.Net Web API CI/CD Pipeline Implementation in Azure DevOps

This repository contains a simple .NET service that serves a single endpoint: `/count`. Each call to this endpoint increments a counter and returns the number of times the endpoint has been called.

.CountControllerAPI -> CountController.cs -- C# Source File

using Microsoft.AspNetCore.Mvc;

namespace CountAPI.Controllers

{

    [ApiController]

    [Route("[controller]")]

    public class CountController : ControllerBase

    {

        private static int count = 0;

        [HttpGet]

        public IActionResult GetCount()

        {

            count++;

            return Ok(count);

        }

    }

}

For example I have created an app that fetches data from API. I have two pages [Home, MyCard]. Inside of Home page, I fetched a list of products from API. It has a button to add the product to the Card.  
Inside the MyCard page, I fetched a list of products that were added by the user. Inside each product list on the Card page, I added a **counter widget** as a component to increment the number of products.

**Step1: Created a New project in Azure DevOps**

**1.** Clicked on **New project**.

2. Entered a name and selected **Private.**

3. Create

**Step2: Pushed given code from local computer to Azure Repos through Git**

1. Right click on the given repository folder, open gitbash command prompt.
2. Initialize the repo folder with git init, it will initialize the repo and crates .git hidden folder.
3. git add.
4. git commit –m “MyFirstCommit”
5. git remote add origin <https://ramyamuppalla09/devopstask/dotnetrepo>

**Step3. Created a CI Pipeline (Build)**

1. Pipeline -> Created Pipeline
2. Choosen Azure Repos Git
3. Selected my repo
4. Starter pipeline or YAML file in repo
5. Replaced the YAML with a simple .Net pipeline

trigger:

  branches:

    include:

      - main

pool:

  vmImage: 'windows-latest'

steps:

- task: UseDotNet@2

  inputs:

    packageType: 'sdk'

    version: '7.0.x'

- script: dotnet test

  displayName: 'Run Unit Tests'

- script: dotnet build --configuration Release

  displayName: 'Build Application'

- script: dotnet publish --configuration Release --output $(Build.ArtifactStagingDirectory)

  displayName: 'Publish Application'

- task: PublishBuildArtifacts@1

  inputs:

    PathtoPublish: '$(Build.ArtifactStagingDirectory)'

    ArtifactName: 'drop'

**Step4: ADD CD (Release) Pipeline**

1. If deploying to Azure Web App

Set up Azure resources:

* Azure Sucscription: to host my web app.
* Resource Group: Created a resource group to organize my Azure resources.
* App Service Plan:
* Web App: Created a web app within the App Service Plan.

az webapp create –resource-group <rg-name> --plan <plan-name> --<app-name> -- runtime “DOTNETCORE | 7.0”

1. Create a release pipeline
2. Go to Pipelines -> Releases
3. Selected New Pipeline
4. Added Artifact: Selected my build pipeline
5. Added a stage (e.g. deploy to Dev)
6. Inside the stage, added a task

Task type: Azure App Service deploy

App type: web App on Windows

Set the App name and package path $(System.defaultWorkingDirectory)/ramyaapp/dotnet.zip)

Here are best practices for PR workflows, CI/CD triggers, and approval gates specifically in Azure DevOps:

Pull Request (PR) Workflows in Azure DevOps

Best Practices

1. Use Branch Policies:

* Enforce minimum number of reviewers.
* Require linked work items and successful builds before PR completion.
* Block PRs with merge conflicts or unresolved comments.

1. Use Auto-complete Feature:

* Enable auto-complete so PRs merge automatically once all conditions (reviews, builds, etc.) are satisfied.

1. Comment & Review Templates:

* Use PR templates to guide reviewers.
* Define clear checklist items (code coverage, design patterns, unit tests).

1. Limit PR Size:

* Encourage small, frequent PRs.
* Large PRs reduce review effectiveness and increase risk.

1. Status Checks:

* Include build validation pipelines and code analysis tools (e.g., SonarQube) as required checks.

CI Triggers (Continuous Integration) in Azure Pipelines Best Practices

1. Use YAML Pipelines:

* Source-controlled and reusable.
* Modularize using templates for shared logic.

1. Trigger on PR and Branch Updates:

yaml

trigger:

branches:

include:

- main

- develop

pr:

branches:

include:

- main

1. Pull Request Validation Pipelines:

* Create a separate pipeline specifically for validating PRs (lightweight, fast).

1. Test Strategy:

* Run unit tests, style checks, security scans.
* Use test impact analysis to only run affected tests.

1. Artifact Publishing:

* Publish build artifacts at the end of CI to reuse in CD.

yaml

- task: PublishBuildArtifacts@1

inputs:

pathToPublish: '$(Build.ArtifactStagingDirectory)'

artifactName: 'drop'

CD Triggers (Continuous Delivery) in Azure Pipelines Best Practices

1. Use Multi-Stage YAML Pipelines:

* Define stages for build, dev, qa, staging, and production.
* Clear visibility and consistent deployment logic.

1. Environment-Based Deployment:

yaml

stages:

- stage: DeployToDev

jobs:

- deployment: DevDeployment

environment: 'dev'

1. Use Runtime Parameters:

* Pass configuration values like feature flags, secrets, or environment-specific settings securely using variable groups or Azure Key Vault.

1. Automate Lower Environments, Gate Upper:

* Auto-deploy to dev and QA.
* Require manual approvals or checks for staging and production.

1. Rollback Strategy:

* Always retain and version artifacts.
* Enable redeploy of previous successful runs.

Approval Gates

1. Pre-Deployment Approvals:

* Use for staging, UAT, and production environments.
* Require sign-off from appropriate roles (Dev Lead, QA, Product Owner).

1. Post-Deployment Gates:

* Health checks, telemetry validation, or manual verification.
* Automate checks for performance metrics or error rates.

1. Environment-Specific Policies:

* Customize gates per environment (e.g., stricter rules for production).
* Use Azure DevOps environment approvals and checks for this.

1. Security & Compliance Checks:

* Include security scans (e.g., SAST/DAST).
* Enforce license checks or policy validation using tools like SonarQube, or Azure Policy.