Agenda: Integrating Quality Tests in Pipeline

- Add Unit Tests to your Application
- Integrating Unit Test with CI Pipeline
- Add the Test Widget to Dashboard
- Perform Code Coverage Testing using Cobertura
- Filtering Tasks based on branch being built

Add Unit Tests to your application

1. Add the following to **HelloWorldApp.Web** Application

MathOperations.cs

```
public class MathOperations
{
  public double Add(double num1, double num2)
  {
    return num1 + num2;
  }
  public double Subtract(double num1, double num2)
  {
    return num1 - num2;
  }
  public double Divide(double num1, double num2)
  {
    return num1 / num2;
  }
  public double Multiply(double num1, double num2)
  {
    // To trace error while testing, writing + operator instead of * operator.
    return num1 + num2;
  }
}
```

2. **Add a Test Project to the Solution**

```
dotnet new nunit --name HelloWorldApp.Tests
dotnet sln HelloWorldApp.sln add HelloWorldApp.Tests\HelloWorldApp.Tests.csproj
dotnet add HelloWorldApp.Tests\HelloWorldApp.Tests.csproj reference
HelloWorldApp.Web\HelloWorldApp.Web.csproj
```

3. Add the following class to Test Project.

```
using NUnit.Framework;
using HelloWorldApp.Web;

public class MathOperationsTests
{
    [Test()]
    public void AddTest()
    {
        MathOperations bm = new MathOperations();
        double res = bm.Add(10, 10);
        Assert.AreEqual(res, 20);
    }

    [Test()]
    public void SubtractTest()
    {
        MathOperations bm = new MathOperations();
        double res = bm.Subtract(10, 10);
        Assert.AreEqual(res, 0);
    }

    [Test()]
    public void DivideTest()
    {
        MathOperations bm = new MathOperations();
        double res = bm.Divide(10, 5);
        Assert.AreEqual(res, 2);
    }

    [Test()]
    public void MultiplyTest()
    {
        MathOperations bm = new MathOperations();
        double res = bm.Multiply(10, 5);
        Assert.AreEqual(res, 50);
    }
}
```

4. Add the following Task to YAML

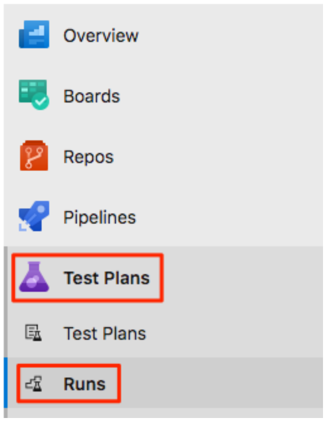
```yaml
- task: DotNetCoreCLI@2
  displayName: 'Run unit tests - $(buildConfiguration)'
  inputs:
    command: 'test'
    arguments: '--no-build --configuration $(buildConfiguration)'
    publishTestResults: true
    projects: '**/*Tests.csproj'
```

Updated YAML file will as below

```yaml
trigger:
- none

pool:
  vmImage: 'vs2017-win2016'

variables:
  buildConfiguration: 'Release'
  dotnetsdk: 3.1.x

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

- task: DotNetCoreCLI@2
  inputs:
    command: 'build'
    projects: '**/*.csproj'
    arguments: '--configuration $(buildConfiguration)'
```

```
- task: DotNetCoreCLI@2
   displayName: 'Run unit tests - $(buildConfiguration)'
   inputs:
     command: 'test'
     arguments: '--no-build --configuration $(buildConfiguration)'
     publishTestResults: true
     projects: '**/*Tests.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()
```

5. Navigate back to the pipeline summary.
6. Move to the **Tests** tab.
7. In Azure DevOps, select **Test Plans**, and then select **Runs**.



You see the most recent test runs, including the one you just ran.

8. Double-click the most recent test run.

You see a summary of the results.

Add the widget to the Dashboard

1. In your Azure DevOps project, select **Overview**, and then select **Dashboards**.

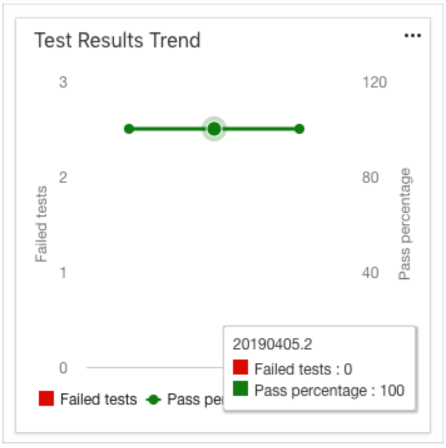
   Note

   If you ran the template to create the Azure DevOps project, you won't see the dashboard widgets you set up in previous modules.
2. Select **Edit**.
3. In the **Add Widget** pane, search for **Test Results Trend**.
4. Drag **Test Results Trend** to the canvas.
5. Select the gear icon to configure the widget.

   a. Under **Build pipeline**, select your pipeline.

   b. Keep the other default settings.
6. Select **Save**.
7. Select **Done Editing**.

Although the widget displays only one test run, you now have a way to visualize and track test runs over time. Here's an example that shows a few successful test runs.



If you begin to see test failures, you can click a point on the graph to navigate directly to that build.

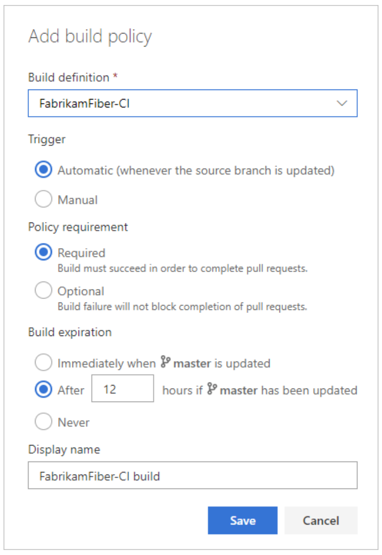
Validate Pull Request based on Build Pipeline result

- Set a policy requiring changes in a pull request to build successfully with the protected branch before the pull request can be completed. Build policies reduce breaks and keep your test results passing.
- Build policies help even if you're using continuous integration (CI) on your development branches to catch problems early.
- If a build validation policy is enabled, a new build is queued when either a new pull request is created, or if changes are pushed to an existing pull request targeting the branch. The build policy then evaluates the results of the build to determine whether the pull request can be completed.

Note that the steps required are different for Azure Repos and GitHub Projects

Steps for Azure Repos and TFS Repositories

1. Azure Repos → Branches → Select a branch → **Branch Policies** → Build Validation →
2. Choose **Add build policy** and configure your options in **Add build policy**.



Add build policy

Build definition *

FabrikamFiber-CI

Trigger

[✓ Automatic (whenever the source branch is updated)]

Manual

Policy requirement

[✓ Required]
Build must succeed in order to complete pull requests.

Optional
Build failure will not block completion of pull requests.

Build expiration

Immediately when master is updated

[✓ After] 12 hours if master has been updated

Never

Display name

FabrikamFiber-CI build

1. Select the **Build definition**.
2. Choose the type of **Trigger**. Select **Automatic (whenever the source branch is updated)** or **Manual**.
3. Select the **Policy requirement**. If you choose **Required**, builds must complete successfully to complete pull requests. Choose **Optional** to provide a notification of the build failure but still allow pull requests to complete.
4. Set a **Build expiration** to make sure that updates to your protected branch don't break changes for open pull requests.
   - **Immediately when <branch name> is updated**: This option sets the build policy status in a pull request to ***failed*** when the protected branch is updated. Requeue a build to refresh the build status. This setting ensures that the changes in pull requests build successfully even as the protected branch changes. This option is best for teams that have important branches with a lower volume of changes. Teams working in busy development branches may find it disruptive to wait for a build to complete every time the protected branch is updated.
   - **After n hours if branch name has been updated**: This option expires the current policy status when the protected branch updates if the passing build is older than the threshold entered. This option is a compromise between always requiring a build when the protected branch updates and never requiring one. This choice is excellent for reducing the number of builds when your protected branch has frequent updates.

- **Never**: Updates to the protected branch don't change the policy status. This value reduces the number of builds for your branch. It can cause problems when closing pull requests that haven't updated recently.

5. Enter an optional **Display name** for this build policy. This name identifies the policy on the **Branch policies** page. If you don't specify a display name, the policy uses the build definition name.
6. Select **Save**.

To Test Policy

1. To test the policy navigate to the **Pull request** menu in Azure Pipelines or TFS.
2. Select **New pull request**. Ensure your topic branch is set to merge into your master branch. Select **create**.
3. Your screen displays the **policy** being executed.
4. Select the **policy name** to examine the build. If the build succeeds your code will be merged to master. If the build fails the merge is blocked.

For GitHub:

Classic Editor:

Edit Pipeline → Select **Triggers**. Enable the **Pull request validation** trigger. Ensure you include the **master branch** under **Branch filters** → Save

OR

YAML Editor:

```
pr:
- master
```

Or

Pipeline → Edit → ". . ." → Triggers → Check **Override the YAML pull request trigger from here**

To Test

1. Edit a file in branch and **Create the pull request**.
2. Navigate back to your build pipeline. A build will be queued or completed for the merge commit of your pull request.

Life Cylce so far:

```
Pull Master
  Create Feature Branch
     Edit Code in Local Feature Branch
                      Push to remote feature branch
                                 Create Pull request from Feature to Master
```

Validation Pipeline is run (Branch Validation Policy)

If Failure, Wait for corrected code...from Local Feature Branch

If Success, Complete Pull Request.

Start Build Pipeline to produce Artifacts

Start Release Pipeline to Publish Build Artifacts to Target Env.

Perform Code Coverage Testing

- Code coverage tells us how much of our code has associated unit tests.
- **Coverlet** is a cross-platform, code-coverage library for .NET Core.
- We can add in *code coverage*. That will tell us the percentage of our code that has unit tests. We can use a tool called "**coverlet**" to collect coverage information when the tests run.
- Code coverage results are written to an XML file so that they can be processed by another tool. Azure Pipelines supports Cobertura and JaCoCo coverage result formats.

1. **Add the following NuGet Packages in the Test Project**
   - coverlet.msbuild
   - **Microsoft.CodeCoverage**
   - Microsoft.NET.Test.Sdk
   - ReportGenerator.Core - [by Daniel Palme]

Update the .csproj of Unit Test project

```
<ItemGroup>
  <PackageReference Include="coverlet.msbuild" Version="2.9.0">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
  </PackageReference>
  <PackageReference Include="Microsoft.CodeCoverage" Version="16.8.0" />
  <PackageReference Include="ReportGenerator.Core" Version="4.7.1" />
  <PackageReference Include="nunit" Version="3.12.0" />
  <PackageReference Include="NUnit3TestAdapter" Version="3.15.1" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.4.0" />
</ItemGroup>
```

2. Run the following dotnet test command to run your unit tests and collect code coverage:

```
dotnet test --configuration Release /p:CollectCoverage=true /p:CoverletOutputFormat=cobertura
/p:CoverletOutput=./TestResults/Coverage/ /p:threshold=80 /p:thresholdType=line /p:thresholdStat=total
```

This command resembles the one you ran previously. The /p: flags tell coverlet which code coverage format to use and where to place the results.

As a result of this command, we will have: **coverage.cobertura.xml** (coverage results in Cobertura format).

To convert Cobertura coverage results to a format that's human-readable, they can use a tool called ReportGenerator .

3. Run the following dotnet tool install command to install ReportGenerator**: (THIS IS FOR WINDOWS)**

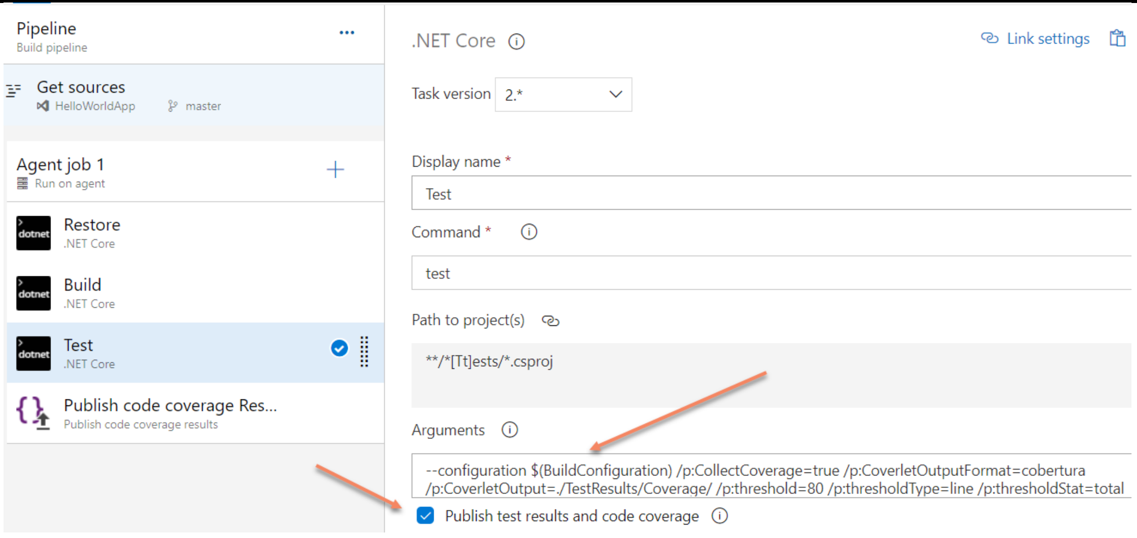
```
dotnet tool install --global dotnet-reportgenerator-globaltool
```

4. Run the following **reportgenerator** command to convert the Cobertura file to HTML:

```
reportgenerator -reports:./HelloWorldApp.Tests/TestResults/Coverage/coverage.cobertura.xml -
targetdir:./CodeCoverage -reporttypes:HtmlInline_AzurePipelines
```

- ReportGenerator provides a number of formats, including HTML. The HTML formats create detailed reports for each class in a .NET project.
- Specifically, there's an HTML format called **HtmlInline_AzurePipelines**, which provides a visual appearance that matches Azure Pipelines.

Classic Pipeline



Task version: 2.*

Display name *: Test

Command *: test

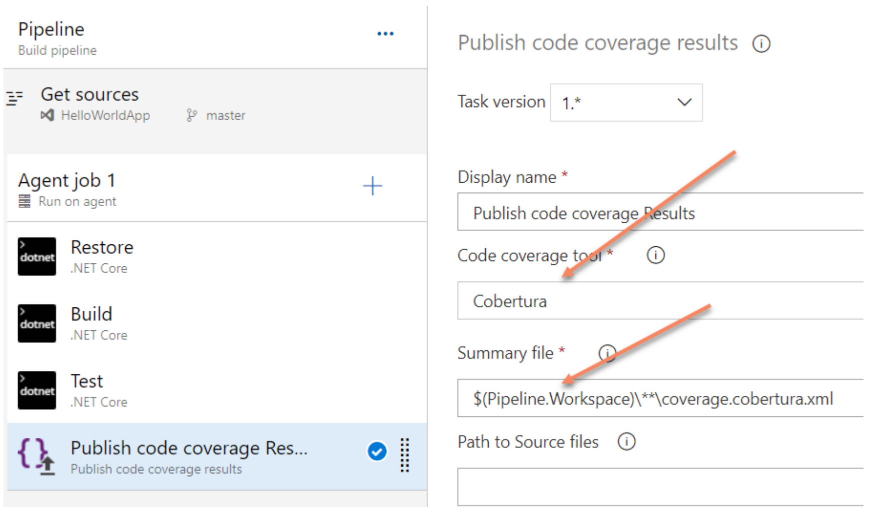
Path to project(s): **/*[Tt]ests/*.csproj

Arguments: --configuration $(BuildConfiguration) /p:CollectCoverage=true /p:CoverletOutputFormat=cobertura /p:CoverletOutput=./TestResults/Coverage/ /p:threshold=80 /p:thresholdType=line /p:thresholdStat=total

☑ Publish test results and code coverage

Arguments:

--configuration $(buildConfiguration) /p:threshold=80 /p:thresholdType=line /p:thresholdStat=total /p:CollectCoverage=true /p:CoverletOutputFormat=cobertura /p:CoverletOutput=./TestResults/Coverage/


Pipeline
Build pipeline
Get sources
HelloWorldApp master
Agent job 1
Run on agent
Restore
.NET Core
Build
.NET Core
Test
.NET Core
Publish code coverage Res...
Publish code coverage results

Publish code coverage results
Task version: 1.*
Display name *: Publish code coverage Results
Code coverage tool *: Cobertura
Summary file *: $(Pipeline.Workspace)**\coverage.cobertura.xml
Path to Source files:

YAML Pipeline

```yaml
trigger:
- none

pool:
  vmImage: "windows-latest"

variables:
  buildConfiguration: 'Release'

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

- task: DotNetCoreCLI@2
  inputs:
    command: 'build'
    projects: '**/*.csproj'
    arguments: '--configuration $(buildConfiguration)'

- task: DotNetCoreCLI@2
  displayName: 'Run unit tests - $(buildConfiguration)'
  inputs:
    command: 'test'
    arguments: '--no-build --configuration $(buildConfiguration) /p:threshold=80 /p:thresholdType=line
/p:thresholdStat=total /p:CollectCoverage=true /p:CoverletOutputFormat=cobertura /p:CoverletOutput=./Test
Results/Coverage/'
```

```
    publishTestResults: true
    projects: '**/*Test.csproj'

~~- task: DotNetCoreCLI@2~~
~~  displayName: 'Install ReportGenerator'~~
~~  inputs:~~
~~    command: custom~~
~~    custom: tool~~
~~    arguments: 'install --global dotnet-reportgenerator-globaltool'~~

~~- script: reportgenerator -reports:$(Build.SourcesDirectory)/**/coverage.cobertura.xml -~~
~~targetdir:$(Build.SourcesDirectory)/CodeCoverage -reporttypes:HtmlInline_AzurePipelines~~
~~ displayName: 'Create code coverage report'~~
~~  condition: 'succeededOrFailed()'~~

- task: PublishCodeCoverageResults@1
  displayName: 'Publish code coverage report'
  inputs:
    codeCoverageTool: 'cobertura'
    summaryFileLocation: '$(Pipeline.Workspace)/**/coverage.cobertura.xml'
  condition: 'succeededOrFailed()'
```

Watch Azure Pipelines run the tests

When the build finishes, Navigate back to the **summary page** and select the **Code Coverage** tab.