**Exercise**

**Fixing a failed test**

At this point, you have a way to run unit tests as changes move through the build pipeline. You also have a way to measure the amount of code that's covered by your tests.

It's always a good idea to run your tests locally before you submit changes to the pipeline. But what happens when someone forgets and submits a change that breaks the build?

In this unit, you help the team fix a broken build that's caused by a failing unit test. Here, you will:

* Get starter code from GitHub.
* Add code coverage tools to your project.
* Push the code up to your repository.
* Watch the pipeline automatically run and the unit tests fail.
* Reproduce the failure locally.
* Analyze and fix the failure.
* Push up a fix and watch the build succeed.

First, let's check in with Mara and Andy.

**A quick chat**

Mara shows Andy the updated build configuration on Azure Pipelines. Andy likes what he sees, especially the dashboard widgets.

**Andy:** Last time we met, we got code coverage working on your laptop. It's great to see the same thing working in the build pipeline!

**Mara:** Yes, I'm glad we got things working locally first. Fitting it into the build pipeline was easy after that.

**Andy:** I'll be sure to add a unit test the next time I add a feature. It'll be great to increase our code coverage! In fact, Amita is waiting on this new leaderboard feature so she can do user testing on it. I promised her I'd have the change ready by today. See you later!

**Review the new unit test**

Andy's latest feature involves the leaderboard. He needs to get the number of scores from the database, so he decides to write a unit test to verify the IDocumentDBRepository`1.GetItemsAsync method.

Here's what the test looks like. You don't need to add any code just yet.

C#Copy

**[**TestCase(0, ExpectedResult=0)**]**

**[**TestCase(1, ExpectedResult=1)**]**

**[**TestCase(10, ExpectedResult=10)**]**

publicintReturnRequestedCount**(**int **count)**

{

constint **PAGE = 0;** // take the first page of results

// Fetch the scores.

Task<IEnumerable<Score>> scoresTask = \_scoreRepository.GetItemsAsync(

**score =>** true**,** // return all scores

**score => 1,** // we don't care about the order

PAGE,

**count** // fetch this number of results

);

IEnumerable<Score> scores = scoresTask.Result;

// Verify that we received the specified number of items.

return **scores.Count();**

}

Recall that in an NUnit test, TestCase provides inline data to use to test that method. NUnit calls the ReturnRequestedCount unit test method like this:

C#

**ReturnRequestedCount(0);**

**ReturnRequestedCount(1);**

**ReturnRequestedCount(10);**

This test also uses the ExpectedResult property to simplify the test code and help make its intention clear. NUnit automatically compares the return value against the value of this property, removing the need to explicitly call the assertion.

Andy chooses a few values that represent typical queries. He also includes 0 to cover that edge case.

**Fetch the branch from GitHub**

As you did earlier, you fetch the failed-test branch from GitHub and check out, or switch to, that branch.

1. In Visual Studio Code, open the integrated terminal.
2. Run the following git fetch and git checkout commands to download a branch named failed-test from the Microsoft repository and switch to that branch:

**Bash**

**git fetch upstream failed-test**

**git checkout -b failed-test upstream/failed-test**

We named the branch failed-test for learning purposes. In practice, you would name a branch after its purpose or feature.

1. Run these commands to create a local tool manifest file, install the ReportGenerator tool, and add the coverlet.msbuild package to your tests project:

**Bash**

**dotnet new tool-manifest**

**dotnet tool install dotnet-reportgenerator-globaltool**

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

You need this step because the failed-test branch does not contain the work you added to the unit-tests branch.

1. Add your test project file and your tool manifest file to the staging index and commit your changes.

**Bash**

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

**git add .config/dotnet-tools.json**

**git commit -m "Configure code coverage tests"**

1. Run the following git push command to upload the failed-test branch to your GitHub repository:

**Bash**

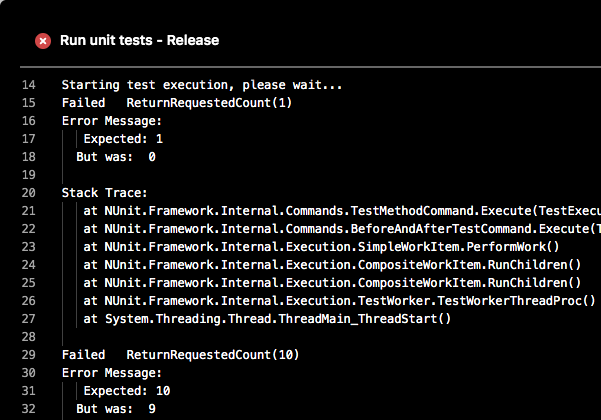
**git push origin failed-test**

**See the test failure in the pipeline**

Let's say that Andy was in a hurry and pushed up his work without running the tests one final time. Luckily, the pipeline can help you catch issues early when there are unit tests. You can see that here.

1. In Azure Pipelines, trace the build as it runs through the pipeline.
2. Expand the **Run unit tests - Release** task as it runs.

You see that the ReturnRequestedCount test method fails.



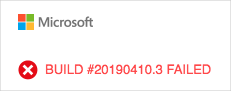
The test passes when the input value is 0, but it fails when the input value is 1 or 10.

The build is published to the pipeline only when the previous task succeeds. Here, the build wasn't published because the unit tests failed. This prevents others from accidentally obtaining a broken build.

In practice, you won't always manually trace the build as it runs. Here are a few ways you might discover the failure:

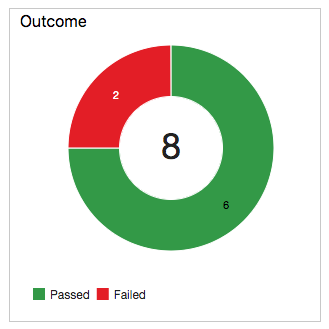
* **An email notification from Azure DevOps**

You can configure Azure DevOps to send you an email notification when the build is complete. The subject line starts with "[Build failed]" when the build fails.



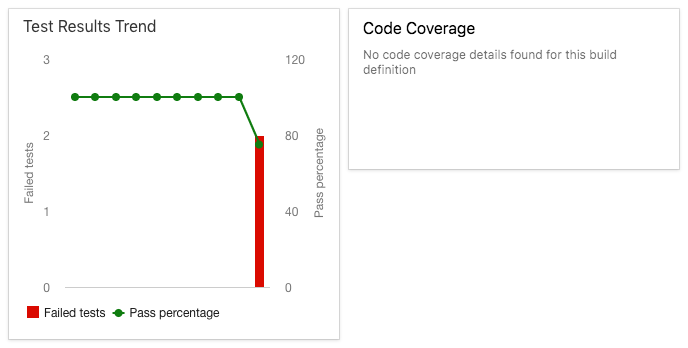
* **Azure Test Plans**

In Azure DevOps, select **Test Plans**, and then select **Runs**. You see the recent test runs, including the one that just ran. Select the latest completed test. You see that two of the eight tests failed.



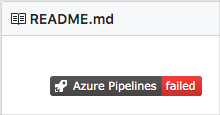
* **The dashboard**

In Azure DevOps, select **Overview**, and then select **Dashboards**. You see the failure appear in the **Test Results Trend** widget. The **Code Coverage** widget is blank, which indicates that code coverage was not run.



* **The build badge**

Although the failed-test branch doesn't include the build badge in the *README.md* file, here's what you would see on GitHub when the build fails:



**Analyze the test failure**

When unit tests fail, you ordinarily have two choices, depending on the nature of the failure:

* If the test reveals a defect in the code, you fix the code and rerun the tests.
* If the functionality changes, adjust the test to match the new requirements.

Mara notices the build failure and checks in with Andy.

**Mara:** Hey, Andy. I just saw the email notification in my inbox. It looks like the build is broken.

**Andy:** Yes, I just saw that as well. I already started to investigate. It looks like we have a failing unit test. Would you mind taking a look with me?

**Mara:** Sure, let's take a look. I say we start by verifying that we can reproduce the failure on your computer.

**Reproduce the failure locally**

In this section, you reproduce the failure locally, just like Mara and Andy.

1. In Visual Studio Code, open the integrated terminal.
2. In the terminal, run this dotnet build command to build the application:

**Bash**

**dotnet build --configuration Release**

1. In the terminal, run this dotnet test command to run the unit tests:

**Bash**

**dotnet test --no-build --configuration Release**

You see the same errors that you saw in the pipeline. Here's part of the output:

**OutputCopy**

X ReturnRequestedCount(10) [6ms]

Error Message:

Expected: 10

But was: 9

Stack Trace:

at NUnit.Framework.Internal.Commands.TestMethodCommand.Execute(TestExecutionContext context)

at NUnit.Framework.Internal.Commands.BeforeAndAfterTestCommand.Execute(TestExecutionContext context)

at NUnit.Framework.Internal.Execution.SimpleWorkItem.PerformWork()

at NUnit.Framework.Internal.Execution.CompositeWorkItem.RunChildren()

at NUnit.Framework.Internal.Execution.CompositeWorkItem.RunChildren()

at NUnit.Framework.Internal.Execution.TestWorker.TestWorkerThreadProc()

at System.Threading.Thread.ThreadMain\_ThreadStart()

Test Run Failed.

Total tests: 8

Passed: 6

Failed: 2

Total time: 1.3882 Seconds

**Find the cause of the error**

Mara notices that each failed test produces a result that's off by one. For example, when 10 is expected, the test returns 9.

Mara and Andy look at the source code for the method that's being tested, LocalDocumentDBRepository`1.GetItemsAsync. They see this:

C#Copy

public **Task<IEnumerable<T>> GetItemsAsync(**

**Expression<Func<T,** bool**>> queryPredicate,**

**Expression<Func<T,** int**>> orderDescendingPredicate,**

int **page = 1,** int **pageSize = 10**

)

{

var **result = \_items.AsQueryable()**

**.Where(queryPredicate)** // filter

**.OrderByDescending(orderDescendingPredicate)** // sort

**.Skip(page \* pageSize)** // find page

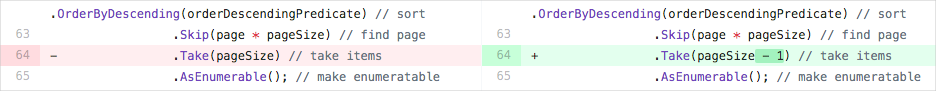
**.Take(pageSize - 1)** // take items

**.AsEnumerable();** // make enumeratable

return **Task<IEnumerable<T>>.FromResult(result);**

}

They examine the file on GitHub and notice that it was recently changed.



Mara suspects that pageSize - 1 is returning one fewer results and that this should be just pageSize.

**Mara:** Andy, do you remember why you made this change?

**Andy:** I was experimenting with something and I must have forgotten to change it back. It looks like changing back to the original code will fix things.

Mara and Andy decide to change the code back to its original state and then verify that the unit tests pass.

**Tip**

Discussion and collaboration can also happen on GitHub. You can comment on a pull request or open an issue.

**Fix the error**

In this section, you fix the error by changing the code back to its original state and running the tests to verify the fix.

1. In Visual Studio Code, open *Tailspin.SpaceGame.Web/LocalDocumentDBRepository.cs* from the file explorer.
2. Modify the GetItemsAsync method as shown here:

**C#Copy**

public **Task<IEnumerable<T>> GetItemsAsync(**

**Expression<Func<T,** bool**>> queryPredicate,**

**Expression<Func<T,** int**>> orderDescendingPredicate,**

int **page = 1,** int **pageSize = 10**

)

{

var **result = \_items.AsQueryable()**

**.Where(queryPredicate)** // filter

**.OrderByDescending(orderDescendingPredicate)** // sort

**.Skip(page \* pageSize)** // find page

**.Take(pageSize)** // take items

**.AsEnumerable();** // make enumeratable

return **Task<IEnumerable<T>>.FromResult(result);**

}

This version changes pageSize - 1 to pageSize.

1. Save the file.
2. In the integrated terminal, build the application.

**Bash**

**dotnet build --configuration Release**

You see that the build succeeds.

In practice, you might run the app and briefly try it out. For learning purposes, we'll skip that for now.

1. In the terminal, run the unit tests.

**Bash**

**dotnet test --no-build --configuration Release**

You see that the tests pass.

**Output**

Starting test execution, please wait...

Test Run Successful.

Total tests: 8

Passed: 8

Total time: 1.2506 Seconds

1. In the integrated terminal, add each modified file to the index, commit the changes, and push the branch up to GitHub.

**Bash**

**git add .**

**git commit -m "Return correct number of items"**

**git push origin failed-test**

**Tip**

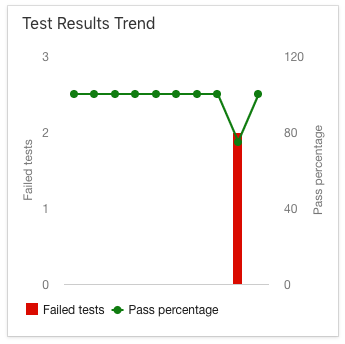
The dot (.) in this git add example is a wildcard character. It matches all unstaged files in the current directory and all subdirectories.

Before you use this wildcard character, it's a good practice to run git status to ensure that you're staging the files you intend to stage.

1. Return to Azure Pipelines. Watch the change move through the pipeline. The tests pass, and the overall build succeeds.

Optionally, to verify the test results, you can select the **Tests** and **Code Coverage** tabs when the build completes.

You can also check out the dashboard to view the updated results trend.



**Andy:** Great! We fixed the build! I'm sorry for breaking it. I was in a hurry and I forgot to run the tests one final time.

**Mara:** That's OK. The fix was easy enough. And we caught it early, *way* before it reached QA or production. Now Amita has a clean build to work from.