Azure Pipelines

Index:

Terminology

1) Pipeline: It is a workflow that designs how the build, tests and deployments steps are run.

2) Agents: Agent is an installed software that is used to run a job one at a time. You need at least one agent to build or deploy your code. As the code increases, the number of agents used are going to increase proportionately.

3) Agent Pool: An agent pool is multiple number of agents working together. For Azure services, we have predefined agent pool that has Microsoft-hosted agents.

4) Stage: Stages are the imaginary divisions in a pipeline like “build this app", "run these tests", and "deploy to pre-production" . They are logical boundaries in your pipeline where you can pause the pipeline and perform various checks.

i. We can run arrange stages into dependency graph so that one job runs before another.

ii. There is a limit of 256 jobs for a stage.

5) Jobs: It is the smallest unit of work in pipelines. It is a series of steps that runs as a unit. A job can be agent less. A step that is run by jobs can either be a task or script.

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

7) **Triggers:** It is a set up that automatically runs the pipeline and decides when to run a pipeline. We can configure a trigger to run a pipeline after code is pushed to repository or schedule times to run the pipeline.

->Types of Triggers:

- **Pipeline Triggers**: Used in yaml file to run one pipeline after completion of other.
- **Build Completion** Trigger: Used in Classic text editor to run one pipeline after completion after the other.

- **Schedule Triggers**: The are independent of repository and runs a pipeline according to schedule.
- **Pull-Request**(PR) **Triggers**: It is triggered when a pull request is opened with target branch or when updates are made to pull request.
- **Continuous Integration(CI) Trigger:** It triggers whenever you push an update or specific tags to specific branches.
- **Gated check-in:** It is supported only for TFVC repositories.
- **Comment triggers** are supported only for GitHub repositories.
- **Continuous deployment triggers** help you start classic releases after a classic build or YAML pipeline completes.
- **Stage triggers** in classic release are used to configure how each stage in a classic release is triggered.

8) Environments: It is a collection of resources where you can deploy your applications. A pipeline might deploy the app to one or more environments after build is completed and tests are run. It contains one or more virtual machines, containers or web apps.

OpenPipelines->Environments->CreateNewEnvironments->EnterInformation->Create.

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

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

11) Artifact: Artifacts is a collection of files or packages published by run.

Introduction

Azure pipelines is a one of the **devops cloud service** that is used to automatically build and test the project code until finally you deploy it to any number of target users.

1) Now what it does is, it combines **continuous integration and continuous deployment** to build, test and deploy project code to target users.

2) In simple terms, Azure pipelines uses CI/CD pipelines or Build/Release pipelines that automates the activities performed developer on the code.

3) As the name suggests, Continuous integration combines all the code produced by developer's and builds and test the code. Continuous deployment on the other hand is an extension of continuous integration which helps in deploying the code.

Both of these processes are done automatically.

4) In devops, short forms like **CI/CD** are interchangeably used for continuous integration and continuous deployment.

5) First of all, an important prerequisite that needs to be met before using CI/CD is that the project code/Source code must reside in **version control system** i.e we need to push the code in the version control repositories.

6) Well, some of the Version controls supported by azure are **GitHub (Distributed VCS),** Git Hub Enterprise, Azure GIT Repo and TFVC (centralized VCS), Bitbucket cloud (webbased VCS) and subversion (centralized VCS). You can choose any of these repos to push the code, but Git Hub is the one most widely used.

7) In azure pipelines, most of the languages are supported like ruby, **.Net**, PHP, java, python, java script etc.

Prerequisites:

1) Create an organizations in Azure Devops.

2)Store/Push the project source code in version control repository.

YAML File Vs Classic Editor

There are two platforms where we can use pipelines through YAML file as code and classic editor on web portal.

- In both methods, you need to configure azure pipelines to use Git Repo and push the code the code to version control repository. This action activates a default triggers the pipeline.

YAML file:

In here, the pipeline is versioned with code and the main structure is based on code.

- This build structure can be modified by altering **azure-pipelines.yml file.**
- Because it is a code, it goes through standard code review or pull request process.
- The fact that it is a text file makes easy for us to share between team members for collaboration.
- There is an **Azure Pipelines Extension** for **VsCode**. It helps with syntax highlighting and auto completion.

Classic Editor:

It is used to create a pipeline without using yaml file. Here, a build and Release pipelines are created to build, test the code and consume, deploy the artifacts to the target.

- Unlike yaml file, where we need to use code, the classic editor uses **GUI** to get do build and release pipelines.
- The build pipeline will create an **artifact** that will run tasks like deployment to particular targets.

Note: Few features are supported either in yaml or Classic Editor. For instance, the Classic editor does not support **container jobs**, but the YAML editor does. Similarly, the Classic editor supports **gates**, but the YAML editor does not.

Pipelines using YAML File

1) Every branch can modify the pipeline by modifying **azure-pipeline.ymls**.

2) The process is very simple here,

Editcode->EditYamlFile->PushCodetoRepo->AzurePipelines->DeploytoTarget.

3) The pipeline is versioned with your code. It follows the same branching structure.

4)You get validation of your changes through code reviews in pull requests and branch build policies.

4) Change to the build process might cause a break or result in an unexpected outcome. Because the change is in version control with the rest of your code base, you can more easily identify the issue.

Create a pipeline:

1) Prerequisites:

- Create a GitHub account
- Create an Azure devops Organization. If your team members already created it, make sure you were made administrator for the project you need to work on.
- Ability to run pipelines on Microsoft-Hosted agents.

2) Sign-in to your Azure DevOps organization and go to your project. Go to Pipelines, and then select New pipeline.

3) First select GitHub as the location of your source code. Sign-in to GitHub. When you see the list of repositories, select your repository.

4) We might be redirected to GitHub to install the Azure Pipelines app. If so, select Approve & install.

5) Azure Pipelines will analyze your repository and recommend the ASP.NET Core pipeline template. When your new pipeline appears, select Save and run.

6) You're prompted to commit a new azure-pipelines.yml file to your repository. select Save and run again.

7) To make changes to your pipeline, select it in the Pipelines page, and then Edit the azure-pipelines.yml file.

Customizing a pipeline:

1) Go to Pipelines, select the pipeline you created, and choose Edit in the context menu of the pipeline to open the YAML editor for the pipeline. Following content appears.

```yaml
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: Maven@3
  inputs:
    mavenPomFile: 'pom.xml'
    mavenOptions: '-Xmx3072m'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.8'
    jdkArchitectureOption: 'x64'
    publishJUnitResults: false
    testResultsFiles: '**/surefire-reports/TEST-*.xml'
    goals: 'package'
```

-> Here, trigger: represents the branch opened.

Pool: vmImage: represents the platform the code is running on.

Steps: represent the tasks to be run.

2) To change the platform, you can edit the vmImage:

-> for instance, in above picture platform is “ubuntu-latest”. You can change vmImage to “windows-latest” to change the platform from linux to windows.

3) We can add more scripts or tasks as steps to the pipelines.

->For instance, in the above snippet a task named maven handles testing and publishes the result.

->To add a task that publishes code coverage results, add the following code to the file.

-task: PublishCodeCoverageResults@1 inputs: codeCoverageTool: "JaCoCo" summaryFileLocation:

"$(System.DefaultWorkingDirectory)/**/site/jacoco/jacoco.xml" reportDirectory: "$(System.DefaultWorkingDirectory)/**/site/jacoco" failIfCoverageEmpty: true

->After saving the change, we can view our test and code coverage results in test and coverage tab.

4) Build and test pipelines on multiple platforms or versions using **strategy** and **matrix**.

->If we write the following code, we can run a job on three different platforms.

```
strategy:
    matrix:
        linux:
            imageName: "ubuntu-latest"
        mac:
            imageName: "macOS-latest"
        windows:
            imageName: "windows-latest"
    maxParallel: 3 pool:
vmImage: $(imageName)
```

->To use multiple versions on a single platform

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

And also in maven task, replace the line “jdkversionoption: 1.11” to jdkversionoption:$(jdkVersion).

-> If you want to build multiple platforms on multiple version,

```
strategy:
```

```
matrix:
        jdk10_linux:
                imageName: "ubuntu-latest"
                jdkVersion: "1.10"
        jdk11_windows:
                imageName: "windows-latest"
                jdkVersion: "1.11"
    maxParallel: 2 pool: vmImage: $(imageName)
```

5) Customize CI triggers: Triggers cause pipelines to run. We can run pipelines on specific branches we want.

-> By default, the trigger runs on **main** branch.

```
Trigger:
    -main.
```

->If we want a pipeline to run after a pull request validation, change “trigger” to “pr”

```
Pr:
    -main
    -releases/*
```

Pipelines using Classic Editor

1) **Prerequisites:** The following steps must completed.

-> Fork the Microsoft/PowerAppsTestAutomation project on GitHub.

-> Create a new Test URL .json file in the repo with the App Test URLs you want to run from the pipeline.

2) **Create a Pipeline:**

-> Sign in to your Azure DevOps instance.

-> Select an existing project or create a new project.

-> Select Pipelines in the left menu.

-> Select Create Pipeline:

-> Select Use the classic editor

-> Select GitHub as the source.

-> Select ... (ellipsis) from the right side of Repository input.

-> Enter the name of your project on GitHub, and then Select it.

-> Select Continue.

-> In the Select a template screen, select Empty job.

-> Save your pipeline.

3) **Add Tasks to the pipeline:** You'll now add new job tasks and configure the tasks to run the tests from the pipeline in this sequence:

- Configure screen resolution using PowerShell:

  -> Select + next to Agent job 1

  -> Search for PowerShell.

  -> Select Add to add a PowerShell task to the job

  -> Select the task.

  -> Select Inline as the script type, and enter the following in the script window:

  " # Set agent screen resolution to 1920x1080 to avoid sizing issues with Portal

  Set-DisplayResolution -Width 1920 -Height 1080 -Force

  # Wait 10 seconds

Start-Sleep -s 10

Verify Screen Resolution is set to 1920x1080

Get-DisplayResolution”

- Restore NuGet packages:

  -> Select + next to Agent job 1

  -> Search for NuGet.

  -> Select Add to add a NuGet task to the job.

  -> Select the task.

  -> Select … (ellipsis) in the Path to solution, packages.config, or project.json configuration field.

  -> Select the PowerAppsTestAutomation.sln solution file.

  -> Select OK: NuGet package.

- Build the PowerAppsTestAutomation solution:

  -> Select + next to Agent job 1

  -> Search for Visual Studio build.

  -> Select Add to add a Visual Studio build task to the job.

  -> Select the task.

  -> Select … (ellipsis) in the Solution configuration field.

  -> Select the PowerAppsTestAutomation.sln solution file.

  -> Select OK.

- Add Visual Studio Tests for Google Chrome:

  -> Select + next to Agent job 1

  -> Search for Visual Studio Test.

  -> Select Add to add a Visual Studio Test task to the job.

  -> Select the task.

  -> Remove the default entries in the Test files text field and add the following: “**\Microsoft.PowerApps.TestAutomation.Tests\bin\\Debug\Microsoft.PowerApps.TestAutomation.Tests.dll”

-> Enter “TestCategory=PowerAppsTestAutomation” in the Test filter criteria field.

-> Select Test mix contains UI tests.

-> Select … (ellipsis) in the Settings file field.

->Expand the Microsoft.PowerApps.TestAutomation.Tests, select the patestautomation.runsettings file, and then select OK:

->Copy the following in the Override test run parameters field.

“OnlineUsername "$(OnlineUsername)" -OnlinePassword "$(OnlinePassword)" -BrowserType "$(BrowserTypeChrome)" -OnlineUrl "$(OnlineUrl)" -UsePrivateMode "$(UsePrivateMode)" -TestAutomationURLFilePath "$(TestAutomationURLFilePath)" -DriversPath "$(ChromeWebDriver)"

-> Enter Run Power Apps Test Automation Tests via $(BrowserTypeChrome) or similar in the Test run title field.

- Add Visual Studio Tests for Mozilla Firefox

  -> Right-click the Add Visual Studio Tests for Chrome task and select Clone tasks.

  -> Select the task and update the following areas:

  a)**Title**: Run Power Apps Test Automation Tests via $(BrowserTypeFirefox)

  b)Override test run parameters

  “ Copy -OnlineUsername "$(OnlineUsername)" -OnlinePassword "$(OnlinePassword)" -BrowserType "$(BrowserTypeFirefox)" -OnlineUrl "$(OnlineUrl)" -UsePrivateMode "$(UsePrivateMode)" -TestAutomationURLFilePath "$(TestAutomationURLFilePath)" -DriversPath "$(GeckoWebDriver)"

  c)**Test Run Title**: Run Power Apps Test Automation Tests via $(BrowserTypeFirefox)

- Run and analyze tests:

-> To validate that your tests are executing successfully, select Queue and then
-> select Run. Your job will start running. Run job. As the job runs,
-> select the job to see a detailed status on each of the tasks running.
->When the job completes, you can view the high-level job summary, and any errors or warnings. By selecting the Tests tab, you can view specific details on the test cases you've executed.
-> Select RunTestAutomation test to drill into the details on what test case has failed. In the Attachments tab, you can see the summary of the test execution and which test cases have failed or passed in your test suite.