: buildNormal
  steps:
  - script: echo Building the normal, unsensitive part
- ${{ if eq(variables['Build.SourceBranchName'], 'refs/heads/main') }}:
  - job: buildMainOnly
    steps:
    - script: echo Building the restricted part that only builds for main
branch
```

Require certain syntax with extends templates

Templates can iterate over and alter/disallow any YAML syntax. Iteration can force the use of particular YAML syntax including the above features.

A template can rewrite user steps and only allow certain approved tasks to run. You can, for example, prevent inline script execution.

> ⚠ **Warning**
>
> In the example below, the steps type "bash", "powershell", "pwsh" and "script" are prevented from executing. For full lockdown of ad-hoc scripts, you would also need to block "BatchScript" and "ShellScript".

YAML

```yaml
# template.yml
parameters:
- name: usersteps
  type: stepList
  default: []
steps:
- ${{ each step in parameters.usersteps }}:
  - ${{ if not(or(startsWith(step.task, 'Bash'),startsWith(step.task,
'CmdLine'),startsWith(step.task, 'PowerShell'))) }}:
    - ${{ step }}
  # The lines below will replace tasks like Bash@3, CmdLine@2, PowerShell@2
  - ${{ else }}:
    - ${{ each pair in step }}:
        ${{ if eq(pair.key, 'inputs') }}:
          inputs:
            ${{ each attribute in pair.value }}:
              ${{ if eq(attribute.key, 'script') }}:
                script: echo "Script removed by template"
              ${{ else }}:
                ${{ attribute.key }}: ${{ attribute.value }}
```

```
        ${{ elseif ne(pair.key, 'displayName') }}:
          ${{ pair.key }}: ${{ pair.value }}

          displayName: 'Disabled by template: ${{ step.displayName }}'
```

YAML

```yaml
# azure-pipelines.yml
extends:
  template: template.yml
  parameters:
    usersteps:
    - task: MyTask@1
    - script: echo This step will be stripped out and not run!
    - bash: echo This step will be stripped out and not run!
    - powershell: echo "This step will be stripped out and not run!"
    - pwsh: echo "This step will be stripped out and not run!"
    - script: echo This step will be stripped out and not run!
    - task: CmdLine@2
      displayName: Test - Will be stripped out
      inputs:
        script: echo This step will be stripped out and not run!
    - task: MyOtherTask@2
```

Type-safe parameters

Templates and their parameters are turned into constants before the pipeline runs. Template parameters provide type safety to input parameters. For instance, it can restrict which pools can be used in a pipeline by offering an enumeration of possible options rather than a freeform string.

YAML

```yaml
# template.yml
parameters:
- name: userpool
  type: string
  default: Azure Pipelines
  values:
  - Azure Pipelines
  - private-pool-1
  - private-pool-2

pool: ${{ parameters.userpool }}
steps:
- script: # ... removed for clarity
```

YAML

```yaml
# azure-pipelines.yml
extends:
  template: template.yml
  parameters:
    userpool: private-pool-1
```

Set required templates

To require that a specific template gets used, you can set the required template check for a resource or environment. The required template check can be used when extending from a template.

You can check on the status of a check when viewing a pipeline job. When a pipeline doesn't extend from the require template, the check will fail and the run will stop. You will see that your check failed.

When the required template is used, you'll see that your check passed.

Here the template `params.yml` is required with an approval on the resource. To trigger the pipeline to fail, comment out the reference to `params.yml`.

YAML

```yaml
# params.yml
parameters:
- name: yesNo
  type: boolean
  default: false
- name: image
  displayName: Pool Image
  type: string
  default: ubuntu-latest
```

```
  values:
  - windows-latest
  - ubuntu-latest
  - macOS-latest

steps:
- script: echo ${{ parameters.yesNo }}
- script: echo ${{ parameters.image }}
```

YAML

```
# azure-pipeline.yml

resources:
 containers:
     - container: my-container
       endpoint: my-service-connection
       image: mycontainerimages

extends:
    template: params.yml
    parameters:
        yesNo: true
        image: 'windows-latest'
```

Additional steps

A template can add steps without the pipeline author having to include them. These steps can be used to run credential scanning or static code checks.

YAML

```
# template to insert a step before and after user steps in every job
parameters:
  jobs: []

jobs:
- ${{ each job in parameters.jobs }}: # Each job
  - ${{ each pair in job }}:  # Insert all properties other than "steps"
      ${{ if ne(pair.key, 'steps') }}:
        ${{ pair.key }}: ${{ pair.value }}
    steps:                              # Wrap the steps
    - task: CredScan@1                  # Pre steps
    - ${{ job.steps }}                  # Users steps
    - task: PublishMyTelemetry@1        # Post steps
      condition: always()
```

Template enforcement

A template is only a security mechanism if you can enforce it. The control point to enforce use of templates is a protected resource. You can configure approvals and checks on your agent pool or other protected resources like repositories. For an example, see Add a repository resource check.

Next steps

Next, learn about taking inputs safely through variables and parameters.

How to securely use variables and parameters in your pipeline

Article • 09/08/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

This article discusses how to securely use variables and parameters to gather input from pipeline users. If you'd like to learn more about using variables and parameters, see:

- Define variables
- Use predefined variables
- Use runtime parameters
- Template types & usage

Variables

Variables can be a convenient way to collect information from the user up front. You can also use variables to pass data from step to step within a pipeline.

But use variables with caution. Newly created variables, whether they're defined in YAML or written by a script, are read-write by default. A downstream step can change the value of a variable in a way that you don't expect.

For instance, your script reads:

```batch
msbuild.exe myproj.proj -property:Configuration=$(MyConfig)
```

A preceding step could set `MyConfig` to `Debug & deltree /y c:`. Although this example would only delete the contents of your build agent, you can imagine how this setting could easily become far more dangerous.

You can make variables read-only. System variables like `Build.SourcesDirectory`, task output variables, and queue-time variables are always read-only. Variables that are created in YAML or created at run time by a script can be designated as read-only. When a script or task creates a new variable, it can pass the `isReadonly=true` flag in its logging command to make the variable read-only.

In YAML, you can specify read-only variables by using a specific key:

YAML

```yaml
variables:
- name: myReadOnlyVar
  value: myValue
  readonly: true
```

Queue-time variables

When defining a variable in the Pipelines UI editor, you can choose to let users override its value when running the pipeline. We call such a variable a queue-time variable. Queue-time variables are always defined in the Pipelines UI editor.

Queue-time variables are exposed to the end user when they manually run a pipeline, and they can change their values.

Limit variables that can be set at queue time

The UI and REST API used to run a pipeline provide means for users to define new variables at queue time.

In the early days of Azure Pipelines, this functionality had some issues:

- It allowed users to define new variables that aren't explicitly defined by the pipeline author in the definition.
- It allowed users to override system variables.

To correct these issues, we defined a setting to limit variables that can be set at queue time. With this setting enabled, only those variables that are explicitly marked as "Settable at queue time" can be set. In other words, you can set any variables at queue time unless this setting is enabled.

The setting is designed to work at organization level and at project level.

1. Organization level. When the setting is on, it enforces that, for all pipelines in all projects in the organization, only those variables that are explicitly marked as "Settable at queue time" can be set. When the setting is off, each project can choose whether to restrict variables set at queue time or not. The setting is a toggle under Organization Settings -> Pipelines -> Settings. Only Project

Collection Administrators can enable or disable it.

2. Project level. When the setting is on, it enforces that, for all pipelines in the project, only those variables that are explicitly marked as "Settable at queue time" can be set. If the setting is on at the organization level, then it is on for all projects and can't be turned off. The setting is a toggle under Project Settings -> Pipelines -> Settings. Only Project Administrators can enable or disable it.

Lets look at an example. Say the setting is on and your pipeline defines a variable named `my_variable` that isn't settable at queue time.

Next, assume you wish to run the pipeline. The *Variables* panel doesn't show any variables, and the *Add variable* button is missing.

Using the Builds - Queue and the Runs - Run Pipeline REST API calls to queue a pipeline run and set the value of `my_variable` or of a new variable will fail with an error similar to the following.

JSON

```
{
  "$id": "1",
  "innerException": null,
  "message": "You can't set the following variables (my_variable). If you want to be able to set these variables, then edit the pipeline and select Settable at queue time on the variables tab of the pipeline editor.",
  "typeName": "Microsoft.Azure.Pipelines.WebApi.PipelineValidationException, Microsoft.Azure.Pipelines.WebApi",
  "typeKey": "PipelineValidationException",
  "errorCode": 0,
  "eventId": 3000
}
```

Parameters

Unlike variables, pipeline parameters can't be changed by a pipeline while it's running. Parameters have data types such as `number` and `string`, and they can be restricted to a subset of values. Restricting the parameters is useful when a user-configurable part of the pipeline should take a value only from a constrained list. The setup ensures that the pipeline won't take arbitrary data.

Enable shell tasks arguments parameter validation

Pipelines can reference tasks that are executed in the pipeline. Several tasks included in Azure DevOps have an arguments parameter that lets you specify more options for the task.

When the setting *Enable shell tasks arguments parameter validation* is enabled, the arguments parameter is reviewed for any characters that may not be executed correctly

by the shell. Example characters include semi-colons, quotes, and parentheses.

Similar to the *Limit variables that can be set at queue time* option, *Enable shell tasks arguments parameter validation* may be configured at the organization-level at **Settings** > **Pipelines** > **Settings** or at the project-level at **Settings** > **Pipelines** > **Settings**.

When enabled, there's a detected validation issue, an error message like the following is logged: `Detected characters in arguments that may not be executed correctly by the shell. Please escape special characters using backtick (`).`

To resolve the issue, adjust the arguments by escaping special characters as indicated in the error message.

When *Enable shell tasks arguments parameter validation* is enabled, validation is applied to the arguments parameter in the following tasks.

- PowerShell
- BatchScript
- Bash
- Ssh
- AzureFileCopy
- WindowsMachineFileCopy

Next steps

After you secure your inputs, you also need to secure your shared infrastructure.

Recommendations to secure shared infrastructure in Azure Pipelines

Article • 10/16/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

Protected resources in Azure Pipelines are an abstraction of real infrastructure. Follow these recommendations to protect the underlying infrastructure.

Use Microsoft-hosted pools

Microsoft-hosted pools offer isolation and a clean virtual machine for each run of a pipeline. If possible, use Microsoft-hosted pools rather than self-hosted pools.

Separate agents for each project

An agent can be bound to only one pool. You might want to share agents across projects by sharing the pool with multiple projects. In other words, multiple projects might run jobs on the same agent, one after another. Although this practice saves infrastructure costs, it can allow lateral movement.

To eliminate that form of lateral movement and to prevent one project from "poisoning" an agent for another project, keep separate agent pools with separate agents for each project.

Use low-privileged accounts to run agents

It's tempting but dangerous to run the agent under an identity that can directly access Azure DevOps resources. This problematic setup is common in organizations that use Microsoft Entra ID. If you run the agent under an identity that's backed by Microsoft Entra ID, then it can directly access Azure DevOps APIs without using the job's access token. You should instead run the agent as a non-privileged local account such as *Network Service*.

Azure DevOps has a group that's misleadingly named *Project Collection Service Accounts*. By inheritance, members of Project Collection Service Accounts are also members of Project Collection Administrators. Customers sometimes run their build agents by using an identity that's backed by Microsoft Entra ID and that's a member of Project Collection

Service Accounts. If adversaries run a pipeline on one of these build agents, then they can take over the entire Azure DevOps organization.

We've also seen self-hosted agents run under highly privileged accounts. Often, these agents use privileged accounts to access secrets or production environments. But if adversaries run a compromised pipeline on one of these build agents, then they can access those secrets. Then the adversaries can move laterally through other systems that are accessible through those accounts.

To keep your systems secure, use the lowest-privileged account to run self-hosted agents. For example, use your machine account or a managed service identity. Let Azure Pipelines manage access to secrets and environments.

Minimize the scope of service connections

Service connections must be able to access only the resources that they require. For instance, an Azure service connection should use Azure Resource Manager and service principals that are scoped to the resources that they need to access. They shouldn't have broad contributor rights for the entire Azure subscription.

When you create a new Azure Resource Manager service connection, always select a resource group. Ensure that your resource group contains only the VMs or resources that the build requires. Similarly, when you configure the GitHub app, grant access only to the repositories that you want to build by using Azure Pipelines.

Next steps

Consider a few general recommendations for security.

Other security considerations

Article • 01/27/2023

Azure DevOps Services | Azure DevOps Server 2022 | Azure DevOps Server 2020

There are a handful of other things you should consider when securing pipelines.

Relying on PATH

Relying on the agent's `PATH` setting is dangerous. It may not point where you think it does, since a previous script or tool could have altered it. For security-critical scripts and binaries, always use a fully qualified path to the program.

Logging of secrets

Azure Pipelines attempts to scrub secrets from logs wherever possible. This filtering is on a best-effort basis and can't catch every way that secrets can be leaked. Avoid echoing secrets to the console, using them in command line parameters, or logging them to files.

Lock down containers

Containers have a few system-provided volume mounts mapping in the tasks, the workspace, and external components required to communicate with the host agent. You can mark any or all of these volumes read-only.

YAML

```yaml
resources:
  containers:
  - container: example
    image: ubuntu:22.04
    mountReadOnly:
      externals: true
      tasks: true
      tools: true
      work: false  # the default; shown here for completeness
```

Most people should mark the first three read-only and leave `work` as read-write. If you know you won't write to the work directory in a given job or step, go ahead and make

`work` read-only as well. If you have tasks in your pipeline, which self-modify, you may need to leave `tasks` read-write.

Control available tasks

You can disable the ability to install and run tasks from the Marketplace. This will allow you greater control over the code that executes in a pipeline. You may also disable all the in-the-box tasks (except Checkout, which is a special action on the agent). We recommend that you don't disable in-the-box tasks under most circumstances.

Tasks directly installed with tfx are always available. With both of these features enabled, **only** those tasks are available.

Use the Auditing service

Many pipeline events are recorded in the Auditing service. Review the audit log periodically to ensure no malicious changes have slipped past. Visit `https://dev.azure.com/ORG-NAME/_settings/audit` to get started.

Next steps

Return to the overview and make sure you've covered every article.

Download permission report for a release

Article • 02/24/2023

Azure DevOps Services

To determine the effective permissions of users and groups for a release, you can download the permissions report. Requesting the report generates an email with a link to download the report. The report lists the effective permissions for the release you select, for each user and group specified at the time the report is generated. Inherited permissions come from a parent group which you can view from the web portal. The report is a json-formatted report that you can open using Power BI or other json reader.

You can also use the Permissions Report REST API to download the report.

Prerequisites

- To download the permissions report, you must be a member of the **Project Collection Administrators** group. The user interface button won't appear for users who aren't a member of this group.

  To find a member of the **Project Collection Administrators** group, see Look up a project collection administrator.

Open the security dialog for the release

You can download the report for a specific release from the release's Security dialog.

- Open the web portal, navigate to **Pipelines**>**Releases**, and choose the release you want to download permissions for. Choose ⋮ **More actions** and then choose **Security**.

Download report

1. Choose **Download detailed report**.

The following message displays indicating the request was submitted to the server.

2. Once you receive the email from Azure DevOps Notifications, open it and choose **Download Report**.

   A report labeled **PermissionsReport**_*GUID*.json appears in your **Downloads** folder.

> ⓘ **Important**
>
> Reports are automatically deleted after 28 days of the request.

Related articles

- Set different levels of pipeline permissions
- Manage permissions with command line tool
- Permissions Report REST API

CI/CD baseline architecture with Azure Pipelines

Article • 05/08/2023

This article describes a high-level DevOps workflow for deploying application changes to staging and production environments in Azure. The solution uses continuous integration/continuous deployment (CI/CD) practices with Azure Pipelines.

> **Important**
>
> This article covers a general CI/CD architecture using Azure Pipelines. It is not intended to cover the specifics of deploying to different environments, such as Azure App Services, Virtual Machines, and Azure Power Platform. Deployment platform specifics are covered in separate articles.

Architecture

Download a Visio file of this architecture.

> **Note**
>
> Although this article covers CI/CD for application changes, Azure Pipelines can also be used to build CI/CD pipelines for infrastructure as code (IaC) changes.

Dataflow

The data flows through the scenario as follows:

1. **PR pipeline** - A pull request (PR) to Azure Repos Git triggers a PR pipeline. This pipeline runs fast quality checks. These checks should include:

    - Building the code, which requires pulling dependencies from a dependency management system.
    - The use of tools to analyze the code, such as static code analysis, linting, and security scanning
    - Unit tests

    If any of the checks fail, the pipeline run ends and the developer will have to make the required changes. If all checks pass, the pipeline should require a PR review. If the PR review fails, the pipeline ends and the developer will have to make the required changes. If all the checks and PR reviews pass, the PR will successfully merge.

2. **CI pipeline** - A merge to Azure Repos Git triggers a CI pipeline. This pipeline runs the same checks as the PR pipeline with some important additions. The CI pipeline runs integration tests. These integration tests shouldn't require the deployment of the solution, as the build artifacts haven't been created yet. If the integration tests require secrets, the pipeline gets those secrets from Azure Key Vault. If any of the checks fail, the pipeline ends and the developer will have to make the required changes. The result of a successful run of this pipeline is the creation and publishing of build artifacts

3. **CD pipeline trigger** - The publishing of artifacts triggers the CD pipeline.

4. **CD release to staging** - The CD pipeline downloads the build artifacts that are created in the CI pipeline and deploys the solution to a staging environment. The pipeline then runs acceptance tests against the staging environment to validate the deployment. If any acceptance test fails, the pipeline ends and the developer will have to make the required changes. If the tests succeed, a manual validation task can be implemented to require a person or group to validate the deployment and resume the pipeline.

5. **CD release to production** - If the manual intervention is resumed, or there's no manual intervention implemented, the pipeline releases the solution to production. The pipeline should run smoke tests in production to ensure the release is working as expected. If a manual intervention step results in a cancel, the release fails, or the smoke tests fail, the release is rolled back, the pipeline ends and the developer will have to make the required changes.

6. **Monitoring** - Azure Monitor collects observability data such as logs and metrics so that an operator can analyze health, performance, and usage data. Application

Insights collects all application-specific monitoring data, such as traces. Azure Log Analytics is used to store all that data.

Components

- An Azure Repos Git repository serves as a code repository that provides version control and a platform for collaborative projects.

- Azure Pipelines provides a way to build, test, package and release application and infrastructure code. This example has three distinct pipelines with the following responsibilities:
  - PR pipelines validate code before allowing a PR to merge through linting, building and unit testing.
  - CI pipelines run after code is merged. They perform the same validation as PR pipelines, but add integration testing and publish build artifacts if everything succeeds.
  - CD pipelines deploy build artifacts, run acceptance tests, and release to production.

- Azure Artifact Feeds allow you to manage and share software packages, such as Maven, npm, and NuGet. Artifact feeds allow you to manage the lifecycle of your packages, including versioning, promoting, and retiring packages. This helps you to ensure that your team is using the latest and most secure versions of your packages.

- Key Vault provides a way to manage secure data for your solution, including secrets, encryption keys, and certificates. In this architecture, it's used to store application secrets. These secrets are accessed through the pipeline. Secrets can be accessed by Azure Pipelines with a Key Vault task or by linking secrets from Key Vault.

- Monitor is an observability resource that collects and stores metrics and logs, application telemetry, and platform metrics for the Azure services. Use this data to monitor the application, set up alerts, dashboards, and perform root cause analysis of failures.

- Application Insights is a monitoring service that provides real-time insights into the performance and usage of your web applications.

- Log Analytics workspace provides a central location where you can store, query, and analyze data from multiple sources, including Azure resources, applications, and services.

Alternatives

While this article focuses on Azure Pipelines, you could consider these alternatives:

- Azure DevOps Server (previously known as Team Foundation Server) could be used as an on-premises substitute.
- Jenkins is an open source tool used to automate builds and deployments.
- GitHub Actions allow you to automate your CI/CD workflows directly from GitHub.
- GitHub Repositories can be substituted as the code repository. Azure Pipelines integrates seamlessly with GitHub repositories.

This article focuses on general CI/CD practices with Azure Pipelines. The following are some compute environments to which you could consider deploying:

- App Services is an HTTP-based service for hosting web applications, REST APIs, and mobile back ends. You can develop in your favorite language, and applications run and scale with ease on both Windows and Linux-based environments. Web Apps supports deployment slots like staging and production. You can deploy an application to a staging slot and release it to the production slot.
- Azure Virtual Machines handles workloads that require a high degree of control, or depend on OS components and services that aren't possible with Web Apps (for example, the Windows GAC, or COM).
- Azure Power Platform is a collection of cloud services that enable users to build, deploy, and manage applications without the need for infrastructure or technical expertise.
- Azure Functions is a serverless compute platform that you can use to build applications. With Functions, you can use triggers and bindings to integrate services. Functions also support deployment slots like staging and production. You can deploy an application to a staging slot and release it to the production slot.
- Azure Kubernetes Service (AKS) is a managed Kubernetes cluster in Azure. Kubernetes is an open source container orchestration platform.
- Azure Container Apps allows you to run containerized applications on a serverless platform.

Scenario details

Using proven CI and CD practices to deploy application or infrastructure changes provides various benefits including:

- **Shorter release cycles** - Automated CI/CD processes allow you to deploy faster than manual practices. Many organizations deploy multiple times per day.
- **Better code quality** - Quality gates in CI pipelines, such as linting and unit testing, result in higher quality code.
- **Decreased risk of releasing** - Proper CI/CD practices dramatically decreases the risk of releasing new features. The deployment can be tested prior to release.
- **Increased productivity** - Automated CI/CD frees developers from working on manual integrations and deployments so they can focus on new features.
- **Enable rollbacks** - While proper CI/CD practices lower the number of bugs or regressions that are released, they still occur. CI/CD can enable automated rollbacks to earlier releases.

Potential use cases

Consider Azure Pipelines and CI/CD processes for:

- Accelerating application development and development lifecycles.
- Building quality and consistency into an automated build and release process.
- Increasing application stability and uptime.

Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see Microsoft Azure Well-Architected Framework.

Operational excellence

- Consider implementing Infrastructure as Code (IaC) to define your infrastructure and to deploy it in your pipelines.

- Consider using one of the Tokenization Tasks available in the VSTS marketplace, in the context often refer to a process where sensitive information (such as API keys, passwords, or other secrets) is replaced with tokens or placeholders during deployment or configuration.

- Use Release Variables in your release definitions to drive configuration changes of your environments. Release variables can be scoped to an entire release or a given

environment. When using variables for secret information, ensure that you select the padlock icon.

- Consider using Self-hosted agents if you're deploying to resources running in a secured virtual network. You might also consider self-hosted agents if you're running a high volume of builds. In cases of high build volumes, self-hosted agents can be used to speed up builds in a cost efficient manner.
- Consider using Application Insights and other monitoring tools as early as possible in your release pipeline. Many organizations only begin monitoring in their production environment. By monitoring your other environments, you can identify bugs earlier in the development process and avoid issues in your production environment.
- Consider using separate monitoring resources for production.
- Consider using YAML pipelines instead of the Classic interface. YAML pipelines can be treated like other code. YAML pipelines can be checked in to source control and versioned, for example.
- Consider using YAML Templates to promote reuse and simplify pipelines. For example, PR and CI pipelines are similar. A single parameterized template could be used for both pipelines.
- Consider creating environments beyond staging and production to support activities such as manual user acceptance testing, performance and load testing, and rollbacks.

Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see Overview of the cost optimization pillar.

Azure DevOps costs depend on the number of users in your organization that require access, along with other factors like the number of concurrent build/releases required and number of test users. For more information, see Azure DevOps pricing.

This pricing calculator provides an estimate for running Azure DevOps with 20 users.

Azure DevOps is billed on a per-user per-month basis. There might be more charges depending on concurrent pipelines needed, in addition to any additional test users or user basic licenses.

Security

- Consider the security benefits of using Microsoft-hosted agents when choosing whether to use Microsoft-hosted or self-hosted agents.
- Ensure all changes to environments are done through pipelines. Implement role-based access controls (RBAC) on the principle of least privilege, preventing users from accessing environments.
- Consider integrating steps in Azure Pipelines to track dependencies, manage licensing, scan for vulnerabilities, and keep dependencies to date.

Next steps

Review the following resources to learn more about CI/CD and Azure DevOps:

- What is DevOps?
- DevOps at Microsoft - How we work with Azure DevOps
- Create a CI/CD pipeline for .NET with Azure DevOps Projects
- What is Azure Repos?
- What is Azure Pipelines?
- Azure DevOps
- App Service overview
- Introduction to Azure Functions
- Azure Key Vault basic concepts
- Azure Monitor overview

Related resources

- DevOps Checklist
- CI/CD for Azure VMs
- CI/CD for Containers
- Build a CI/CD pipeline for microservices on Kubernetes

Azure Pipelines architecture for Azure Web Apps

Article • 05/08/2023

> **Important**
>
> CI/CD for Azure Web Apps is a variant of **Design a CI/CD pipeline using Azure DevOps**. This article focuses on the Web Apps-specific facets of deployment.

Azure Web Apps is a fast and simple way to create web apps using ASP.NET, Java, Node.js, Python, and other languages and frameworks. Deliver value faster to your customers with a continuous integration and continuous deployment (CI/CD) pipeline that pushes each of your changes automatically to Azure Web Apps.

Architecture

Download a Visio file of this architecture.

Dataflow

This section assumes you have read Azure Pipelines baseline architecture and only focuses on the considerations specifics to deploying a workload to Azure App Services.

1. **PR pipeline** - *Same as the baseline*
2. **CI pipeline** - Same as the baseline, except the build artifacts created for Web Apps is a Web Deploy package.
3. **CD pipeline trigger** - *Same as the baseline*

4. **CD release to staging** - Same as the baseline with 2 exceptions: 1) the build artifact that is downloaded is the Web Deploy Package and 2) the package is deployed to a staging slot in App Services.

5. **CD release to production** - Same as the baseline with 2 exceptions: 1) the release to production for a Web App swaps the production and staging slot, and 2) the rollback for Web Apps swaps production and staging slots back.

6. **Monitoring** - *same as the baseline*

Components

This section assumes you have read Azure Pipelines baseline architecture components section and only focuses on the considerations specifics to deploying a workload to Azure App Services.

- Azure App Service: Azure App Service is an HTTP-based service for hosting web applications, REST APIs, and mobile back ends. Azure Web Apps are actually applications hosted in Azure App Service.

- Azure Web Apps: Quickly create and deploy mission-critical Web apps at scale. Azure Web Apps has many offerings, including Windows Web Apps, Linux Web Apps, and Web Apps for Containers.

Considerations

This section assumes you have read the considerations section in Azure Pipelines baseline architecture and only focuses on the considerations specifics to deploying a workload to Azure App Services.

Operational Excellence

- Consider implementing environments beyond just staging and production to enable things like rollbacks, manual acceptance testing, and performance testing. The act of using staging as the rollback environment keeps you from being able to use that environment for other purposes.

Next steps

- Get started with continuous deployment to Azure App Service
- Get started with Git in Azure Repos

- Deploy to App Service using Azure Pipelines
- Deploy to Azure Web App for Containers
- Configure continuous deployment with custom containers in Azure App Service
- Learn about work item integration with Application Insights
- Link GitHub commits, pull requests, and issues to work items in Azure Boards

Related resources

- CI/CD baseline architecture with Azure Pipelines

Azure Pipelines architecture with DevTest Labs

Article • 05/08/2023

> **Important**
>
> CI/CD with DevTest Labs is a variant of **Design a CI/CD pipeline using Azure DevOps**. This article focuses on the specifics of deploying to a DevTest Labs staging environments.

DevTest Labs allow you to provision Windows and Linux environments by using reusable templates and artifacts. These environments can be useful for developers, but can also be used in CI/CD pipelines for provisioning staging environments. See Azure DevTest Labs scenarios to see if DevTest labs is a good fit for your scenario.

This article describes a high-level DevOps workflow for deploying application changes using continuous integration (CI) and continuous deployment (CD) practices using Azure Pipelines. A DevTest Labs environment is used for the staging environment.

Architecture

Download a Visio file of this architecture.

Dataflow

This section assumes you have read Azure Pipelines baseline architecture and only focuses on the specifics of deploying a workload to Azure DevTest Labs for staging.

1. **PR pipeline** - *Same as the baseline*
2. **CI pipeline** - *Same as the baseline*
3. **CD pipeline trigger** - *Same as the baseline*
4. **CD create DevTest Labs staging environment** - This step creates the DevTest Labs environment which acts as the staging environment. The step includes:
    - Create Azure DevTest Labs environment in a staging subscription.
    - Deploy an ARM template to the DevTest Labs environment. Virtual Machine images can be stored in a shared image gallery.
    - Perform any post deployment steps to properly configure the staging environment.
5. **CD release to staging** - Same as the baseline with one exception. The staging environment is a DevTest Labs environment.
6. **CD release to production** - *Same as the baseline*
7. **Monitoring** - *same as the baseline*

Components

This section assumes you have read Azure Pipelines baseline architecture components section and only focuses on the specifics of deploying a workload to Azure DevTest Labs for staging.

- Azure DevTest Labs is a service for creating, using, and managing environments used for development, testing and deployment purposes. The service allows you to easily deploy pre-configured environments in a cost-effective manner.

Alternatives

- An alternative to creating the DevTest Labs staging environment as part of the CD process, you can pre-create the environment outside of the pipeline. This will have the positive benefit of speeding up the pipeline. This alternative will stop the ability to tear down the environment after the pipeline is complete, increasing the cost.
- In situations where VM Image Builder and a Shared Image Gallery don't work, you can set up an image factory to build VM images from the CI/CD pipeline and distribute them automatically to any Azure DevTest Labs registered to those images. For more information, see Run an image factory from Azure DevOps.

- Additional environments, beyond staging could be created and deployed to as part of the CD pipeline. These environments could support activities like performance testing and user acceptance testing.

Considerations

This section assumes you have read the considerations section in Azure Pipelines baseline architecture and only focuses on the specifics of deploying a workload to Azure DevTest Labs for staging.

Cost Optimization

- Consider using Azure DevTest Labs policies and procedures to control costs

Operational Excellence

- Consider implementing environments beyond just staging and production to enable things like rollbacks, manual acceptance testing, and performance testing. The act of using staging as the rollback environment keeps you from being able to use that environment for other purposes.

Next steps

- Create a lab in Azure DevTest Labs
- Integrate DevTest Labs into Azure Pipelines

Related resources

- CI/CD baseline architecture with Azure Pipelines
- CI/CD for IaaS applications

Azure Pipelines architecture for IaaS

Article • 05/08/2023

> **Important**
>
> CI/CD for IaaS applications is a variant of **Design a CI/CD pipeline using Azure DevOps**. This article focuses on the specifics of deploying web applications to Azure Virtual Machines.

Azure Virtual Machines is an option for hosting custom applications when you want flexible and granular management of your compute. Virtual machines (VMs) should be subject to the same level of engineering rigor as Platform-as-a-Service (PaaS) offerings throughout the development lifecycle. For example, implementing automated build and release pipelines to push changes to the VMs.

This article describes a high-level DevOps workflow for deploying application changes to VMs using continuous integration (CI) and continuous deployment (CD) practices using Azure Pipelines.

Architecture

Download a Visio file of this architecture.

Dataflow

This section assumes you have read Azure Pipelines baseline architecture and only focuses on the specifics of deploying a workload to Azure Virtual Machines.

1. **PR pipeline** - *Same as the baseline*

2. **CI pipeline** - Same as the baseline, except the build artifacts created for deploying a Web App to IaaS is a Web Deploy package

3. **CD pipeline trigger** - *Same as the baseline*

4. **CD release to staging** - Same as the baseline with 2 exceptions: 1) the build artifact that is downloaded is the Web Deploy Package and 2) the package is deployed to a staging Azure Virtual Machine.

5. **CD release to production** - Same as the baseline with 2 exceptions:

   a. The release to production is done by updating Azure Traffic Manager to swap staging and production. This strategy can be accomplished by having a Traffic Manager profile with two endpoints, where production is enabled and staging is disabled. To swap staging and production, disable production and enable staging.
   b. A rollback can be accomplished by updating Azure Traffic Manager to swap production and staging back.

6. **Monitoring** - *same as the baseline*

Components

This section assumes you have read Azure Pipelines baseline architecture components section and only focuses on the specifics of deploying a workload to Azure Virtual Machines.

- Azure Virtual Machines provide on-demand, high-scale, secure, virtualized infrastructure using Windows or Linux servers. Virtual Machines are used in this architecture to host workloads.

- Virtual Machine Scale Sets let you create and manage a group of identical load-balanced VMs. The number of VM instances can automatically increase or decrease in response to demand or a defined schedule. Scale sets can also be used to host workloads.

- Azure Traffic Manager is a DNS-based traffic load balancer that you can use to distribute traffic to configured endpoints. In this architecture, Traffic Manager is the single entrypoint for clients and is configured with multiple endpoints, representing the production Virtual Machine and the staging Virtual Machine. The production Virtual Machine endpoint is enabled and staging is disabled.

Alternatives

This article focuses on the use of Azure Traffic Manager as the load balancer. Azure offers various Load balancing options that you could consider.

Considerations

This section assumes you have read the considerations section in Azure Pipelines baseline architecture and only focuses on the considerations specifics to deploying a workload to Azure Virtual Machines.

Operational Excellence

- Because Traffic Manager is DNS-based, client caching of IP addresses introduces latency. Even though you might enable one endpoint and disable another in Traffic Manager, clients will continue to use their cached IP address until the DNS Time-to-live (TTL) expires. Consider load balancing options that act at layer 4 or layer 7.

- Consider implementing environments beyond just staging and production to enable things like rollbacks, manual acceptance testing, and performance testing. The act of using staging as the rollback environment keeps you from being able to use that environment for other purposes.

Next steps

- Integrate DevTest Labs into Azure Pipelines
- Create and deploy VM Applications

Related resources

- CI/CD baseline architecture with Azure Pipelines
- Run a Linux VM on Azure

Use Azure Pipelines with Microsoft Teams

Article • 04/24/2023

Azure DevOps Services

The Azure Pipelines app for Microsoft Teams lets you monitor events for your pipelines. You can set up and get notifications in your Teams channel for releases, pending approvals, completed builds, and so on. You can also approve releases from within your Teams channel.

> **Note**
>
> This feature is only available on Azure DevOps Services. Typically, new features are introduced in the cloud service first, and then made available on-premises in the next major version or update of Azure DevOps Server. To learn more, see **Azure DevOps Feature Timeline**.

Prerequisites

- You must have an Azure DevOps project. For more information, see Create a project.
- To set up pipeline subscriptions, you must be a Project Administrator.

Set up the Azure Pipelines app

1. In Microsoft Teams, go to the **Apps** store, search for **Azure Pipelines**, and then select **Azure Pipelines**.

2. Select the **Open** dropdown arrow, and then select **Add to a team**.

3. Select or enter your team name, and then choose **Set up a bot**.

4. In the Teams conversation pane, enter `@azurePipelines signin`.

5. Select **Sign in** and complete authentication to Azure Pipelines.

Use commands

Use the following commands to monitor all pipelines in a project or only specific pipelines.

- **Monitor all pipelines in a project.** The URL can be to any page within your project, except URLs to pipelines. For example, `@azure pipelines subscribe https://dev.azure.com/myorg/myproject/`.

  ```
  @azure pipelines subscribe [project url]
  ```

- **Monitor a specific pipeline:** The pipeline URL can be to any page within your pipeline that has a `definitionId` or `buildId/releaseId` present in the URL. For example, `@azure pipelines subscribe https://dev.azure.com/myorg/myproject/_build?definitionId=123`.

  ```
  @azure pipelines subscribe [pipeline url]
  ```

  Another example of a release pipeline that's subscribed to the *Release deployment started*, *Release deployment completed*, and *Release deployment approval pending*

notifications is `@azure pipelines subscribe https://dev.azure.com/myorg/myproject/_release?definitionId=123&view=mine&_a=releases`.

> **Note**
>
> - We don't support deployment approvals that have applied the **Revalidate identity of approver before completing the approval** policy.
> - Enable 'Third party application access via OAuth' to receive notifications for the organization in Azure DevOps.

Manage subscriptions

When you subscribe to a pipeline, a few subscriptions get created by default without any filters applied. You might want to customize these subscriptions. For example, you might want to get notified only when builds fail or when deployments get pushed to a production environment. The Azure Pipelines app supports filters to customize what you see in your channel. To manage your subscriptions, complete the following steps.

1. Run the `@azure pipelines subscriptions` command.

2. Select **View all subscriptions**. In the list of subscriptions, if there's a subscription you don't want, select **Remove**.

3. Scroll down and select **Add subscription**.

4. Select the required pipeline and event.

5. Select the appropriate filters, and then **Save**.

Example 1: Get notifications for failed builds.

Example 2: Get notifications only if the deployments get pushed to the production environment.

ⓘ **Note**

- Team Administrators can't remove or modify subscriptions that are created by Project Administrators.

- Notifications aren't supported inside chat/direct messages.

Approve deployments from your channel

You can approve deployments from within your channel without going to the Azure Pipelines portal. Subscribe to the *Release deployment approval pending* notification for classic Releases or the *Run stage waiting for approval* notification for YAML pipelines. Both of these subscriptions get created by default when you subscribe to the pipeline.

Whenever the running of a stage is pending for approval, a notification card with options to approve or reject the request gets posted in the channel. Approvers can review the details of the request in the notification and take appropriate action. In the following example, the deployment was approved and the approval status shows on the card.

The Azure Pipelines app supports all of the checks and approval scenarios present in the Azure Pipelines portal. You can approve requests as an individual or for a team.

Search and share pipeline information using compose extension

To help users search and share information about pipelines, Azure Pipelines app for Microsoft Teams supports compose extension. You can now search for pipelines by pipeline ID or by pipeline name. For compose extension to work, users must sign in to the Azure Pipelines project that they're interested in either by running `@azure pipelines signin` command or by signing in to the compose extension directly.

Previews of pipeline URLs

When you add a pipeline URL to Teams, you see a preview similar to the following images. The preview helps to keep pipeline-related conversations relevant and up-to-date. You can choose between compact and expanded cards.

Example 1: Build URL preview

Example 2: Release URL preview

Once you're signed in, this feature works for all channels in a team in Microsoft Teams.

Unsubscribe from a pipeline channel

Use the following command to unsubscribe from all pipelines within a project.

```
@azure pipelines unsubscribe all [project url]
```

Example: Unsubscribe all with URL

```
@azure pipelines unsubscribe all https://dev.azure.com/myorg/myproject
```

This command deletes all the subscriptions related to any pipeline in the project and removes the pipelines from the channel.

> **Important**
>
> Only Project Administrators can run this command.

Link your notifications

All notifications linked to a particular run of a pipeline get linked together.

Example 1: Compact view of linked notifications.

Example 2: Expanded view of linked notifications.

Use commands in Azure Pipelines app

See the following commands supported by the Azure Pipelines app:

Slash command	Functionality
@azure pipelines subscribe [pipeline url/ project url]	Subscribe to a pipeline or all pipelines in a project to receive notifications
@azure pipelines subscriptions	Add or remove subscriptions for this channel
@azure pipelines feedback	Report a problem or suggest a feature
@azure pipelines help	Get help on the slash commands
@azure pipelines signin	Sign in to your Azure Pipelines account
@azure pipelines signout	Sign out from your Azure Pipelines account
@azure pipelines unsubscribe all [project url]	Remove all pipelines (belonging to a project) and their associated subscriptions from a channel

Connect multi-tenants

If you're using a different email or tenant for Microsoft Teams and Azure DevOps, do the following steps to sign in and connect based on your use case.

Use case	Email ID & tenant in Microsoft Teams	Email ID & tenant in Azure DevOps	Sign in action
1	email1@abc.com (tenant 1)	email1@abc.com (tenant 1)	Select **Sign in**
2	email1@abc.com (tenant 1)	email2@pqr.com (tenant 2)	- Sign in to Azure DevOps. - In the same browser, start a new tab and go to https://teams.microsoft.com/. - Run the `sign in` command and select **Sign in**.
3	email1@abc.com (tenant 1)	email2@pqr.com (tenant 2)	- Select **Sign in with different email address**. - In the email ID picker, use the email2 to sign in.
4	email1@abc.com (tenant 1)	email2@pqr.com (non-default tenant 3)	Currently not supported.

Troubleshoot

In the **same browser**, start a new tab and sign in to `https://teams.microsoft.com/`. Run the `@Azure Pipelines signout` command and then run the `@Azure Pipelines signin` command in the channel where the Azure Pipelines app for Microsoft Teams is installed.

Select the `Sign in` button and you get redirected to a consent page like the one in the following example. Ensure that the directory shown beside the email is same as what you chose in the previous step. Accept and complete the sign in process.

If these steps don't resolve your authentication issue, reach out to us at Developer Community.

Related articles

- Use Azure Boards with Microsoft Teams
- Use Azure Repos with Microsoft Teams

Use Azure Pipelines with Slack

Article • 03/28/2023

Azure DevOps Services

With Azure Pipelines app for Slack, Slack users can easily track the events occurring within their pipelines. The app allows users to establish and oversee subscriptions for various pipeline events, such as builds, releases, pending approvals, and more. Notifications for these events are then delivered directly to users' Slack channels

> **Note**
>
> This feature is only available on Azure DevOps Services. Typically, new features are introduced in the cloud service first, and then made available on-premises in the next major version or update of Azure DevOps Server. To learn more, see **Azure DevOps Feature Timeline**.

Prerequisites

- In Slack, you need to have permission to install an app to your Slack workspace.
- For Azure DevOps, you need to be in the Project Collection Administrators or Project Administrators group to set up a Slack subscription.

Install Azure Pipelines app

Navigate to Azure Pipelines Slack app to install the Azure Pipelines app to your Slack workspace. Once added, you'll see a welcome message from the app as below. Use the `/azpipelines` handle to start interacting with the app.

Connect to your pipeline

Once the app has been installed in your Slack workspace, you can connect the app to any pipeline you want to monitor. You'll be asked to authenticate to Azure Pipelines

before running any commands.

To start monitoring all pipelines in a project, use the following slash command inside a channel:

```
/azpipelines subscribe [project url]
```

The project URL can link to any page within your project (except URLs to pipelines). For example: */azpipelines subscribe https://dev.azure.com/myorg/myproject/*

You can also monitor a specific pipeline using the following command:

```
/azpipelines subscribe [pipeline url]
```

The pipeline URL can link to any page within your pipeline that has a `definitionId` or a `buildId/releaseId` in the URL. For example: */azpipelines subscribe https://dev.azure.com/myorg/myproject/_build?definitionId=123*, or: */azpipelines subscribe https://dev.azure.com/myorg/myproject/_release?definitionId=123&view=mine&_a=releases*

The subscribe command gets you started with a few subscriptions by default. Here are the default notifications enabled for the following pipeline type:

Build pipelines	Release pipelines	YAML pipelines
Build completed notification	*Release deployment started*, *Release deployment completed* and *Release deployment approval pending* notifications	*Run stage state changed* and *Run stage waiting for approval* notifications

Manage subscriptions

To manage the subscriptions for a channel, use the following command: `/azpipelines subscriptions`

This command lists all the current subscriptions for the channel and allows you to add or remove subscriptions.

> **Note**
>
> Team administrators are not able to remove or modify subscriptions created by Project administrators.

Customize subscriptions

By default, when subscribing to a pipeline, several subscriptions are generated that do not have any filters applied. However, it is common for users to want to personalize these subscriptions according to their preferences. For instance, users may wish to receive notifications only for failed builds or when deployments are made to production. The Azure Pipelines app offers the option to apply filters, enabling users to customize the messages they receive in their channel. To customize a subscription:

1. Run the `/azpipelines subscriptions` command to list all your subscriptions.

2. Select **Add subscription**.

3. Select the event you want to subscribe to, and then select your desired configuration.

4. Select **Save** when you're done.

Example: Get notifications only for failed builds

Approve deployments

You can approve deployments from within your Slack channel without navigating to the Azure Pipelines portal by subscribing to the *Release deployment approval pending* notifications (classic releases) or the *Run stage waiting for approval* notifications (YAML pipelines). Both subscriptions are created by default when you subscribe to a pipeline.

The Azure Pipelines app for Slack enables you to handle all the checks and approval scenarios that are available in the Azure Pipelines portal. These include single approver, multiple approvers, and team-based approval. You have the option to approve requests either individually or on behalf of a team.

Remove all subscriptions

To declutter your channel, you can make use of the following commands to unsubscribe from all pipelines within a project. For example: */azpipelines unsubscribe all https://dev.azure.com/myorg/myproject*.

```
/azpipelines unsubscribe all [project url]
```

> **Important**
>
> This command can only be executed by project administrators.

Commands reference

Here are all the commands supported by the Azure Pipelines app for Slack:

Slash command	Functionality
/azpipelines subscribe [pipeline url/ project url]	Subscribe to a pipeline or all pipelines in a project to receive notifications
/azpipelines subscriptions	Add or Remove subscriptions for this channel
/azpipelines feedback	Report a problem or suggest a feature

Slash command	Functionality
/azpipelines help	Get help on the slash commands
/azpipelines signin	Sign in to your Azure Pipelines account
/azpipelines signout	Sign out from your Azure Pipelines account
/azpipelines unsubscribe all [project url]	Remove all pipelines (belonging to a project) and their associated subscriptions from a channel

Notifications in Private channels

The Azure Pipelines app can help you monitor the pipelines activity in your private channels as well. You'll need to invite the bot to your private channel by using `/invite @azpipelines`. Once the bot has been added, you can configure and control your notifications in the same manner as you would for a public channel.

> **Note**
>
> - You can use the Azure Pipelines app for Slack only with a project hosted on Azure DevOps Services at this time.
> - The user has to be an admin of the project containing the pipeline to set up the subscriptions
> - Notifications are currently not supported inside direct messages
> - Deployment approvals which have 'Revalidate identity of approver before completing the approval' policy applied, are not supported
> - 'Third party application access via OAuth' must be enabled to receive notifications for the organization in Azure DevOps (Organization Settings -> Security -> Policies)

Troubleshooting

If you are experiencing the following errors when using the Azure Pipelines App for Slack, follow the procedures in this section.

Sorry, something went wrong. Please try again

The Azure Pipelines app uses the OAuth authentication protocol, and requires Third-party application access via OAuth to be enabled. To enable this setting, navigate to

Organization Settings > **Security** > **Policies**, and turn on the **Third-party application access via OAuth for the organization**.

Configuration failed. Please make sure that the organization exists and that you have sufficient permissions

Sign out of Azure DevOps by navigating to this URL: `https://aka.ms/VsSignout`.

Open a private/incognito browser window and navigate to `https://aex.dev.azure.com/me` and sign in. Select the directory that contains the organization where the pipeline you want to subscribe to is located.

Using the **same browser**, open a new tab and go to `https://slack.com`. Log in to your workspace using the web client, and then run the `/azpipelines signout` command followed by the `/azpipelines signin`.

Select the `Sign in` button and you'll be redirected to a consent page as shown in the example below. Verify that the directory displayed next to your email address matches the one selected in the previous step. Select **Accept** to complete the sign-in process.

Related articles

- Azure Boards with Slack
- Azure Repos with Slack
- Create a service hook for Azure DevOps with Slack

Integrate with ServiceNow change management

Article • 01/18/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Azure Pipelines support integration with ServiceNow to improve collaboration between development and IT teams. By including change management in release pipelines, teams can reduce the risks associated with changes and follow service management methodologies such as ITIL while taking full advantage of Azure Pipelines.

In this article, you'll learn how to:

- ✓ Configure ServiceNow instances.
- ✓ Include ServiceNow change management process as a release gate.
- ✓ Monitor change management process from release pipelines.
- ✓ Keep ServiceNow change requests updated with deployment results.

Prerequisites

- This tutorial expands on Use approvals and gates and Define approvals & checks.
- An Azure DevOps organization. Create an organization, if you don't have one already.
- A non-developer instance of ServiceNow.

Configure the ServiceNow instance

1. Install the Azure Pipelines extension on your ServiceNow instance. You'll need **Hi** credentials to complete the installation. See Buying Overview for more details on installing apps from the ServiceNow store.
2. Create a new user in ServiceNow and grant it the following role: `x_mioms_azpipeline.pipelinesExecution`.

Set up the Azure DevOps organization

1. Install the ServiceNow Change Management extension on your Azure DevOps organization.

2. Create a new ServiceNow service connection in your Azure DevOps project as follows. Alternatively, you can also use OAuth2 authentication.

Configure the release pipeline

1. Navigate to your release pipeline, and then select the **Pre-deployment conditions** icon. Select **Gates** and the **ServiceNow Change Management** pre-deployment gate.

2. Select the service connection you created earlier and fill out the required fields as follows:

Deployment gates + Add

ServiceNow change management Enabled

ServiceNow Change Management

Display name *

ServiceNow change management

ServiceNow connection * | Manage

VendorInstance + New

Short Description *

Deployment to $(Release.EnvironmentName) of Release $(Release.ReleaseId)

Description

Deployment to $(Release.EnvironmentName) of Release $(Release.ReleaseId)

Category *

Software

Priority *

Low

Risk *

Low

Impact *

2 - Medium

Configuration Item

7300

Assignment Group

eCAB Approval

Schedule of change request

Advanced

Success criteria

Desired status of change request *

Implement

Output Variables

Reference name

gate1

Variables list

gate1.CHANGE_REQUEST_NUMBER

gate1.CHANGE_SYSTEM_ID

- **ServiceNow connection**: Connection to the ServiceNow instance used for change management.

- **Short description**: A summary of the change.
- **Description**: A detailed description of the change.
- **Category**: The category of the change. Example: Hardware, Network, Software.
- **Priority**: Priority of the change.
- **Risk**: The risk level for the change.
- **Impact**: The effect that the change has on business.
- **Configuration Item**: Configuration item (CI) that the change applies to.
- **Assignment group**: The group that the change is assigned to.
- **Schedule of change request**: Schedule of the change as honored by the ServiceNow workflow. Date and time should be in UTC and format should be *yyyy-MM-ddTHH:mm:ssZ*. Example: 2018-01-31T07:56:59Z.
- **Additional change request parameters**: Name must be field name (not label) prefixed with 'u_'. Example: *u_backout_plan*. Value must be a valid value in ServiceNow. Invalid entries are ignored.
- **Desired state of change request**: The gate would succeed and the pipeline continues when the change request status is the same as the provided value.
- **Advanced**: Specifies an expression that controls when this gate should succeed. The change request is defined as *root['result']* in the response from ServiceNow. Example - "and(eq(root['result'].state, 'New'),eq(root['result'].risk, 'Low'))". See Expressions for more details.
- **Output Variables** : You must specify a reference name to be able to use output variables in your deployment workflow. Gate variables can be accessed by using "PREDEPLOYGATE" as a "prefix" in **an agentless job**. For example when the reference name is set to 'gate1', the change number can be obtained as follows: *$(PREDEPLOYGATE.gate1.CHANGE_REQUEST_NUMBER)*.
- **CHANGE_REQUEST_NUMBER** : Number of the change request.
- **CHANGE_SYSTEM_ID** : System Id of the change request.

3. At the end of your release pipeline, add an **Agentless job** with a task **Update ServiceNow Change Request**.

- **ServiceNow connection**: Connection to the ServiceNow instance used for change management.
- **Change request number**: Number of the change request to update.
- **Updated status of change request** : Status to set for the change request. This input is available if *Update status* is selected.
- **Close code and Close notes**: Return status.

> ⓘ **Note**
>
> The Update ServiceNow Change Request task will fail if none of the change request fields are updated during execution. ServiceNow ignores invalid fields and values passed to the task.

Create a release pipeline

1. Select **Create release** to start a new release pipeline.

2. Your pipeline should create a new change request in ServiceNow as part of the pre-deployment conditions you created earlier.

3. The pipeline will wait for all the gates to succeed within the same sample interval. To check the change number, select the status icon to view your pipeline logs.

4. The change request will be queued in ServiceNow and can be viewed by the change owner.

5. The release pipeline that triggered the new change request can be found under the **Azure DevOps Pipeline metadata** section.

6. When the change is ready for implementation (moved to **Implement** state), the pipeline will resume execution and the gate status should return *succeeded*.

7. The change request will be closed automatically after deployment.

Yaml pipelines

This tutorial assumes you have a yaml pipeline with a single stage that deploys to a "latest" environment.

Add a check

1. Navigate to your environment "latest", select the ellipsis button and then select **Approvals and checks**.

2. Select the plus sign to add a new check, and then add the **ServiceNow Change Management** check to your environment. Use the same configuration you used for your pre-deployment gate.

Add the yaml task

1. Add a server job to your stage to update the change request.

```yaml
stages:
- stage: Stage1
  jobs:
  - deployment: DeployLatest
    environment: Latest
#   Rest of the tasks
  - job: UpdateChangeRequest
    steps:
    - task: UpdateServiceNowChangeRequest@2
      displayName: 'Update ServiceNow Change Request'
      inputs:
        ServiceNowConnection: 'snow06'
        NewStatus: 'Assess'
        WorkNotes: 'sample work notes'
        otherParameters: '{"u_hello": "hello"}'
pool: server
```

2. Save and run your pipeline. A new change request will be automatically created and the pipeline will pause and wait for the checks to complete.

3. Once the checks are completed, the pipeline should resume execution. The change request will be closed automatically after deployment.

FAQs

Q: What versions of ServiceNow are supported?

A: We support the following versions: Kingston, London, New York, Paris, Quebec, Rome, San Diego, and Tokyo.

Q: What types of change request are supported?

A: Normal, standard and emergency change requests are supported with this integration.

Q: How do I set additional change properties?

A: You can specify additional change properties from the **Additional change request parameters** field. Use a key-value pairs JSON format, with the name being the field name (not the label) prefixed with `u_`.

Q: Can I update custom fields in the change request with additional change request parameters?

A: If custom fields are defined in the change request, you must add mapping for custom fields in Import set transform map.

Q: I don't see drop-down values populated for Category, Status, and other fields. What should I do?

A: Change Management Core and Change Management - State Model plugins must be active on your ServiceNow instance for the drop-downs to work. See Upgrade change management and Update change request states for more details.

Resources

- Configure your release pipelines for safe deployments
- Twitter sentiment as a release gate
- GitHub issues as a release gate
- Author custom gates.
- ServerTaskHelper Library example

Related articles

- Release gates and approvals
- Define approvals and checks
- Set up manual intervention
- Use gates and approvals to control your deployment
- Add stages, dependencies, & conditions
- Release triggers

Migrate your Classic pipeline to YAML

Article • 07/03/2023

Azure DevOps Services

Get started with Azure Pipelines by converting your existing Classic pipeline to use YAML. With a YAML-based pipeline, you can implement your CI/CD strategy as code and see its history, compare versions, blame, annotate, and so on.

When you convert your Classic pipeline, the end product is two pipelines. You'll have one new YAML pipeline and a Classic pipeline that can be retired. Your Classic pipeline's run history will remain in the Classic pipeline.

Prerequisites

Make sure you have the following items before you begin.

- An Azure account with an active subscription. Create an account for free.
- An active Azure DevOps organization. Sign up for Azure Pipelines.
- A working pipeline that uses the Classic user interface (UI) editor.
- A sample YAML pipeline file in your code. Create a sample YAML pipeline in the following section.

Create a sample YAML pipeline

Do the following steps to create a sample YAML pipeline, which you'll update later with your exported code from the Classic UI editor.

1. Sign in to your organization (`https://dev.azure.com/{yourorganization}`) and select your project.

2. Select **Pipelines**, and then **New pipeline**.

3. Select the location for your source code as either **GitHub** or **Azure Repos Git**.

4. Select a repository.

5. On the Configure your pipeline page, select **Starter pipeline**.

6. Select **Save and run**.

7. Enter your commit message, select **Commit directly to the main branch**, and then choose **Save and run** again. A new run starts and it's committed to the repository. Wait for the run to finish.

Export your Classic pipeline

Do the following steps to export your Classic pipeline to a YAML file that you can use in the editor.

1. Open your Classic pipeline.

2. Select the ellipses (...), and then select **Export to YAML**.

3. Open the downloaded YAML file in your code editor.

4. If your YAML pipeline includes variables defined in the Classic UI, define the variables again in your pipeline settings UI or in your YAML file. For more information, see Define variables.

5. Review any `cron` schedules in your YAML file. By default, `cron` schedules in YAML are in UTC. In Classic pipelines, they are in the organization's timezone. For more information, see Configure schedules for pipelines.

6. Use the Task Assistant to make any other changes to the YAML file. The Task Assistant is a pane on the right side of the screen, which helps you correctly create and modify YAML steps.

7. **Save and run** your pipeline.

Clean up resources

If you're not going to use this sample pipeline anymore, delete it from your project. Deletion is permanent and includes all builds and associated artifacts.

1. Select the ellipses (...) and select **Delete**.

2. Enter the name of your pipeline to permanently delete it, and then select **Delete**.

FAQ

Is there a task in YAML pipelines to create work items when there's a build failure?

YAML pipelines don't have a Create work item on failure setting like classic build pipelines. You have a couple of options for creating this functionality yourself.

- You can use a script or PowerShell task and call the REST API.
- You can use Azure CLI to call az boards work-item create in your pipeline. See an example of using the CLI to create a bug on failure.

Next steps

Learn about the feature differences between YAML and Classic pipelines.

Use Azure Pipelines

Related articles

- Customize your pipeline
- YAML pipeline editor basics
- Library of assets
- Define approvals and checks

Migrate from Jenkins to Azure Pipelines

Article • 05/30/2023

Azure DevOps Services

Jenkins, an open-source automation server, has traditionally been installed by enterprises in their own data centers and managed on-premises. Many providers also offer managed Jenkins hosting.

Alternatively, Azure Pipelines is a cloud native continuous integration pipeline, providing the management of multi-stage pipelines and build agent virtual machines hosted in the cloud.

Azure Pipelines offers a fully on-premises option as well with Azure DevOps Server, for those customers who have compliance or security concerns that require them to keep their code and build within the enterprise data center.

In addition, Azure Pipelines supports hybrid cloud and on-premises models. Azure Pipelines can manage build and release orchestration and enabling build agents, both in the cloud and installed on-premises.

This document provides a guide to translate a Jenkins pipeline configuration to Azure Pipelines, information about moving container-based builds and selecting build agents, mapping environment variables, and how to handle success and failures of the build pipeline.

Configuration

You'll find a familiar transition from a Jenkins declarative pipeline into an Azure Pipelines YAML configuration. The two are conceptually similar, supporting "configuration as code" and allowing you to check your configuration into your version control system. Unlike Jenkins, however, Azure Pipelines uses the industry-standard YAML to configure the build pipeline.

The concepts between Jenkins and Azure Pipelines and the way they're configured are similar. A Jenkinsfile lists one or more *stages* of the build process, each of which contains one or more *steps* that are performed in order. For example, a "build" stage may run a task to install build-time dependencies, then perform a compilation step. While a "test" stage may invoke the test harness against the binaries that were produced in the build stage.

For example:

Jenkinsfile

```
pipeline {
    agent none
    stages {
        stage('Build') {
            steps {
                sh 'npm install'
                sh 'npm run build'
            }
        }
        stage('Test') {
            steps {
                sh 'npm test'
            }
        }
    }
}
```

The jenkinsfile translates easily to an Azure Pipelines YAML configuration, with a *job* corresponding to each stage, and steps to perform in each job:

azure-pipelines.yml

YAML

```yaml
jobs:
- job: Build
  steps:
  - script: npm install
  - script: npm run build
- job: Test
  steps:
  - script: npm test
```

Visual Configuration

If you aren't using a Jenkins declarative pipeline with a Jenkinsfile, and are instead using the graphical interface to define your build configuration, then you may be more comfortable with the classic editor in Azure Pipelines.

Container-Based Builds

Using containers in your build pipeline allows you to build and test within a docker image that has the exact dependencies that your pipeline needs, already configured. It

saves you from having to include a build step that installs more software or configures the environment. Both Jenkins and Azure Pipelines support container-based builds.

In addition, both Jenkins and Azure Pipelines allow you to share the build directory on the host agent to the container volume using the `-v` flag to docker. This allows you to chain multiple build jobs together that can use the same sources and write to the same output directory. This is especially useful when you use many different technologies in your stack; you may want to build your backend using a .NET Core container and your frontend with a TypeScript container.

For example, to run a build in an Ubuntu 20.04 ("Focal") container, then run tests in an Ubuntu 22.04 ("Jammy") container:

Jenkinsfile

```
pipeline {
    agent none
    stages {
        stage('Build') {
            agent {
                docker {
                    image 'ubuntu:focal'
                    args '-v $HOME:/build -w /build'
                }
            }
            steps {
                sh 'make'
            }
        }
        stage('Test') {
            agent {
                docker {
                    image 'ubuntu:jammy'
                    args '-v $HOME:/build -w /build'
                }
            }
            steps {
                sh 'make test'
            }
        }
    }
}
```

Azure Pipelines provides container jobs to enable you to run your build within a container:

azure-pipelines.yml

YAML

```yaml
resources:
  containers:
  - container: focal
    image: ubuntu:focal
  - container: jammy
    image: ubuntu:jammy

jobs:
- job: build
  container: focal
  steps:
  - script: make
- job: test
  dependsOn: build
  container: jammy
  steps:
  - script: make test
```

In addition, Azure Pipelines provides a docker task that allows you to run, build, or push an image.

Agent Selection

Jenkins offers build agent selection using the `agent` option to ensure that your build pipeline - or a particular stage of the pipeline - runs on a particular build agent machine. Similarly, Azure Pipelines offers many options to configure where your build environment runs.

Hosted Agent Selection

Azure Pipelines offers cloud hosted build agents for Linux, Windows, and macOS builds. To select the build environment, you can use the vmimage keyword. For example, to select a macOS build:

YAML

```yaml
pool:
  vmimage: macOS-latest
```

Additionally, you can specify a `container` and specify a docker image for finer grained control over how your build is run.

On-premises Agent Selection

If you host your build agents on-premises, then you can define the build agent "capabilities" based on the architecture of the machine or the software that you've installed on it. For example, if you've set up an on-premises build agent with the `java` capabilities, then you can ensure that your job runs on it using the `demands` keyword:

YAML

```yaml
pool:
  demands: java
```

Environment Variables

In Jenkins, you typically define environment variables for the entire pipeline. For example, to set two environment variables, `CONFIGURATION=debug` and `PLATFORM=x86`:

Jenkinsfile

```
pipeline {
    environment {
        CONFIGURATION = 'debug'
        PLATFORM      = 'x64'
    }
}
```

Similarly, in Azure Pipelines you can configure variables that are used both within the YAML configuration and are set as environment variables during job execution:

azure-pipelines.yml

YAML

```yaml
variables:
  configuration: debug
  platform: x64
```

Additionally, in Azure Pipelines you can define variables that are set only during a particular job:

azure-pipelines.yml

YAML

```yaml
jobs:
- job: debug build
  variables:
    configuration: debug
  steps:
  - script: ./build.sh $(configuration)
- job: release build
  variables:
    configuration: release
  steps:
  - script: ./build.sh $(configuration)
```

Predefined Variables

Both Jenkins and Azure Pipelines set a number of environment variables to allow you to inspect and interact with the execution environment of the continuous integration system.

Description	Jenkins	Azure Pipelines
A unique numeric identifier for the current build invocation.	`BUILD_NUMBER`	`BUILD_BUILDNUMBER`
A unique identifier (not necessarily numeric) for the current build invocation.	`BUILD_ID`	`BUILD_BUILDID`
The URL that displays the build logs.	`BUILD_URL`	This isn't set as an environment variable in Azure Pipelines but can be derived from other variables.[1]
The name of the machine that the current build is running on.	`NODE_NAME`	`AGENT_NAME`
The name of this project or build definition.	`JOB_NAME`	`RELEASE_DEFINITIONNAME`
A string for identification of the build; the build number is a good unique identifier.	`BUILD_TAG`	`BUILD_BUILDNUMBER`
A URL for the host executing the build.	`JENKINS_URL`	`SYSTEM_TEAMFOUNDATIONCOLLECTIONURI`
A unique identifier for the build executor or build agent that is currently running.	`EXECUTOR_NUMBER`	`AGENT_NAME`

Description	Jenkins	Azure Pipelines
The location of the checked out sources.	`WORKSPACE`	`BUILD_SOURCESDIRECTORY`
The Git Commit ID corresponding to the version of software being built.	`GIT_COMMIT`	`BUILD_SOURCEVERSION`
Path to the Git repository on GitHub, Azure Repos or another repository provider.	`GIT_URL`	`BUILD_REPOSITORY_URI`
The Git branch being built.	`GIT_BRANCH`	`BUILD_SOURCEBRANCH`

[1] To derive the URL that displays the build logs in Azure Pipelines, combine the following environment variables in this format:

```
${SYSTEM_TEAMFOUNDATIONCOLLECTIONURI}/${SYSTEM_TEAMPROJECT}/_build/results?buildId=${BUILD_BUILDID}
```

Success and Failure Handling

Jenkins allows you to run commands when the build has finished, using the `post` section of the pipeline. You can specify commands that run when the build succeeds (using the `success` section), when the build fails (using the `failure` section) or always (using the `always` section). For example:

Jenkinsfile

```
post {
    always {
        echo "The build has finished"
    }
    success {
        echo "The build succeeded"
    }
    failure {
        echo "The build failed"
    }
}
```

Similarly, Azure Pipelines has a rich conditional execution framework that allows you to run a job, or steps of a job, based on many conditions including pipeline success or failure.

To emulate Jenkins `post`-build conditionals, you can define jobs that run based on the `always()`, `succeeded()` or `failed()` conditions:

azure-pipelines.yml

YAML

```yaml
jobs:
- job: always
  steps:
  - script: echo "The build has finished"
    condition: always()

- job: success
  steps:
  - script: echo "The build succeeded"
    condition: succeeded()

- job: failed
  steps:
  - script: echo "The build failed"
    condition: failed()
```

In addition, you can combine other conditions, like the ability to run a task based on the success or failure of an individual task, environment variables, or the execution environment, to build a rich execution pipeline.

Migrate from Travis to Azure Pipelines

Article • 03/02/2023

Azure DevOps Services

This purpose of this guide is to help you migrate from Travis to Azure Pipelines. This guide describes shows how to translate from a Travis configuration to an Azure Pipelines configuration.

> We need your help to make this guide better! Submit comments or contribute your changes directly.

Key differences

There are many differences between Travis and Azure Pipelines, including:

- Travis builds have *stages*, *jobs* and *phases*, while Azure Pipelines has steps that can be arranged and executed in an arbitrary order or grouping that you choose.
- Azure Pipelines allows job definitions and steps to be stored in separate YAML files in the same or a different repository, enabling steps to be shared across multiple pipelines.
- Azure Pipelines provides full support for building and testing on Microsoft-managed Linux, Windows, and macOS images. For more information about hosted agents, see Microsoft-hosted agents.

Prerequisites

- A GitHub account where you can create a repository. Create one for free.
- An Azure DevOps organization. Create one for free. If your team already has one, then make sure you're an administrator of the Azure DevOps project that you want to use.
- An ability to run pipelines on Microsoft-hosted agents. You can either purchase a parallel job or you can request a free tier.
- Basic knowledge of Azure Pipelines. If you're new to Azure Pipelines, see the following to learn more about Azure Pipelines and how it works prior to starting your migration:
  - Create your first pipeline
  - Key concepts for new Azure Pipelines users

Language

Travis uses the `language` keyword to identify the prerequisite build environment to set up for your build. For example, to select Node.JS 16.x:

.travis.yml

```yaml
language: node_js
node_js:
  - 16
```

Microsoft-hosted agents contain the SDKs for many languages by default. To use a specific language version, you may need to use a language selection task to set up the environment.

For example, to select Node.JS 16.x:

azure-pipelines.yml

```yaml
steps:
- task: NodeTool@0
  inputs:
    versionSpec: '16.x'
```

Language mappings

The `language` keyword in Travis implies both that version of language tools be used and that many build steps be implicitly performed. In Azure Pipelines, you need to specify the commands that you want to run.

Here's a translation guide from the `language` keyword to the commands that are executed automatically for the most commonly used languages:

Language	Commands

`c`
`cpp`
	`./configure`
	`make`
	`make install`

`csharp`	`nuget restore [solution.sln]` `msbuild /p:Configuration=Release [solution.sln]`

Language	Commands
`clojure`	`lein deps` `lein test`
`go`	`go get -t -v ./...` `make` or `go test`

`java`
`groovy`
	Gradle:
	`gradle assemble`
	`gradle check`
	
	Maven:
	`mvn install -DskipTests=true -Dmaven.javadoc.skip=true -B -V`
	`mvn test -B`
	
	Ant:
	`ant test`

`node_js`	`npm install` or `npm ci` or `yarn` `npm test`
`objective-c` `swift`	`pod install` or `bundle exec pod install` `xcodebuild -scheme [scheme] build test | xcpretty`

`perl`	`cpanm --quiet --installdeps --notest .`
	
	Build.PL:
	`perl ./Build.pl`
	`./Build test`
	
	Makefile.PL:
	`perl Makefile.PL`
	`make test`
	
	Makefile:
	`make test`

`php`	`phpunit`
`python`	`pip install -r requirements.txt`
`ruby`	`bundle install --jobs=3 --retry=3` `rake`

In addition, less common languages can be enabled but require another dependency installation step or execution inside a docker container:

Language	Commands
`crystal`	`docker run -v $(pwd):/src -w /src crystallang/crystal shards install` `docker run -v $(pwd):/src -w /src crystallang/crystal crystal spec`

`d`	`sudo wget http://master.dl.sourceforge.net/project/d-apt/files/d-apt.list -O /etc/apt/sources.list.d/d-apt.list`
	`sudo apt-get update`
	`sudo apt-get -y --allow-unauthenticated install --reinstall d-apt-keyring`
	`sudo apt-get update`
	`sudo apt-get install dmd-compiler dub`
	`dub test --compiler=dmd`

`dart`	`wget https://dl-ssl.google.com/linux/linux_signing_key.pub -O - | sudo apt-key add -`
	`wget https://storage.googleapis.com/download.dartlang.org/linux/debian/dart_stable.list -O /etc/apt/sources.list.d/dart_stable.list`
	`sudo apt-get update`
	`sudo apt-get install dart`
	`/usr/lib/dart/bin/pub get`
	`/usr/lib/dart/bin/pub run test`

`erlang`	`sudo apt-get install rebar`
	`rebar get-deps`
	`rebar compile`
	`rebar skip_deps=true eunit`

`elixir`	`sudo apt-get install elixir`
	`mix local.rebar --force`
	`mix local.hex --force`
	`mix deps.get`
	`mix test`

`haskell`	`sudo apt-get install cabal-install`
	`cabal install --only-dependencies --enable-tests`
	`cabal configure --enable-tests`
	`cabal build`
	`cabal test`

`haxe`	`sudo apt-get install haxe`
	`yes | haxelib install [hxml]`
	`haxe [hxml]`

`julia`	`sudo apt-get install julia`
	`julia -e "using Pkg; Pkg.build(); end"`
	`julia --check-bounds=yes -e "Pkg; Pkg.test(coverage=true); end"`

`nix`	`docker run -v $(pwd):/src -w /src nixos/nix nix-build`

Language	Commands
`perl6`	`sudo apt-get install rakudo` `PERL6LIB=lib prove -v -r --exec=perl6 t/`

`rust`	`curl -sSf https://sh.rustup.rs | sh -s -- -y`
	`cargo build --verbose`
	`cargo test --verbose`

`scala`	`echo "deb https://repo.scala-sbt.org/scalasbt/debian /" | /etc/apt/sources.list.d/sbt.list`
	`sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823`
	`sudo apt-get update`
	`sudo apt-get install sbt`
	`sbt ++2.11.6 test`

`smalltalk`	`docker run -v $(pwd):/src -w /src hpiswa/smalltalkci smalltalkci`

Multiple language selections

You can also configure an environment that supports building different applications in multiple languages. For example, to ensure the build environment targets both Node.JS 16.x and Ruby 3.2 or better:

azure-pipelines.yml

YAML

```yaml
steps:
- task: NodeTool@0
  inputs:
    versionSpec: '8.x'
- task: UseRubyVersion@0
  inputs:
    versionSpec: '>= 3.2'
```

Phases

In Travis, steps are defined in a fixed set of named phases such as `before_install` or `before_script`. Azure Pipelines doesn't have named phases and steps can be grouped, named, and organized in whatever way makes sense for the pipeline.

For example:

.travis.yml

```yaml
before_install:
    - npm install -g bower
install:
    - npm install
    - bower install
script:
    - npm run build
    - npm test
```

azure-pipelines.yml

```yaml
steps:
- script: npm install -g bower
- script: npm install
- script: bower install
- script: npm run build
- script: npm test
```

Alternatively, steps can be grouped together and optionally named:

azure-pipelines.yml

```yaml
steps:
- script: |
    npm install -g bower
    npm install
    bower install
  displayName: 'Install dependencies'
- script: npm run build
- script: npm test
```

Parallel jobs

Travis provides parallelism by letting you define a stage, which is a group of jobs that are executed in parallel. A Travis build can have multiple stages; once all jobs in a stage have completed, the next stage starts.

With Azure Pipelines, you can make each step or stage dependent on any other step. In this way, you specify which steps run serially, and which can run in parallel. So you can

fan out with multiple steps run in parallel after the completion of one step, and then fan back in with a single step that runs afterward. This model gives you options to define complex workflows if necessary. For now, here's a simple example:

For example, to run a build script, then upon its completion run both the unit tests and the integration tests in parallel, and once all tests have finished, package the artifacts and then run the deploy to pre-production:

.travis.yml

YAML

```yaml
jobs:
  include:
  - stage: build
    script: ./build.sh
  - stage: test
    script: ./test.sh unit_tests
  - script: ./test.sh integration_tests
  - stage: package
    script: ./package.sh
  - stage: deploy
    script: ./deploy.sh pre_prod
```

azure-pipelines.yml

YAML

```yaml
jobs:
- job: build
  steps:
  - script: ./build.sh
- job: test1
  dependsOn: build
  steps:
  - script: ./test.sh unit_tests
- job: test2
```

```
  dependsOn: build
  steps:
  - script: ./test.sh integration_tests
- job: package
  dependsOn:
  - test1
  - test2
  script: ./package.sh
- job: deploy
  dependsOn: package
  steps:
  - script: ./deploy.sh pre_prod
```

Advanced parallel execution

In Azure Pipelines you have more options and control over how you orchestrate your pipeline.

For example, a team has a set of fast-running unit tests, and another set of and slower integration tests. The team wants to begin creating the .ZIP file for a release as soon as the unit are completed because they provide high confidence that the build provides a good package. But before they deploy to pre-production, they want to wait until all tests have passed:

In Azure Pipelines they can do it this way:

azure-pipelines.yml

YAML

```
jobs:
- job: build
  steps:
  - script: ./build.sh
- job: test1
  dependsOn: build
```

```
  steps:
  - script: ./test.sh unit_tests
- job: test2
  dependsOn: build
  steps:
  - script: ./test.sh integration_tests
- job: package
  dependsOn: test1
  script: ./package.sh
- job: deploy
  dependsOn:
  - test1
  - test2
  - package
  steps:
  - script: ./deploy.sh pre_prod
```

Step reuse

In Travis you can use matrices to run multiple executions across a single configuration. In Azure Pipelines you can use matrices in the same way, but you can also implement configuration reuse with templates.

Example: Environment variable in a matrix

One of the most common ways to run several builds with a slight variation is to change the execution using environment variables. For example, your build script can look for the presence of an environment variable and change the way your software is built, or the way it's tested.

You can use a matrix to have run a build configuration several times, once for each value in the environment variable. For example, to run a given script three times, each time with a different setting for an environment variable:

.travis.yml

YAML

```yaml
os: osx
env:
  matrix:
  - MY_ENVIRONMENT_VARIABLE: 'one'
  - MY_ENVIRONMENT_VARIABLE: 'two'
  - MY_ENVIRONMENT_VARIABLE: 'three'
script: echo $MY_ENVIRONMENT_VARIABLE
```

azure-pipelines.yml

```yaml
pool:
  vmImage: 'macOS-latest'
strategy:
  matrix:
    set_env_to_one:
      MY_ENVIRONMENT_VARIABLE: 'one'
    set_env_to_two:
      MY_ENVIRONMENT_VARIABLE: 'two'
    set_env_to_three:
      MY_ENVIRONMENT_VARIABLE: 'three'
steps:
- script: echo $(MY_ENVIRONMENT_VARIABLE)
```

Example: Language versions in a matrix

Another common scenario is to run against several different language environments. Travis supports an implicit definition using the `language` keyword, while Azure Pipelines expects an explicit task to define how to configure that language version.

You can use the environment variable matrix options in Azure Pipelines to enable a matrix for different language versions. For example, you can set an environment variable in each matrix variable that corresponds to the language version that you want to use, then in the first step, use that environment variable to run the language configuration task:

.travis.yml

```yaml
os: linux
matrix:
  include:
  - rvm: 2.3.7
  - rvm: 2.4.4
  - rvm: 2.5.1
script: ruby --version
```

azure-pipelines.yml

```yaml
vmImage: 'ubuntu-latest'
strategy:
```

```
  matrix:
    ruby 2.3:
      ruby_version: '2.3.7'
    ruby 2.4:
      ruby_version: '2.4.4'
    ruby 2.5:
      ruby_version: '2.5.1'
steps:
- task: UseRubyVersion@0
  inputs:
    versionSpec: $(ruby_version)
- script: ruby --version
```

Example: Operating systems within a matrix

It's also common to run builds in multiple operating systems. Travis supports this definition using the `os` keyword, while Azure Pipelines lets you configure the operating system by selecting the pool to run in using the `vmImage` keyword.

For example, you can set an environment variable in each matrix variable that corresponds to the operating system image that you want to use. Then you can set the machine pool to the variable you've set:

.travis.yml

YAML

```yaml
matrix:
  include:
  - os: linux
  - os: windows
  - os: osx
script: echo Hello, world!
```

azure-pipelines.yml

YAML

```yaml
strategy:
  matrix:
    linux:
      imageName: 'ubuntu-latest'
    mac:
      imageName: 'macOS-latest'
    windows:
      imageName: 'windows-latest'

pool:
```

```
  vmImage: $(imageName)

steps:
- script: echo Hello, world!
```

Success and failure handling

Travis allows you to specify steps that run when the build succeeds, using the `after_success` phase, or when the build fails, using the `after_failure` phase. With Azure Pipelines you can define success and failure conditions based on the result of any step, which enables more flexible and powerful pipelines.

.travis.yml

YAML

```yaml
build: ./build.sh
after_success: echo Success
after_failure: echo Failed
```

azure-pipelines.yml

YAML

```yaml
steps:
- script: ./build.sh
- script: echo Success
  condition: succeeded()
- script: echo Failed
  condition: failed()
```

Advanced success and failure handling

In Azure Pipelines you can program a flexible set of dependencies and conditions for flow control between jobs.
You can configure jobs to run based on the success or failure of previous jobs or based on environment variables. You can even configure jobs to always run, regardless of the success of other jobs.

For example, if you want to run a script when the build fails, but only if it's running as a build on the main branch:

azure-pipelines.yml

YAML

```yaml
jobs:
- job: build
  steps:
  - script: ./build.sh
- job: alert
  dependsOn: build
  condition: and(failed(), eq(variables['Build.SourceBranch'],
'refs/heads/main'))
  steps:
  - script: ./sound_the_alarms.sh
```

Predefined variables

Both Travis and Azure Pipelines set multiple environment variables to allow you to inspect and interact with the execution environment of the CI system.

In most cases, there's an Azure Pipelines variable to match the environment variable in Travis. Here's a list of commonly used environment variables in Travis and their analog in Azure Pipelines:

Travis	Azure Pipelines	Description
`CI=true` or `TRAVIS=true`	`TF_BUILD=True`	Indicates that your build is running in the CI system; useful for scripts that are also intended to be run locally during development.
`TRAVIS_BRANCH`	**CI builds**: `BUILD_SOURCEBRANCH` **Pull request builds**: `SYSTEM_PULLREQUEST_TARGETBRANCH`	The name of the branch the build was queued for, or the name of the branch the pull request is targeting.
`TRAVIS_BUILD_DIR`	`BUILD_SOURCESDIRECTORY`	The location of your checked out source and the default working directory.
`TRAVIS_BUILD_NUMBER`	`BUILD_BUILDID`	A unique numeric identifier for the current build invocation.
`TRAVIS_COMMIT`	**CI builds**: `BUILD_SOURCEVERSION`	The commit ID currently being built.

Travis	Azure Pipelines	Description
`TRAVIS_COMMIT`	**Pull request builds**: `git rev-parse HEAD^2`	For pull request validation builds, Azure Pipelines sets `BUILD_SOURCEVERSION` to the resulting merge commit of the pull request into main; this command identifies the pull request commit itself.
`TRAVIS_COMMIT_MESSAGE`	`BUILD_SOURCEVERSIONMESSAGE`	The log message of the commit being built.
`TRAVIS_EVENT_TYPE`	`BUILD_REASON`	The reason the build was queued; a map of values is in the "build reasons" table below.
`TRAVIS_JOB_NAME`	`AGENT_JOBNAME`	The name of the current job, if specified.
`TRAVIS_OS_NAME`	`AGENT_OS`	The operating system that the job is running on; a map of values is in the "operating systems" table below.
`TRAVIS_PULL_REQUEST`	**Azure Repos**: `SYSTEM_PULLREQUEST_PULLREQUESTID` **GitHub**: `SYSTEM_PULLREQUEST_PULLREQUESTNUMBER`	The pull request number that triggered this build. (For GitHub builds, this is a unique identifier that is *not* the pull request number.)
`TRAVIS_PULL_REQUEST_BRANCH`	`SYSTEM_PULLREQUEST_SOURCEBRANCH`	The name of the branch where the pull request originated.
`TRAVIS_PULL_REQUEST_SHA`	**Pull request builds**: `git rev-parse HEAD^2`	For pull request validation builds, Azure Pipelines sets `BUILD_SOURCEVERSION` to the resulting merge commit of the pull request into main; this command identifies the pull request commit itself.
`TRAVIS_PULL_REQUEST_SLUG`		The name of the forked repository, if the pull request originated in a fork. There's no analog to this in Azure Pipelines.

Travis	Azure Pipelines	Description
`TRAVIS_REPO_SLUG`	`BUILD_REPOSITORY_NAME`	The name of the repository that this build is configured for.
`TRAVIS_TEST_RESULT`	`AGENT_JOBSTATUS`	Travis sets this value to `0` if all previous steps have succeeded (returned `0`). For Azure Pipelines, check that `AGENT_JOBSTATUS=Succeeded`.
`TRAVIS_TAG`	`BUILD_SOURCEBRANCH`	If this build was queued by the creation of a tag then this is the name of that tag. For Azure Pipelines, the `BUILD_SOURCEBRANCH` is set to the full Git reference name, for example, `refs/tags/tag_name`.
`TRAVIS_BUILD_STAGE_NAME`		The name of the stage in Travis. As we saw earlier, Azure Pipelines handles flow control using jobs. You can reference `AGENT_JOBNAME`.

Build Reasons:

The `TRAVIS_EVENT_TYPE` variable contains values that map to values provided by the Azure Pipelines `BUILD_REASON` variable:

Travis	Azure Pipelines	Description
`push`	`IndividualCI`	The build is a continuous integration build from a push.
`pull_request`	`PullRequest`	The build was queued to validate a pull request.
`api`	`Manual`	The build was queued by the REST API or a manual request on the web page.
`cron`	`Schedule`	The build was scheduled.

Operating Systems:

The `TRAVIS_OS_NAME` variable contains values that map to values provided by the Azure Pipelines `AGENT_OS` variable:

Travis	Azure Pipelines	Description
`linux`	`Linux`	The build is running on Linux.
`osx`	`Darwin`	The build is running on macOS.
`windows`	`Windows_NT`	The build is running on Windows.

To learn more, see Predefined environment variables.

If there isn't a variable for the data you need, then you can use a shell command to get it. For example, a good substitute of an environment variable containing the commit ID of the pull request being built is to run a git command: `git rev-parse HEAD^2`.

Building specific branches

By default, both Travis and Azure Pipelines perform CI builds on all branches. Similarly, both systems allow you to limit these builds to specific branches. In Azure Pipelines, the list of branches to build should be listed in the `include` list and the branches *not* to build should be listed in the `exclude list. Wildcards are supported.

For example, to build only the main branch and those that begin with the word "releases":

.travis.yml

YAML

```yaml
branches:
  only:
  - main
  - /^releases.*/
```

azure-pipelines.yml

YAML

```yaml
trigger:
  branches:
    include:
    - main
    - releases*
```

Output caching

Travis supports caching dependencies and intermediate build output to improve build times. Azure Pipelines doesn't support caching intermediate build output, but does offer integration with Azure Artifacts for dependency storage.

Git submodules

Travis and Azure Pipelines both clone git repos "recursively" by default. This means that submodules are cloned by the agent, which is useful since submodules usually contain dependencies. However, the extra cloning takes time, so if you don't need the dependencies then you can disable cloning submodules:

.travis.yml

```yaml
git:
  submodules: false
```

azure-pipelines.yml

```yaml
checkout:
  submodules: false
```

Create a virtual network isolated environment for build-deploy-test scenarios

Article • 04/05/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Network Virtualization provides ability to create multiple virtual networks on a shared physical network. Isolated virtual networks can be created using SCVMM Network Virtualization concepts. VMM uses the concept of logical networks and corresponding VM networks to create isolated networks of virtual machines.

- You can create an isolated network of virtual machines that span across different hosts in a host-cluster or a private cloud.
- You can have VMs from different networks residing in the same host machine and still be isolated from each other.
- You can define IP address from any IP pool of your choice for a VM Network.

See also: Hyper-V Network Virtualization Overview.

To create a virtual network isolated environment:

- Ensure you meet the prerequisite conditions described in this section.
- Set up Network Virtualization using SCVMM. This is a one-time setup task you don't need to repeat. Follow these steps.
- Decide on the network topology you want to use. You'll specify this when you create the virtual network. The options and steps are described in this section.
- Enable your build-deploy-test scenario as shown in these steps.
- You can perform a range of operations to manage VMs using SCVMM. For examples, see SCVMM deployment.

Prerequisites

- SCVMM Server 2012 R2 or later.
- Window 2012 R2 host machines with Hyper-V set up with at least two physical NICs attached.
- One NIC (perhaps external) with corporate network or Internet access.
- One NIC configured in Trunk Mode with a VLAN ID (such as 991) and routable IP subnets (such as 10.10.30.1/24). You network administrator can configure this.
- All Hyper-V hosts in the host group have the same VLAN ID. This host group will be used for your isolated networks.

Verify the setup is working correctly by following these steps:

1. Open an RDP session to each of the host machines and open an administrator PowerShell session.
2. Run the command `Get-NetVirtualizationProviderAddress`. This gets the provider address for the physical NIC configured in trunk mode with a VLAN ID.

```
Administrator: Windows PowerShell
PS C:\Windows\system32> Get-NetVirtualizationProviderAddress

ProviderAddress   : 10.10.30.175
InterfaceIndex    : 15
PrefixLength      : 24
VlanID            : 991
AddressState      : Preferred
MACAddress        : 001dd8b71f38
ManagedByCluster  : False

PS C:\Windows\system32>
```

3. Go to another host and open an administrator PowerShell session. Ping other machines using the command `ping -p <Provider address>`. This confirms all host machines are connected to a physical NIC in trunk mode with IPs routable across the host machines. If this test fails, contact your network administrator.

```
Administrator: Windows PowerShell
PS C:\Windows\system32> ping -p 10.10.30.175

Pinging 10.10.30.175 with 32 bytes of data:
Reply from 10.10.30.175: bytes=32 time<1ms TTL=128
Reply from 10.10.30.175: bytes=32 time<1ms TTL=128
Reply from 10.10.30.175: bytes=32 time<1ms TTL=128
Reply from 10.10.30.175: bytes=32 time<1ms TTL=128

Ping statistics for 10.10.30.175:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
PS C:\Windows\system32>
```

Back to list of tasks

Create a Network Virtualization layer in SCVMM

Setting up a network visualization layer in SCVMM includes creating logical networks, port profiles, logical switches, and adding the switches to the Hyper-V hosts.

Create logical networks

1. Log into the SCVMM admin console.

2. Go to **Fabric** -> **Networking** -> **Logical Networks** -> **Create new Logical Network**.

3. In the popup, enter an appropriate name and select **One Connected Network** -> **Allow new networks created on this logical network to use network virtualization**, then select **Next**.

4. Add a new **Network Site** and select the host group to which the network site will be scoped. Enter the VLAN ID used to configure physical NIC in the Hyper-V host group and the corresponding routable IP subnet(s). To assist tracking, change the network site name to one that is memorable.

5. Select **Next** and **Save**.

6. Create an IP pool for the new logical networks, enter a name, and select **Next**.

7. Select **Use and existing network site** and select **Next**. Enter the routable IP address range your network administrator configured for your VLAN and select **Next**. If you have multiple routable IP subnets associated with your VLAN, create an IP pool for each one.

8. Provide the gateway address. By default, you can use the first IP address in your subnet.

9. Select **Next** and leave the existing DNS and WINS settings. Complete the creation of the network site.

10. Now create another **Logical Network** for external Internet access, but this time select **One Connected network** -> **Create a VM network with same name to allow virtual machines to access this logical network directly** and then select **Next**.

11. Add a network site and select the same host group, but this time add the VLAN as `0`. This means the communication uses the default access mode NIC (Internet).

12. Select **Next** and **Save**.

13. The result should look like the following in your administrator console after creating the logical networks.

External	Fully compliant					
Network Virtualization	Fully compliant					
ip Pool	Fully compliant	10.10.30.0/24	10.10.30.1	10.10.30.254	247	247

Create port profiles

1. Go to **Fabric** -> **Networking** -> **Port profiles** and **Create Hyper-V port profile**.

2. Select **Uplink port profile** and select **Hyper-V Port** as the load-balancing algorithm, then select **Next**.

3. Select the Network Virtualization site created previously and choose the **Enable Hyper-V Network Virtualization** checkbox, then save the profile.

4. Now create another Hyper-V port profile for external logical network. Select **Uplink** mode and **Host default** as the load-balancing algorithm, then select **Next**.

5. Select the other network site to be used for external communication, but and this time don't enable network virtualization. Then save the profile.

Create logical switches

1. Go to **Fabric** -> **Networking** -> **Logical switches** and **Create Logical Switch**.

2. In the getting started wizard, select **Next** and enter a name for the switch, then select **Next**.

3. Select **Next** to open to **Uplink** tab. Select **Add uplink port profile** and add the network virtualization port profile you just created.

4. Select **Next** and save the logical switch.

5. Now create another logical switch for the external network for Internet communication. This time add the other uplink port profile you created for the external network.

Add logical switches to Hyper-V hosts

1. Go to **VM and Services** -> [Your host group] -> [each of the host machines in turn].

2. Right select and open the **Properties** -> **Virtual Switches** tab.

3. Select **New Virtual Switch** -> **New logical switch for network virtualization**. Assign the physical adapter you configured in trunk mode and select the network virtualization port profile.

4. Create another logical switch for external connectivity, assign the physical adapter used for external communication, and select the external port profile.

5. Do the same for all the Hyper-V hosts in the host group.

This is a one-time configuration for a specific host group of machines. After completing this setup, you can dynamically provision your isolated network of virtual machines using the **SCVMM extension** in TFS and Azure Pipelines builds and releases.

Back to list of tasks

Create the required virtual network topology

Isolated virtual networks can be broadly classified into three topologies.

Topology 1: AD-backed Isolated VMs

- A boundary VM with Internet/TFS connectivity.
- An Azure Pipelines/TFS deployment group agent installed on the boundary VM.
- An AD-DNS VM if you want to use a local Active Directory domain.
- Isolated app VMs where you deploy and test your apps.

Topology 2: Non-AD backed isolated VMs

- A boundary VM with Internet/TFS connectivity.
- An Azure Pipelines/TFS deployment group agent installed on the boundary VM.
- Isolated app VMs where you deploy and test your apps.

Topology 3: AD-backed non-isolated VMs

- A boundary VM with Internet/TFS connectivity.
- An Azure Pipelines/TFS deployment group agent installed on the boundary VM.
- An AD-DNS VM if you want to use a local Active Directory domain.
- App VMs that are also connected to the external network where you deploy and test your apps.

You can create any of the above topologies using the SCVMM extension, as shown in the following steps.

1. Open your TFS or Azure Pipelines instance and install the **SCVMM extension** if not already installed. For more information, see SCVMM deployment.

   > The **SCVMM task** provides a more efficient way capability to perform lab management operations using build and release pipelines. You can manage SCVMM environments, provision isolated virtual networks, and implement build-deploy-test scenarios.

2. In a build or release pipeline, add a new **SCVMM** task.

3. In the **SCVMM** task, select a service connection for the SCVMM server where you want to provision your virtual network and then select **New Virtual machines using Templates/Stored VMs and VHDs** to provision your VMs.

4. You can create VMs from templates, stored VMs, and VHD/VHDx. Choose the appropriate option and enter the VM names and corresponding source information.

5. In case of topologies **1** and **2**, leave the **VM Network name** empty, which will clear all the old VM networks present in the created VMs (if any). For topology **3**, you must provide information about the external VM network here.

6. Enter the **Cloud Name** of the host where you want to provision your isolated network. In case of private cloud, ensure the host machines added to the cloud are connected to the same logical and external switches as explained above.

7. Select the **Network Virtualization** option to create the virtualization layer.

8. Based on the topology you would like to create, decide if the network requires an Active Directory VM. For example, to create Topology **2** (AD-backed isolated network), you require an Active directory VM. Select the **Add Active Directory VM** checkbox, enter the AD VM name and the stored VM source. Also enter the static IP address configured in the AD VM source and the DNS suffix.

9. Enter the settings for the **VM Network** and subnet you want to create, and the backing-logical network you created in the previous section (Logical Networks). Ensure the VM network name is unique. If possible, append the release name for easier tracking later.

10. In the **Boundary Virtual Machine options** section, set **Create boundary VM for communication with Azure Pipelines/TFS**. This will be the entry point for external communication.

11. Enter the boundary VM name and the source template (the boundary VM source should always be a VM template), and enter name of the existing external VM network you created for external communication.

12. Provide details for configuring the boundary VM agent to communicate with Azure Pipelines/TFS. You can configure a deployment agent or an automation agent. This agent will be used for app deployments.

13. Ensure the agent name you provide is unique. This will be used as demand in succeeding job properties so that the correct agent will be selected. If you selected the deployment group agent option, this parameter is replaced by the value of the tag, which must also be unique.

14. Ensure the boundary VM template has the agent configuration files downloaded and saved in the VHD before the template is created. Use this path as the agent installation path above.

Enable your build-deploy-test scenario

1. Create a new job in your pipeline, after your network virtualization job.

2. Based on the boundary VM agent (deployment group agent or automation agent) that is created as part of your boundary VM provisioning, choose **Deployment group job** or **Agent job**.

3. In the job properties, select the appropriate deployment group or automation agent pool.

4. In the case of an automation pool, add a new **Demand** for **Agent.Name** value. Enter the unique name used in the network virtualization job. In the case of deployment group job, you must set the tag in the properties of the group machines.

5. Inside the job, add the tasks you require for deployment and testing.

6. After testing is completed, you can destroy the VMs by using the **Delete VM** task option.

Now you can create release from this release pipeline. Each release will dynamically provision your isolated virtual network and run your deploy and test tasks in the environment. You can find the test results in the release summary. After your tests are

completed, you can automatically decommission your environments. You can create as many environments as you need with just a select from **Azure Pipelines**.

Back to list of tasks

See also

- SCVMM deployment
- Hyper-V Network Virtualization Overview

FAQ

Help and support

- See our troubleshooting page
- Get advice on Stack Overflow, and feel free to post your questions, search for answers, or suggest a feature on our Azure DevOps Developer Community. Support page.

Azure Pipelines task reference

Article • 12/21/2023

A task performs an action in a pipeline. For example, a task can build an app, interact with Azure resources, install a tool, or run a test. Tasks are the building blocks for defining automation in a pipeline.

The articles in this section describe the built-in tasks for Azure Pipelines and specify the semantics for attributes that hold special meaning for each task.

Please refer to the YAML Reference for *steps.task* for details on the general attributes supported by tasks.

For how-tos and tutorials about authoring pipelines using tasks, including creating custom tasks, custom extensions, and finding tasks on the Visual Studio Marketplace, see Tasks concepts and Azure Pipelines documentation.

Important

To view the task reference for tasks available for your platform, make sure that you select the correct Azure DevOps version from the version selector which is located above the table of contents. Feature support differs depending on whether you are working from Azure DevOps Services or an on-premises version of Azure DevOps Server.
To learn which on-premises version you are using, see **Look up your Azure DevOps platform and version**.

Build tasks

Expand table

Task	Description
.NET Core DotNetCoreCLI@2 DotNetCoreCLI@1 DotNetCoreCLI@0	Build, test, package, or publish a dotnet application, or run a custom dotnet command.
Advanced Security AutoBuild AdvancedSecurity-Codeql-Autobuild@1	Attempts to build the repository by finding and building project files in the source folder.
Advanced Security Initialize CodeQL AdvancedSecurity-Codeql-Init@1	Initializes the CodeQL database in preparation for building.
Advanced Security Perform CodeQL analysis AdvancedSecurity-Codeql-Analyze@1	Finalizes the CodeQL database and runs the analysis queries.
Android Build AndroidBuild@1	AndroidBuild@1 is deprecated. Use Gradle.
Android Signing AndroidSigning@3 AndroidSigning@2 AndroidSigning@1	Sign and align Android APK files.
Ant Ant@1	Build with Apache Ant.
Azure IoT Edge AzureIoTEdge@2	Build and deploy an Azure IoT Edge image.
CMake CMake@1	Build with the CMake cross-platform build system.
Container Build ContainerBuild@0	Container Build Task.
Docker Docker@2 Docker@1 Docker@0	Build or push Docker images, login or logout, start or stop containers, or run a Docker command.
Docker Compose DockerCompose@0	Build, push or run multi-container Docker applications. Task can be used with Docker or Azure Container registry.

Task	Description
Download GitHub Nuget Packages DownloadGitHubNugetPackage@1	Restore your nuget packages using dotnet CLI.
Go Go@0	Get, build, or test a Go application, or run a custom Go command.
Gradle Gradle@3 Gradle@2 Gradle@1	Build using a Gradle wrapper script.
Grunt Grunt@0	Run the Grunt JavaScript task runner.
gulp gulp@1 gulp@0	Run the gulp Node.js streaming task-based build system.
Index sources and publish symbols PublishSymbols@2 PublishSymbols@1	Index your source code and publish symbols to a file share or Azure Artifacts symbol server.
Jenkins queue job JenkinsQueueJob@2	Queue a job on a Jenkins server.
Jenkins Queue Job JenkinsQueueJob@1	Queue a job on a Jenkins server.
Maven Maven@4 Maven@3 Maven@2 Maven@1	Build, test, and deploy with Apache Maven.
MSBuild MSBuild@1	Build with MSBuild.
Prepare Analysis Configuration SonarQubePrepare@5 SonarQubePrepare@4	Prepare SonarQube analysis configuration.
Publish Quality Gate Result SonarQubePublish@5 SonarQubePublish@4	Publish SonarQube's Quality Gate result on the Azure DevOps build result, to be used after the actual analysis.
Run Code Analysis SonarQubeAnalyze@5 SonarQubeAnalyze@4	Run scanner and upload the results to the SonarQube server.

Task	Description
Visual Studio build VSBuild@1	Build with MSBuild and set the Visual Studio version property.
Xamarin.Android XamarinAndroid@1	Build an Android app with Xamarin.
Xamarin.iOS XamariniOS@2 XamariniOS@1	Build an iOS app with Xamarin on macOS.
Xcode Xcode@5 Xcode@4	Build, test, or archive an Xcode workspace on macOS. Optionally package an app.
Xcode Build Xcode@3 Xcode@2	Build an Xcode workspace on macOS.
Xcode Package iOS XcodePackageiOS@0	Generate an .ipa file from Xcode build output using xcrun (Xcode 7 or below).

Deploy tasks

Expand table

Task	Description
App Center distribute AppCenterDistribute@3 AppCenterDistribute@2 AppCenterDistribute@1 AppCenterDistribute@0	Distribute app builds to testers and users via Visual Studio App Center.
ARM template deployment AzureResourceManagerTemplateDeployment@3	Deploy an Azure Resource Manager (ARM) template to all the deployment scopes.
Azure App Service Classic (Deprecated) AzureWebPowerShellDeployment@1	Create or update Azure App Service using Azure PowerShell.
Azure App Service deploy AzureRmWebAppDeployment@4 AzureRmWebAppDeployment@3 AzureRmWebAppDeployment@2	Deploy to Azure App Service a web, mobile, or API app using Docker, Java, .NET, .NET Core, Node.js, PHP, Python, or Ruby.
Azure App Service manage AzureAppServiceManage@0	Start, stop, restart, slot swap, slot delete, install site extensions or enable continuous monitoring for an Azure App Service.

Task	Description
Azure App Service Settings AzureAppServiceSettings@1	Update/Add App settings an Azure Web App for Linux or Windows.
Azure CLI AzureCLI@2 AzureCLI@1	Run Azure CLI commands against an Azure subscription in a PowerShell Core/Shell script when running on Linux agent or PowerShell/PowerShell Core/Batch script when running on Windows agent.
Azure CLI Preview AzureCLI@0	Run a Shell or Batch script with Azure CLI commands against an azure subscription.
Azure Cloud Service deployment AzureCloudPowerShellDeployment@2 AzureCloudPowerShellDeployment@1	Deploy an Azure Cloud Service.
Azure Container Apps Deploy AzureContainerApps@1 AzureContainerApps@0	An Azure DevOps Task to build and deploy Azure Container Apps.
Azure Database for MySQL deployment AzureMysqlDeployment@1	Run your scripts and make changes to your Azure Database for MySQL.
Azure file copy AzureFileCopy@5 AzureFileCopy@4 AzureFileCopy@3 AzureFileCopy@2 AzureFileCopy@1	Copy files to Azure Blob Storage or virtual machines.
Azure Function on Kubernetes AzureFunctionOnKubernetes@1 AzureFunctionOnKubernetes@0	Deploy Azure function to Kubernetes cluster.
Azure Functions Deploy AzureFunctionApp@2 AzureFunctionApp@1	Update a function app with .NET, Python, JavaScript, PowerShell, Java based web applications.
Azure Functions for container AzureFunctionAppContainer@1	Update a function app with a Docker container.
Azure Key Vault AzureKeyVault@2 AzureKeyVault@1	Download Azure Key Vault secrets.
Azure Monitor alerts (Deprecated) AzureMonitorAlerts@0	Configure alerts on available metrics for an Azure resource (Deprecated).
Azure PowerShell AzurePowerShell@5	Run a PowerShell script within an Azure environment.

Task	Description
AzurePowerShell@4 AzurePowerShell@3 AzurePowerShell@2 AzurePowerShell@1	
Azure resource group deployment AzureResourceGroupDeployment@2	Deploy an Azure Resource Manager (ARM) template to a resource group and manage virtual machines.
Azure Resource Group Deployment AzureResourceGroupDeployment@1	Deploy, start, stop, delete Azure Resource Groups.
Azure Spring Apps AzureSpringCloud@0	Deploy applications to Azure Spring Apps and manage deployments.
Azure SQL Database deployment SqlAzureDacpacDeployment@1	Deploy an Azure SQL Database using DACPAC or run scripts using SQLCMD.
Azure VM scale set deployment AzureVmssDeployment@0	Deploy a virtual machine scale set image.
Azure Web App AzureWebApp@1	Deploy an Azure Web App for Linux or Windows.
Azure Web App for Containers AzureWebAppContainer@1	Deploy containers to Azure App Service.
Build machine image PackerBuild@1 PackerBuild@0	Build a machine image using Packer, which may be used for Azure Virtual machine scale set deployment.
Check Azure Policy compliance AzurePolicyCheckGate@0	Security and compliance assessment for Azure Policy.
Chef Chef@1	Deploy to Chef environments by editing environment attributes.
Chef Knife ChefKnife@1	Run scripts with Knife commands on your Chef workstation.
Copy files over SSH CopyFilesOverSSH@0	Copy files or build artifacts to a remote machine over SSH.
Deploy to Kubernetes KubernetesManifest@1 KubernetesManifest@0	Use Kubernetes manifest files to deploy to clusters or even bake the manifest files to be used for deployments using Helm charts.
IIS web app deploy IISWebAppDeploymentOnMachineGroup@0	Deploy a website or web application using Web Deploy.
IIS Web App deployment (Deprecated) IISWebAppDeployment@1	Deploy using MSDeploy, then create/update websites and app pools.

Task	Description
IIS web app manage IISWebAppManagementOnMachineGroup@0	Create or update websites, web apps, virtual directories, or application pools.
Invoke REST API InvokeRESTAPI@1 InvokeRESTAPI@0	Invoke a REST API as a part of your pipeline.
Kubectl Kubernetes@1 Kubernetes@0	Deploy, configure, update a Kubernetes cluster in Azure Container Service by running kubectl commands.
Manual intervention ManualIntervention@8	Pause deployment and wait for manual intervention.
Manual validation ManualValidation@0	[PREVIEW] Pause a pipeline run to wait for manual interaction. Works only with YAML pipelines.
MySQL database deploy MysqlDeploymentOnMachineGroup@1	Run scripts and make changes to a MySQL Database.
Package and deploy Helm charts HelmDeploy@0	Deploy, configure, update a Kubernetes cluster in Azure Container Service by running helm commands.
PowerShell on target machines PowerShellOnTargetMachines@3	Execute PowerShell scripts on remote machines using PSSession and Invoke-Command for remoting.
PowerShell on Target Machines PowerShellOnTargetMachines@2 PowerShellOnTargetMachines@1	Execute PowerShell scripts on remote machine(s).
Service Fabric application deployment ServiceFabricDeploy@1	Deploy an Azure Service Fabric application to a cluster.
Service Fabric Compose deploy ServiceFabricComposeDeploy@0	Deploy a Docker Compose application to an Azure Service Fabric cluster.
SQL Server database deploy SqlDacpacDeploymentOnMachineGroup@0	Deploy a SQL Server database using DACPAC or SQL scripts.
SQL Server database deploy (Deprecated) SqlServerDacpacDeployment@1	Deploy a SQL Server database using DACPAC.
SSH SSH@0	Run shell commands or a script on a remote machine using SSH.
Windows machine file copy WindowsMachineFileCopy@2	Copy files to remote Windows machines.

Task	Description
WindowsMachineFileCopy@1	

Package tasks

Expand table

Task	Description
Cargo authenticate (for task runners) CargoAuthenticate@0	Authentication task for the cargo client used for installing Cargo crates distribution.
CocoaPods CocoaPods@0	Install CocoaPods dependencies for Swift and Objective-C Cocoa projects.
Conda environment CondaEnvironment@1 CondaEnvironment@0	This task is deprecated. Use `conda` directly in script to work with Anaconda environments.
Download Github Npm Package DownloadGithubNpmPackage@1	Install npm packages from GitHub.
Maven Authenticate MavenAuthenticate@0	Provides credentials for Azure Artifacts feeds and external maven repositories.
npm Npm@1 Npm@0	Install and publish npm packages, or run an npm command. Supports npmjs.com and authenticated registries like Azure Artifacts.
npm authenticate (for task runners) npmAuthenticate@0	Don't use this task if you're also using the npm task. Provides npm credentials to an .npmrc file in your repository for the scope of the build. This enables npm task runners like gulp and Grunt to authenticate with private registries.
NuGet NuGetCommand@2	Restore, pack, or push NuGet packages, or run a NuGet command. Supports NuGet.org and authenticated feeds like Azure Artifacts and MyGet. Uses NuGet.exe and works with .NET Framework apps. For .NET Core and .NET Standard apps, use the .NET Core task.
NuGet authenticate NuGetAuthenticate@1 NuGetAuthenticate@0	Configure NuGet tools to authenticate with Azure Artifacts and other NuGet repositories. Requires NuGet >= 4.8.5385, dotnet >= 6, or MSBuild >= 15.8.166.59604.
NuGet command NuGet@0	Deprecated: use the “NuGet” task instead. It works with the new Tool Installer framework so you can easily use new versions of NuGet without waiting for a task update, provides

Task	Description
	better support for authenticated feeds outside this organization/collection, and uses NuGet 4 by default.
NuGet Installer NuGetInstaller@0	Installs or restores missing NuGet packages. Use NuGetAuthenticate@0 task for latest capabilities.
NuGet packager NuGetPackager@0	Deprecated: use the “NuGet” task instead. It works with the new Tool Installer framework so you can easily use new versions of NuGet without waiting for a task update, provides better support for authenticated feeds outside this organization/collection, and uses NuGet 4 by default.
NuGet publisher NuGetPublisher@0	Deprecated: use the “NuGet” task instead. It works with the new Tool Installer framework so you can easily use new versions of NuGet without waiting for a task update, provides better support for authenticated feeds outside this organization/collection, and uses NuGet 4 by default.
NuGet Restore NuGetRestore@1	Restores NuGet packages in preparation for a Visual Studio Build step.
PyPI publisher PyPIPublisher@0	Create and upload an sdist or wheel to a PyPI-compatible index using Twine.
Python pip authenticate PipAuthenticate@1 PipAuthenticate@0	Authentication task for the pip client used for installing Python distributions.
Python twine upload authenticate TwineAuthenticate@1 TwineAuthenticate@0	Authenticate for uploading Python distributions using twine. Add '-r FeedName/EndpointName --config-file $(PYPIRC_PATH)' to your twine upload command. For feeds present in this organization, use the feed name as the repository (-r). Otherwise, use the endpoint name defined in the service connection.
Universal packages UniversalPackages@0	Download or publish Universal Packages.
Xamarin Component Restore XamarinComponentRestore@0	This task is deprecated. Use 'NuGet' instead.

Test tasks

Expand table

Task	Description
App Center test AppCenterTest@1	Test app packages with Visual Studio App Center.
Azure Load Testing AzureLoadTest@1	Automate performance regression testing with Azure Load Testing.
Container Structure Test ContainerStructureTest@0	Uses container-structure-test (https://github.com/GoogleContainerTools/container-structure-test) to validate the structure of an image based on four categories of tests - command tests, file existence tests, file content tests and metadata tests.
Mobile Center Test VSMobileCenterTest@0	Test mobile app packages with Visual Studio Mobile Center.
Publish code coverage results PublishCodeCoverageResults@2 PublishCodeCoverageResults@1	Publish any of the code coverage results from a build.
Publish test results PublishTestResults@1	Publish test results to Azure Pipelines.
Publish Test Results PublishTestResults@2	Publish test results to Azure Pipelines.
Run functional tests RunVisualStudioTestsusingTestAgent@1	Deprecated: This task and it's companion task (Visual Studio Test Agent Deployment) are deprecated. Use the 'Visual Studio Test' task instead. The VSTest task can run unit as well as functional tests. Run tests on one or more agents using the multi-agent job setting. Use the 'Visual Studio Test Platform' task to run tests without needing Visual Studio on the agent. VSTest task also brings new capabilities such as automatically rerunning failed tests.
Visual Studio Test VSTest@2 VSTest@1	Run unit and functional tests (Selenium, Appium, Coded UI test, etc.) using the Visual Studio Test (VsTest) runner. Test frameworks that have a Visual Studio test adapter such as MsTest, xUnit, NUnit, Chutzpah (for JavaScript tests using QUnit, Mocha and Jasmine), etc. can be run. Tests can be distributed on multiple agents using this task (version 2).
Visual Studio test agent deployment DeployVisualStudioTestAgent@2	DeployVisualStudioTestAgent@2 is deprecated. Use the Visual Studio Test task to run unit and functional tests.
Visual Studio Test Agent Deployment DeployVisualStudioTestAgent@1	Deploy and configure Test Agent to run tests on a set of machines.

Task	Description
Xamarin Test Cloud XamarinTestCloud@1	[Deprecated] Test mobile apps with Xamarin Test Cloud using Xamarin.UITest. Instead, use the 'App Center test' task.

Tool tasks

Expand table

Task	Description
.NET Core SDK/runtime installer DotNetCoreInstaller@1 DotNetCoreInstaller@0	Acquire a specific version of the .NET Core SDK from the internet or local cache and add it to the PATH.
Docker CLI installer DockerInstaller@0	Install Docker CLI on agent machine.
Duffle tool installer DuffleInstaller@0	Install a specified version of Duffle for installing and managing CNAB bundles.
Go tool installer GoTool@0	Find in cache or download a specific version of Go and add it to the PATH.
Helm tool installer HelmInstaller@1 HelmInstaller@0	Install Helm on an agent machine.
Install Azure Func Core Tools FuncToolsInstaller@0	Install Azure Func Core Tools.
Java tool installer JavaToolInstaller@0	Acquire a specific version of Java from a user-supplied Azure blob or the tool cache and sets JAVA_HOME.
Kubectl tool installer KubectlInstaller@0	Install Kubectl on agent machine.
Kubelogin tool installer KubeloginInstaller@0	Helps to install kubelogin.
NuGet tool installer NuGetToolInstaller@1 NuGetToolInstaller@0	Acquires a specific version of NuGet from the internet or the tools cache and adds it to the PATH. Use this task to change the version of NuGet used in the NuGet tasks.
Use .NET Core UseDotNet@2	Acquires a specific version of the .NET Core SDK from the internet or the local cache and adds it to the PATH. Use this task to change the version of .NET Core used in subsequent tasks. Additionally provides proxy support.

Task	Description
Use Node.js ecosystem UseNode@1 NodeTool@0	Set up a Node.js environment and add it to the PATH, additionally providing proxy support.
Use Python version UsePythonVersion@0	Use the specified version of Python from the tool cache, optionally adding it to the PATH.
Use Ruby version UseRubyVersion@0	Use the specified version of Ruby from the tool cache, optionally adding it to the PATH.
Visual Studio test platform installer VisualStudioTestPlatformInstaller@1	Acquire the test platform from nuget.org or the tool cache. Satisfies the 'vstest' demand and can be used for running tests and collecting diagnostic data using the Visual Studio Test task.

Utility tasks

Expand table

Task	Description
Advanced Security Dependency Scanning AdvancedSecurity-Dependency-Scanning@1	Scan for open source dependency vulnerabilities in your source code.
Archive files ArchiveFiles@2	Compress files into .7z, .tar.gz, or .zip.
Archive Files ArchiveFiles@1	Archive files using compression formats such as .7z, .rar, .tar.gz, and .zip.
Azure Network Load Balancer AzureNLBManagement@1	Connect or disconnect an Azure virtual machine's network interface to a Load Balancer's back end address pool.
Bash Bash@3	Run a Bash script on macOS, Linux, or Windows.
Batch script BatchScript@1	Run a Windows command or batch script and optionally allow it to change the environment.
Cache Cache@2	Cache files between runs.
Cache (Beta) CacheBeta@1	Cache files between runs.

Task	Description
CacheBeta@0	
Command Line CmdLine@2 CmdLine@1	Run a command line script using Bash on Linux and macOS and cmd.exe on Windows.
Copy and Publish Build Artifacts CopyPublishBuildArtifacts@1	CopyPublishBuildArtifacts@1 is deprecated. Use the Copy Files task and the Publish Build Artifacts task instead.
Copy files CopyFiles@2	Copy files from a source folder to a target folder using patterns matching file paths (not folder paths).
Copy Files CopyFiles@1	Copy files from source folder to target folder using minimatch patterns (The minimatch patterns will only match file paths, not folder paths).
cURL Upload Files cURLUploader@2 cURLUploader@1	Use cURL's supported protocols to upload files.
Decrypt file (OpenSSL) DecryptFile@1	Decrypt a file using OpenSSL.
Delay Delay@1	Delay further execution of a workflow by a fixed time.
Delete files DeleteFiles@1	Delete folders, or files matching a pattern.
Deploy Azure Static Web App AzureStaticWebApp@0	Build and deploy an Azure Static Web App.
Download artifacts from file share DownloadFileshareArtifacts@1	Download artifacts from a file share, like \share\drop.
Download build artifacts DownloadBuildArtifacts@1 DownloadBuildArtifacts@0	Download files that were saved as artifacts of a completed build.
Download GitHub Release DownloadGitHubRelease@0	Downloads a GitHub Release from a repository.
Download package DownloadPackage@1 DownloadPackage@0	Download a package from a package management feed in Azure Artifacts.
Download Pipeline Artifacts DownloadPipelineArtifact@2	Download build and pipeline artifacts.

Task	Description
DownloadPipelineArtifact@1 DownloadPipelineArtifact@0	
Download secure file DownloadSecureFile@1	Download a secure file to the agent machine.
Extract files ExtractFiles@1	Extract a variety of archive and compression files such as .7z, .rar, .tar.gz, and .zip.
File transform FileTransform@2 FileTransform@1	Replace tokens with variable values in XML or JSON configuration files.
FTP upload FtpUpload@2 FtpUpload@1	Upload files using FTP.
GitHub Comment GitHubComment@0	Write a comment to your GitHub entity i.e. issue or a pull request (PR).
GitHub Release GitHubRelease@1 GitHubRelease@0	Create, edit, or delete a GitHub release.
Install Apple certificate InstallAppleCertificate@2	Install an Apple certificate required to build on a macOS agent machine.
Install Apple Certificate InstallAppleCertificate@1 InstallAppleCertificate@0	Install an Apple certificate required to build on a macOS agent.
Install Apple provisioning profile InstallAppleProvisioningProfile@1	Install an Apple provisioning profile required to build on a macOS agent machine.
Install Apple Provisioning Profile InstallAppleProvisioningProfile@0	Install an Apple provisioning profile required to build on a macOS agent.
Install SSH key InstallSSHKey@0	Install an SSH key prior to a build or deployment.
Invoke Azure Function AzureFunction@1 AzureFunction@0	Invoke an Azure Function.
Jenkins download artifacts JenkinsDownloadArtifacts@1	Download artifacts produced by a Jenkins job.
Node.js tasks runner installer NodeTaskRunnerInstaller@0	Install specific Node.js version to run node tasks.

Task	Description
Notation Notation@0	Azure Pipepine Task for setting up Notation CLI, sign and verify with Notation.
PowerShell PowerShell@2 PowerShell@1	Run a PowerShell script on Linux, macOS, or Windows.
Publish build artifacts PublishBuildArtifacts@1	Publish build artifacts to Azure Pipelines or a Windows file share.
Publish Pipeline Artifacts PublishPipelineArtifact@1 PublishPipelineArtifact@0	Publish (upload) a file or directory as a named artifact for the current run.
Publish Pipeline Metadata PublishPipelineMetadata@0	Publish Pipeline Metadata to Evidence store.
Publish To Azure Service Bus PublishToAzureServiceBus@1 PublishToAzureServiceBus@0	Sends a message to Azure Service Bus using a service connection (no agent is required).
Python script PythonScript@0	Run a Python file or inline script.
Query Azure Monitor alerts AzureMonitor@1	Observe the configured Azure Monitor rules for active alerts.
Query Classic Azure Monitor alerts AzureMonitor@0	Observe the configured classic Azure Monitor rules for active alerts.
Query work items queryWorkItems@0	Execute a work item query and check the number of items returned.
Review App ReviewApp@0	Use this task under deploy phase provider to create a resource dynamically.
Service Fabric PowerShell ServiceFabricPowerShell@1	Run a PowerShell script in the context of an Azure Service Fabric cluster connection.
Shell script ShellScript@2	Run a shell script using Bash.
Update Service Fabric App Versions ServiceFabricUpdateAppVersions@1	Automatically updates the versions of a packaged Service Fabric application.
Update Service Fabric manifests ServiceFabricUpdateManifests@2	Automatically update portions of application and service manifests in a packaged Azure Service Fabric application.
Xamarin License XamarinLicense@1	[Deprecated] Upgrade to free version of Xamarin: https://store.xamarin.com .

Open source

These tasks are open source on GitHub. Feedback and contributions are welcome. See Pipeline task changelog for a list of task changes, including a historical record of task updates.

FAQ

What are task input aliases?

Inputs to a task are identified by a `label`, `name`, and may include one or more optional `aliases`. The following example is an excerpt from the source code for the **Known Hosts Entry** input of the InstallSSHKey@0 task.

JSON

```
{
    "name": "hostName",
    "aliases": [
        "knownHostsEntry"
    ],
    "label": "Known Hosts Entry"
    ...
}
```

Before YAML pipelines were introduced in 2019, pipelines were created and edited using a UI based pipeline editor, and only the `label` was used by pipeline authors to reference a task input.

When YAML pipelines were introduced in 2019, pipeline authors using YAML started using the task input `name` to refer to a task input. In some cases, the task input names weren't descriptive, so `aliases` were added to provide additional descriptive names for task inputs.

For example, the `InstallSSHKey@0` task has a **Known Hosts Entry** input named `hostName` that expects an entry from a **known_hosts** file. The **Known Hosts Entry** label in the classic pipeline designer makes this clear, but it isn't as clear when using the `hostName` name in a YAML pipeline. Task input aliases were introduced to allow task authors to provide decriptive names for their previously authored tasks, and for the `InstallSSHKey@0` task, a `knownHostsEntry` alias was added, while keeping the original `hostName` name for compatibility with existing pipelines using that name.

Any items in a task input's `aliases` are interchangeable with the `name` in a YAML pipeline. The following two YAML snippets are functionally identical, with the first example using the `knownHostsEntry` alias and the second example using `hostName`.

yml

```yml
- task: InstallSSHKey@0
  inputs:
```

```
    # Using knownHostsEntry alias
    knownHostsEntry: 'sample known hosts entry line'
    # Remainder of task inputs omitted

- task: InstallSSHKey@0
  inputs:
    # Using hostName name
    hostName: 'sample known hosts entry line'
    # Remainder of task inputs omitted
```

Starting with Azure DevOps Server 2019.1, the YAML pipeline editor was introduced, which provides an intellisense type functionality.

The YAML pipeline editor uses the Yamlschema - Get REST API to retrieve the schema used for validation in the editor. If a task input has an alias, the schema promotes the alias to the primary YAML name for the task input, and the alias is suggested by the intellisense.

The following example is the **Known Hosts Entry** task input for the `InstallSSHKey@0` task from the YAML schema, with `knownHostsEntry` listed in the name position and `hostName` in the `aliases` collection.

JSON

```
"properties": {
    "knownHostsEntry": {
      "type": "string",
      "description": "Known Hosts Entry",
      "ignoreCase": "key",
      "aliases": [
        "hostName"
      ]
    },
```

Because the intellisense in the YAML pipeline editor displays `knownHostsEntry`, and the YAML generated by the task assistant uses `knownHostsEntry` in the generated YAML, the task reference displays the `alias` from the task source code as the YAML name for a task input. If a task has more than one alias (there are a few that have two aliases), the first alias is used as the name.

Why did the task reference change?

The Azure Pipelines tasks reference documentation moved to its current location to support the following improvements.

- Task articles are generated using the task source code from the Azure Pipelines tasks open source repository.
- Task input names and aliases are generated from the task source so they are always up to date.
- YAML syntax blocks are generated from the task source so they are up to date.
- Supports community contributions with integrated user content such as enhanced task input descriptions, remarks and examples.
- Provides task coverage for all supported Azure DevOps versions.
- Updated every sprint to cover the latest updates.

To contribute, see Contributing to the tasks content.

Where can I learn step-by-step how to build my app?

Build your app

Can I add my own build tasks?

Yes: Add a build task

What are installer tasks?

To learn more about tool installer tasks, see Tool installers.

Troubleshoot pipeline runs

Article • 12/04/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

If your pipeline run fails to complete, you can use the diagnostic information and logs provided by the pipeline run summary page to help troubleshoot the issue.

View logs

Select the error message to view the logs for the task that failed to complete.

The logs page is displayed, with the error selected. In this example, there's an error in the `cmd-line` task, where the `echo` command is entered as `ech`.

You can view the raw log for the task by choosing **View raw log**, and you can search the log using **Find**.

Scan the logs of the failing task for error information and clues as to why the task is failing. By default, nonverbose logs are generated by a pipeline run. If the default logs don't indicate the cause of the issue, you can get more information by configuring verbose logs.

Error analysis page

Troubleshooting assistance is available using the **Error analysis** page. Move the mouse over the error information line and choose the **View analysis** icon.

Choose **View agent** for self-hosted agents (or **About hosted agent image** for Microsoft-hosted agents) to view more information about the agent used to run the pipeline, and **View log** to view the pipeline run logs.

Choose the name of the task below **Run-time details** to view information about the task.

In this example, you can see that there's an error in the `Value` of the `Script`. Choose **About this task** to view the documentation for the task.

If the issue isn't apparent from the pipeline run summary page or browsing the logs, check the following Common issues section, and see Review logs to diagnose pipeline issues for information on downloading complete logs which include additional diagnostic information.

Common issues

- Job time-out
- Issues downloading code
- My pipeline is failing on a command-line step such as MSBUILD
- File or folder in use errors
- Intermittent or inconsistent MSBuild failures
- Process stops responding
- Line endings for multiple platforms
- Variables having ' (single quote) appended
- Service Connection related issues

Task insights for failed pipeline runs

Azure DevOps provides a **Task Insights for Failed Pipeline Runs** setting, that when enabled, provides pop-up notifications of build failures with a link to view a report.

To configure this setting, navigate to Preview features, find **Task Insights for Failed Pipeline Runs**, and choose the desired setting.

Job time-out

A pipeline can run for a long time and then fail due to job time-out. Job timeout closely depends on the agent being used. Free Microsoft hosted agents have a max timeout of 60 minutes per job for a private repository and 360 minutes for a public repository. To increase the max timeout for a job, you can opt for any of the following.

- Buy a Microsoft hosted agent which will give you 360 minutes for all jobs, irrespective of the repository used
- Use a self-hosted agent to rule out any timeout issues due to the agent

Learn more about job timeout.

> ⓘ **Note**
>
> If your Microsoft-hosted agent jobs are timing out, ensure that you haven't specified a pipeline timeout that is less than the max timeout for a job. To check, see **Timeouts**.

Issues downloading code

- My pipeline is failing on a checkout step
- Team Foundation Version Control (TFVC) issues

My pipeline is failing on a checkout step

If you're using a `checkout` step on an Azure Repos Git repository in your organization that is in a different project than your pipeline, ensure that the **Limit job authorization scope to current project** setting is disabled, or follow the steps in Scoped build identities to ensure that your pipeline has access to the repository.

When your pipeline can't access the repository due to limited job authorization scope, you will receive the error `Git fetch failed with exit code 128` and your logs will contain an entry similar to `Remote: TF401019: The Git repository with name or identifier <your repo name> does not exist or you do not have permissions for the operation you are attempting.`

If your pipeline is failing immediately with `Could not find a project that corresponds with the repository`, ensure that your project and repository name are correct in the `checkout` step or the repository resource declaration.

Team Foundation Version Control (TFVC) issues

- Get sources not downloading some files
- Get sources through Team Foundation Proxy

Get sources not downloading some files

This might be characterized by a message in the log "All files up to date" from the `tf get` command. Verify the built-in service identity has permission to download the sources. Either the identity *Project Collection Build Service* or *Project Build Service* will need permission to download the sources, depending on the selected authorization scope on General tab of the build pipeline. In the version control web UI, you can browse the project files at any level of the folder hierarchy and check the security settings.

Get sources through Team Foundation Proxy

The easiest way to configure the agent to get sources through a Team Foundation Proxy is set environment variable `TFSPROXY` that point to the TFVC proxy server for the agent's run as user.

Windows:

Windows Command Prompt

```
    set TFSPROXY=http://tfvcproxy:8081
    setx TFSPROXY=http://tfvcproxy:8081 // If the agent service is running
as NETWORKSERVICE or any service account you can't easily set user level
environment variable
```

macOS/Linux:

Bash

```
    export TFSPROXY=http://tfvcproxy:8081
```

My pipeline is failing on a command-line step such as MSBUILD

It is helpful to narrow whether a build or release failure is the result of an Azure Pipelines/TFS product issue (agent or tasks). Build and release failures might also result from external commands.

Check the logs for the exact command-line executed by the failing task. Attempting to run the command locally from the command line might reproduce the issue. It can be helpful to run the command locally from your own machine, and/or sign in to the machine and run the command as the service account.

For example, is the problem happening during the MSBuild part of your build pipeline (for example, are you using either the MSBuild or Visual Studio Build task)? If so, then try running the same MSBuild command on a local machine using the same arguments. If you can reproduce the problem on a local machine, then your next steps are to investigate the MSBuild problem.

File layout

The location of tools, libraries, headers, and other things needed for a build might be different on the hosted agent than from your local machine. If a build fails because it can't find one of these files, you can use the below scripts to inspect the layout on the agent. This might help you track down the missing file.

Create a new YAML pipeline in a temporary location (e.g. a new repo created for the purpose of troubleshooting). As written, the script searches directories on your path. You can optionally edit the `SEARCH_PATH=` line to search other places.

YAML

```yaml
# Script for Linux and macOS
pool: { vmImage: ubuntu-latest } # or whatever pool you use
steps:
- checkout: none
- bash: |
    SEARCH_PATH=$PATH  # or any colon-delimited list of paths
    IFS=':' read -r -a PathDirs <<< "$SEARCH_PATH"
    echo "##[debug] Found directories"
    for element in "${PathDirs[@]}"; do
        echo "$element"
    done;
    echo;
    echo;
    echo "##[debug] Found files"
    for element in "${PathDirs[@]}"; do
        find "$element" -type f
    done
```

YAML

```yaml
# Script for Windows
pool: { vmImage: windows-2019 } # or whatever pool you use
```

```
steps:
- checkout: none
- powershell: |
    $SEARCH_PATH=$Env:Path
    Write-Host "##[debug] Found directories"
    ForEach ($Dir in $SEARCH_PATH -split ";") {
      Write-Host "$Dir"
    }
    Write-Host ""
    Write-Host ""
    Write-Host "##[debug] Found files"
    ForEach ($Dir in $SEARCH_PATH -split ";") {
      Get-ChildItem $Dir -File -ErrorAction Continue | ForEach-Object -
Process {
        Write-Host $_.FullName
      }
    }
```

Differences between local command prompt and agent

Keep in mind, some differences are in effect when executing a command on a local machine and when a build or release is running on an agent. If the agent is configured to run as a service on Linux, macOS, or Windows, then it is not running within an interactive logged-on session. Without an interactive logged-on session, UI interaction and other limitations exist.

File or folder in use errors

`File or folder in use` errors are often indicated by error messages such as:

- `Access to the path [...] is denied.`
- `The process cannot access the file [...] because it is being used by another process.`
- `Access is denied.`
- `Can't move [...] to [...]`

Troubleshooting steps:

- Detect files and folders in use
- Anti-virus exclusion
- MSBuild and /nodeReuse:false
- MSBuild and /maxcpucount:[n]

Detect files and folders in use

On Windows, tools like Process Monitor can be to capture a trace of file events under a specific directory. Or, for a snapshot in time, tools like Process Explorer or Handle can be used.

Anti-virus exclusion

Anti-virus software scanning your files can cause file or folder in use errors during a build or release. Adding an anti-virus exclusion for your agent directory and configured "work folder" can help to identify anti-virus software as the interfering process.

MSBuild and /nodeReuse:false

If you invoke MSBuild during your build, make sure to pass the argument `/nodeReuse:false` (short form `/nr:false`). Otherwise MSBuild processes will remain running after the build completes. The processes remain for some time in anticipation of a potential subsequent build.

This feature of MSBuild can interfere with attempts to delete or move a directory - due to a conflict with the working directory of the MSBuild process(es).

The MSBuild and Visual Studio Build tasks already add `/nr:false` to the arguments passed to MSBuild. However, if you invoke MSBuild from your own script, then you would need to specify the argument.

MSBuild and /maxcpucount:[n]

By default the build tasks such as MSBuild and Visual Studio Build run MSBuild with the `/m` switch. In some cases this can cause problems such as multiple process file access issues.

Try adding the `/m:1` argument to your build tasks to force MSBuild to run only one process at a time.

File-in-use issues might result when leveraging the concurrent-process feature of MSBuild. Not specifying the argument `/maxcpucount:[n]` (short form `/m:[n]`) instructs MSBuild to use a single process only. If you are using the MSBuild or Visual Studio Build tasks, you might need to specify "/m:1" to override the "/m" argument that is added by default.

Intermittent or inconsistent MSBuild failures

If you're experiencing intermittent or inconsistent MSBuild failures, try instructing MSBuild to use a single-process only. Intermittent or inconsistent errors might indicate that your target configuration is incompatible with the concurrent-process feature of MSBuild. See MSBuild and /maxcpucount:[n].

Process stops responding

Process stops responding causes and troubleshooting steps:

- Waiting for Input
- Process dump
- WiX project

Waiting for Input

A process that stops responding might indicate that a process is waiting for input.

Running the agent from the command line of an interactive logged on session might help to identify whether a process is prompting with a dialog for input.

Running the agent as a service might help to eliminate programs from prompting for input. For example in .NET, programs might rely on the System.Environment.UserInteractive Boolean to determine whether to prompt. When the agent is running as a Windows service, the value is false.

Process dump

Analyzing a dump of the process can help to identify what a deadlocked process is waiting on.

WiX project

Building a WiX project when custom MSBuild loggers are enabled, can cause WiX to deadlock waiting on the output stream. Adding the additional MSBuild argument `/p:RunWixToolsOutOfProc=true` will work around the issue.

Line endings for multiple platforms

When you run pipelines on multiple platforms, you can sometimes encounter problems with different line endings. Historically, Linux and macOS used linefeed (LF) characters while Windows used a carriage return plus a linefeed (CRLF). Git tries to compensate for

the difference by automatically making lines end in LF in the repo but CRLF in the working directory on Windows.

Most Windows tools are fine with LF-only endings, and this automatic behavior can cause more problems than it solves. If you encounter issues based on line endings, we recommend you configure Git to prefer LF everywhere. To do this, add a .gitattributes file to the root of your repository. In that file, add the following line:

```
* text eol=lf
```

Variables having ' (single quote) appended

If your pipeline includes a Bash script that sets variables using the `##vso` command, you might see an additional `'` appended to the value of the variable you set. This occurs because of an interaction with `set -x`. The solution is to disable `set -x` temporarily before setting a variable. The Bash syntax for doing that is `set +x`.

Bash

```bash
set +x
echo ##vso[task.setvariable variable=MY_VAR]my_value
set -x
```

Why does this happen?

Many Bash scripts include the `set -x` command to assist with debugging. Bash will trace exactly what command was executed and echo it to stdout. This will cause the agent to see the `##vso` command twice, and the second time, Bash will have added the `'` character to the end.

For instance, consider this pipeline:

YAML

```yaml
steps:
- bash: |
    set -x
    echo ##vso[task.setvariable variable=MY_VAR]my_value
```

On stdout, the agent will see two lines:

```bash
##vso[task.setvariable variable=MY_VAR]my_value
+ echo '##vso[task.setvariable variable=MY_VAR]my_value'
```

When the agent sees the first line, `MY_VAR` will be set to the correct value, "my_value". However, when it sees the second line, the agent will process everything to the end of the line. `MY_VAR` will be set to "my_value'".

Libraries aren't installed for Python application when script executes

When a Python application is deployed, in some cases, a CI/CD pipeline runs and the code is deployed successfully, but the *requirements.txt* file that's responsible for installing all dependency libraries doesn't execute.

To install the dependencies, use a post-deployment script in the App Service deployment task. The following example shows the command you must use in the post-deployment script. You can update the script for your scenario.

```
D:\home\python364x64\python.exe -m pip install -r requirements.txt
```

Service Connection related issues

To troubleshoot issues related to service connections, see Service connection troubleshooting.

Enable Storage Explorer to deploy static content like .css and .js to a static website from Azure DevOps via Azure Pipelines

In this scenario, you can use the Azure File Copy task to upload content to the website. You can use any of the tools described in Uploading content to upload content to the web container.

Next steps

- Review logs

Troubleshoot pipeline triggers

Article • 12/19/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

If a pipeline doesn't start at all, check the following common trigger related issues.

- UI settings override YAML trigger setting
- Disable implied YAML CI trigger setting is enabled
- Pull request triggers not supported with Azure Repos
- Branch filters misconfigured in CI and PR triggers
- Scheduled trigger time zone conversions
- UI settings override YAML scheduled triggers

> **Note**
>
> An additional reason that runs may not start is that your organization goes dormant five minutes after the last user signs out of Azure DevOps. After that, each of your build pipelines will run one more time. For example, while your organization is dormant:
>
> - A nightly build of code in your organization will run only one night until someone signs in again.
> - CI builds of an Other Git repo will stop running until someone signs in again.

UI settings override YAML trigger setting

YAML pipelines can have their `trigger` and `pr` trigger settings overridden in the pipeline settings UI. If your `trigger` or `pr` triggers don't seem to be firing, check that setting. While editing your pipeline, choose **...** and then **Triggers**.

Check the **Override the YAML trigger from here** setting for the types of trigger (**Continuous integration** or **Pull request validation**) available for your repo.

Pull request triggers not supported with Azure Repos

If your `pr` trigger isn't firing, and you are using Azure Repos, it is because `pr` triggers aren't supported for Azure Repos. In Azure Repos Git, branch policies are used to implement pull request build validation. For more information, see Branch policy for pull request validation.

Disable implied YAML CI trigger setting is enabled

YAML pipelines are configured by default with a CI trigger on all branches, unless the Disable implied YAML CI trigger setting, introduced in Azure DevOps sprint 227, is

enabled. The **Disable implied YAML CI trigger** setting can be configured at the organization level or at the project level, and by default, the setting is not enabled.

If your pipelines use the default implicit CI trigger, and they stop working, check this setting. When the **Disable implied YAML CI trigger** setting is enabled, CI triggers for YAML pipelines are not enabled if the YAML pipeline doesn't have a `trigger` section.

Branch filters misconfigured in CI and PR triggers

When you define a YAML PR or CI trigger, you can specify both `include` and `exclude` clauses for branches, tags, and paths. Ensure that the `include` clause matches the details of your commit and that the `exclude` clause doesn't exclude them. For more information, see pr and trigger.

> **Note**
>
> If you specify an `exclude` clause without an `include` clause, it is equivalent to specifying `*` in the `include` clause.

Scheduled trigger time zone conversions

YAML scheduled triggers are set using UTC time zone. If your scheduled triggers don't seem to be firing at the right time, confirm the conversions between UTC and your local time zone, taking into account the day setting as well. For more information, see Scheduled triggers.

UI settings override YAML scheduled triggers

If your YAML pipeline has both YAML scheduled triggers and UI defined scheduled triggers, only the UI defined scheduled triggers are run. To run the YAML defined scheduled triggers in your YAML pipeline, you must remove the scheduled triggers defined in the pipeline settings UI.

To access the pipeline settings UI from a YAML pipeline, edit your pipeline, choose **...** and then **Triggers**.

Remove all scheduled triggers.

Once all UI scheduled triggers are removed, a push must be made in order for the YAML scheduled triggers to start running. For more information, see Scheduled triggers.

I need more help. I found a bug. I've got a suggestion. Where do I go?

Get subscription, billing, and technical support

Report any problems or submit feedback at Developer Community.

We welcome your suggestions:

Troubleshoot pipeline failure to start

Article • 10/14/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

If your pipeline queues but never starts, check the following items.

- Parallel job limits - no available agents or you have hit your free limits
- Can't access Azure Key Vault behind firewall from Azure DevOps
- You don't have enough concurrency
- Your job might be waiting for approval
- All available agents are in use
- Demands that don't match the capabilities of an agent
- Check Azure DevOps status for a service degradation

> **Note**
>
> The following scenarios won't consume a parallel job:
>
> - If you use release pipelines or multi-stage YAML pipelines, then a run consumes a parallel job only when it's being actively deployed to a stage. While the release is waiting for an approval or a manual intervention, it does not consume a parallel job.
> - When you run a server job or deploy to a deployment group using release pipelines, you don't consume any parallel jobs.
>
> Learn more: **How a parallel job is consumed by a pipeline**, **Add Pre-deployment approvals**, **Server jobs**, **Deployment groups**

Parallel job limits - no available agents or you have hit your free limits

If you're currently running other pipelines, you might not have any remaining parallel jobs, or you might have hit your free limits.

Check for available parallel jobs

> **Note**
>
> Azure Pipelines has temporarily disabled the automatic free grant of Microsoft-hosted parallel jobs in new organizations for public projects and for certain private projects. If you don't have any parallel jobs, your pipelines will fail with the following error: `##[error]No hosted parallelism has been purchased or granted. To request a free parallelism grant, please fill out the following form https://aka.ms/azpipelines-parallelism-request`. Check your Microsoft-hosted parallel jobs as described in the following section, and if you have zero parallel jobs, you can request a free grant of parallel jobs. To request the free grant of parallel jobs for your organization, submit **a request**. Please allow 2-3 business days to respond to your grant request.

To check your limits, navigate to **Project settings**, **Parallel jobs**.

If you're using Microsoft-hosted agents, check the parallel job limits for **Microsoft-hosted** for **Private projects** or **Public projects**, depending on whether your Azure

DevOps project is a private project (default) or public project.

After reviewing the limits, check concurrency to see how many jobs are currently running and how many are available.

Can't access Azure Key Vault behind firewall from Azure DevOps

If you can't access Azure Key Vault from your pipeline, the firewall might be blocking the Azure DevOps Services agent IP address. The IP addresses published in the weekly JSON file ↗ must be allowlisted. For more information, see Microsoft-hosted agents: Networking.

You don't have enough concurrency

To check how much concurrency you have:

1. To check your limits, navigate to **Project settings**, **Parallel jobs**.

You can also reach this page by navigating to `https://dev.azure.com/{org}/_settings/buildqueue?_a=concurrentJobs`, or choosing **manage parallel jobs** from the logs.

```
Pool: Azure Pipelines
Image: ubuntu-16.04
Queued: Just now [manage parallel jobs]
```

2. Determine which pool you want to check concurrency on (Microsoft hosted or self hosted pools), and choose **View in-progress jobs**.

3. You'll see text that says **Currently running X/X jobs**. If both numbers are the same, pending jobs will wait until currently running jobs complete.

You can view all jobs, including queued jobs, by selecting **Agent pools** from the **Project settings**.

In this example, the concurrent job limit is one, with one job running and one queued up. When all agents are busy running jobs, as in this example, the following message is displayed when additional jobs are queued: `The agent request is not running because all potential agents are running other requests. Current position in queue: 1`. In this example the job is next in the queue so its position is one.

Your job might be waiting for approval

Your pipeline might not move to the next stage because it's waiting on approval. For more information, see Define approvals and checks.

All available agents are in use

Jobs might wait if all your agents are currently busy. To check your agents:

1. Navigate to `https://dev.azure.com/{org}/_settings/agentpools`

2. Select the agent pool to check, in this example **FabrikamPool**, and choose **Agents**.

This page shows all the agents currently online/offline and in use. You can also add additional agents to the pool from this page.

Demands that don't match the capabilities of an agent

If your pipeline has demands that don't meet the capabilities of any of your agents, your pipeline won't start. If only some of your agents have the desired capabilities and they're currently running other pipelines, your pipeline will be stalled until one of those agents becomes available.

To check the capabilities and demands specified for your agents and pipelines, see Capabilities.

> **Note**
>
> Capabilities and demands are typically used only with self-hosted agents. If your pipeline has demands that don't match the system capabilities of the agent, unless you have explicitly labelled the agents with matching capabilities, your pipelines won't get an agent.

Check Azure DevOps status for a service degradation

Check the Azure DevOps Service Status Portal for any issues that might cause a service degradation, such as increased queue time for agents. For more information, see Azure DevOps Service Status.

I need more help. I found a bug. I've got a suggestion. Where do I go?

Get subscription, billing, and technical support

Report any problems or submit feedback at Developer Community.

We welcome your suggestions:

Review logs to diagnose pipeline issues

Article • 11/28/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Pipeline logs provide a powerful tool for determining the cause of pipeline failures.

A typical starting point is to review the logs in your completed build or release. You can view logs by navigating to the pipeline run summary and selecting the job and task. If a certain task is failing, check the logs for that task.

In addition to viewing logs in the pipeline build summary, you can download complete logs which include additional diagnostic information, and you can configure more verbose logs to assist with your troubleshooting.

Configure verbose logs

To assist with troubleshooting, you can configure your logs to be more verbose.

- To configure verbose logs for a single run, you can start a new build by choosing **Run pipeline** and selecting **Enable system diagnostics**, **Run**.

- To configure verbose logs for all runs, you can add a variable named `system.debug` and set its value to `true`.

Azure pipeline logs can now capture resource utilization metrics such as memory, CPU usage and available disk space. The logs also include resources used by the pipeline agent and child processes including tasks run in a job. If you suspect your pipeline job may run into resource constraints, enable verbose logs to have resource utilization information injected into pipeline logs. This works on any agent, independent from hosting model.

View and download logs

To view individual logs for each step, navigate to the build results for the run, and select the job and step.

To download all logs, navigate to the build results for the run, select ..., and choose **Download logs**.

In addition to the pipeline diagnostic logs, the following specialized log types are available, and may contain information to help you troubleshoot.

- Worker diagnostic logs
- Agent diagnostic logs

- Other logs

Worker diagnostic logs

You can get the diagnostic log of the completed build that was generated by the worker process on the build agent. Look for the `worker` log file that has the date and time stamp of your completed build. For example, `worker_20160623-192022-utc_6172.log`.

Agent diagnostic logs

Agent diagnostic logs provide a record of how the agent was configured and what happened when it ran. Look for the `agent` log files. For example, `agent_20160624-144630-utc.log`. There are two kinds of agent log files:

- The log file generated when you ran `config.cmd`. This log:
  - Includes this line near the top: `Adding Command: configure`
  - Shows the configuration choices made.
- The log file generated when you ran `run.cmd`. This log:
  - Cannot be opened until the process is terminated.
  - Attempts to connect to your Azure DevOps organization or Team Foundation Server.
  - Shows when each job was run, and how it completed

Both logs show how the agent capabilities were detected and set.

Network diagnostics for self-hosted agents

Set the value of `Agent.Diagnostic` to `true` to collect the following additional logs that can be used for troubleshooting network issues for self-hosted agents.

Expand table

File	Information	Applies to
`cloudinit.*`	Cloud-init completed successfully (if used)	Linux
`BrokenPackages.*`	Packages are in a consistent state	Linux

File	Information	Applies to
`Agent.*`	Environment variables	Linux, Windows
`waagentConf.txt`	Azure VM agent (waagent.conf)	Azure: Linux, Windows
`environment.txt` / `agent.*`	Account group membership list	Windows

> **Note**
>
> `Agent.Diagnostic` is set to `true` automatically when `System.Debug` is set to `true`.
>
> The `Agent.Diagnostic` variable and logs described in this section are available with Agent v2.200.0 and higher.

For more information, see agent troubleshooting in the microsoft/azure-pipelines-agent Azure Pipelines agent open-source agent repository.

Other logs

Inside the diagnostic logs you will find `environment.txt` and `capabilities.txt`.

The `environment.txt` file has various information about the environment within which your build ran. This includes information like what tasks are run, whether or not the firewall is enabled, PowerShell version info, and some other items. We continually add to this data to make it more useful.

The `capabilities.txt` file provides a clean way to see all capabilities installed on the build machine that ran your build.

HTTP trace logs

- Use built-in HTTP tracing
- Use full HTTP tracing - Windows
- Use full HTTP tracing - macOS and Linux

> **Important**
>
> HTTP traces and trace files can contain passwords and other secrets. Do **not** post them on a public sites.

Use built-in HTTP tracing

If your agent is version 2.114.0 or newer, you can trace the HTTP traffic headers and write them into the diagnostic log. Set the `VSTS_AGENT_HTTPTRACE` environment variable before you launch the agent.listener.

Bash

```
Windows:
    set VSTS_AGENT_HTTPTRACE=true

macOS/Linux:
    export VSTS_AGENT_HTTPTRACE=true
```

Use full HTTP tracing - Windows

1. Start Fiddler.
2. We recommend you listen only to agent traffic. File > Capture Traffic off (F12)
3. Enable decrypting HTTPS traffic. Tools > Fiddler Options > HTTPS tab. Decrypt HTTPS traffic
4. Let the agent know to use the proxy:

   Windows Command Prompt

   ```
   set VSTS_HTTP_PROXY=http://127.0.0.1:8888
   ```

5. Run the agent interactively. If you're running as a service, you can set as the environment variable in control panel for the account the service is running as.
6. Restart the agent.

Use full HTTP tracing - macOS and Linux

Use Charles Proxy (similar to Fiddler on Windows) to capture the HTTP trace of the agent.

1. Start Charles Proxy.
2. Charles: Proxy > Proxy Settings > SSL Tab. Enable. Add URL.
3. Charles: Proxy > Mac OSX Proxy. Recommend disabling to only see agent traffic.

```bash
export VSTS_HTTP_PROXY=http://127.0.0.1:8888
```

4. Run the agent interactively. If it's running as a service, you can set in the .env file. See nix service

5. Restart the agent.

Classic release and artifacts variables

Article • 02/28/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Classic release and artifacts variables are a convenient way to exchange and transport data throughout your pipeline. Each variable is stored as a string and its value can change between runs of your pipeline.

Variables are different from Runtime parameters which are only available at template parsing time.

As you compose the tasks for deploying your application into each stage in your DevOps CI/CD processes, variables will help you to:

- Define a more generic deployment pipeline once, and then customize it easily for each stage. For example, a variable can be used to represent the connection string for web deployment, and the value of this variable can be changed from one stage to another. These are **custom variables**.
- Use information about the context of the particular release, stage, artifacts, or agent in which the deployment pipeline is being run. For example, your script may need access to the location of the build to download it, or to the working directory on the agent to create temporary files. These are **default variables**.

> **Note**
>
> For YAML pipelines, see **user-defined variables** and **predefined variables** for more details.

Default variables

Information about the execution context is made available to running tasks through default variables. Your tasks and scripts can use these variables to find information about the system, release, stage, or agent they are running in. With the exception of **System.Debug**, these variables are read-only and their values are automatically set by the system. Some of the most significant variables are described in the following tables. To view the full list, see View the current values of all variables.

> **Tip**
>
> You can view the **current values of all variables** for a release, and use a default variable to **run a release in debug mode**.

System

Variable name	Description
System.TeamFoundationServerUri	The URL of the service connection in TFS or Azure Pipelines. Use this from your scripts or tasks to call Azure Pipelines REST APIs. Example: `https://fabrikam.vsrm.visualstudio.com/`
System.TeamFoundationCollectionUri	The URL of the Team Foundation collection or Azure Pipelines. Use this from your scripts or tasks to call REST APIs on other services such as Build and Version control. Example: `https://dev.azure.com/fabrikam/`
System.CollectionId	The ID of the collection to which this build or release belongs. Not available in TFS 2015. Example: `6c6f3423-1c84-4625-995a-f7f143a1e43d`
System.DefinitionId	The ID of the release pipeline to which the current release belongs. Not available in TFS 2015. Example: `1`
System.TeamProject	The name of the project to which this build or release belongs. Example: `Fabrikam`
System.TeamProjectId	The ID of the project to which this build or release belongs. Not available in TFS 2015. Example: `79f5c12e-3337-4151-be41-a268d2c73344`

Variable name	Description
System.ArtifactsDirectory	The directory to which artifacts are downloaded during deployment of a release. The directory is cleared before every deployment if it requires artifacts to be downloaded to the agent. Same as Agent.ReleaseDirectory and System.DefaultWorkingDirectory. Example: `C:\agent\_work\r1\a`
System.DefaultWorkingDirectory	The directory to which artifacts are downloaded during deployment of a release. The directory is cleared before every deployment if it requires artifacts to be downloaded to the agent. Same as Agent.ReleaseDirectory and System.ArtifactsDirectory. Example: `C:\agent\_work\r1\a`
System.WorkFolder	The working directory for this agent, where subfolders are created for every build or release. Same as Agent.RootDirectory and Agent.WorkFolder. Example: `C:\agent\_work`
System.Debug	This is the only system variable that can be *set* by the users. Set this to true to run the release in debug mode to assist in fault-finding. Example: `true`

Release

Variable name	Description
Release.AttemptNumber	The number of times this release is deployed in this stage. Not available in TFS 2015. Example: `1`
Release.DefinitionEnvironmentId	The ID of the stage in the corresponding release pipeline. Not available in TFS 2015. Example: `1`
Release.DefinitionId	The ID of the release pipeline to which the current release belongs. Not available in TFS 2015. Example: `1`

Variable name	Description
Release.DefinitionName	The name of the release pipeline to which the current release belongs. Example: `fabrikam-cd`
Release.Deployment.RequestedFor	The display name of the identity that triggered (started) the deployment currently in progress. Not available in TFS 2015. Example: `Mateo Escobedo`
Release.Deployment.RequestedForEmail	The email address of the identity that triggered (started) the deployment currently in progress. Not available in TFS 2015. Example: `mateo@fabrikam.com`
Release.Deployment.RequestedForId	The ID of the identity that triggered (started) the deployment currently in progress. Not available in TFS 2015. Example: `2f435d07-769f-4e46-849d-10d1ab9ba6ab`
Release.DeploymentID	The ID of the deployment. Unique per job. Example: `254`
Release.DeployPhaseID	The ID of the phase where deployment is running. Example: `127`
Release.EnvironmentId	The ID of the stage instance in a release to which the deployment is currently in progress. Example: `276`
Release.EnvironmentName	The name of stage to which deployment is currently in progress. Example: `Dev`
Release.EnvironmentUri	The URI of the stage instance in a release to which deployment is currently in progress. Example: `vstfs://ReleaseManagement/Environment/276`

Variable name	Description
Release.Environments.{stage-name}.status	The deployment status of the stage. Example: `InProgress`
Release.PrimaryArtifactSourceAlias	The alias of the primary artifact source Example: `fabrikam\_web`
Release.Reason	The reason for the deployment. Supported values are: `ContinuousIntegration` - the release started in Continuous Deployment after a build completed. `Manual` - the release started manually. `None` - the deployment reason has not been specified. `Schedule` - the release started from a schedule.
Release.ReleaseDescription	The text description provided at the time of the release. Example: `Critical security patch`
Release.ReleaseId	The identifier of the current release record. Example: `118`
Release.ReleaseName	The name of the current release. Example: `Release-47`
Release.ReleaseUri	The URI of current release. Example: `vstfs://ReleaseManagement/Release/118`
Release.ReleaseWebURL	The URL for this release. Example: `https://dev.azure.com/fabrikam/f3325c6c/_release?releaseId=392&_a=release-summary`
Release.RequestedFor	The display name of identity that triggered the release. Example: `Mateo Escobedo`
Release.RequestedForEmail	The email address of identity that triggered the release. Example: `mateo@fabrikam.com`
Release.RequestedForId	The ID of identity that triggered the release. Example: `2f435d07-769f-4e46-849d-10d1ab9ba6ab`

Variable name	Description
Release.SkipArtifactsDownload	Boolean value that specifies whether or not to skip downloading of artifacts to the agent. Example: `FALSE`
Release.TriggeringArtifact.Alias	The alias of the artifact which triggered the release. This is empty when the release was scheduled or triggered manually. Example: `fabrikam\_app`

Release-stage

Variable name	Description
Release.Environments.{stage name}.Status	The status of deployment of this release within a specified stage. Not available in TFS 2015. Example: `NotStarted`

Agent

Variable name	Description
Agent.Name	The name of the agent as registered with the agent pool. This is likely to be different from the computer name. Example: `fabrikam-agent`
Agent.MachineName	The name of the computer on which the agent is configured. Example: `fabrikam-agent`
Agent.Version	The version of the agent software. Example: `2.109.1`
Agent.JobName	The name of the job that is running, such as Release or Build. Example: `Release`

Variable name	Description
Agent.HomeDirectory	The folder where the agent is installed. This folder contains the code and resources for the agent. Example: `C:\agent`
Agent.ReleaseDirectory	The directory to which artifacts are downloaded during deployment of a release. The directory is cleared before every deployment if it requires artifacts to be downloaded to the agent. Same as System.ArtifactsDirectory and System.DefaultWorkingDirectory. Example: `C:\agent\_work\r1\a`
Agent.RootDirectory	The working directory for this agent, where subfolders are created for every build or release. Same as Agent.WorkFolder and System.WorkFolder. Example: `C:\agent\_work`
Agent.WorkFolder	The working directory for this agent, where subfolders are created for every build or release. Same as Agent.RootDirectory and System.WorkFolder. Example: `C:\agent\_work`
Agent.DeploymentGroupId	The ID of the deployment group the agent is registered with. This is available only in deployment group jobs. Not available in TFS 2018 Update 1. Example: `1`

General Artifact

For each artifact that is referenced in a release, you can use the following artifact variables. Not all variables are meaningful for each artifact type. The table below lists the default artifact variables and provides examples of the values that they have depending on the artifact type. If an example is empty, it implies that the variable is not populated for that artifact type.

Replace the `{alias}` placeholder with the value you specified for the artifact alias or with the default value generated for the release pipeline.

Variable name	Description

Variable name	Description
Release.Artifacts.{alias}.DefinitionId	The identifier of the build pipeline or repository. Azure Pipelines example: `1` GitHub example: `fabrikam/asp`
Release.Artifacts.{alias}.DefinitionName	The name of the build pipeline or repository. Azure Pipelines example: `fabrikam-ci` TFVC example: `$/fabrikam` Git example: `fabrikam` GitHub example: `fabrikam/asp (main)`
Release.Artifacts.{alias}.BuildNumber	The build number or the commit identifier. Azure Pipelines example: `20170112.1` Jenkins/TeamCity example: `20170112.1` TFVC example: `Changeset 3` Git example: `38629c964` GitHub example: `38629c964`
Release.Artifacts.{alias}.BuildId	The build identifier. Azure Pipelines example: `130` Jenkins/TeamCity example: `130` GitHub example: `38629c964d21fe405ef830b7d0220966b82c9e11`
Release.Artifacts.{alias}.BuildURI	The URL for the build. Azure Pipelines example: `vstfs://build-release/Build/130` GitHub example: `https://github.com/fabrikam/asp`
Release.Artifacts.{alias}.SourceBranch	The full path and name of the branch from which the source was built. Azure Pipelines example: `refs/heads/main`
Release.Artifacts.{alias}.SourceBranchName	The name only of the branch from which the source was built. Azure Pipelines example: `main`
Release.Artifacts.{alias}.SourceVersion	The commit that was built. Azure Pipelines example: `bc0044458ba1d9298cdc649cb5dcf013180706f7`

Variable name	Description
Release.Artifacts.{alias}.Repository.Provider	The type of repository from which the source was built. Azure Pipelines example: `Git`
Release.Artifacts.{alias}.RequestedForID	The identifier of the account that triggered the build. Azure Pipelines example: `2f435d07-769f-4e46-849d-10d1ab9ba6ab`
Release.Artifacts.{alias}.RequestedFor	The name of the account that requested the build. Azure Pipelines example: `Mateo Escobedo`
Release.Artifacts.{alias}.Type	The type of artifact source, such as Build. Azure Pipelines example: `Build` Jenkins example: `Jenkins` TeamCity example: `TeamCity` TFVC example: `TFVC` Git example: `Git` GitHub example: `GitHub`
Release.Artifacts.{alias}.PullRequest.TargetBranch	The full path and name of the branch that is the target of a pull request. This variable is initialized only if the release is triggered by a pull request flow. Azure Pipelines example: `refs/heads/main`
Release.Artifacts.{alias}.PullRequest.TargetBranchName	The name only of the branch that is the target of a pull request. This variable is initialized only if the release is triggered by a pull request flow. Azure Pipelines example: `main`

See also Artifact source alias

Primary Artifact

You designate one of the artifacts as a primary artifact in a release pipeline. For the designated primary artifact, Azure Pipelines populates the following variables.

Variable name	Same as
Build.DefinitionId	Release.Artifacts.{Primary artifact alias}.DefinitionId
Build.DefinitionName	Release.Artifacts.{Primary artifact alias}.DefinitionName

Variable name	Same as
Build.BuildNumber	Release.Artifacts.{Primary artifact alias}.BuildNumber
Build.BuildId	Release.Artifacts.{Primary artifact alias}.BuildId
Build.BuildURI	Release.Artifacts.{Primary artifact alias}.BuildURI
Build.SourceBranch	Release.Artifacts.{Primary artifact alias}.SourceBranch
Build.SourceBranchName	Release.Artifacts.{Primary artifact alias}.SourceBranchName
Build.SourceVersion	Release.Artifacts.{Primary artifact alias}.SourceVersion
Build.Repository.Provider	Release.Artifacts.{Primary artifact alias}.Repository.Provider
Build.RequestedForID	Release.Artifacts.{Primary artifact alias}.RequestedForID
Build.RequestedFor	Release.Artifacts.{Primary artifact alias}.RequestedFor
Build.Type	Release.Artifacts.{Primary artifact alias}.Type
Build.PullRequest.TargetBranch	Release.Artifacts.{Primary artifact alias}.PullRequest.TargetBranch
Build.PullRequest.TargetBranchName	Release.Artifacts.{Primary artifact alias}.PullRequest.TargetBranchName

Use default variables

You can use the default variables in two ways - as parameters to tasks in a release pipeline or in your scripts.

You can directly use a default variable as an input to a task. For example, to pass `Release.Artifacts.{Artifact alias}.DefinitionName` for the artifact source whose alias is **ASPNET4.CI** to a task, you would use `$(Release.Artifacts.ASPNET4.CI.DefinitionName)`.

To use a default variable in your script, you must first replace the `.` in the default variable names with `_`. For example, to print the value of artifact variable `Release.Artifacts.{Artifact alias}.DefinitionName` for the artifact source whose alias is **ASPNET4.CI** in a PowerShell script, you would use `$env:RELEASE_ARTIFACTS_ASPNET4_CI_DEFINITIONNAME`.

Note that the original name of the artifact source alias, `ASPNET4.CI`, is replaced by `ASPNET4_CI`.

View the current values of all variables

1. Open the pipelines view of the summary for the release, and choose the stage you are interested in. In the list of steps, choose **Initialize job**.

2. This opens the log for this step. Scroll down to see the values used by the agent for this job.

Run a release in debug mode

Show additional information as a release executes and in the log files by running the entire release, or just the tasks in an individual release stage, in debug mode. This can help you resolve issues and failures.

- To initiate debug mode for an entire release, add a variable named `System.Debug` with the value `true` to the **Variables** tab of a release pipeline.
- To initiate debug mode for a single stage, open the **Configure stage** dialog from the shortcut menu of the stage and add a variable named `System.Debug` with the value `true` to the **Variables** tab.
- Alternatively, create a variable group containing a variable named `System.Debug` with the value `true` and link this variable group to a release pipeline.

> **Tip**
>
> If you get an error related to an Azure RM service connection, see **How to: Troubleshoot Azure Resource Manager service connections**.

Custom variables

Custom variables can be defined at various scopes.

- Share values across all of the definitions in a project by using variable groups. Choose a variable group when you need to use the same values across all the

definitions, stages, and tasks in a project, and you want to be able to change the values in a single place. You define and manage variable groups in the **Library** tab.

- Share values across all of the stages by using **release pipeline variables**. Choose a release pipeline variable when you need to use the same value across all the stages and tasks in the release pipeline, and you want to be able to change the value in a single place. You define and manage these variables in the **Variables** tab in a release pipeline. In the Pipeline Variables page, open the Scope drop-down list and select "Release". By default, when you add a variable, it is set to Release scope.

- Share values across all of the tasks within one specific stage by using **stage variables**. Use a stage-level variable for values that vary from stage to stage (and are the same for all the tasks in an stage). You define and manage these variables in the **Variables** tab of a release pipeline. In the Pipeline Variables page, open the Scope drop-down list and select the required stage. When you add a variable, set the Scope to the appropriate environment.

Using custom variables at project, release pipeline, and stage scope helps you to:

- Avoid duplication of values, making it easier to update all occurrences as one operation.

- 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.

> **ⓘ Important**
>
> The values of the hidden (secret) variables are securely stored on the server and cannot be viewed by users after they are saved. During a deployment, the Azure Pipelines release service decrypts these values when referenced by the tasks and passes them to the agent over a secure HTTPS channel.

> **ⓘ Note**
>
> Creating custom variables can overwrite standard variables. For example, the PowerShell **Path** environment variable. If you create a custom `Path` variable on a Windows agent, it will overwrite the `$env:Path` variable and PowerShell won't be able to run.

Use custom variables

To use custom variables in your build and release tasks, simply enclose the variable name in parentheses and precede it with a $ character. For example, if you have a variable named **adminUserName**, you can insert the current value of that variable into a parameter of a task as `$(adminUserName)`.

> **Note**
>
> Variables in different groups that are linked to a pipeline in the same scope (for example, job or stage) will collide and the result may be unpredictable. Ensure that you use different names for variables across all your variable groups.

Define and modify your variables in a script

To define or modify a variable from a script, use the `task.setvariable` logging command. Note that the updated variable value is scoped to the job being executed, and does not flow across jobs or stages. Variable names are transformed to uppercase, and the characters "." and " " are replaced by "_".

For example, `Agent.WorkFolder` becomes `AGENT_WORKFOLDER`. On Windows, you access this as `%AGENT_WORKFOLDER%` or `$env:AGENT_WORKFOLDER`. On Linux and macOS, you use `$AGENT_WORKFOLDER`.

> **Tip**
>
> You can run a script on a:
>
> - **Windows agent** using either a **Batch script task** or **PowerShell script task**.
> - **macOS** or **Linux** agent using a **Shell script task**.

Batch

Batch script

Set the `sauce` and `secret.Sauce` variables

```
bat
```

```
@echo ##vso[task.setvariable variable=sauce]crushed tomatoes
@echo ##vso[task.setvariable variable=secret.Sauce;issecret=true]crushed
tomatoes with garlic
```

Read the variables

Arguments

arguments

```
"$(sauce)" "$(secret.Sauce)"
```

Script

bat

```
@echo off
set sauceArgument=%~1
set secretSauceArgument=%~2
@echo No problem reading %sauceArgument% or %SAUCE%
@echo But I cannot read %SECRET_SAUCE%
@echo But I can read %secretSauceArgument% (but the log is redacted so I
do not spoil
     the secret)
```

Console output from reading the variables:

Output

```
No problem reading crushed tomatoes or crushed tomatoes
But I cannot read
But I can read ******** (but the log is redacted so I do not spoil the
secret)
```

Troubleshoot ARM service connections

Article • 10/16/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

This article presents the common troubleshooting scenarios to help you resolve issues you may encounter when creating an Azure Resource Manager service connection. See Manage service connections to learn how to create, edit, and secure service connections.

What happens when you create an ARM service connection?

If you don't have a service connection, you can create one as follows:

1. From within your project, select **Project settings**, and then select **Service connections**.

2. Select **New service connection** to add a new service connection, and then select **Azure Resource Manager**. Select **Next** when you are done.

3. Select **Service principal (automatic)**, and then select **Next.

4. Select **Subscription**, and then select your subscription from the drop-down list. Fill out the form and then select **Save** when you are done.

When you save your new ARM service connection, Azure DevOps then:

1. Connects to the Microsoft Entra tenant for to the selected subscription.
2. Creates an application in Microsoft Entra ID on behalf of the user.
3. After the application has been successfully created, assign the application as a contributor to the selected subscription.
4. Creates an Azure Resource Manager service connection using this application's details.

Note

To create service connections you must be added to the Endpoint Creator group in your project settings: **Project settings** > **Service connections** > **Security**. Contributors are added to this group by default.

Troubleshooting scenarios

Below are some of the issues that may occur when creating service connections:

- Insufficient privileges to complete the operation
- Failed to obtain an access token
- A valid refresh token was not found
- Failed to assign contributor role
- Subscription isn't listed when creating a service connection
- Some subscriptions are missing from the subscription drop down menu
- Automatically created service principal secret has expired
- Failed to obtain the JSON Web Token (JWT)
- Azure subscription is not passed from the previous task output

Insufficient privileges to complete the operation

This typically occurs when the system attempts to create an application in Microsoft Entra ID on your behalf.

This is a permission issue that may be due to the following causes:

- The user has only guest permission in the directory
- The user is not authorized to add applications in the directory

The user has only guest permission in the directory

The best approach to resolve this issue, while granting only the minimum additional permissions to the user, is to increase the Guest user permissions as follows.

1. Sign in to the Azure portal using an administrator account. The account should be an owner, global administrator, or user account administrator.

2. Select **Microsoft Entra ID** in the left navigation bar.

3. Ensure you are editing the appropriate directory corresponding to the user subscription. If not, select **Switch directory** and log in using the appropriate credentials if required.

4. Select **Users** from the **Manage** section.
5. Select **User settings**.
6. Select **Manage external collaboration settings** from the **External users** section.
7. Change the **Guest user permissions are limited** option to **No**.

Alternatively, if you are prepared to give the user additional permissions (administrator-level), you can make the user a member of the **Global administrator** role. To do so follow the steps below:

> ⚠ **Warning**
>
> Users who are assigned to the Global administrator role can read and modify every administrative setting in your Microsoft Entra organization. As a best practice, we recommend that you assign this role to fewer than five people in your organization.

1. Sign in to the Azure portal using an administrator account. The account should be an owner, global administrator, or user account administrator.
2. Select **Microsoft Entra ID** from the left navigation pane.
3. Ensure you are editing the appropriate directory corresponding to the user subscription. If not, select **Switch directory** and log in using the appropriate credentials if required.
4. Select **Users** from the **Manage** section.
5. Use the search box to search for the user you want to manage.
6. Select **Directory role** from the **Manage** section, and then change the role to **Global administrator**. Select **Save** when you are done.

It typically takes 15 to 20 minutes to apply the changes globally. The user then can try recreating the service connection.

The user is not authorized to add applications in the directory

You must have permissions to add integrated applications in the directory. The directory administrator has permissions to change this setting.

1. Select **Microsoft Entra ID** in the left navigation pane.

2. Ensure you are editing the appropriate directory corresponding to the user subscription. If not, select **Switch directory** and log in using the appropriate credentials if required.

3. Select **Users**, and then select **User settings**.

4. Under **App registrations**, and then change the **Users can register applications** option to **Yes**.

You can also create the service principal with an existing user who already has the required permissions in Microsoft Entra ID. See Create an Azure Resource Manager service connection with an existing service principal for more information.

Failed to obtain an access token or a valid refresh token was not found

These errors typically occur when your session has expired. To resolve these issues:

1. Sign out of Azure DevOps.
2. Open an InPrivate or incognito browser window and navigate to Azure DevOps.
3. Sign in using the appropriate credentials.
4. Select your organization and your project.
5. Create your service connection.

Failed to assign Contributor role

This error typically occurs when you do not have **Write** permission for the selected Azure subscription.

To resolve this issue, ask the subscription administrator to assign you the appropriate role in Microsoft Entra ID.

Subscription isn't listed when creating a service connection

A maximum of 50 Azure subscriptions are listed in the various Azure subscription drop-down menus (billing, service connection, etc.). If you're setting up a service connection and you have more than 50 Azure subscriptions, some of your subscriptions won't be listed. In this scenario, complete the following steps:

1. Create a new, native Microsoft Entra user in the Microsoft Entra instance of your Azure subscription.

2. Set up the Microsoft Entra user so that it has the proper permissions to set up billing or create service connections. For more information, see Add a user who can set up billing for Azure DevOps.

3. Add the Microsoft Entra user to the Azure DevOps org with a **Stakeholder** access level, and then add it to the **Project Collection Administrators** group (for billing), or ensure that the user has sufficient permissions in the Team Project to create service connections.

4. Log in to Azure DevOps with the new user credentials, and set up a billing. You'll only see one Azure subscription in the list.

Some subscriptions are missing from the list of subscriptions

This issue can be fixed by changing the **supported account types** settings and defining who can use your application. To do so, follow the steps below:

1. Sign in to the Azure portal.

2. If you have access to multiple tenants, use the **Directory + subscription** filter in the top menu to select the tenant in which you want to register an application.

3. Select **Microsoft Entra ID** from the left pane.

4. Select **App registrations**.

5. Select you application from the list of registered applications.

6. Under **Authentication**, select **Supported account types**.

7. Under **Supported account types**, *Who can use this application or access this API?* select **Accounts in any organizational directory**.

8. Select **Save** when you are done.

Service principal's token expired

An issue that often arises with service principals that are automatically created is that the service principal's token expires and needs to be renewed. However, if you have an issue with refreshing the token, see valid refresh token was not found.

To renew the access token for an automatically created service principal:

1. Go to **Project settings** > **Service connections**, and then select the service connection you want to modify.
2. Select **Edit** in the upper-right corner, and the select **Verify**.
3. Select **Save**.

Your service principal's token has now been renewed for two more years.

> ⓘ **Note**
>
> This operation is available even if the service principal's token has not expired.

Failed to obtain the JWT by using the service principal client ID

This issue occurs when you try to verify a service connection that has an expired secret.

To resolve this issue:

1. Go to **Project settings** > **Service connections**, and then select the service connection you want to modify.
2. Select **Edit** in the upper-right corner, and then make any change to your service connection. The easiest and recommended change is to add a description.
3. Select **Save** to save the service connection.

   > ⓘ **Note**
   >
   > Select **Save**. Don't try to verify the service connection at this step.

4. Exit the service connection edit window, and then refresh the service connections page.

5. Select **Edit** in the upper-right corner, and now select **Verify**.

6. Select **Save** to save your service connection.

Azure subscription is not passed from the previous task output

When you set your Azure subscription dynamically for your release pipeline and want to consume the output variable from a preceding task, you might encounter this issue.

To resolve the issue, ensure that the values are defined within the variables section of your pipeline. You can then pass this variable between your pipeline's tasks.

What authentication mechanisms are supported? How do managed identities work?

An Azure Resource Manager service connection can connect to an Azure subscription by using a Service Principal Authentication (SPA) or managed identity authentication. Managed identities for Azure resources provide Azure services with an automatically managed identity in Microsoft Entra ID. You can use this identity to authenticate to any service that supports Microsoft Entra authentication without persisting credentials in code or in the service connection.

To learn about managed identities for virtual machines, see Assigning roles.

> **Note**
>
> Managed identities aren't supported in Microsoft-hosted agents. In this scenario, you must **set up a self-hosted agent** on an Azure VM and configure a managed identity for that VM.

Related articles

- Troubleshoot pipeline runs
- Review pipeline logs
- Define variables
- Classic release and artifacts variables

YAML schema reference for Azure Pipelines

Article • 12/21/2023

The YAML schema reference for Azure Pipelines is a detailed reference for YAML pipelines that lists all supported YAML syntax and their available options.

To create a YAML pipeline, start with the pipeline definition. For more information about building YAML pipelines, see Customize your pipeline.

The YAML schema reference does not cover tasks. For more information about tasks, see the Azure Pipelines tasks index.

Definitions

pipeline

A pipeline is one or more stages that describe a CI/CD process. The pipeline definition contains the documentation for root level properties like `name`.

extends

Extends a pipeline using a template.

jobs

Specifies the jobs that make up the work of a stage.

jobs.deployment

A deployment job is a special type of job. It's a collection of steps to run sequentially against the environment.

jobs.deployment.environment

Target environment name and optionally a resource name to record the deployment history; format: environment-name.resource-name.

jobs.deployment.strategy

Execution strategy for this deployment.

jobs.deployment.strategy.canary

Canary Deployment strategy.

jobs.deployment.strategy.rolling

Rolling Deployment strategy.

jobs.deployment.strategy.runOnce
RunOnce Deployment strategy.

jobs.job
A job is a collection of steps run by an agent or on a server.

jobs.job.container
Container resource name.

jobs.job.strategy
Execution strategy for this job.

jobs.job.uses
Any resources required by this job that are not already referenced.

jobs.template
A set of jobs defined in a template.

parameters
Specifies the runtime parameters passed to a pipeline.

parameters.parameter
Pipeline template parameters.

pool
Which pool to use for a job of the pipeline.

pool.demands
Demands (for a private pool).

pr
Pull request trigger.

resources
Resources specifies builds, repositories, pipelines, and other resources used by the pipeline.

resources.builds
List of build resources referenced by the pipeline.

resources.builds.build
A build resource used to reference artifacts from a run.

resources.containers
List of container images.

resources.containers.container

A container resource used to reference a container image.

resources.containers.container.trigger

Specify none to disable, true to trigger on all image tags, or use the full syntax as described in the following examples.

resources.packages

List of package resources.

resources.packages.package

A package resource used to reference a NuGet or npm GitHub package.

resources.pipelines

List of pipeline resources.

resources.pipelines.pipeline

A pipeline resource.

resources.pipelines.pipeline.trigger

Specify none to disable, true to include all branches, or use the full syntax as described in the following examples.

resources.pipelines.pipeline.trigger.branches

Branches to include or exclude for triggering a run.

resources.repositories

List of repository resources.

resources.repositories.repository

A repository resource is used to reference an additional repository in your pipeline.

resources.webhooks

List of webhooks.

resources.webhooks.webhook

A webhook resource enables you to integrate your pipeline with an external service to automate the workflow.

resources.webhooks.webhook.filters

List of trigger filters.

resources.webhooks.webhook.filters.filter

Webhook resource trigger filter.

schedules
The schedules list specifies the scheduled triggers for the pipeline.

schedules.cron
A scheduled trigger specifies a schedule on which branches are built.

stages
Stages are a collection of related jobs.

stages.stage
A stage is a collection of related jobs.

stages.template
You can define a set of stages in one file and use it multiple times in other files.

steps
Steps are a linear sequence of operations that make up a job.

steps.bash
Runs a script in Bash on Windows, macOS, and Linux.

steps.checkout
Configure how the pipeline checks out source code.

steps.download
Downloads artifacts associated with the current run or from another Azure Pipeline that is associated as a pipeline resource.

steps.downloadBuild
Downloads build artifacts.

steps.getPackage
Downloads a package from a package management feed in Azure Artifacts or Azure DevOps Server.

steps.powershell
Runs a script using either Windows PowerShell (on Windows) or pwsh (Linux and macOS).

steps.publish
Publishes (uploads) a file or folder as a pipeline artifact that other jobs and pipelines can consume.

steps.pwsh
Runs a script in PowerShell Core on Windows, macOS, and Linux.

steps.reviewApp

Downloads creates a resource dynamically under a deploy phase provider.

steps.script

Runs a script using cmd.exe on Windows and Bash on other platforms.

steps.task

Runs a task.

steps.template

Define a set of steps in one file and use it multiple times in another file.

target

Tasks run in an execution context, which is either the agent host or a container.

target.settableVariables

Restrictions on which variables that can be set.

trigger

Continuous integration (push) trigger.

variables

Define variables using name/value pairs.

variables.group

Reference variables from a variable group.

variables.name

Define variables using name and full syntax.

variables.template

Define variables in a template.

Supporting definitions

> **Note**
>
> Supporting definitions are not intended for use directly in a pipeline. Supporting definitions are used only as part of other definitions, and are included here for reference.

deployHook

Used to run steps that deploy your application.

includeExcludeFilters

Lists of items to include or exclude.

includeExcludeStringFilters

Items to include or exclude.

mountReadOnly

Volumes to mount read-only, the default is all false.

onFailureHook

Used to run steps for rollback actions or clean-up.

onSuccessHook

Used to run steps for rollback actions or clean-up.

onSuccessOrFailureHook

Used to run steps for rollback actions or clean-up.

postRouteTrafficHook

Used to run the steps after the traffic is routed. Typically, these tasks monitor the health of the updated version for defined interval.

preDeployHook

Used to run steps that initialize resources before application deployment starts.

routeTrafficHook

Used to run steps that serve the traffic to the updated version.

workspace

Workspace options on the agent.

YAML schema documentation conventions

The YAML schema reference is a detailed reference guide to Azure Pipelines YAML pipelines. It includes a catalog of all supported YAML capabilities and the available options.

Here are the syntax conventions used in the YAML schema reference.

- To the left of `:` is a literal keyword used in pipeline definitions.
- To the right of `:` is a data type. The data type can be a primitive type like **string** or a reference to a rich structure defined elsewhere in this reference.
- The notation `[` *datatype* `]` indicates an array of the mentioned definition type. For instance, `[ string ]` is an array of strings.

- The notation `{` *datatype* `:` *datatype* `}` indicates a mapping of one data type to another. For instance, `{ string: string }` is a mapping of strings to strings.
- The symbol `|` indicates there are multiple data types available for the keyword. For instance, `job | template` means either a job definition or a template reference is allowed.

See also

This reference covers the schema of an Azure Pipelines YAML file. To learn the basics of YAML, see Learn YAML in Y Minutes. Azure Pipelines doesn't support all YAML features. Unsupported features include anchors, complex keys, and sets. Also, unlike standard YAML, Azure Pipelines depends on seeing `stage`, `job`, `task`, or a task shortcut like `script` as the first key in a mapping.

Expressions

Article • 10/10/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

> **Important**
>
>
>
>
> Select the version of this article that corresponds to your platform and version. The version selector is above the table of contents. **Look up your Azure DevOps platform and version**.

Expressions can be used in many places where you need to specify a string, boolean, or number value when authoring a pipeline. When an expression returns an array, normal indexing rules apply and the index starts with `0`.

The most common use of expressions is in conditions to determine whether a job or step should run.

YAML

```yaml
# Expressions are used to define conditions for a step, job, or stage
steps:
- task: ...
  condition: <expression>
```

Another common use of expressions is in defining variables. Expressions can be evaluated at compile time or at run time. Compile time expressions can be used anywhere; runtime expressions can be used in variables and conditions. Runtime

expressions are intended as a way to compute the contents of variables and state (example: `condition`).

YAML

```yaml
# Two examples of expressions used to define variables
# The first one, a, is evaluated when the YAML file is compiled into a plan.
# The second one, b, is evaluated at runtime.
# Note the syntax ${{}} for compile time and $[] for runtime expressions.
variables:
  a: ${{ <expression> }}
  b: $[ <expression> ]
```

The difference between runtime and compile time expression syntaxes is primarily what context is available. In a compile-time expression (`${{ <expression> }}`), you have access to `parameters` and statically defined `variables`. In a runtime expression (`$[ <expression> ]`), you have access to more `variables` but no parameters.

In this example, a runtime expression sets the value of `$(isMain)`. A static variable in a compile expression sets the value of `$(compileVar)`.

YAML

```yaml
variables:
  staticVar: 'my value' # static variable
  compileVar: ${{ variables.staticVar }} # compile time expression
  isMain: $[eq(variables['Build.SourceBranch'], 'refs/heads/main')] #
runtime expression

steps:
  - script: |
      echo ${{variables.staticVar}} # outputs my value
      echo $(compileVar) # outputs my value
      echo $(isMain) # outputs True
```

An expression can be a literal, a reference to a variable, a reference to a dependency, a function, or a valid nested combination of these.

Literals

As part of an expression, you can use boolean, null, number, string, or version literals.

YAML

```yaml
# Examples
variables:
```

```
  someBoolean: ${{ true }} # case insensitive, so True or TRUE also works
  someNumber: ${{ -1.2 }}
  someString: ${{ 'a b c' }}
  someVersion: ${{ 1.2.3 }}
```

Boolean

`True` and `False` are boolean literal expressions.

Null

Null is a special literal expression that's returned from a dictionary miss, e.g. (`variables['noSuch']`). Null can be the output of an expression but cannot be called directly within an expression.

Number

Starts with '-', '.', or '0' through '9'.

String

Must be single-quoted. For example: `'this is a string'`.

To express a literal single-quote, escape it with a single quote. For example: `'It''s OK if they''re using contractions.'`.

You can use a pipe character (`|`) for multiline strings.

YAML

```yaml
myKey: |
  one
  two
  three
```

Version

A version number with up to four segments. Must start with a number and contain two or three period (`.`) characters. For example: `1.2.3.4`.

Variables

As part of an expression, you may access variables using one of two syntaxes:

- Index syntax: `variables['MyVar']`
- Property dereference syntax: `variables.MyVar`

In order to use property dereference syntax, the property name must:

- Start with `a-Z` or `_`
- Be followed by `a-Z` `0-9` or `_`

Depending on the execution context, different variables are available.

- If you create pipelines using YAML, then pipeline variables are available.
- If you create build pipelines using classic editor, then build variables are available.
- If you create release pipelines using classic editor, then release variables are available.

Variables are always strings. If you want to use typed values, then you should use parameters instead.

> **Note**
>
> There is a limitation for using variables with expressions for both Classical and YAML pipelines when setting up such variables via variables tab UI. Variables that are defined as expressions shouldn't depend on another variable with expression in value since **it isn't guaranteed** that both expressions will be evaluated properly. For example we have variable `a` whose value `$[ <expression> ]` is used as a part for the value of variable `b`. Since the order of processing variables isn't guaranteed variable `b` could have an incorrect value of variable `a` after evaluation.
>
> Described constructions are only allowed while setup variables through **variables keyword** in YAML pipeline. It is required to place the variables in the order they should be processed to get the correct values after processing.

Functions

The following built-in functions can be used in expressions.

and

- Evaluates to `True` if all parameters are `True`

- Min parameters: 2. Max parameters: N
- Casts parameters to Boolean for evaluation
- Short-circuits after first `False`
- Example: `and(eq(variables.letters, 'ABC'), eq(variables.numbers, 123))`

coalesce

- Evaluates the parameters in order (left to right), and returns the first value that does not equal null or empty-string.
- No value is returned if the parameter values all are null or empty strings.
- Min parameters: 2. Max parameters: N
- Example: `coalesce(variables.couldBeNull, variables.couldAlsoBeNull, 'literal so it always works')`

contains

- Evaluates `True` if left parameter String contains right parameter
- Min parameters: 2. Max parameters: 2
- Casts parameters to String for evaluation
- Performs ordinal ignore-case comparison
- Example: `contains('ABCDE', 'BCD')` (returns True)

containsValue

- Evaluates `True` if the left parameter is an array, and any item equals the right parameter. Also evaluates `True` if the left parameter is an object, and the value of any property equals the right parameter.
- Min parameters: 2. Max parameters: 2
- If the left parameter is an array, convert each item to match the type of the right parameter. If the left parameter is an object, convert the value of each property to match the type of the right parameter. The equality comparison for each specific item evaluates `False` if the conversion fails.
- Ordinal ignore-case comparison for Strings
- Short-circuits after the first match

> **Note**

There is no literal syntax in a YAML pipeline for specifying an array. This function is of limited use in general pipelines. It's intended for use in the **pipeline decorator context** with system-provided arrays such as the list of steps.

You can use the `containsValue` expression to find a matching value in an object. Here is an example that demonstrates looking in list of source branches for a match for `Build.SourceBranch`.

YAML

```yaml
parameters:
- name: branchOptions
  displayName: Source branch options
  type: object
  default:
    - refs/heads/main
    - refs/heads/test

jobs:
  - job: A1
    steps:
    - ${{ each value in parameters.branchOptions }}:
      - script: echo ${{ value }}

  - job: B1
    condition: ${{ containsValue(parameters.branchOptions,
variables['Build.SourceBranch']) }}
    steps:
      - script: echo "Matching branch found"
```

convertToJson

- Take a complex object and outputs it as JSON.
- Min parameters: 1. Max parameters: 1.

YAML

```yaml
parameters:
  - name: listOfValues
    type: object
    default:
      this_is:
        a_complex: object
        with:
          - one
          - two

steps:
```

```yaml
- script: |
    echo "${MY_JSON}"
  env:
    MY_JSON: ${{ convertToJson(parameters.listOfValues) }}
```

Script output:

```JSON
{
  "this_is": {
    "a_complex": "object",
    "with": [
      "one",
      "two"
    ]
  }
}
```

counter

- This function can only be used in an expression that defines a variable. It cannot be used as part of a condition for a step, job, or stage.
- Evaluates a number that is incremented with each run of a pipeline.
- Parameters: 2. `prefix` and `seed`.
- Prefix is a string expression. A separate value of counter is tracked for each unique value of prefix. The `prefix` should use UTF-16 characters.
- Seed is the starting value of the counter

You can create a counter that is automatically incremented by one in each execution of your pipeline. When you define a counter, you provide a `prefix` and a `seed`. Here is an example that demonstrates this.

```YAML
variables:
  major: 1
  # define minor as a counter with the prefix as variable major, and seed as
100.
  minor: $[counter(variables['major'], 100)]

steps:
- bash: echo $(minor)
```

The value of `minor` in the above example in the first run of the pipeline will be 100. In the second run it will be 101, provided the value of `major` is still 1.

If you edit the YAML file, and update the value of the variable `major` to be 2, then in the next run of the pipeline, the value of `minor` will be 100. Subsequent runs will increment the counter to 101, 102, 103, ...

Later, if you edit the YAML file, and set the value of `major` back to 1, then the value of the counter resumes where it left off for that prefix. In this example, it resumes at 102.

Here is another example of setting a variable to act as a counter that starts at 100, gets incremented by 1 for every run, and gets reset to 100 every day.

> ⓘ **Note**
>
> `pipeline.startTime` is not available outside of expressions. `pipeline.startTime` formats `system.pipelineStartTime` into a date and time object so that it is available to work with expressions. The default time zone for `pipeline.startTime` is UTC. You can **change the time zone** for your organization.

YAML

```yaml
jobs:
- job:
  variables:
    a: $[counter(format('{0:yyyyMMdd}', pipeline.startTime), 100)]
  steps:
  - bash: echo $(a)
```

Here is an example of having a counter that maintains a separate value for PRs and CI runs.

YAML

```yaml
variables:
  patch: $[counter(variables['build.reason'], 0)]
```

Counters are scoped to a pipeline. In other words, its value is incremented for each run of that pipeline. There are no project-scoped counters.

endsWith

- Evaluates `True` if left parameter String ends with right parameter

- Min parameters: 2. Max parameters: 2
- Casts parameters to String for evaluation
- Performs ordinal ignore-case comparison
- Example: `endsWith('ABCDE', 'DE')` (returns True)

eq

- Evaluates `True` if parameters are equal
- Min parameters: 2. Max parameters: 2
- Converts right parameter to match type of left parameter. Returns `False` if conversion fails.
- Ordinal ignore-case comparison for Strings
- Example: `eq(variables.letters, 'ABC')`

format

- Evaluates the trailing parameters and inserts them into the leading parameter string
- Min parameters: 1. Max parameters: N
- Example: `format('Hello {0} {1}', 'John', 'Doe')`
- Uses .NET custom date and time format specifiers for date formatting (`yyyy`, `yy`, `MM`, `M`, `dd`, `d`, `HH`, `H`, `m`, `mm`, `ss`, `s`, `f`, `ff`, `ffff`, `K`)
- Example: `format('{0:yyyyMMdd}', pipeline.startTime)`. In this case `pipeline.startTime` is a special date time object variable.
- Escape by doubling braces. For example: `format('literal left brace {{ and literal right brace }}')`

ge

- Evaluates `True` if left parameter is greater than or equal to the right parameter
- Min parameters: 2. Max parameters: 2
- Converts right parameter to match type of left parameter. Errors if conversion fails.
- Ordinal ignore-case comparison for Strings
- Example: `ge(5, 5)` (returns True)

gt

- Evaluates `True` if left parameter is greater than the right parameter
- Min parameters: 2. Max parameters: 2

- Converts right parameter to match type of left parameter. Errors if conversion fails.
- Ordinal ignore-case comparison for Strings
- Example: `gt(5, 2)` (returns True)

in

- Evaluates `True` if left parameter is equal to any right parameter
- Min parameters: 1. Max parameters: N
- Converts right parameters to match type of left parameter. Equality comparison evaluates `False` if conversion fails.
- Ordinal ignore-case comparison for Strings
- Short-circuits after first match
- Example: `in('B', 'A', 'B', 'C')` (returns True)

join

- Concatenates all elements in the right parameter array, separated by the left parameter string.
- Min parameters: 2. Max parameters: 2
- Each element in the array is converted to a string. Complex objects are converted to empty string.
- If the right parameter is not an array, the result is the right parameter converted to a string.

In this example, a semicolon gets added between each item in the array. The parameter type is an object.

YAML

```yaml
parameters:
- name: myArray
  type: object
  default:
    - FOO
    - BAR
    - ZOO

variables:
   A: ${{ join(';',parameters.myArray) }}

steps:
  - script: echo $A # outputs FOO;BAR;ZOO
```

le

- Evaluates `True` if left parameter is less than or equal to the right parameter
- Min parameters: 2. Max parameters: 2
- Converts right parameter to match type of left parameter. Errors if conversion fails.
- Ordinal ignore-case comparison for Strings
- Example: `le(2, 2)` (returns True)

length

- Returns the length of a string or an array, either one that comes from the system or that comes from a parameter
- Min parameters: 1. Max parameters 1
- Example: `length('fabrikam')` returns 8

lower

- Converts a string or variable value to all lowercase characters
- Min parameters: 1. Max parameters 1
- Returns the lowercase equivalent of a string
- Example: `lower('FOO')` returns `foo`

lt

- Evaluates `True` if left parameter is less than the right parameter
- Min parameters: 2. Max parameters: 2
- Converts right parameter to match type of left parameter. Errors if conversion fails.
- Ordinal ignore-case comparison for Strings
- Example: `lt(2, 5)` (returns True)

ne

- Evaluates `True` if parameters are not equal
- Min parameters: 2. Max parameters: 2
- Converts right parameter to match type of left parameter. Returns `True` if conversion fails.
- Ordinal ignore-case comparison for Strings
- Example: `ne(1, 2)` (returns True)

not

- Evaluates `True` if parameter is `False`
- Min parameters: 1. Max parameters: 1
- Converts value to Boolean for evaluation
- Example: `not(eq(1, 2))` (returns True)

notIn

- Evaluates `True` if left parameter is not equal to any right parameter
- Min parameters: 1. Max parameters: N
- Converts right parameters to match type of left parameter. Equality comparison evaluates `False` if conversion fails.
- Ordinal ignore-case comparison for Strings
- Short-circuits after first match
- Example: `notIn('D', 'A', 'B', 'C')` (returns True)

or

- Evaluates `True` if any parameter is `True`
- Min parameters: 2. Max parameters: N
- Casts parameters to Boolean for evaluation
- Short-circuits after first `True`
- Example: `or(eq(1, 1), eq(2, 3))` (returns True, short-circuits)

replace

- Returns a new string in which all instances of a string in the current instance are replaced with another string
- Min parameters: 3. Max parameters: 3
- `replace(a, b, c)`: returns a, with all instances of b replaced by c
- Example: `replace('https://www.tinfoilsecurity.com/saml/consume','https://www.tinfoilsecurity.com','http://server')` (returns `http://server/saml/consume`)

split

- Splits a string into substrings based on the specified delimiting characters
- Min parameters: 2. Max parameters: 2

- The first parameter is the string to split
- The second parameter is the delimiting characters
- Returns an array of substrings. The array includes empty strings when the delimiting characters appear consecutively or at the end of the string
- Example:

```yml
variables:
- name: environments
  value: prod1,prod2
steps:
  - ${{ each env in split(variables.environments, ',')}}:
    - script: ./deploy.sh --environment ${{ env }}
```

- Example of using split() with replace():

```yml
parameters:
- name: resourceIds
  type: object
  default:
  -
/subscriptions/mysubscription/resourceGroups/myResourceGroup/providers/
Microsoft.Network/loadBalancers/kubernetes-internal
  -
/subscriptions/mysubscription02/resourceGroups/myResourceGroup02/provid
ers/Microsoft.Network/loadBalancers/kubernetes
- name: environments
  type: object
  default:
  - prod1
  - prod2

trigger:
- main

steps:
- ${{ each env in parameters.environments }}:
  - ${{ each resourceId in parameters.resourceIds }}:
      - script: echo ${{ replace(split(resourceId, '/')[8], '-', '_')
}}_${{ env }}
```

startsWith

- Evaluates `True` if left parameter string starts with right parameter
- Min parameters: 2. Max parameters: 2
- Casts parameters to String for evaluation

- Performs ordinal ignore-case comparison
- Example: `startsWith('ABCDE', 'AB')` (returns True)

upper

- Converts a string or variable value to all uppercase characters
- Min parameters: 1. Max parameters 1
- Returns the uppercase equivalent of a string
- Example: `upper('bah')` returns `BAH`

xor

- Evaluates `True` if exactly one parameter is `True`
- Min parameters: 2. Max parameters: 2
- Casts parameters to Boolean for evaluation
- Example: `xor(True, False)` (returns True)

Job status check functions

You can use the following status check functions as expressions in conditions, but not in variable definitions.

always

- Always evaluates to `True` (even when canceled). Note: A critical failure may still prevent a task from running. For example, if getting sources failed.

canceled

- Evaluates to `True` if the pipeline was canceled.

failed

- For a step, equivalent to `eq(variables['Agent.JobStatus'], 'Failed')`.
- For a job:
  - With no arguments, evaluates to `True` only if any previous job in the dependency graph failed.
  - With job names as arguments, evaluates to `True` only if any of those jobs failed.

succeeded

- For a step, equivalent to `in(variables['Agent.JobStatus'], 'Succeeded', 'SucceededWithIssues')`
- Use with `dependsOn` when working with jobs and you want to evaluate whether a previous job was successful. Jobs are designed to run in parallel while stages run sequentially.
- For a job:
  - With no arguments, evaluates to `True` only if all previous jobs in the dependency graph succeeded or partially succeeded.
  - With job names as arguments, evaluates to `True` if all of those jobs succeeded or partially succeeded.
  - Evaluates to `False` if the pipeline is canceled.

succeededOrFailed

- For a step, equivalent to `in(variables['Agent.JobStatus'], 'Succeeded', 'SucceededWithIssues', 'Failed')`

- For a job:
  - With no arguments, evaluates to `True` regardless of whether any jobs in the dependency graph succeeded or failed.
  - With job names as arguments, evaluates to `True` whether any of those jobs succeeded or failed.
  - You may want to use `not(canceled())` instead when there are previous skipped jobs in the dependency graph.

> This is like `always()`, except it will evaluate `False` when the pipeline is canceled.

Conditional insertion

You can use `if`, `elseif`, and `else` clauses to conditionally assign variable values or set inputs for tasks. You can also conditionally run a step when a condition is met.

Conditionals only work when using template syntax. Learn more about variable syntax.

For templates, you can use conditional insertion when adding a sequence or mapping. Learn more about conditional insertion in templates.

Conditionally assign a variable

yml

```yml
variables:
  ${{ if eq(variables['Build.SourceBranchName'], 'main') }}: # only works if you have a main branch
    stageName: prod

pool:
  vmImage: 'ubuntu-latest'

steps:
- script: echo ${{variables.stageName}}
```

Conditionally set a task input

yml

```yml
pool:
  vmImage: 'ubuntu-latest'

steps:
- task: PublishPipelineArtifact@1
  inputs:
    targetPath: '$(Pipeline.Workspace)'
    ${{ if eq(variables['Build.SourceBranchName'], 'main') }}:
      artifact: 'prod'
    ${{ else }}:
      artifact: 'dev'
    publishLocation: 'pipeline'
```

Conditionally run a step

If there is no variable set, or the value of `foo` does not match the `if` conditions, the `else` statement will run. Here the value of `foo` returns true in the `elseif` condition.

YAML

```yaml
variables:
  - name: foo
    value: contoso # triggers elseif condition

pool:
  vmImage: 'ubuntu-latest'

steps:
```

```
- script: echo "start"
- ${{ if eq(variables.foo, 'adaptum') }}:
  - script: echo "this is adaptum"
- ${{ elseif eq(variables.foo, 'contoso') }}: # true
  - script: echo "this is contoso"
- ${{ else }}:
  - script: echo "the value is not adaptum or contoso"
```

Each keyword

You can use the `each` keyword to loop through parameters with the object type.

YAML

```yaml
parameters:
- name: listOfStrings
  type: object
  default:
  - one
  - two

steps:
- ${{ each value in parameters.listOfStrings }}:
  - script: echo ${{ value }}
```

Additionally, you can iterate through nested elements within an object.

YAML

```yaml
parameters:
- name: listOfFruits
  type: object
  default:
  - fruitName: 'apple'
    colors: ['red','green']
  - fruitName: 'lemon'
    colors: ['yellow']
steps:
- ${{ each fruit in parameters.listOfFruits }} :
  - ${{ each fruitColor in fruit.colors}} :
    - script: echo ${{ fruit.fruitName}} ${{ fruitColor }}
```

Dependencies

Expressions can use the dependencies context to reference previous jobs or stages. You can use dependencies to:

- Reference the job status of a previous job
- Reference the stage status of a previous stage
- Reference output variables in the previous job in the same stage
- Reference output variables in the previous stage in a stage
- Reference output variables in a job in a previous stage in the following stage

The context is called `dependencies` for jobs and stages and works much like variables. If you refer to an output variable from a job in another stage, the context is called `stageDependencies`.

If you experience issues with output variables having quote characters (`'` or `"`) in them, see this troubleshooting guide.

Dependency syntax overview

The syntax of referencing output variables with dependencies varies depending on the circumstances. Here's an overview of the most common scenarios. Note that there may be times when alternate syntax also works.

Type

Description

stage to stage dependency (different stages)

Reference an output variable from a previous stage in a job in a different stage in a condition in `stages`.

- Syntax: `and(succeeded(), eq(stageDependencies.<stage-name>.outputs['<job-name>.<step-name>.<variable-name>'], 'true'))`
- Example: `and(succeeded(), eq(stageDependencies.A.outputs['A1.printvar.shouldrun'], 'true'))`

job to job dependency (same stage)

Reference an output variable in a different job in the same stage in `stages`.

- Syntax: `and(succeeded(), eq(dependencies.<stage-name>.outputs['<step-name>.<variable-name>'], 'true'))`
- Example: `and(succeeded(), eq(dependencies.A.outputs['printvar.shouldrun'], 'true'))`

Job to stage dependency (different stages)

Reference an output variable in a different stage in a `job`.

- Syntax: `eq(stageDependencies.<stage-name>.<job-name>.outputs['<step-name>.<variable-name>'], 'true')`
- Example: `eq(stageDependencies.A.A1.outputs['printvar.shouldrun'], 'true')`

Stage to stage dependency (deployment job)

Reference output variable in a deployment job in a different stage in `stages`.

- Syntax: `eq(dependencies.<stage-name>.outputs['<deployment-job-name>.<deployment-job-name>.<step-name>.<variable-name>'], 'true')`
- Example: `eq(dependencies.build.outputs['build_job.build_job.setRunTests.runTests'], 'true')`

Stage to stage dependency (deployment job with resource)

Reference an output variable in a deployment job that includes a resource in different stage in `stages`.

- Syntax: `eq(dependencies.<stage-name>.outputs['<deployment-job-name>.<Deploy_resource-name>.<step-name>.<variable-name>'], 'true')`
- Example: `eq(dependencies.build.outputs['build_job.Deploy_winVM.setRunTests.runTests'], 'true')`

There are also different syntaxes for output variables in deployment jobs depending on the deployment strategy. For more information, see Deployment jobs.

Stage to stage dependencies

Structurally, the `dependencies` object is a map of job and stage names to `results` and `outputs`. Expressed as JSON, it would look like:

JSON

```
"dependencies": {
  "<STAGE_NAME>" : {
    "result": "Succeeded|SucceededWithIssues|Skipped|Failed|Canceled",
    "outputs": {
        "jobName.stepName.variableName": "value"
    }
  },
  "...": {
```

```
    // another stage
  }
}
```

> **Note**
>
> The following examples use standard pipeline syntax. If you're using deployment pipelines, both variable and conditional variable syntax will differ. For information about the specific syntax to use, see **Deployment jobs**.

Use this form of `dependencies` to map in variables or check conditions at a stage level.

In this example, there are two stages, A and B. Stage A has the condition `false` and won't ever run as a result. Stage B runs if the result of Stage A is `Succeeded`, `SucceededWithIssues`, or `Skipped`. Stage B will run because Stage A was skipped.

YAML

```yaml
stages:
- stage: A
  condition: false
  jobs:
  - job: A1
    steps:
    - script: echo Job A1
- stage: B
  condition: in(dependencies.A.result, 'Succeeded', 'SucceededWithIssues',
'Skipped')
  jobs:
  - job: B1
    steps:
    - script: echo Job B1
```

Stages can also use output variables from another stage. In this example, there are also two stages. Stage A includes a job, A1, that sets an output variable `shouldrun` to `true`. Stage B runs when `shouldrun` is `true`. Because `shouldrun` is `true`, Stage B runs. Note that `stageDependencies` is used in the condition because you are referring to an output variable in a different stage.

YAML

```yaml
stages:
- stage: A
  jobs:
  - job: A1
    steps:
```

```
      - bash: echo "##vso[task.setvariable
variable=shouldrun;isOutput=true]true"
      # or on Windows:
      # - script: echo ##vso[task.setvariable
variable=shouldrun;isOutput=true]true
        name: printvar

- stage: B
  condition: and(succeeded(),
eq(stageDependencies.A.outputs['A1.printvar.shouldrun'], 'true'))
  dependsOn: A
  jobs:
  - job: B1
    steps:
    - script: echo hello from Stage B
```

> **Note**
>
> By default, each stage in a pipeline depends on the one just before it in the YAML file. If you need to refer to a stage that isn't immediately prior to the current one, you can override this automatic default by adding a `dependsOn` section to the stage.

Job to job dependencies within one stage

At the job level within a single stage, the `dependencies` data doesn't contain stage-level information.

JSON

```
"dependencies": {
  "<JOB_NAME>": {
    "result": "Succeeded|SucceededWithIssues|Skipped|Failed|Canceled",
    "outputs": {
      "stepName.variableName": "value1"
    }
  },
  "...": {
    // another job
  }
}
```

In this example, there are three jobs (a, b, and c). Job a will always be skipped because of `condition: false`. Job b runs because there are no associated conditions. Job c runs because all of its dependencies either succeed (job b) or are skipped (job a).

YAML

```yaml
jobs:
- job: a
  condition: false
  steps:
  - script: echo Job a
- job: b
  steps:
  - script: echo Job b
- job: c
  dependsOn:
  - a
  - b
  condition: |
    and
    (
      in(dependencies.a.result, 'Succeeded', 'SucceededWithIssues',
'Skipped'),
      in(dependencies.b.result, 'Succeeded', 'SucceededWithIssues',
'Skipped')
    )
  steps:
  - script: echo Job c
```

In this example, Job B depends on an output variable from Job A.

YAML

```yaml
jobs:
- job: A
  steps:
  - bash: echo "##vso[task.setvariable
variable=shouldrun;isOutput=true]true"
  # or on Windows:
  # - script: echo ##vso[task.setvariable
variable=shouldrun;isOutput=true]true
    name: printvar

- job: B
  condition: and(succeeded(),
eq(dependencies.A.outputs['printvar.shouldrun'], 'true'))
  dependsOn: A
  steps:
  - script: echo hello from B
```

Job to job dependencies across stages

At the job level, you can also reference outputs from a job in a previous stage. This requires using the `stageDependencies` context.

JSON

```
"stageDependencies": {
  "<STAGE_NAME>" : {
    "<JOB_NAME>": {
      "result": "Succeeded|SucceededWithIssues|Skipped|Failed|Canceled",
      "outputs": {
          "stepName.variableName": "value"
      }
    },
    "...": {
      // another job
    }
  },
  "...": {
    // another stage
  }
}
```

In this example, job B1 will run if job A1 is skipped. Job B2 will check the value of the output variable from job A1 to determine whether it should run.

YAML

```
stages:
- stage: A
  jobs:
  - job: A1
    steps:
     - bash: echo "##vso[task.setvariable
variable=shouldrun;isOutput=true]true"
     # or on Windows:
     # - script: echo ##vso[task.setvariable
variable=shouldrun;isOutput=true]true
       name: printvar

- stage: B
  dependsOn: A
  jobs:
  - job: B1
    condition: in(stageDependencies.A.A1.result, 'Skipped') # change
condition to `Succeeded and stage will be skipped`
    steps:
    - script: echo hello from Job B1
  - job: B2
    condition: eq(stageDependencies.A.A1.outputs['printvar.shouldrun'],
'true')
    steps:
     - script: echo hello from Job B2
```

If a job depends on a variable defined by a deployment job in a different stage, then the syntax is different. In the following example, the job `run_tests` runs if the `build_job` deployment job set `runTests` to `true`. Notice that the key used for the `outputs` dictionary is `build_job.setRunTests.runTests`.

yml

```yml
stages:
- stage: build
  jobs:
  - deployment: build_job
    environment:
      name: Production
    strategy:
      runOnce:
        deploy:
          steps:
          - task: PowerShell@2
            name: setRunTests
            inputs:
              targetType: inline
              pwsh: true
              script: |
                $runTests = "true"
                echo "setting runTests: $runTests"
                echo "##vso[task.setvariable
variable=runTests;isOutput=true]$runTests"

- stage: test
  dependsOn:
  - 'build'
  jobs:
    - job: run_tests
      condition:
eq(stageDependencies.build.build_job.outputs['build_job.setRunTests.runTests
'], 'true')
      steps:
        ...
```

Deployment job output variables

If a stage depends on a variable defined by a deployment job in a different stage, then the syntax is different. In the following example, the stage `test` depends on the deployment `build_job` setting `shouldTest` to `true`. Notice that in the `condition` of the `test` stage, `build_job` appears twice.

yml

```yaml
stages:
- stage: build
  jobs:
  - deployment: build_job
    environment:
      name: Production
    strategy:
      runOnce:
        deploy:
          steps:
          - task: PowerShell@2
            name: setRunTests
            inputs:
              targetType: inline
              pwsh: true
              script: |
                $runTests = "true"
                echo "setting runTests: $runTests"
                echo "##vso[task.setvariable
variable=runTests;isOutput=true]$runTests"

- stage: test
  dependsOn:
  - 'build'
  condition:
eq(dependencies.build.outputs['build_job.build_job.setRunTests.runTests'],
'true')
  jobs:
    - job: A
      steps:
        - script: echo Hello from job A
```

In the example above, the condition references an environment and not an environment resource. To reference an environment resource, you'll need to add the environment resource name to the dependencies condition. In the following example, condition references an environment virtual machine resource named `vmtest`.

yml

```yaml
stages:
- stage: build
  jobs:
  - deployment: build_job
    environment:
      name: vmtest
      resourceName: winVM2
      resourceType: VirtualMachine
    strategy:
      runOnce:
        deploy:
          steps:
```

```
          - task: PowerShell@2
            name: setRunTests
            inputs:
              targetType: inline
              pwsh: true
              script: |
                $runTests = "true"
                echo "setting runTests: $runTests"
                echo "##vso[task.setvariable
variable=runTests;isOutput=true]$runTests"

- stage: test
  dependsOn:
  - 'build'
  condition:
eq(dependencies.build.outputs['build_job.Deploy_winVM2.setRunTests.runTests'
], 'true')
  jobs:
  - job: A
    steps:
     - script: echo Hello from job A
```

Filtered arrays

When operating on a collection of items, you can use the `*` syntax to apply a filtered array. A filtered array returns all objects/elements regardless their names.

As an example, consider an array of objects named `foo`. We want to get an array of the values of the `id` property in each object in our array.

JSON

```
[
    { "id": 1, "a": "avalue1"},
    { "id": 2, "a": "avalue2"},
    { "id": 3, "a": "avalue3"}
]
```

We could do the following:

`foo.*.id`

This tells the system to operate on `foo` as a filtered array and then select the `id` property.

This would return:

JSON

```
[ 1, 2, 3 ]
```

Type casting

Values in an expression may be converted from one type to another as the expression gets evaluated. When an expression is evaluated, the parameters are coalesced to the relevant data type and then turned back into strings.

For example, in this YAML, the values `True` and `False` are converted to `1` and `0` when the expression is evaluated. The function `lt()` returns `True` when the left parameter is less than the right parameter.

YAML

```yaml
variables:
  firstEval: $[lt(False, True)] # 0 vs. 1, True
  secondEval: $[lt(True, False)] # 1 vs. 0, False

steps:
- script: echo $(firstEval)
- script: echo $(secondEval)
```

In this example, the values `variables.emptyString` and the empty string both evaluate as empty strings. The function `coalesce()` evaluates the parameters in order, and returns the first value that does not equal null or empty-string.

YAML

```yaml
variables:
  coalesceLiteral: $[coalesce(variables.emptyString, '', 'literal value')]

steps:
- script: echo $(coalesceLiteral) # outputs literal value
```

Detailed conversion rules are listed further below.

From / To	Boolean	Null	Number	String	Version
Boolean	-	-	Yes	Yes	-
Null	Yes	-	Yes	Yes	-
Number	Yes	-	-	Yes	Partial

From / To	Boolean	Null	Number	String	Version
String	Yes	Partial	Partial	-	Partial
Version	Yes	-	-	Yes	-

Boolean

To number:

- `False` → `0`
- `True` → `1`

To string:

- `False` → `'False'`
- `True` → `'True'`

Null

- To Boolean: `False`
- To number: `0`
- To string: `''` (the empty string)

Number

- To Boolean: `0` → `False`, any other number → `True`
- To version: Must be greater than zero and must contain a non-zero decimal. Must be less than Int32.MaxValue (decimal component also).
- To string: Converts the number to a string with no thousands separator and no decimal separator.

String

- To Boolean: `''` (the empty string) → `False`, any other string → `True`
- To null: `''` (the empty string) → `Null`, any other string not convertible
- To number: `''` (the empty string) → 0, otherwise, runs C#'s `Int32.TryParse` using InvariantCulture and the following rules: AllowDecimalPoint | AllowLeadingSign | AllowLeadingWhite | AllowThousands | AllowTrailingWhite. If `TryParse` fails, then it's not convertible.

- To version: runs C#'s `Version.TryParse`. Must contain Major and Minor component at minimum. If `TryParse` fails, then it's not convertible.

Version

- To Boolean: `True`
- To string: Major.Minor or Major.Minor.Build or Major.Minor.Build.Revision.

FAQ

I want to do something that is not supported by expressions. What options do I have for extending Pipelines functionality?

You can customize your Pipeline with a script that includes an expression. For example, this snippet takes the `BUILD_BUILDNUMBER` variable and splits it with Bash. This script outputs two new variables, `$MAJOR_RUN` and `$MINOR_RUN`, for the major and minor run numbers. The two variables are then used to create two pipeline variables, `$major` and `$minor` with task.setvariable. These variables are available to downstream steps. To share variables across pipelines see Variable groups.

YAML

```yaml
steps:
- bash: |
    MAJOR_RUN=$(echo $BUILD_BUILDNUMBER | cut -d '.' -f1)
    echo "This is the major run number: $MAJOR_RUN"
    echo "##vso[task.setvariable variable=major]$MAJOR_RUN"

    MINOR_RUN=$(echo $BUILD_BUILDNUMBER | cut -d '.' -f2)
    echo "This is the minor run number: $MINOR_RUN"
    echo "##vso[task.setvariable variable=minor]$MINOR_RUN"

- bash: echo "My pipeline variable for major run is $(major)"
- bash: echo "My pipeline variable for minor run is $(minor)"
```

File matching patterns reference

Article • 04/01/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Pattern syntax

A pattern is a string or list of newline-delimited strings. File and directory names are compared to patterns to include (or sometimes exclude) them in a task. You can build up complex behavior by stacking multiple patterns. See fnmatch for a full syntax guide.

Match characters

Most characters are used as exact matches. What counts as an "exact" match is platform-dependent: the Windows filesystem is case-insensitive, so the pattern "ABC" would match a file called "abc". On case-sensitive filesystems, that pattern and name would not match.

The following characters have special behavior.

- `*` matches zero or more characters within a file or directory name. See examples.
- `?` matches any single character within a file or directory name. See examples.
- `[]` matches a set or range of characters within a file or directory name. See examples.
- `**` recursive wildcard. For example, `/hello/**/*` matches all descendants of `/hello`.

Extended globbing

- `?(hello|world)` - matches `hello` or `world` zero or one times
- `*(hello|world)` - zero or more occurrences
- `+(hello|world)` - one or more occurrences
- `@(hello|world)` - exactly once
- `!(hello|world)` - not `hello` or `world`

Note, extended globs cannot span directory separators. For example, `+(hello/world|other)` is not valid.

Comments

Patterns that begin with `#` are treated as comments.

Exclude patterns

Leading `!` changes the meaning of an include pattern to exclude. You can include a pattern, exclude a subset of it, and then re-include a subset of that: this is known as an "interleaved" pattern.

Multiple `!` flips the meaning. See examples.

You must define an include pattern before an exclude one. See examples.

Escaping

Wrapping special characters in `[]` can be used to escape literal glob characters in a file name. For example the literal file name `hello[a-z]` can be escaped as `hello[[]a-z]`.

Slash

`/` is used as the path separator on Linux and macOS. Most of the time, Windows agents accept `/`. Occasions where the Windows separator (`\`) must be used are documented.

Examples

Basic pattern examples

Asterisk examples

Example 1: Given the pattern `*Website.sln` and files:

```
ConsoleHost.sln
ContosoWebsite.sln
FabrikamWebsite.sln
Website.sln
```

The pattern would match:

```
ContosoWebsite.sln
FabrikamWebsite.sln
Website.sln
```

Example 2: Given the pattern `*Website/*.proj` and paths:

```
ContosoWebsite/index.html
ContosoWebsite/ContosoWebsite.proj
FabrikamWebsite/index.html
FabrikamWebsite/FabrikamWebsite.proj
```

The pattern would match:

```
ContosoWebsite/ContosoWebsite.proj
FabrikamWebsite/FabrikamWebsite.proj
```

Question mark examples

Example 1: Given the pattern `log?.log` and files:

```
log1.log
log2.log
log3.log
script.sh
```

The pattern would match:

```
log1.log
log2.log
log3.log
```

Example 2: Given the pattern `image.???` and files:

```
image.tiff
image.png
image.ico
```

The pattern would match:

```

image.png
image.ico
```

Character set examples

Example 1: Given the pattern `Sample[AC].dat` and files:

```

SampleA.dat
SampleB.dat
SampleC.dat
SampleD.dat
```

The pattern would match:

```

SampleA.dat
SampleC.dat
```

Example 2: Given the pattern `Sample[A-C].dat` and files:

```

SampleA.dat
SampleB.dat
SampleC.dat
SampleD.dat
```

The pattern would match:

```

SampleA.dat
SampleB.dat
```

```
SampleC.dat
```

Example 3: Given the pattern `Sample[A-CEG].dat` and files:

```

SampleA.dat
SampleB.dat
SampleC.dat
SampleD.dat
SampleE.dat
SampleF.dat
SampleG.dat
SampleH.dat
```

The pattern would match:

```

SampleA.dat
SampleB.dat
SampleC.dat
SampleE.dat
SampleG.dat
```

Recursive wildcard examples

Given the pattern `**/*.ext` and files:

```

sample1/A.ext
sample1/B.ext
sample2/C.ext
sample2/D.not
```

The pattern would match:

```

sample1/A.ext
sample1/B.ext
sample2/C.ext
```

Exclude pattern examples

Given the pattern:

```
*
!*.xml
```

and files:

```
ConsoleHost.exe
ConsoleHost.pdb
ConsoleHost.xml
Fabrikam.dll
Fabrikam.pdb
Fabrikam.xml
```

The pattern would match:

```
ConsoleHost.exe
ConsoleHost.pdb
Fabrikam.dll
Fabrikam.pdb
```

Double exclude

Given the pattern:

```
*
!*.xml
!!Fabrikam.xml
```

and files:

```
ConsoleHost.exe
ConsoleHost.pdb
ConsoleHost.xml
Fabrikam.dll
```

```
Fabrikam.pdb
Fabrikam.xml
```

The pattern would match:

```

ConsoleHost.exe
ConsoleHost.pdb
Fabrikam.dll
Fabrikam.pdb
Fabrikam.xml
```

Folder exclude

Given the pattern:

```

**
!sample/**
```

and files:

```

ConsoleHost.exe
ConsoleHost.pdb
ConsoleHost.xml
sample/Fabrikam.dll
sample/Fabrikam.pdb
sample/Fabrikam.xml
```

The pattern would match:

```

ConsoleHost.exe
ConsoleHost.pdb
ConsoleHost.xml
```

File transforms and variable substitution reference

Article • 11/28/2022

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Some tasks, such as the Azure App Service Deploy task version 3 and later and the IIS Web App Deploy task, allow users to configure the package based on the environment specified. These tasks use **msdeploy.exe**, which supports the overriding of values in the **web.config** file with values from the **parameters.xml** file. However, file transforms and variable substitution are **not confined to web app files**. You can use these techniques with any XML or JSON files.

> **Note**
>
> File transforms and variable substitution are also supported by the separate **File Transform task** for use in Azure Pipelines. You can use the File Transform task to apply file transformations and variable substitutions on any configuration and parameters files.

Configuration substitution is specified in the **File Transform and Variable Substitution Options** section of the settings for the tasks. The transformation and substitution options are:

- XML transformation
- XML variable substitution
- JSON variable substitution

When the task runs, it first performs XML transformation, XML variable substitution, and JSON variable substitution on configuration and parameters files. Next, it invokes **msdeploy.exe**, which uses the **parameters.xml** file to substitute values in the **web.config** file.

XML Transformation

XML transformation supports transforming the configuration files (`*.config` files) by following Web.config Transformation Syntax and is based on the environment to which the web package will be deployed. This option is useful when you want to add, remove or modify configurations for different environments. Transformation will be applied for other configuration files including Console or Windows service application configuration files (for example, **FabrikamService.exe.config**).

Configuration transform file naming conventions

XML transformation will be run on the `*.config` file for transformation configuration files named `*.Release.config` or `*.<stage>.config` and will be executed in the following order:

1. `*.Release.config` (for example, **fabrikam.Release.config**)
2. `*.<stage>.config` (for example, **fabrikam.Production.config**)

For example, if your package contains the following files:

- Web.config

- Web.Debug.config
- Web.Release.config
- Web.Production.config

and your stage name is **Production**, the transformation is applied for `Web.config` with `Web.Release.config` followed by `Web.Production.config`.

XML transformation example

1. Create a Web Application package with the necessary configuration and transform files. For example, use the following configuration files:

   Configuration file

   XML

   ```xml
   <?xml version="1.0" encoding="utf-8"?>
   <configuration>
     <connectionStrings>
       <add name="DefaultConnection"
            connectionString="Data Source=(LocalDb)\\MSDB;DbFilename=aspcore-local.mdf;" />
     </connectionStrings>
     <appSettings>
       <add key="webpages:Version" value="3.0.0.0" />
       <add key="webpages:Enabled" value="false" />
     </appSettings>
     <system.web>
       <authentication mode="None" />
       <compilation targetFramework="4.5" debug="true" />
     </system.web>
   </configuration>
   ```

   Transform file

   XML

   ```xml
   <?xml version="1.0"?>
   <configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
       <connectionStrings>
         <add name="MyDB"
              connectionString="Data Source=ReleaseSQLServer;Initial
   Catalog=MyReleaseDB;Integrated Security=True"
              xdt:Transform="Insert" />
       </connectionStrings>
     <appSettings>
       <add xdt:Transform="Replace" xdt:Locator="Match(key)" key="webpages:Enabled"
   value="true" />
     </appSettings>
     <system.web>
       <compilation xdt:Transform="RemoveAttributes(debug)" />
     </system.web>
   </configuration>
   ```

   This example transform configuration file does three things:

   - It adds a new database connection string inside the `ConnectionStrings` element.
   - It modifies value of `Webpages:Enabled` inside the `appSettings` element.

- It removes the `debug` attribute from the `compilation` element inside the `System.Web` element.

> For more information, see Web.config Transformation Syntax for Web Project Deployment Using Visual Studio

2. Create a release pipeline with a stage named **Release**.

3. Add an **Azure App Service Deploy** task and set (tick) the **XML transformation** option.

4. Save the release pipeline and start a new release.

5. Open the `Web.config` file to see the transformations from `Web.Release.config`.

```XML
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <connectionStrings>
    <add name="DefaultConnection"
         connectionString="Data Source=(LocalDb)\\MSDB;DbFilename=aspcore-local.mdf;" />
  <add name="MyDB"
       connectionString="Data Source=ReleaseSQLServer;Initial
Catalog=MyReleaseDB;Integrated Security=True" />
  </connectionStrings>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="true" />
  </appSettings>
  <system.web>
    <authentication mode="None" />
    <compilation targetFramework="4.5" />
  </system.web>
</configuration>
```

XML transformation notes

- You can use this technique to create a default package and deploy it to multiple stages.

- XML transformation takes effect only when the configuration file and transform file are in the same folder within the specified package.

- By default, MSBuild applies the transformation as it generates the web package if the `<DependentUpon>` element is already present in the transform file in the `*.csproj` file. In such cases, the **Azure App**

Service Deploy task will fail because there is no further transformation applied on the `Web.config` file. Therefore, it is recommended that the `<DependentUpon>` element is removed from all the transform files to disable any build-time configuration when using XML transformation.

- Set the **Build Action** property for each of the transformation files (`Web.config`) to **Content** so that the files are copied to the root folder.

```XML
...
<Content Include="Web.Debug.config">
   <DependentUpon>Web.config</DependentUpon>
</Content>
<Content Include="Web.Release.config">
   <DependentUpon>Web.config</DependentUpon>
</Content>
...
```

XML variable substitution

This feature enables you to modify configuration settings in configuration files (`*.config` files) inside web packages and XML parameters files (`parameters.xml`). In this way, the same package can be configured based on the environment to which it will be deployed.

Variable substitution takes effect only on the `applicationSettings`, `appSettings`, `connectionStrings`, and `configSections` elements of configuration files. If you are looking to substitute values outside of these elements you can use a (`parameters.xml`) file, however you will need to use a 3rd party pipeline task to handle the variable substitution.

XML variable substitution example

As an example, consider the task of changing the following values in `Web.config`:

```XML
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSection>
        <section name="entityFramework" />
    </configSection>
    <connectionStrings>
        <!-- Change connectionString in this line: -->
        <add name="DefaultConnection"
             connectionString="Data Source=(LocalDB)\LocalDB;FileName=Local.mdf" />
    </connectionStrings>
    <appSettings>
        <add key="ClientValidationEnabled" value="true" />
        <add key="UnobstructiveJavascriptEnabled" value="true" />
        <!-- Change AdminUserName in this line: -->
        <add key="AdminUserName" value="__AdminUserName__" />
        <!-- Change AdminPassword in this line: -->
        <add key="AdminPassword" value="__AdminPassword__" />
    </appSettings>
    <entityFramework>
        <defaultConnectionFactory type="System.Data.Entity.LocalDbConnectionFactory">
            <parameters></parameters>
```

```
        </defaultConnectionFactory>
        <providers>
            <!-- Change invariantName in this line: -->
            <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer" />
        </providers>
    </entityFramework>
</configuration>
```

1. Create a release pipeline with a stage named **Release**.

2. Add an **Azure App Service Deploy** task and set (tick) the **XML variable substitution** option.

3. Define the required values in release pipeline variables:

Name	Value	Secure	Scope
DefaultConnection	Data Source=(ProdDB)\MSSQLProdDB;AttachFileName=Local.mdf	No	Release
AdminUserName	ProdAdminName	No	Release
AdminPassword	[your-password]	Yes	Release
invariantName	System.Data.SqlClientExtension	No	Release

4. Save the release pipeline and start a new release.

5. Open the `Web.config` file to see the variable substitutions.

```XML
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSection>
        <section name="entityFramework" />
    </configSection>
    <connectionStrings>
        <add name="DefaultConnection"
             connectionString="Data Source=(ProdDB)\MSSQLProdDB;AttachFileName=Local.mdf"
/>
    </connectionStrings>
    <appSettings>
        <add key="ClientValidationEnabled" value="true" />
        <add key="UnobstructiveJavascriptEnabled" value="true" />
        <add key="AdminUserName" value="ProdAdminName" />
        <add key="AdminPassword" value="*password_masked_for_display*" />
    </appSettings>
```

```
    <entityFramework>
        <defaultConnectionFactory type="System.Data.Entity.LocalDbConnectionFactory">
            <parameters></parameters>
        </defaultConnectionFactory>
        <providers>
            <provider invariantName="System.Data.SqlClientExtension"
                      type="System.Data.Entity.SqlServer" />
        </providers>
    </entityFramework>
  </configuration>
```

XML variable substitution notes

- By default, ASP.NET applications have a default parameterized connection attribute. These values are overridden only in the `parameters.xml` file inside the web package.
- Because substitution occurs before deployment, the user can override the values in `Web.config` using `parameters.xml` (inside the web package) or a `setparameters` file.

JSON variable substitution

This feature substitutes values in the JSON configuration files. It overrides the values in the specified JSON configuration files (for example, `appsettings.json`) with the values matching names of release pipeline and stage variables.

To substitute variables in specific JSON files, provide newline-separated list of JSON files. File names must be specified relative to the root folder. For example, if your package has this structure:

Folders

```
/WebPackage(.zip)
  /---- content
    /----- website
      /---- appsettings.json
      /---- web.config
      /---- [other folders]
  /--- archive.xml
  /--- systeminfo.xml
```

and you want to substitute values in **appsettings.json**, enter the relative path from the root folder; for example `content/website/appsettings.json`. Alternatively, use wildcard patterns to search for specific JSON files. For example, `**/appsettings.json` returns the relative path and name of files named **appsettings.json**.

JSON variable substitution example

As an example, consider the task of overriding values in this JSON file:

JSON

```
{
  "Data": {
    "DefaultConnection": {
      "ConnectionString": "Data Source=(LocalDb)\\MSDB;AttachDbFilename=aspcore-local.mdf;"
    },
```

```
    "DebugMode": "enabled",
    "DBAccess": {
      "Administrators": ["Admin-1", "Admin-2"],
      "Users": ["Vendor-1", "vendor-3"]
    },
    "FeatureFlags": {
      "Preview": [
        {
          "newUI": "AllAccounts"
        },
        {
          "NewWelcomeMessage": "Newusers"
        }
      ]
    }
  }
}
```

The task is to override the values of **ConnectionString**, **DebugMode**, the first of the **Users** values, and **NewWelcomeMessage** at the respective places within the JSON file hierarchy.

Classic

1. Create a release pipeline with a stage named **Release**.
2. Add an **Azure App Service Deploy** task and enter a newline-separated list of JSON files to substitute the variable values in the **JSON variable substitution** textbox. Files names must be relative to the root folder. You can use wildcards to search for JSON files. For example: `**/*.json` means substitute values in all the JSON files within the package.

3. Define the required substitution values in release pipeline or stage variables.

Name	Value	Secure	Scope
Data.DebugMode	disabled	No	Release
Data.DefaultConnection.ConnectionString	Data Source=(prodDB)\MSDB;AttachDbFilename=prod.mdf;	No	Release
Data.DBAccess.Users.0	Admin-3	Yes	Release
Data.FeatureFlags.Preview.1.NewWelcomeMessage	AllAccounts	No	Release

4. Save the release pipeline and start a new release.

5. After the transformation, the JSON will contain the following:

JSON

```
{
  "Data": {
    "DefaultConnection": {
      "ConnectionString": "Data Source=(prodDB)\MSDB;AttachDbFilename=prod.mdf;"
    },
    "DebugMode": "disabled",
    "DBAccess": {
      "Administrators": ["Admin-1", "Admin-2"],
      "Users": ["Admin-3", "vendor-3"]
    },
    "FeatureFlags": {
      "Preview": [
        {
          "newUI": "AllAccounts"
        },
        {
          "NewWelcomeMessage": "AllAccounts"
        }
      ]
    }
  }
}
'''
```

JSON variable substitution notes

- To substitute values in nested levels of the file, concatenate the names with a period (`.`) in hierarchical order.
- A JSON object may contain an array whose values can be referenced by their index. For example, to substitute the first value in the **Users** array shown above, use the variable name `DBAccess.Users.0`. To update the value in **NewWelcomeMessage**, use the variable name `FeatureFlags.Preview.1.NewWelcomeMessage`. However, the file transform task has the ability to transform entire arrays in JSON files. You can also use `DBAccess.Users = ["NewUser1","NewUser2","NewUser3"]`.
- Only **String** substitution is supported for JSON variable substitution.
- Substitution is supported for only UTF-8 and UTF-16 LE encoded files.
- If the file specification you enter does not match any file, the task will fail.
- Variable name matching is case-sensitive.
- Variable substitution is applied for only the JSON keys predefined in the object hierarchy. It does not create new keys.
- If a variable name includes periods ("."), the transformation will attempt to locate the item within the hierarchy. For example, if the variable name is `first.second.third`, the transformation process will

search for:

JSON

```
"first" : {
  "second": {
    "third" : "value"
  }
}
```

as well as `"first.second.third" : "value"`.

Logging commands

Article • 10/26/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

Logging commands are how tasks and scripts communicate with the agent. They cover actions like creating new variables, marking a step as failed, and uploading artifacts. Logging commands are useful when you're troubleshooting a pipeline.

> **Important**
>
> We make an effort to mask secrets from appearing in Azure Pipelines output, but you still need to take precautions. Never echo secrets as output. Some operating systems log command line arguments. Never pass secrets on the command line. Instead, we suggest that you map your secrets into environment variables.
>
> We never mask substrings of secrets. If, for example, "abc123" is set as a secret, "abc" isn't masked from the logs. This is to avoid masking secrets at too granular of a level, making the logs unreadable. For this reason, secrets should not contain structured data. If, for example, "{ "foo": "bar" }" is set as a secret, "bar" isn't masked from the logs.

Type	Commands
Task commands	AddAttachment, Complete, LogDetail, LogIssue, PrependPath, SetEndpoint, SetProgress, SetVariable, SetSecret, UploadFile, UploadSummary
Artifact commands	Associate, Upload
Build commands	AddBuildTag, UpdateBuildNumber, UploadLog
Release commands	UpdateReleaseName

Logging command format

The general format for a logging command is:

```
##vso[area.action property1=value;property2=value;...]message
```

There are also a few formatting commands with a slightly different syntax:

```

##[command]message
```

To invoke a logging command, echo the command via standard output.

Bash

Bash

```bash
#!/bin/bash
echo "##vso[task.setvariable variable=testvar;]testvalue"
```

File paths should be given as absolute paths: rooted to a drive on Windows, or beginning with `/` on Linux and macOS.

> **Note**
>
> Please note that you can't use the `set -x` command before a logging command when you are using Linux or macOS. See **troubleshooting**, to learn how to disable `set -x` temporarily for Bash.

Formatting commands

> **Note**
>
> Use UTF-8 encoding for logging commands.

These commands are messages to the log formatter in Azure Pipelines. They mark specific log lines as errors, warnings, collapsible sections, and so on.

The formatting commands are:

```
```

```
##[group]Beginning of a group
##[warning]Warning message
##[error]Error message
##[section]Start of a section
##[debug]Debug text
##[command]Command-line being run
##[endgroup]
```

You can use the formatting commands in a bash or PowerShell task.

Bash

YAML

```yaml
steps:
- bash: |
    echo "##[group]Beginning of a group"
    echo "##[warning]Warning message"
    echo "##[error]Error message"
    echo "##[section]Start of a section"
    echo "##[debug]Debug text"
    echo "##[command]Command-line being run"
    echo "##[endgroup]"
```

Those commands will render in the logs like this:

That block of commands can also be collapsed, and looks like this:

Task commands

LogIssue: Log an error or warning

`##vso[task.logissue]error/warning message`

Usage

Log an error or warning message in the timeline record of the current task.

Properties

- `type` = `error` or `warning` (Required)
- `sourcepath` = source file location
- `linenumber` = line number
- `columnnumber` = column number
- `code` = error or warning code

Example: Log an error

Bash

Bash

```bash
#!/bin/bash
echo "##vso[task.logissue type=error]Something went very wrong."
exit 1
```

Tip

`exit 1` is optional, but is often a command you'll issue soon after an error is logged. If you select **Control Options: Continue on error**, then the `exit 1` will result in a partially successful build instead of a failed build. As an alternative, you can also use `task.logissue type=error`.

Example: Log a warning about a specific place in a file

Bash

Bash

```bash
#!/bin/bash
echo "##vso[task.logissue
```

```
type=warning;sourcepath=consoleapp/main.cs;linenumber=1;columnnumber=1;code=100;]Found something that could be a problem."
```

SetProgress: Show percentage completed

`##vso[task.setprogress]current operation`

Usage

Set progress and current operation for the current task.

Properties

- `value` = percentage of completion

Example

Bash

Bash

```bash
echo "Begin a lengthy process..."
for i in {0..100..10}
do
   sleep 1
   echo "##vso[task.setprogress value=$i;]Sample Progress Indicator"
done
echo "Lengthy process is complete."
```

To see how it looks, save and queue the build, and then watch the build run. Observe that a progress indicator changes when the task runs this script.

Complete: Finish timeline

`##vso[task.complete]current operation`

Usage

Finish the timeline record for the current task, set task result and current operation. When result not provided, set result to succeeded.

Properties

- `result` =
  - `Succeeded` The task succeeded.
  - `SucceededWithIssues` The task ran into problems. The build will be completed as partially succeeded at best.
  - `Failed` The build will be completed as failed. (If the **Control Options: Continue on error** option is selected, the build will be completed as partially succeeded at best.)

Example

Log a task as succeeded.

```
##vso[task.complete result=Succeeded;]DONE
```

Set a task as failed. As an alternative, you can also use `exit 1`.

```yaml
- bash: |
    if [ -z "$SOLUTION" ]; then
      echo "##vso[task.logissue type=error;]Missing template parameter
\"solution\""
      echo "##vso[task.complete result=Failed;]"
    fi
```

LogDetail: Create or update a timeline record for a task

`##vso[task.logdetail]current operation`

Usage

Creates and updates timeline records. This is primarily used internally by Azure Pipelines to report about steps, jobs, and stages. While customers can add entries to the timeline, they won't typically be shown in the UI.

The first time we see `##vso[task.detail]` during a step, we create a "detail timeline" record for the step. We can create and update nested timeline records base on `id` and `parentid`.

Task authors must remember which GUID they used for each timeline record. The logging system will keep track of the GUID for each timeline record, so any new GUID will result a new timeline record.

Properties

- `id` = Timeline record GUID (Required)
- `parentid` = Parent timeline record GUID
- `type` = Record type (Required for first time, can't overwrite)
- `name` = Record name (Required for first time, can't overwrite)
- `order` = order of timeline record (Required for first time, can't overwrite)
- `starttime` = `Datetime`
- `finishtime` = `Datetime`
- `progress` = percentage of completion
- `state` = `Unknown` | `Initialized` | `InProgress` | `Completed`
- `result` = `Succeeded` | `SucceededWithIssues` | `Failed`

Examples

Create new root timeline record:

```
##vso[task.logdetail id=new guid;name=project1;type=build;order=1]create new
timeline record
```

Create new nested timeline record:

```
##vso[task.logdetail id=new guid;parentid=exist timeline record
guid;name=project1;type=build;order=1]create new nested timeline record
```

Update exist timeline record:

```
##vso[task.logdetail id=existing timeline record
guid;progress=15;state=InProgress;]update timeline record
```

SetVariable: Initialize or modify the value of a variable

`##vso[task.setvariable]value`

Usage

Sets a variable in the variable service of taskcontext. The first task can set a variable, and following tasks are able to use the variable. The variable is exposed to the following tasks as an environment variable.

When `issecret` is set to `true`, the value of the variable will be saved as secret and masked out from log. Secret variables aren't passed into tasks as environment variables and must instead be passed as inputs.

When `isoutput` is set to `true` the syntax to reference the set variable varies based on whether you are accessing that variable in the same job, a future job, or a future stage. Additionally, if `isoutput` is set to `false` the syntax for using that variable within the same job is distinct. See levels of output variables to determine the appropriate syntax for each use case.

See set variables in scripts and define variables for more details.

Properties

- `variable` = variable name (Required)
- `issecret` = boolean (Optional, defaults to false)
- `isoutput` = boolean (Optional, defaults to false)
- `isreadonly` = boolean (Optional, defaults to false)

Examples

Bash

Set the variables:

YAML

```yaml
- bash: |
    echo "##vso[task.setvariable variable=sauce;]crushed tomatoes"
    echo "##vso[task.setvariable
variable=secretSauce;issecret=true]crushed tomatoes with garlic"
    echo "##vso[task.setvariable
variable=outputSauce;isoutput=true]canned goods"
  name: SetVars
```

Read the variables:

YAML

```yaml
- bash: |
    echo "Non-secrets automatically mapped in, sauce is $SAUCE"
    echo "Secrets are not automatically mapped in, secretSauce is
$SECRETSAUCE"
    echo "You can use macro replacement to get secrets, and they'll be
masked in the log: $(secretSauce)"
```

Console output:

```
Non-secrets automatically mapped in, sauce is crushed tomatoes
Secrets are not automatically mapped in, secretSauce is
You can use macro replacement to get secrets, and they'll be masked in the
log: ***
Future jobs can also see canned goods
Future jobs can also see canned goods
```

SetSecret: Register a value as a secret

`##vso[task.setsecret]value`

Usage

The value is registered as a secret for the duration of the job. The value will be masked out from the logs from this point forward. This command is useful when a secret is transformed (e.g. base64 encoded) or derived.

Note: Previous occurrences of the secret value will not be masked.

Examples

Bash

Set the variables:

YAML

```yaml
- bash: |
    NEWSECRET=$(echo $OLDSECRET|base64)
```

```
    echo "##vso[task.setsecret]$NEWSECRET"
  name: SetSecret
  env:
    OLDSECRET: "SeCrEtVaLuE"
```

Read the variables:

YAML

```
- bash: |
    echo "Transformed and derived secrets will be masked: $(echo
$OLDSECRET|base64)"
  env:
    OLDSECRET: "SeCrEtVaLuE"
```

Console output:

```
Transformed and derived secrets will be masked: ***
```

SetEndpoint: Modify a service connection field

`##vso[task.setendpoint]value`

Usage

Set a service connection field with given value. Value updated will be retained in the endpoint for the subsequent tasks that execute within the same job.

Properties

- `id` = service connection ID (Required)
- `field` = field type, one of `authParameter`, `dataParameter`, or `url` (Required)
- `key` = key (Required, unless `field` = `url`)

Examples

```
##vso[task.setendpoint id=000-0000-
0000;field=authParameter;key=AccessToken]testvalue
```

```
##vso[task.setendpoint id=000-0000-
0000;field=dataParameter;key=userVariable]testvalue
##vso[task.setendpoint id=000-0000-
0000;field=url]https://example.com/service
```

AddAttachment: Attach a file to the build

`##vso[task.addattachment]value`

Usage

Upload and attach attachment to current timeline record. These files aren't available for download with logs. These can only be referred to by extensions using the type or name values.

Properties

- `type` = attachment type (Required)
- `name` = attachment name (Required)

Example

```
##vso[task.addattachment
type=myattachmenttype;name=myattachmentname;]c:\myattachment.txt
```

UploadSummary: Add some Markdown content to the build summary

`##vso[task.uploadsummary]local file path`

Usage

Upload and attach summary Markdown from a .md file in the repository to current timeline record. This summary shall be added to the build/release summary and not available for download with logs. The summary should be in UTF-8 or ASCII format. The summary will appear on the **Extensions** tab of your pipeline run. Markdown rendering on the Extensions tab is different from Azure DevOps wiki rendering.

Examples

```
##vso[task.uploadsummary]$(System.DefaultWorkingDirectory)/testsummary.md
```

It's a short hand form for the command

```
##vso[task.addattachment
type=Distributedtask.Core.Summary;name=testsummaryname;]c:\testsummary.md
```

UploadFile: Upload a file that can be downloaded with task logs

`##vso[task.uploadfile]local file path`

Usage

Upload user interested file as additional log information to the current timeline record. The file shall be available for download along with task logs.

Example

```
##vso[task.uploadfile]c:\additionalfile.log
```

PrependPath: Prepend a path to the PATH environment variable

`##vso[task.prependpath]local file path`

Usage

Update the PATH environment variable by prepending to the PATH. The updated environment variable will be reflected in subsequent tasks.

Example

```
##vso[task.prependpath]c:\my\directory\path
```

Artifact commands

Associate: Initialize an artifact

`##vso[artifact.associate]artifact location`

Usage

Create a link to an existing Artifact. Artifact location must be a file container path, VC path or UNC share path.

Properties

- `artifactname` = artifact name (Required)
- `type` = artifact type (Required) `container` | `filepath` | `versioncontrol` | `gitref` | `tfvclabel`

Examples

- **container**

  ```
  ##vso[artifact.associate
  type=container;artifactname=MyServerDrop]#/1/build
  ```

- **filepath**

  ```
  ##vso[artifact.associate
  type=filepath;artifactname=MyFileShareDrop]\\MyShare\MyDropLocation
  ```

- **versioncontrol**

```
##vso[artifact.associate
type=versioncontrol;artifactname=MyTfvcPath]$/MyTeamProj/MyFolder
```

- **gitref**

```
##vso[artifact.associate
type=gitref;artifactname=MyTag]refs/tags/MyGitTag
```

- **tfvclabel**

```
##vso[artifact.associate type=tfvclabel;artifactname=MyTag]MyTfvcLabel
```

- **Custom Artifact**

```
##vso[artifact.associate
artifactname=myDrop;artifacttype=myartifacttype]https://downloads.visua
lstudio.com/foo/bar/package.zip
```

Upload: Upload an artifact

`##vso[artifact.upload]local file path`

Usage

Upload a local file into a file container folder, and optionally publish an artifact as `artifactname`.

Properties

- `containerfolder` = folder that the file will upload to, folder will be created if needed.
- `artifactname` = artifact name. (Required)

Example

```
##vso[artifact.upload
containerfolder=testresult;artifactname=uploadedresult]c:\testresult.trx
```

> **Note**
>
> The difference between Artifact.associate and Artifact.upload is that the first can be used to create a link to an existing artifact, while the latter can be used to upload/publish a new Artifact.

Build commands

UploadLog: Upload a log

`##vso[build.uploadlog]local file path`

Usage

Upload user interested log to build's container "`logs\tool`" folder.

Example

```
##vso[build.uploadlog]c:\msbuild.log
```

UpdateBuildNumber: Override the automatically generated build number

`##vso[build.updatebuildnumber]build number`

Usage

You can automatically generate a build number from tokens you specify in the pipeline options. However, if you want to use your own logic to set the build number, then you can use this logging command.

Example

```
##vso[build.updatebuildnumber]my-new-build-number
```

AddBuildTag: Add a tag to the build

`##vso[build.addbuildtag]build tag`

Usage

Add a tag for current build. You can expand the tag with a predefined or user-defined variable. For example, here a new tag gets added in a Bash task with the value `last_scanned-$(currentDate)`. You can't use a colon with AddBuildTag.

Example

YAML

```yaml
- task: Bash@3
    inputs:
    targetType: 'inline'
    script: |
        last_scanned="last_scanned-$(currentDate)"
        echo "##vso[build.addbuildtag]$last_scanned"
    displayName: 'Apply last scanned tag'
```

Release commands

UpdateReleaseName: Rename current release

`##vso[release.updatereleasename]release name`

Usage

Update the release name for the running release.

> ⓘ **Note**

Supported in Azure DevOps and Azure DevOps Server beginning in version 2020.

Example

```
##vso[release.updatereleasename]my-new-release-name
```

Artifact policy checks

Article • 02/11/2022

Azure DevOps Services

Artifact policies are enforced before deploying to critical environments such as production. These policies are evaluated against all the deployable artifacts in the given pipeline run and block the deployment if the artifacts don't comply. Adding a check to evaluate Artifact requires the custom policy to be configured. This guide describes how custom policies can be created.

> **Note**
>
> Currently, the supported artifact types are for container images and Kubernetes environments

Prerequisites

Use Rego for defining policy that is easy to read and write.

Familiarize yourself with Rego query language. Basics will do.

To support structured document models like JSON, Rego extends Datalog. Rego queries are assertions on data stored in OPA. These queries can be used to define policies that enumerate instances of data that violate the expected state of the system.

Creating custom policies

Below are the sample policies shared. Based on your requirements, you can build your own set of policies.

Check specific project/pipeline

This policy checks if the images are built by Azure Pipelines and Pipeline-foo. For this to work, the pipeline definition should override the name field to something like: **AzureDevOps_$(BuildDefinitionName)_$(Date:yyyyMMdd)$(Rev:.r)**. See more about naming pipeline runs here.

```
allowedBuilder := "AzureDevOps_pipeline-foo"

checkBuilder[errors] {
    trace("Check if images are built by Azure Pipelines")
    resourceUri := values[index].build.resourceUri
    image := fetchImage(resourceUri)
    builder := values[index].build.build.provenance.builderVersion
    trace(sprintf("%s: builder", [builder]))
    not startswith(builder, "allowedBuilder")
    errors := sprintf("%s: image not built by Azure Pipeline [%s]",
[image,builder])
}

fetchRegistry(uri) = reg {
    out := regex.find_n("//.*/", uri, 1)
    reg = trim(out[0], "/")
}

fetchImage(uri) = img {
    out := regex.find_n("/.*@", uri, 1)
    img := trim(out[0], "/@")
}
```

Check allowed registries

This policy checks if the images are from allowed registries only.

```
allowlist = {
 "gcr.io/myrepo",
 "raireg1.azurecr.io"
}

checkregistries[errors] {
    trace(sprintf("Allowed registries: %s", [concat(", ", allowlist)]))
    resourceUri := values[index].image.resourceUri
    registry := fetchRegistry(resourceUri)
    image := fetchImage(resourceUri)
    not allowlist[registry]
    errors := sprintf("%s: source registry not permitted", [image])
}

fetchRegistry(uri) = reg {
    out := regex.find_n("//.*/", uri, 1)
    reg = trim(out[0], "/")
}

fetchImage(uri) = img {
    out := regex.find_n("/.*@", uri, 1)
```

```
    img := trim(out[0], "/@")
}
```

Check forbidden ports

This policy checks for any forbidden ports exposed in the container image.

```

forbiddenPorts = {
    "80",
    "22"
}

checkExposedPorts[errors] {
    trace(sprintf("Checking for forbidden exposed ports: %s", [concat(", ",
forbiddenPorts)]))
    layerInfos := values[index].image.image.layerInfo
    layerInfos[x].directive == "EXPOSE"
    resourceUri := values[index].image.resourceUri
    image := fetchImage(resourceUri)
    ports := layerInfos[x].arguments
    trace(sprintf("exposed ports: %s", [ports]))
    forbiddenPorts[ports]
    errors := sprintf("%s: image exposes forbidden port %s", [image,ports])
}

fetchRegistry(uri) = reg {
    out := regex.find_n("//.*/", uri, 1)
    reg = trim(out[0], "/")
}

fetchImage(uri) = img {
    out := regex.find_n("/.*@", uri, 1)
    img := trim(out[0], "/@")
}
```

Check prior deployments

This policy checks if the image has been pre-deployed to one/more of the environments before being deployed to specific environment/resources with Check configured.

```

predeployedEnvironments = {
    "env/resource1",
    "env2/resource3"
}
```

```
checkDeployedEnvironments[errors] {
    trace(sprintf("Checking if the image has been pre-deployed to one of:
[%s]", [concat(", ", predeployedEnvironments)]))
    deployments := values[index].deployment
    deployedAddress := deployments[i].deployment.address
    trace(sprintf("deployed to : %s",[deployedAddress]))
    resourceUri := deployments[i].resourceUri
    image := fetchImage(resourceUri)
    not predeployedEnvironments[deployedAddress]
    trace(sprintf("%s: fails pre-deployed environment condition. found %s",
[image,deployedAddress]))
    errors := sprintf("image %s fails pre-deployed environment condition.
found %s", [image,deployedAddress])
}

fetchRegistry(uri) = reg {
    out := regex.find_n("//.*/", uri, 1)
    reg = trim(out[0], "/")
}

fetchImage(uri) = img {
    out := regex.find_n("/.*@", uri, 1)
    img := trim(out[0], "/@")
}
```

az pipelines

Reference

> **Note**
>
> This reference is part of the **azure-devops** extension for the Azure CLI (version 2.30.0 or higher). The extension will automatically install the first time you run an **az pipelines** command. **Learn more** about extensions.

Manage Azure Pipelines.

This command group is a part of the azure-devops extension.

Commands

Name	Description	Type	Status
az pipelines agent	Manage agents.	Extension	GA
az pipelines agent list	Get a list of agents in a pool.	Extension	GA
az pipelines agent show	Show agent details.	Extension	GA
az pipelines build	Manage builds.	Extension	GA
az pipelines build cancel	Cancels if build is running.	Extension	GA
az pipelines build definition	Manage build definitions.	Extension	GA
az pipelines build definition list	List build definitions.	Extension	GA
az pipelines build definition show	Get the details of a build definition.	Extension	GA
az pipelines build list	List build results.	Extension	GA
az pipelines build queue	Request (queue) a build.	Extension	GA
az pipelines build show	Get the details of a build.	Extension	GA
az pipelines build tag	Manage build tags.	Extension	GA
az pipelines build tag add	Add tag(s) for a build.	Extension	GA
az pipelines build tag delete	Delete a build tag.	Extension	GA
az pipelines build tag list	Get tags for a build.	Extension	GA

Name	Description	Type	Status
az pipelines create	Create a new Azure Pipeline (YAML based).	Extension	GA
az pipelines delete	Delete a pipeline.	Extension	GA
az pipelines folder	Manage folders for organizing pipelines.	Extension	GA
az pipelines folder create	Create a folder.	Extension	GA
az pipelines folder delete	Delete a folder.	Extension	GA
az pipelines folder list	List all folders.	Extension	GA
az pipelines folder update	Update a folder name or description.	Extension	GA
az pipelines list	List pipelines.	Extension	GA
az pipelines pool	Manage agent pools.	Extension	GA
az pipelines pool list	List agent pools.	Extension	GA
az pipelines pool show	Show agent pool details.	Extension	GA
az pipelines queue	Manage agent queues.	Extension	GA
az pipelines queue list	List agent queues.	Extension	GA
az pipelines queue show	Show details of agent queue.	Extension	GA
az pipelines release	Manage releases.	Extension	GA
az pipelines release create	Request (create) a release.	Extension	GA
az pipelines release definition	Manage release definitions.	Extension	GA
az pipelines release definition list	List release definitions.	Extension	GA
az pipelines release definition show	Get the details of a release definition.	Extension	GA
az pipelines release list	List release results.	Extension	GA
az pipelines release show	Get the details of a release.	Extension	GA
az pipelines run	Queue (run) a pipeline.	Extension	GA
az pipelines runs	Manage pipeline runs.	Extension	GA
az pipelines runs artifact	Manage pipeline run artifacts.	Extension	GA

Name	Description	Type	Status
az pipelines runs artifact download	Download a pipeline artifact.	Extension	GA
az pipelines runs artifact list	List artifacts associated with a run.	Extension	GA
az pipelines runs artifact upload	Upload a pipeline artifact.	Extension	GA
az pipelines runs list	List the pipeline runs in a project.	Extension	GA
az pipelines runs show	Show details of a pipeline run.	Extension	GA
az pipelines runs tag	Manage pipeline run tags.	Extension	GA
az pipelines runs tag add	Add tag(s) for a pipeline run.	Extension	GA
az pipelines runs tag delete	Delete a pipeline run tag.	Extension	GA
az pipelines runs tag list	Get tags for a pipeline run.	Extension	GA
az pipelines show	Get the details of a pipeline.	Extension	GA
az pipelines update	Update a pipeline.	Extension	GA
az pipelines variable	Manage pipeline variables.	Extension	GA
az pipelines variable-group	Manage variable groups.	Extension	GA
az pipelines variable-group create	Create a variable group.	Extension	GA
az pipelines variable-group delete	Delete a variable group.	Extension	GA
az pipelines variable-group list	List variable groups.	Extension	GA
az pipelines variable-group show	Show variable group details.	Extension	GA
az pipelines variable-group update	Update a variable group.	Extension	GA
az pipelines variable-group variable	Manage variables in a variable group.	Extension	GA
az pipelines variable-group variable create	Add a variable to a variable group.	Extension	GA
az pipelines variable-group variable delete	Delete a variable from variable group.	Extension	GA
az pipelines variable-group variable list	List the variables in a variable group.	Extension	GA
az pipelines variable-group variable update	Update a variable in a variable group.	Extension	GA

Name	Description	Type	Status
az pipelines variable create	Add a variable to a pipeline.	Extension	GA
az pipelines variable delete	Delete a variable from pipeline.	Extension	GA
az pipelines variable list	List the variables in a pipeline.	Extension	GA
az pipelines variable update	Update a variable in a pipeline.	Extension	GA

az pipelines create

Create a new Azure Pipeline (YAML based).

Azure CLI

```
az pipelines create --name
                    [--branch]
                    [--description]
                    [--detect {false, true}]
                    [--folder-path]
                    [--org]
                    [--project]
                    [--queue-id]
                    [--repository]
                    [--repository-type {github, tfsgit}]
                    [--service-connection]
                    [--skip-first-run {false, true}]
                    [--yaml-path]
```

Examples

Create an Azure Pipeline from local checkout repository context

Azure CLI

```
Repository name/url (--repository), type (--repository-type) and branch name
(--branch) will be detected from the local git repository
az pipelines create --name 'ContosoBuild' --description 'Pipeline for
contoso project'
```

Create an Azure Pipeline for a repository hosted on Github using clone url

Azure CLI

```
az pipelines create --name 'ContosoBuild' --description 'Pipeline for
contoso project'
--repository https://github.com/SampleOrg/SampleRepo --branch master
```

Create an Azure Pipeline for a repository hosted on Github organization SampleOrg, repository name SampleRepo

Azure CLI

```
az pipelines create --name 'ContosoBuild' --description 'Pipeline for
contoso project'
--repository SampleOrg/SampleRepoName --branch master --repository-type
github
```

Create an Azure Pipeline for a repository hosted in a Azure Repo in the same project

Azure CLI

```
az pipelines create --name 'ContosoBuild' --description 'Pipeline for
contoso project'
--repository SampleRepoName --branch master --repository-type tfsgit
```

Create an Azure Pipeline for a repository with the pipeline yaml already checked in into the repository

Azure CLI

```
Service connection required for non Azure Repos can be optionally provided
in the command to run it non interatively
az pipelines create --name 'ContosoBuild' --description 'Pipeline for
contoso project'
--repository https://github.com/SampleOrg/SampleRepo --branch master --yml-
path azure-pipelines.yml [--service-connection SERVICE_CONNECTION]
```

Required Parameters

`--name`

Name of the new pipeline.

Optional Parameters

`--branch`

Branch name for which the pipeline will be configured. If omitted, it will be auto-detected from local repository.

`--description`

Description for the new pipeline.

`--detect`

Automatically detect organization.
accepted values: false, true

`--folder-path`

Path of the folder where the pipeline needs to be created. Default is root folder. e.g. "user1/test_pipelines".

`--org --organization`

Azure DevOps organization URL. You can configure the default organization using az devops configure -d organization=ORG_URL. Required if not configured as default or picked up via git config. Example: https://dev.azure.com/MyOrganizationName/.

`--project -p`

Name or ID of the project. You can configure the default project using az devops configure -d project=NAME_OR_ID. Required if not configured as default or picked up via git config.

`--queue-id`

Id of the queue in the available agent pools. Will be auto detected if not specified.

`--repository`

Repository for which the pipeline needs to be configured. Can be clone url of the git repository or name of the repository for a Azure Repos or Owner/RepoName in case of GitHub repository. If omitted it will be auto-detected from the remote url of local git repository. If name is mentioned instead of url, --repository-type argument is also required.

`--repository-type`

Type of repository. If omitted, it will be auto-detected from remote url of local repository. 'tfsgit' for Azure Repos, 'github' for GitHub repository.
accepted values: github, tfsgit

`--service-connection`

Id of the Service connection created for the repository for GitHub repository. Use command az devops service-endpoint -h for creating/listing service_connections. Not required for Azure Repos.

`--skip-first-run --skip-run`

Specify this flag to prevent the first run being triggered by the command. Command will return a pipeline if run is skipped else it will output a pipeline run.
accepted values: false, true

`--yaml-path --yml-path`

Path of the pipelines yaml file in the repo (if yaml is already present in the repo).

▼ Global Parameters

`--debug`

Increase logging verbosity to show all debug logs.

`--help -h`

Show this help message and exit.

`--only-show-errors`

Only show errors, suppressing warnings.

`--output -o`

Output format.
accepted values: json, jsonc, none, table, tsv, yaml, yamlc
default value: json

`--query`

JMESPath query string. See http://jmespath.org/ for more information and examples.

`--subscription`

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

`--verbose`

Increase logging verbosity. Use --debug for full debug logs.

az pipelines delete

Delete a pipeline.

Azure CLI

```
az pipelines delete --id
                    [--detect {false, true}]
                    [--org]
                    [--project]
                    [--yes]
```

Required Parameters

`--id`

ID of the pipeline.

Optional Parameters

`--detect`

Automatically detect organization.

accepted values: false, true

`--org --organization`

Azure DevOps organization URL. You can configure the default organization using az devops configure -d organization=ORG_URL. Required if not configured as default or picked up via git config. Example: https://dev.azure.com/MyOrganizationName/.

`--project -p`

Name or ID of the project. You can configure the default project using az devops configure -d project=NAME_OR_ID. Required if not configured as default or picked up via git config.

`--yes -y`

Do not prompt for confirmation.
default value: False

▼ Global Parameters

`--debug`

Increase logging verbosity to show all debug logs.

`--help -h`

Show this help message and exit.

`--only-show-errors`

Only show errors, suppressing warnings.

`--output -o`

Output format.
accepted values: json, jsonc, none, table, tsv, yaml, yamlc
default value: json

`--query`

JMESPath query string. See http://jmespath.org/ for more information and examples.

`--subscription`

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

`--verbose`

Increase logging verbosity. Use --debug for full debug logs.

az pipelines list

List pipelines.

Azure CLI

```
az pipelines list [--detect {false, true}]
                  [--folder-path]
                  [--name]
                  [--org]
                  [--project]
                  [--query-order {ModifiedAsc, ModifiedDesc, NameAsc,
NameDesc, None}]
                  [--repository]
                  [--repository-type {bitbucket, git, github,
githubenterprise, svn, tfsgit, tfsversioncontrol}]
                  [--top]
```

Optional Parameters

`--detect`

Automatically detect organization.
accepted values: false, true

`--folder-path`

If specified, filters to definitions under this folder.

`--name`

Limit results to pipelines with this name or starting with this name. Examples: "FabCI" or "Fab*".

`--org --organization`

Azure DevOps organization URL. You can configure the default organization using az devops configure -d organization=ORG_URL. Required if not configured as default or picked up via git config. Example: https://dev.azure.com/MyOrganizationName/ .

`--project -p`

Name or ID of the project. You can configure the default project using az devops configure -d project=NAME_OR_ID. Required if not configured as default or picked up via git config.

`--query-order`

Order of the results.
accepted values: ModifiedAsc, ModifiedDesc, NameAsc, NameDesc, None

`--repository`

Limit results to pipelines associated with this repository.

`--repository-type`

Limit results to pipelines associated with this repository type. It is mandatory to pass 'repository' argument along with this argument.
accepted values: bitbucket, git, github, githubenterprise, svn, tfsgit, tfsversioncontrol

`--top`

Maximum number of pipelines to list.

▼ Global Parameters

`--debug`

Increase logging verbosity to show all debug logs.

`--help -h`

Show this help message and exit.

`--only-show-errors`

Only show errors, suppressing warnings.

`--output -o`

Output format.

accepted values: json, jsonc, none, table, tsv, yaml, yamlc

default value: json

`--query`

JMESPath query string. See http://jmespath.org/ for more information and examples.

`--subscription`

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

`--verbose`

Increase logging verbosity. Use --debug for full debug logs.

az pipelines run

Queue (run) a pipeline.

Azure CLI

```
az pipelines run [--branch]
                 [--commit-id]
                 [--detect {false, true}]
                 [--folder-path]
                 [--id]
                 [--name]
                 [--open]
                 [--org]
                 [--parameters]
                 [--project]
                 [--variables]
```

Optional Parameters

`--branch`

Name of the branch on which the pipeline run is to be queued. Example: refs/heads/master or master or refs/pull/1/merge or refs/tags/tag.

`--commit-id`

Commit-id on which the pipeline run is to be queued.

`--detect`

Automatically detect organization.
accepted values: false, true

`--folder-path`

Folder path of pipeline. Default is root level folder.

`--id`

ID of the pipeline to queue. Required if --name is not supplied.

`--name`

Name of the pipeline to queue. Ignored if --id is supplied.

`--open`

Open the pipeline results page in your web browser.
default value: False

`--org --organization`

Azure DevOps organization URL. You can configure the default organization using az devops configure -d organization=ORG_URL. Required if not configured as default or picked up via git config. Example: https://dev.azure.com/MyOrganizationName/.

`--parameters`

Space separated "name=value" pairs for the parameters you would like to set.

`--project -p`

Name or ID of the project. You can configure the default project using az devops configure -d project=NAME_OR_ID. Required if not configured as default or picked up via git config.

`--variables`

Space separated "name=value" pairs for the variables you would like to set.

▼ Global Parameters

`--debug`

Increase logging verbosity to show all debug logs.

`--help -h`

Show this help message and exit.

`--only-show-errors`

Only show errors, suppressing warnings.

`--output -o`

Output format.
accepted values: json, jsonc, none, table, tsv, yaml, yamlc
default value: json

`--query`

JMESPath query string. See http://jmespath.org/ for more information and examples.

`--subscription`

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

`--verbose`

Increase logging verbosity. Use --debug for full debug logs.

az pipelines show

Get the details of a pipeline.

Azure CLI

```
az pipelines show [--detect {false, true}]
                  [--folder-path]
                  [--id]
                  [--name]
                  [--open]
                  [--org]
                  [--project]
```

Optional Parameters

`--detect`

Automatically detect organization.

accepted values: false, true

`--folder-path`

Folder path of pipeline. Default is root level folder.

`--id`

ID of the pipeline.

`--name`

Name of the pipeline. Ignored if --id is supplied.

`--open`

Open the pipeline summary page in your web browser.

default value: False

`--org --organization`

Azure DevOps organization URL. You can configure the default organization using az devops configure -d organization=ORG_URL. Required if not configured as default or picked up via git config. Example: https://dev.azure.com/MyOrganizationName/.

`--project -p`

Name or ID of the project. You can configure the default project using az devops configure -d project=NAME_OR_ID. Required if not configured as default or picked up via git config.

▼ Global Parameters

`--debug`

Increase logging verbosity to show all debug logs.

`--help -h`

Show this help message and exit.

`--only-show-errors`

Only show errors, suppressing warnings.

`--output -o`

Output format.
accepted values: json, jsonc, none, table, tsv, yaml, yamlc
default value: json

`--query`

JMESPath query string. See http://jmespath.org/ for more information and examples.

`--subscription`

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

`--verbose`

Increase logging verbosity. Use --debug for full debug logs.

az pipelines update

Update a pipeline.

Azure CLI

```
az pipelines update --id
                    [--branch]
                    [--description]
                    [--detect {false, true}]
                    [--new-folder-path]
                    [--new-name]
                    [--org]
                    [--project]
                    [--queue-id]
                    [--yaml-path]
```

Required Parameters

`--id`

Id of the pipeline to update.

Optional Parameters

`--branch`

Branch name for which the pipeline will be configured.

`--description`

New description for the pipeline.

`--detect`

Automatically detect organization.
accepted values: false, true

`--new-folder-path`

New full path of the folder to move the pipeline to. e.g. "user1/production_pipelines".

`--new-name`

New updated name of the pipeline.

`--org --organization`

Azure DevOps organization URL. You can configure the default organization using az devops configure -d organization=ORG_URL. Required if not configured as default or picked up via git config. Example: https://dev.azure.com/MyOrganizationName/.

`--project -p`

Name or ID of the project. You can configure the default project using az devops configure -d project=NAME_OR_ID. Required if not configured as default or picked up via git config.

`--queue-id`

Queue id of the agent pool where the pipeline needs to run.

`--yaml-path --yml-path`

Path of the pipelines yaml file in the repo.

▼ Global Parameters

`--debug`

Increase logging verbosity to show all debug logs.

`--help -h`

Show this help message and exit.

`--only-show-errors`

Only show errors, suppressing warnings.

`--output -o`

Output format.

accepted values: json, jsonc, none, table, tsv, yaml, yamlc

default value: json

`--query`

JMESPath query string. See http://jmespath.org/ for more information and examples.

`--subscription`

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

`--verbose`

Increase logging verbosity. Use --debug for full debug logs.

Use deployment rings with extension releases

Article • 02/21/2023

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019 | TFS 2018

With deployment rings, you can gradually deploy and validate changes to your extension in production, while limiting the affect on your users.

We don't recommend deploying to all production environments at the same time, which exposes all users to the changes. A gradual rollout exposes users to the changes over time, validating the changes in production with fewer users.

The following table shows the differences for affected areas when you're using rings vs. no rings.

Without rings	Affected area	With rings
Manual and error prone	Build	Automated and consistent
Manual and error prone	Release	Automated and consistent
Hours	Time to build (TTB)	Seconds
Days	Time to release (TTR)	Minutes
Call from user	Issue detection	Proactive
Days to weeks	Issue resolution	Minutes to days

For more information, see Configuring your release pipelines for safe deployments .

Prerequisites

- Review CI/CD Pipelines and Approvals for detailed documentation of pipelines and the approval features for releases.

Assign user types

Determine which users are best suited for each user type. Communicate the opportunity to provide feedback and the risk levels at each tier, as it's critical to setting expectations

and ensuring success. In the following example, users get divided into three groups in production:

- **Canaries**: voluntarily test features as soon as they're available.
- **Early adopters**: voluntarily preview releases, considered more refined than the canary bits.
- **Users**: consume the products, after they've passed through canaries and early adopters.

Map the topology

Map the topology of your extension to the ringed deployment model to limit the impact of change on your users and deliver value. For our open-source community extensions, we mostly use ring-based deployment to progressively expose a new release to canary, early adopters, and users.

At the application level, the composition of Azure DevOps extensions is easy to digest, scale, and deploy independently.

Each extension does the following tasks:

- Has one of more web and script files
- Interfaces with Core client
- Interfaces with REST client and REST APIs
- Persists state in cache or resilient storage

At the infrastructure level, the extensions get published to the Marketplace. Once you install the extension in your organization, it gets hosted by the Azure DevOps service portal, with state persisted to Azure storage or the extension data storage.

The extension topology is perfectly suited for the ring deployment model and to publish the extension to each deployment ring:

- A **private** development version for the canary ring
- A **private** preview version for the early adopter ring
- A **public** production version for the users ring

> **Tip**
>
> Publish your extension as *private*, to control exposure to invited users.

Move changes through deployment rings

See the following example flow of moving changes through deployment rings.

Use the Azure DevOps Developer Tools Build Tasks extension to package and publish extensions to the Marketplace.

1. A developer from the Countdown Widget extension project commits a change to the GitHub repository.
2. The commit triggers a continuous integration build.

3. The new build triggers a continuous deployment trigger, which automatically starts the **Canaries** environment deployment.
4. The **Canaries** deployment publishes a private extension to the Marketplace and shares it with predefined organizations. Only the **Canaries** are affected by the change.
5. The **Canaries** deployment triggers the **Early Adopter** environment deployment. A pre-deployment approval gate requires any one of the authorized users to approve the release.

6. The **Early Adopter** deployment publishes a private extension to the marketplace and shares it with predefined organizations. Both the **Canaries** and **Early Adopter** are affected by the change.
7. The **Early Adopter** deployment triggers the **Users** environment deployment. A stricter pre-deployment approval gate requires all of the authorized users to approve the release.

8. The **Users** deployment publishes a public extension to the marketplace. At this stage, everyone who installed the extension in their organization is affected by the change.
9. It's key to realize that the effect increases as your change moves through the rings. Exposing the change to the **Canaries** and the **Early Adopters** gives you two opportunities to validate the change and hotfix critical bugs before you release to production.

Monitor issues

Monitoring and alerts can help you detect and mitigate issues. Determine what type of data is important, for example: infrastructure issues, violations, and feature usage. Focus on actionable alerts to avoid users ignoring them and missing high priority issues.

> **Tip**
>
> Start with high-level views of your data, visual dashboards that you can watch from afar and drill-down, as needed. Perform regular housekeeping of your views and remove all noise. A visual dashboard tells a better story than lots of notification emails, often filtered and forgotten by email rules.

Use Team Project Health and other extensions to build an overview of your pipelines, lead and cycle times, and gather other information. In the sample dashboard, it's evident that there are 34 successful builds, 21 successful releases, 1 failed release, and 2 releases in progress.

Is there a dependency on feature flags?

No. Sometimes you might need a certain functionality to be deployed as part of a release, but not initially exposed to users. Feature flags give you fine-grained control of features that are included in your change. For example, if you're not fully confident about a feature you can use feature flags to **hide** the feature in one or all of the deployment rings. You can enable all features in the canaries ring, and fine-tune a subset for the early adopters and production users, as shown in the following image.

For more information, see Progressive experimentation with feature flags.

FAQ

Q: How do you know that a change can be deployed to the next ring?

A: You should have a consistent checklist for the users approving a release.

Q: How long do you wait before you push a change to the next ring?

There's no fixed duration or "cool off" period. It depends on how long it takes for you to complete all release validations successfully.

Q: How do you manage a hotfix?

A: The ring deployment model allows you to process a hotfix like any other change. The sooner you catch an issue, the sooner you can deploy a hotfix with no effect to downstream rings.

Q: How do you deal with variables that span shared release environments?

A: Refer to Default and custom release variables.

Q: How can you manage secrets used by the pipeline?

A: To safeguard cryptographic keys and other secrets used by your pipelines, see Azure Key Vault.

Related articles

- Safe deployment practices
- Progressive experimentation with feature flags
- Configure your release pipelines for safe deployments.

Progressive experimentation with feature flags

Article • 11/28/2022

As DevOps teams shift to an Agile methodology that focuses on continuous delivery of features, the need to control how they become available to users becomes increasingly important. Feature flags are a great solution for limiting user access to new features, either for marketing purposes or for testing in production.

Decoupling deployment and exposure

With *feature flags*, a team can choose whether a given set of features is visible in the user experience and/or invoked within the functionality. New features can be built and deployed as part of the ordinary development process without having those features available for broad access. The deployment of features is conveniently decoupled from their exposure.

Flags provide runtime control down to individual user

Flags also provide granular control all the way down to the individual user. When it's time to enable a feature, whether for one user, a small group, or everyone, the team can simply change the feature flag in order to light it up without having to redeploy.

The scope of a feature flag will vary based on the nature of the feature and the audience. In some cases, a feature flag will automatically enable the functionality for everyone. In other cases, a feature will be enabled on a user-by-user basis. Teams can also use feature flags to allow users to opt in to enable a feature, if they so desire. There's really no limit to the way the feature flags are implemented.

Support early feedback and experimentation

Feature flags are a great way to support early experimentation. Some features can have rough edges early on, which may be interesting only to the earliest of adopters. Trying to push those not-quite-ready features onto a broader audience could produce dissatisfaction. But the benefit of gathering feedback from users willing to deal with an in-progress feature is invaluable.

Quick off switch

Sometimes it's helpful to be able to turn something off. For example, suppose a new feature isn't working the way it was intended, and there are side effects that cause problems elsewhere. You can use feature flags to quickly turn off the new functionality in order to roll back to trusted behavior without having to redeploy. While feature flags are often thought of in terms of user interface features, they can also easily be used for changes in architecture or infrastructure.

Standard stages

Microsoft uses a standard rollout process to turn on feature flags. There are two separate concepts: *rings* are for deployments, and *stages* are for feature flags. Learn more about rings and stages.

Stages are all about disclosure or exposure. For example, the first stage could be for a team's account and the personal accounts of members. Most users wouldn't see anything new because the only place flags are turned on is for this first stage. This allows a team to fully use and experiment with it. Once the team signs off, select customers would be able to opt into it via the second stage of feature flags.

Opt in

It's a good practice to allow users to opt in to feature flags when feasible. For example, the team may expose a preview panel associated with the user's preferences or settings.

Use flags with telemetry

Feature flags provide a way to incrementally expose updates. However, teams must continuously monitor the right metrics to gauge readiness for broader exposure. These metrics should include usage behavior, as well as the impact of the updates on the health of the system. It's important to avoid the trap of assuming everything is okay just because nothing bad seems to be happening.

A feature flag example

Consider the example below. The team added a couple buttons here for **Cherry-pick** and **Revert** in the pull request UI. These were deployed using feature flags.

Define feature flags

The first feature exposed was the **Revert** button. The solution uses an XML file to define all of the feature flags. There's one file per service in this case, which creates an incentive to remove old flags to prevent the section from getting really long. The team will delete old flags because there's a natural motivation to control the size of that file.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  In this group we should register Azure DevOps specific features and sets
their states.
-->
<ServicingStepGroup name="AzureDevOpsFeatureAvailability" … >
  <Steps>
    <!-- Feature Availability -->
    <ServicingStep name="Register features"
stepPerformer="FeatureAvailability" … >
      <StepData>
        <!--specifying owner to allow implicit removal of features -->
        <Features owner="AzureDevOps">
           <!-- Begin TFVC/Git -->
```

```
            <Feature name="SourceControl.Revert" description="Source control
revert features" />
```

A common server framework encourages reuse and economies of scale across the whole team. Ideally, the project will have infrastructure in place so that a developer can simply define a flag in a central store and have the rest of the infrastructure handled for them.

Check feature flags at runtime

The feature flag used here is named **SourceControl.Revert**. Here's the actual TypeScript from that page that illustrates the call for a feature availability check.

TypeScript

```
private addRevertButton(): void {
 if (FeatureAvailability.isFeatureEnabled(Flags.SourceControlRevert)) {
     this._calloutButtons.unshift(
         <button onClick={ () => Dialogs.revertPullRequest(
             this.props.repositoryContext,
             this.props.pullRequest.pullRequestContract(),

this.props.pullRequest.branchStatusContract().sourceBranchStatus,

this.props.pullRequest.branchStatusContract().targetBranchStatus)
         }
         >
             {VCResources.PullRequest_Revert_Button}
         </button>
        );
     }
}
```

The example above illustrates usage in TypeScript, but it could just as easily be accessed using C#. The code checks to see if the feature is enabled and, if so, renders a button to provide the functionality. If the flag isn't enabled, then the button is skipped.

Control a feature flag

A good feature flag platform will provide multiple ways to manage whether a given flag is set. Typically, there are usage scenarios for the flag to be controlled via PowerShell and web interface. For PowerShell, all that really needs to be exposed are ways to get and set a feature flag's status, along with optional parameters for things like specific user account identifiers, if applicable.

Control feature flags through web UI

The following example uses the web UI exposed for this product by the team. Note the feature flag for **SourceControl.Revert**. There are two personal accounts listed here: **hallux** and **buckh-westeur**. The state is set for **hallux**, which happens to be in North Central, and cleared for the other account in West Europe.

The nature of the feature flag will drive the way in which the features are exposed. In some cases, the exposure will follow a ring and stage model. In others, users may opt in through configuration UI, or even by emailing the team for access.

Considerations for feature flags

Most feature flags can be retired once a feature has been rolled out to everyone. At that point, the team can delete all references to the flag in code and configuration. It's a good practice to include a feature flag review, such as at the beginning of each sprint.

At the same time, there may be a set of feature flags that persist for various reasons. For example, the team may want to keep a feature flag that branches something infrastructural for a period of time after the production service has fully switched over. However, keep in mind that this potential codepath could be reactivated in the future

during an explicit clearing of the feature flag, so it needs to be tested and maintained until the option is removed.

Feature flags and branching strategy

Feature flags enable development teams to include incomplete features in `main` without affecting anybody else. As long as the codepath is isolated behind a feature flag, it's generally safe to build and publish that code without side effects impacting normal usage. But if there are cases where a feature requires dependencies, such as when exposing a REST endpoint, teams must consider how those dependencies can create security or maintenance work even without the feature being exposed.

Feature flags to mitigate risk

Sometimes new features have the potential to introduce destructive or disruptive changes. For example, the product may be undergoing a transformation from a wide database schema to a long one. In that scenario, the developer should create a feature branch for a small amount of time. They then make the destabilizing changes on the branch and keep the feature behind a flag. A popular practice is for teams to then merge changes up to `main` as soon as they're not causing any harm. This wouldn't be feasible without the ability to keep the unfinished feature hidden behind a feature flag.

Feature flags help work in main

If you follow the common-sense practices discussed in the Develop phase, working in `main` is a good way to tighten a DevOps cycle. When combined with feature flags, developers can quickly merge features upstream and push them through the test gauntlet. Quality code can quickly get published for testing in production. After a few sprints, developers will recognize the benefits of feature flags and use them proactively.

How to decide whether to use a feature flag

The feature teams own the decision as to whether they need a feature flag or not for a given change. Not every change requires one, so it's a judgment call for a developer when they choose make a given change. In the case of the **Revert** feature discussed earlier, it was important to use a feature flag to control exposure. Allowing teams to own key decisions about their feature area is part of enabling autonomy in an effective DevOps organization.

Build vs. buy

While it's possible to build your own feature flag infrastructure, adopting a platform like LaunchDarkly is generally recommended. It's preferable to invest in building features instead of rebuilding feature flag functionality.

Next steps

Learn more about using feature flags in an ASP.NET Core app.

Azure DevOps Services REST API Reference

Article • 03/31/2023

Welcome to the Azure DevOps Services/Azure DevOps Server REST API Reference.

Representational State Transfer (REST) APIs are service endpoints that support sets of HTTP operations (methods), which provide create, retrieve, update, or delete access to the service's resources. This article walks you through:

- The basic components of a REST API request/response pair.
- Overviews of creating and sending a REST request, and handling the response.

> Most REST APIs are accessible through our client libraries, which can be used to greatly simplify your client code.

Components of a REST API request/response pair

A REST API request/response pair can be separated into five components:

1. The **request URI**, in the following form: `VERB https://{instance}[/{team-project}]/_apis[/{area}]/{resource}?api-version={version}`

    - *instance*: The Azure DevOps Services organization or TFS server you're sending the request to. They are structured as follows:
        - Azure DevOps Services: `dev.azure.com/{organization}`
        - TFS: `{server:port}/tfs/{collection}` (the default port is 8080, and the value for collection should be `DefaultCollection` but can be any collection)
    - *resource path*: The resource path is as follows: `_apis/{area}/{resource}`. For example `_apis/wit/workitems`.
    - *api-version*: Every API request should include an api-version to avoid having your app or service break as APIs evolve. api-versions are in the following format: `{major}.{minor}[-{stage}[.{resource-version}]]`, for example:
        - `api-version=1.0`
        - `api-version=1.2-preview`
        - `api-version=2.0-preview.1`

> Note: *area* and *team-project* are optional, depending on the API request. Check out the TFS to REST API version mapping matrix below to find which REST API versions apply to your version of TFS.

2. HTTP **request message header** fields:

   - A required HTTP method (also known as an operation or verb), which tells the service what type of operation you are requesting. Azure REST APIs support GET, HEAD, PUT, POST, and PATCH methods.
   - Optional additional header fields, as required by the specified URI and HTTP method. For example, an Authorization header that provides a bearer token containing client authorization information for the request.

3. Optional HTTP **request message body** fields, to support the URI and HTTP operation. For example, POST operations contain MIME-encoded objects that are passed as complex parameters.

   - For POST or PUT operations, the MIME-encoding type for the body should be specified in the Content-type request header as well. Some services require you to use a specific MIME type, such as `application/json`.

4. HTTP **response message header** fields:

   - An HTTP status code, ranging from 2xx success codes to 4xx or 5xx error codes. Alternatively, a service-defined status code may be returned, as indicated in the API documentation.
   - Optional additional header fields, as required to support the request's response, such as a `Content-type` response header.

5. Optional HTTP **response message body** fields:

   - MIME-encoded response objects may be returned in the HTTP response body, such as a response from a GET method that is returning data. Typically, these objects are returned in a structured format such as JSON or XML, as indicated by the `Content-type` response header. For example, when you request an access token from Azure AD, it will be returned in the response body as the `access_token` element, one of several name/value paired objects in a data collection. In this example, a response header of `Content-Type: application/json` is also included.

Create the request

Authenticate

There are many ways to authenticate your application or service with Azure DevOps Services or TFS. The following table is an excellent way to decide which method is the best for you:

Type of application	Description	example	Authentication mechanism	Code samples
Interactive client-side	Client application, that allows user interaction, calling Azure DevOps Services REST APIs	Console application enumerating projects in an organization	Microsoft Authentication Library (MSAL)	sample
Interactive JavaScript	GUI based JavaScript application	AngularJS single page app displaying project information for a user	MSAL	sample
Non-interactive client-side	Headless text only client side application	Console app displaying all bugs assigned to a user	Device Profile	sample
Interactive web	GUI based web application	Custom Web dashboard displaying build summaries	OAuth	sample
TFS application	TFS app using the Client OM library	TFS extension displaying team bug dashboards	Client Libraries	sample
Azure DevOps Services Extension	Azure DevOps Services extension	Azure DevOps extension samples	VSS Web Extension SDK	sample walkthrough

> **Note:** You can find more information on authentication on our authentication guidance page.

Assemble the request

Azure DevOps Services

For Azure DevOps Services, *instance* is `dev.azure.com/{organization}`, so the pattern looks like this:

```
VERB https://dev.azure.com/{organization}/_apis[/{area}]/{resource}?api-version={version}
```

For example, here's how to get a list of team projects in a Azure DevOps Services organization.

```dos
curl -u {username}[:{personalaccesstoken}] https://dev.azure.com/{organization}/_apis/projects?api-version=2.0
```

If you wish to provide the personal access token through an HTTP header, you must first convert it to a Base64 string (the following example shows how to convert to Base64 using C#). (Certain tools like Postman applies a Base64 encoding by default. If you are trying the API via such tools, Base64 encoding of the PAT is not required) The resulting string can then be provided as an HTTP header in the format:

```
Authorization: Basic BASE64PATSTRING
```

Here it is in C# using the [HttpClient class](/previous-versions/visualstudio/hh193681(v=vs.118).

```C#
public static async void GetProjects()
{
    try
    {
        var personalaccesstoken = "PAT_FROM_WEBSITE";

        using (HttpClient client = new HttpClient())
        {
            client.DefaultRequestHeaders.Accept.Add(
                new
System.Net.Http.Headers.MediaTypeWithQualityHeaderValue("application/json"))
;

            client.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Basic",
                Convert.ToBase64String(
                    System.Text.ASCIIEncoding.ASCII.GetBytes(
                        string.Format("{0}:{1}", "",
personalaccesstoken))));
```

```
            using (HttpResponseMessage response = await client.GetAsync(

"https://dev.azure.com/{organization}/_apis/projects"))
            {
                response.EnsureSuccessStatusCode();
                string responseBody = await
response.Content.ReadAsStringAsync();
                Console.WriteLine(responseBody);
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
}
```

Most samples on this site use Personal Access Tokens as they're a compact example for authenticating with the service. However, there are a variety of authentication mechanisms available for Azure DevOps Services including MSAL, OAuth and Session Tokens. Refer to the Authentication section for guidance on which one is best suited for your scenario.

TFS

For TFS, `instance` is `{server:port}/tfs/{collection}` and by default the port is 8080. The default collection is `DefaultCollection`, but can be any collection.

Here's how to get a list of team projects from TFS using the default port and collection.

dos

```
curl -u {username}[:{personalaccesstoken}]
https://{server}:8080/tfs/DefaultCollection/_apis/projects?api-version=2.0
```

The examples above use personal access tokens, which requires that you create a personal access token.

Process the response

You should get a response like this.

JSON

```
{
    "value": [
```

```
        {
            "id": "eb6e4656-77fc-42a1-9181-4c6d8e9da5d1",
            "name": "Fabrikam-Fiber-TFVC",
            "url": "https://dev.azure.com/fabrikam-fiber-
inc/_apis/projects/eb6e4656-77fc-42a1-9181-4c6d8e9da5d1",
            "description": "TeamFoundationVersionControlprojects",
            "collection": {
                "id": "d81542e4-cdfa-4333-b082-1ae2d6c3ad16",
                "name": "DefaultCollection",
                "url": "https: //dev.azure.com/fabrikam-fiber-
inc/_apis/projectCollections/d81542e4-cdfa-4333-b082-1ae2d6c3ad16",
                "collectionUrl": "https: //dev.azure.com/fabrikam-fiber-
inc/DefaultCollection"
            },
            "defaultTeam": {
                "id": "66df9be7-3586-467b-9c5f-425b29afedfd",
                "name": "Fabrikam-Fiber-TFVCTeam",
                "url": "https://dev.azure.com/fabrikam-fiber-
inc/_apis/projects/eb6e4656-77fc-42a1-9181-4c6d8e9da5d1/teams/66df9be7-3586-
467b-9c5f-425b29afedfd"
            }
        },
        {
            "id": "6ce954b1-ce1f-45d1-b94d-e6bf2464ba2c",
            "name": "Fabrikam-Fiber-Git",
            "url": "https://dev.azure.com/fabrikam-fiber-
inc/_apis/projects/6ce954b1-ce1f-45d1-b94d-e6bf2464ba2c",
            "description": "Gitprojects",
            "collection": {
                "id": "d81542e4-cdfa-4333-b082-1ae2d6c3ad16",
                "name": "DefaultCollection",
                "url": "https://dev.azure.com/fabrikam-fiber-
inc/_apis/projectCollections/d81542e4-cdfa-4333-b082-1ae2d6c3ad16",
                "collectionUrl": "https://dev.azure.com/fabrikam-fiber-
inc/DefaultCollection"
            },
            "defaultTeam": {
                "id": "8bd35c5e-30bb-4834-a0c4-d576ce1b8df7",
                "name": "Fabrikam-Fiber-GitTeam",
                "url": "https://dev.azure.com/fabrikam-fiber-
inc/_apis/projects/6ce954b1-ce1f-45d1-b94d-e6bf2464ba2c/teams/8bd35c5e-30bb-
4834-a0c4-d576ce1b8df7"
            }
        }
    ],
    "count": 2
}
```


The response is JSON. That's generally what you'll get back from the REST APIs although there are a few exceptions, like Git blobs.

Now you should be able to look around the specific API areas like work item tracking or Git and get to the resources that you need. Keep reading to learn more about the

general patterns that are used in these APIs.

API and TFS version mapping

Below you'll find a quick mapping of REST API versions and their corresponding TFS releases. All API versions will work on the server version mentioned as well as later versions.

TFS Version	REST API Version	Build Version
Azure DevOps Server vNext	7.1	
Azure DevOps Server 2022	7.0	versions >= `19.205.33122.1`
Azure DevOps Server 2020	6.0	versions >= `18.170.30525.1`
Azure DevOps Server 2019	5.0	versions >= `17.143.28621.4`
TFS 2018 Update 3	4.1	versions >= `16.131.28106.2`
TFS 2018 Update 2	4.1	versions >= `16.131.27701.1`
TFS 2018 Update 1	4.0	versions >= `16.122.27409.2`
TFS 2018 RTW	4.0	versions >= `16.122.27102.1`
TFS 2017 Update 2	3.2	versions >= `15.117.26714.0`
TFS 2017 Update 1	3.1	versions >= `15.112.26301.0`
TFS 2017 RTW	3.0	versions >= `15.105.25910.0`
TFS 2015 Update 4	2.3	versions >= `14.114.26403.0`
TFS 2015 Update 3	2.3	versions >= `14.102.25423.0`
TFS 2015 Update 2	2.2	versions >= `14.95.25122.0`
TFS 2015 Update 1	2.1	versions >= `14.0.24712.0`
TFS 2015 RTW	2.0	versions >= `14.0.23128.0`

Related Content

Check out the Integrate documentation for REST API samples and use cases.

- Authentication guidance
- Samples

Client Libraries

Discover the client libraries for these REST APIs.

- .NET conceptual documentation and .NET reference documentation
- Go
- Node.js
- Python
- Swagger 2.0
- Web Extensions SDK

Where are the earlier versions of REST APIs? (Before 4.1)

We recently made a change to our engineering system and documentation generation process; we made this change to provide clearer, more in-depth, and more accurate documentation for everyone trying to use these REST APIs. Due to technical constraints, we are only able to document **API Version 4.1 and newer** using this method. We believe the documentation for API Version 4.1 and newer will be easier to use due to this change.

If you are working in TFS or are looking for the older versions of REST APIs, you can take a look at the REST API Overview for TFS 2015, 2017, and 2018.

Get started with Azure DevOps CLI

Article • 03/27/2023

Azure DevOps Services

With the Azure DevOps extension for Azure Command Line Interface (CLI), you can manage many Azure DevOps Services from the command line. CLI commands enable you to streamline your tasks with faster and flexible interactive canvas, bypassing user interface workflows.

> **Note**
>
> The Azure DevOps Command Line Interface (CLI) is only available for use with Azure DevOps Services. The Azure DevOps extension for the Azure CLI does not support any version of Azure DevOps Server.

To start using the Azure DevOps extension for Azure CLI, perform the following steps:

1. Install Azure CLI: Follow the instructions provided in Install the Azure CLI to set up your Azure CLI environment. At a minimum, your Azure CLI version must be 2.10.1. You can use `az --version` to validate.

2. Add the Azure DevOps extension:

   ```
   az extension add --name azure-devops
   ```

   You can use `az extension list` or `az extension show --name azure-devops` to confirm the installation.

3. Sign in: Run `az login` to sign in. Note that we support only interactive or log in using user name and password with `az login`. To sign in using a Personal Access Token (PAT), see Sign in via Azure DevOps Personal Access Token (PAT).

4. Configure defaults: We recommend you set the default configuration for your organization and project. Otherwise, you can set these within the individual commands themselves.

```
az devops configure --defaults
organization=https://dev.azure.com/contoso project=ContosoWebApp
```

Command usage

Adding the Azure DevOps Extension adds `devops`, `pipelines`, `artifacts`, `boards`, and `repos` groups. For usage and help content for any command, enter the **-h** parameter, for example:

Azure CLI

```
az devops -h
```

Output

```
Group
    az devops : Manage Azure DevOps organization level operations.
        Related Groups
        az pipelines: Manage Azure Pipelines
        az boards: Manage Azure Boards
        az repos: Manage Azure Repos
        az artifacts: Manage Azure Artifacts.

Subgroups:
    admin            : Manage administration operations.
    extension        : Manage extensions.
    project          : Manage team projects.
    security         : Manage security related operations.
    service-endpoint : Manage service endpoints/service connections.
    team             : Manage teams.
    user             : Manage users.
    wiki             : Manage wikis.

Commands:
    configure        : Configure the Azure DevOps CLI or view your
configuration.
    feedback         : Displays information on how to provide feedback to
the Azure DevOps CLI team.
    invoke           : This command will invoke request for any DevOps area
and resource. Please use
                       only json output as the response of this command is
not fixed. Helpful docs -
                       https://learn.microsoft.com/rest/api/azure/devops/.
    login            : Set the credential (PAT) to use for a particular
organization.
    logout           : Clear the credential for all or a particular
organization.
```

Open items in browser

You can use `--open` switch to open any artifact in Azure DevOps portal in your default browser.

For example :

Azure CLI

```
az pipelines build show --id 1 --open
```

This command shows the details of build with `id 1` on the command-line and also opens it in the default browser.

Related articles

- Sign in via Azure DevOps Personal Access Token (PAT)
- Output formats
- Index to az devops examples
- Azure DevOps CLI Extension GitHub Repo

Azure Lab Services documentation

Learn how to use Azure Lab Services to quickly set up a development, test, hackathon, or a lab for your team or students in the cloud.

About Lab Services

OVERVIEW

What is Azure Lab Services?

What's new in Lab Services?

Lab Services concepts

Lab plans

TUTORIAL

Set up resources for creating labs

HOW-TO GUIDE

Add a lab Creator

Configure auto-shutdown

Connect to a virtual network

Create & manage labs

QUICKSTART

Create and connect to a lab

TUTORIAL

Set up a lab for classroom training

Create a lab in Teams or Canvas

HOW-TO GUIDE

HOW-TO GUIDE

Manage a lab

Add and manage lab users

Create and manage schedules

Manage the virtual machine pool

Customize & publish templates

Access labs

HOW-TO GUIDE

Access a lab

Connect to a lab VM

Set or reset passwords

Explore reference content

REFERENCE

REST

Use a lab environment for your devops

Article • 09/01/2022

Applies to: Visual Studio Visual Studio for Mac Visual Studio Code

A lab environment is a collection of virtual and physical machines that you can use to develop and test applications. A lab environment can contain multiple roles needed to test multi-tiered applications, such as workstations, web servers, and database servers. In addition, you can use a build-deploy-test workflow with your lab environment to automate the process of building, deploying, and running automated tests on your application.

> **Important**
>
> Visual Studio no longer supports **Microsoft Test Manager and Lab Management** for automated testing. Similarly, **SCVMM** and XAML Builds are no longer supported. We recommend **Azure DevTest Labs** instead.