Ready to innovate?

The Visual COBOL 5.0 Azure DevOps and Serverless Computing Walkthrough

June 2019

Visual COBOL 5.0—blue sky thinking

Ready to build your Cloud story?

This is primarily a how-to technical guide that enables COBOL and non-COBOL developers to modernize legacy applications using the Cloud—it's all about bridging the old with the new.

New to Visual COBOL? This is your guided tour of everything it can do towards modernizing core COBOL applications. **Already on board?** This is the update that explains how to take your applications beyond the next level and on to the Cloud.

What will you learn?
Much of this Guide focuses on the technical, practical aspect of creating next-gen apps from COBOL code. Among other new skills, you will discover how to…

- Bring a COBOL application into Visual Studio or Eclipse
- Edit, compile and debug COBOL applications using the IDE
- Modernize COBOL apps using .NET and C#
- Create and deploy a COBOL microservice as a Serverless application in the Cloud
- Build, test and publish your application via a DevOps pipeline
- Understand the latest native Cloud technologies

What's new in Visual COBOL 5.0?

This latest update of our unrivalled development experience significantly extends Visual COBOL's capabilities. It brings the Cloud closer, enabling access to DevOps and Serverless computing for COBOL systems.

5.0

For Micro Focus, Visual COBOL 5.0 is where meet our customers' need for application modernization using the Cloud. It's where the tools within Visual COBOL 5.0 help you deliver innovation into the hands of your customer, that much faster.

Let's talk tooling.

Micro Focus Visual COBOL is a family of COBOL application development tools. They provide the advanced editing and debugging features within Visual Studio and Eclipse. This solution enables developers to modernize COBOL-based applications across Windows and Linux, including .NET, JVM and Docker container and Cloud platforms. More here.

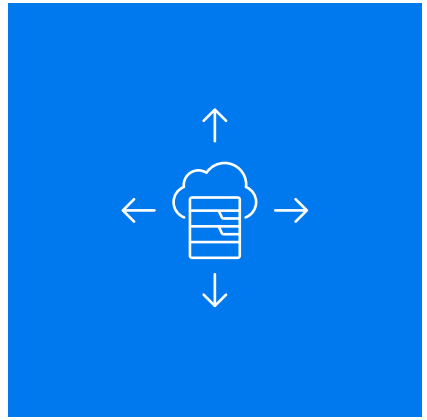
Developers can target a broad range of platforms using the Visual COBOL compiler, including Common Intermediate Language, the basis for .NET.

Run COBOL applications in .NET and take advantage of the .NET framework APIs and simplify integration with other .NET languages, such as C#.

Visual COBOL also includes a complete object oriented syntax, streamlined for .NET. Procedural COBOL is also supported and makes it possible to take your existing COBOL applications into .NET. More here.

A quick word on serverless computing...

Imagine a time when the code changes you made yesterday are in the hands of your customers today. Where deployment is as simple as the click of a button and applications automatically scale to meet the needs of peak demand.



Serverless computing is the next innovation in public Cloud. Automatically deploying and managing your applications - you can keep focused on the job of writing software.

. . . and Azure DevOps

This range of software tools, hosted in the Azure cloud, can accelerate software delivery. The tools include:

Boards

Agile planning and monitoring tools

Repos

Configuration management systems

Pipelines

Continuous Integration and Deployment automation

More here

Who can use this Guide?

Anyone with programming skills, in any language, primarily those working in COBOL, C# and .NET

While COBOL programmers will build on their current capabilities, because COBOL is so easy to learn, those beginning from a low base will soon be coding with confidence.

Do I need Visual COBOL to use it?

Yes. Download a trial from the Cloud, Azure, AWS or here. New to Visual COBOL? Check out these tutorials. And if you ever need help, go straight to the mothership. It takes two minutes to register for the Visual COBOL forum of the Micro Focus Community website—and no time at all to get your question answered…

Let's do this.

Your kit list

Visual COBOL to compile and run the COBOL application. No license? No problem. Download the trial version from here.

Visual Studio 2017 or 2019 to create and edit the COBOL and C# application code. Trial the professional version—the license covers the trial—or use the free community edition.

An Azure subscription to deploy your application to the Cloud and an Azure DevOps account. Sign up for free Azure credits.

Now, let's build a COBOL microservice in .Net

We're going to extract the business logic from a sample COBOL application - a simple green-screen loan calculator – to use an API.

Step 1 for us today is to **download the COBOL source code.**

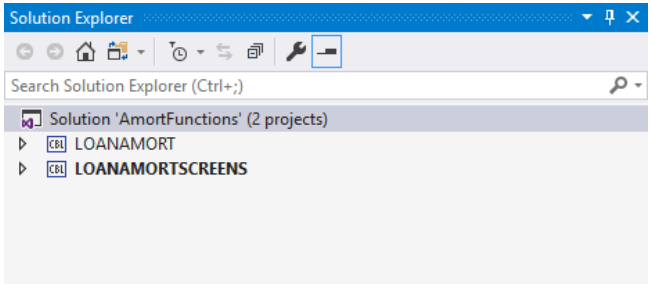
Download the program source code, to a temporary location, as a Zip file from here. Each folder has a different part of the **COBOL LoanCalc Application.**

Step 2 is to **understand the source code**

We'll run it as a standalone application and use Visual Studio to compile, run and debug it.

To **open the Visual Studio solution,** browse to the source code folder and double click the AmortFunctions.sln file. When it opens in Visual Studio, make the Solution Explorer window visible.

Of these two COBOL projects, **LOANAMORT** is the main code. It processes loan payment schedules. **LOANAMORTSCREENS** is the console-based user interface.



Open the **LOANAMORT.cbl** file within the LOANAMORT project. It's a simple program that calculates a payment schedule based on three factors:

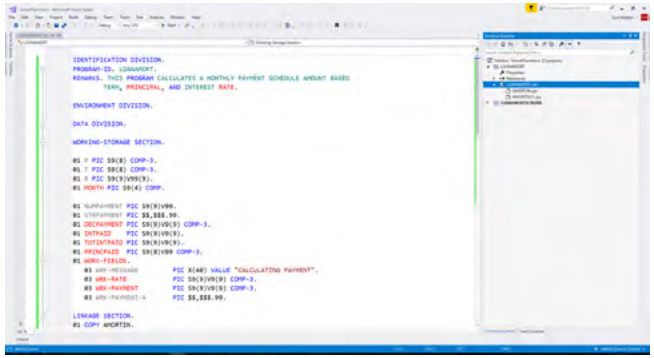
PRINCIPAL – the amount to borrow

LOANTERM – the duration of the loan in months

RATE – the interest charged during the loan term

The program data is an array denoting the monthly payment schedule, and the total amount paid.

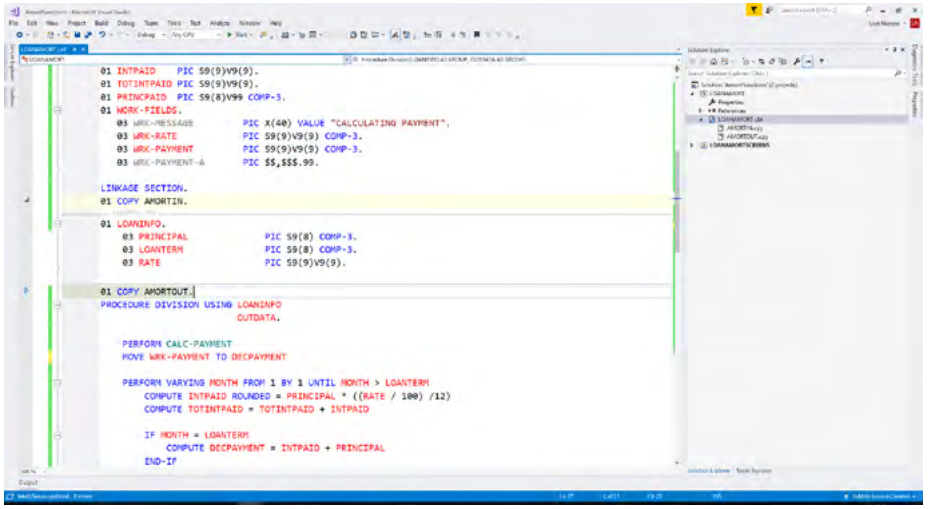
The **LOANAMORTSCREENS** application provides the user interface.

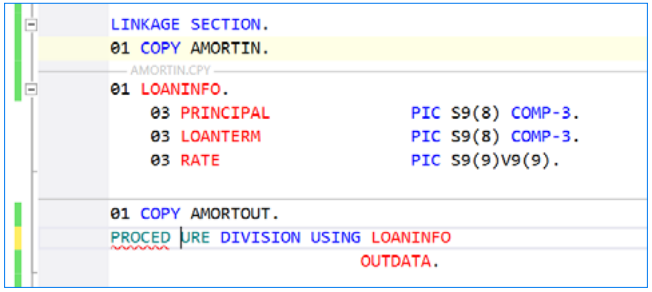


A quick note about Visual Studio for COBOL development

Visual Studio has bags of features for COBOL development. Here's just a couple to get you started:

- *Expand the program in the solution explorer to view the program's copybooks within the fully functional COBOL editor in Visual Studio*
- *Keywords and data items are colorized*
- *Click the arrow in the margin next to a COPY statement to see copybook contents*
- *Hover over program fields to see information about their type and usage*
- *The editor compiles your code in the background and flags up mistakes with a red squiggle. Give it a go by inserting a deliberate coding error, but don't forget to undo your change—**ctrl-z***





Next—compile both projects using **Build->Build Solution** from the menu. Make sure it's error free by checking the Output window

```
       LINKAGE SECTION.
       01 COPY AMORTIN.
      ---------- AMORTIN.CPY ----------
       01 LOANINFO.
           03 PRINCIPAL                PIC S9(8) COMP-3.
           03 LOANTERM                 PIC S9(8) COMP-3.
           03 RATE                     PIC S9(9)V9(9).

       01 COPY AMORTOUT.
       PROCEDURE DIVISION USING LOANINFO
                                OUTDATA.
```

```
Output
Show output from: Build
1>------ Build started: Project: LOANAMORT, Configuration: Debug Any CPU ------
1>  * Compiling "C:\AmortLoan - Orig\LOANAMORT\LOANAMORT.cbl"
1>  * Generating LOANAMORT
1>  The project has been built with 'ilsmartlinkage' compile directive.
1>  LOANAMORT -> C:\AmortLoan - Orig\LOANAMORT\bin\Debug\LOANAMORT.dll
2>------ Build started: Project: LOANAMORTSCREENS, Configuration: Debug Any CPU ------
2>  * Compiling "C:\AmortLoan - Orig\LOANAMORTSCREENS\LOANAMORTSCREENS.cbl"
2>  * Generating LOANAMORTSCREENS
2>  LOANAMORTSCREENS -> C:\AmortLoan - Orig\LOANAMORTSCREENS\bin\Debug\LOANAMORTSCREENS.exe
========== Build: 2 succeeded or up-to-date, 0 failed, 0 skipped ==========
Output
```

Let's run and debug the program

- Right-click the **LOANAMORTSCREENS** project. Choose **Set** as **Startup Project**
- Press **F5** to run and debug the application and follow the on screen instructions
- Press **CTRL-C** to stop the application

```
c:\amortloan - orig\loanamortscreens\bin\debug\LOANAMORTSCREENS.exe
********************************************************************************
* LOAN AMORT                                                                   *
********************************************************************************

PRINCIPAL : 00001000
LOAN TERM :  60
RATE      :  3.50



PAYMENT #001 TOTAL    $18.19 INT     $2.91 PRINCIPAL    $15.27
PAYMENT #002 TOTAL    $18.19 INT     $2.87 PRINCIPAL    $15.31
PAYMENT #003 TOTAL    $18.19 INT     $2.82 PRINCIPAL    $15.36
PAYMENT #004 TOTAL    $18.19 INT     $2.78 PRINCIPAL    $15.40
PAYMENT #005 TOTAL    $18.19 INT     $2.74 PRINCIPAL    $15.45
PAYMENT #006 TOTAL    $18.19 INT     $2.69 PRINCIPAL    $15.49
PAYMENT #007 TOTAL    $18.19 INT     $2.65 PRINCIPAL    $15.53
PAYMENT #008 TOTAL    $18.19 INT     $2.60 PRINCIPAL    $15.58
PAYMENT #009 TOTAL    $18.19 INT     $2.56 PRINCIPAL    $15.63
FINAL PAYMENT:
PAYMENT #00000060 TOTAL    $19.05 INT      $.05 PRINCIPAL    $19.00
TOTAL INTEREST    $91.65
```

Let's debug the code.

Terminate the application if it is running and press **F11** to step the through the code a few lines

Now, hover over fields to examine their values. Want to set a breakpoint in a line of the LOANAMORT program code? Press **F9** where you want the debugger to stop

Press **F5** to resume running the application. It should stop at your breakpoint. Stop debugging the code.

Step 3 is to **create an API.**

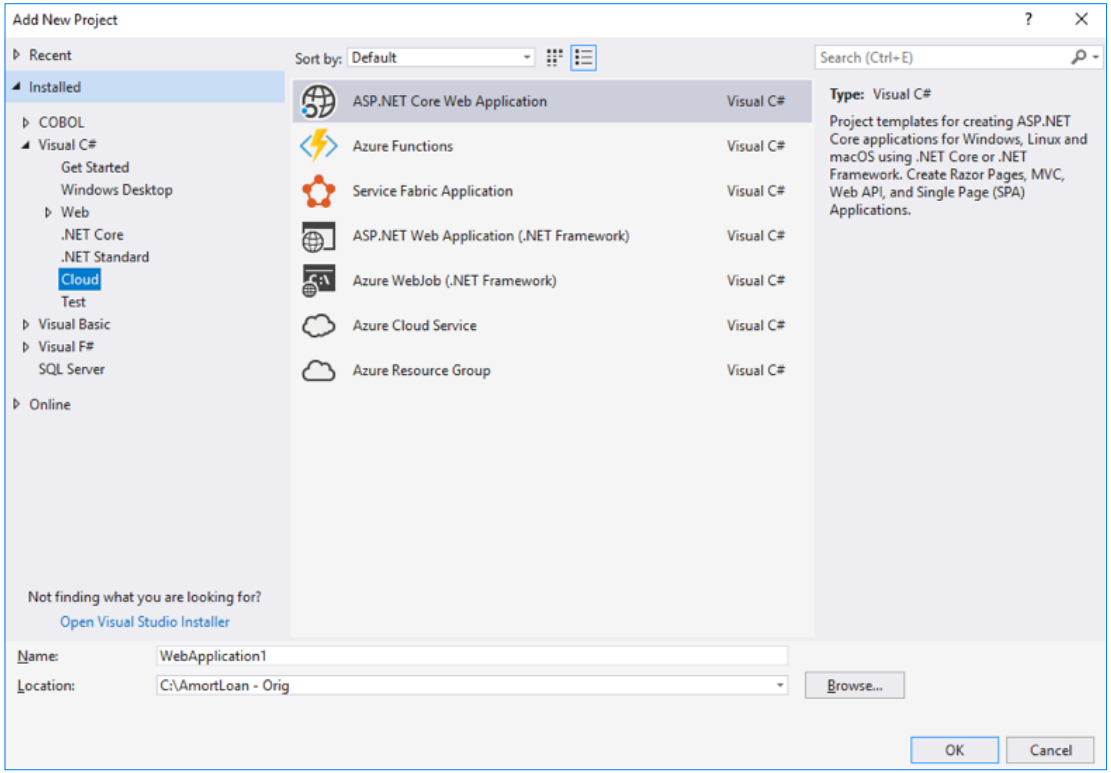
We're a step closer to building an API than you think. The Visual COBOL compiler is creating a .NET executable enabling easy integration for COBOL applications with C#.

So let's create a C# project providing the entry point for an API that will call COBOL to do the loan calculations.

While you don't need an Azure subscription you may need to install some Azure tools into Visual Studio as we progress. Here's a quick hack for checking you have Azure support.

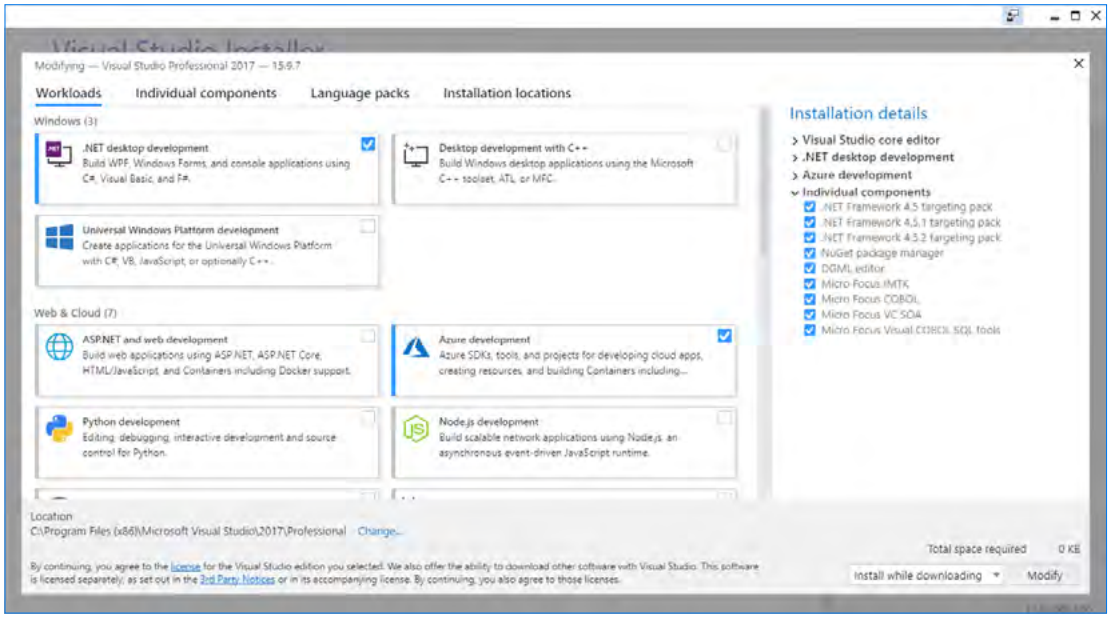
Right click the solution item in the Solution Explorer and add-> New Project before selecting **Visual C# templates.** Expand to see the full list.

Look for the project templates beneath the 'Cloud' heading. If is **not** as shown in the next image, you'll need to install **Azure Workload Support** into Visual Studio.



Installing Azure workload support into Visual Studio

It's easy. Click **Modify** in the Visual Studio installer and **tick the Azure Developments** workload option, and update Visual Studio. You may need to restart Visual Studio when you're done – re-open the solution when finished.



Checking you have Azure support installed - do you see Cloud templates?

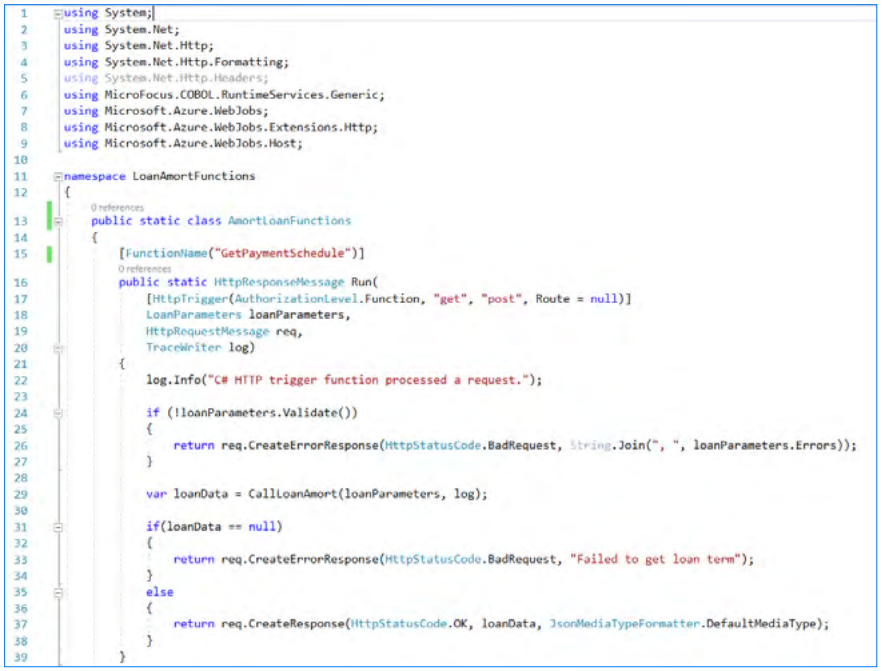
Add the C# API

We've already created the C# project to get you started. Add it to your solution. Right-click your solution in the Solution Explorer and choose add->Existing Project.

Open the **LoanAmortFunctions** project folder, select **LoanAmortFunctions.csproj** and click **Open.** Spotted an error? You may need to install Azure Workload support.

Once you have built the solution, check for errors before moving onto the next steps.

Let's take a look at the C# code.



```
1  using System;
2  using System.Net;
3  using System.Net.Http;
4  using System.Net.Http.Formatting;
5  using System.Net.Http.Headers;
6  using MicroFocus.COBOL.RuntimeServices.Generic;
7  using Microsoft.Azure.WebJobs;
8  using Microsoft.Azure.WebJobs.Extensions.Http;
9  using Microsoft.Azure.WebJobs.Host;
10
11 namespace LoanAmortFunctions
12 {
       0 references
13     public static class AmortLoanFunctions
14     {
15         [FunctionName("GetPaymentSchedule")]
           0 references
16         public static HttpResponseMessage Run(
17             [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)]
18             LoanParameters loanParameters,
19             HttpRequestMessage req,
20             TraceWriter log)
21         {
22             log.Info("C# HTTP trigger function processed a request.");
23
24             if (!loanParameters.Validate())
25             {
26                 return req.CreateErrorResponse(HttpStatusCode.BadRequest, String.Join(", ", loanParameters.Errors));
27             }
28
29             var loanData = CallLoanAmort(loanParameters, log);
30
31             if(loanData == null)
32             {
33                 return req.CreateErrorResponse(HttpStatusCode.BadRequest, "Failed to get loan term");
34             }
35             else
36             {
37                 return req.CreateResponse(HttpStatusCode.OK, loanData, JsonMediaTypeFormatter.DefaultMediaType);
38             }
39         }
```

Open the **AmortLoanFunctions.cs** file in the C# **LoanAmortFunctions** project. It'll look like this:

- Line 13: The class that handles the API
- Line 15: The name given to our API
- Line 16: The method executed when a client calls the API
- Line 18: loanParameters contains the loan amount, rate and term. Values passed in on the URL are placed into this data item
- Line 24: Checks the parameters are correct
- Line 29: This is where the call to the COBOL program takes place
- Line 37: If all is correct, we return the payment schedule in JSON format

Open the **LoanParameters.cs** file in the C# **LoanAmortFunctions** project.

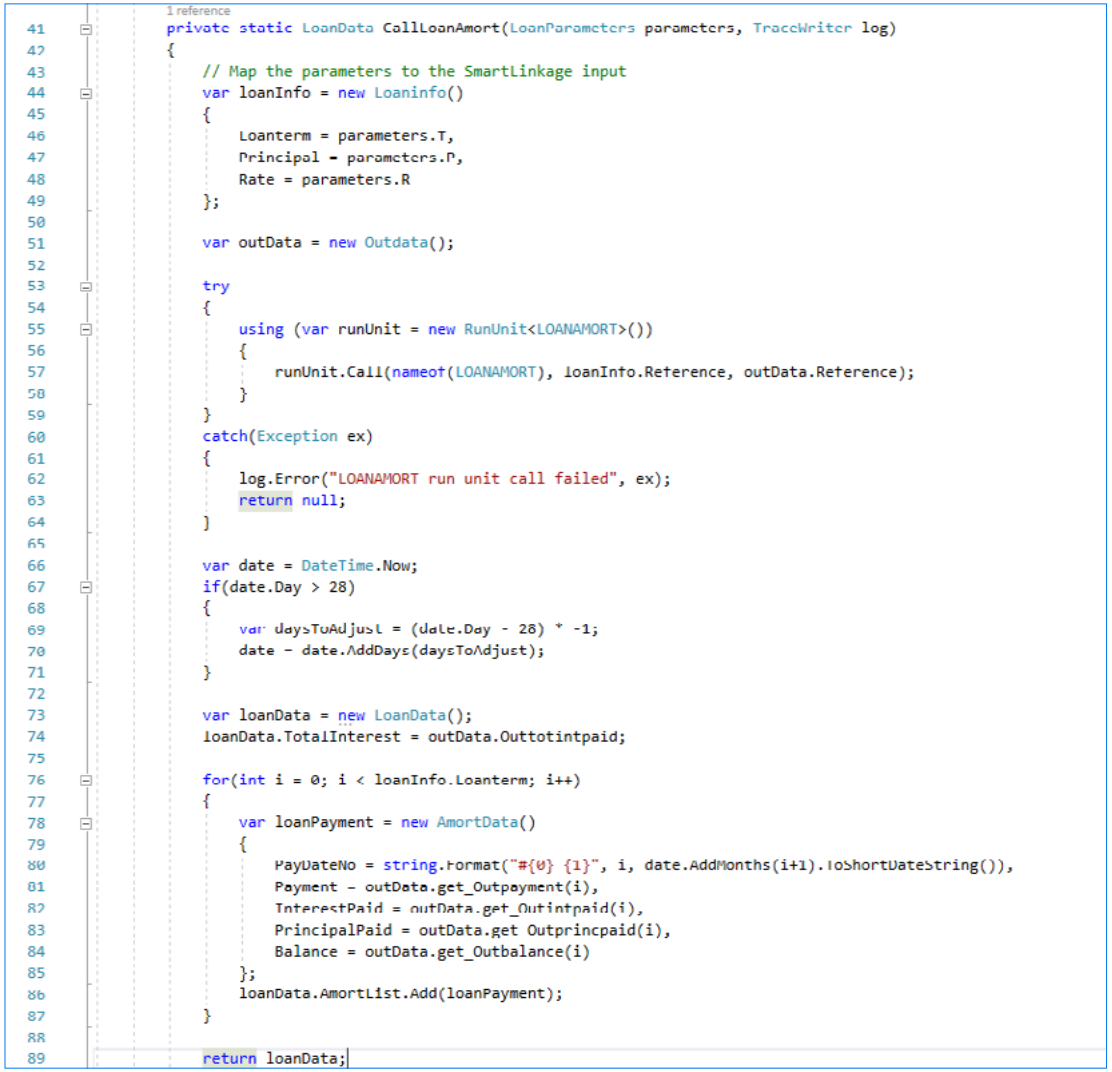
- Line 5: This is the input loan payment class declared in C#
- It contains fields for term (T), rate (R) and principal amount (P)
- The values of these fields will be referenced in the URL when we invoke the API
- Line 17: A helper method to verify the parameters

```
1   using System.Collections.Generic;
2
3   namespace LoanAmortFunctions
4   {
        3 references
5       public class LoanParameters
6       {
            2 references
7           public int P { get; set; }
            2 references
8           public int T { get; set; }
            1 reference
9           public decimal R { get; set; }
            6 references
10          public IList<string> Errors { get; }
11
            0 references
12          public LoanParameters()
13          {
14              Errors = new List<string>();
15          }
16
            1 reference
17          internal bool Validate()
18          {
19              Errors.Clear();
20
21              if (P < 1)
22                  Errors.Add("Principal must be greater than 0");
23
24              if (T < 1)
25                  Errors.Add("Term must be greater that 0 ");
26
27              return Errors.Count == 0;
28          }
29      }
30  }
```

Now, return to the open **AmortLoanFunctions.cs** file in the editor.

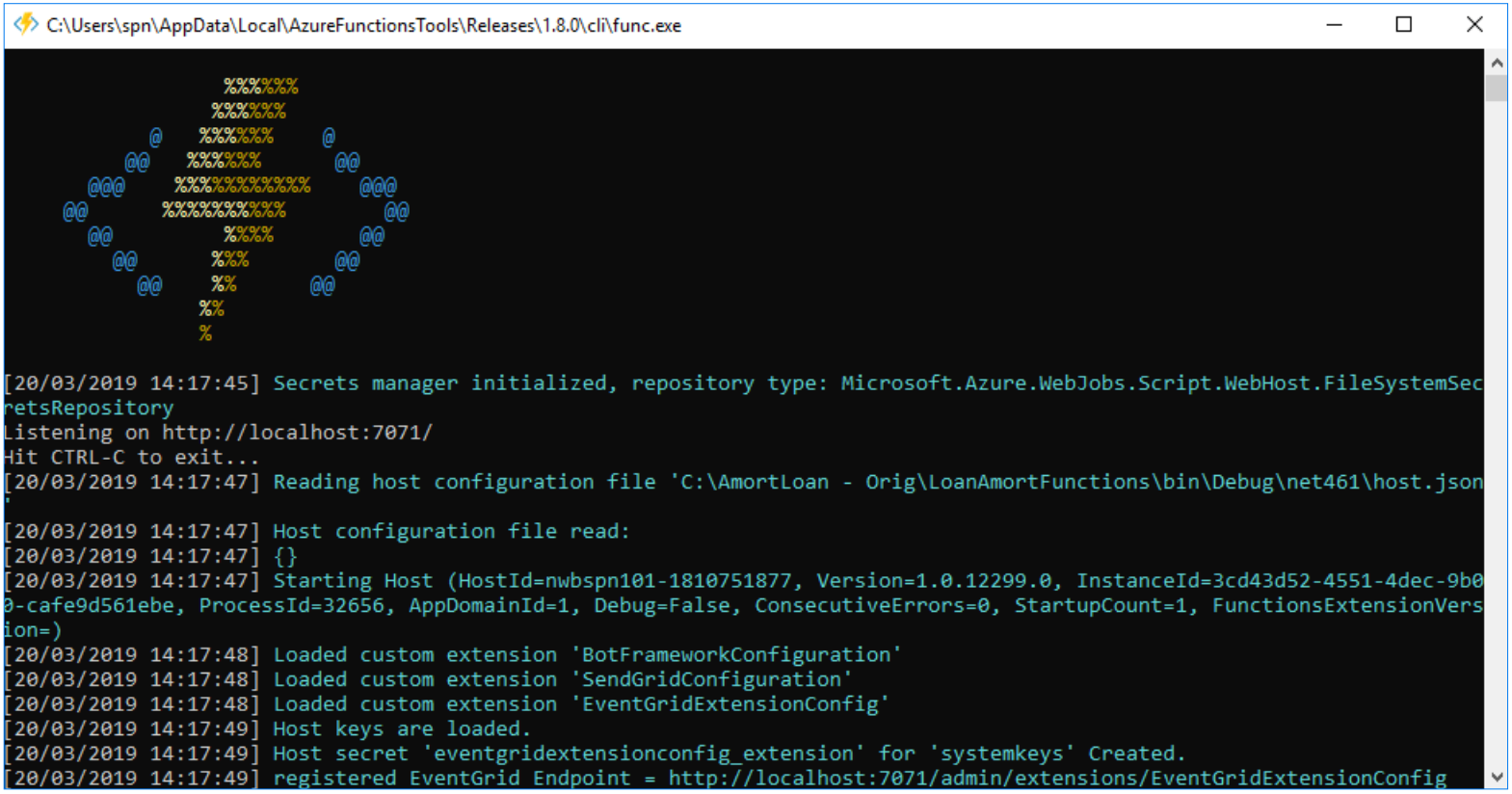
- Line 41: This method is called to process the API request
- Line 44: The parameters passed into the API are placed into a new data structure called ***loaninfo.***

  This data item corresponds to what the COBOL program expects to receive. The COBOL compiler created the ***Loaninfo*** class and it matches the parameters specified in the linkage section of the COBOL program.
- Line 51: **Outdata** is also a class generated by the compiler, and contains the output parameters returned by the COBOL program.
- Line 57: The call to the COBOL program happens in the **run unit,** a Micro Focus API for C# developers. The run unit isolates this call to the COBOL program, any data it uses and sub programs it calls into a single unit of work, separate from any other invocation of the program. Many clients can simultaneously call the API; so run units isolate each request without needing to adapt the COBOL program.
- Line 73: The result returned from the COBOL program is then converted into a **C# data structure.** The code iterates over every item in the array returned by COBOL, and formats it with extra information to show details of every monthly payment.

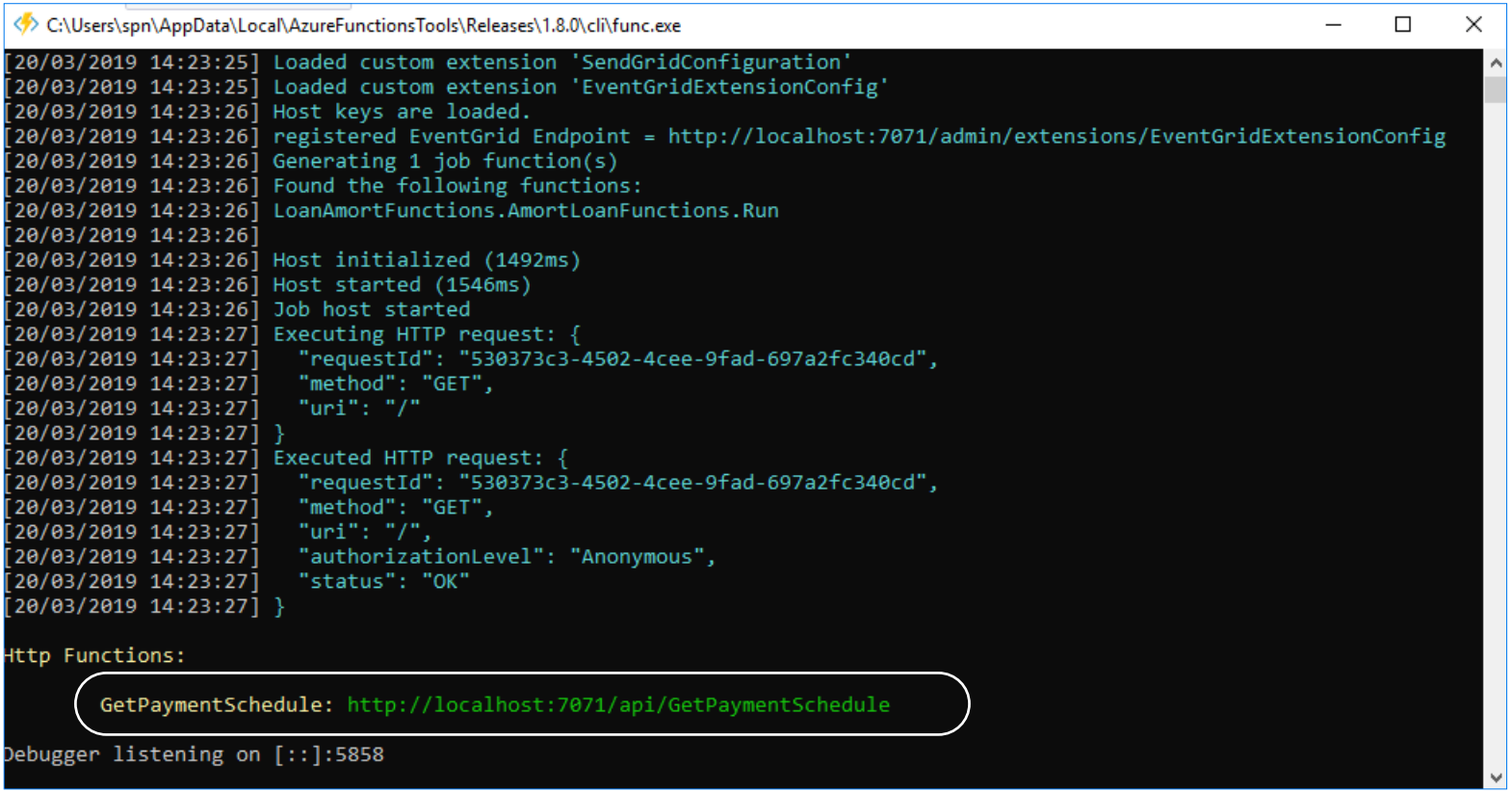


```
1 reference
41  private static LoanData CallLoanAmort(LoanParameters parameters, TraceWriter log)
42  {
43      // Map the parameters to the SmartLinkage input
44      var loanInfo = new Loaninfo()
45      {
46          Loanterm = parameters.T,
47          Principal = parameters.P,
48          Rate = parameters.R
49      };
50
51      var outData = new Outdata();
52
53      try
54      {
55          using (var runUnit = new RunUnit<LOANAMORT>())
56          {
57              runUnit.Call(nameof(LOANAMORT), loanInfo.Reference, outData.Reference);
58          }
59      }
60      catch(Exception ex)
61      {
62          log.Error("LOANAMORT run unit call failed", ex);
63          return null;
64      }
65
66      var date = DateTime.Now;
67      if(date.Day > 28)
68      {
69          var daysToAdjust = (date.Day - 28) * -1;
70          date = date.AddDays(daysToAdjust);
71      }
72
73      var loanData = new LoanData();
74      loanData.TotalInterest = outData.Outtotintpaid;
75
76      for(int i = 0; i < loanInfo.Loanterm; i++)
77      {
78          var loanPayment = new AmortData()
79          {
80              PayDateNo = string.Format("#{0} {1}", i, date.AddMonths(i+1).ToShortDateString()),
81              Payment = outData.get_Outpayment(i),
82              InterestPaid = outData.get_Outintpaid(i),
83              PrincipalPaid = outData.get Outprincpaid(i),
84              Balance = outData.get_Outbalance(i)
85          };
86          loanData.AmortList.Add(loanPayment);
87      }
88
89      return loanData;
```

You can run and debug the API locally, you don't need to deploy to Azure just yet. So make **LoanAmortFunctions** the start-up project. Once you **hit F5** to begin debugging, the Azure Emulator should fire up. Visual Studio may prompt you to install this software. Do you see a popup message from the Windows Defender Firewall? Click **Allow Access** to continue.



This is the Azure emulator starting up



Copy this URL

The C# function is listening on a port for a request. Let's test the function using a browser. So paste the URL from the Azure emulator into a browser window. Your browser will dictate the output, but it should look like this:

```
▼<Error>
  ▼<Message>
     Principal must be greater than 0, Term must be greater that 0
   </Message>
 </Error>
```

If we are going to get more meaningful results, we will need to pass some parameters into the URL.

Add the following parameters onto your URL: **?P=100&T=12&R=5**

P, T and R correspond to the Principal, Term and Rate parameters your application will use. Your complete URL will look something like this…

http://localhost:7071/api/GetPaymentSchedule?P=100&T=12&R=5

{"AmortList":[{"PayDateNo":"#0 20/04/2019","Balance":" $92.00","InterestPaid":"
$.41","PrincipalPaid":" $8.14","Payment":" $8.56"},{"PayDateNo":"#1
20/05/2019","Balance":" $84.00","InterestPaid":" $.38","PrincipalPaid":"
$8.17","Payment":" $8.56"},{"PayDateNo":"#2 20/06/2019","Balance":"
$76.00","InterestPaid":" $.35","PrincipalPaid":" $8.21","Payment":"
$8.56"},{"PayDateNo":"#3 20/07/2019","Balance":" $68.00","InterestPaid":"
$.31","PrincipalPaid":" $8.24","Payment":" $8.56"},{"PayDateNo":"#4
20/08/2019","Balance":" $60.00","InterestPaid":" $.28","PrincipalPaid":"
$8.27","Payment":" $8.56"},{"PayDateNo":"#5 20/09/2019","Balance":"
$52.00","InterestPaid":" $.25","PrincipalPaid":" $8.31","Payment":"
$8.56"},{"PayDateNo":"#6 20/10/2019","Balance":" $44.00","InterestPaid":"
$.21","PrincipalPaid":" $8.34","Payment":" $8.56"},{"PayDateNo":"#7
20/11/2019","Balance":" $36.00","InterestPaid":" $.18","PrincipalPaid":"
$8.37","Payment":" $8.56"},{"PayDateNo":"#8 20/12/2019","Balance":"
$28.00","InterestPaid":" $.15","PrincipalPaid":" $8.41","Payment":"
$8.56"},{"PayDateNo":"#9 20/01/2020","Balance":" $20.00","InterestPaid":"
$.11","PrincipalPaid":" $8.44","Payment":" $8.56"},{"PayDateNo":"#10
20/02/2020","Balance":" $12.00","InterestPaid":" $.08","PrincipalPaid":"
$8.47","Payment":" $8.56"},{"PayDateNo":"#11 20/03/2020","Balance":"
$.00","InterestPaid":" $.05","PrincipalPaid":" $12.00","Payment":"
$12.05"}],"TotalInterest":" $2.80"}

This is the payment schedule in JSON format returned from the COBOL program. This is what the browser displays in its raw form.

Now, it's time to **debug your work.**

With the emulator still running, set a breakpoint **F9** in the first line of the **C# run** method in **AmortLoanFunctions.cs** (It's on line 22.) Invoke the browser again and step the through the code, line-by-line

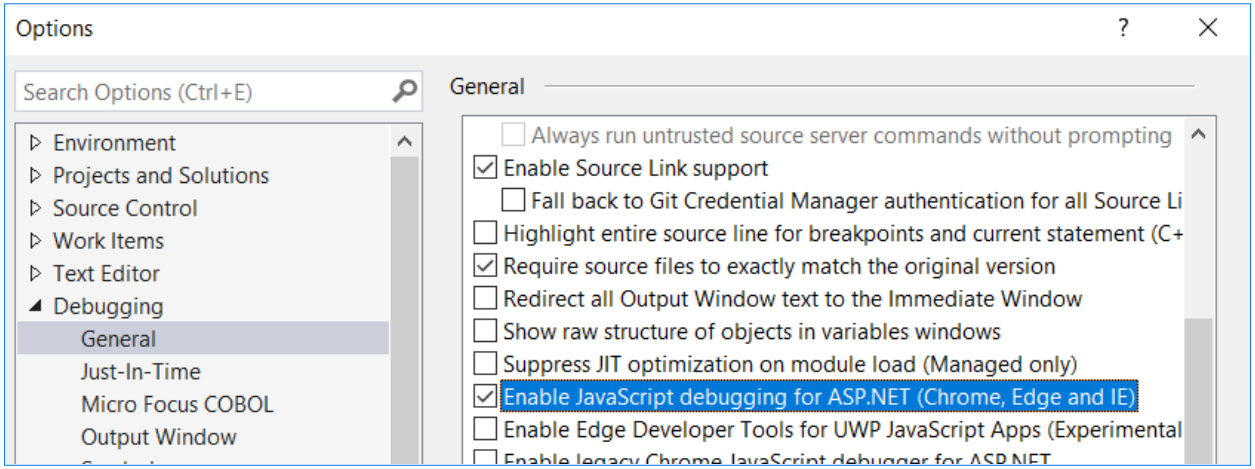
You may see a popup asking if you want to continue being notified of automatic step-overs. Click No.

Examine the parameters, step from **C#** into **COBOL** and back again. Stop debugging when you're finished and the Azure emulator Windows will close.

Now, we're going to **connect a modern user interface - a web browser client - to your API.**

This browser based UI is written in Javascript and you'll need to turn on Javascript debugging.

Select Debug→Options→General from the IDE menu.
Tick the box as shown below:



Use the **Add Existing project menu** to add the **LoanAmortWebUI** project to your solution. Select **LoanAmortWebUI.csproj** and click Open.

See an error message about 'Shared Web extensions failing'? Restart Visual Studio to resolve the issue.

The **API** and **Web Client** project must run together. So, right click the C# function project **LoanAmortFunctions** and select Debug->Start new instance to launch the Azure emulator.

Now, right click the UI project **LoanAmortWebUI**. Select Debug->Start new instance. This should launch a browser page.

Do you see an exception about part of the path being missing? Does it point to **bin\roslyn\csc.exe**?

From the **Visual Studio** menu select Tools→NuGet Package Manager→Pacakage Manager Console

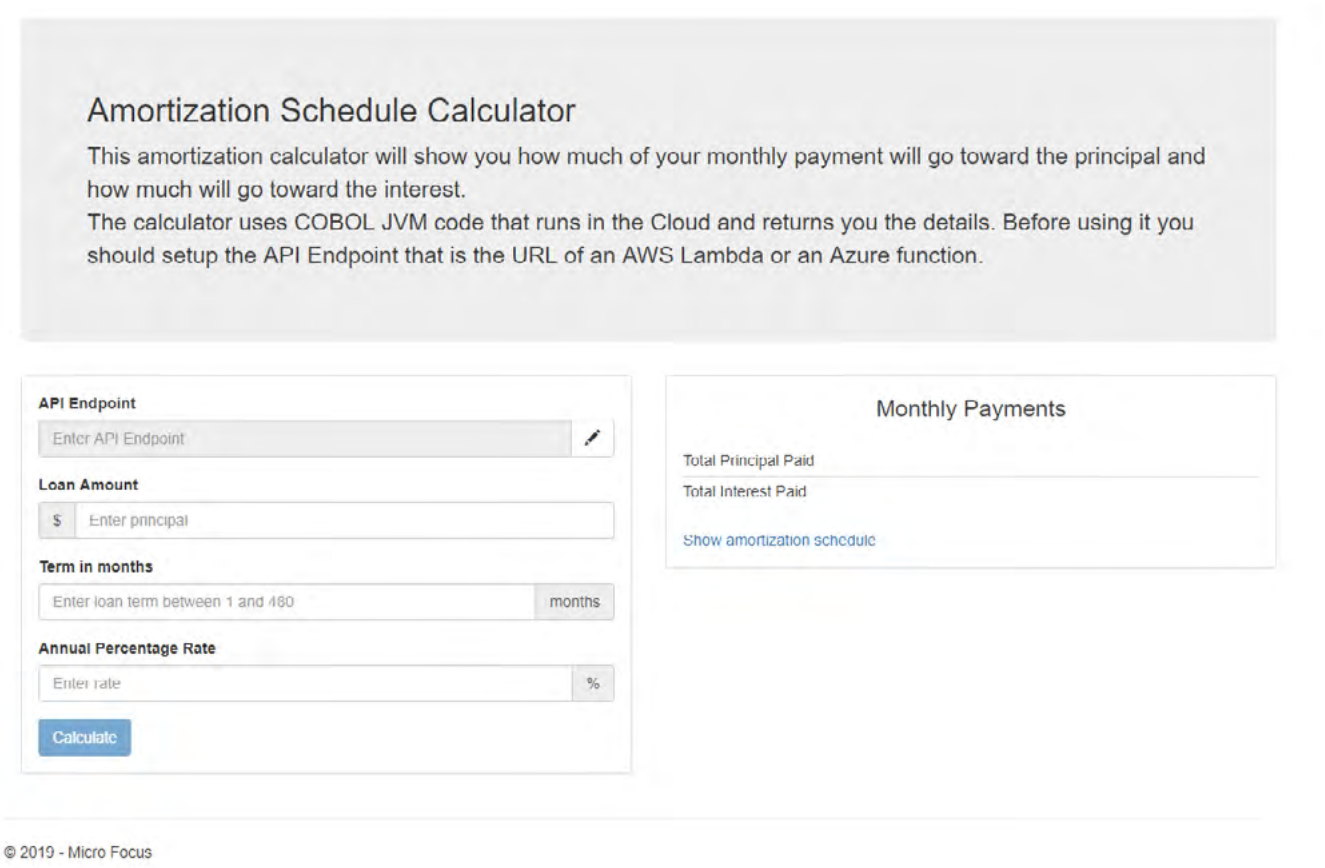
and run the following command:

PM> Update-Package Microsoft.CodeDom.Providers.DotNetCompilerPlatform–r

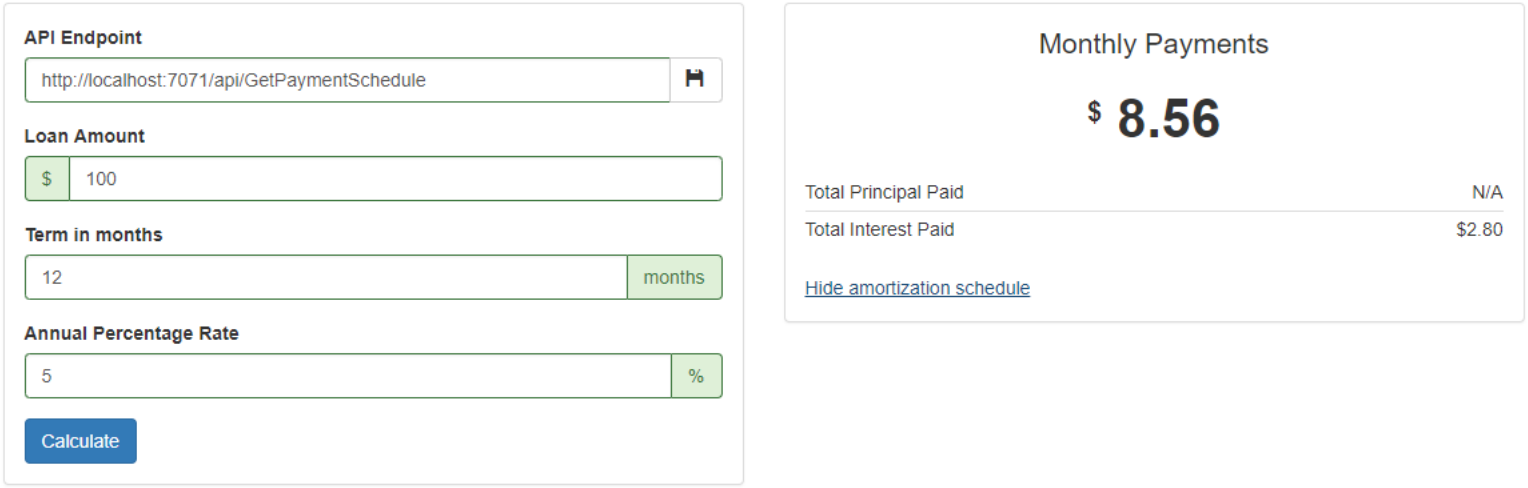
to add the necessary project support.

Relaunching the two projects will prompt a browser, presenting a web UI for your loan application. So, paste the URL from the Azure emulator into the **web browser end point field.** Enter your parameters into the browser page, set breakpoints in the C# code, and debug.

The new and improved web UI for the loan payment calculator



The payment schedule for the loan as returned by the COBOL application



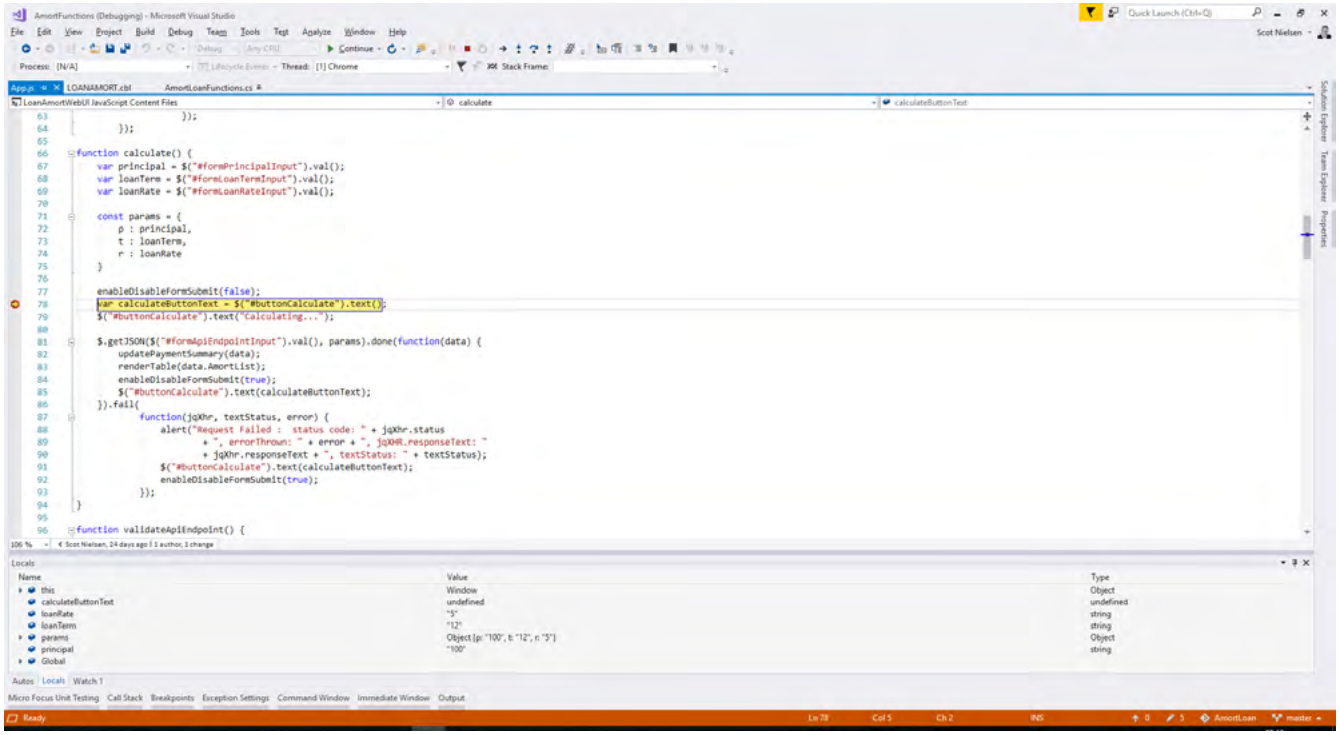
PayDateNo	Balance	InterestPaid	PrincipalPaid	Payment
#0 20/04/2019	$92.00	$.41	$8.14	$8.56
#1 20/05/2019	$84.00	$.38	$8.17	$8.56
#2 20/06/2019	$76.00	$.35	$8.21	$8.56
#3 20/07/2019	$68.00	$.31	$8.24	$8.56
#4 20/08/2019	$60.00	$.28	$8.27	$8.56
#5 20/09/2019	$52.00	$.25	$8.31	$8.56
#6 20/10/2019	$44.00	$.21	$8.34	$8.56
#7 20/11/2019	$36.00	$.18	$8.37	$8.56
#8 20/12/2019	$28.00	$.15	$8.41	$8.56
#9 20/01/2020	$20.00	$.11	$8.44	$8.56
#10 20/02/2020	$12.00	$.08	$8.47	$8.56
#11 20/03/2020	$.00	$.05	$12.00	$12.05

Your UI project is a browser-based application using Javascript to invoke the COBOL API. Check it out—open the **App.js file** in the scripts folder, scroll down to the **calculate()** function.

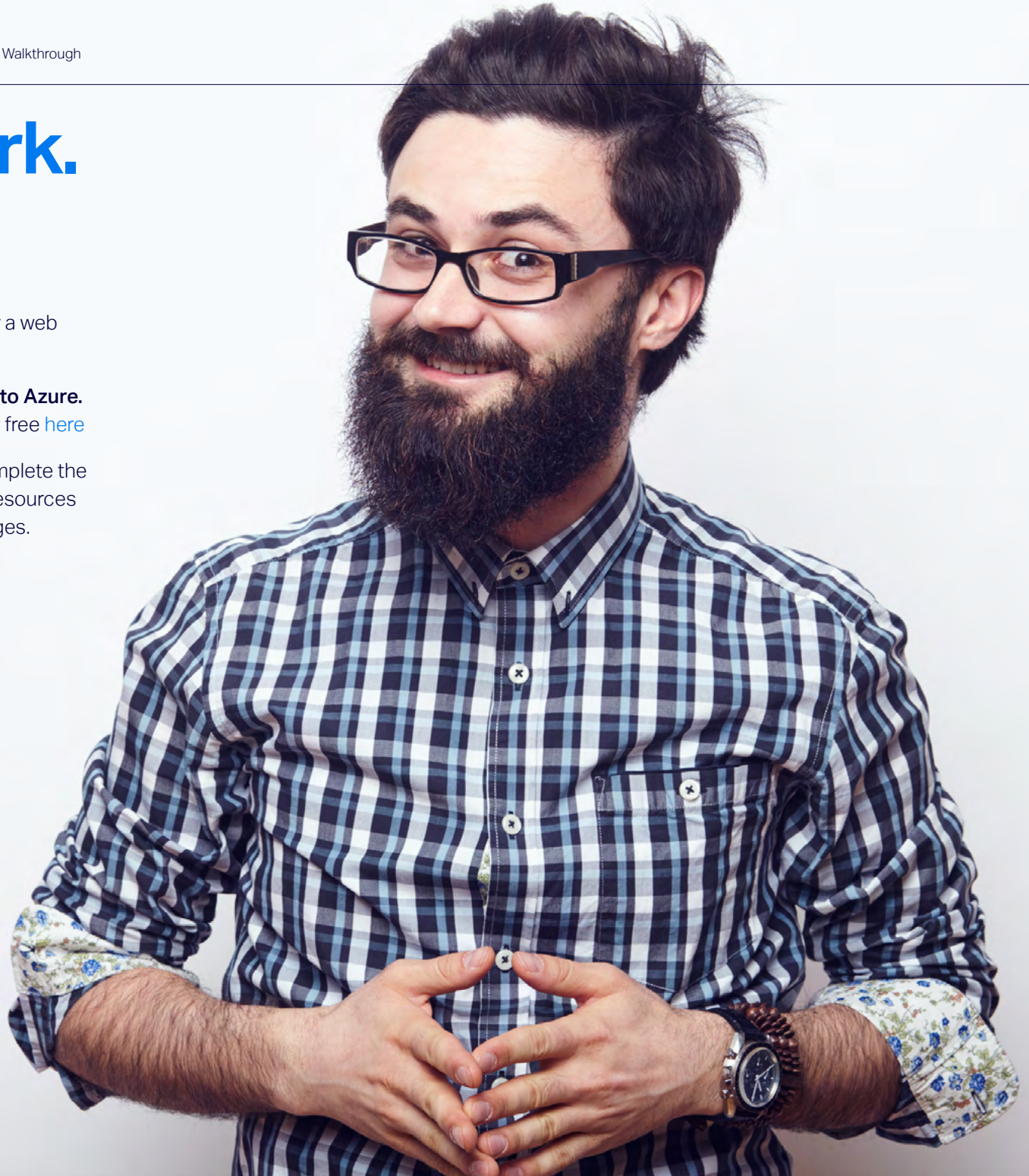
Set a breakpoint and debug through the Javascript to see how it obtains parameters from the form and sends them to the API.

If Javascript debugging is disabled, enable it. This will automatically restart the UI project. Re-enter the parameters in the web page.

Debugging Javascript on the client



Hey. Nice work.

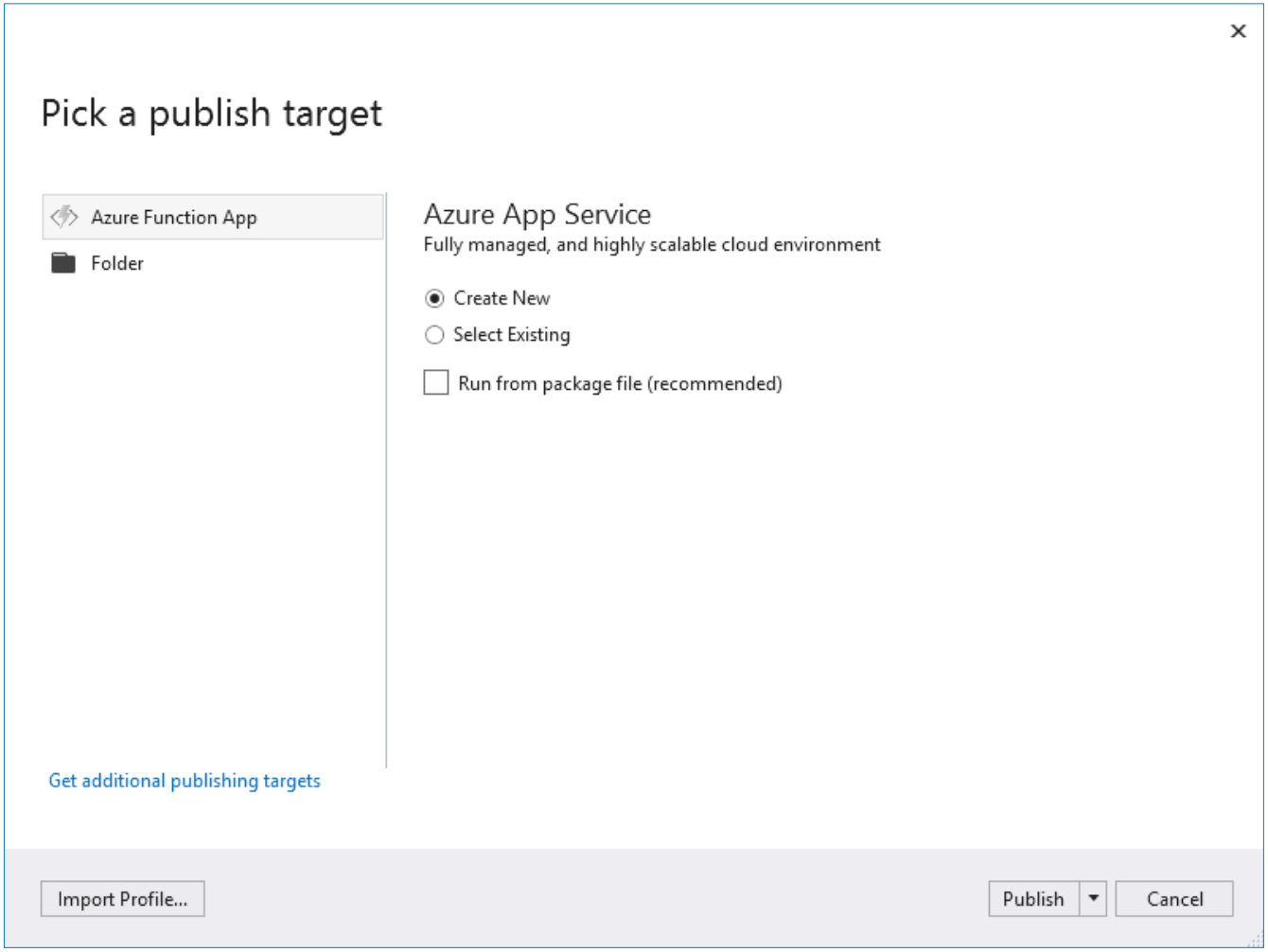


You have a COBOL program being invoked by a web browser using a JSON API. That's pretty cool.

But we can go further. Let's **publish your API to Azure.** Don't have an Azure subscription? Sign up for free here

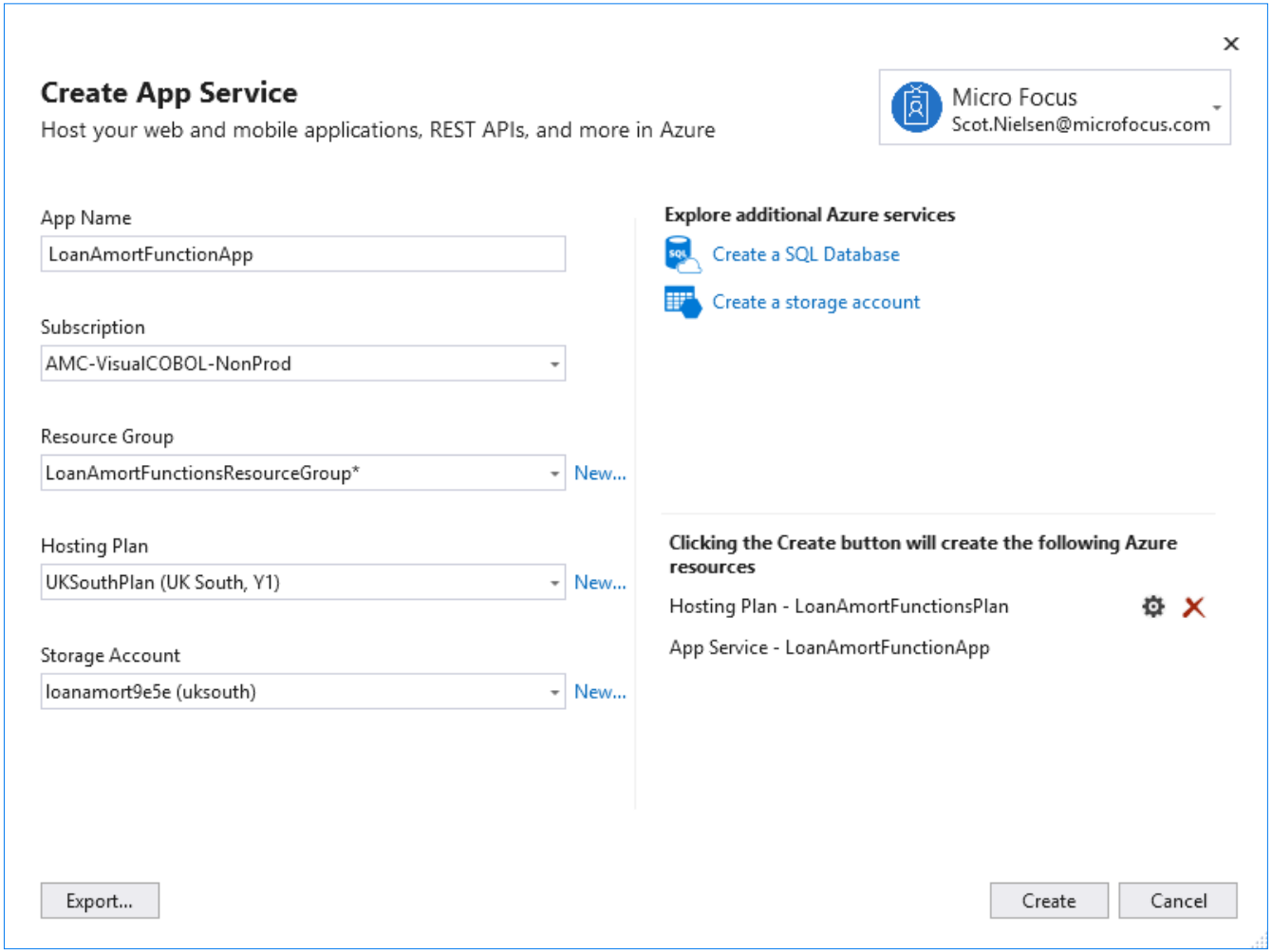
The free Azure tier should be sufficient to complete the walkthrough. But remember to shutdown all resources once you're done to avoid unnecessary charges.

The API can be published directly from Visual Studio. Right click the C# Functions Project **LoanAmortFunctions** and choose 'Publish'. Click the 'New Profile' link and you'll see this screen (overleaf). Select the options shown and click 'Publish'.

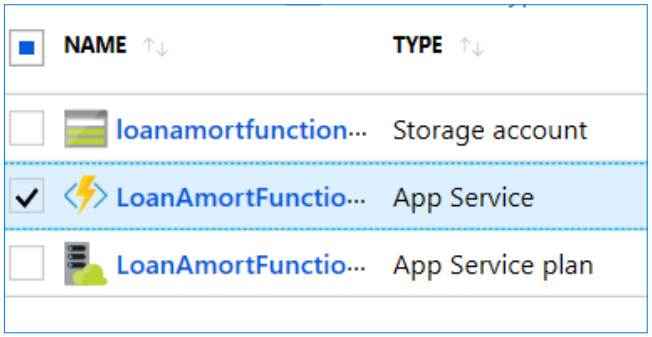


Use the ID you've used to connect to Visual Studio in the subscription. Your App Name needs to be unique, so you'll need to change this.

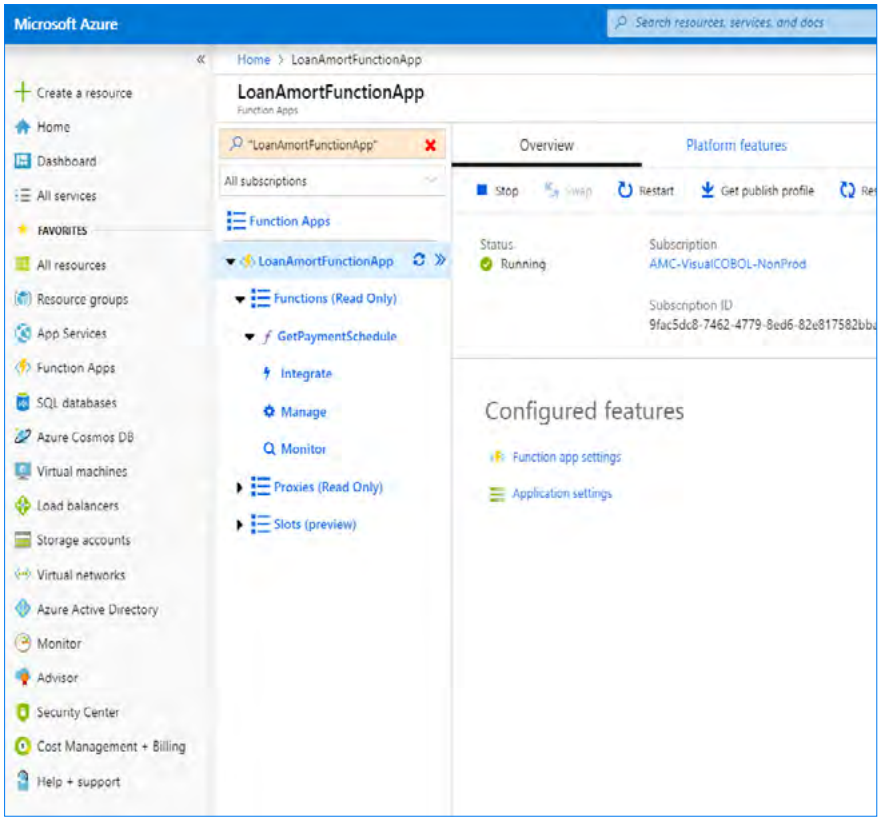
Choose a hosting plan nearest your location. Leave the other fields with their default settings, then click 'Create' to deploy the function into Azure. It may take a few minutes.



Now, using a browser, log into the Azure portal, check out the All Resources section and make sure your Function App is deployed.



Click on the link for the type App Service to display details

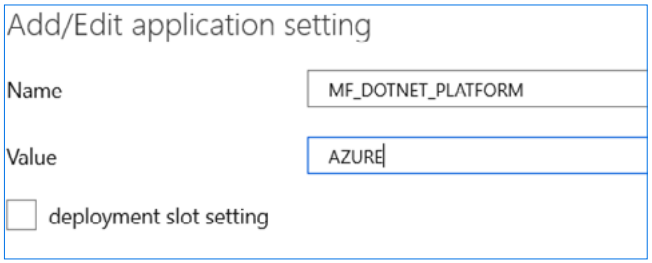


Make sure you configure your function before going further!

Important step klaxon! Add some extra configuration to the function

Click the **'Application settings'** link beneath the 'Configured features' section. It's near the bottom of the App Service 'Details' page.

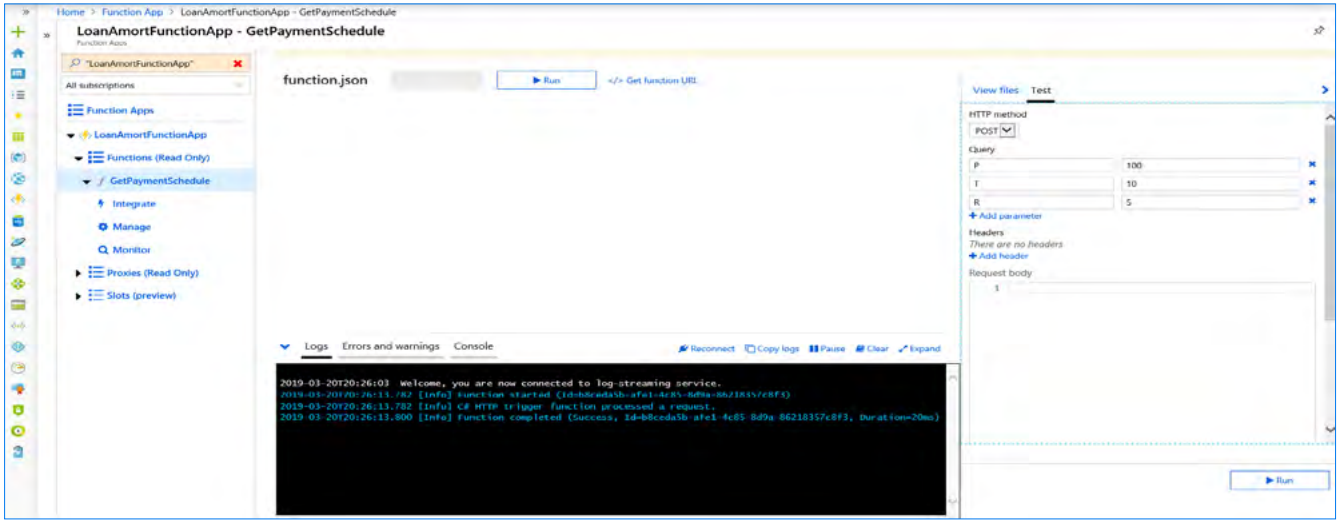
In Application Settings, click on the 'New Application setting' link and add the following:



Name=MF_DOTNET_PLATFORM
Value=AZURE

Click 'Update' and 'Save'.

Test your function from within the portal by clicking **GetPaymentSchedule,** and then **Run.** Use the parameters as shown and click Run again.



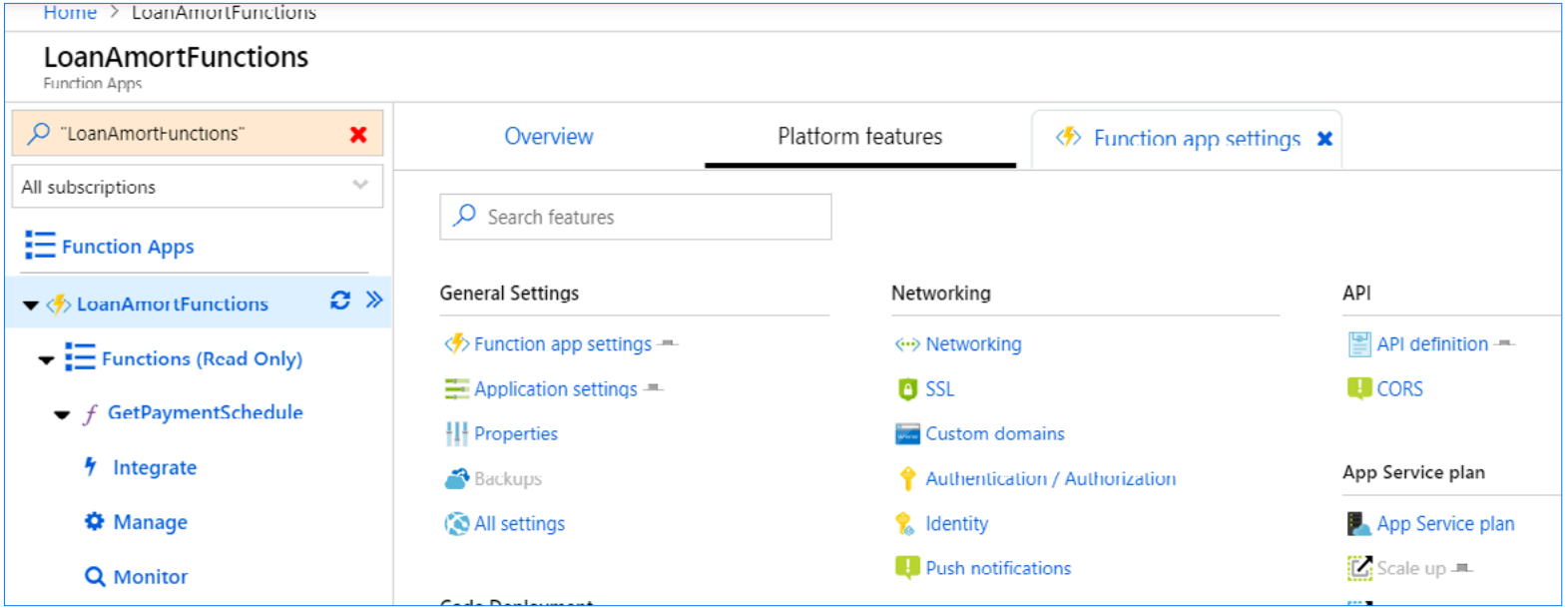
Want to use a browser to test your API? Grab your specific function URL from the function app settings and then paste it into a browser:

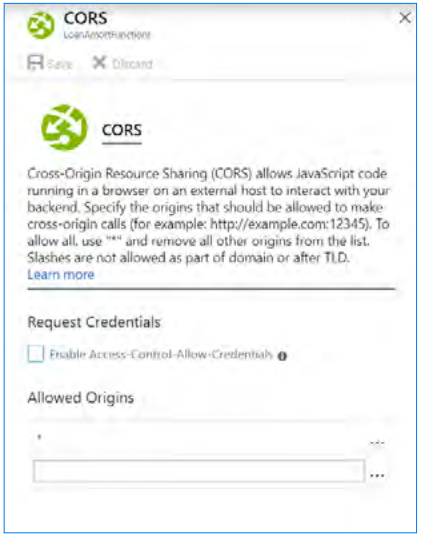
https://YOURFUNCTIONURL/api/GetPaymentSchedule?P=100&T=10&R=5

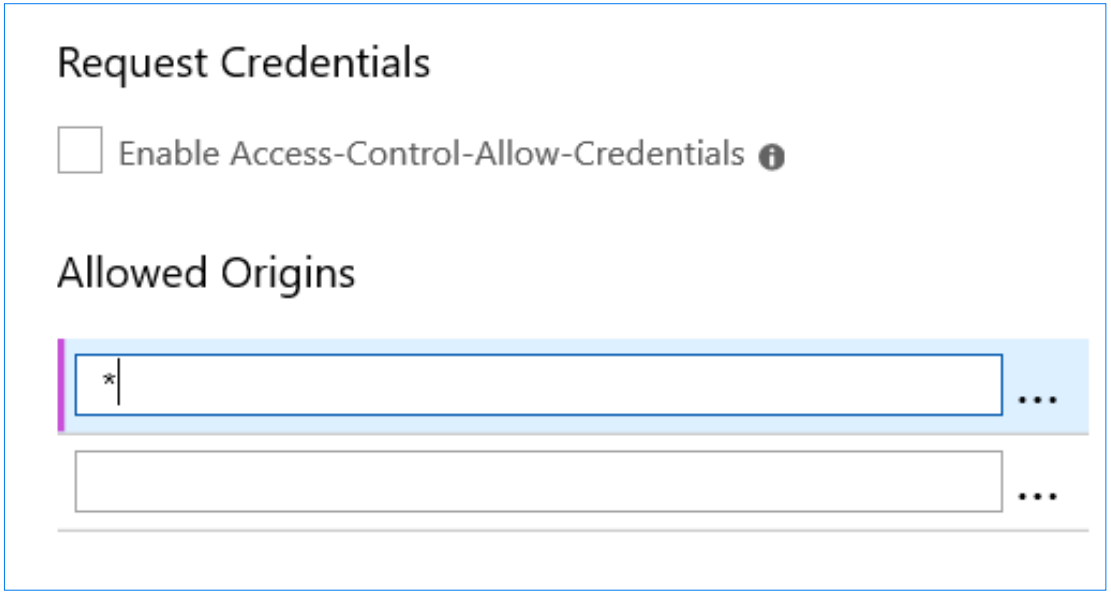
[Results from your Azure hosted function.]


```
{"AmortList":[{"PayDateNo":"#0 4/20/2019","Balance":"    $90.00","InterestPaid":"     $.41","PrincipalPaid":"
$9.81","Payment":"   $10.23"},{"PayDateNo":"#1 5/20/2019","Balance":"    $80.00","InterestPaid":"     $.37","PrincipalPaid":"
$9.85","Payment":"   $10.23"},{"PayDateNo":"#2 6/20/2019","Balance":"    $70.00","InterestPaid":"     $.33","PrincipalPaid":"
$9.89","Payment":"   $10.23"},{"PayDateNo":"#3 7/20/2019","Balance":"    $60.00","InterestPaid":"     $.29","PrincipalPaid":"
$9.93","Payment":"   $10.23"},{"PayDateNo":"#4 8/20/2019","Balance":"    $50.00","InterestPaid":"     $.25","PrincipalPaid":"
$9.98","Payment":"   $10.23"},{"PayDateNo":"#5 9/20/2019","Balance":"    $40.00","InterestPaid":"     $.20","PrincipalPaid":"
$10.02","Payment":"   $10.23"},{"PayDateNo":"#6 10/20/2019","Balance":"    $30.00","InterestPaid":"     $.16","PrincipalPaid":"
$10.06","Payment":"   $10.23"},{"PayDateNo":"#7 11/20/2019","Balance":"    $20.00","InterestPaid":"     $.12","PrincipalPaid":"
$10.10","Payment":"   $10.23"},{"PayDateNo":"#8 12/20/2019","Balance":"    $10.00","InterestPaid":"     $.08","PrincipalPaid":"
$10.14","Payment":"   $10.23"},{"PayDateNo":"#9 1/20/2020","Balance":"     $.00","InterestPaid":"    $.04","PrincipalPaid":"
$10.00","Payment":"   $10.04"}],"TotalInterest":"    $2.29"}
```

We're going to test your API using the web project. But you'll need to make some configuration changes to the function settings in Azure first. So open the function and click on '**Platform Features**' and then **CORS**.

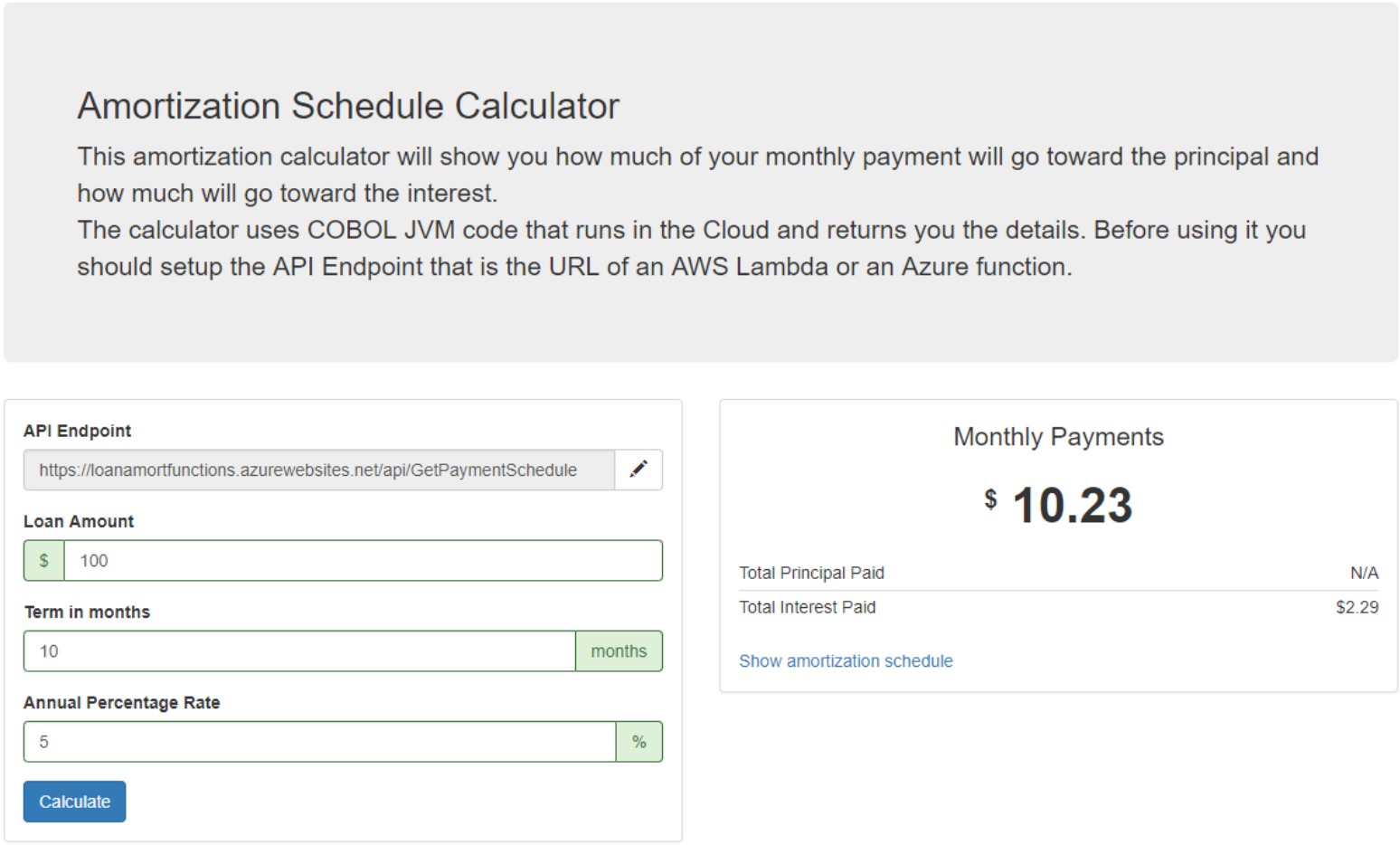






Now, remove all the end points in the **Allows Origins** list and replace. with a single *. Then restart the function and run the C# web project again. This time, change the endpoint URL field so it references the Azure API.

Your browser is now connected the Azure hosted function.



Hats off. Getting this far is an achievement.

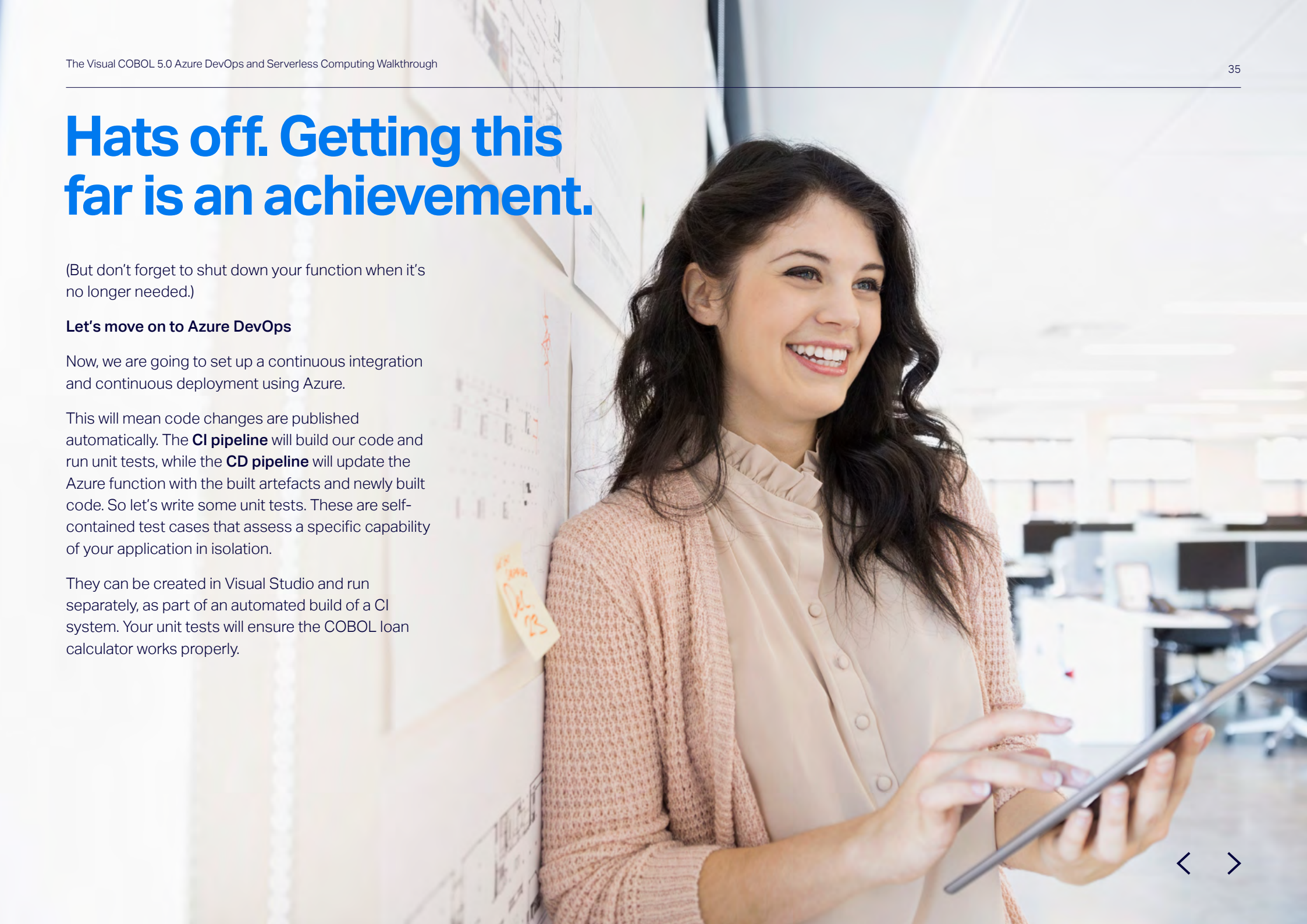
(But don't forget to shut down your function when it's no longer needed.)

Let's move on to Azure DevOps

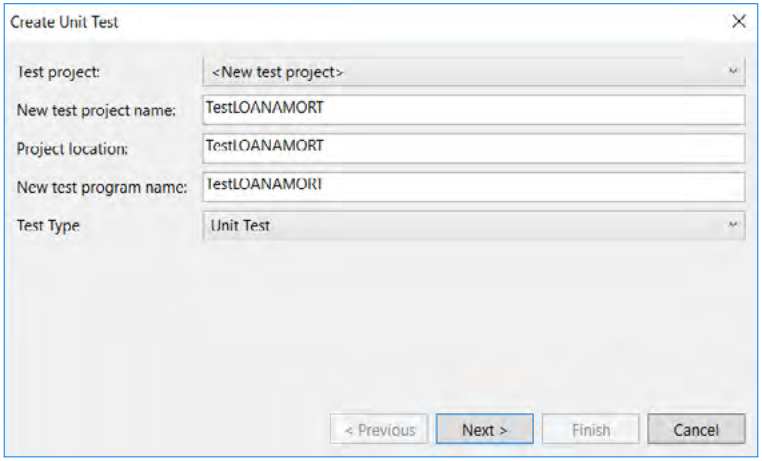
Now, we are going to set up a continuous integration and continuous deployment using Azure.

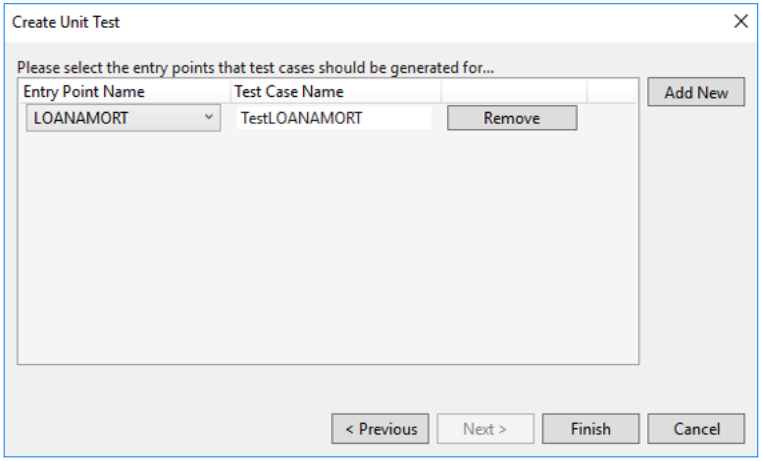
This will mean code changes are published automatically. The **CI pipeline** will build our code and run unit tests, while the **CD pipeline** will update the Azure function with the built artefacts and newly built code. So let's write some unit tests. These are self-contained test cases that assess a specific capability of your application in isolation.

They can be created in Visual Studio and run separately, as part of an automated build of a CI system. Your unit tests will ensure the COBOL loan calculator works properly.



So, open the COBOL **LOANAMORT.cbl** file in Visual Studio. Right-click the code in the editor and select 'Create Unit Test'. Click 'Finish' to create a new project named **TestLOANAMORT**. Now, we're going to review the unit test code.





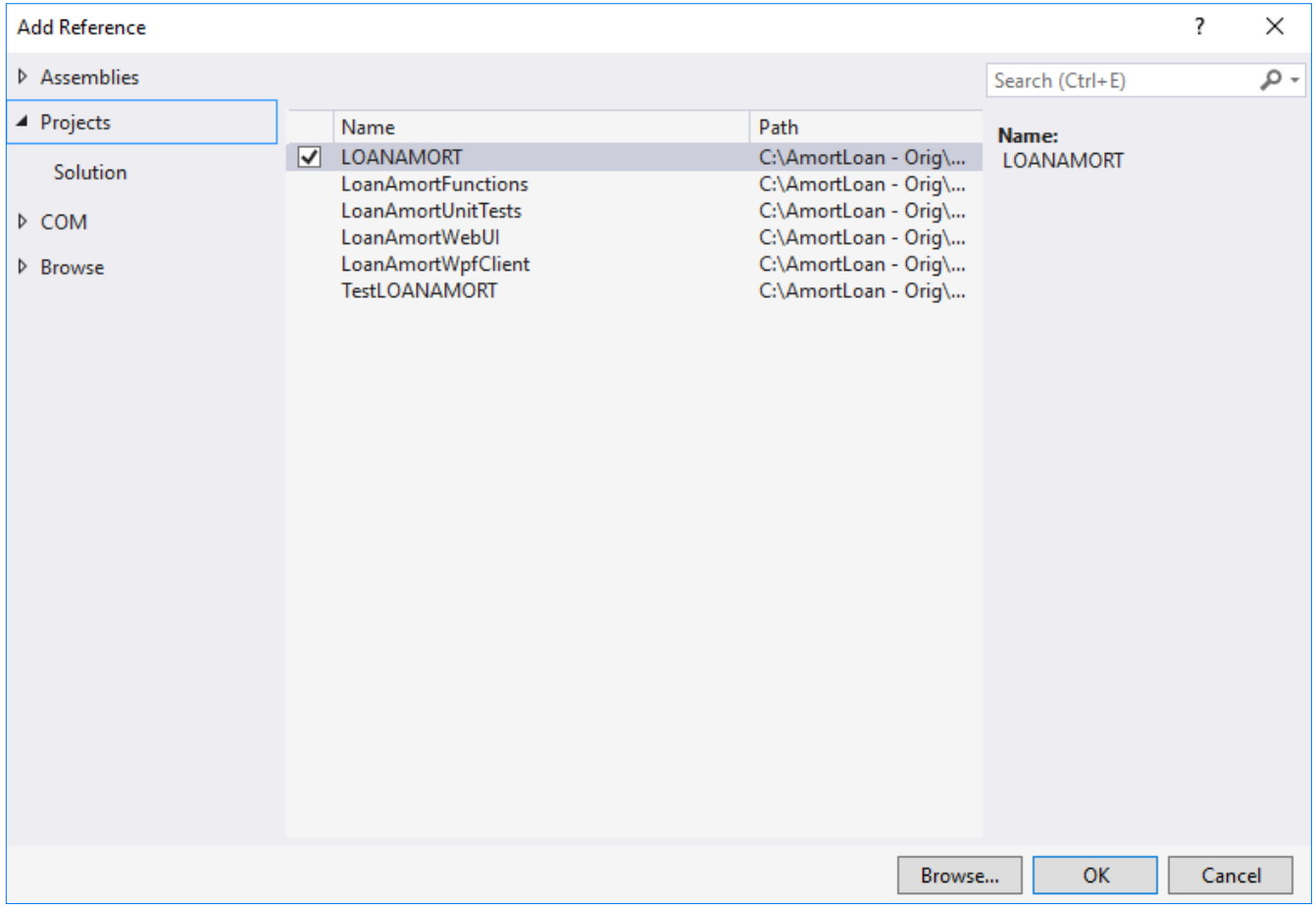
```
1       *> Test Fixture for LOANAMORT, LOANAMORT
2
3       copy "mfunit_prototypes.cpy".
4
5       program-id. TestLOANAMORT.
6       working-storage section.
7       copy "mfunit.cpy".
8       78 TEST-TESTLOANAMORT value "TestLOANAMORT".
9       01 pp procedure-pointer.
10
11      *> Program linkage data
12      01 LOANINFO.
13        03 PRINCIPAL PIC S9(8) COMP-3.
14        03 LOANTERM PIC S9(8) COMP-3.
15        03 RATE PIC S9(9)V9(9).
16      01 OUTDATA.
17        03 PAYMENTS occurs 480 depending on LOANTERM.
18          05 OUTINTPAID PIC $$,$(3).99.
19          05 OUTPRINCPAID PIC $$,$(3).99.
20          05 OUTPAYMENT PIC $$,$(3).99.
21          05 OUTBALANCE PIC $(3),$(3).99.
22        03 OUTTOTINTPAID PIC $$,$(3).99.
23
24      procedure division.
25          goback returning 0
26      .
27
28      entry MFU-TC-PREFIX & TEST-TESTLOANAMORT.
29
30          call "LOANAMORT" using
31                      by reference LOANINFO
32                      by reference OUTDATA
33
34          *> Verify the outputs here
35          goback returning MFU-PASS-RETURN-CODE
36      .
```

Your first COBOL unit test. Doesn't do much yet

Line 5: this program has been automatically generated based on the LOANAMORT program. You can use this program to create different test cases

Line 11-22: these are the parameters used in the LINKAGE SECTION of the LOANAMORT program

Line 28: this is a single test case. This entry point will be called by the unit testing framework



To add a reference to your COBOL project. Right-click the **References Node** in the solution explorer and click 'Add Reference' in the unit test project and add a reference to the **LOANAMORT** project.

Let's create a test case.

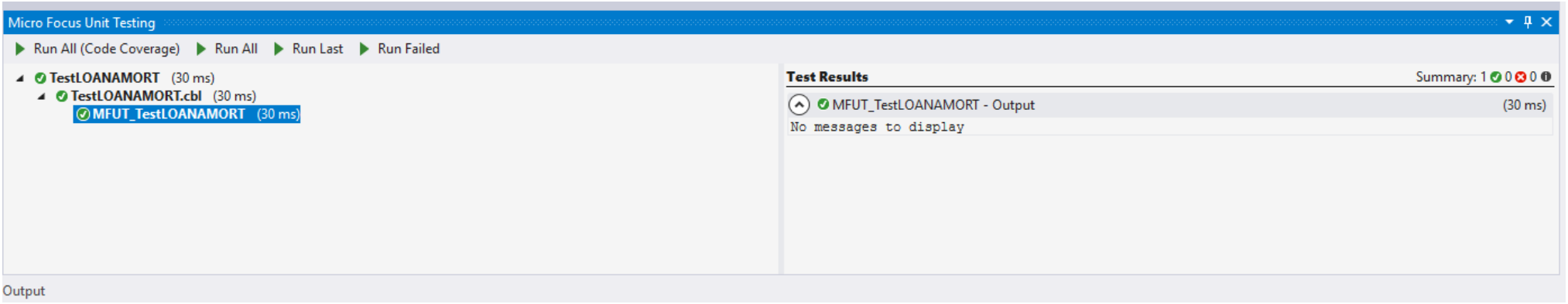
```
000028 entry MFU-TC-PREFIX & TEST-TESTLOANAMORT.
000029
000030     move 0 to PRINCIPAL
000031
000032     call "LOANAMORT" using
000033                 by reference LOANINFO
000034                 by reference OUTDATA
000035
000036     *> Verify the outputs here
000037     if function numval-c(OUTTOTINTPAID) not = 0
000038         call "MFU_ASSERT_FAIL_Z" using "Total paid should be zero is: " & OUTTOTINTPAID & x"00"
000039         goback returning MFU-FAIL-RETURN-CODE
000040     end-if
000041
000042     goback returning MFU-PASS-RETURN-CODE
```

Add this code to Test Behaviour when a zero value loan is requested.

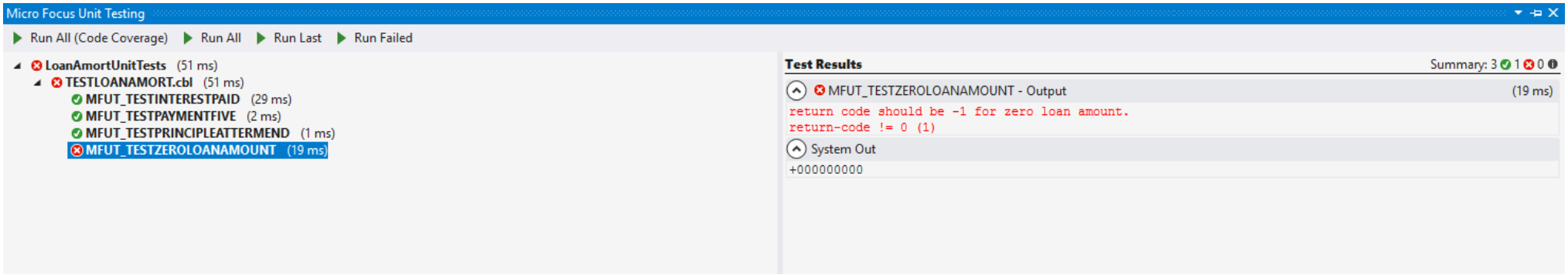
The **OUTTOTINTPAID** field should be zero.

To run the Unit Test, make the test project the Start Up project and hit **F5**.

Results should be in green in the Unit Test window. Like this…



So you've created a unit test. A single test isn't going to get us far but to save you the trouble of writing any more, we've written a bank of tests for you. Let's import them. The **LoanAmortUnitTests** project includes several unit tests. So add it to the solution. Make it the Startup project, and run the tests. One test should fail. We'll fix this failing test case later.



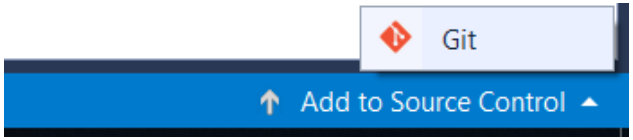
Step 2 is to **set up an Azure DevOps project…**

Azure DevOps has most of the software you need to create CI and CD pipelines. It's also a source code repository. Now, we will set up a CI/CD pipeline to build, test and publish the function after successful changes.

You'll need to sign up for Azure DevOps. It's free from here.

We're going to create an Azure repo - a source code management system for storing your code, and any changes, directly from within Visual Studio. So let's add a project.

Click 'Add to Source Control' on the solution explorer, and select **Git.**

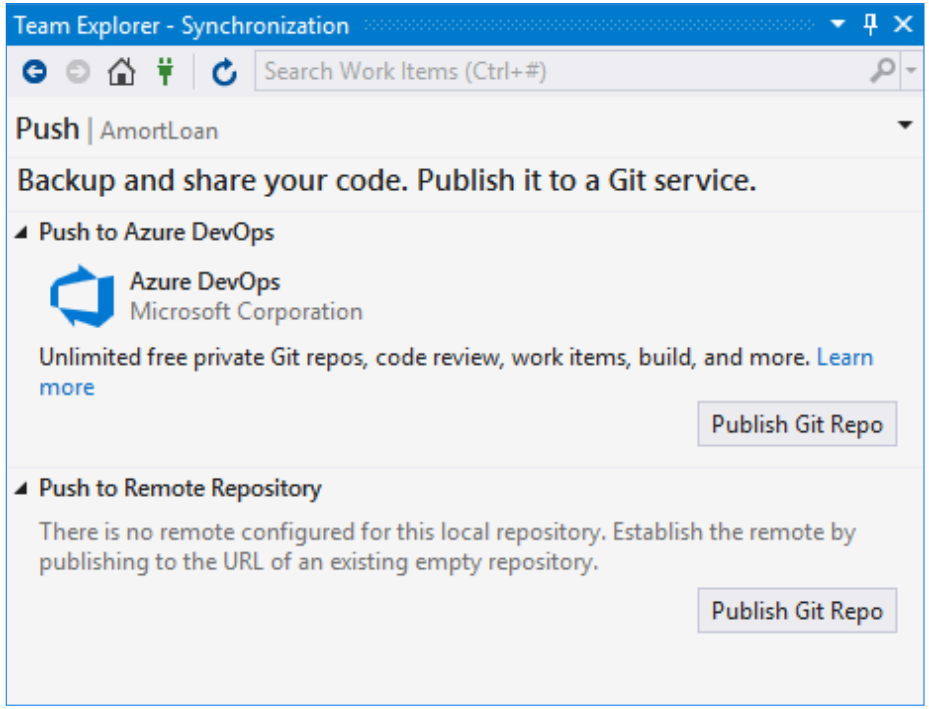


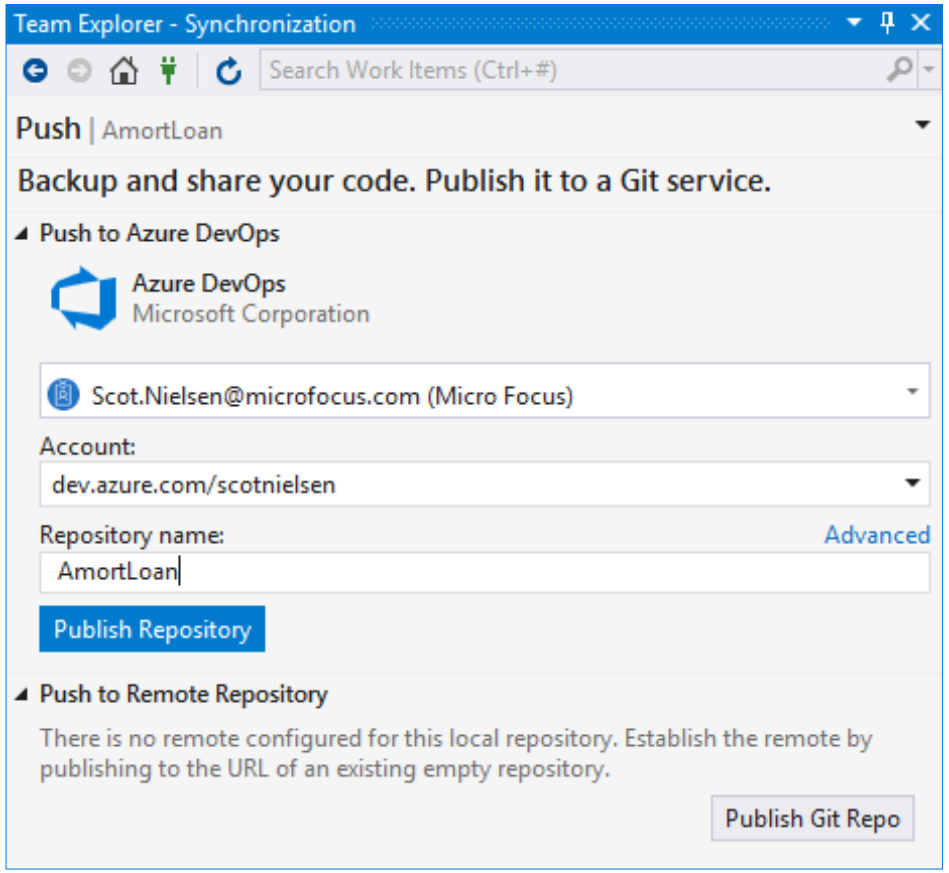
You should now see the **Team Explorer**

Synchronization window. Not there?

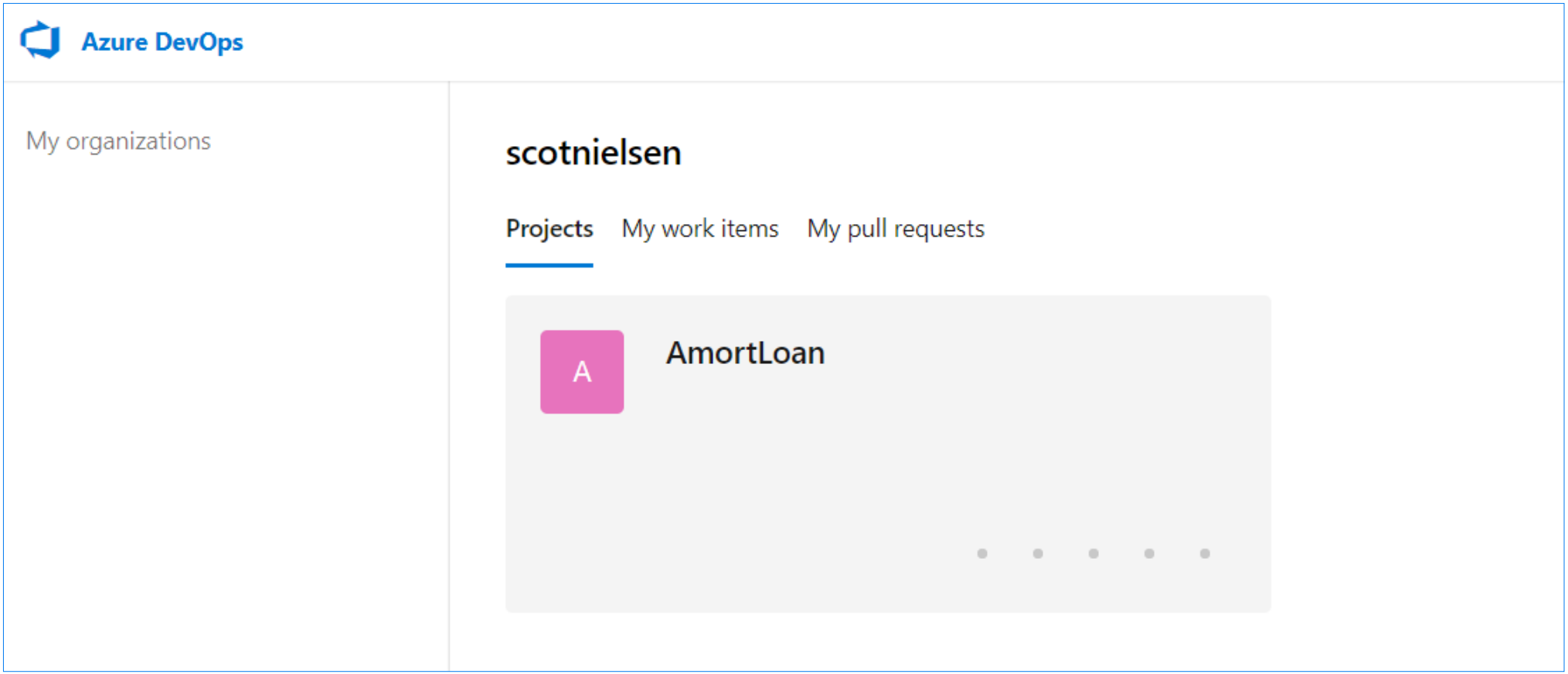
Click the number link (0 in the graphic} to the right of the uncommitted pushes arrow.

Click **Publish Git Repo** under 'Push to Azure DevOps'. Enter your Azure DevOps credentials, give your project a name and click **Publish Repository**.

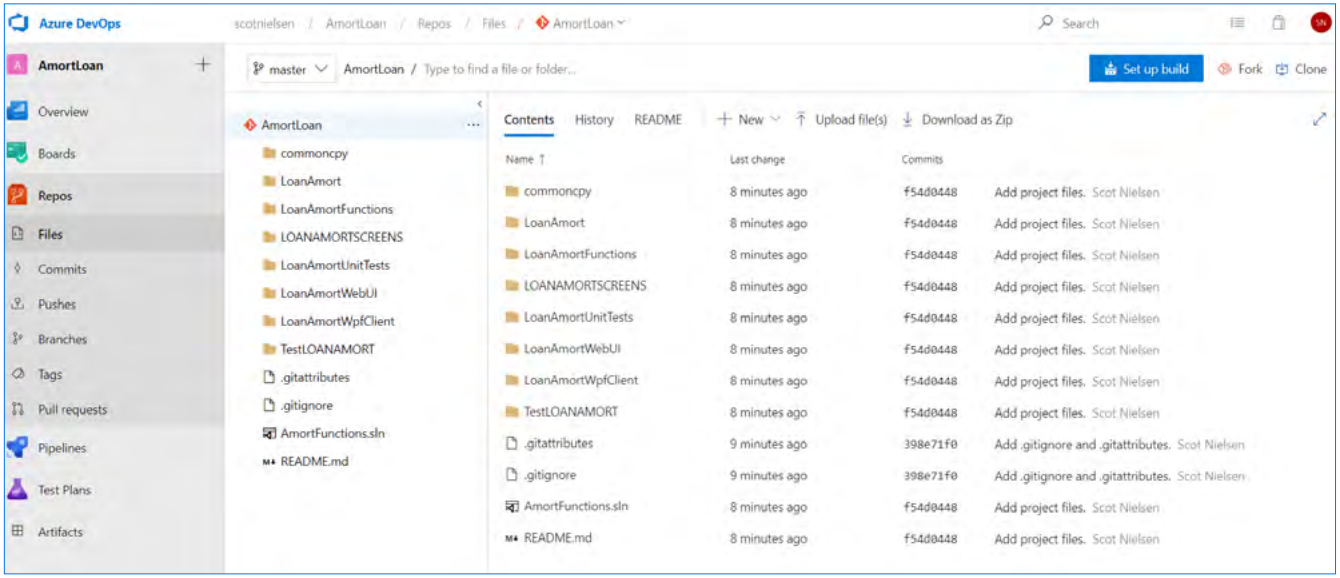




Want to see your project in action? Then log in to Azure DevOps.

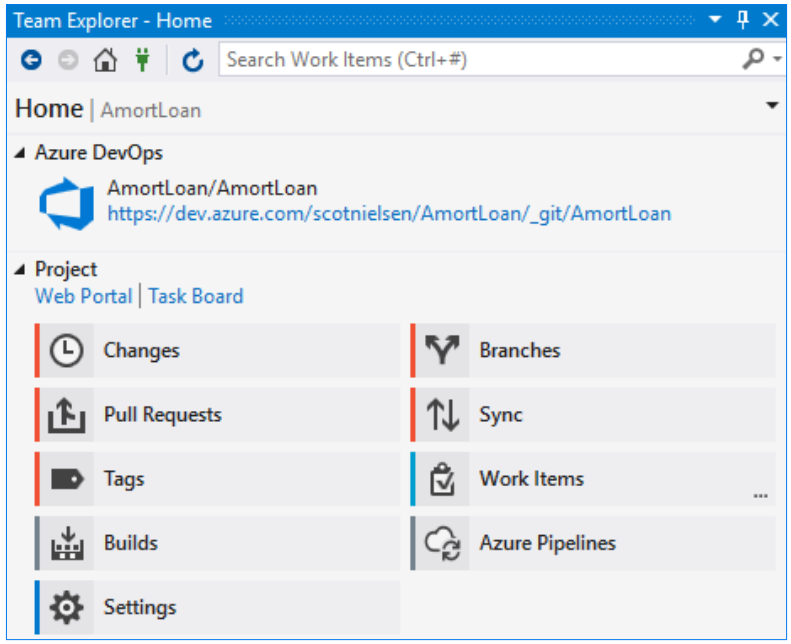


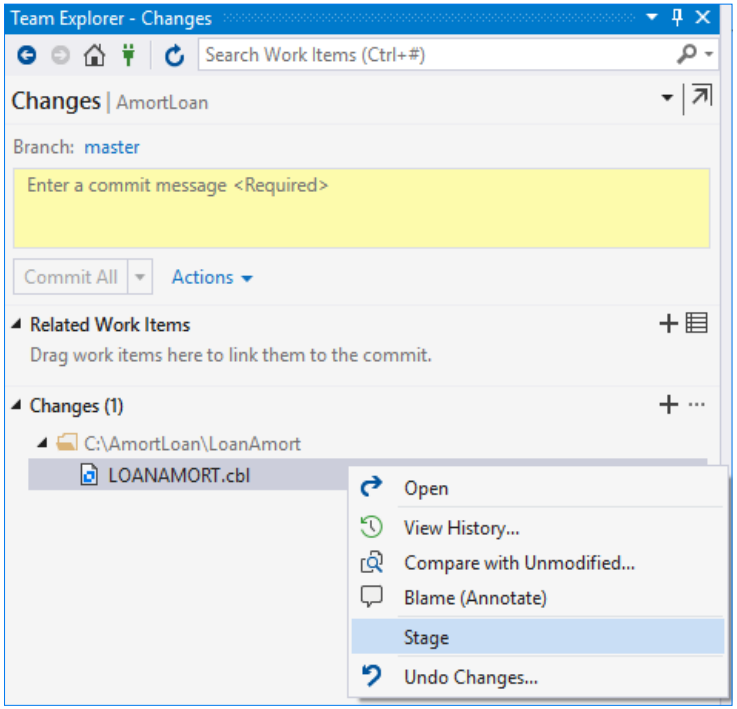
Here's how your code will look under Repos...



Want to make a test change to the code? Add a **TODO** to the **LOANAMORT.CBL** program and close it in the editor. Save and build the code. It'll look like this...

```
30      LINKAGE SECTION.
31      01 COPY AMORTIN.
32      01 COPY AMORTOUT.
33
34      * TODO: This API has limited validation on input paramsters
35
36      PROCEDURE DIVISION USING LOANINFO
37                               OUTDATA.
38
39          PERFORM CALC-PAYMENT
40          MOVE WRK-PAYMENT TO DECPAYMENT
41
```



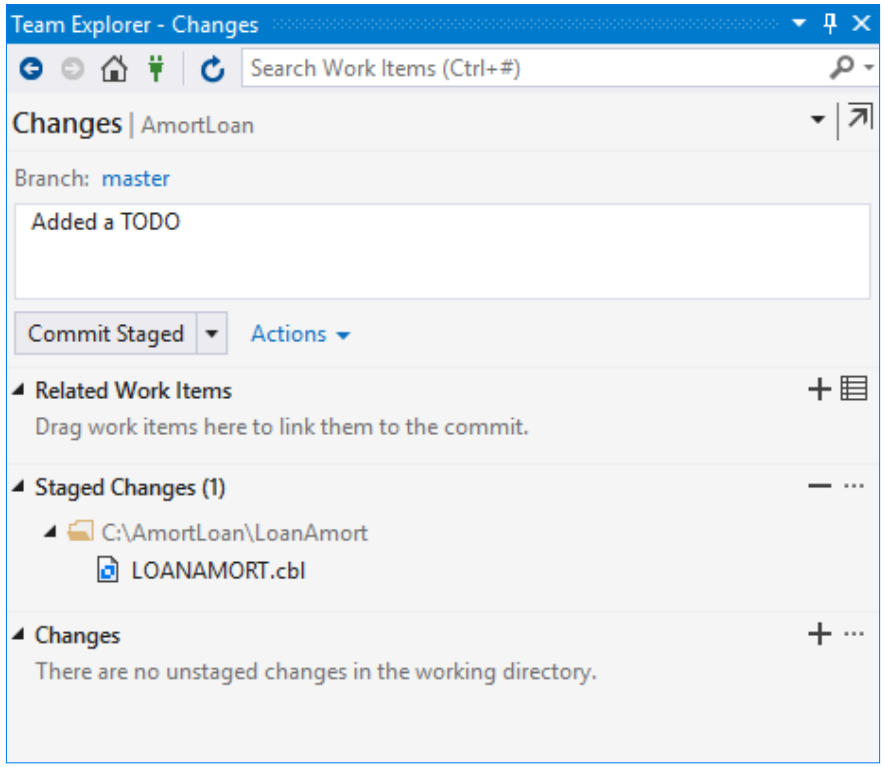


Commit your code to the repo

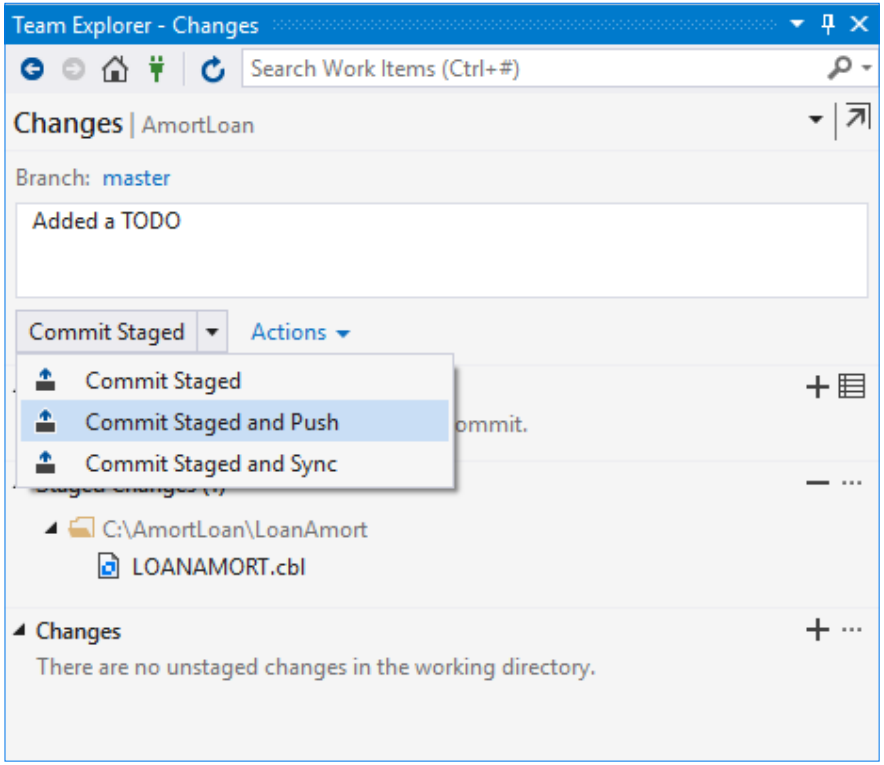
Make sure the Team Explorer is visible. You can find it on the view menu. Once it's open, click the home icon. It's on top of the Team Explorer Window. Then click 'Changes' and you should see the modified file. So right-click on the filename and select Stage.

Your commit log should look something like this.

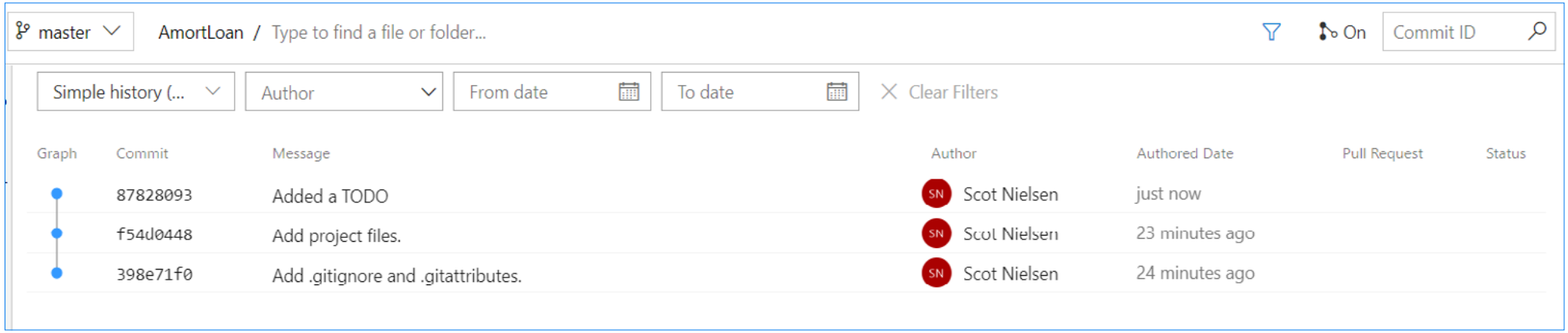
Now, add a commit message, such as '**Added a TODO**' and then click **Commit Staged and Push**



Click Commit Staged and Push to send your changed file into the repo.



Want to check you can see your Commit? Open Azure DevOps and click the Commits link in the repo section.



Next step – let's **set up a Continuous Integration pipeline** to monitor the source code system and run tasks if the code changes. We'll create tasks to extract, build, test and archive the results and a machine for Azure DevOps to build our code. Here's the step-by-step guide.

Open Azure DevOps. Then click

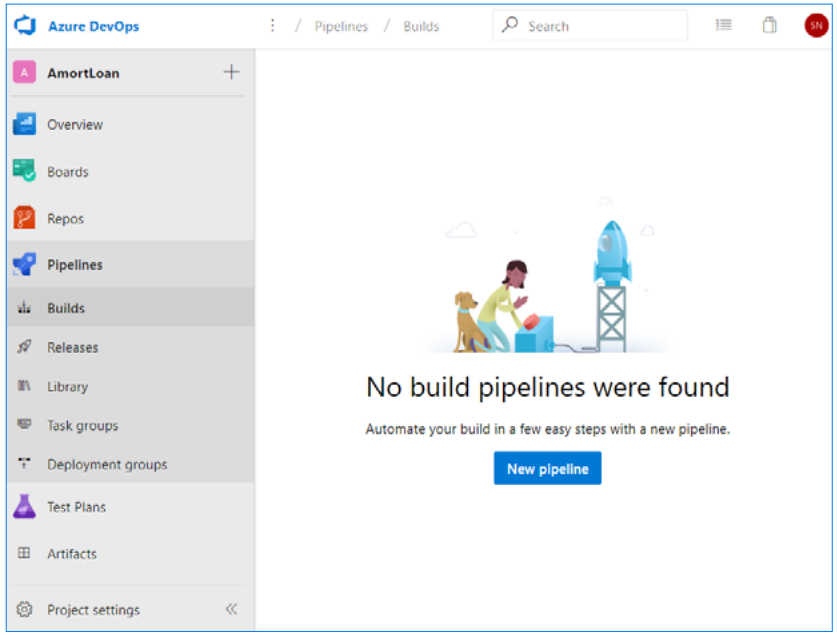
The Pipelines Tab

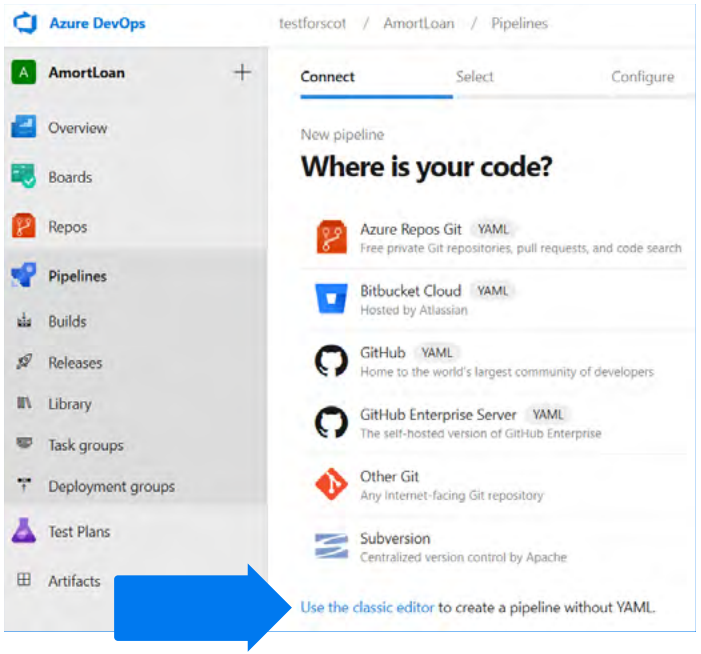
New pipeline

Use Classic Editor.

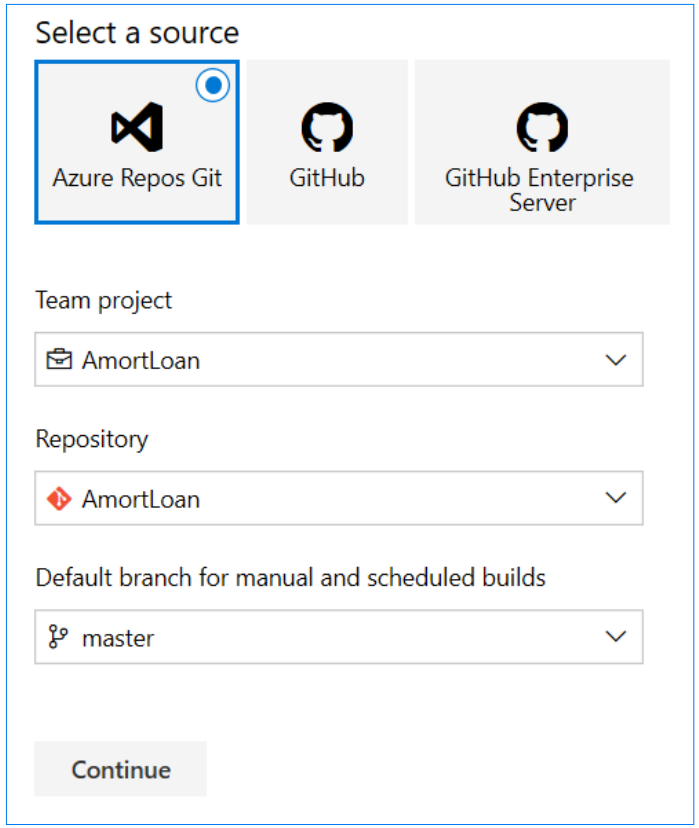
Once you have chosen **Azure Repos Git**, select your project and click **Continue**. You'll go to the next screen. There, click **Empty Job** to create an empty pipeline.

Now, we'll create the tasks.

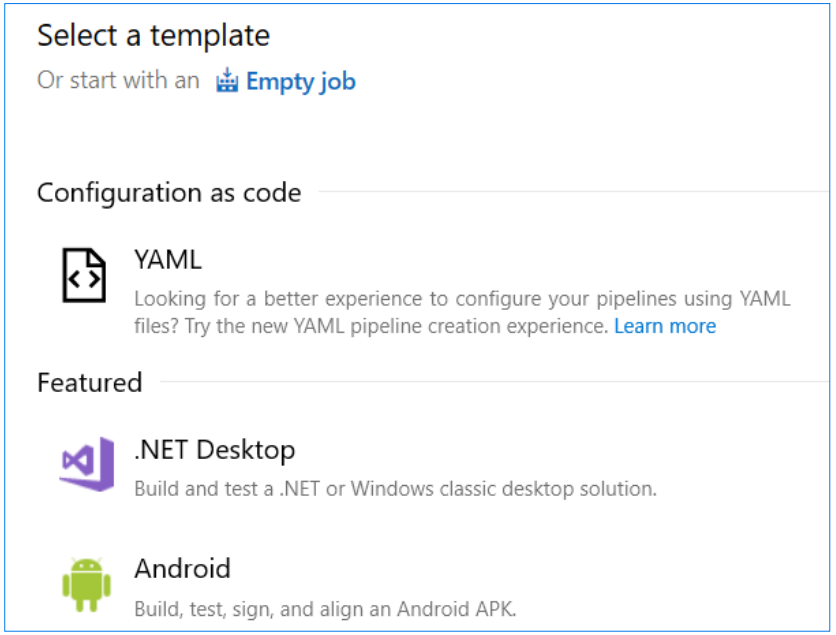




Configuring where your source code is coming from



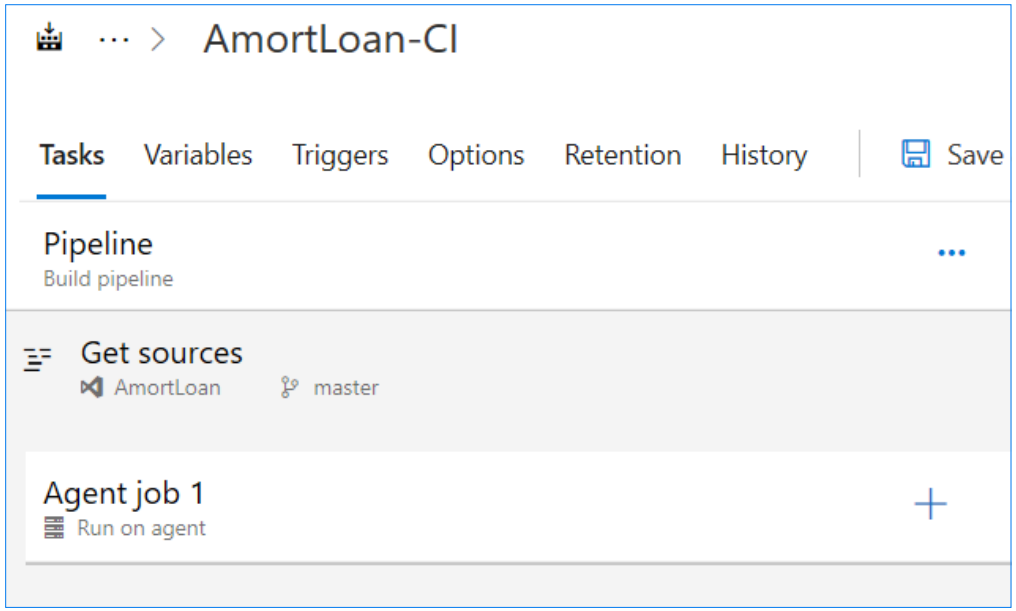
Click empty job to get started.



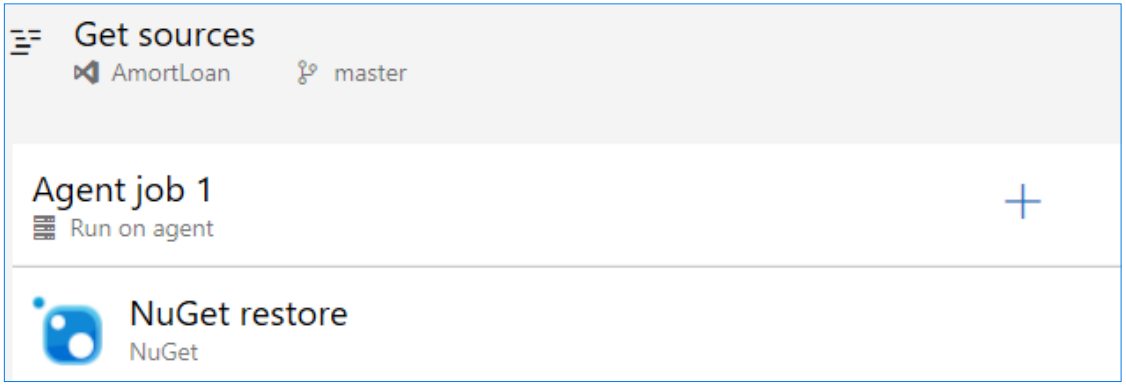
Let's get to it and add these tasks to the pipeline.

o add the Nuget task, **click the + sign** next to Agent job 1 to add a new task…

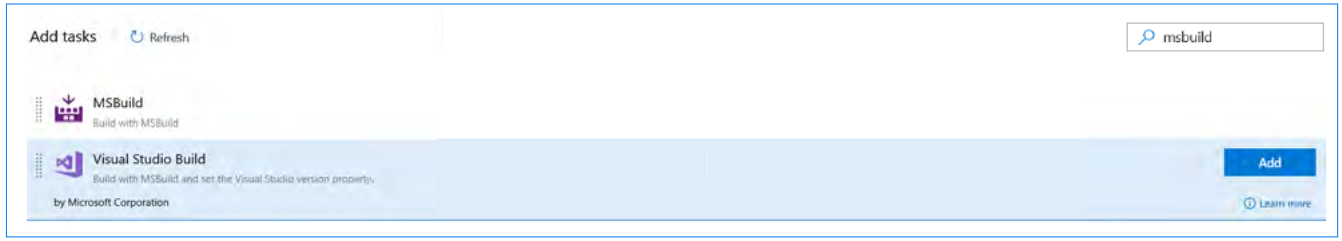
…. type **'nuget'** into the search field on the Add task dialog….



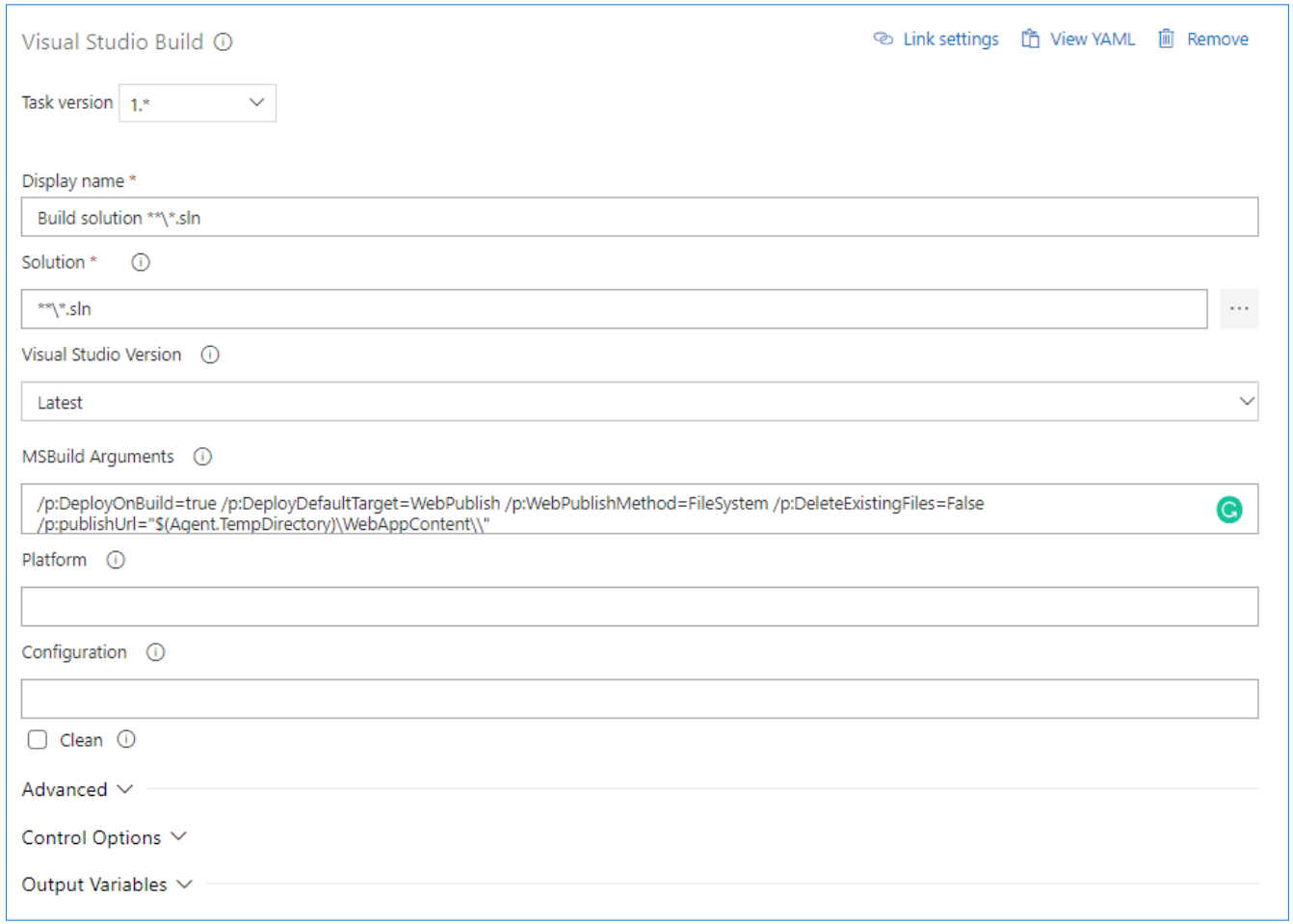
.... and click on Add for the NuGet Restore task....



.... And click on the newly added task to view it, using the default settings.



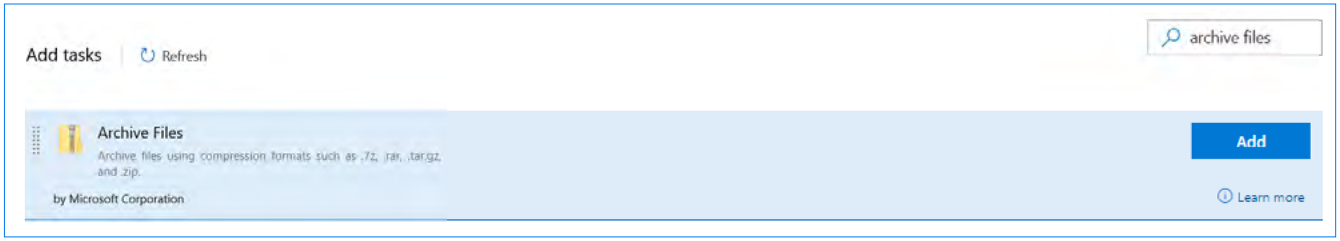
Add an MSBuild task, **click the + sign** next to Agent job 1 again to add a new task, then type "build with MSBuild" into the search field on the Add task dialog and click Add.



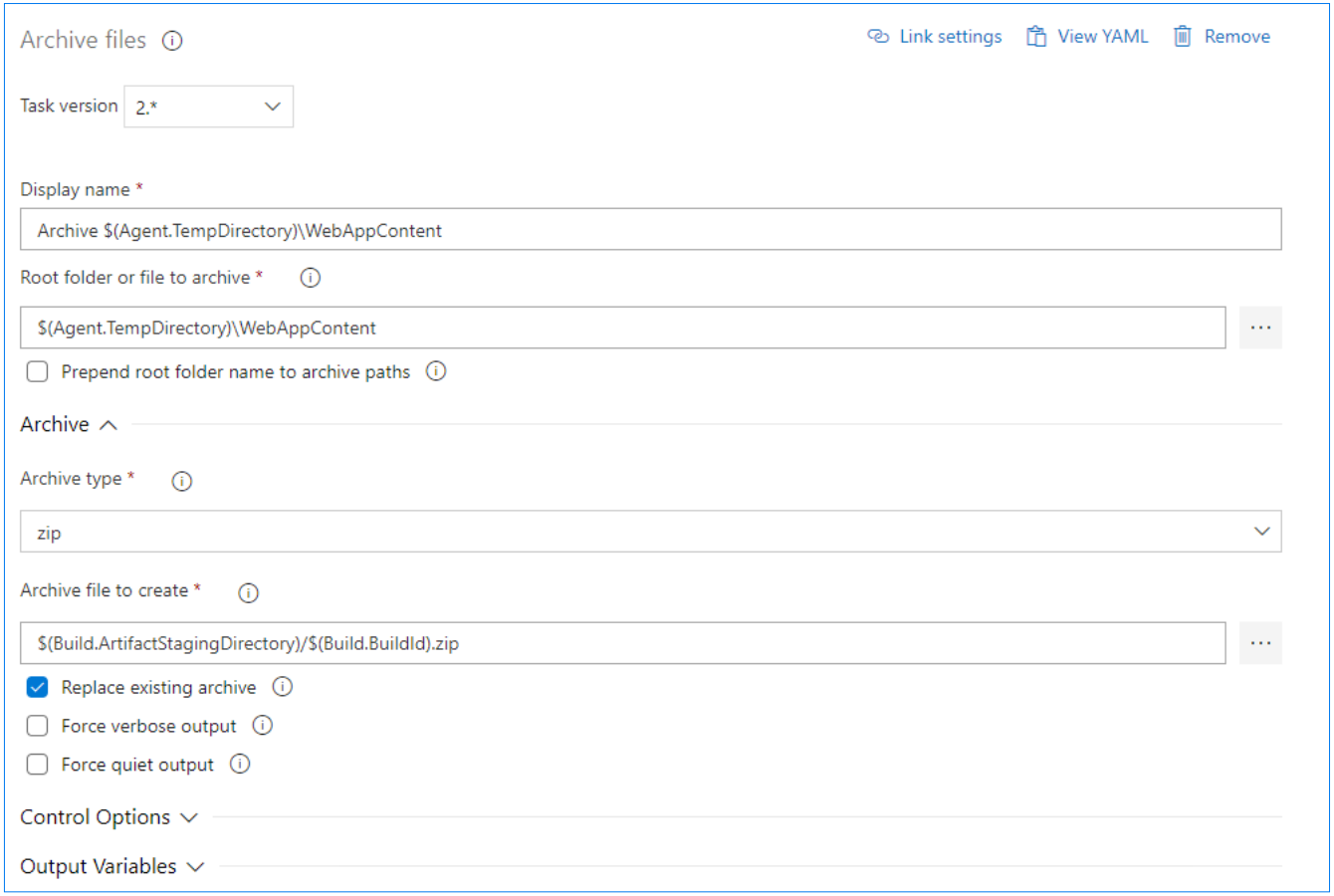
To configure the Visual Studio Build Task, click on the newly added task to open it for configuration.

Add this text to MSBuild Arguments field:

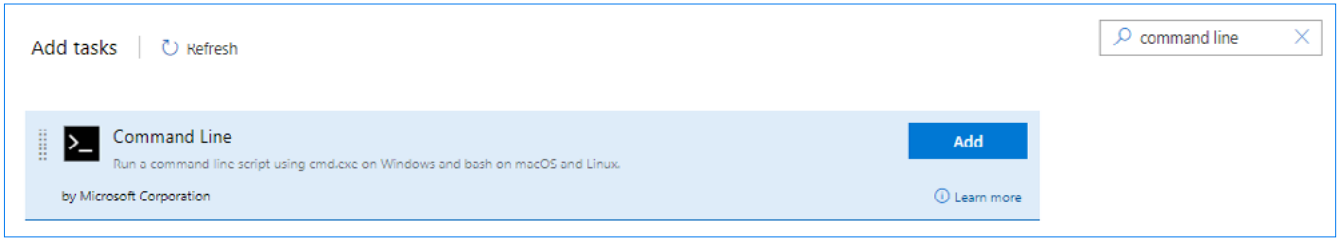
/p:DeployOnBuild=true
/p:DeployDefaultTarget=WebPublish
/p:WebPublishMethod=FileSystem
/p:DeleteExistingFiles=False
/p:publishUrl="$(Agent.TempDirectory)
\WebAppContent\\"



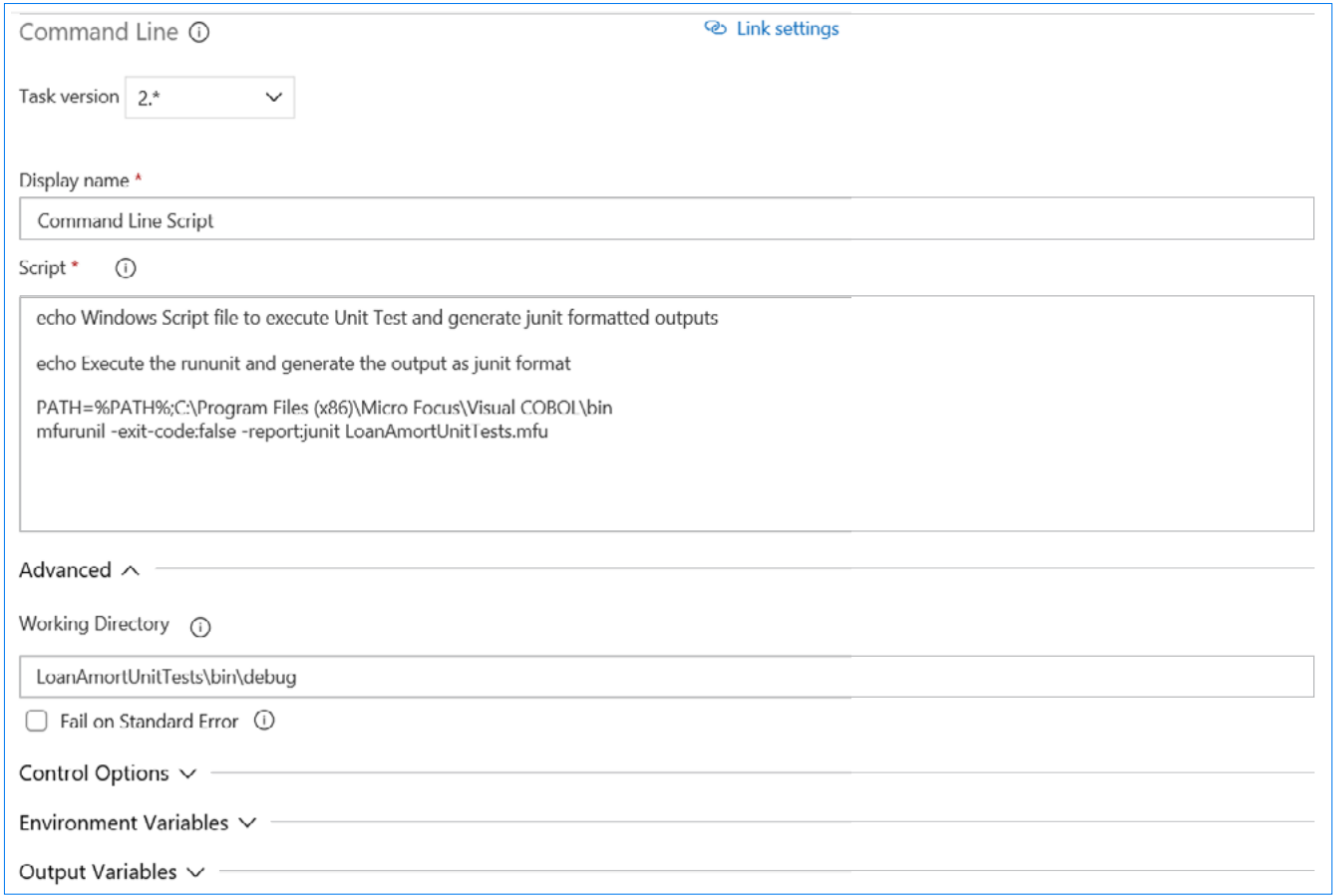
To add the Archive Files Task, **click the + sign** next to Agent job 1 again to add a new task, then type "archive files" into the search field on the Add task dialog and click Add for the Archive Files task.



To configure the Archive Files Task, click on the newly added task to open it for configuration. Then, add **$(Agent.TempDirectory)\WebAppContent** to the **Root folder or file to archive** entry field.



To add the Command Line Task, **click the + sign** next to Agent job 1 again to add a new task, then **type "command line"** into the search field on the Add task dialog, and click **Add** for the Command line task.



Now, let's configure it.

Click on the newly added task to open it. Add the following text to the **Scripts** field:

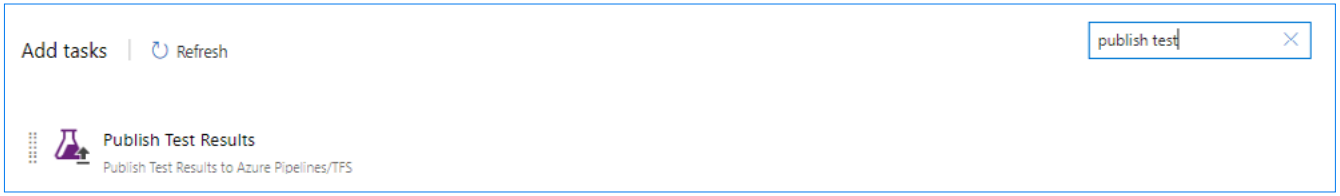
echo Windows Script file to execute Unit Test and generate junit formatted outputs

echo Execute the rununit and generate the output as junit format

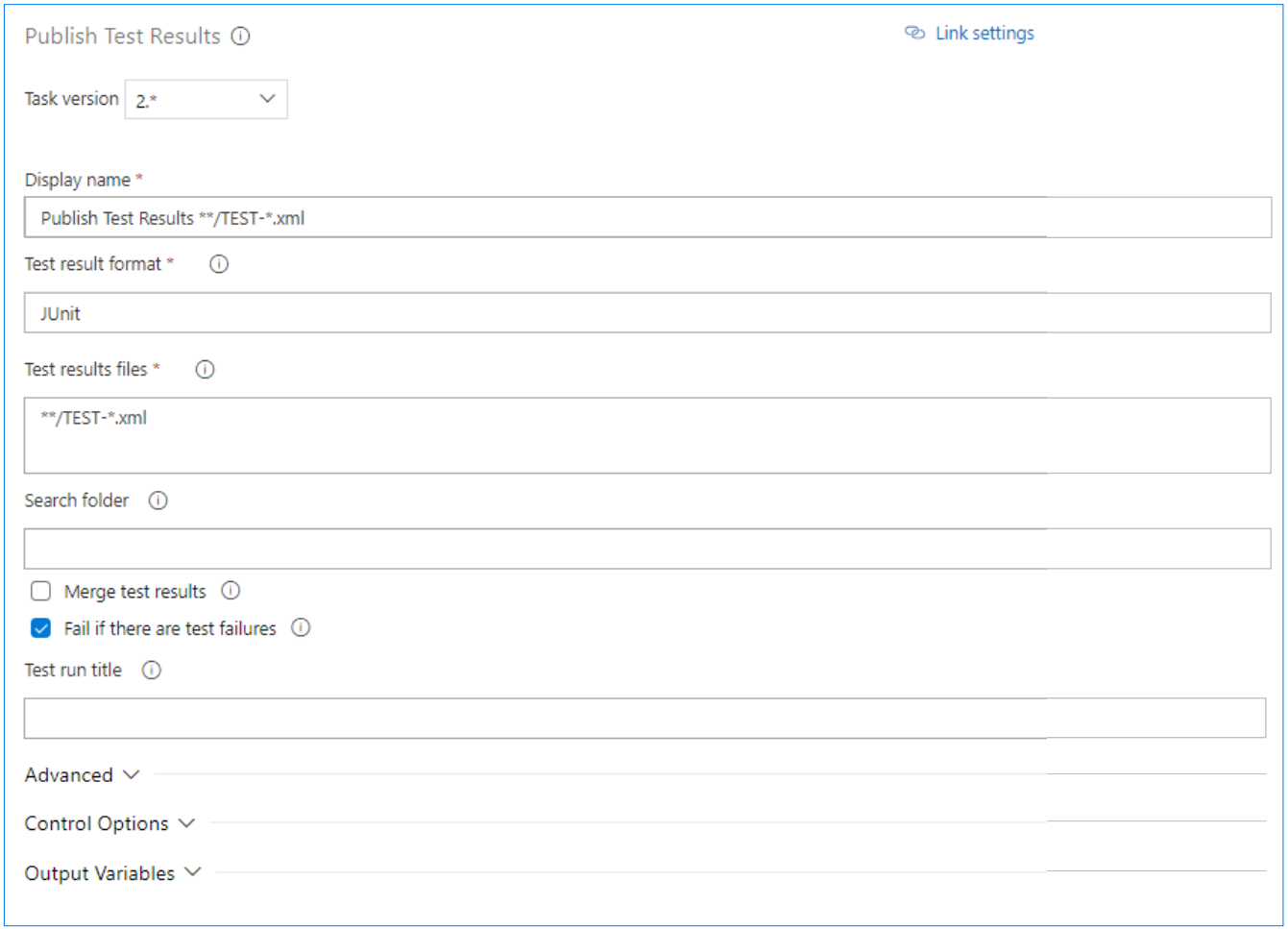
PATH=%PATH%;C:\Program Files (x86)\Micro Focus\Visual COBOL\bin

mfurunil -exit-code:false -report:junit LoanAmortUnitTests.mfu

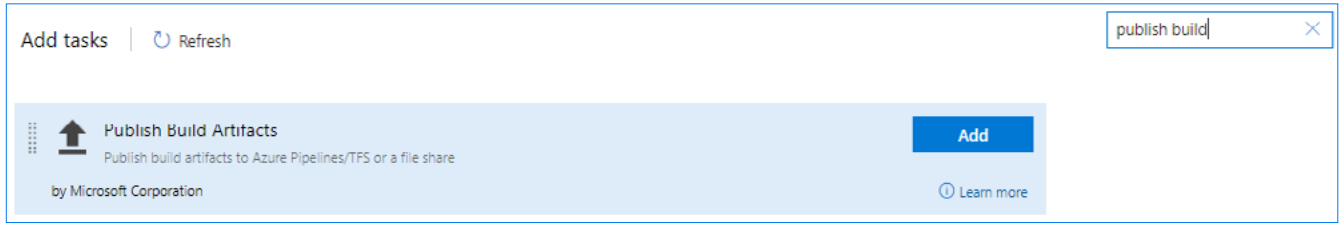
Then, add **LoanAmortUnitTests\bin\Debug** under Advanced in the Working Directory field.

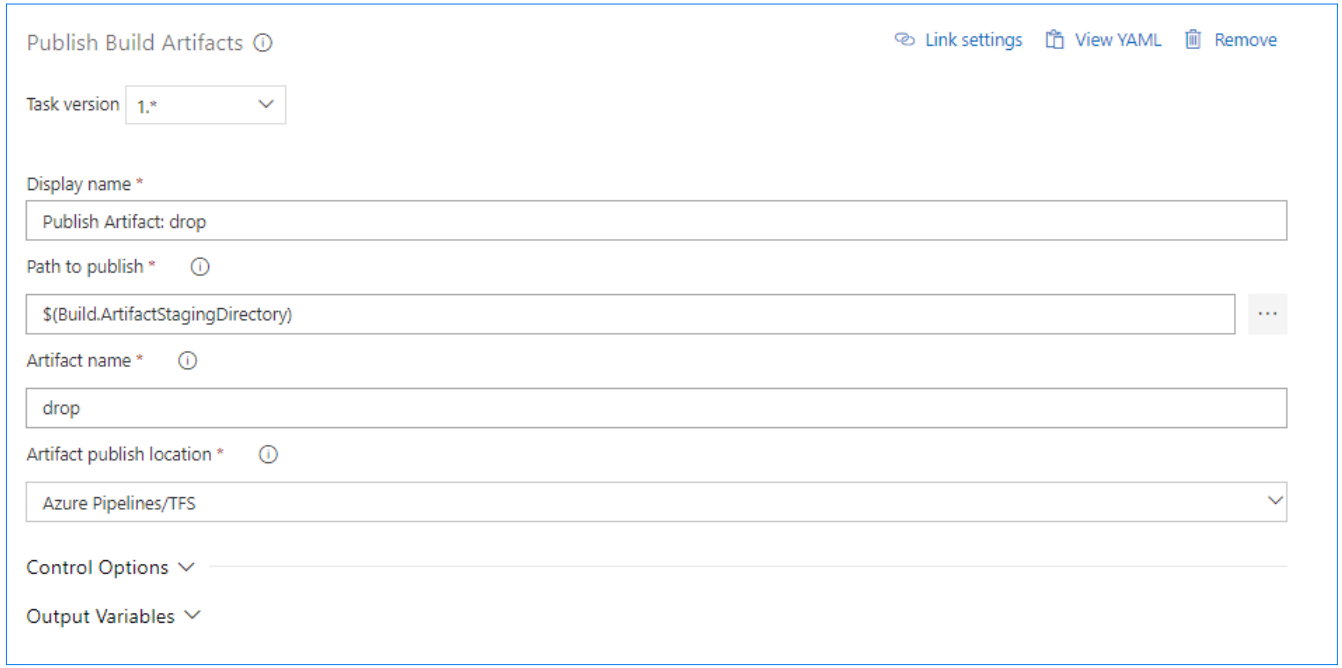


To add the Publish Test Results Task, **click the + sign** next to Agent job 1 again to add a new task. Then type **"publish test"** into the search field on the Add task dialog, and **click Add** for the Publish Test Results task.



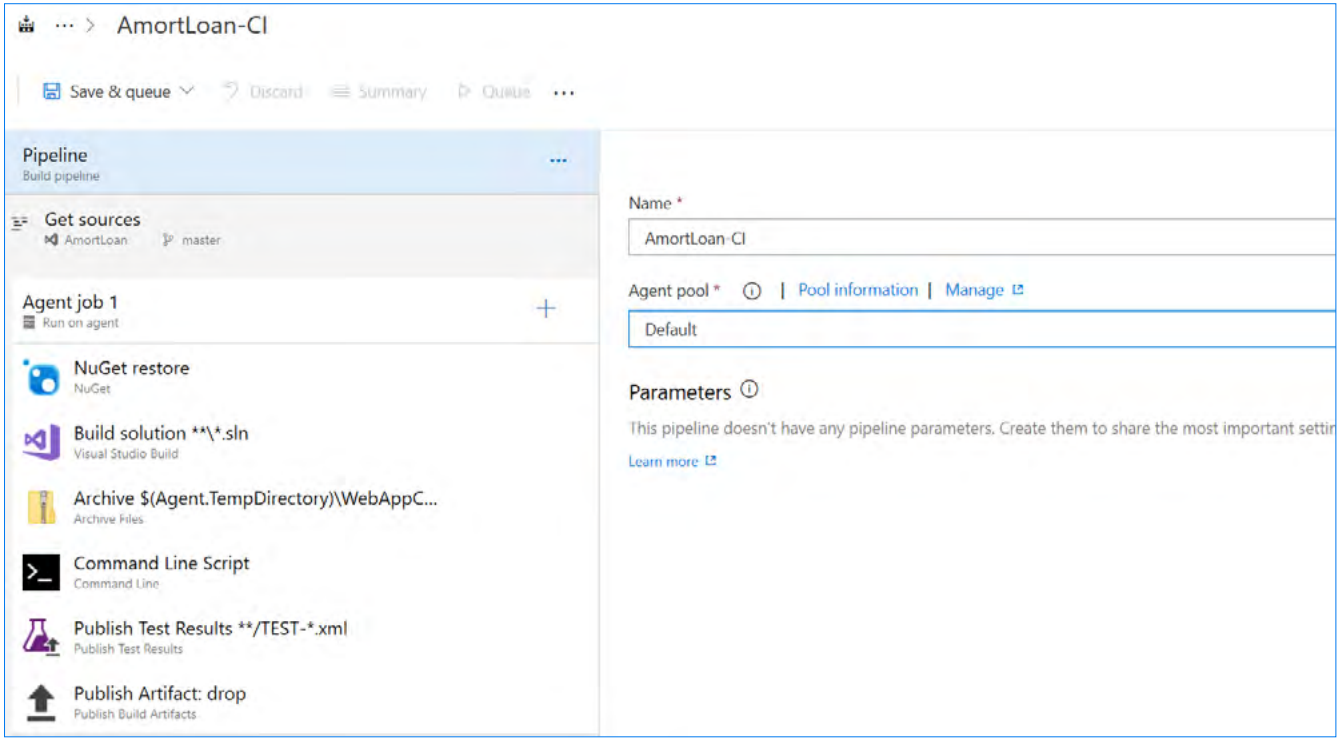
Now we're going to **configure the Publish Test Results Task**. So, again, click on the newly added task to open it for configuration. **Check the 'Fail if there are test failures'** option to turn it on.



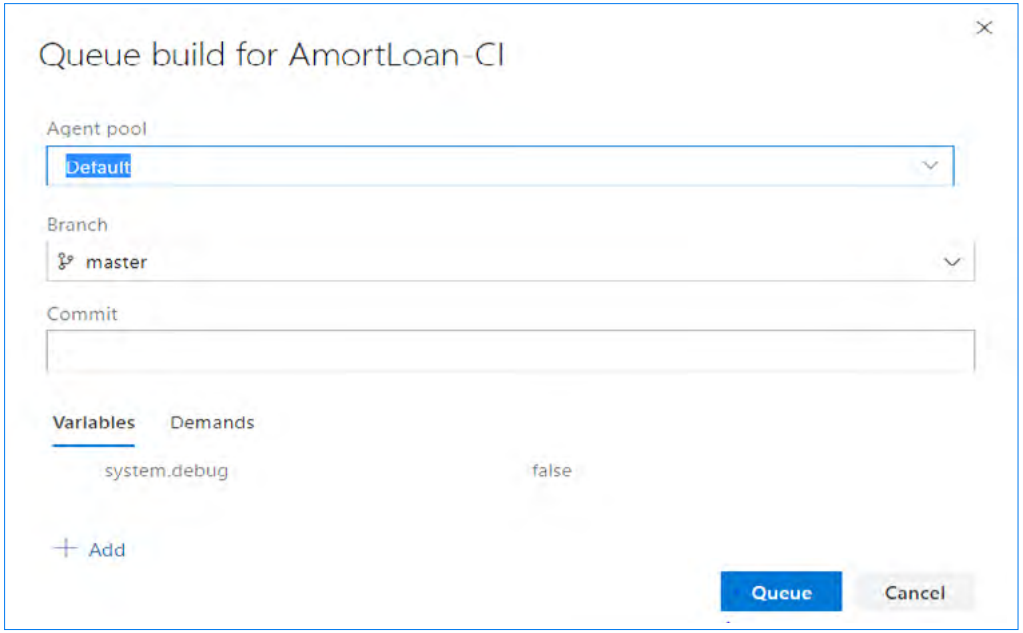


To add the Publish Build Artifacts Task, **click the + sign** next to Agent job 1 to add a new task. **Type "publish build"** into the search field on the Add task dialog and **click Add** for the Publish Build Artifacts task. Click on the newly added task to view it – we'll use the default settings.

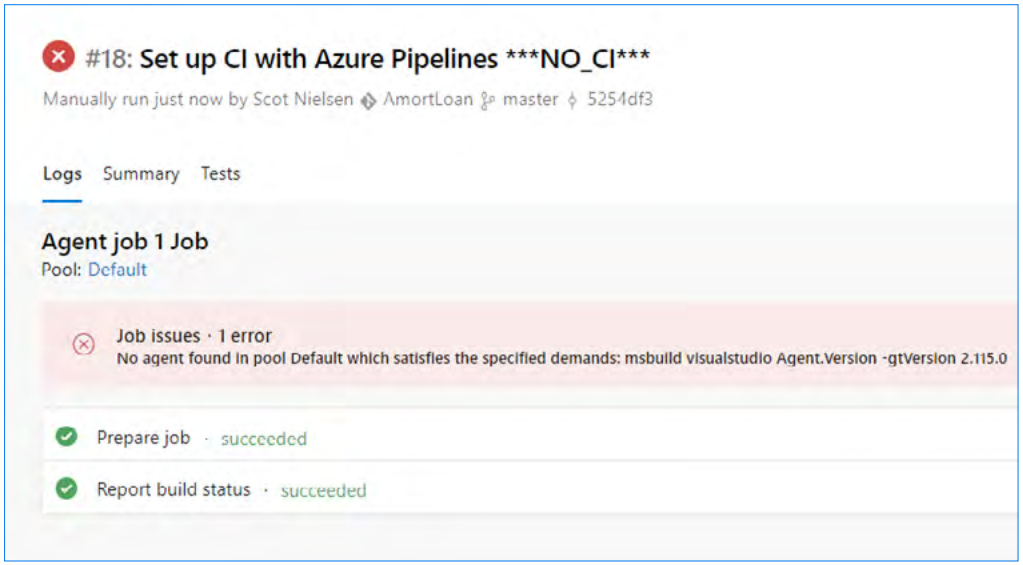
Now, **click on Pipeline** and change the Agent pool setting to **Default. Click Save and Queue** and select **Save** and accept defaults. Like this:



Let's perform a test build. **Click Queue** to start the pipeline process. Like this:



The build pictured will fail because Azure DevOps needs a build machine to compile your code. So let's set one up.



Step 4: Setting up an Azure DevOps build machine.

The machine we'll use to create our application will contain the Visual COBOL compiler tools. Things to note:

Azure will ask this machine to build the source code and run tests. It would usually be a standalone machine, running Visual COBOL, used specifically for CI purposes, and either on premise or in Azure

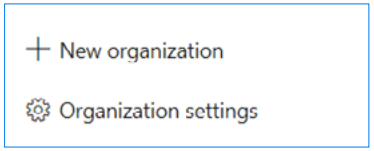
For this tutorial, we'll configure your machine to act as the CI build machine for Azure DevOps. You'll need to install a Build Agent - Microsoft software Azure DevOps will use to build your code.

Either download it onto your machine. You'll need to get the Build Agent from the Azure DevOps site.

Go to the Settings page for your organization - click the link in bottom left of the portal - and select Agent Pools.

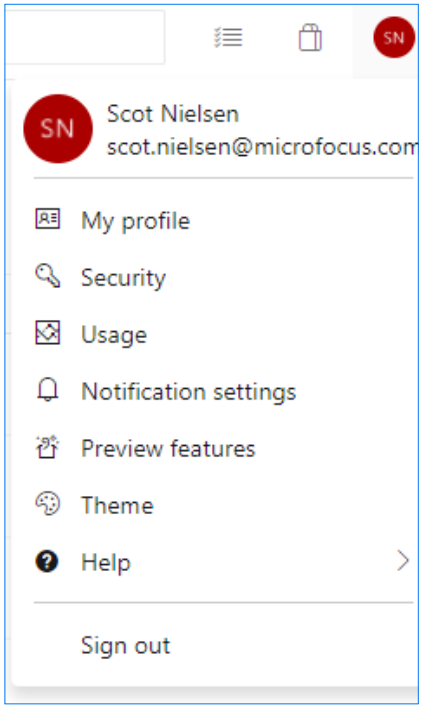
Make sure you click Organization Settings, not Project Settings to find the Build Agent.

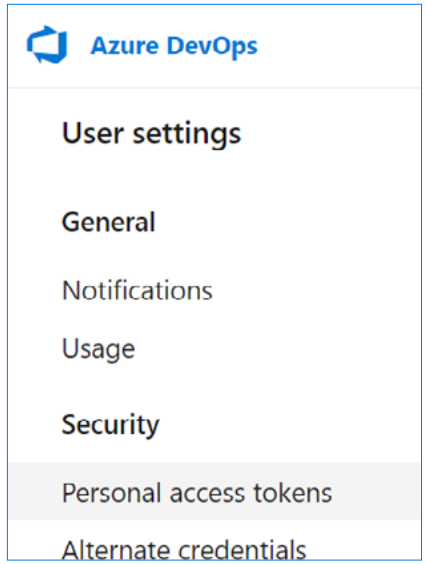
Once you've downloaded the Build Agent, extract the files to a folder on your machine.



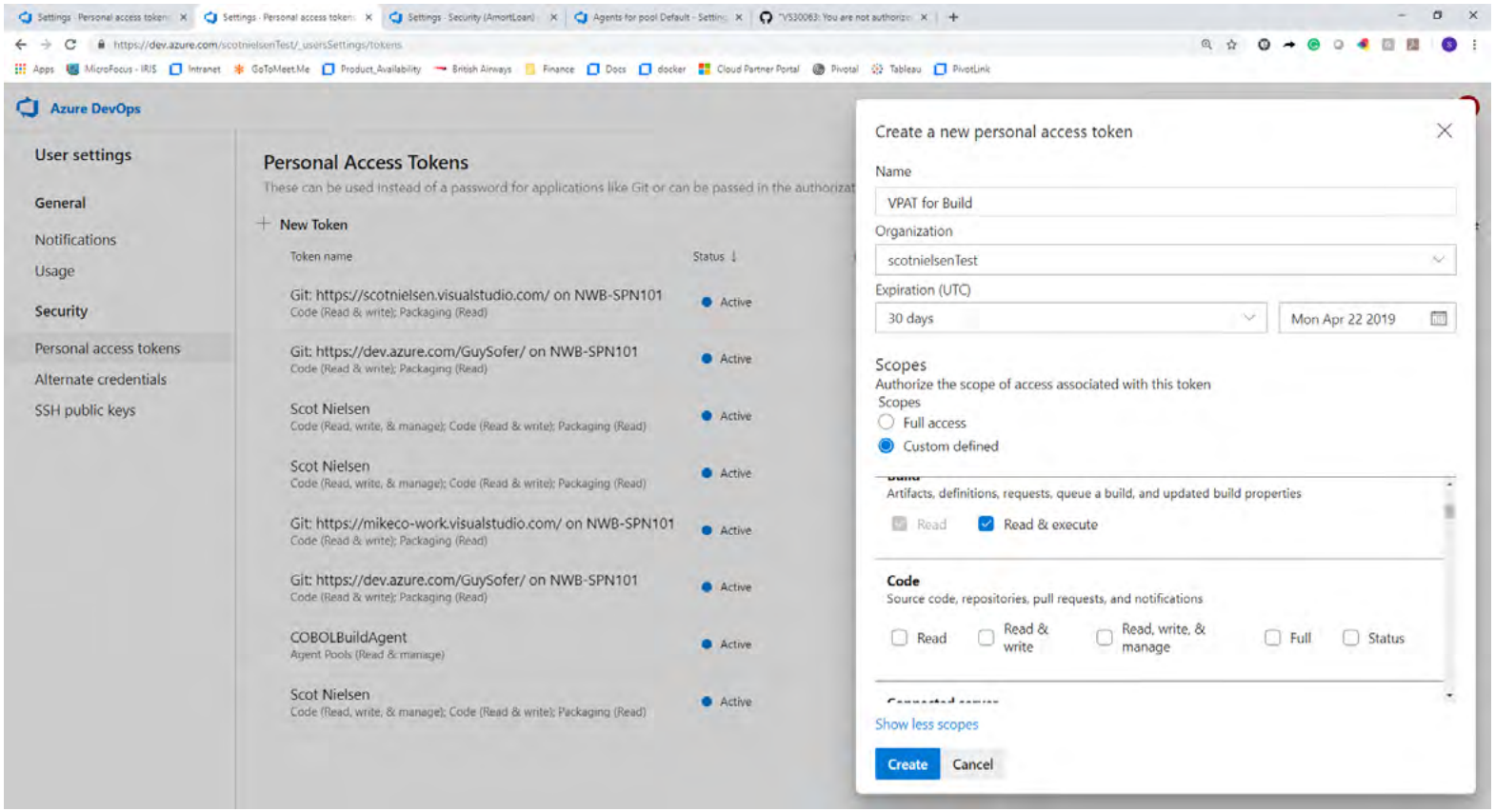
Got to the Organization Settings to locate the Build Agent software in the Agent Pools category.

Open your User Settings in Azure DevOps, **click Security** and **Personal Access Tokens**





Create a new token and name it **PAT for CI build machine**.
The PAT will require Read+Execute privileges for **Build** and Read+Manage for Agent Pools.
Don't see Build and **Agent Pools** in the list? **Click the Show all scopes** link at the bottom.
Save the PAT in text file for use in the next step.

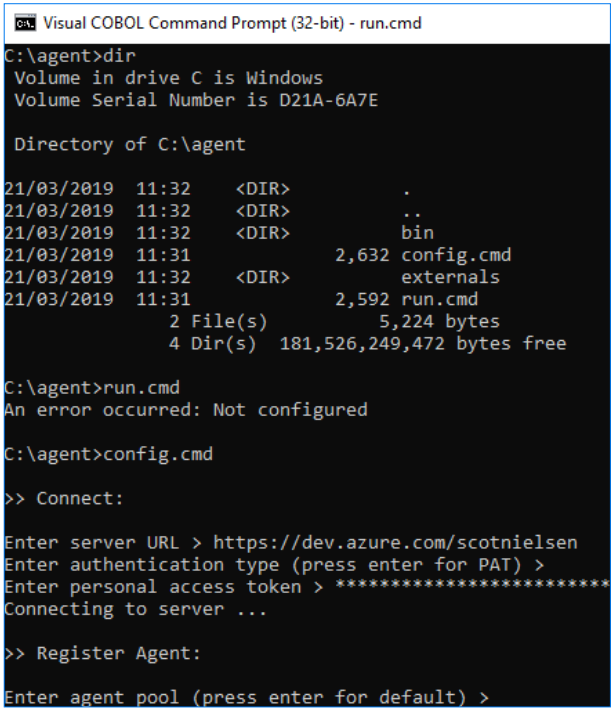


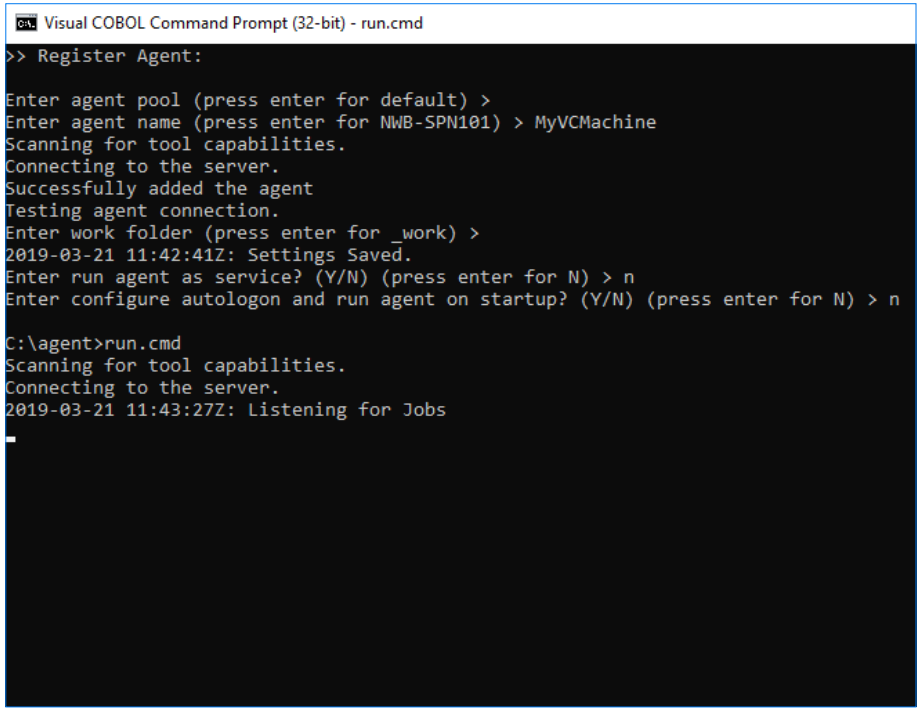
Make sure you give the PAT privileges for Build and Agent Pools

To configure the build agent, open a command prompt and navigate to the directory. Execute config.cmd and when prompted:

- Server url: <name of your Azure DevOps organization>
- eg. https://dev.azure.com/MyName
- Authentication type: PAT
- PAT: <paste your PAT created in the previous step>
- Enter Agent pool: <default>
- Agent name: VCBuildMachine

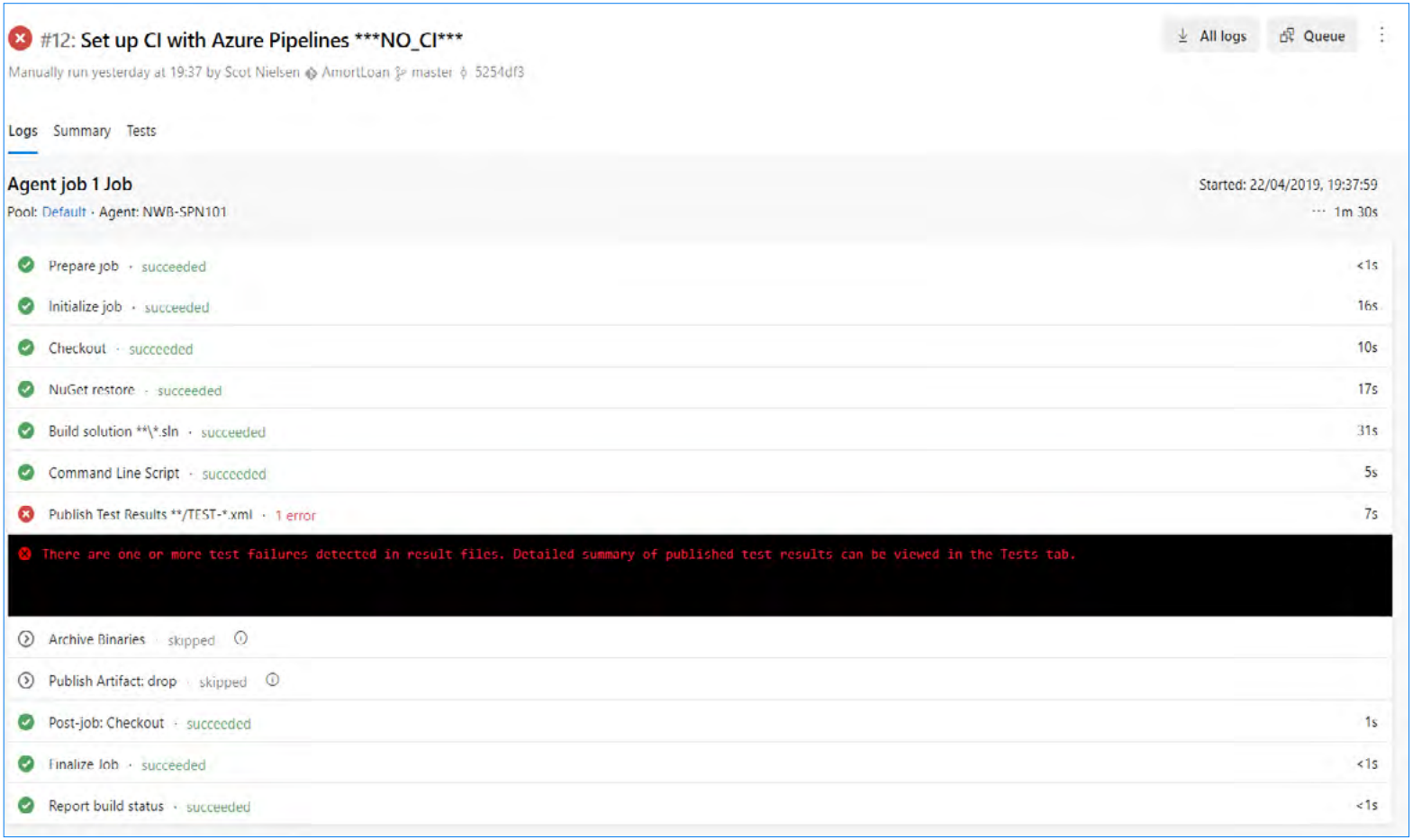
You can accept the rest of the defaults. You do not install as a service or enable AutoLogon. Here are the screengrabs.



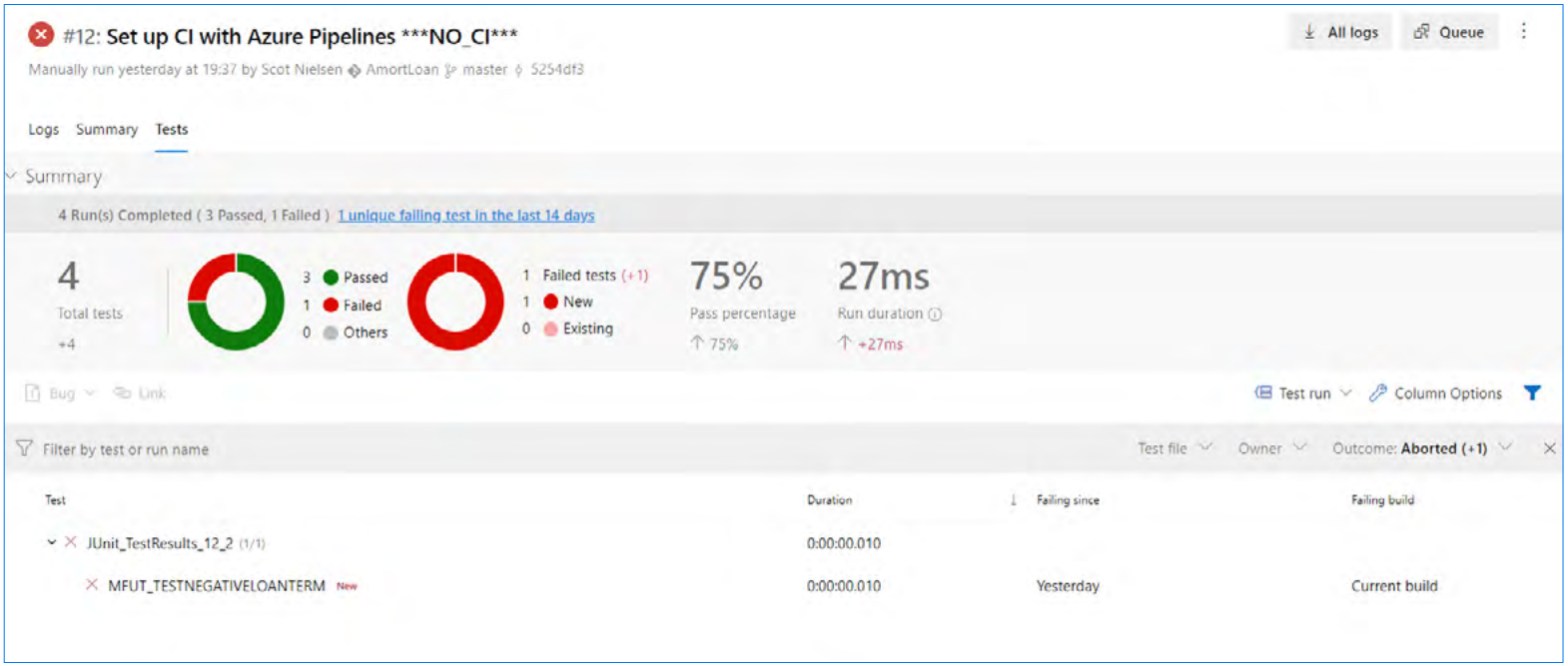


Let's **start the Build Agent**. From the command prompt, type: **run.cmd.** The build agent should start, and wait for jobs. Open the **Agent Pools page** under the Organization settings in the Azure DevOps site. Your build agent should be registered and online.

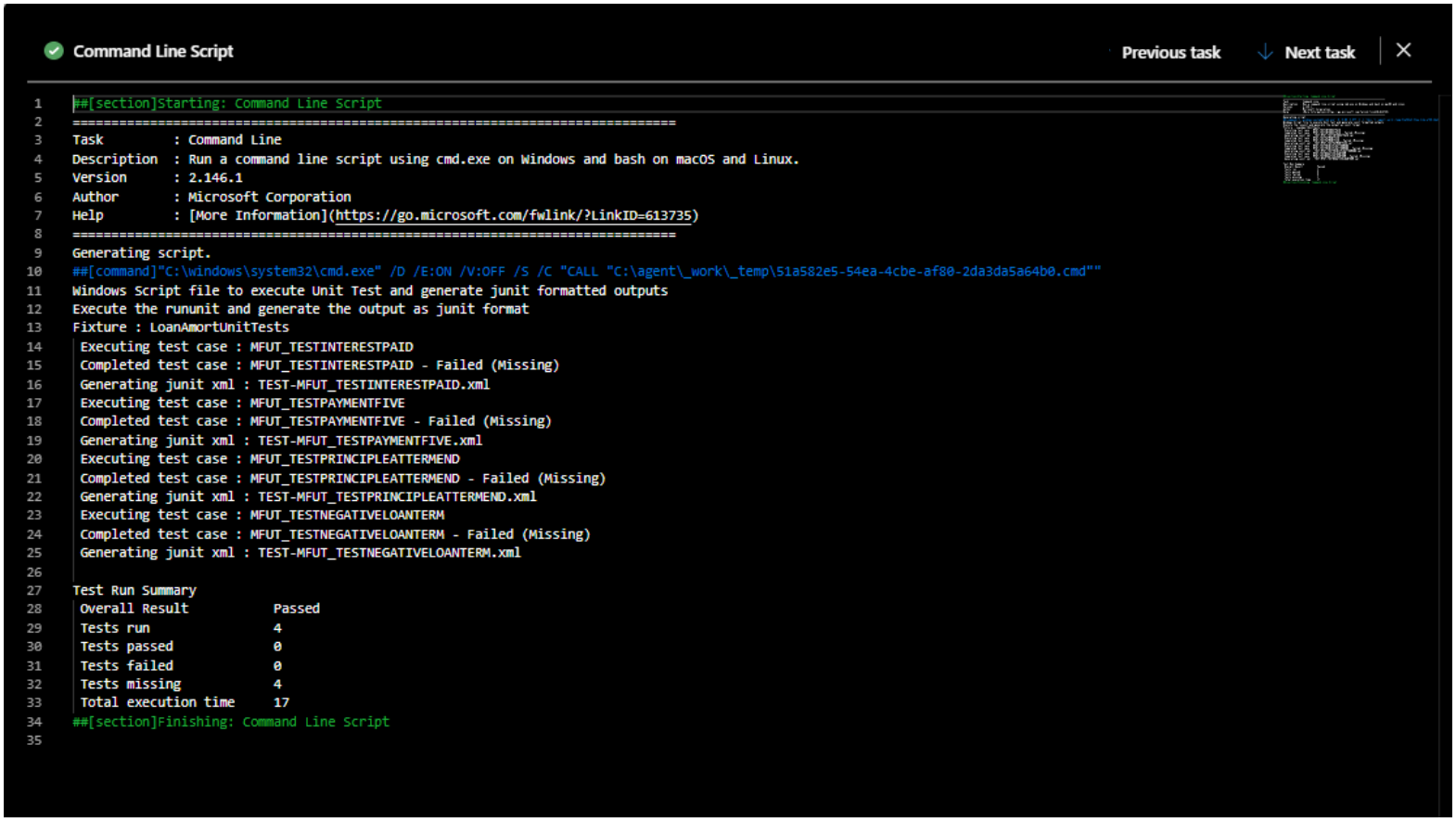
Now, **click Queue** on the Build pipelines page. The build should execute, and your build agent will accept the job. Monitor progress by clicking the build in the Azure pipelines page. Once everything has successfully compiled, a failing test should prompt an error. We'll fix this later. For now, **click Tests** to see more test run details. It will look like this.



Results from your build should look like this



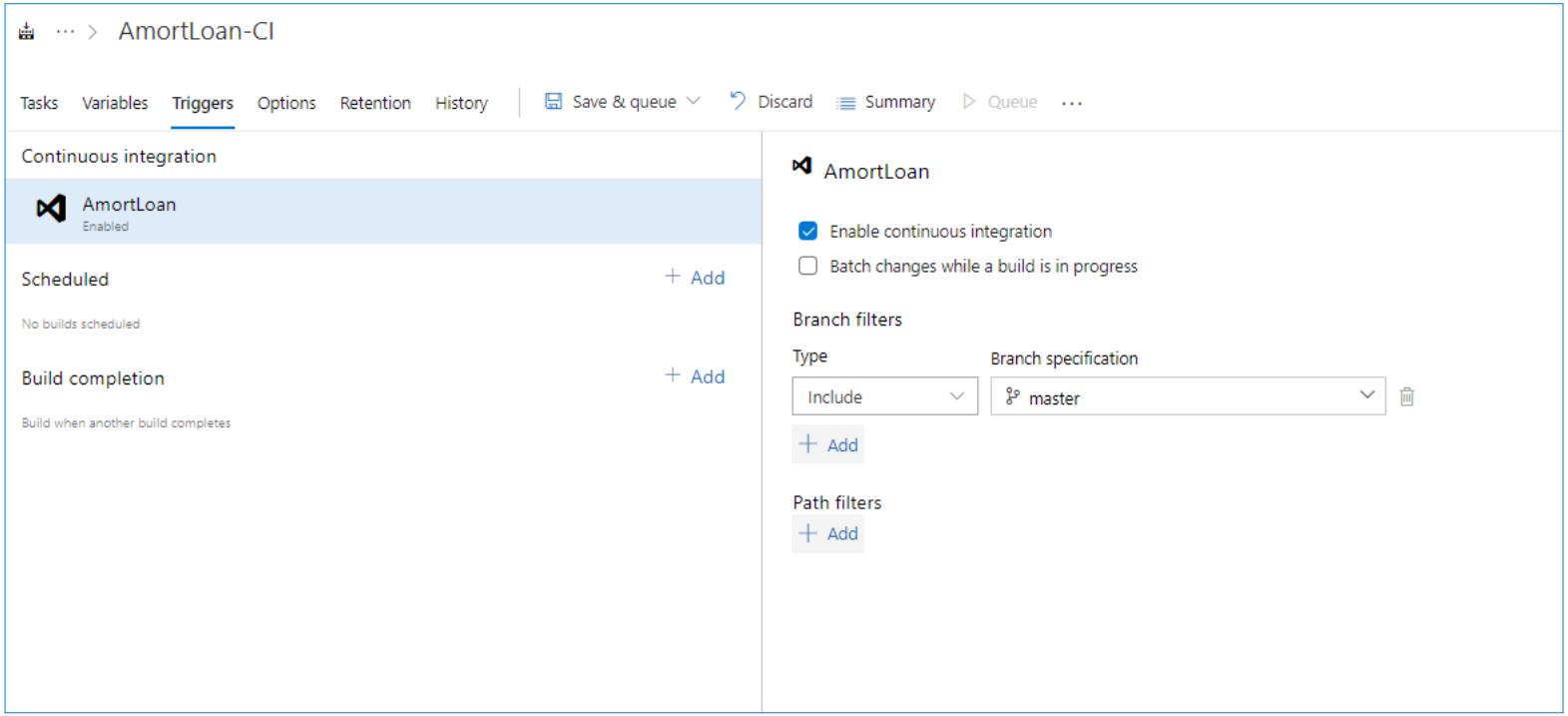
The pipeline will show you the results of your test run



Take a look through the results of your build logs by clicking on any of the steps

Let's step things up a little. Right now, your CI process is triggered manually. We are going to change this to an automated step whenever the code repo changes. First step is to edit the pipeline and **change the Triggers** section to enable continuous integration.

So, as you did in a previous step, make a small code change and commit this to the Azure Repo. The CI process should now automatically trigger. It will look like this.



Kudos.

You now have a CI system setup that will build and test your code following each commit.

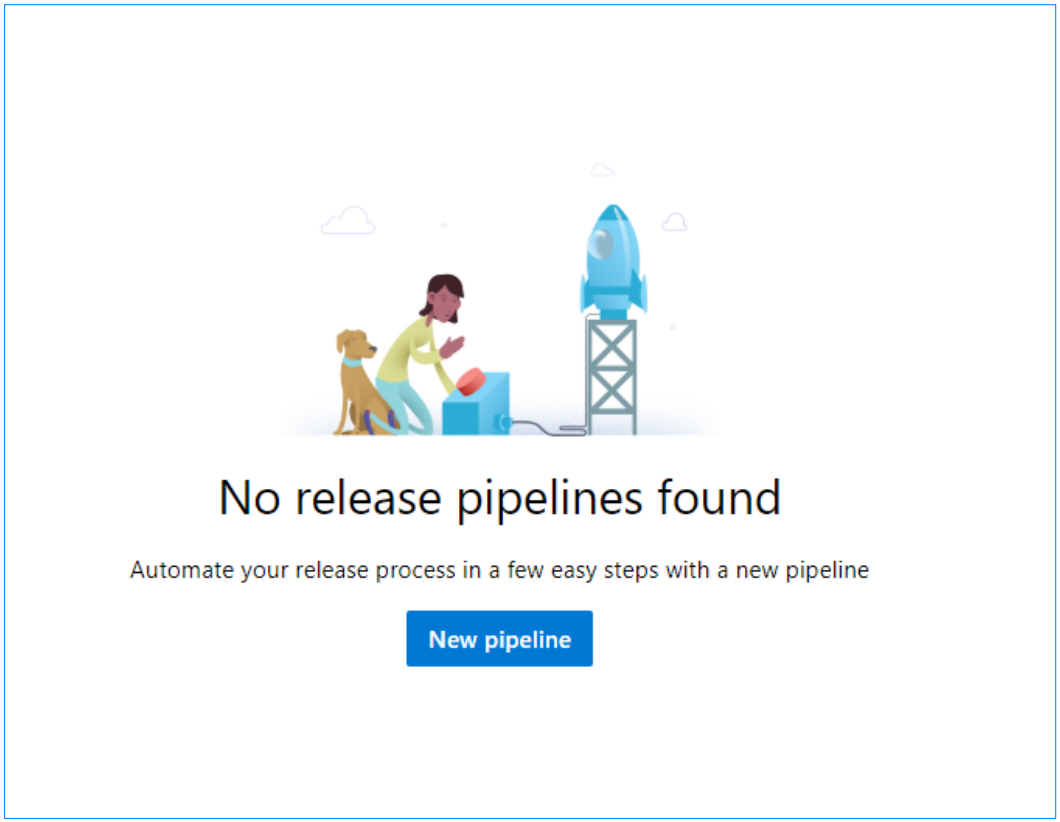
The next step is to set up a Continuous Delivery pipeline that will publish the newly committed code to your Azure function.

Step 5: Setup a continuous deployment pipeline to automate application deployment.

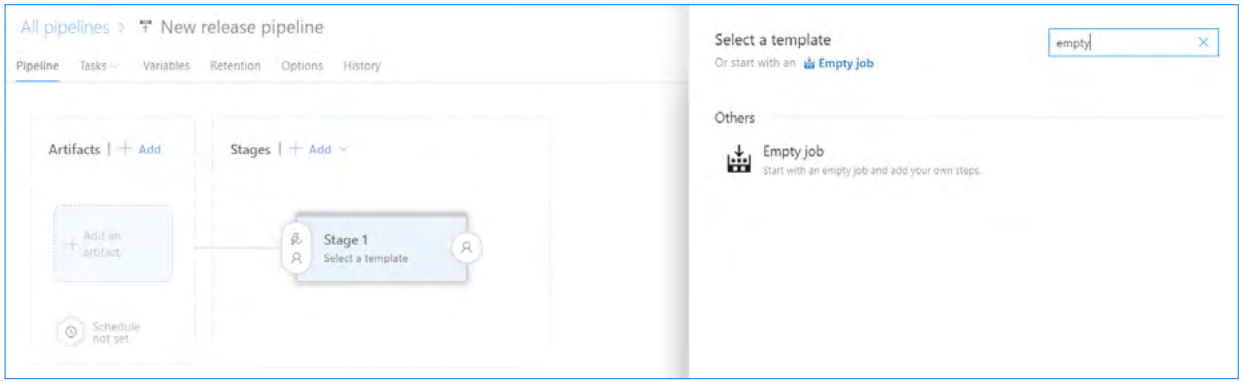
We're going to create a second pipeline that will execute following your CI pipeline.

It's a relatively straightforward process with very few steps.

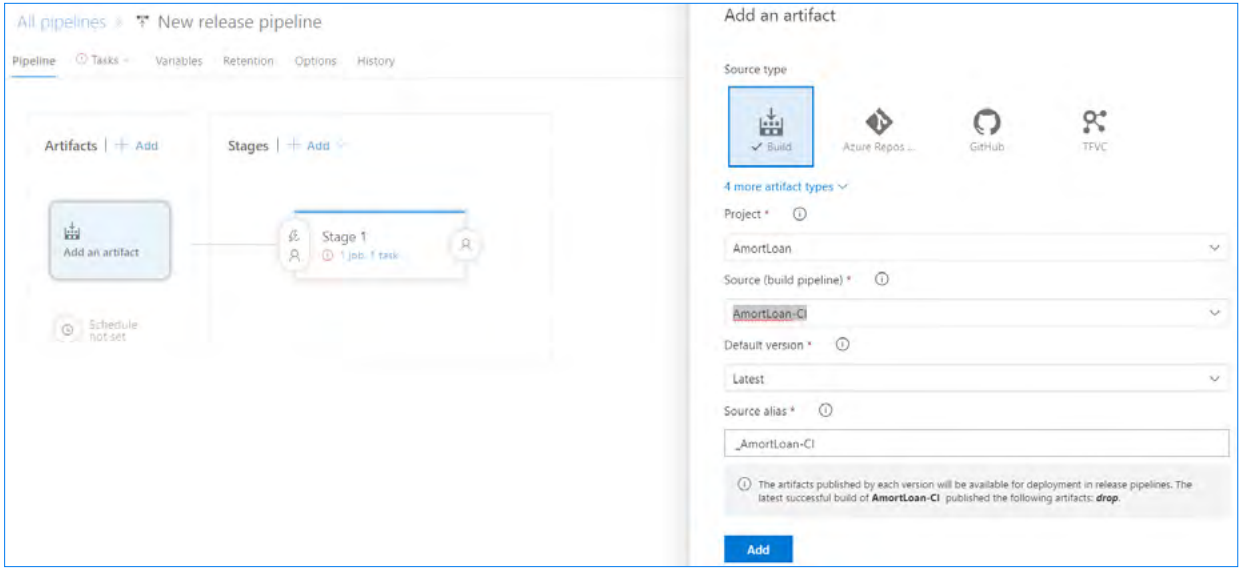
In the Azure DevOps project, **click Pipelines, Releases, New Pipeline.**



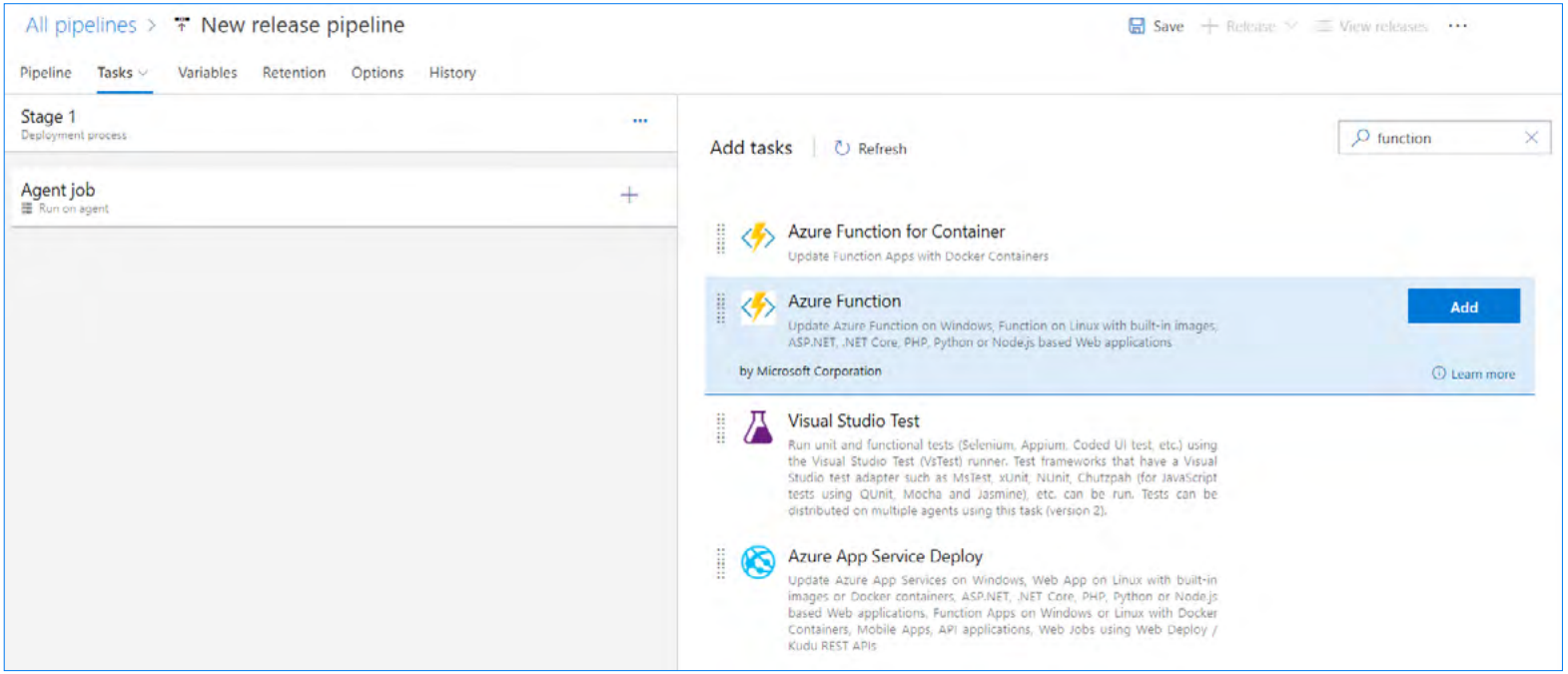
Type Empty in the search box and **Create and Empty Job**



Click Artifacts and specify the Azure DevOps project containing your sources

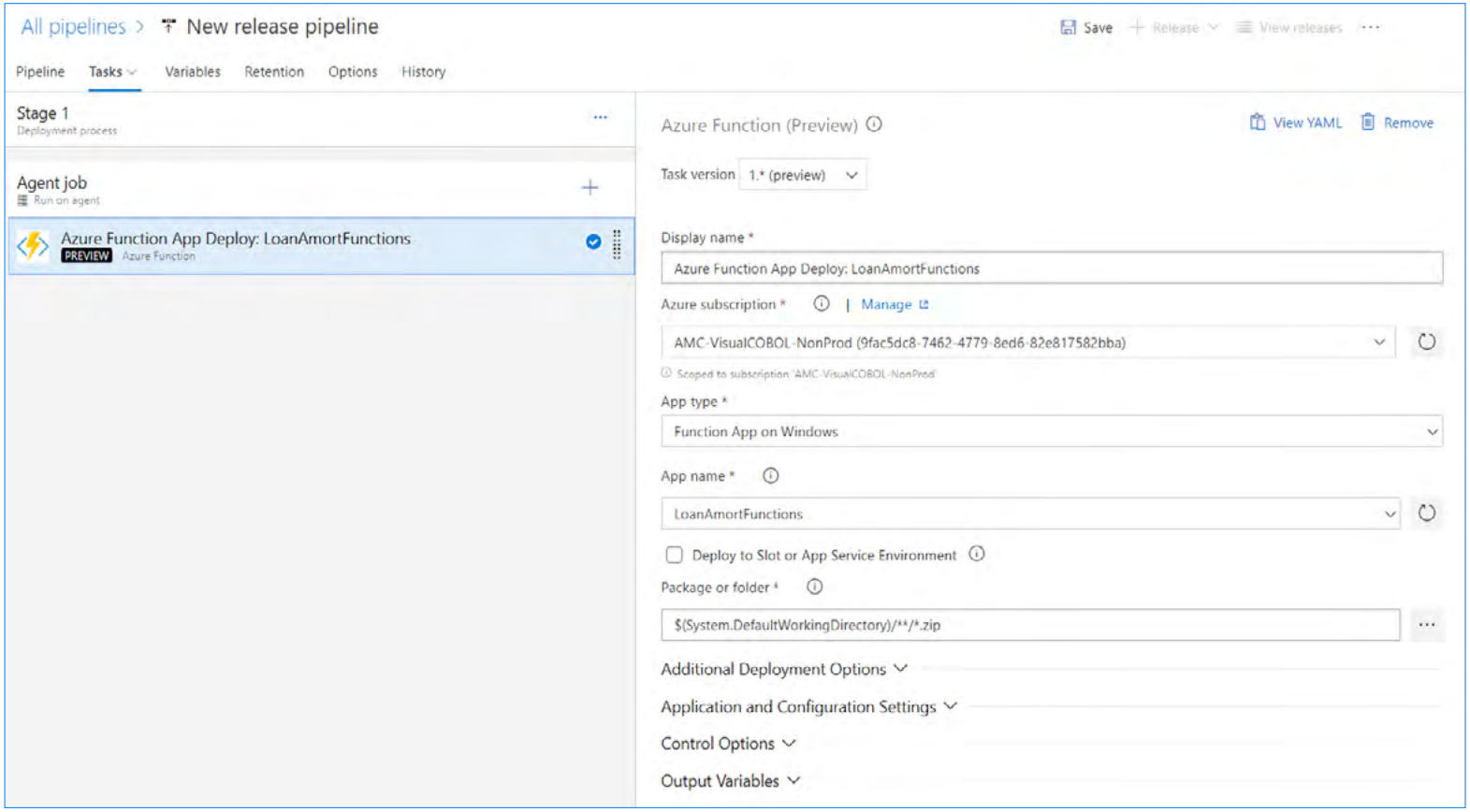


Click Tasks, then search for **Function**, and **Add an Azure Function** task

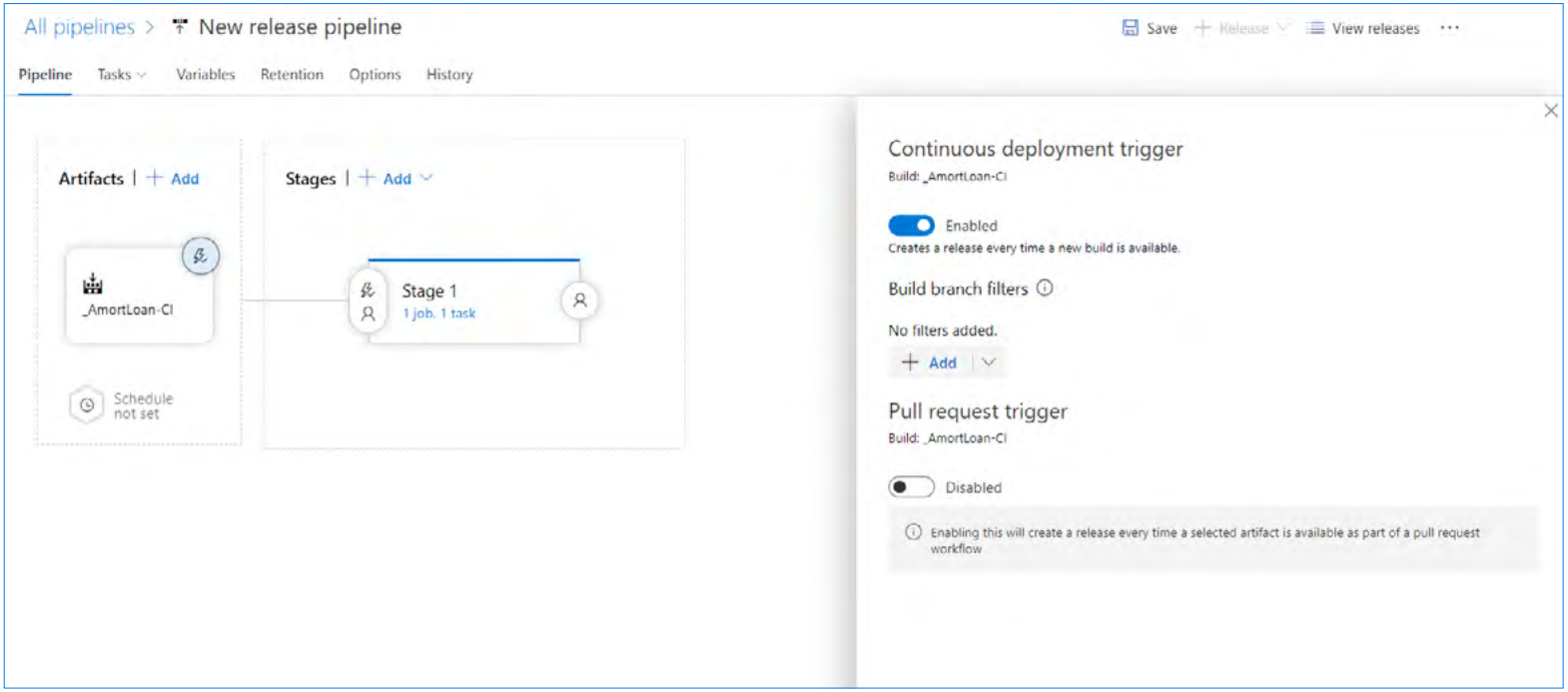


Configure your task with your subscription details and your function's name.

Click Authorize and log in to the DevOps portal when prompted.



To automate the pipeline, **click the lightning bolt** in the artefacts box and set the CD trigger to **Enabled**.



We're going to **queue a build** to test your pipeline. The test failure should cause the CD pipeline to fail to execute.

Fix it by either amending the code so that the test case passes and committing it to your repo, or change the tests step in the CI pipeline to continue on error.

Fix the failing test case by returning **-1** if a negative loan term is requested: like this.

```
PROCEDURE DIVISION USING LOANINFO
                         OUTDATA.

    IF LOANTERM = -1
       goback returning -1
    END-IF

    PERFORM CALC-PAYMENT
    MOVE WRK-PAYMENT TO DECPAYMENT
```

Make the change, run the tests, and commit your code if they pass.

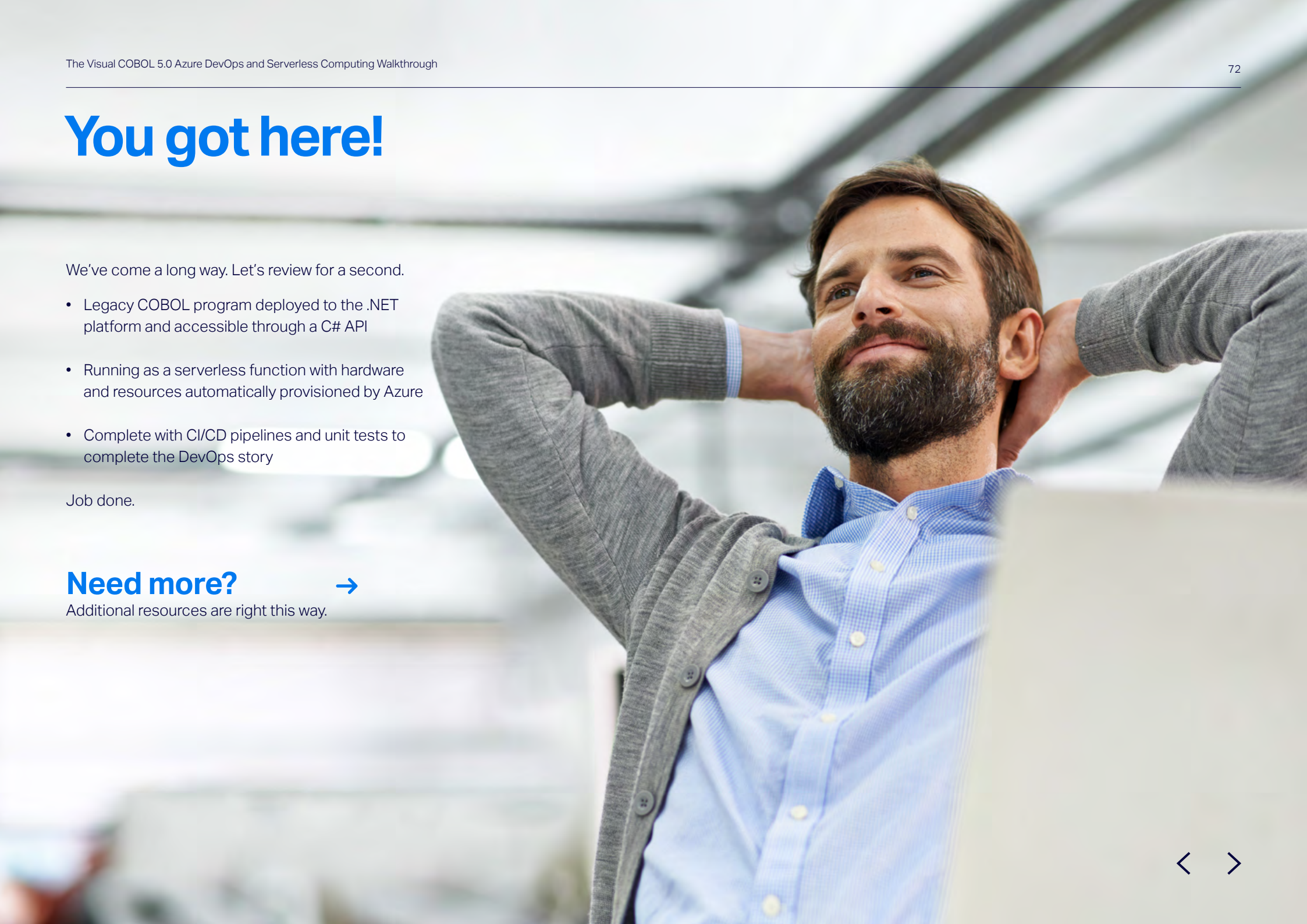
We can continue with errors by unchecking the **Fail is there are test failures** option.

Now, let's **clean up**.

Used the consumption plan for your Azure Function? You will only accumulate charges when the function is invoked. If you're done, take if offline or delete it to prevent invocation.

If you ever want to revisit this work, redeploy your function straight from Visual Studio.

You got here!



We've come a long way. Let's review for a second.

- Legacy COBOL program deployed to the .NET platform and accessible through a C# API
- Running as a serverless function with hardware and resources automatically provisioned by Azure
- Complete with CI/CD pipelines and unit tests to complete the DevOps story

Job done.

Need more? →

Additional resources are right this way.