**Exercise**

**Perform code coverage testing**

Much like the tool you use for unit testing, the tool you use for code coverage depends on the programming language and application framework you use.

When you target .NET Core applications to run on Linux, coverlet  is a popular option. Coverlet is a cross-platform, code-coverage library for .NET Core. Before we add code coverage to the pipeline, let's check in with the team.

Amita feels much better. She's seeing actual progress in catching bugs and in easily viewing test results. What's more, it hasn't taken long for Andy and Mara to implement the improvements. Amita, Andy, and Mara talk a bit more about unit testing.

**Andy:** I like the idea of unit testing, but I've never found a good way to know when we're done, when we have complete coverage. Are there any good tools we can use with Microsoft Azure Pipelines?

**Mara:** 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.

**Andy:** That's cool. Sounds like we can establish a baseline and improve over time.

**Mara:** That's right. Eventually we can even configure the build to fail if we don't meet a given threshold. That would help keep us honest. But for now, we can just see how much is covered.

**Andy:** Great. Getting reports on how much code is covered by unit tests will help us to identify code paths that aren't covered. We can increase the coverage gradually, and that will help save us from feeling overwhelmed by how much there is to do.

**Amita:** I'm really excited about the unit tests. I mostly do manual testing. I focus on the customer's perspective. I don't just look for bugs. I make sure the software does what it's specified to do, that the UI works, and that the user has a good experience.

**Mara:** That perspective is so important and definitely needs a human being. Right now, Andy and I are working on automated tests, software that tests the software. We're concentrating on tests that execute as the software moves through the build pipeline.

That means the two types of tests we've already talked about. The unit tests test individual components and are really fast. Code coverage tells us how much of our code has associated unit tests.

**Andy:** We should also think about doing lint testing from the command line, before the build. Lint testing can help us catch bugs, programming errors, and coding style problems really early.

**Amita:** What about regression tests?

**Mara:** I think of regression tests and unit tests as almost the same thing. After we fix a bug, we should run the unit tests again. This ensures that our changes haven't broken any units that were already tested.

**Amita:** OK, so does that leave integration testing?

**Andy:** Integration testing is a bit different. We do integration testing after the build, on the server. Although unit tests help you verify a single component like a function or method, integration testing verifies that multiple components work together. I don't think we're ready for integration tests quite yet.

But at some point we also need to think about security and compliance. We should work with the security team to figure out how we can test against their security policies. I remember that was a concern of Tim's and I'd like to bring him into the process more.

**Mara:** Lots to do.

**Amita:** Thanks for the rundown! I'm off. Keep me posted.

**Andy:** Ready to do some code coverage?

**Mara:** Let's get started.

**How is code coverage done in .NET Core?**

The way you collect code coverage depends on what programming language and frameworks you're using, as well as what code coverage tools are available.

Mara and Andy do some investigation around code coverage for .NET Core applications. Here's what they find:

* Visual Studio on Windows provides a way to perform code coverage.
* However, because the team is building on Linux, they can use coverlet , a cross-platform code coverage library for .NET Core.

The unit test project requires the coverlet.msbuild  NuGet package.

* 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.

Mara and Andy decide to try Cobertura.

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

**How can I manage .NET Core tools?**

A .NET Core tool such as ReportGenerator is a special NuGet package that contains a console application. You can manage a .NET Core tool as a global tool or as a local tool.

A global tool is installed in a centralized location and can be called from any directory. One version of a global tool is used for all directories on the machine.

A local tool is a more isolated copy of a .NET Core tool that's scoped to a specific directory. Scope enables different directories to contain different versions of the same tool.

You use a *manifest file* to manage local tools for a given directory. This file is in JSON format and is typically named *dotnet-tools.json*. A manifest file enables you to describe the specific tool versions that you need to build or run your application.

When you include the manifest file in source control along with your application sources, developers and build systems can run the dotnet tool restore command to install all of the tools listed in the manifest file. When you need a newer version of a local tool, you simply update the version in the manifest file.

To keep things more isolated, in this module you work with local tools. You create a tool manifest that includes the ReportGenerator tool. You also modify your build pipeline to install the ReportGenerator tool to convert code coverage results to a human-readable format.

**Run code coverage locally**

Before Mara and Andy write any pipeline code, they decide to try things manually to verify the process. Follow along with their process:

1. In Visual Studio Code, open the integrated terminal.
2. Run the following dotnet new command to create a local tool manifest file.

**Bash**

dotnet new tool-manifest

The command creates a file named *.config/dotnet-tools.json*.

1. Run the following dotnet tool install command to install ReportGenerator:

**Bash**

dotnet tool install dotnet-reportgenerator-globaltool

This command installs the latest version of ReportGenerator and adds an entry to the tool manifest file.

1. Run the following dotnet add package command to add the coverlet.msbuild package to the *Tailspin.SpaceGame.Web.Tests* project:

**Bash**

dotnet add Tailspin.SpaceGame.Web.Tests package coverlet.msbuild

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

**Bash**

dotnet test --no-build \

--configuration Release \

/p:CollectCoverage=true \

/p:CoverletOutputFormat=cobertura \

/p:CoverletOutput=./TestResults/Coverage/

If the command fails, try running it like this:

**Bash**

MSYS2\_ARG\_CONV\_EXCL="\*" dotnet test --no-build \

--configuration Release \

/p:CollectCoverage=true \

/p:CoverletOutputFormat=cobertura \

/p:CoverletOutput=./TestResults/Coverage/

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.

1. Run the following dotnet tool run command to use ReportGenerator to convert the Cobertura file to HTML:

**Bash**

dotnet tool run reportgenerator \

-reports:./Tailspin.SpaceGame.Web.Tests/TestResults/Coverage/coverage.cobertura.xml \

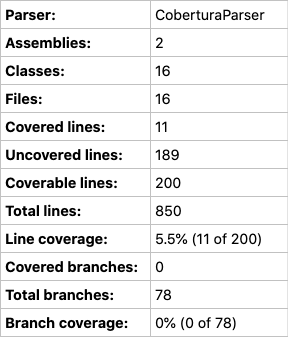
-targetdir:./CodeCoverage \

-reporttypes:HtmlInline\_AzurePipelines

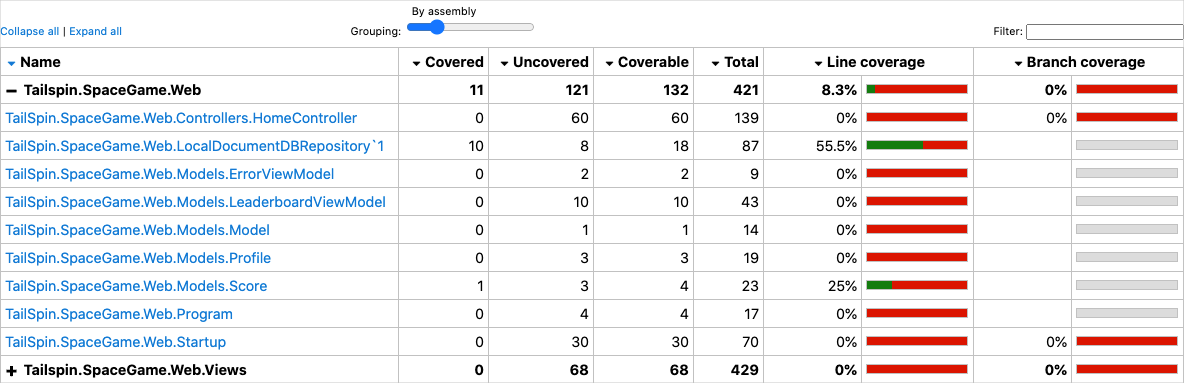
A number of HTML files appear in the *CodeCoverage* folder at the root of the project.

1. In Visual Studio Code, expand the *CodeCoverage* folder, right-click *index.htm*, and then select **Reveal in Explorer** (**Reveal in Finder** on macOS or **Open Containing Folder** on Linux).
2. In Windows Explorer (Finder on macOS), double-click *index.htm* to open it in a web browser.

You see the coverage report summary.

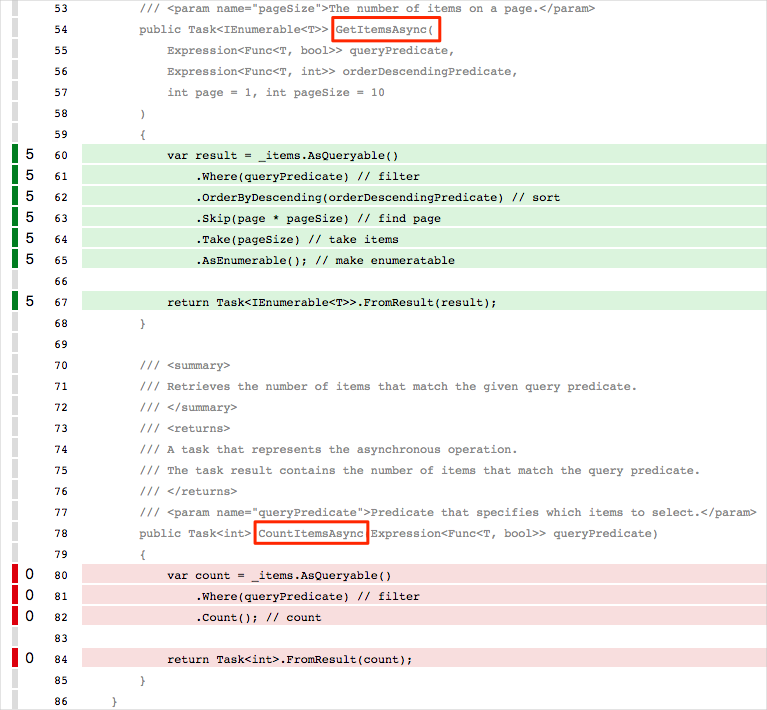


1. Scroll to the bottom of the page to see a coverage breakdown by class type.



1. Select the link to TailSpin.SpaceGame.Web.LocalDocumentDBRepository`1 to view further details.

Notice that the GetItemsAsync method is covered by unit tests, but the CountItemsAsync method has no coverage.



This makes sense, because the FetchOnlyRequestedGameRegion test method calls the GetItemsAsync method but does not call the CountItemsAsync method. (To review the test code, see the *DocumentDBRepository\_GetItemsAsyncShould.cs* file.)

**Create a branch**

Now that you can build a code coverage report locally, you're ready to add tasks to your build pipeline, which performs the same tasks.

In this part, you create a branch named code-coverage, based on the unit-tests branch, to hold your work. In practice, you would ordinarily create this branch from the master branch.

1. In Visual Studio Code, open the integrated terminal.
2. In the terminal, run the following git checkout command to create a branch named code-coverage:

**Bash**

git checkout -b code-coverage

**Add build tasks**

In this section, you add tasks that measure code coverage to your build pipeline.

1. In Visual Studio Code, modify *azure-pipelines.yml* like this:

**ymlCopy**

trigger:

- '\*'

pool:

vmImage: 'ubuntu-18.04'

demands:

- npm

variables:

buildConfiguration: 'Release'

wwwrootDir: 'Tailspin.SpaceGame.Web/wwwroot'

dotnetSdkVersion: '3.1.300'

steps:

- task: UseDotNet@2

displayName: 'Use .NET Core SDK $(dotnetSdkVersion)'

inputs:

version: '$(dotnetSdkVersion)'

- task: Npm@1

displayName: 'Run npm install'

inputs:

verbose: false

- script: './node\_modules/.bin/node-sass $(wwwrootDir) --output $(wwwrootDir)'

displayName: 'Compile Sass assets'

- task: gulp@1

displayName: 'Run gulp tasks'

- script: 'echo "$(Build.DefinitionName), $(Build.BuildId), $(Build.BuildNumber)" > buildinfo.txt'

displayName: 'Write build info'

workingDirectory: $(wwwrootDir)

- task: DotNetCoreCLI@2

displayName: 'Restore project dependencies'

inputs:

command: 'restore'

projects: '\*\*/\*.csproj'

- task: DotNetCoreCLI@2

displayName: 'Build the project - $(buildConfiguration)'

inputs:

command: 'build'

arguments: '--no-restore --configuration $(buildConfiguration)'

projects: '\*\*/\*.csproj'

- task: DotNetCoreCLI@2

displayName: 'Install .NET Core tools from local manifest'

inputs:

command: custom

custom: tool

arguments: 'restore'

- task: DotNetCoreCLI@2

displayName: 'Run unit tests - $(buildConfiguration)'

inputs:

command: 'test'

arguments: '--no-build --configuration $(buildConfiguration) /p:CollectCoverage=true /p:CoverletOutputFormat=cobertura /p:CoverletOutput=$(Build.SourcesDirectory)/TestResults/Coverage/'

publishTestResults: true

projects: '\*\*/\*.Tests.csproj'

- task: DotNetCoreCLI@2

displayName: 'Create code coverage report'

inputs:

command: custom

custom: tool

arguments: 'run reportgenerator -reports:$(Build.SourcesDirectory)/\*\*/coverage.cobertura.xml -targetdir:$(Build.SourcesDirectory)/CodeCoverage -reporttypes:HtmlInline\_AzurePipelines'

- task: PublishCodeCoverageResults@1

displayName: 'Publish code coverage report'

inputs:

codeCoverageTool: 'cobertura'

summaryFileLocation: '$(Build.SourcesDirectory)/\*\*/coverage.cobertura.xml'

- task: DotNetCoreCLI@2

displayName: 'Publish the project - $(buildConfiguration)'

inputs:

command: 'publish'

projects: '\*\*/\*.csproj'

publishWebProjects: false

arguments: '--no-build --configuration $(buildConfiguration) --output $(Build.ArtifactStagingDirectory)/$(buildConfiguration)'

zipAfterPublish: true

- task: PublishBuildArtifacts@1

displayName: 'Publish Artifact: drop'

condition: succeeded()

This version builds upon your existing configuration. Here's a summary of what's new:

| **TABLE 1** | | |
| --- | --- | --- |
| **Azure Pipelines task** | **Display name** | **Description** |
| DotNetCoreCLI@2 | Install .NET Core tools from local manifest | Installs tools listed in the manifest file, *dotnet-tools.json* |
| DotNetCoreCLI@2 | Run unit tests - $(buildConfiguration) | Runs unit tests and also collects code coverage in Cobertura format |
| DotNetCoreCLI@2 | Create code coverage report | Converts Cobertura output to HTML |
| PublishCodeCoverageResults@1 | Publish code coverage report | Publishes the report to the pipeline |

**Commit your changes and push the branch to GitHub**

Here you push your changes to GitHub and see the pipeline run. Recall that you're currently in the code-coverage branch.

Although not required, here you add and commit each file separately so that each change is associated with a descriptive commit message.

1. In Visual Studio Code, go to the terminal.
2. Add and commit the *Tailspin.SpaceGame.Web.Tests.csproj* file, which now contains a reference to the coverlet.msbuild package:

**Bash**

git add Tailspin.SpaceGame.Web.Tests/Tailspin.SpaceGame.Web.Tests.csproj

git commit -m "Add coverlet.msbuild package"

1. Add and commit the tool manifest file, *dotnet-tools.json*:

**Bash**

git add .config/dotnet-tools.json

git commit -m "Add code coverage"

1. Add and commit *azure-pipelines.yml*, which contains your updated build configuration:

**Bash**

git add azure-pipelines.yml

git commit -m "Add code coverage"

1. Push the code-coverage branch to GitHub.

**Bash**

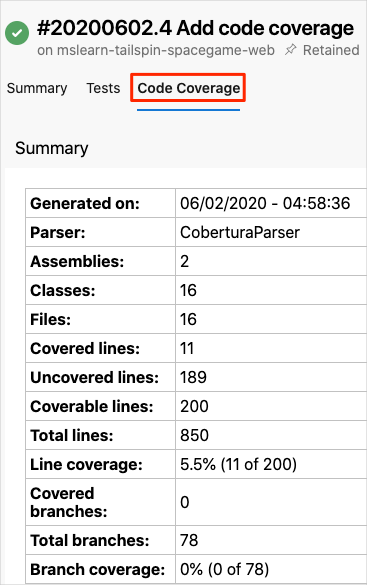
git push origin code-coverage

**Watch Azure Pipelines run the tests**

Here you see the tests run in the pipeline and then visualize the results from Azure Test Plans.

1. In Azure Pipelines, trace the build through each of the steps.
2. When the build finishes, navigate back to the summary page and select the **Code Coverage** tab.

You view the same results that you did when you ran the tests locally.



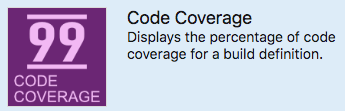
As an optional step, you can explore the results from Azure Pipelines.

**Add the dashboard widget**

In the previous part, you added the **Test Results Trend** widget to your dashboard, which lets others quickly review test result trends over time.

Here you add a second widget that summarizes code coverage.

1. In a new browser tab, go to marketplace.visualstudio.com .
2. On the **Azure DevOps** tab, search for **code coverage**.
3. Select **Code Coverage Widgets** (published by Shane Davis).
4. Select **Get it free**.
5. In the drop-down list, select your Azure DevOps organization.
6. Select **Install**.
7. Go back to Azure DevOps.
8. Go to **Overview** > **Dashboards**.
9. Select **Edit**.
10. Search for **Code Coverage**, and then select **Code Coverage**.



1. Drag **Code Coverage** to the canvas.
2. Select the gear icon to configure the widget.
3. Keep all the default settings, except for:
   * Width: Enter **2**.
   * Build definition: Select your pipeline.
   * Coverage measurement: Enter **Lines**.
4. Select **Save**.
5. Select **Done Editing**.

The widget shows the percentage of code your unit tests cover.



You now have code coverage set up in your pipeline. Although your existing code coverage is low, you have a baseline that you can improve over time.

Later, you can configure coverlet to check to see whether your tests provide a minimum threshold of coverage. Your threshold might be 30 percent, 50 percent, or 80 percent coverage, depending on your requirements. The build will fail if less than this amount is covered by your tests.

**Remove code coverage files**

Recall that when you ran Reportgenerator earlier, a number of HTML files appeared in the *CodeCoverage* folder at the root of the project.

These HTML files are not intended to be included in source control, and you no longer need them. Although the project's *.gitignore* file is already set up to ignore anything in the *CodeCoverage* directory, it's a good idea to delete these files so that they're not added to your Git repository in future modules.

In Visual Studio Code, go to the terminal window and then, in your project's root directory, run this command:

Bash

rm -rf CodeCoverage/