

AZURE DEVOPS NOTES

BY NEXTOPS



Azure DevOps is a set of development tools and services provided by Microsoft to help teams plan, develop, test, and deliver software efficiently. It encompasses a variety of components that cater to different stages of the software development lifecycle. here's a brief overview of some key Azure DevOps components:

Azure Boards:

Purpose: Agile project management and work tracking.

Features: Work items (user stories, tasks, bugs), backlogs, sprint planning, and dashboards.

Azure Repos:

Purpose: Version control system.

Features: Git repositories for source control, centralized version control with Team Foundation Version Control (TFVC), pull requests, branching strategies.

Azure Pipelines:

Purpose: Continuous Integration and Continuous Delivery (CI/CD).

Features: Build and release pipelines, support for multiple languages and platforms, integration with various deployment targets.

Azure Test Plans:

Purpose: Comprehensive testing management.

Features: Test case management, test execution, exploratory testing, integration with various test automation frameworks.

Azure Artifacts:

Purpose: Package management.

Features: Store and manage packages, artifacts, and dependencies, integration with build and release pipelines.

Azure DevOps Services:

Purpose: Cloud-based services for development.



Features: Hosted versions of Azure Boards, Azure Repos, Azure Pipelines, Azure Test Plans, and Azure Artifacts.

Azure DevOps Server:

Purpose: On-premises version of Azure DevOps services.

Features: Provides a self-hosted solution for organizations that need to manage their development lifecycle within their own infrastructure.

Azure DevOps Extensions:

Purpose: Customization and integration.

Features: Marketplace with a wide range of extensions for additional functionality, integrations with third-party tools, and custom extensions for specific needs.

These components work together to provide a comprehensive set of tools for managing the entire software development lifecycle, from project planning and coding to testing and deployment.

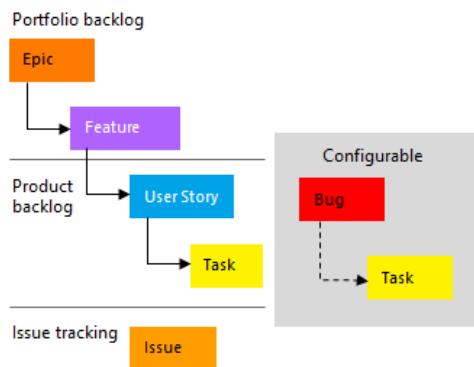
Azure Boards

Azure Boards is a work tracking system that is part of the Azure DevOps services. It is designed to help teams plan, track, and discuss work across the entire development process. Azure Boards supports Agile methodologies and provides flexibility for teams to customize their processes. Below is an elaboration on different processes, work item types, and other options in Azure Boards:

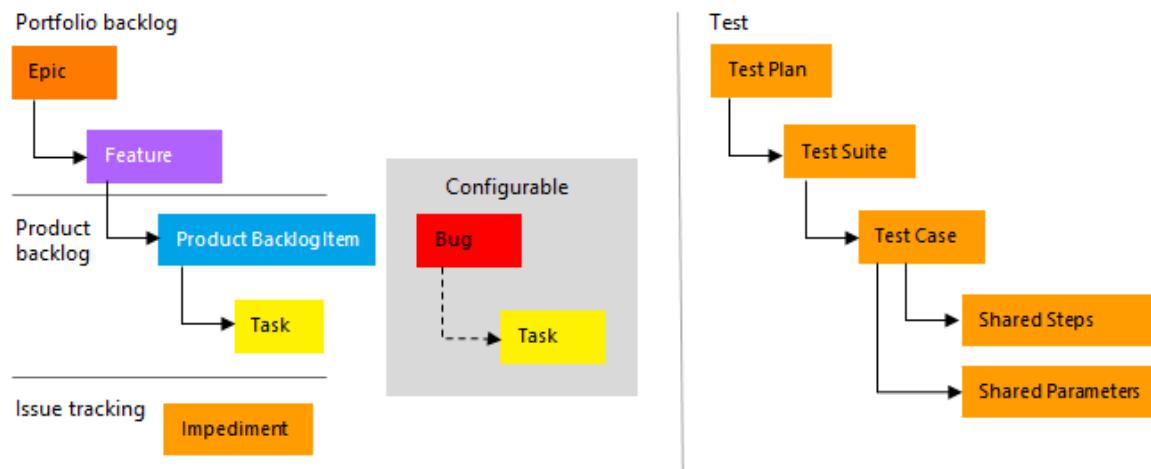
1. Processes:

Azure Boards supports different process models to accommodate various project management methodologies. The 3 primary process models are:

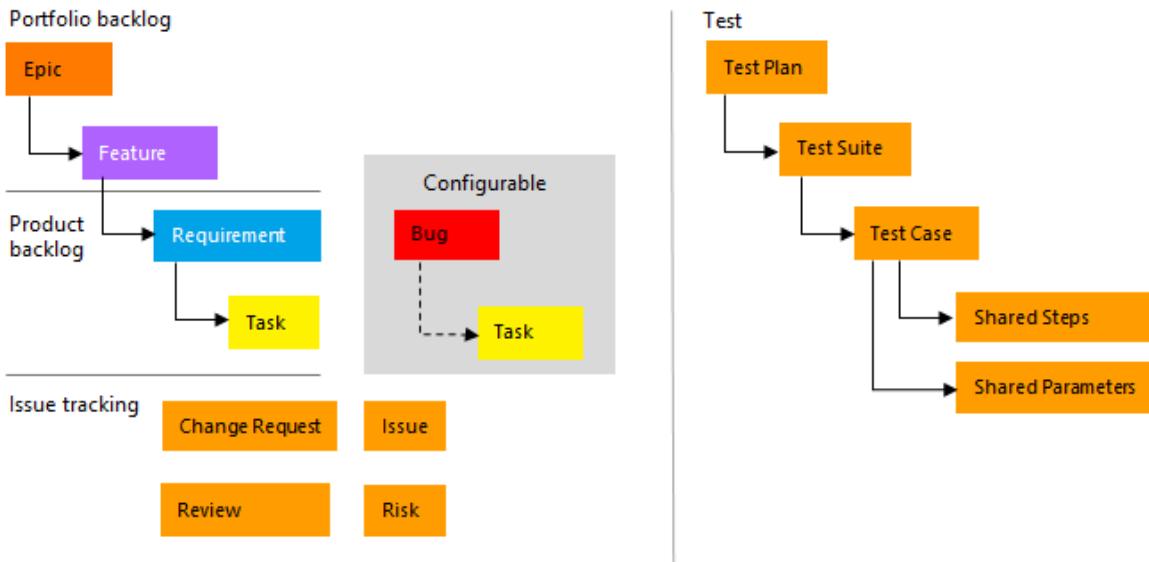
Agile: Emphasizes iterative and incremental development. It is suitable for projects where requirements evolve through collaboration between cross-functional teams.



Scrum: A framework that follows fixed-length iterations (sprints) and includes roles like Scrum Master, Product Owner, and the Scrum Team.



CMMI (Capability Maturity Model Integration): Provides a set of best practices for process improvement and is suitable for organizations that require a higher level of process control.



2. Work Item Types:

Work items in Azure Boards represent tasks, issues, or units of work that need to be tracked. Different work item types are available to cover various aspects of the development process. Common work item types include:

Epic:

Purpose: Represents a large body of work that can be broken down into smaller items such as user stories or tasks.

Characteristics: Typically, epics span multiple sprints and are used to group related user stories or features.

Usage: Epics are used for high-level planning and organizing work on a larger scale.

User Story:

Purpose: Describes functionality from an end user's perspective.

Characteristics: Short, simple descriptions of a feature or functionality. Typically written in non-technical language.

Usage: User stories/Features are used to capture and prioritize requirements from the end user's point of view.

Product Backlog Item (PBI):

Purpose: Represents a high-level functionality or a feature that may span multiple sprints.

Characteristics: Larger than a user story, often breaking down into multiple user stories.

Usage: PBIs are used to plan and prioritize features at a higher level in the product backlog.

Task:

Purpose: Represents a piece of work that needs to be completed within a user story or a PBI.

Characteristics: Specific and detailed breakdown of work required to complete a user story or PBI.

Usage: Tasks are used to track the progress of work at a granular level within a sprint.

Bug:

Purpose: Tracks code defects or issues reported during testing or development.

Characteristics: Describes unexpected behavior or problems in the code that need to be addressed.

Usage: Bugs are used to manage and prioritize the resolution of software defects.

Feature:

Purpose: Represents a high-level group of functionality within the product.

Characteristics: Features are often comprised of multiple epics or user stories.

Usage: Features are used for organizing and managing higher-level functionality within the product.

Impediment:

Purpose: Represents obstacles or issues that are blocking the progress of the team.

Characteristics: Describes challenges or problems that need to be addressed to keep the team on track.

Usage: Impediments are used to bring attention to issues that may affect the team's ability to deliver.

Test Case:

Purpose: Represents a set of test steps and expected results for validating a particular functionality.

Characteristics: Describes conditions and expected outcomes for testing a specific feature.

Usage: Test cases are used in the testing phase to ensure that the implemented features meet the specified requirements.

3. Backlogs:

Backlogs in Azure Boards are used to prioritize and organize work items. There are two primary types of backlogs:

- Product Backlog: Contains user stories, features, and other work items that are not yet assigned to a specific sprint.
- Sprint Backlog: Contains work items that are planned for a specific sprint.

4. Sprints:

Sprints are time-boxed iterations during which a specific set of work items is completed. Azure Boards allows teams to plan and manage sprints, including features like sprint planning, capacity planning, and burndown charts.

5. Boards and Dashboards:

Azure Boards provides Kanban boards that allow teams to visualize and manage their work using a Kanban approach. Dashboards provide a customizable view of project metrics, work item status, and other relevant information.

6. Queries:

Teams can create and save queries to filter and sort work items based on specific criteria. This helps in finding and organizing work items efficiently.

7. Integration with Azure Repos and Pipelines:

Azure Boards integrates seamlessly with Azure Repos for source code version control and Azure Pipelines for CI/CD. This integration ensures a streamlined development process from planning to deployment.

8. Customization:

Azure Boards allows teams to customize their processes, work item types, and fields to adapt to their specific requirements. This flexibility is crucial for teams following unique workflows.

9. Integration with Teams and Notifications:

Azure Boards integrates with Microsoft Teams, allowing teams to stay informed about project updates directly within their collaboration platform. Additionally, teams can set up notifications to receive alerts on changes to work items.

10. Reporting:

Azure Boards provides built-in reporting features, including burndown and velocity charts, to help teams track progress and make data-driven decisions.

Azure Repos

Azure Repos is a version control system provided by Microsoft as part of the Azure DevOps services. It supports both Git and Team Foundation Version Control (TFVC) to help teams manage and track changes to their source code. In this response, I'll focus on Git commands since Azure Repos primarily uses Git for version control.

Key Git Commands:

git init

Purpose: Initializes a new Git repository.

Usage: git init

git clone

Purpose: Creates a copy of a remote repository locally.

Usage: git clone <repository-url>

git add

Purpose: Adds changes to the staging area, preparing them for commit.

Usage: git add <file(s)>

git commit

Purpose: Records changes made to the repository.

Usage: git commit -m "Commit message"

git status

Purpose: Shows the status of changes as untracked, modified, or staged.

Usage: git status

git pull

Purpose: Fetches changes from a remote repository and merges them into the local branch.

Usage: git pull

git push

Purpose: Uploads local branch commits to a remote repository.

Usage: git push <remote> <branch>

git branch

Purpose: Lists existing branches or creates a new branch.

Usage: git branch (list branches), git branch <branch-name> (create branch)

git checkout

Purpose: Switches to a different branch or commit.

Usage: git checkout <branch-name> or git checkout <commit-hash>

git merge

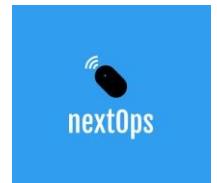
Purpose: Combines changes from different branches.

Usage: git merge <branch-name>

git log

Purpose: Displays a log of commits.

Website: <https://www.nextops.in> , YouTube: <https://www.youtube.com/c/nextopsvideos>
WhatsApp: +91 73309 77091, Email: support@nextops.in



Usage: git log

git fetch

Purpose: Retrieves changes from a remote repository without merging them.

Usage: git fetch

git remote

Purpose: Manages connections to remote repositories.

Usage: git remote -v (list remotes), git remote add <remote-name> <repository-url> (add remote)

git diff

Purpose: Shows changes between commits, branches, or between the working directory and the staging area.

Usage: git diff or git diff <commit1> <commit2>

git reset

Purpose: Resets the repository's state to a specific commit.

Usage: git reset --hard <commit-hash> (discarding changes), git reset --soft <commit-hash> (keeping changes)

git tag

Purpose: Tags a specific commit with a version identifier.

Usage: git tag -a <tag-name> -m "Tag message" <commit-hash>

git stash

Purpose: Temporarily saves changes that are not ready to be committed.

Usage: git stash, git stash apply (to reapply stashed changes)

`git stash push` saves the work in progress without committing it.

`git stash list` to list all the saved

git stash apply	to add it to staging area again
git stash pop	to remove the WIP from stash
git init	
git branch	
git branch -m main	Rename Branch
git status	create a new file and issue untracked files will show up in red color then issue
git add . (or) git add <filename>	then issue
git status	again changes to be committed show up in green color if you want to untrack issue
git rm --cached <filename>	then issue
git commit	without a comment and see what it says
git commit -m "commit msg"	
git config --global user.name "username"	
git config --global user.email "emailid"	
git log	lists all the commits but the output is not good HEAD -> main reflects the current branch we are in
git branch <branch_name>	to create a new branch
git branch	lists out all the branches
git log	observe HEAD -> main, new_branch

git checkout <branch_name>	to switch to new branch
git status	to check which branch we are in
git log	to verify the branch HEAD -> new_branch, main
	now edit the existing file and issue
git status	it shows changes not staged for commit
	create another new file and issue
git status	again, now it shows changed not staged for commit (modified file) and untracked files (another new file)
	to only commit modified file in a separate commit
git add <modified_file>	then issue
git status	it shows changes to be committed an untracked files
committed not	then issue git commit, so only modified file gets committed not
	another new file.
git commit -m "commit msg"	then issue
git status	only shows untracked files
	then issue
git log	and observe new commit is pointing to new_branch and old commit is pointing to old branch HEAD ->
new_branch	then finally issue
git log --all --oneline	shows history from all branches and in one line
git add <another_new_file>	
git commit -m "commit msg"	
git status	after this
git checkout main	
git status	it is clean, and files from new_branch are missing here
git merge <new_branch>	to merge changes from new_branch to main branch

git log --all --oneline	shows history from all branches and in one line
git branch -d <new_branch>	to delete a branch

Azure Pipelines

Azure Pipelines is a continuous integration and continuous delivery (CI/CD) service provided by Microsoft as part of the Azure DevOps suite. It enables teams to automate the building, testing, and deployment of their applications. With Azure Pipelines, you can set up robust and flexible workflows to streamline the software development process. Let's elaborate on the various aspects and options available in Azure Pipelines:

1. Pipeline Types:

Azure Pipelines supports different pipeline types to accommodate various development scenarios:

Build Pipelines: These pipelines automate the process of building and packaging your application.

Release Pipelines: These pipelines automate the deployment and release of your application.

2. YAML-based Pipelines:

Azure Pipelines allows you to define your build and release pipelines using YAML syntax, making it easy to version control, review changes, and manage as code.

3. Build Pipelines:

- Triggers: Define when a build should be triggered, such as on every push, on a scheduled basis, or manually.
- Agent Pools: Specify the environment in which the build runs, such as Microsoft-hosted agents or your own self-hosted agents.
- Jobs and Steps: Organize tasks into jobs, and each job consists of a sequence of steps. Steps can include tasks like restoring dependencies, building code, running tests, and publishing artifacts.
- Artifacts: Store and publish build artifacts for deployment in the release pipeline.

4. Release Pipelines:

- Stages: Define different environments or stages for your deployment pipeline, such as Dev, Test, and Production.
- Approvals: Configure manual approvals before deploying to specific stages.
- Deployments: Define tasks and jobs for deploying your application to different environments. Support for multi-stage deployments.

- Variables: Use variables to parameterize your deployment process.
- Artifacts: Define artifacts to deploy, which can be the output of a build pipeline or external artifacts.

5. Integration with Azure Repos and GitHub:

Seamless integration with Azure Repos and GitHub repositories allows you to trigger builds on code changes and manage your CI/CD pipelines alongside your source code.

6. Multi-Platform Support:

Azure Pipelines supports building and deploying applications for various platforms, including Windows, Linux, macOS, and different programming languages.

7. Agent Pools and Agents:

Microsoft-Hosted Agents: Azure provides pre-configured agents hosted by Microsoft with a variety of tools and runtimes.

Self-Hosted Agents: You can set up your own agents on your infrastructure or cloud environments for more customization.

8. Environments:

Define Environments: Organize and manage your deployment targets in logical environments.

Environment Checks: Perform checks before and after deploying to environments.

9. Integration with Azure Services:

Azure Pipelines integrates seamlessly with various Azure services, allowing you to deploy to Azure App Service, Azure Kubernetes Service (AKS), Azure Functions, etc.

10. Tasks and Extensions:

A rich marketplace of tasks and extensions allows you to extend the functionality of your pipelines with integrations for third-party tools and services.

11. Security and Compliance:

Azure Pipelines provides features like role-based access control (RBAC), audit logs, and secure storage for secrets to ensure security and compliance.

12. Parallel Jobs and Matrix Builds:

Execute jobs in parallel to speed up the build process. Matrix builds allow running multiple configurations of a build in parallel.

13. Agent Pools and Scaling:

Azure Pipelines allows you to scale your build and release pipelines by distributing the workload across multiple agents in parallel.

14. Deployment Gates:

Integrate deployment gates to enable automatic or manual checks on your deployment before moving to the next stage.

15. Monitoring and Reporting:

Gain insights into the status and health of your pipelines with detailed logs, reports, and integration with Azure Monitor.

Agent Pools and Agents

In Azure DevOps, agent pools and agents play a crucial role in the execution of build and release pipelines. They allow you to run your CI/CD processes on the necessary infrastructure, whether it's hosted by Microsoft (Microsoft-hosted agents) or on your own infrastructure (self-hosted agents). Let's delve into the details of agent pools and agents:

Agent Pools:

An agent pool is a logical group that contains one or more build and release agents. These pools help organize and manage agents based on shared characteristics, permissions, or requirements. In Azure DevOps, you can create multiple agent pools to accommodate different scenarios or teams within your organization. Here are key points about agent pools:

Types of Agent Pools:

Default Agent Pool: All projects have a default agent pool, and it is shared across all projects within an organization.

User-Created Agent Pools: You can create additional agent pools to group agents based on specific requirements, such as security, geographical location, or team ownership.

Permissions:

Security: Agent pools have security settings to control who can manage and use agents within the pool.

Agent Pool Administrators: Users with the appropriate permissions can manage agent pools and configure settings.

Organization Scope:

Agent pools are organized at the organization level and can be shared across projects within the same organization.

Use Cases:

Scenario-Based Pools: Create pools for specific scenarios, such as a pool for Linux agents, a pool for Windows agents, etc.

Project-Specific Pools: Organize agents based on project-specific requirements.

Agents:

An agent is a piece of software responsible for running tasks in a build or release pipeline. It communicates with Azure Pipelines and executes the defined tasks on the target machine. Agents can be either hosted by Microsoft (Microsoft-hosted agents) or hosted on your own infrastructure (self-hosted agents). Here are key points about agents:

Microsoft-Hosted Agents:

Pre-configured Environments: Microsoft provides agents with pre-configured environments for various platforms (Windows, Linux, macOS).

Maintenance-Free: Microsoft-hosted agents are maintained by Microsoft, and you don't need to worry about managing their infrastructure.

Self-Hosted Agents:

Customizable Environments: You can set up self-hosted agents on your infrastructure, allowing for more customization of the build/release environment.

Security: Self-hosted agents can be configured with security settings and are often used in scenarios where the build/release environment requires specific configurations or access to local resources.

Scaling: Self-hosted agents can be scaled horizontally by adding more machines to the agent pool, allowing for increased parallelism.

Agent Pools and Agents Relationship:

Agent Pool Assignment: Each agent is associated with an agent pool.

Multiple Agents in a Pool: An agent pool can have multiple agents, and these agents can run concurrently.

Assignable to Multiple Pools: An agent can be assigned to multiple agent pools if needed.

Registration and Communication:

Agent Registration: Agents need to be registered with Azure DevOps before they can be used in pipelines.

Communication: Agents communicate with Azure DevOps to receive tasks, download required dependencies, and report back the results.

Pipeline Execution:

Task Execution: Agents execute the tasks defined in the build or release pipeline.

Parallel Execution: Multiple agents in the same pool can execute tasks concurrently, improving pipeline performance.

Retention Policies:

Agent Retention Policies: Define how long an agent is kept running after it has finished a job, reducing startup time for subsequent jobs.

Use Cases:

Microsoft-Hosted Agents:

Suitable for projects with generic build and deployment requirements.

Convenient for projects with minimal customization needs.

Self-Hosted Agents:

Ideal for projects with specific build or deployment requirements that require custom environments.

Useful for accessing local resources or interacting with on-premises systems.

Pipeline Hierarchy

In Azure Pipelines, the hierarchy of a pipeline is structured to allow the definition of complex, multi-stage workflows for building, testing, and deploying applications. The key components of this hierarchy are stages, jobs, and steps. Let's explore each of them in detail:

1. Pipeline:

A pipeline is the top-level container that defines the entire CI/CD process for your application. It consists of one or more stages.

2. Stages:

A stage is a logical division within a pipeline that represents a phase in the overall process. Stages can be used to separate different parts of your deployment pipeline, such as deploying to different environments (e.g., Dev, Test, Prod). The stages within a pipeline run sequentially, and each stage can contain one or more jobs.

Key Properties:

- Name: A user-defined name for the stage.
- Dependencies: Stages can have dependencies on other stages, defining the order of execution.

Usage:

- Parallel Execution: Stages can run in parallel if there are no dependencies between them.
- Sequential Execution: By defining dependencies, you can ensure stages run in a specific order.

3. Jobs:

A job is a collection of steps that run on the same agent. It represents a unit of work, such as building code, running tests, or deploying to a specific environment. Multiple jobs within the same stage can run in parallel on different agents or sequentially on the same agent.

Key Properties:

- Name: A user-defined name for the job.
- Agent Pool: The pool of agents where the job will run.
- Environment: The target environment for the job, which may be defined for specific scenarios.

Usage:

- Parallel Execution: Multiple jobs within the same stage can execute in parallel.
- Sequential Execution: Jobs within a stage can be configured to run sequentially.

4. Steps:

A step is an individual task within a job. It represents a single action, such as running a script, copying files, or invoking a build task. Steps define the actions that need to be performed to accomplish the goals of the job.

Key Properties:

- Name: A user-defined name for the step.
- Script: The script or command to be executed as part of the step.
- Inputs: Additional configuration parameters specific to the task being executed.

Usage:

- Task Execution: Steps define the tasks that make up a job.
- Customization: Steps can be customized with specific scripts or commands.

Example YAML-based Pipeline Hierarchy:

```
stages:  
- stage: Build  
  jobs:  
    - job: BuildJob  
      pool:  
        vmImage: 'windows-latest'  
      steps:  
        - script: echo 'Building the application'  
  
    - stage: Test  
      jobs:  
        - job: TestJob  
          pool:  
            vmImage: 'windows-latest'  
          steps:  
            - script: echo 'Running tests'  
  
    - stage: Deploy  
      jobs:  
        - job: DeployJob  
          pool:  
            vmImage: 'windows-latest'
```

```
steps:  
- script: echo 'Deploying to production'
```

Deployment Groups

Deployment groups in Azure Pipelines are a concept designed to simplify the deployment of applications to multiple target machines, such as servers or virtual machines, in a more orchestrated and organized manner. A deployment group is a collection of target machines grouped together for deploying applications.

Key Concepts:

Deployment Pool:

A deployment pool is a logical grouping of target machines. Each deployment pool contains one or more deployment groups.

Deployment Group:

A deployment group is a set of target machines with similar characteristics, such as operating system, role, or environment. It is created within a deployment pool and represents a group of machines where you want to deploy your application.

Agent:

An agent is a piece of software installed on each target machine within a deployment group. The agent is responsible for executing deployment tasks on the target machine.

Use Case and Detailed Explanation:

Scenario:

Consider a scenario where you have a web application that needs to be deployed to multiple servers in different environments, such as Dev, Test, and Production. Instead of managing deployments to each server individually, you can use deployment groups to simplify the process.

Steps to Set Up Deployment Groups:

Create Deployment Pool:

Start by creating a deployment pool. This is done at the project level in Azure DevOps.

Website: <https://www.nextops.in> , YouTube: <https://www.youtube.com/c/nextopsvideos>
WhatsApp: +91 73309 77091, Email: support@nextops.in

Define the characteristics that group machines together. For example, you might create a deployment pool for machines running Windows Server 2019.

Add Deployment Groups:

Within the deployment pool, create deployment groups. Each deployment group represents a specific environment (e.g., Dev, Test, Prod).

Assign target machines to each deployment group. These machines could be virtual machines, physical servers, or even cloud-based instances.

Install Agents:

Install the Azure Pipelines Agent on each target machine within a deployment group. Each agent is associated with a specific deployment group and is responsible for executing deployment tasks.

Configure Deployment Jobs:

In your Azure Pipelines YAML or through the Azure DevOps interface, configure deployment jobs that target the deployment groups.

Define deployment steps, such as copying files, running scripts, or invoking other tasks required for deploying your application.

Deploy to Multiple Machines:

When you trigger a deployment, Azure Pipelines will automatically distribute the deployment tasks to all the machines within the specified deployment group.

Parallel deployment to multiple machines ensures faster and more efficient deployments.

Benefits of Deployment Groups:

Orchestration:

Deployment groups provide a centralized and orchestrated way to deploy applications to multiple machines. Tasks are automatically distributed to all machines within a deployment group.

Environment Configuration:

Deployment groups allow you to organize machines based on their environment (e.g., Dev, Test, Prod). This helps in managing configurations specific to each environment.

Parallel Execution:

Deployments to different machines within a deployment group can run in parallel, improving deployment speed.

Centralized Management:

Machines within a deployment group can be managed centrally. Updates to deployment tasks or configurations can be applied uniformly to all machines in the group.

Rollback and Monitoring:

Deployment groups support rollback scenarios. If an issue is detected during deployment, it can be rolled back easily.

Monitoring capabilities allow you to track the progress of deployments on each machine.

In Azure DevOps, environments are logical entities that help organize and manage deployment targets within a pipeline. Environments provide a way to group resources, such as servers or Kubernetes clusters, and define the stages through which an application passes during its deployment lifecycle. Environments are primarily used in the context of release pipelines to model and control the progression of an application through different deployment stages.

Here are key aspects of environments in Azure DevOps:

1. Logical Containers:

Purpose: Environments are logical containers that represent different deployment targets or stages in your release pipeline.

Use Case: You might have environments for development, testing, staging, and production.

2. Deployment Targets:

Purpose: Environments define the target destinations for your application, such as servers, Kubernetes clusters, or other infrastructure.

Use Case: An environment could represent a set of servers where you deploy and test your application.

3. Environment Resources:

Purpose: Resources within an environment represent the actual deployment targets, like virtual machines, physical servers, or other infrastructure components.

Use Case: Resources in an environment might include specific servers or clusters.

4. Role-Based Access Control (RBAC):

Purpose: Environments support RBAC to control who can view, modify, or deploy to specific environments.

Use Case: You can restrict access to production environments to only authorized personnel.

5. Security and Compliance:

Purpose: Environments can be configured with security policies and compliance checks.

Use Case: Implement security checks or approvals before deploying to critical environments.

6. Policies and Checks:

Purpose: Environments allow you to define policies and checks that must be satisfied before deploying to that environment.

Use Case: Implement pre-deployment and post-deployment checks in each environment.

7. Pre- and Post-Deployment Approvals:

Purpose: Environments support manual approvals before deploying to the environment.

Use Case: Require manual approval before deploying to a production environment.

8. Integration with Release Pipelines:

Purpose: Environments are primarily used within release pipelines to model the deployment stages.

Use Case: Define stages in a release pipeline corresponding to different environments.

9. Environment Variables:

Purpose: Environment-specific variables can be defined, allowing you to parameterize your deployment process for each environment.

Use Case: Configure connection strings or other environment-specific settings dynamically.

10. RetentionPolicy:

Purpose: Environments can be configured with a retention policy to keep or automatically delete deployments after a certain period.

Use Case: Control how long deployment history is retained for auditing or compliance purposes.

11. Integration with Azure Services:

Purpose: Environments seamlessly integrate with various Azure services, facilitating deployments to Azure resources.

Use Case: Directly deploy applications to Azure App Service, Kubernetes Service, etc., as part of your environment.

12. Deployment Gates:

Purpose: Implement deployment gates in environments to automatically validate the health of the deployment.

Use Case: Perform automated checks, like monitoring metrics or running automated tests, before allowing the deployment to proceed.

13. Monitoring and Insights:

Purpose: Environments provide insights and monitoring capabilities to track the status of deployments in each environment.

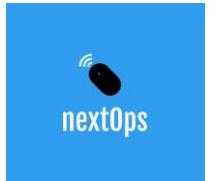
Use Case: Monitor deployment progress and detect issues in real-time.

Library and Secure Files

In Azure Pipelines, libraries and secure files are features that help manage and securely store shared resources such as variables, credentials, or files. These features contribute to the efficiency, security, and maintainability of CI/CD workflows.

Library:

Azure Pipelines libraries are used to store and manage shared resources such as variable groups and secure file variables. These resources can be used across multiple pipelines, providing a centralized and maintainable way to manage configurations.



Key Components of Libraries:

Variable Groups:

Variable groups are a set of variables defined in a library that can be linked to multiple pipelines.

They allow you to define variables like connection strings, API keys, or any configuration that needs to be reused across multiple pipelines.

Changes to a variable group propagate to all pipelines using that group.

Secure Files:

Secure files are files that contain sensitive information, such as certificates or private keys.

Secure files are uploaded to the library, and they can be associated with pipeline jobs securely.

Only jobs running on agents can access the contents of secure files.

Use Cases:

Storing shared variables, like database connection strings, in a variable group.

Managing sensitive files, like SSL certificates, in secure files.

Secure Files:

Secure files in Azure Pipelines provide a secure way to upload and share files with sensitive information across pipelines. These files are securely stored and can be associated with pipeline jobs for deployment or other purposes.

Key Features of Secure Files:

Secure Storage:

- Secure files are stored securely and can only be accessed during job execution on agent machines.

Linking to Jobs:

- Secure files can be linked to specific jobs within a pipeline, allowing the files to be used in the context of that job.

Uploading and Downloading:

- Secure files can be uploaded to the Azure Pipelines library and downloaded during pipeline execution on agent machines.

Website: <https://www.nextops.in> , **YouTube:** <https://www.youtube.com/c/nextopsvideos>
WhatsApp: +91 73309 77091, **Email:** support@nextops.in

By leveraging libraries and secure files in Azure Pipelines, teams can centralize the management of shared resources, enhance security by storing sensitive information securely, and promote consistency across pipelines. These features contribute to the maintainability and reliability of CI/CD workflows.

Variables (User defined and Pre-defined)

In Azure Pipelines, variables are used to store and manage configuration settings or values that can be reused across various tasks and stages within a pipeline. Variables provide a way to parameterize your pipeline, making it more flexible and allowing you to customize the behavior of tasks and jobs. There are two main types of variables in Azure Pipelines: user-defined variables and predefined variables.

1. User-Defined Variables:

Defining User-Defined Variables:

You can define user-defined variables in Azure Pipelines using the variables keyword in the YAML pipeline file.

```
variables:
```

```
  myVariable: 'This is a user-defined variable'  
  connection_string: $(myVariable)-$(Build.BuildId)
```

2. Predefined Variables:

Azure Pipelines provides a set of predefined variables that contain information about the build or release environment. These variables are automatically populated by the system, and you can use them without explicitly defining them in your pipeline.

Examples of Predefined Variables:

Build.BuildId: The ID of the current build.

Build.SourceBranch: The source branch from which the build was triggered.

Build.SourceVersion: The source version for the current build.

System.DefaultWorkingDirectory: The default working directory for tasks in a job.

Using Predefined Variables:

You can reference predefined variables using the `$(variable_name)` syntax. For example:

```
variables:  
  myBuildId: $(Build.BuildId)  
  
jobs:  
- job: MyJob  
  pool:  
    vmImage: 'windows-latest'  
  steps:  
- task: PowerShell@2  
  inputs:  
    targetType: 'inline'  
    script: |  
      Write-Host "Current Build ID: $(myBuildId)"
```

Reference: [Predefined variables - Azure Pipelines | Microsoft Learn](#)

Tasks

In Azure Pipelines, tasks are the building blocks of your build and release pipelines. Tasks are individual units of work that perform specific actions, such as running a script, copying files, or deploying an application. Each task represents a step in the pipeline and contributes to the overall workflow. Azure Pipelines provides a variety of built-in tasks, and you can also create custom tasks to suit your specific needs.

Built-in Tasks:

Azure Pipelines comes with a rich set of built-in tasks for common actions like copying files, running scripts, publishing artifacts, and more.

Custom Tasks:

You can create custom tasks using scripts or container-based tasks to perform actions specific to your application or environment.

Task Groups:

Task groups allow you to encapsulate a sequence of tasks into a single reusable unit. This promotes maintainability and reusability across pipelines.

Marketplace Tasks:

The Azure DevOps Marketplace offers a variety of extension tasks that can be added to your pipelines to integrate with third-party services, tools, or platforms.

Reference: [Azure Pipelines task reference | Microsoft Learn](#)

Triggering a pipeline

In Azure DevOps, there are several ways to trigger a pipeline. The triggering mechanism depends on the type of pipeline (build or release) and the specific requirements of your workflow. Here are some common ways to trigger pipelines:

1. Manual Trigger:

You can manually trigger a pipeline when you need to run it on-demand. This is useful for scenarios like testing changes, running ad-hoc builds, or triggering releases manually.

Build Pipeline:

- Navigate to the "Pipelines" page.
- Find and select the build pipeline you want to trigger.
- Click on the "Run pipeline" button.

Release Pipeline:

- Navigate to the "Releases" page.
- Find and select the release pipeline you want to trigger.
- Click on the "Create a release" button, then click on "Create."

2. Continuous Integration Trigger:

Continuous Integration (CI) triggers automatically start a build when changes are pushed to the repository. This ensures that your code is automatically built and tested whenever there are changes.

Build Pipeline:

In the YAML definition, configure the trigger section to specify branches or paths that should trigger the build on changes.

```
trigger:
```

```
  branches:
```

```
    include:
```

Website: <https://www.nextops.in> , YouTube: <https://www.youtube.com/c/nextopsvideos>
WhatsApp: +91 73309 77091, Email: support@nextops.in

- main

Classic Editor (UI):

Navigate to the "Triggers" tab in the build pipeline.

Enable the "Enable continuous integration" option.

3. Scheduled Trigger:

You can schedule pipelines to run at specific times using a scheduled trigger. This is useful for periodic tasks, such as nightly builds or scheduled releases.

Build Pipeline:

In the YAML definition, configure the schedules section to specify the schedule for triggering the build.

schedules:

```
- cron: "0 0 * * *"  
  displayName: Daily midnight build  
branches:  
  include:  
    - main
```

Classic Editor (UI):

Navigate to the "Triggers" tab in the build pipeline.

Enable the "Enable scheduled builds" option.

4. Branch Filters:

You can configure pipelines to trigger only for specific branches or paths, ensuring that changes in certain branches trigger the pipeline.

Build Pipeline:

In the YAML definition, configure the trigger section with branch filters.

trigger:

branches:

Website: <https://www.nextops.in> , YouTube: <https://www.youtube.com/c/nextopsvideos>
 WhatsApp: +91 73309 77091, Email: support@nextops.in

```
include:
```

- main
- feature/*

Classic Editor (UI):

Navigate to the "Triggers" tab in the build pipeline.

Specify branch filters under the "Branch filters" section.

5. Pull Request Trigger:

For build pipelines, you can configure triggers to run builds automatically when pull requests are created or updated.

```
pr:
```

```
  branches:
```

```
    include:
```

- '*'

Classic Editor (UI):

Navigate to the "Triggers" tab in the build pipeline.

Enable the "Enable pull request validation" option.

Parallel Jobs

In Azure DevOps, parallel jobs refer to the capability of running multiple jobs concurrently within a single build or release pipeline. Parallel jobs enable the concurrent execution of tasks, allowing for faster overall build or release times. This feature is particularly beneficial when dealing with large and complex projects, as it allows teams to distribute workloads across multiple agents, taking advantage of parallel processing.

Here are key points regarding parallel jobs in Azure DevOps:

1. Concurrency:

Parallel jobs enable concurrent execution of jobs within a single stage of a pipeline.

Concurrency refers to the number of parallel jobs that can run simultaneously.

2. Parallelism in Jobs:

Jobs within a stage can be configured to run in parallel, providing a mechanism to divide the workload.

Each job runs independently, and their execution is not dependent on the completion of others.

3. Benefits:

Performance Improvement: Parallel jobs can significantly reduce the overall build or release time by distributing tasks across multiple agents simultaneously.

Resource Utilization: Multiple agents can work on different jobs concurrently, utilizing available resources efficiently.

4. Configuration:

Parallelism is configured at the job level within a stage in the YAML pipeline.

The strategy section in the job definition specifies the parallelism settings.

Example YAML snippet:

```
jobs:  
- job: MyJob  
  pool:  
    vmImage: 'windows-latest'  
  strategy:  
    parallel: 3  
  steps:  
    - script: echo "This is a parallel job"
```

5. Parallel Matrix Builds:

Matrix builds in Azure Pipelines allow for parallel execution of jobs across different dimensions, such as operating systems, platforms, or configurations.

A matrix build runs multiple instances of a job with different configuration values.

6. Parallel Deployment Jobs:

In release pipelines, parallel jobs can be used to deploy different components or environments concurrently.

This is particularly useful in scenarios where multiple environments need to be deployed simultaneously.

Example Use Case:

Consider a scenario where a CI/CD pipeline involves building and testing a web application. The pipeline has multiple stages, including a build stage and a deployment stage. In the build stage, parallel jobs are used to:

Job 1: Build the frontend application.

Job 2: Build the backend application.

In the deployment stage, parallel jobs could be used to deploy the application to different environments concurrently:

Job 3: Deploy to the staging environment.

Job 4: Deploy to the production environment.

By utilizing parallel jobs, each of these tasks can be executed concurrently, reducing the overall pipeline execution time.

Deployment Strategies

In Azure Pipelines, deployment strategies refer to the different approaches or techniques used to deploy applications or updates to different environments. These strategies are designed to manage the release process, mitigate risks, and ensure the smooth delivery of software to various stages in the development lifecycle. Azure Pipelines supports various deployment strategies, and you can customize your deployment process based on your specific requirements.

Here are some common deployment strategies in Azure Pipelines, along with concepts like lifecycle hooks:

1. Rolling Deployment:

Rolling deployment is a gradual and incremental deployment strategy where updates are applied to a subset of instances at a time, gradually rolling through all instances.

It minimizes downtime and allows for easy rollback in case of issues.

Azure Pipelines Configuration:

Define deployment jobs for each stage (e.g., staging, production).

Use strategies like deployment gates to validate the health of each instance before continuing.

2. Blue-Green Deployment:

Blue-Green deployment involves having two environments: "Blue" represents the current production environment, and "Green" represents the new version.

The switch between Blue and Green is done instantly when the new version is validated and ready for production.

Azure Pipelines Configuration:

Configure two environments (e.g., blue and green).

Use deployment jobs to deploy the new version to the green environment.

Use lifecycle hooks to coordinate the switch from blue to green.

3. Canary Deployment:

Canary deployment releases the new version to a small subset of users or instances initially.

It allows monitoring the new version's performance and user feedback before rolling it out to the entire environment.

Azure Pipelines Configuration:

Use deployment jobs to release the new version to a limited set of instances.

Employ deployment gates and monitoring to validate the new version's success before proceeding.

Lifecycle Hooks:

Lifecycle hooks in Azure Pipelines are a set of extensibility points that allow you to inject custom logic or actions at different stages of the deployment process. These hooks enable you to perform additional tasks, validations, or custom actions before or after specific deployment events.

Pre-Deployment Approval:

Triggered before deployment and can include actions like manual approvals or custom validations.

Pre-Deployment Gate:

Executed before deployment and can include checks on release gates, health signals, or custom criteria.

Post-Deployment Approval:

Triggered after deployment and can include post-deployment validations or approvals.

Post-Deployment Gate:

Executed after deployment and can include checks on release gates, health signals, or custom criteria.

Reference: [Deployment jobs - Azure Pipelines | Microsoft Learn](#)

Test Plans

In Azure DevOps, Test Plans are a part of the Azure Test Plans service, which is designed to help teams plan, track, and manage their testing efforts. Test Plans provide a centralized and organized way to manage test cases, track test execution, and collect test results. Here's a detailed explanation of Test Plans in Azure DevOps:

Key Components of Test Plans:

Test Suites:

Website: <https://www.nextops.in> , **YouTube:** <https://www.youtube.com/c/nextopsvideos>
WhatsApp: +91 73309 77091, **Email:** support@nextops.in

Test Suites are containers that organize and group test cases. They help structure your testing efforts into logical groups, such as functional areas or scenarios.

Test Suites can be organized hierarchically, allowing for a nested structure.

Test Cases:

Test Cases define the specific steps, inputs, and expected results for testing a particular functionality. They serve as the building blocks for test execution.

Test Cases can be associated with different Test Suites and Test Plans.

Test Configurations:

Test Configurations allow you to define different configurations for test execution, such as testing on different browsers or operating systems.

Test Configurations can be associated with Test Cases to specify the testing conditions.

Test Runs:

Test Runs are instances of test execution. They represent a specific iteration of running a set of test cases.

Test Runs capture details like the test environment, test configurations, and results.

Test Results:

Test Results provide detailed information about the outcome of individual test cases within a Test Run.

Results include pass/fail status, any associated bugs, and additional details about the test execution.

Parameters and Data-Driven Testing:

Test Plans support parameters and data-driven testing, allowing you to run the same test case with different inputs.

Exploratory Testing:

Azure DevOps supports exploratory testing, allowing testers to perform ad-hoc testing without predefined test cases. Exploratory testing sessions can be recorded and logged.

Test Plan Workflow:

Test Planning:

Create Test Suites and organize Test Cases within them based on your testing requirements.

Define Test Configurations if needed, specifying the conditions under which tests should be executed.

Test Case Design:

Define Test Cases with detailed steps, expected results, and any necessary attachments or links to requirements.

Test Execution:

Create Test Runs to execute a set of Test Cases in a specific environment.

Execute tests manually or leverage automated test scripts.

Capture Results:

As tests are executed, capture results, including pass/fail status and any associated bugs.

Log additional details about the execution environment or issues encountered during testing.

Review and Analyze:

Review Test Results to identify trends, analyze pass rates, and track progress.

Use analytics and reporting features to gain insights into testing effectiveness.

Integration with Azure Boards:

Test Plans are integrated with Azure Boards, allowing seamless collaboration between development and testing teams.

Test Cases can be linked to work items, allowing for traceability between requirements and test coverage.

Bugs identified during testing can be linked to Test Cases and associated work items.

Integration with Azure Pipelines:

Test Plans integrate with Azure Pipelines, enabling the inclusion of automated tests within the continuous integration and continuous delivery (CI/CD) pipeline.

Test results from automated tests are captured and reported within Test Plans.

Test Configurations:

Define different settings for test execution environments.

Support parameters and data-driven testing.

Associated with individual test cases.

Test Suites:

Website: <https://www.nextops.in> , **YouTube:** <https://www.youtube.com/c/nextopsvideos>
WhatsApp: +91 73309 77091, **Email:** support@nextops.in

Organize test cases logically.

Can be static or query-based.

Define the execution order of test cases.

Linked with Test Plans for structured testing.

Azure Artifacts

Azure Artifacts is a part of the Azure DevOps services provided by Microsoft. It is a package management system that helps development teams manage and share dependencies, such as libraries and packages, within their projects. Azure Artifacts provides a secure and centralized repository for hosting and managing packages, enabling teams to streamline their build and release processes.

Here are the key features and components of Azure Artifacts:

1. Package Types:

Azure Artifacts supports multiple package types, including:

NuGet Packages: For .NET projects and other compatible ecosystems.

npm Packages: For Node.js and JavaScript projects.

Maven Packages: For Java projects.

Universal Packages: For any type of package, allowing flexibility.

2. Package Repositories:

Azure Artifacts provides private, secure, and scalable package repositories for each supported package type.

Each repository can be used to store and version packages, ensuring reproducibility and consistency in builds and releases.

3. Scoped Feeds:

Feeds in Azure Artifacts can be scoped to a project or organization, providing isolation and access control.

Scoped feeds enable teams to manage packages within the context of their specific projects.

4. Security and Access Control:



Azure Artifacts integrates with Azure DevOps security and access control mechanisms.

Users and teams can be granted specific permissions to publish, download, or manage packages within feeds.

5. Build Integration:

Azure Artifacts seamlessly integrates with Azure Pipelines, allowing for the automatic publishing and consumption of packages during build and release processes.

Build artifacts, such as NuGet packages, can be published to Azure Artifacts feeds for versioning and sharing.

6. Package Versioning:

Packages in Azure Artifacts are versioned, providing a clear and structured approach to managing dependencies.

Versioning helps ensure that builds are reproducible and that the correct versions of packages are used in projects.

7. Package Search:

Azure Artifacts provides a package search feature, making it easy for developers to discover and consume packages.

Users can search for packages based on name, version, or other criteria.

8. Retention Policies:

Azure Artifacts supports retention policies, allowing organizations to manage the lifecycle of packages.

Retention policies help clean up old or unused packages, optimizing storage and maintaining a clean repository.

9. Universal Packages:

Universal Packages in Azure Artifacts are versatile and can be used to store any type of package or artifact.

They are suitable for scenarios where custom or non-standard artifacts need to be managed.

10. Integration with External Registries:

Azure Artifacts allows you to connect to external package registries, enabling you to proxy and cache packages from public registries like npmjs.org or NuGet.org.

11. Cross-Organization Feeds:

Cross-organization feeds enable sharing packages across different Azure DevOps organizations.

12. Package Views:

Azure Artifacts provides different views for packages, allowing users to see the latest versions, explore package details, and manage package versions.

Use Case Example:

Consider a scenario where a development team is working on a .NET Core project. They can use Azure Artifacts to create a NuGet feed to host and manage their project's NuGet packages. During the build process in Azure Pipelines, the project's packages are automatically published to the NuGet feed. Other teams or projects within the organization can then consume these packages from the feed, ensuring a consistent and reliable set of dependencies.