Continuous Integration using YAML Pipelines

- Introduction to YAML Pipeline

- Building Azure DevOps Pipeline using YAML

- Publishing results to Artifacts

- Triggering Continuous Integration in YAML

- Filtering Tasks based on branch being built

- Using Templates to Build Multiple Configurations

- Build on Multi-Platform pipeline

<div style="background:black;color:white;text-align:center;font-weight:bold;">Introduction to YAML Pipeline</div>

- **YAML**: **Y**et **A**nother **M**arkup **L**anguage

- The pipeline is versioned with your code and follows the same branching structure. You get validation of your changes through code reviews in pull requests and branch build policies.

- Every branch you use can modify the build policy by modifying the **azure-pipelines.yml** file.

- A change to the build process might cause a break or result in an unexpected outcome. Because the change is in version control with the rest of your codebase, you can more easily identify the issue.

Exercise: Build Azure DevOps Pipeline using YAML

1.   Pipeline → **New Pipeline**

2.   Select Tab → Select **HelloWorldApp.Web** repository.

3.   On the **Configure** tab, select **ASP.NET Core** (click on choose more)

4.   In Visual Studio Code, modify *azure-pipelines.yml* as you see here:

```
trigger:
- 'none'

pool:
  vmImage: 'ubuntu-16.04'

steps:
- task: DotNetCoreCLI@2
  displayName: 'Restore project dependencies'
  inputs:
    command: 'restore'
    projects: '**/*.csproj'
```

```
- task: DotNetCoreCLI@2
  displayName: 'Build the project - Release'
  inputs:
   command: 'build'
   arguments: '--no-restore --configuration Release'
   projects: '**/*.csproj'


- task: DotNetCoreCLI@2
  displayName: 'Publish the project - Release'
  inputs:
   command: 'publish'
   projects: '**/*.csproj'
   publishWebProjects: false
   arguments: '--no-build --configuration Release --output $(Build.ArtifactStagingDirectory)/Release'
   zipAfterPublish: true


- task: PublishBuildArtifacts@1
  displayName: 'Publish Artifact: drop'
  condition: succeeded()
```

The **first task** uses the **DotNetCoreCLI@2** task to *publish*, or package, the application's build results (including its dependencies) into a folder.

The **zipAfterPublish** argument specifies to add the built results to a .zip file.

The **second task** uses the PublishBuildArtifacts@1 task to publish the .zip file to Azure Pipelines.

Triggering Continuous Integration in YAML

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

```
trigger:
- master
- releases/*
```

```
# A pipeline with no CI trigger
```

```
trigger: none
```

```
# specific branch build with batching
```

```
trigger:
 batch: true
```

2

```
 branches:
   include:
   - master
```

specific branch build

```
trigger:
 branches:
   include:
   - master
   - releases/*
   exclude:
   - releases/old*
 paths:
   include:
   - docs/*
   exclude:
   - docs/README.md
```

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

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

You can specify the target branches for your pull request builds.

YAML PR triggers are only supported in GitHub and Bitbucket Cloud and for NOT for Azure Repos

```
pr:
- master
- releases/*
```

If **no pr triggers** appear in your YAML file, pull request builds are **automatically enabled** for all branches, as if you

```
pr:
```

3

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

Override YAML triggers

PR and CI triggers that are configured in YAML pipelines can be overridden in the pipeline settings, and by default, new pipelines automatically override YAML PR triggers. To configure this setting, select **Triggers** from the settings menu while editing your YAML pipeline.

<div style="background:black;color:white;text-align:center;">

Using Templates to Build Multiple Configurations

</div>

A *template* enables you to define common build tasks one time and reuse those tasks multiple times.

You call a template from the parent pipeline as a build step. You can pass parameters into a template from the parent pipeline.

Templates combine the content of multiple YAML files into a single pipeline.

Requirement: We need to now repeat the two tasks **Build** and **Publish** but replace **Release** with **Debug**

1. Create a *templates* directory at the root of your project:

2. Create a new File **templates\build.yml**

```
parameters:
  buildConfiguration: 'Release'

steps:
- task: DotNetCoreCLI@2
  displayName: 'Build the project - ${{ parameters.buildConfiguration }}'
  inputs:
   command: 'build'
   arguments: '--no-restore --configuration ${{ parameters.buildConfiguration }}'
   projects: '**/*.csproj'

- task: DotNetCoreCLI@2
  displayName: 'Publish the project - ${{ parameters.buildConfiguration }}'
  inputs:
   command: 'publish'
   projects: '**/*.csproj'
   publishWebProjects: false
   arguments: '--no-build --configuration ${{ parameters.buildConfiguration }} --
output $(Build.ArtifactStagingDirectory)/${{ parameters.BuildConfiguration }}'
```

4

```
zipAfterPublish: true
```

Note the two differences

- In a template file, you use the parameters section instead of variables to define inputs.

- In a template file, you use ${{ }} syntax instead of $() to read a parameter's value. When you read a parameter's value, you include the parameters section in its name. For example, ${{ parameters.buildConfiguration }}.

3. Call the template from the pipeline

4. Place the below two task just above the last task in **azure-pipelines.yml**

```
- template: templates/build.yml
  parameters:
    buildConfiguration: 'Debug'


- template: templates/build.yml
  parameters:
    buildConfiguration: 'Release'
```

5. You see that the pipeline produces a *.zip* file for both the **Debug** configuration and the **Release** configuration.