**Azure Boards**

Azure Boards is a project management tool within Azure DevOps that helps teams plan, track, and discuss work across the entire project lifecycle. It offers features like:

* **Work Planning**: Create and manage backlogs, user stories, and tasks using customizable work item types.
* **Kanban/Scrum Boards**: Visualize and manage work progress using boards tailored for Kanban or Scrum methodologies.
* **Collaboration**: Discuss tasks with your team, attach relevant files, and tag members for better communication.

When creating a new project in Azure Boards, you can choose from four work item processes:

1. **Agile**: Ideal for teams using Agile methodologies, featuring items like Bug, Epic, Feature, Issue, Task, and User Story.
2. **Basic**: A simple, straightforward process with work items such as Epic, Issue, and Task.
3. **CMMI**: A process designed for projects that require a detailed approach, like those using the Capability Maturity Model Integration.
4. **Scrum**: A process built for Scrum teams, which includes work items like Bug, Epic, Feature, Impediment, Product Backlog Item, and Task. The Scrum methodology focuses on breaking down the overall scope into manageable work increments.

**Azure Repos**

Azure Repos is a version control system that supports team collaboration, enabling efficient code management. It offers features such as:

* **Version Control**: Supports both Git and Team Foundation Version Control (TFVC) to manage your code.
* **Collaboration**: Facilitate teamwork through pull requests, code reviews, and change tracking.
* **Branching**: Easily create branches to work on different features or fixes without affecting the main codebase.

**Azure Pipelines**

Azure Pipelines is an automation service for building, testing, and deploying code. It supports:

* **Continuous Integration (CI)**: Automates code building and testing whenever a team member commits changes.
* **Continuous Delivery (CD)**: Deploys applications across various environments (e.g., production, staging) with minimal human intervention.
* **Cross-Platform Support**: Works with multiple languages and platforms like .NET, Java, Node.js, and more.

**Azure Test Plans**

Azure Test Plans provides a comprehensive test management solution to ensure high-quality releases. It includes:

* **Test Case Management**: Create and organize test cases, associate them with requirements, and execute test runs.
* **Test Suites**: Manage related test cases as a suite for better organization.
* **Automated Testing**: Perform automated tests and track results efficiently.

**Azure Artifacts**

Azure Artifacts is a package management service that helps you share and manage packages. It supports:

* **Multiple Package Types**: Works with popular package types like NuGet, Maven, npm, and more.
* **Dependency Management**: Simplifies the management of dependencies within your projects.

These Azure DevOps services collectively enhance software development and project management, promoting efficiency, collaboration, and streamlined deployment.

Azure Boards:

Advantages of work item hierarchy:

1.Project structure

2.Task management

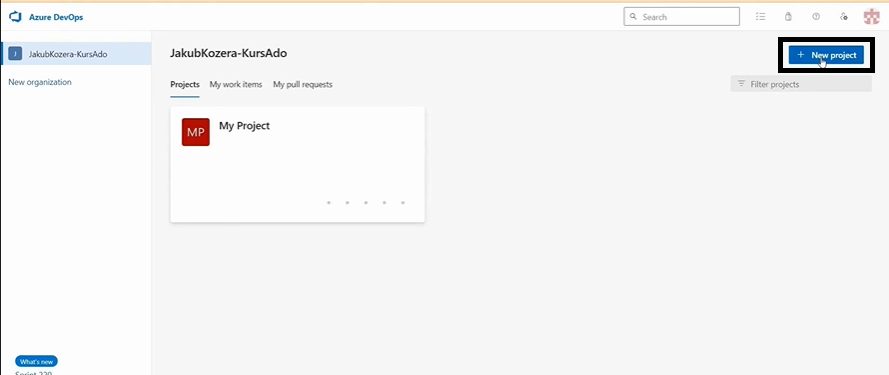
3.Planning and reporting

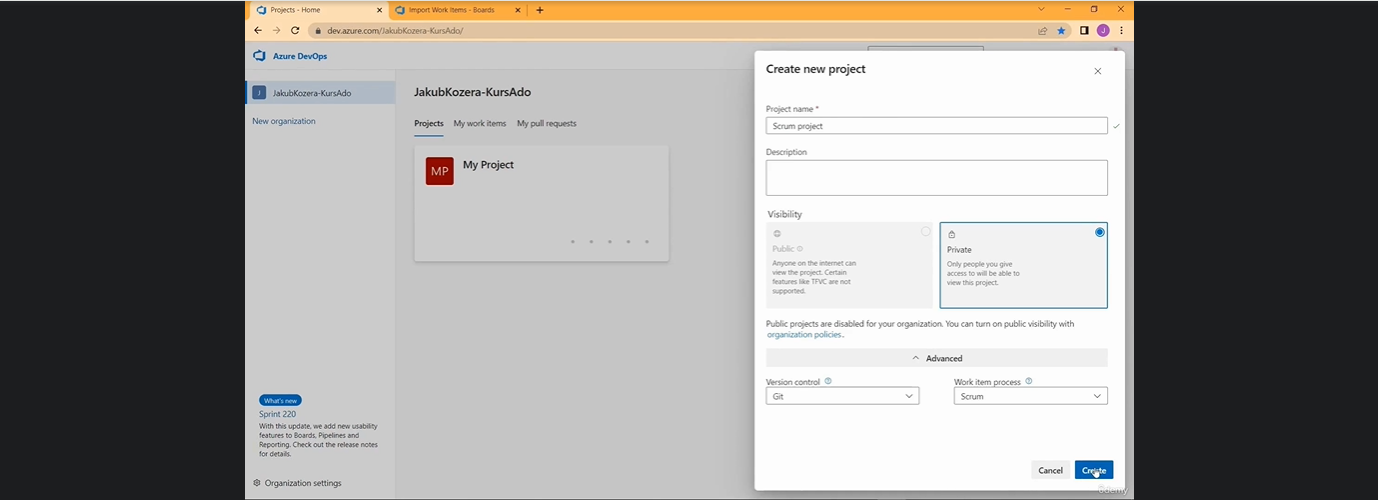
4.Dependency management

5.Analysis control

**Create work items in the Azure boards:**

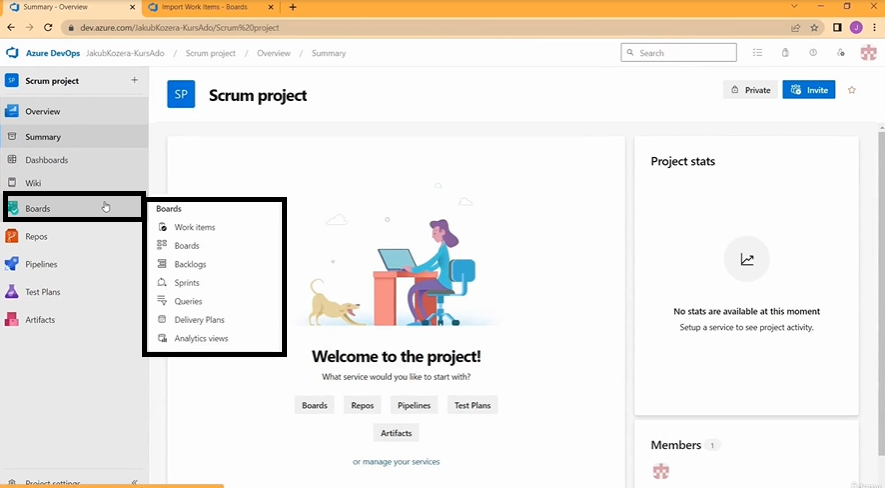
Go to Azure board and create the new project

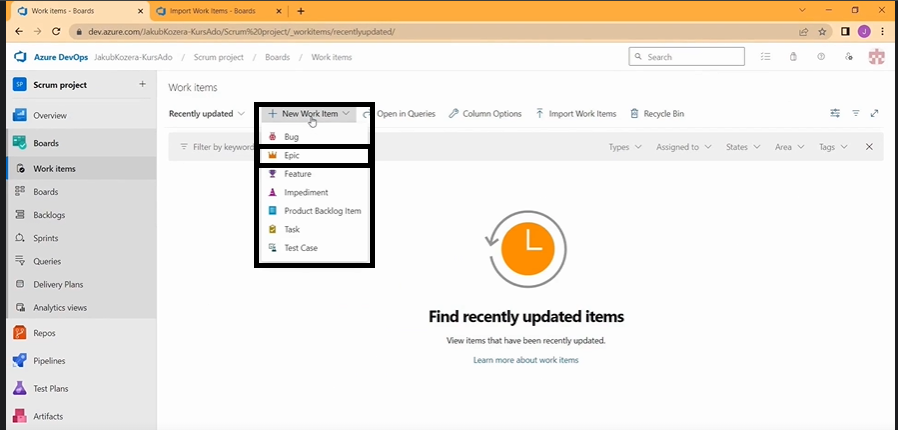




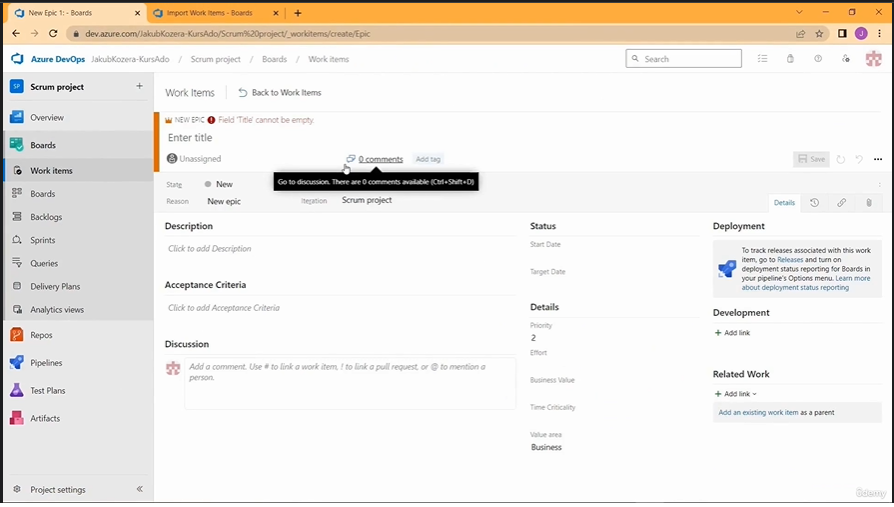
Select the work items as “Scrum”

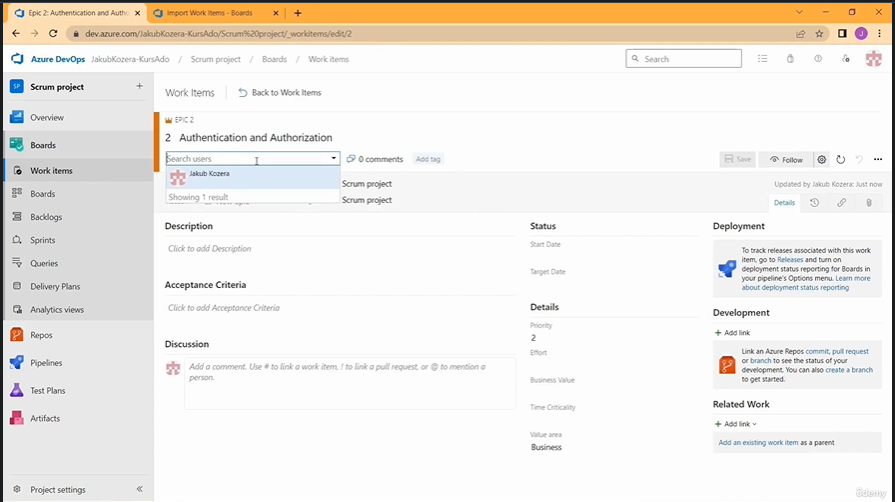
Once projects created then click the “Board” and select the work items

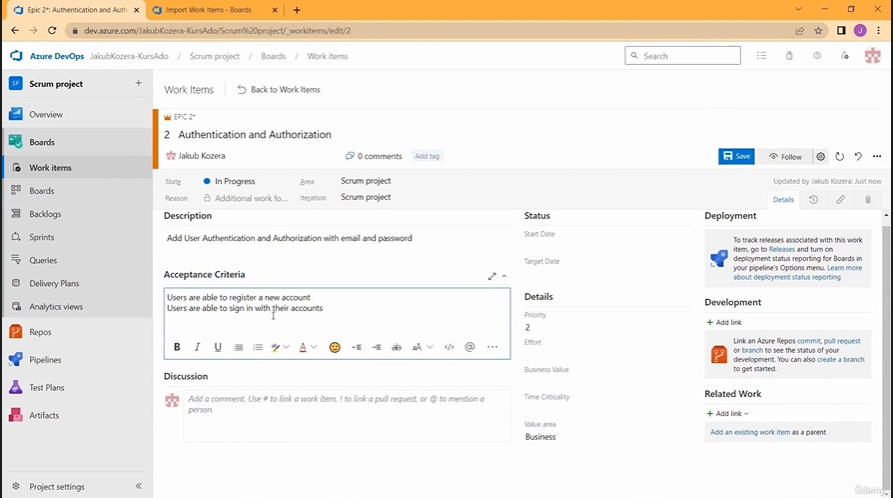
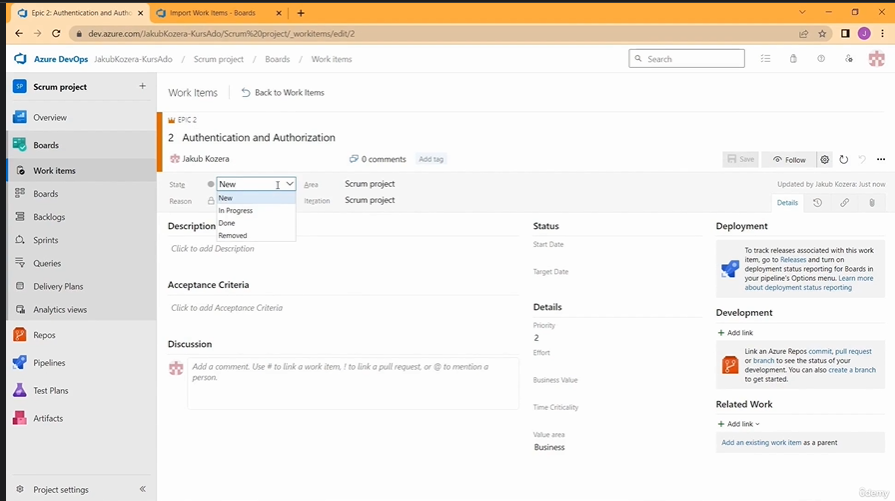


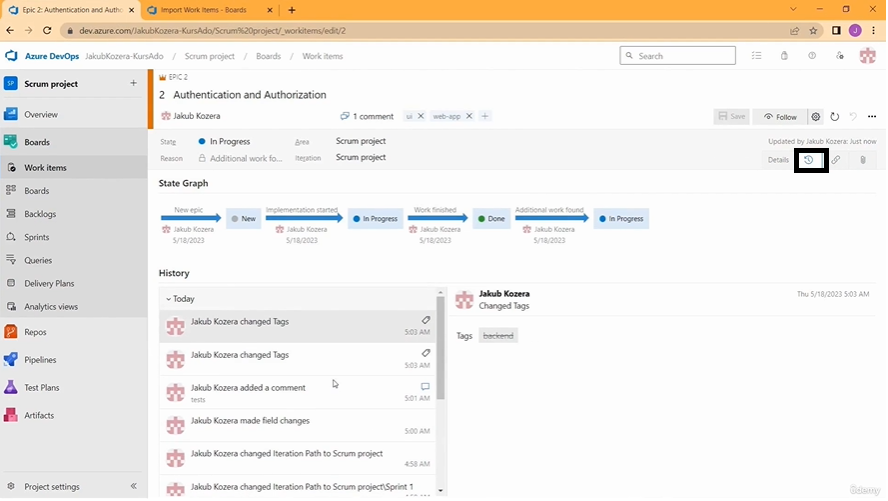


**Epic** is a high-level work item that represents a large body of work that can be broken down into smaller, more manageable pieces. It typically encapsulates a significant feature or a major requirement that needs to be completed over multiple sprints or iterations.









See the history EPIC items

**A product backlog item (PBI)** in Azure DevOps is a single unit of work that's part of a project's product backlog. PBIs can be used to track work, and can include: user stories, epics, specifications, bugs, and change requirements.

 PBIs are defined by the team and are considered for their business value, effort, and relevancy. They are then prioritized for development sprints.

 A product backlog is a project plan that shows what a team intends to deliver. It's an interactive list of work items that corresponds to a team's project plan or roadmap. The backlog can be customized by a team, and can be used to:

* Prioritize work
* Forecast work by sprints
* Link work to portfolio backlog items

Azure Repo’s:

Version control :

It allows you to create repositories , track the change history , create branches, compare versions and make corrections. That’s all for GIT and TFVC repository.

Team work:

Facilitates collaboration between team members with capabilities such as pull request, comments and discussions, developer can effectively collaborate and revise code. The ability to assign roles and access also allow you to control who has access to the repository and what changes they can make

Integration with pipelines:

Azure repos integrates with continuous integration and continuous deployment tools via azure pipelines. You can configure automatic process for compiling, testing and deploying application to various environment.

Benefits of Azure repo’s:

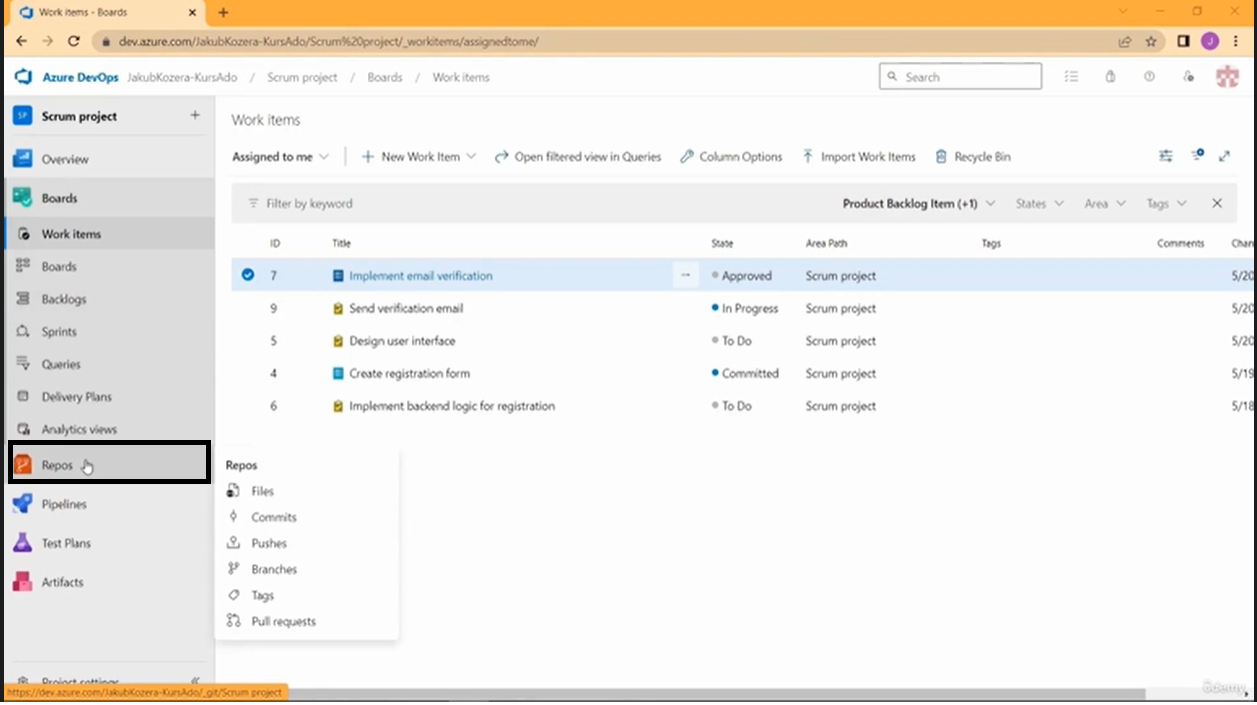
1.Possiblity to create multiple repository for single project

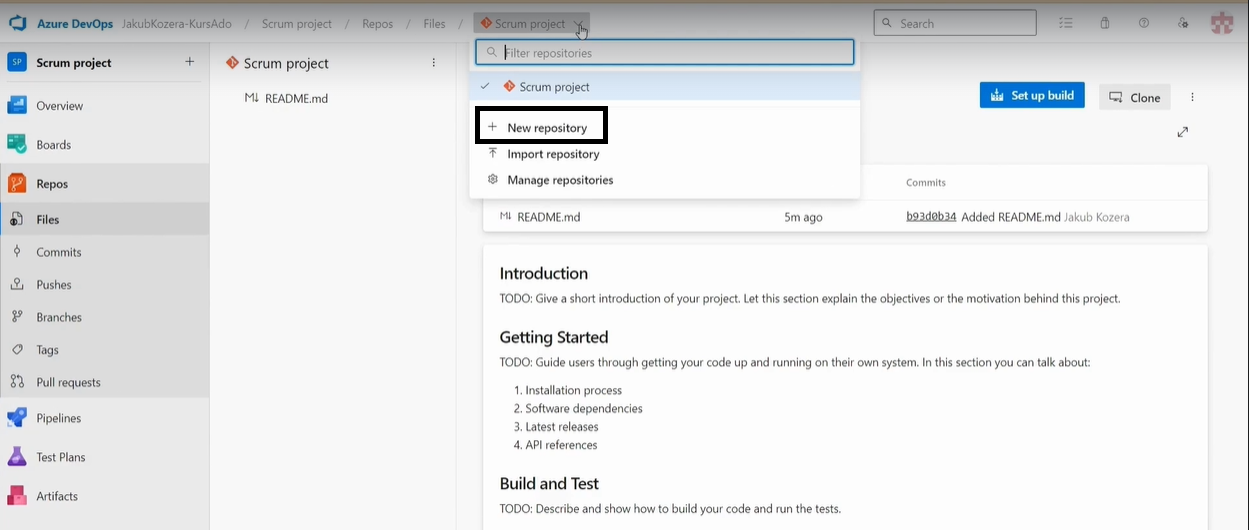
2.Simple import from external sources such as git hub

3.pull request discussion

4.binding work items to specific branches

5.setting policies for branches

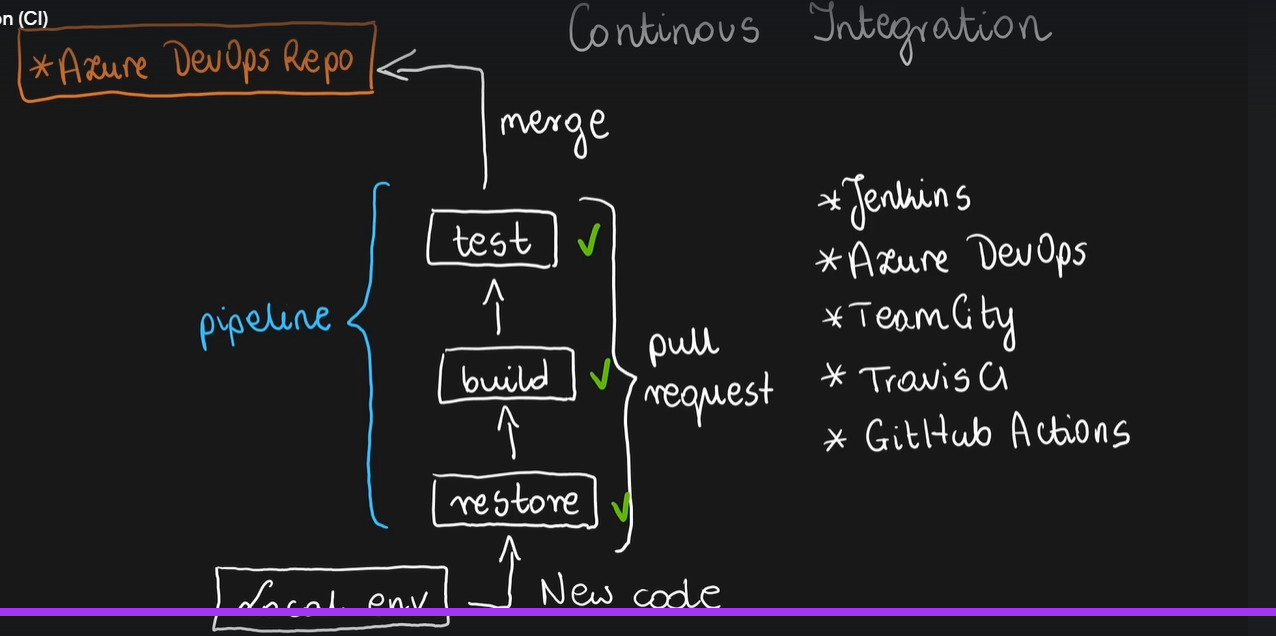




Pipeline:

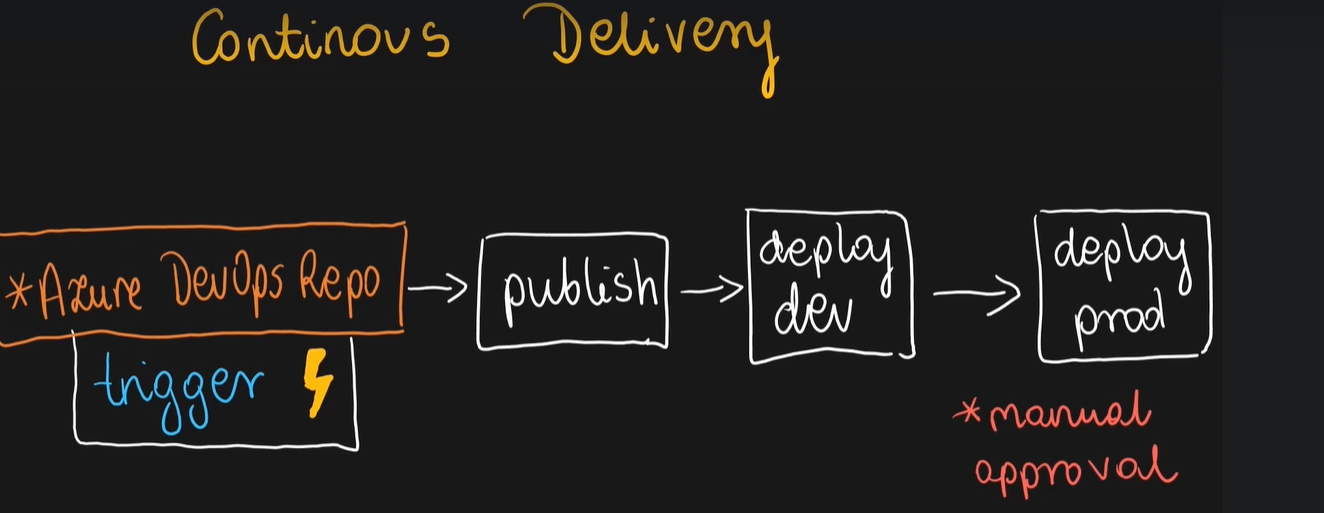
Continuous Integration:

Continuous integration (CI) is a software development practice that automates the process of merging code changes into a shared repository and running tests on the code. The goal of CI is to improve software quality, find and fix bugs faster, and reduce the time it takes to release new updates

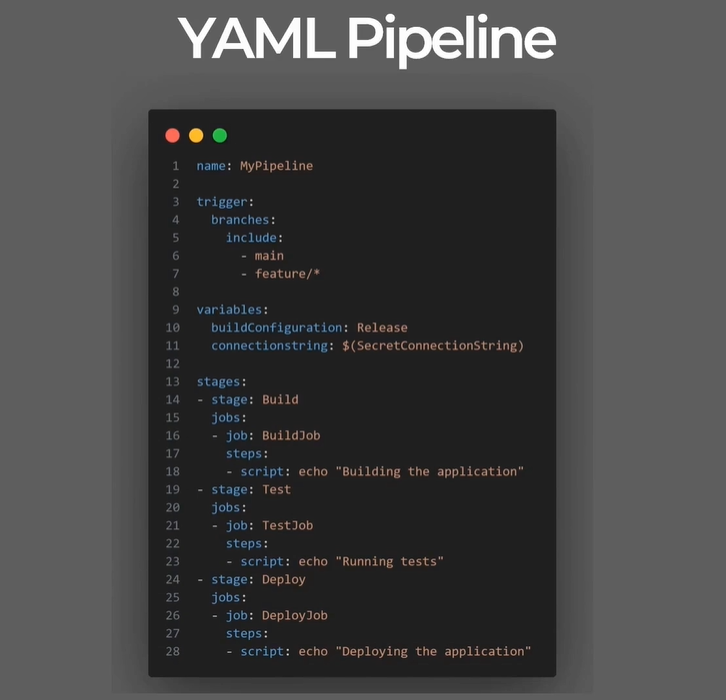


Continous delivery and deployment:

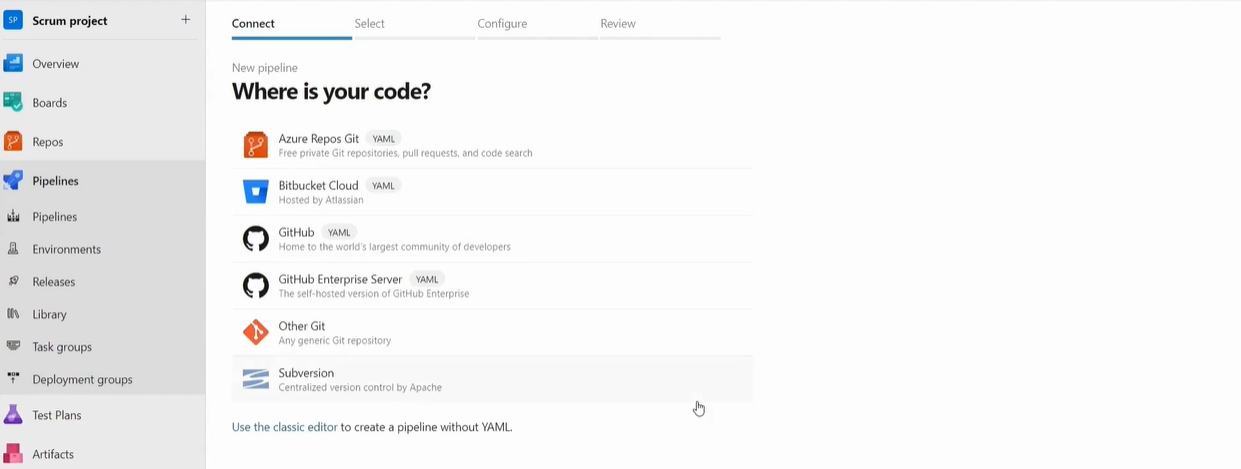
CD builds on continuous integration by automatically deploying code changes to a testing or production environment after the build stage



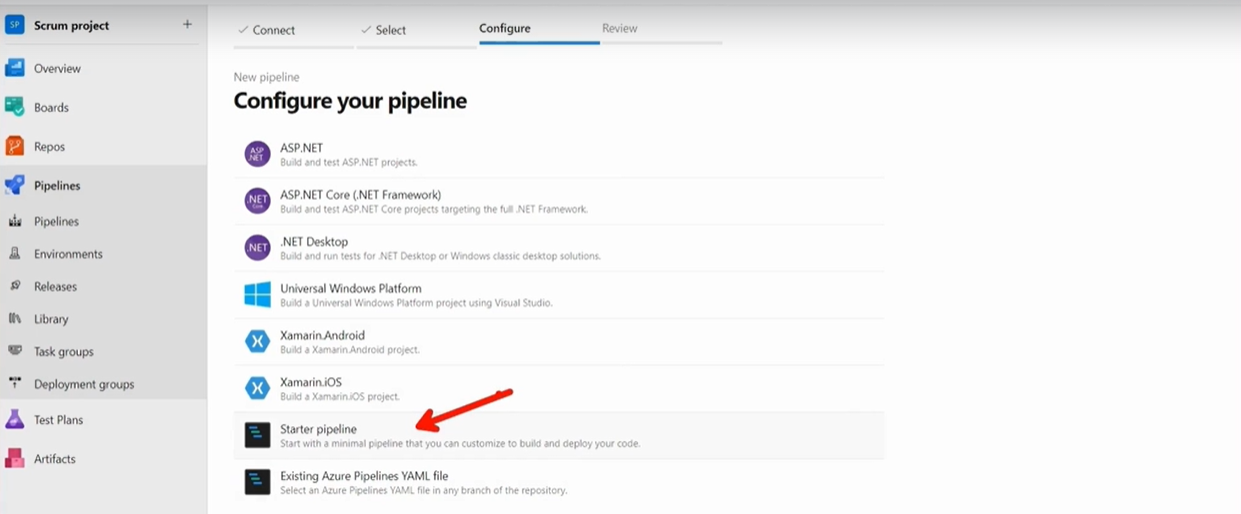
Pipeline:

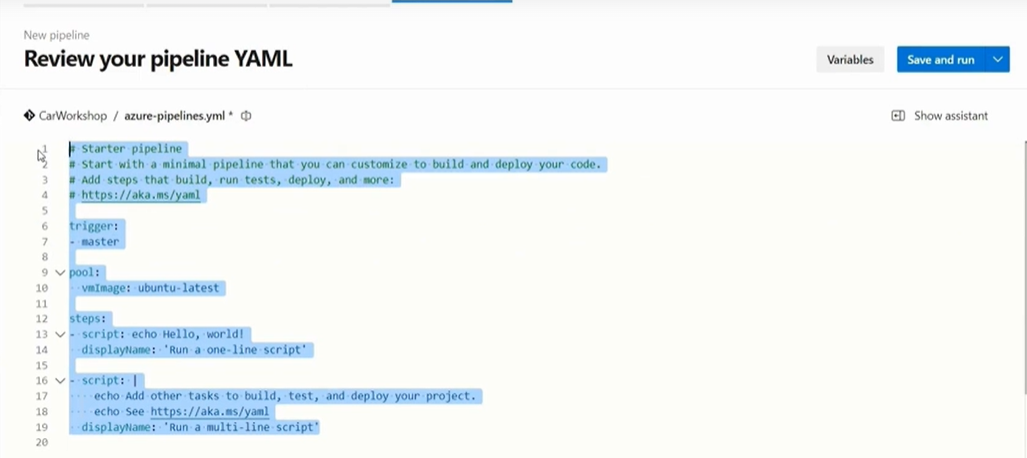


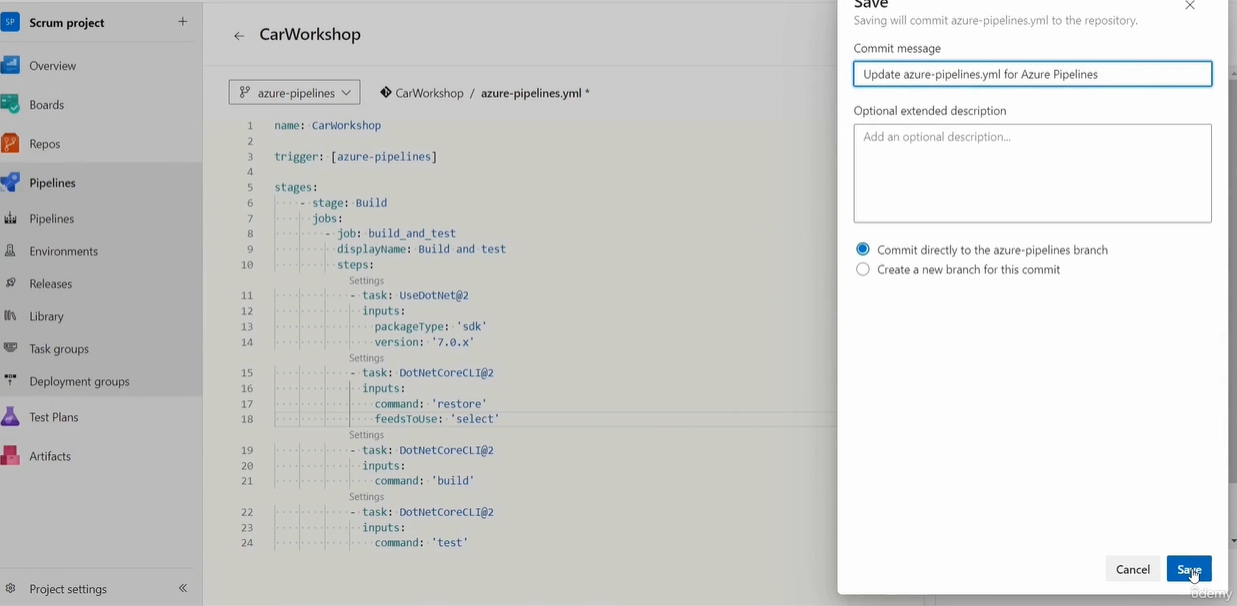


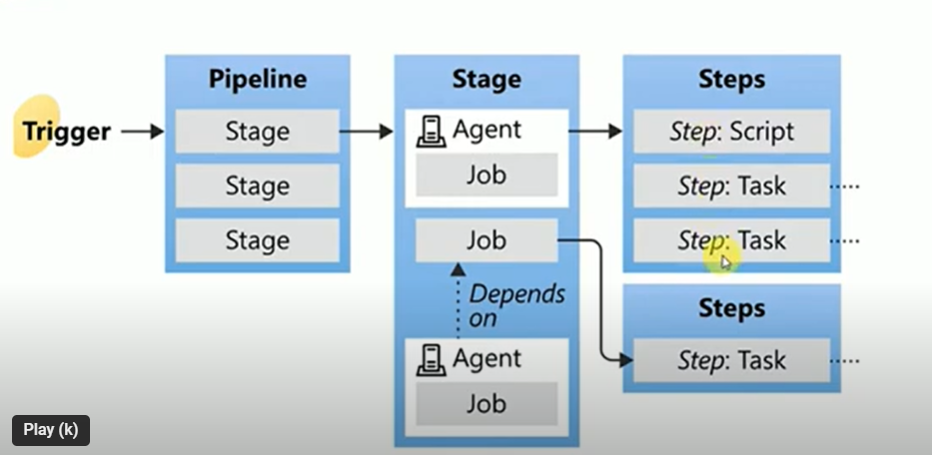


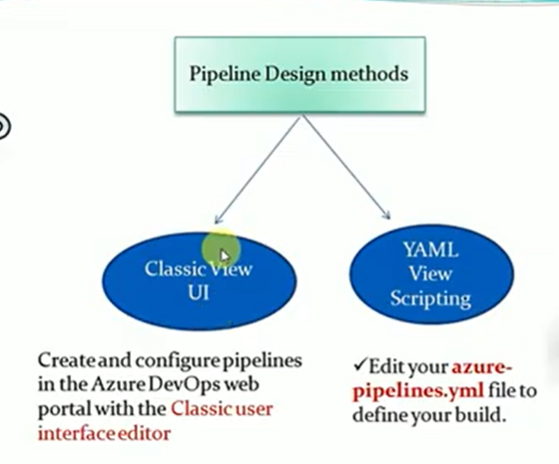
Select the Azure Repos GIT

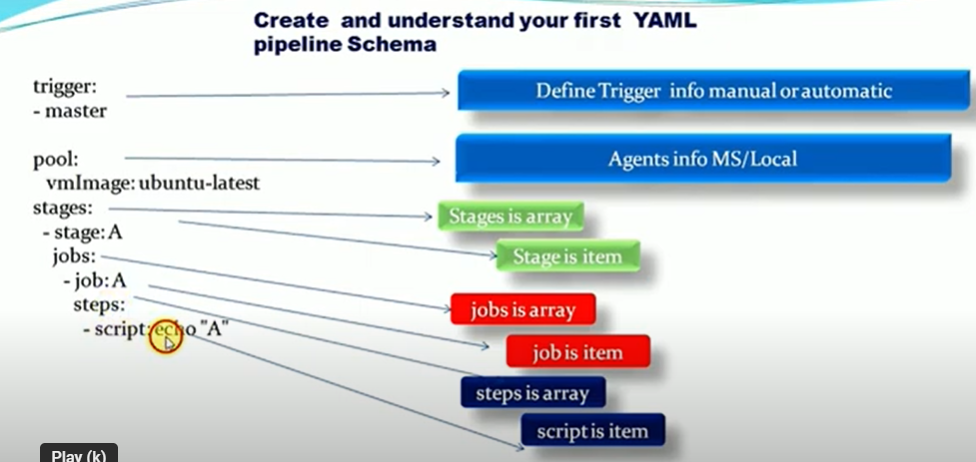












Stages are the major division in a pipeline: "build","run" and "deploy" to pre-production are good examples of stages

every pipeline has at least one stage even if you do not 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



Samples pipeline code for reference:

# azure-pipelines.yml

trigger:

branches:

include:

- main

stages:

# Stage 1: Build

- stage: Build

displayName: "Build Stage"

pool:

vmImage: 'ubuntu-latest' # Pool for this stage

jobs:

- job: BuildJob

displayName: "Building the code"

steps:

- script: |

echo "Building the code..."

# Add your build steps here

displayName: "Run Build Script"

# Stage 2: Test

- stage: Test

displayName: "Test Stage"

dependsOn: Build

condition: succeeded() # This stage runs only if the Build stage succeeds

pool:

vmImage: 'windows-latest' # Pool for this stage

jobs:

- job: TestJob

displayName: "Running Tests"

steps:

- script: |

echo "Running tests..."

# Add your test steps here

displayName: "Run Test Script"

# Stage 3: Deploy

- stage: Deploy

displayName: "Deploy Stage"

dependsOn: Test

condition: and(succeeded(), eq(variables['Build.SourceBranch'], 'refs/heads/main')) # Run only if tests succeed and on main branch

pool:

vmImage: 'ubuntu-latest' # Pool for this stage

jobs:

- job: DeployJob

displayName: "Deploying to Environment"

steps:

- script: |

echo "Deploying the application..."

# Add your deployment steps here

displayName: "Run Deploy Script"

Maven Projects sample pipeline

Setting up a **Maven project in an Azure Pipeline** involves defining a YAML file for your pipeline that specifies how to build, test, and package your Maven project. Here’s a basic example of an Azure Pipeline for a Maven project:

**Step-by-Step Guide for Azure Pipeline YAML for a Maven Project**

yaml

Copy code

# azure-pipelines.yml

trigger:

branches:

include:

- main # The branch that triggers the build

pool:

vmImage: 'ubuntu-latest' # Choose the agent to run the pipeline

stages:

- stage: Build

displayName: "Build Stage"

jobs:

- job: BuildJob

displayName: "Building Maven Project"

steps:

# Step 1: Checkout the code

- task: Checkout@1

displayName: "Checkout Code"

# Step 2: Set up Maven

- task: Maven@3

displayName: "Set up Maven and Build"

inputs:

mavenPomFile: 'pom.xml' # Path to your pom.xml file

goals: 'clean package' # Maven goals (clean, compile, package, etc.)

options: '-B' # Run Maven in batch mode (no interactive input)

javaHomeOption: 'JDKVersion'

jdkVersionOption: '1.8' # Specify the JDK version, e.g., 1.8 or 11

publishJUnitResults: true # Publish JUnit test results if available

testResultsFiles: '\*\*/target/surefire-reports/\*.xml' # Test results location

- stage: Test

displayName: "Test Stage"

dependsOn: Build

condition: succeeded()

jobs:

- job: TestJob

displayName: "Running Tests"

steps:

# Step 3: Run additional tests if needed

- script: |

echo "Running additional tests..."

# Add commands to run custom tests

displayName: "Run Custom Tests"

- stage: Deploy

displayName: "Deploy Stage"

dependsOn: Test

condition: succeeded()

jobs:

- job: DeployJob

displayName: "Deploy Maven Project"

steps:

# Step 4: Deploy your application or package

- script: |

echo "Deploying application..."

# Add deployment steps, such as copying files or running deployment scripts

displayName: "Run Deployment Script"

**Explanation of the YAML File**

1. **trigger**: Specifies which branches trigger the pipeline. In this case, it is set to main.
2. **pool**: Defines the virtual machine (VM) image to use for the pipeline. Here, ubuntu-latest is used.
3. **stages**: Breaks down the pipeline into multiple stages (e.g., Build, Test, Deploy).
4. **jobs**: Specifies the jobs within each stage.
5. **steps**: Defines the tasks and scripts to run:
   * **Checkout Code**: Uses the Checkout@1 task to pull the code from the repository.
   * **Set up Maven and Build**: Uses the Maven@3 task to run Maven commands such as clean package.
   * **Run Tests**: A custom script to run additional tests if needed.
   * **Deploy**: A script to deploy the application or run deployment-related tasks.

**Customization**

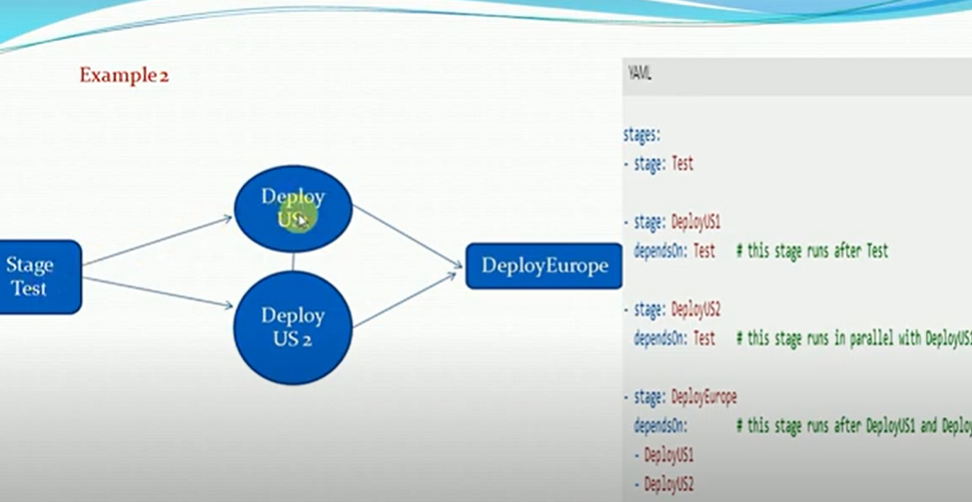
* **Maven Goals**: You can change the Maven goals in the goals field to suit your project needs (e.g., clean compile test).
* **JDK Version**: Adjust the jdkVersionOption to match the Java version your project requires.
* **Test Results**: Ensure the testResultsFiles path is correct for your project's test results.

**Additional Tasks**

* **Code Quality Tools**: You can add tasks for code quality checks, such as SonarQube.
* **Artifact Publishing**: Use the PublishBuildArtifacts@1 task if you want to publish artifacts like JAR or WAR files.

This YAML file provides a basic setup for building, testing, and deploying a Maven project in an Azure Pipeline. You can extend it further based on your project’s requirements.

**Parallel execution of depondsOn the source stage**



Add condition and dependency between stages:

customize the behavious by forcing a stage to run even if a previous stage fails or by specfiying a custom condition

we can specify the conditions under which each stage runs

stages:

- stage: A

jobs:

- job: JobA

**#Stage B run if stage A failed**

- stage: B

depondsOn: A

condition: failed()

**#stage C run if stage B succeeds**

- stage: C

depondsOn:

- A

- B

condition: succeeded('B')

**Job and types of jobs**

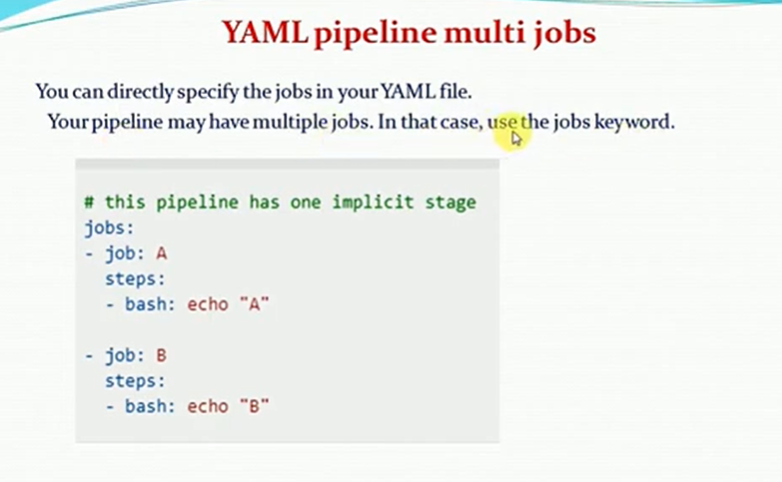
Job is a series of steps that run sequentially as a unit. In other words, a job is the smallest unit of wok that can scheduled to run

Job can be of different types,deponding on where they run

Agentpool jobs - run on an agent in an agent pool

Server jobs - run of the AzureDevops Server

containerjobs - run in a container on agent in an agent pool



pool:

vmImage: ubuntu-latest # Specifies the pool with a virtual machine image to run the job

stages:

- stage: A **# Defines a stage named "A"**

jobs:

- job: JobA **# Defines a job named "JobA"**

timeoutInMinutes: 5 **# The maximum time in minutes the job will run before timing out**

continueOnError: true **# The job will continue even if it encounters an error**

displayName: "this is single demo" **# Display name for the job**

cancelTimeoutInMinutes: 2 **# The time to wait in minutes to force-cancel the job if needed**

**Deployment Job:**

1.RunOnce Strategy

trigger: none # The pipeline will not trigger automatically; you can run it manually.

pool:

vmImage: ubuntu-latest # Specifies the agent pool with the Ubuntu virtual machine image.

stages:

- stage: deploy # Defines a stage named "deploy"

jobs:

- deployment: test # Specifies a deployment job named "test"

**environment: Test** # The deployment environment named "Test"

strategy:

runOnce: # Defines the deployment strategy to run the job once

deploy:

steps: # List of steps to execute in the deployment

- checkout: self # Checks out the source code from the repository

- script: echo "This is test" # Runs a script that echoes "This is test"

2.Example with Rolling Deployment Strategy

A rolling deployment gradually updates a subset of instances at a time. This allows you to test the deployment in a controlled way before fully rolling it out.

yaml

trigger: none

pool:

vmImage: ubuntu-latest

stages:

- stage: deploy

jobs:

- deployment: rollingDeploy

**environment: QA**

strategy:

rolling:

maxParallel: 2 # The maximum number of parallel deployments

deploy:

steps:

- checkout: self

- script: echo "Starting rolling deployment..."

- script: echo "Deploying to a subset of instances"

- script: echo "Rolling deployment completed"

Explanation of the Rolling Deployment YAML

maxParallel: Specifies the maximum number of instances to deploy in parallel. You can adjust this number based on the size of your environment.

steps: Defines the actions for the rolling deployment, which are executed for each subset of instances.

3.Example with Canary Deployment Strategy

A canary deployment releases the new version to a small subset of users or instances. If the deployment is successful, it gradually rolls out to more instances.

yaml

trigger: none

pool:

vmImage: ubuntu-latest

stages:

- stage: deploy

jobs:

- deployment: canaryDeploy

**environment: Prod**

strategy:

canary:

increments: 2 # Number of increments for deployment

deploy:

steps:

- checkout: self

- script: echo "Starting canary deployment..."

- script: echo "Deploying to the first set of instances"

- script: echo "Monitoring for any issues"

- script: echo "Canary deployment completed"

Explanation of the Canary Deployment YAML

increments: Specifies the number of increments in which the deployment will be rolled out. You can adjust this to control the rollout speed.

steps: The deployment steps are executed in increments, allowing you to monitor and validate the deployment before proceeding to the next set of instances.

**1. Build Tasks**

Build tasks are used to compile code, generate binaries, or prepare your application for testing and deployment.

* **Maven@3**: Builds a Java project using Maven.
* **DotNetCoreCLI@2**: Builds a .NET Core project.
* **Gradle@2**: Builds a Java project using Gradle.
* **Npm@1**: Runs npm commands, such as install or build, for JavaScript projects.
* **MSBuild@1**: Builds a Visual Studio project.

**2. Utility Tasks**

Utility tasks help with operations such as copying files, publishing artifacts, setting variables, and more.

* **CopyFiles@2**: Copies files from one location to another on the agent or to a build artifact staging directory.
* **PublishBuildArtifacts@1**: Publishes build artifacts for later stages, such as deployment.
* **ArchiveFiles@2**: Archives files into a .zip file.
* **DeleteFiles@1**: Deletes files from a specified location on the agent.
* **CmdLine@2**: Runs a command line script on the agent.
* **Bash@3**: Runs a Bash script on a Linux agent.
* **PowerShell@2**: Runs a PowerShell script on the agent.

**3. Test Tasks**

Test tasks are used to run tests, collect test results, and publish the results for analysis.

* **VsTest@2**: Runs tests using Visual Studio Test.
* **DotNetCoreCLI@2 (test)**: Runs .NET Core tests.
* **JUnit@1**: Publishes JUnit test results.
* **PublishTestResults@2**: Publishes test results to Azure DevOps for reporting.
* **XcodeTest@2**: Runs tests on an iOS/macOS application built with Xcode.

**4. Package Tasks**

Package tasks are used to package the built application into a format suitable for deployment, such as .zip files, Docker images, or NuGet packages.

* **NuGetCommand@2**: Restores, packs, or pushes NuGet packages.
* **Docker@2 (build)**: Builds a Docker image.
* **DotNetCoreCLI@2 (pack)**: Packs a .NET Core project into a NuGet package.
* **ArchiveFiles@2**: Packages files into an archive format, like .zip.
* **Tar@1**: Creates a tarball archive for Linux-based deployments.

**5. Deploy Tasks**

Deploy tasks are used to release your application to various environments, such as Azure, AWS, on-premises servers, or Kubernetes clusters.

* **AzureWebApp@1**: Deploys a web app to Azure App Service.
* **KubernetesManifest@0**: Deploys to a Kubernetes cluster using manifest files.
* **AzureResourceManagerTemplateDeployment@3**: Deploys Azure resources using ARM templates.
* **AzureVMSS@1**: Deploys updates to a Virtual Machine Scale Set in Azure.
* **AzureFunctionApp@1**: Deploys an Azure Function App.
* **Docker@2 (push)**: Pushes a Docker image to a container registry.

**6. Tool Tasks**

Tool tasks are used to install, configure, or use various tools needed in the pipeline.

* **UseDotNet@2**: Installs a specified version of the .NET Core SDK.
* **NodeTool@0**: Installs a specific version of Node.js.
* **JavaToolInstaller@0**: Installs a specified version of Java.
* **GitToolInstaller@0**: Installs Git on the agent.
* **PythonScript@0**: Runs a Python script on the agent.
* **DockerInstaller@0**: Installs Docker on the agent if it is not already available.

Reference code :

trigger: none

pool:

vmImage: ubuntu-latest

stages:

- stage: Build

jobs:

- job: BuildJob

steps:

- checkout: self

- task: Maven@3 # Build task

inputs:

mavenPomFile: 'pom.xml'

goals: 'clean install'

- task: ArchiveFiles@2 # Utility task

inputs:

rootFolderOrFile: '$(Build.SourcesDirectory)'

includeRootFolder: false

archiveType: 'zip'

archiveFile: '$(Build.ArtifactStagingDirectory)/output.zip'

- stage: Test

jobs:

- job: TestJob

steps:

- task: Maven@3 # Test task

inputs:

mavenPomFile: 'pom.xml'

goals: 'test'

- task: PublishTestResults@2 # Test task

inputs:

testResultsFiles: '\*\*/surefire-reports/\*.xml'

- stage: Package

jobs:

- job: PackageJob

steps:

- task: Docker@2 # Package task

inputs:

command: 'build'

dockerfile: 'Dockerfile'

tags: 'latest'

- task: Docker@2 # Package task

inputs:

command: 'push'

tags: 'latest'

- stage: Deploy

jobs:

- deployment: DeployJob

environment: Production

strategy:

runOnce:

deploy:

steps:

- task: AzureWebApp@1 # Deploy task

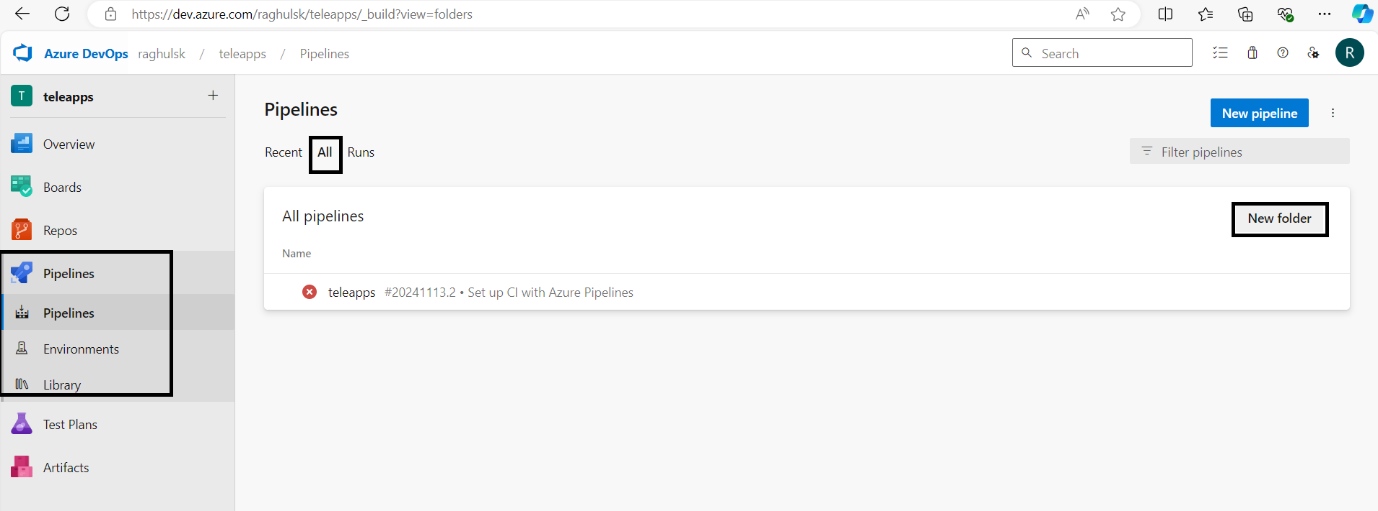
inputs:

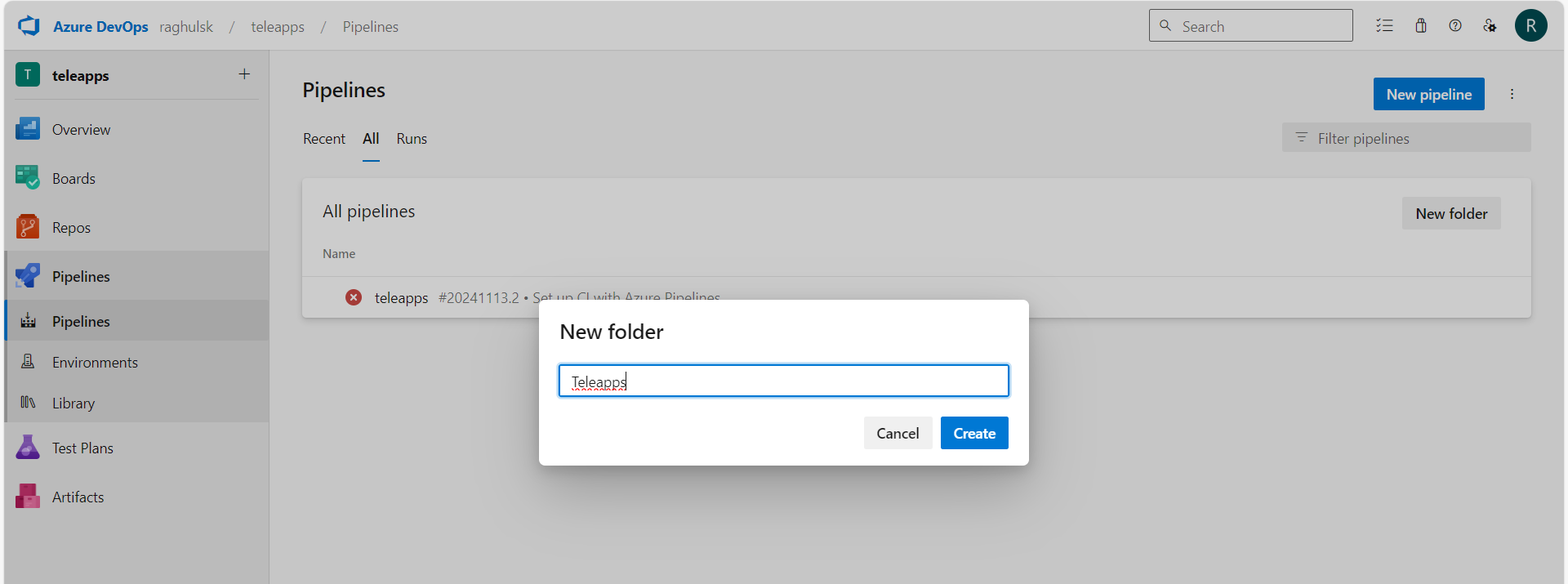
azureSubscription: 'AzureServiceConnection'

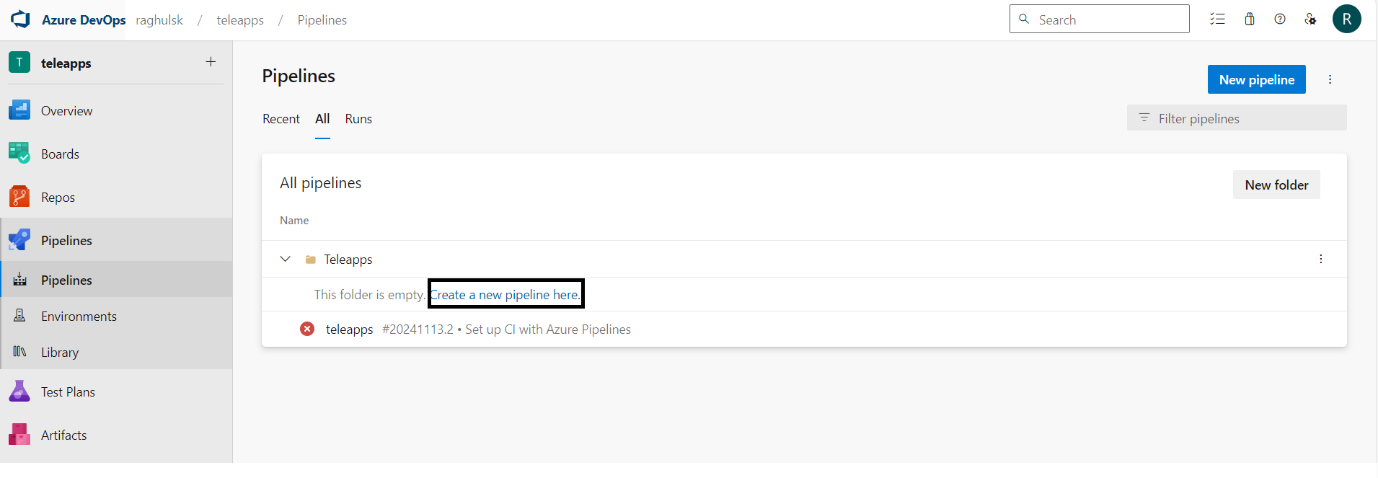
appName: 'MyWebApp'

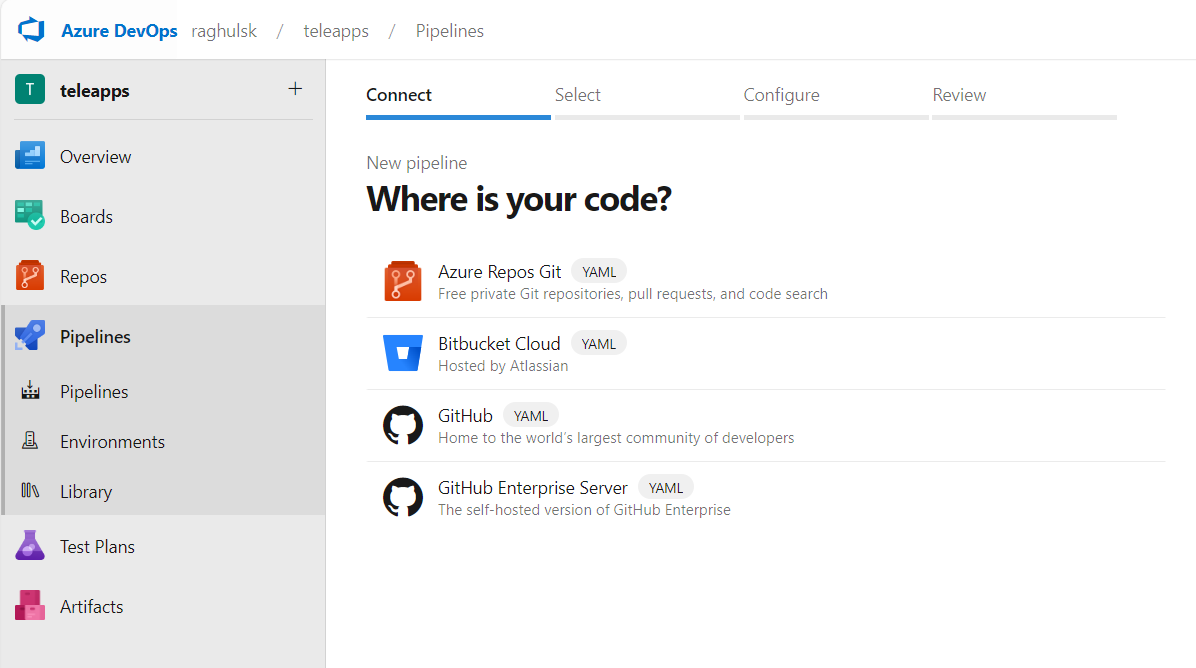
package: '$(Build.ArtifactStagingDirectory)/output.zip'

**Create the separate folder for pipelines**









Azure artifacts:

**Azure Artifacts** is a service within Azure DevOps that provides a package management solution. It allows teams to create and share Maven, npm, NuGet, Python, and Universal packages from both public and private sources. Azure Artifacts simplifies the process of integrating package management into your CI/CD pipelines and makes it easy to manage dependencies for your projects

Azure artifacts use case:

**Azure Artifacts Use Cases: Package Repository, Version Control, and Access Control**

Azure Artifacts is a robust package management solution that integrates with Azure DevOps, providing features for storing, managing, and distributing packages across teams and projects. Here are three key use cases:

**1. Package Repository**

* **Scenario**: Storing and sharing internal packages across different teams or projects in a secure and scalable manner.
* **How Azure Artifacts Helps**:
  + **Private Feeds**: Azure Artifacts allows you to create private feeds to host your packages (e.g., **NuGet**, **npm**, **Maven**, **Python**, **Universal Packages**) and make them available to other teams within the organization.
  + **Centralized Package Management**: You can centralize your shared components, libraries, and tools in Azure Artifacts to avoid duplicating effort and reduce versioning issues.
  + **Upstream Sources**: Azure Artifacts can be configured to pull from public sources (e.g., npm registry, Maven Central) and cache them locally for use in your organization’s build and deployment pipelines

2. **Version Control**

* **Scenario**: Managing versions of packages to ensure that developers always use the correct version, and allowing easy rollback to previous versions when needed.
* **How Azure Artifacts Helps**:
  + **Semantic Versioning**: Azure Artifacts supports semantic versioning, allowing you to define major, minor, and patch versions for your packages. This helps ensure that updates are predictable and backwards-compatible changes are easy to track.
  + **Automatic Versioning**: Packages uploaded to Azure Artifacts are automatically versioned, providing clear visibility into which versions of a package are available, and which one is currently in use.
  + **Versioning History**: You can see the history of each package, allowing you to view past versions and roll back to a previous stable version if necessary.

**Azure Artifacts Use Cases: Package Repository, Version Control, and Access Control**

Azure Artifacts is a robust package management solution that integrates with Azure DevOps, providing features for storing, managing, and distributing packages across teams and projects. Here are three key use cases:

**1. Package Repository**

* **Scenario**: Storing and sharing internal packages across different teams or projects in a secure and scalable manner.
* **How Azure Artifacts Helps**:
  + **Private Feeds**: Azure Artifacts allows you to create private feeds to host your packages (e.g., **NuGet**, **npm**, **Maven**, **Python**, **Universal Packages**) and make them available to other teams within the organization.
  + **Centralized Package Management**: You can centralize your shared components, libraries, and tools in Azure Artifacts to avoid duplicating effort and reduce versioning issues.
  + **Upstream Sources**: Azure Artifacts can be configured to pull from public sources (e.g., npm registry, Maven Central) and cache them locally for use in your organization’s build and deployment pipelines.
* **Example**:
  + A development team creates a custom utility library in Java that is used across multiple projects. The library is stored as a Maven package in Azure Artifacts, and other teams can simply reference it in their projects, ensuring they always use the latest version without needing to manually manage dependencies.

**2. Version Control**

* **Scenario**: Managing versions of packages to ensure that developers always use the correct version, and allowing easy rollback to previous versions when needed.
* **How Azure Artifacts Helps**:
  + **Semantic Versioning**: Azure Artifacts supports semantic versioning, allowing you to define major, minor, and patch versions for your packages. This helps ensure that updates are predictable and backwards-compatible changes are easy to track.
  + **Automatic Versioning**: Packages uploaded to Azure Artifacts are automatically versioned, providing clear visibility into which versions of a package are available, and which one is currently in use.
  + **Versioning History**: You can see the history of each package, allowing you to view past versions and roll back to a previous stable version if necessary.
* **Example**:
  + A team working on a complex web application releases updates to their API client library. With Azure Artifacts, they can maintain multiple versions of the library, and developers can decide whether to update to the latest version or stick with a specific version, ensuring compatibility with other dependencies in their projects.

**3. Access Control**

* **Scenario**: Controlling access to packages to ensure that only authorized users or teams can publish, download, or modify specific packages.
* **How Azure Artifacts Helps**:
  + **Role-Based Access Control (RBAC)**: Azure Artifacts allows you to manage permissions using Azure DevOps’ RBAC model. You can define who has access to which packages, restricting access to sensitive or proprietary packages to authorized users only.
  + **Fine-Grained Permissions**: You can configure permissions at the feed level, allowing teams to share packages with specific members or across specific projects without exposing them to the entire organization.
  + **Auditing and Compliance**: Track who has accessed or published packages and maintain an audit trail of package usage to ensure compliance with security policies and regulatory requirements

**Azure artifacts supported package types:**

1. **NuGet** – For .NET applications.
2. **npm** – For JavaScript and Node.js applications.
3. **Maven** – For Java-based applications.
4. **Python** – For Python-based applications.
5. **Universal Packages** – For non-specific packages like large binaries or custom tools.
6. **Docker** – For Docker container images.
7. **Gradle** – For Java-based applications, typically Android or JVM projects.
8. **Composer** – For PHP applications.
9. **Ruby Gems** – For Ruby-based applications.
10. **Bower** – For front-end libraries (legacy use).
11. **Helm Charts** – For Kubernetes containerized applications.
12. **TFVC** – For version control (not a package type but supported for source code management).

These supported package types enable a wide variety of tools, languages, and frameworks to be integrated into the Azure DevOps pipeline, providing a comprehensive solution for dependency management and distribution.

**Azure Test plan:**

**Azure test plan**

Azure Test Plans is a comprehensive test management solution within Azure DevOps that allows teams to manage, execute, and track the progress of their software testing efforts. It supports a variety of testing methodologies, including manual testing, exploratory testing, and continuous testing in automated workflows.

1.Manual and exploratory testing

1.Planned manual testing

2.User Acceptance testing

3.Exploratory testing

2.automated testing

1.CI/CD Pipeline integration

Execute tests

Publish results

Analyze results

3.traceablity

1.User stories

4.Reporting and analysis