

Dotnet Playbook



DEVOPS

Build, Test and Deploy a REST API with Azure DevOps



LES JACKSON



10TH MAR '19



2

Intro

Use Azure Devops pipelines with Github to continuously integrate and deploy a .Net Core REST API to Microsoft Azure.

What You'll Learn

By the end of this article you'll understand:



Dotnet Playbook

- What is “Azure DevOps”?
- Alternatives to Azure DevOps, (and Github and Azure!)
- How to connect Github and Azure to Azure DevOps
- How to configure the Azure DevOps Pipelines
- How to continuously integrate and deploy to a production Azure site

Ingredients

If you want to follow along with the examples in this tutorial you’ll need the following, (note everything is free unless stated otherwise):

- Visual Studio Code
- .Net Core SDK
- Postman (Optional you can just use a web browser)
- Azure Subscription (Free sign up. Mix of free and paid services)
- Azure DevOps Account (Free for individuals)

What is CI/CD?

Before we talk about specific technologies, (in this case Azure DevOps), we should take a minute to understand CI/CD concepts in general...

To talk about CI/CD, (don’t worry we’ll come onto what it stands for in a minute), is to talk about a pipeline of work”, or if you prefer another analogy: a production line, where

Dotnet Playbook



Clearly, this process will include a number of steps, most, (if not all), we will want to *automate*.

It's essentially about the faster realization of *business value* and is a central foundational idea of *agile software development*. (Don't worry I'm not going to bang that drum too much).

*You could argue, (and in fact I would!), that the *business requirements* are the starting point of the software “build” process. For the purposes of this article though, we'll use code as the start point of the journey.

Enough! What IS CI/CD?

Dotnet Playbook



merging those changes back into the main code “branch” by building and testing that code. As the name would suggest this process is continuous, triggered usually when developers “check-in” code changes to the *code repository*.

The whole point of CI is to ensure that the main, (or master), code branch remains healthy throughout the build activity, and that any new changes introduced by the multiple developers working on the code don’t conflict and break the build.

CD can be a little bit more confusing... Why? Well you’ll hear people using the both the following terms in reference to CD: *Continuous Deployment*, and *Continuous Delivery*.

What's the difference?

Well, if you think of Continuous Delivery as an extension of Continuous Integration it’s the process of automating the *release process*. It ensures that you can deploy software changes frequently and at the press of a button. Continuous Delivery stops just short of automatically pushing changes into production though, that’s where Continuous Deployment comes in...

Continuous deployment goes further than Continuous Delivery, in that code changes will make their way through to production without any *human intervention*, (assuming there are no failures in the CI/CD pipeline, e.g. failing tests).



So which is it?



Dotnet Playbook

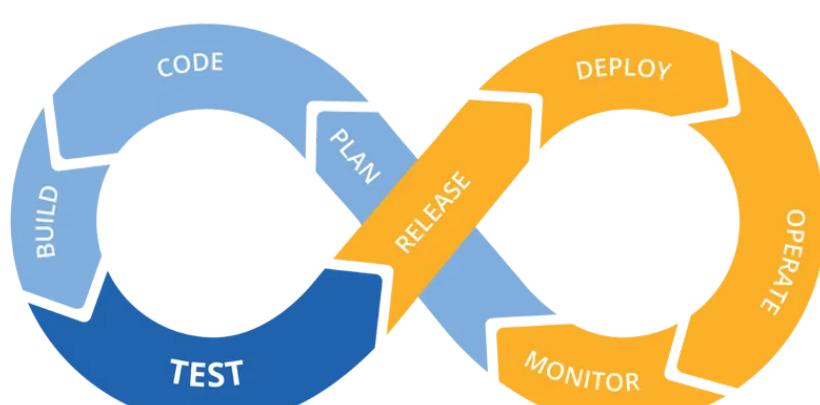
deploy software into production is a business decision, so the idea of Continuous Deployment is still overwhelming for most organizations....

The Pipeline

Google “CI/CD pipeline” and you will come up with a multitude of examples, I however like this one:



You may also see it depicted as an “infinite loop”, which kind of breaks the pipeline concept, but is none the less useful when it comes to understand “DevOps”:



The DevOps Infinite Loop

Coming back to the whole point of this article, (which if you haven’t forgotten is to detail how to use Azure DevOps), we are going to focus on the following elements of the pipeline:

Dotnet Playbook



Pipeline Stages we'll focus on

What is “Azure DevOps”?

Azure DevOps is cloud-based collection of tools that allow development teams to build and release software. It was previously called “Visual Studio Online”, and if you are familiar with the on-premise “Team Foundation Server Solution”, it’s basically that, but in the cloud... (an over-simplification – I know!)

For this article we are going to be focusing exclusively on “pipeline” features it has to offer, and leave the other aspects untouched, (for now).

Alternatives

There are various on-premise and cloud based alternatives: Jenkins is possibly the most “famous” of the on-premise solutions available, but you also have things like:

- Bamboo
- Team City
- Werker
- Circle CI

That list is by no means exhaustive, but for now, we’ll leave these behind and focus on Azure DevOps!

Our API Solution

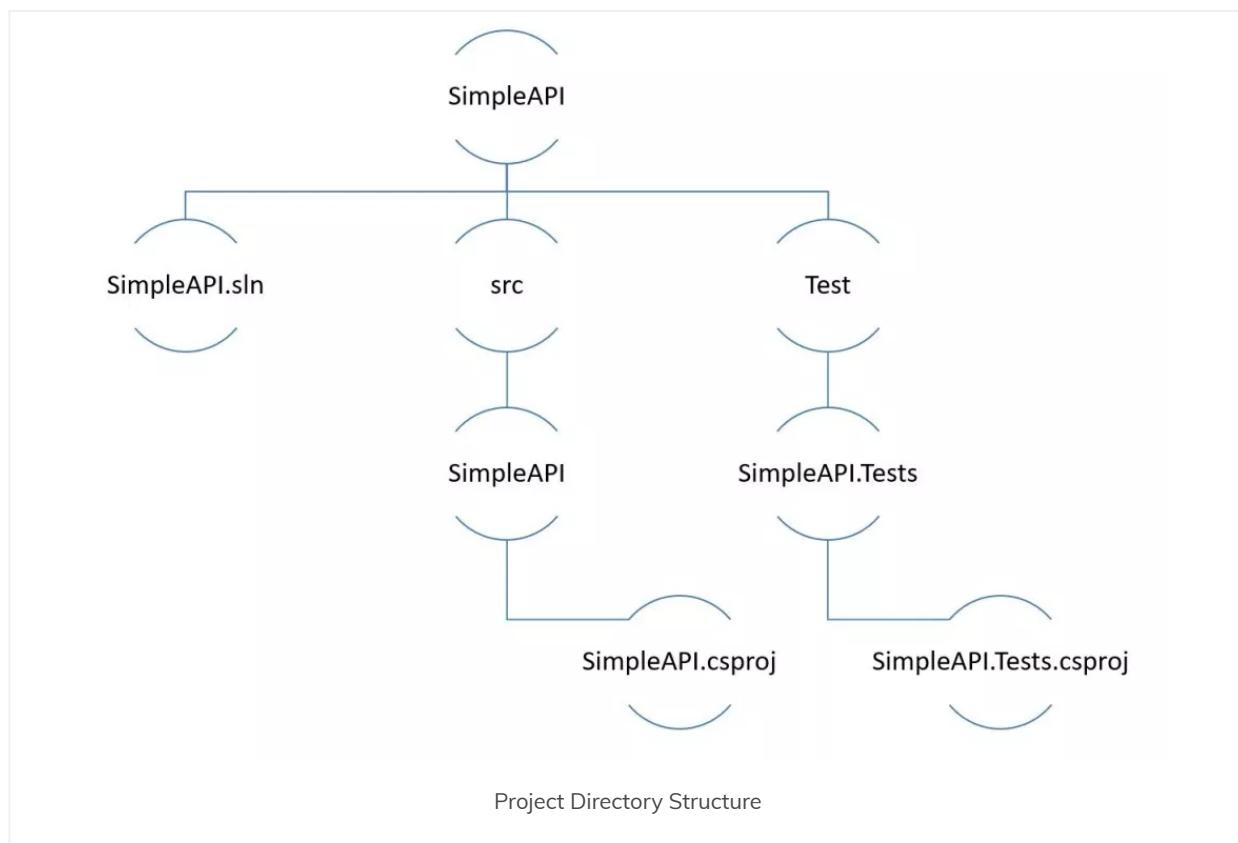
Dotnet Playbook



perfectly decent in proving out the concepts we need to.

Additionally we're going to create an XUnit Project to contain our unit tests, again these will be trivial, (this is not an article on unit testing), but again will be fully functional and demonstrate the core concept of automated testing in a CI/CD pipeline.

We'll "wrap" these 2 projects in a "solution", the folder structure of which is shown below:



Create Our Solution Components

Note: it is assumed that you have the .Net Core SDK already installed, (if not refer back the [Developing a REST API with ASP.Net Core](#) article for more detail on this and other set up requirements).



Dotnet Playbook

- Create 2 sub directories in here, named “src” and “test”
- Open a command prompt and change into the “src” directory
- Issue the following command

```
dotnet new webapi -n SimpleAPI
```

- This should create a sub folder, (SimpleAPI) in “src” containing our template API project
- Change into the “test” directory created above and issue the following command:

```
dotnet new xunit -n SimpleAPI.Tests
```

- This should create a sub folder, (SimpleAPI.Tests) in “test” containing our template Test project
- In the command window, change back into the main solution folder, (perform a directory listing to be sure) – you should see:



Dotnet Playbook

```
Directory: C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI

Mode                LastWriteTime       Length Name
----                -----          -----
d---l      3/03/2019  4:50 PM           0    src
d---l      3/03/2019  4:51 PM           0    test
```

Note: I'm using the terminal within VSCode that will respond to the "ls" command, the standard Windows Command Prompt will not- you'll need to use "dir". (Linux / Max users you'll be fine though!)

- Now issue the following command to create a solution file, (this is not strictly necessary but I like to have one for various reasons)

```
dotnet new sln --name SimpleAPI
```

- This should create a solution file called SimpleAPI.sln

We now want to associate both our “child” projects to our solution, to do so, issue the following command:

```
SimpleAPI.sln add src/SimpleAPI/SimpleAPI.csproj test/SimpleAPI.Tests/Simp
```



You should see output similar to the following

Dotnet Playbook



For more information on the [dotnet sln command](#) visit msdn.

Ok one last thing to do to is to place a “reference” to our SimpleAPI project in our SimpleAPI.Tests project, this will enable us to reference the SimpleAPI project and “test” it from our SimpleAPI.Tests project. You can either manually edit the SimpleAPI.Tests.csproj file, or type the following command:

```
dotnet add test/SimpleAPI.Tests/SimpleAPI.Tests.csproj reference sr...
```



Where the 1st project path is the “host” project and the 2nd project is the referenced project, if done successfully you should see something similar to that below:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> dotnet add test/SimpleAPI.Tests/SimpleAPI.Tests.csproj reference ..\..\src\SimpleAPI\SimpleAPI.csproj` added to the project.  
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI>  
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI>
```

You should also check the contents of the SimpleAPI.Tests .csproj file to ensure the reference is there.



Dotnet Playbook

```
<TargetFramework>netcoreapp2.2</TargetFramework>

<IsPackable>false</IsPackable>
</PropertyGroup>

<ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.9.0" />
    <PackageReference Include="xunit" Version="2.4.0" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.0" />
</ItemGroup>

<ItemGroup>
    <ProjectReference Include="..\..\src\SimpleAPI\SimpleAPI.csproj" />
</ItemGroup>

</Project>
```

You'll also notice the package references to xunit etc.

You can now build both projects, (ensure you are still in the root solution folder), by issuing:

```
dotnet build
```

Note: This is one of the advantages of using a solution file, (you can build both projects from here). Assuming all is well we have finished our initial projects set up!

Run Our API and Unit Test

Run Our API

If you've not done so already start VSCode and open the “solution” **folder**, this will open the solution and the child projects. You can use Visual Studio too, in that case you'd open the solution **file**.

Dotnet Playbook



```
dotnet run
```

This will start the API project, you should see something like:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\src\SimpleAPI> dotnet run
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\lesja\AppData\Local\ASP.NET\DataProtoc
Hosting environment: Development
Content root path: C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\src\SimpleAPI
Now listening on: https://localhost:5001
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

To see the api responding, open a web browser, (or use something like Postman), and browse to:

<https://localhost:5001/api/values>

Some points to note:

- Ensure the address your using matches what is being “listened on”
- The /api/values postfix is calling one of our controller actions in our “ValuesController”

Dotnet Playbook



OK, in the terminal window, use Ctrl + C to shutdown the server.

Unit Testing Our API

Ok, so I want to write 1 simple Unit test to test the response from one of our controller actions. It's super simple and rudimentary, but will illustrate the core concept of testing in CI/CD pipelines... The use-case I mention is also a valid one.

Again, just as this is not a tutorial in REST APIs, nor is this a tutorial in Unit Testing, so I won't go into it in depth. However if you've never heard of Unit Testing, these are tests that developers themselves write in order to test the low level units or functions of their code.

They are:

- Relatively small (or they should be)
- Quick to write
- Cheap
- Are executed first (see pyramid below)
- Abundant

You may also have heard about the “testing pyramid”, we'll a picture paints a thousand, words so here you go:

Dotnet Playbook



Image Copyright Marin Fowler

For more information visit [Martin Fowler's](#) site.

So, in your terminal, navigate not into our Test Project folder:

SimpleAPI/test/SimpleAPI.Tests

And issue the following command:

```
dotnet test
```

You should see something like:

Dotnet Playbook



```
Test run for C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\test\SimpleAPI.Tests\bin\Debug\netcoreapp3.1\SimpleAPI.Tests.dll
Microsoft (R) Test Execution Command Line Tool Version 15.9.0
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...

Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 0.9391 Seconds
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\test\SimpleAPI.Tests>
```

We haven't even done anything yet, and still a test is passing? Well yes as part of the template project we have a `UnitTest1` class with a single shell unit test that doesn't really test anything, so it passes.

We'll leave that there for now.

Open the the `UnitTest1.cs` file in VSCode, and add a new test underneath the existing empty test method, "Test1":

```
ValuesController controller = new ValuesController();
[Fact]
public void GetReturnsCorrectNumber()
{
    var returnValue = controller.Get(1);
    Assert.Equal("Les Jackson", returnValue.Value);
}
```

You'll also need to add an additional "using" directive at the top of the file:

Dotnet Playbook



Note this directive would not resolve had we not placed a reference to API project in the SimpleAPI.Tests.csproj file...

The UnitTests1.cs file should look like this now:

```
using System;
using Xunit;
using SimpleAPI.Controllers;

namespace SimpleAPI.Tests
{
    public class UnitTest1
    {
        [Fact]
        public void Test1()
        {

        }

        ValuesController controller = new ValuesController();[Fact]
        public void GetReturnsCorrectNumber()
        {
            var returnValue = controller.Get(1);
            Assert.Equal("Les Jackson", returnValue.Value);
        }
    }
}
```

Save the file and let's execute our 2 tests now:

```
dotnet test
```

Dotnet Playbook



```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\test\SimpleAPI.Tests> dotnet test
Build started, please wait...
UnitTests.cs(18,42): error CS0012: The type 'ControllerBase' is defined in an assembly that is not referenced. You must add a reference to assembly 'Microsoft.AspNetCore.Mvc.Core, Version=2.2.0.0, Culture=neutral, PublicKeyToken=adb9793829ddaae6'. [C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\test\SimpleAPI.Tests\SimpleAPI.Tests.csproj]
UnitTests.cs(18,31): error CS0012: The type 'ActionResult<>' is defined in an assembly that is not referenced. You must add a reference to assembly 'Microsoft.AspNetCore.Mvc.Core, Version=2.2.0.0, Culture=neutral, PublicKeyToken=adb9793829ddaae6'. [C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\test\SimpleAPI.Tests\SimpleAPI.Tests.csproj]
```

This can simply be resolved by adding the necessary assembly reference to the SimpleAPI.Tests .csproj file, as shown below:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>

    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.9.0" />
    <PackageReference Include="xunit" Version="2.4.0" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.0" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\..\src\SimpleAPI\SimpleAPI.csproj" />
  </ItemGroup>
</Project>
```

Note: After you save the file VSCode may ask to resolve dependencies – of course say yes!

Now run your tests again, (you should know how to do this now), you should see something like:



Dotnet Playbook

```
Starting test execution, please wait...
[xUnit.net 00:00:00.84]      SimpleAPI.Tests.UnitTest1.GetReturnsCorrectNumber [FAIL]
Failed  SimpleAPI.Tests.UnitTest1.GetReturnsCorrectNumber
  error Message:
    Assert.Equal() Failure
      (pos 0)
  Expected: Les Jackson
  Actual:   value
            (pos 0)
  Stack Trace:
    at SimpleAPI.Tests.UnitTest1.GetReturnsCorrectNumber() in C:\Users\lesja\OneDrive\Document
Total tests: 2. Passed: 1. Failed: 1. Skipped: 0.
Test Run Failed.
Test execution time: 1.2653 Seconds
PS C:\Users\Les Jackson\Documents\VSCode\SimpleAPI\test\SimpleAPI\SimpleAPI\
```

Yes using my name in the unit test is highly narcissistic!

What we have here is a failing unit test! Can you guess the reason why?

It's quite simple, our test is calling the Get(int id) method in our controller, (we pass in an arbitrary value of "1" but this could be any integer in this instance and would make no difference). It then "Asserts" that the value returned by the API should be "Les Jackson", when in fact it's passing back the value: "value" – hence it fails.

In order to get the test to pass we need to edit our controller method to look like that listed below:

```
// GET api/values/5
[HttpGet("{id}")]
public ActionResult<string> Get(int id)
{
    return "Les Jackson";
}
```

Save the file and re-run your tests, you should have success!



Dotnet Playbook

```
Test run for C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\test\SimpleAPI.Tests\bin\Debug\netcoreapp3.1\SimpleAPI.dll
Microsoft (R) Test Execution Command Line Tool Version 15.9.0
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...

Total tests: 2. Passed: 2. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 1.1273 Seconds
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\test\SimpleAPI.Tests> █
```

What you have, (kind of), done here is a form of development called: Test Driven Development, where developers will write the Unit Tests **first**, to test as yet un-written functionality they have yet to write. As they start to write the functions, the tests start to pass.

(Technically our method did already exit – but you get the point!)

With our API working and our vast suite of unit tests passing, it's time we placed everything under source control!

Place Under Source Control

If we look back at the build pipeline components we're focusing on, the first stage is “code”:



Now as well as this referring to the obvious, (basically what we've just covered in the last section), it also refers to the code “repository” that developers will submit or



Dotnet Playbook

In order for the build process to start, it has to fetch the code from somewhere – that somewhere is the code repository!

Source Control (Git & Github)

Now there are various code repository solutions out there, but by far the most common is Git, (and those based around Git), to such an extent that “source control” and Git are almost synonyms. Think about “vacuum cleaners” and “Hoover”, (or perhaps now Dyson), and you’ll get the picture.

Again, as with REST APIs and Unit Testing before, this is not a tutorial on Git, (there are plenty of those on the internet already!), so to looking to our friend Wikipedia it describes Git as:

A distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files.

Think about it as the “central source of truth” in relation to your code base.

While you can use Git in a distributed team environment, there are a number of companies that have taken it further placed “Git in the Cloud”, with such examples as:

- Github, (probably the most well recognised – and recently acquired by Microsoft)



Dotnet Playbook

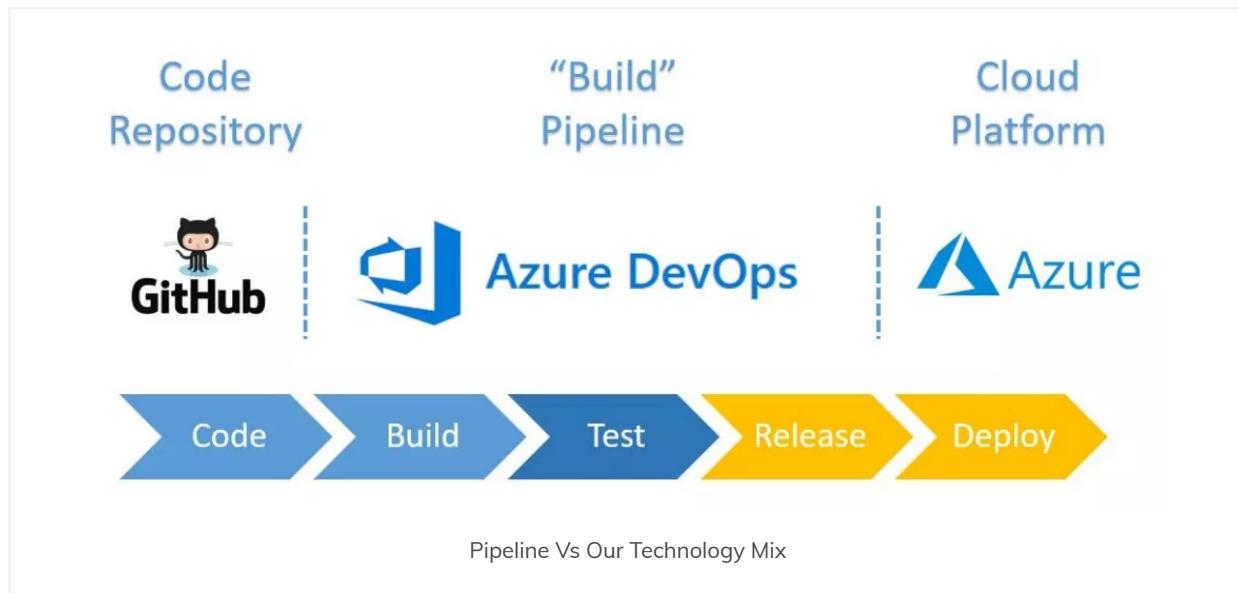
- Gitlabs

We're actually going to use both Git, (locally on our machine), and Github as part of this tutorial.

The Technology In Context

Now just before we move onto using Git and Github, I just wanted to say a few things on the technology, specifically the almost infinite choice and configuration options you have.

For this tutorial we're using the following mix:



Indeed, Azure DevOps actually comes with its own “code repository” feature, (Azure Repos), which means we could do away with Github...

So our mix could look like:



Dotnet Playbook



Azure DevOps



Azure



Or if you want to take Microsoft technologies out of the picture:

Code
Repository



“Build”
Pipeline



Cloud
Platform



Going further, you can even break down the Build -> Test -> Release -> Deploy etc. components into specific technologies... I'm not going to do that here.

The takeaway points I wanted to make were:

1. The relevant sequencing of technologies in our example
2. Make sure you understand the importance of the code repository
3. Be aware of the almost limitless choice of tech



Dotnet Playbook

Set Up Your Git Repo

Again in a terminal / command line in the *main solution directory*, type:

```
git --version
```

You should see something like:

The screenshot shows a terminal window with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is active, displaying the following text:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git --version
git version 2.15.1.windows.2
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> []
```

If you don't, and get an error you probably don't have git installed, (Google “Install git on <insert os here>” and you should be ok).

Assuming you get something similar to the above, (i.e. a version number!), we want to set up our *local git repository* by typing:

```
git init
```

This should initialize a local git repository in the solution directory that will track the code changes in a hidden folder called: *.git*

Dotnet Playbook



```
git status
```

This will show you all the “un-tracked” files in your directory, (basically files that are not under source control), at this stage that is everything:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .vscode/
    SimpleAPI.sln
    src/
    test/

nothing added to commit but untracked files present (use "git add" to track)
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> []
```

.gitignore File

Before we start to track our solution files, (and bring them under source control), there are certain files that you shouldn’t bring under source control, in particular files that are “generated” as the result of a build, primarily as they are surplus to requirements... (and they’re not “source” files’!)

In order to “ignore” these file types you create a file in your “root” solution directory called: `.gitignore`, (note the period ‘.’ at the beginning). Now this can become quite a personal choice on what you want to include or not, but I have provided an example that you can use, (or ignore altogether – excuse the pun!):

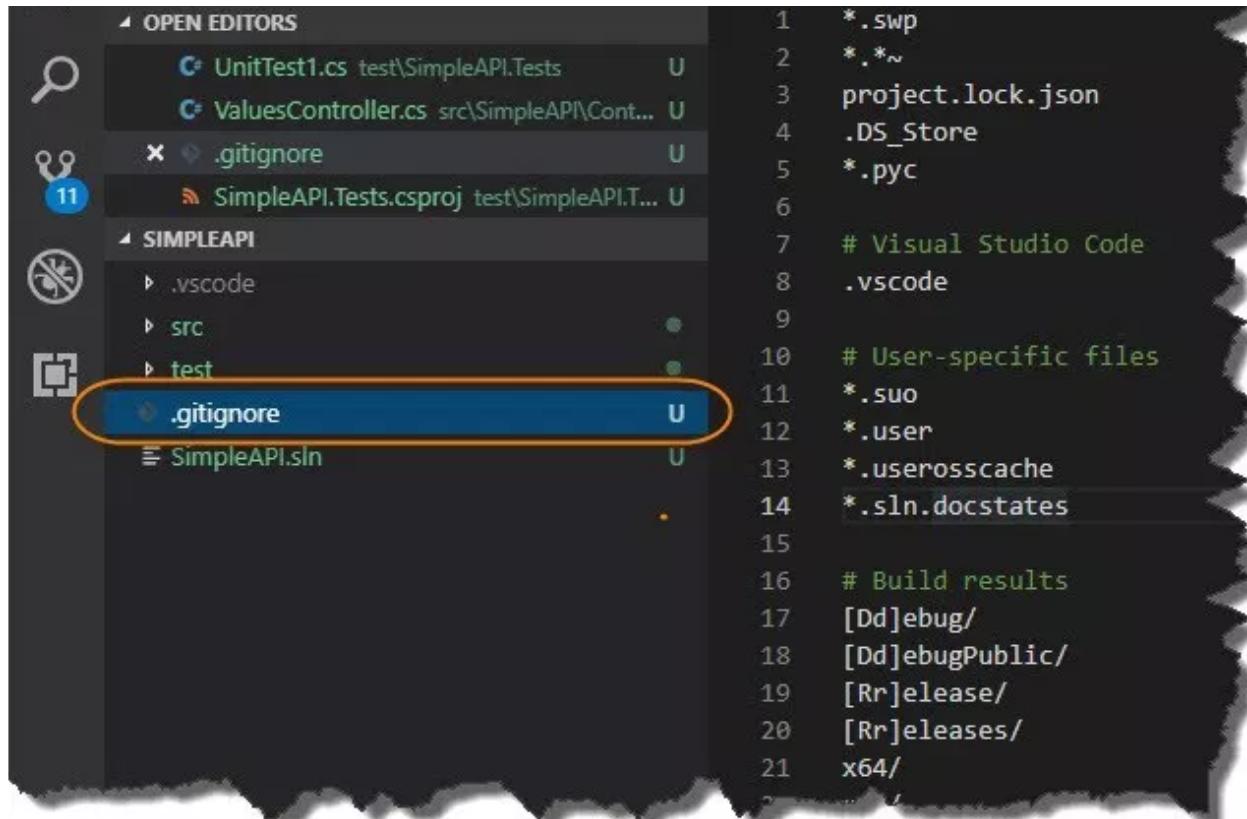
Dotnet Playbook



```
*.*~  
project.lock.json  
.DS_Store  
*.pyc  
  
# Visual Studio Code  
.vscode  
  
# User-specific files  
*.suo  
*.user  
*.userosscache  
*.sln.docstates  
  
# Build results  
[Dd]ebug/  
[Dd]ebugPublic/  
[Rr]elease/  
[Rr]eleases/  
x64/  
x86/  
build/  
bld/  
[Bb]in/  
[Oo]bj/  
msbuild.log  
msbuild.err  
msbuild.wrn  
  
# Visual Studio 2015  
.vs/
```

So if you want to use a .gitignore file, create one, and pop it in your solution directory, as I've done below, (this shows the file in VSCode):

Dotnet Playbook



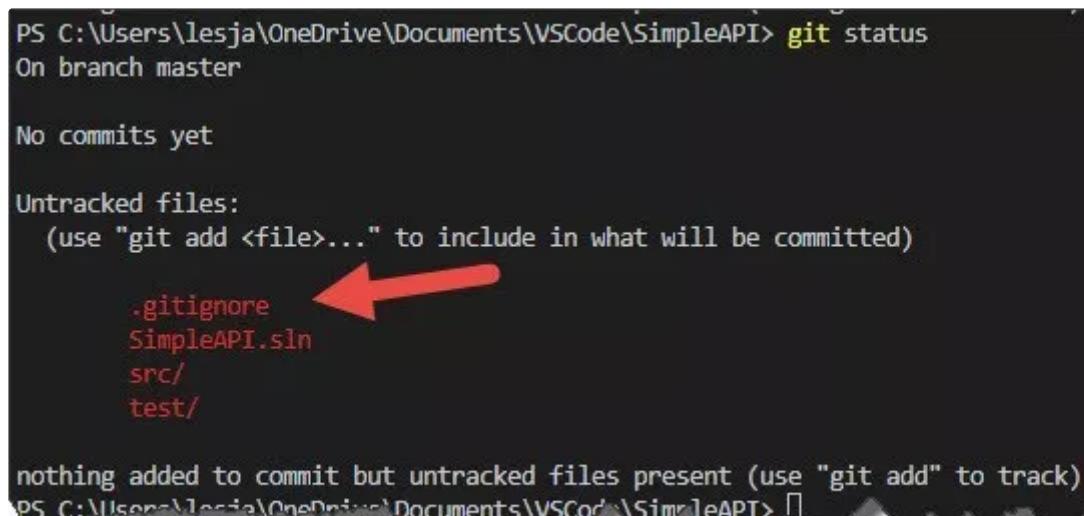
A screenshot of the Visual Studio Code interface. On the left is the sidebar with icons for search, diff, 11 pending changes, and a folder named 'SIMPLEAPI'. Under 'SIMPLEAPI', there are three folders: '.vscode', 'src', and 'test'. Inside 'test' is a file named '.gitignore'. This file is highlighted with a blue selection bar and has a yellow oval around it. To the right of the sidebar is the main editor area showing a list of git ignore patterns:

```

1 *.swp
2 *.*~
3 project.lock.json
4 .DS_Store
5 *.pyc
6
7 # Visual Studio Code
8 .vscode
9
10 # User-specific files
11 *.suo
12 *.user
13 *.userosscache
14 *.sln.docstates
15
16 # Build results
17 [Dd]ebug/
18 [Dd]ebugPublic/
19 [Rr]elease/
20 [Rr]eleases/
21 x64/
22 ...

```

Type “git status” again, and you should see this file now as one of the “un-tracked” files:



```

PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore ←
    SimpleAPI.sln
    src/
    test/

nothing added to commit but untracked files present (use "git add" to track)
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI>

```

Track and Commit Your Files

Ok we want to track “everything”, (except those files ignored!), to so type:

Dotnet Playbook



Followed by:

```
git status
```

You should see:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git add .
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:  .gitignore
    new file:  SimpleAPI.sln
    new file:  src/SimpleAPI/Controllers/ValuesController.cs
    new file:  src/SimpleAPI/Program.cs
    new file:  src/SimpleAPI/Properties/launchSettings.json
    new file:  src/SimpleAPI/SimpleAPI.csproj
    new file:  src/SimpleAPI/Startup.cs
    new file:  src/SimpleAPI/appsettings.Development.json
    new file:  src/SimpleAPI/appsettings.json
    new file:  test/SimpleAPI.Tests/SimpleAPI.Tests.csproj
    new file:  test/SimpleAPI.Tests/UnitTest1.cs

PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI>
```

These files are being tracked and are “staged” for commit.

Finally, we want to “commit” the changes, (essentially lock them in), by typing:

Dotnet Playbook



This is commits the code with a note, (or “message”, hence the -m switch), about that particular commit. You typically use this to describe the changes or additions you have made to the code, (more about this later), you should see:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git commit -m "Initial Commit"
[master (root-commit) aaba667] Initial Commit
 11 files changed, 307 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 SimpleAPI.sln
 create mode 100644 src/SimpleAPI/Controllers/ValuesController.cs
 create mode 100644 src/SimpleAPI/Program.cs
 create mode 100644 src/SimpleAPI/Properties/launchSettings.json
 create mode 100644 src/SimpleAPI/SimpleAPI.csproj
 create mode 100644 src/SimpleAPI/Startup.cs
 create mode 100644 src/SimpleAPI/appsettings.Development.json
 create mode 100644 src/SimpleAPI/appsettings.json
 create mode 100644 test/SimpleAPI.Tests/SimpleAPI.Tests.csproj
 create mode 100644 test/SimpleAPI.Tests/UnitTest1.cs
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> []
```

A quick additional “git status” and you should see:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git status
On branch master
nothing to commit, working tree clean
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> []
```

We have basically placed our solution under *local* source control and have committed all our “changes” to our master branch in our 1st commit.

Dotnet Playbook



pause this tutorial here, and do a bit of Googling to find some additional resources. It's a fairly big subject on its own – I just don't have the time to cover it more here.

Set Up Your Github Repo

Ok so the last section took you through the creation of a local Git repository, and that's fine for tracking code changes on your local machine. However if you're working as part of a larger team, or even as an individual programmer and want to make use of Azure DevOps, (as we do!), we need to configure a "remote Git repository" that we will:

- Push to from our local machine
- Link to an Azure DevOps Build Pipeline to kick off the build process

Jump over to: <https://github.com>, (and if you haven't already – sign up for an account), you should see your own landing page once you've created an account / logged in, here's mine:

Dotnet Playbook



Overview Repositories 6 Projects 0 Stars 0 Follow

Popular repositories

- VP-0-REST-Client
 - C#
 - ★ 4
 - 5
- VP-10-Post-to-a-REST-API
 - C#
- VP-17-Intro-to-Entity-Framework
 - C#
- S02E01...
 - C#
- VP-15-Tel...
 - C#
- VP-1-W...
 - C#

Set your status

Les Jackson
binarythistle

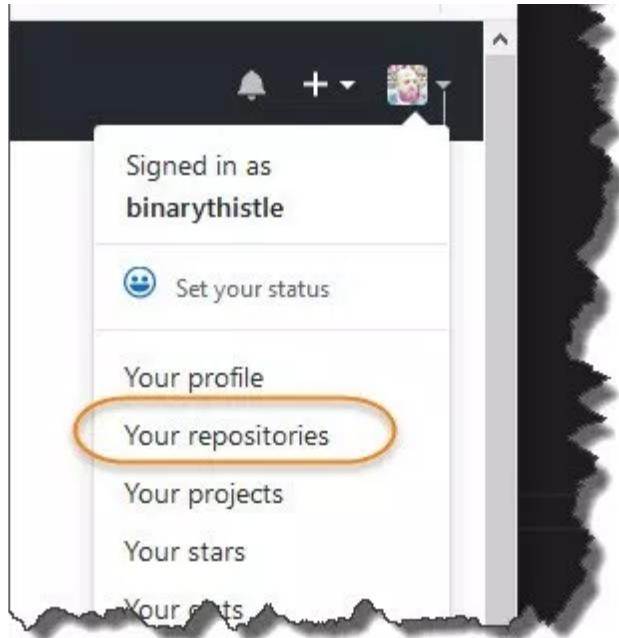
📍 Melbourne

Create a GitHub repository

In the top right hand of the site click on your face, (or whatever the default is if you're not a narcissist), and select "Your repositories":

Dotnet Playbook

≡



The Click “New” and you should see the “Create a new repository” screen:

Dotnet Playbook



The screenshot shows the GitHub repository creation interface. At the top, it says "Owner" with a dropdown set to "binarythistle" and "Repository name" with "SimpleAPI" entered. A green checkmark icon is next to the repository name input field. Below this is a tip: "Great repository names are short and memorable. Need inspiration? How about [congenial-meme?](#)". The "Description (optional)" field is empty. Under "Visibility", "Public" is selected (indicated by a filled circle) and "Anyone can see this repository. You choose who can commit." is shown. "Private" is also listed with "You choose who can see and commit to this repository.". There is an unchecked checkbox for "Initialize this repository with a README" with the note "This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.". Below this are two dropdowns: "Add .gitignore: None" and "Add a license: None". At the bottom is a large green "Create repository" button.

Give the repository a name, (I just named mine after the API Solution, but you can call it anything you like), and select either Public or Private. It doesn't matter which you select but remember your choice as this is important later.

Tip: If you want my advice, for projects like these – I'd just leave public.

Then click “Create Repository”, you should see:

Dotnet Playbook



Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH <https://github.com/binarythistle/SimpleAPI.git>

Get started by creating a new file or uploading an existing file. We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# SimpleAPI" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/binarythistle/SimpleAPI.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/binarythistle/SimpleAPI.git
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

This page details how you can now link and push your local repository to this remote one, (the section I've circled in orange). So copy that text and paste it into your terminal window, (you need to make sure you're still in the root solution folder we were working in above):

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git remote add origin https://github.com/binarythistle/SimpleAPI.git
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git push -u origin master
Counting objects: 19, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (19/19), 4.05 KiB | 1.35 MiB/s, done.
Total 19 (delta 0), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/binarythistle/SimpleAPI.git
 * [new branch] master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI>
```

IMPORTANT: You may get asked to authenticate to Github when you issue the 2nd command:

Dotnet Playbook



I've had some issues with this on Windows until I updated the "Git Credential Manager for Windows", after I updated it was all smooth sailing. Google "Git Credential Manager for Windows" if you're having authentication issues and install the latest version.

What Just Happened?

Well in short:

- We "registered" our remote Github repo with our local repo (1st command)
- We then pushed our local repo up to Github (2nd command)

Note: the 1st command line only needs to be issued once, the 2nd one we'll be using more throughout the rest of the tutorial.

If you refresh your Github repository page, instead of seeing the instructions you just issued, you should see our solution!

Dotnet Playbook

No description, website, or topics provided.

Manage topics

1 commit 1 branch 0 releases 1 contributor

Branch: master New pull request Create new file Upload files Find file Clone or download

binarythistle Initial Commit	Latest commit aaba667 37 minutes ago
src/SimpleAPI	Initial Commit 37 minutes ago
test/SimpleAPI.Tests	Initial Commit 37 minutes ago
.gitignore	Initial Commit 37 minutes ago
SimpleAPI.sln	Initial Commit 37 minutes ago

Help people interested in this repository understand your project by adding a README. Add a README

You'll notice "Initial Commit" as a comment against every file and folder – seem familiar?

Create a Build Pipeline

Finally we get to Azure DevOps!!



Ok, as with GitHub, jump over to: <https://dev.azure.com> and create an account if you don't have one – they're free so no excuses!

Once you have signed in / signed up, click on the "Create project" button to, surprise-surprise, create a project, (this project will contain all our build pipeline stuff)!

Dotnet Playbook



Project name *

Description

Visibility

Public Anyone on the internet can view the project. Certain features like TFVC are not supported.	Private Only people you give access to will be able to view this project.
--	--

By creating this project, you agree to the Azure DevOps [code of conduct](#)

^ Advanced

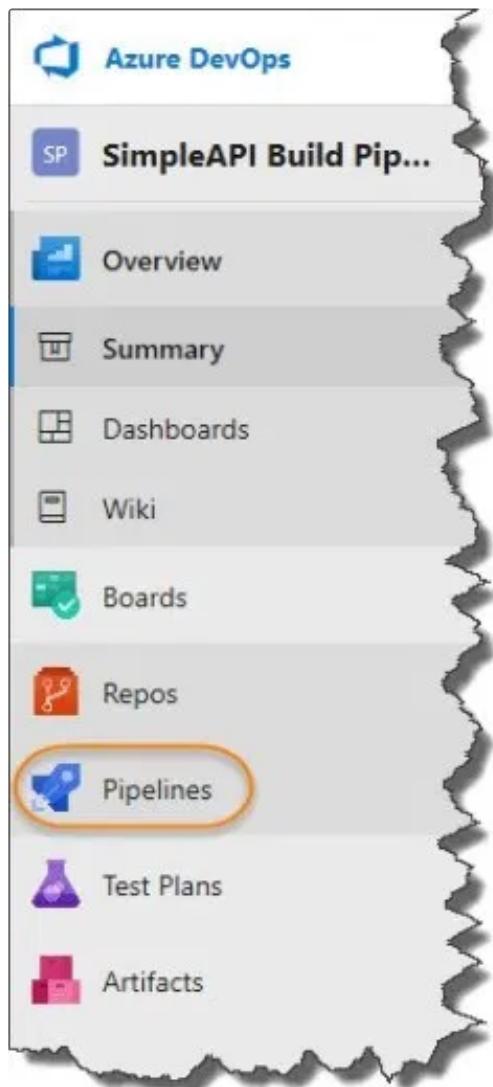
Version control [?](#)

Work item process [?](#)

Make sure:

- You give it a name, (something meaningful)
- Select the same “visibility” that your Github repo has – remember?
 - It will complain if the 2 are different
- Version control set to Git

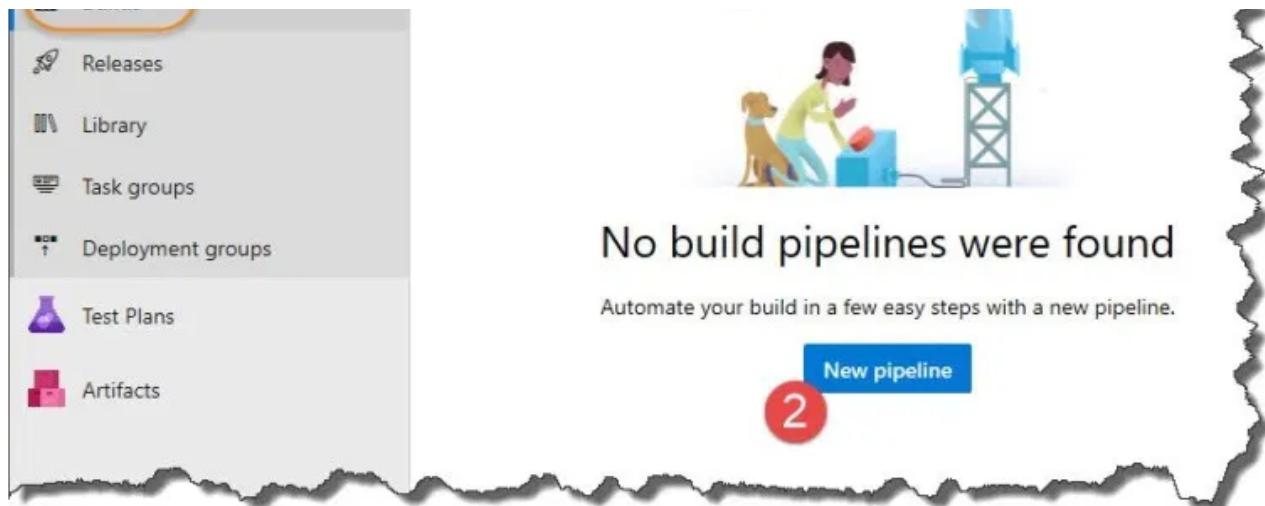
Dotnet Playbook



Azure DevOps has many features, but we'll just be using the "Pipelines" for now...

Select Pipelines, then Builds and then New pipeline:

Dotnet Playbook



The first thing that it asks us is: “Where is your code?”

Well, where do you think? Yeah – that’s right – in Github!

New pipeline

Where is your code?

- Azure Repos**
Free private Git repositories, pull requests, and code search
- GitHub**
Home to the world's largest community of developers
- GitHub Enterprise**
The self-hosted version of GitHub

Use the visual designer to create a pipeline for Bitbucket Cloud, Subversion, TFVC, generic Git, or without YAML.

Dotnet Playbook



IMPORTANT: If this is the 1st time you're doing this, you'll need to give Azure DevOps permission to view your Github account- this is relatively painless and straightforward...

Once you've given Azure DevOps permission to connect to Github, you'll be presented with all your repositories:

The screenshot shows the 'Select a repository' step in the pipeline creation wizard. The top navigation bar includes 'Connect', 'Select' (which is underlined in blue), 'Configure', and 'Create pipeline'. Below the title 'Select a repository', there is a search bar with a filter icon and the placeholder 'Filter by keywords'. To the right of the search bar is a dropdown labeled 'My repositories' with a downward arrow and an 'X' button. A list of repositories is displayed, each with a small profile picture, the repository name, and the last commit date/time. The repositories listed are:

Repository	Last Commit
binarythistle/SimpleAPI	35m ago
binarythistle/S02E01-REST-API-.Net-Core	Thursday
binarythistle/VP-0-REST-Client	22 Jan
binarythistle/VP-1-Working-with-JSON	3 Jun 2018
binarythistle/VP-17-Intro-to-Entity-Framework	9 May 2018
binarythistle/VP-15-Telstra-Messaging-API	13 Feb 2018
binarythistle/VP-10-Post-to-a-REST-API	10 Feb 2018

At the bottom of the list, there is a note: 'Showing the most recently used repositories where you are a collaborator. Or select a specific service connection'.

Pick your repository, (in my case it's "SimpleAPI"), once you click it, Azure DevOps will go off and analyse it to suggest some common pipeline templates, you'll see something like:

Dotnet Playbook



Configure your pipeline



ASP.NET Core recommended

Build and test ASP.NET Core projects targeting .NET Core.



ASP.NET

Build and test ASP.NET projects.



ASP.NET Core (.NET Framework)

Build and test ASP.NET Core projects targeting the full .NET Framework.



Universal Windows Platform

Build a Universal Windows Platform project using Visual Studio.



Xamarin.Android

Build a Xamarin.Android project.



Xamarin.iOS

Build a Xamarin.iOS project.



.NET Desktop

Build and run tests for .NET Desktop or Windows classic desktop solutions.



Starter pipeline

Start with a minimal pipeline that you can customize to build and deploy your code.



Existing Azure Pipelines YAML file

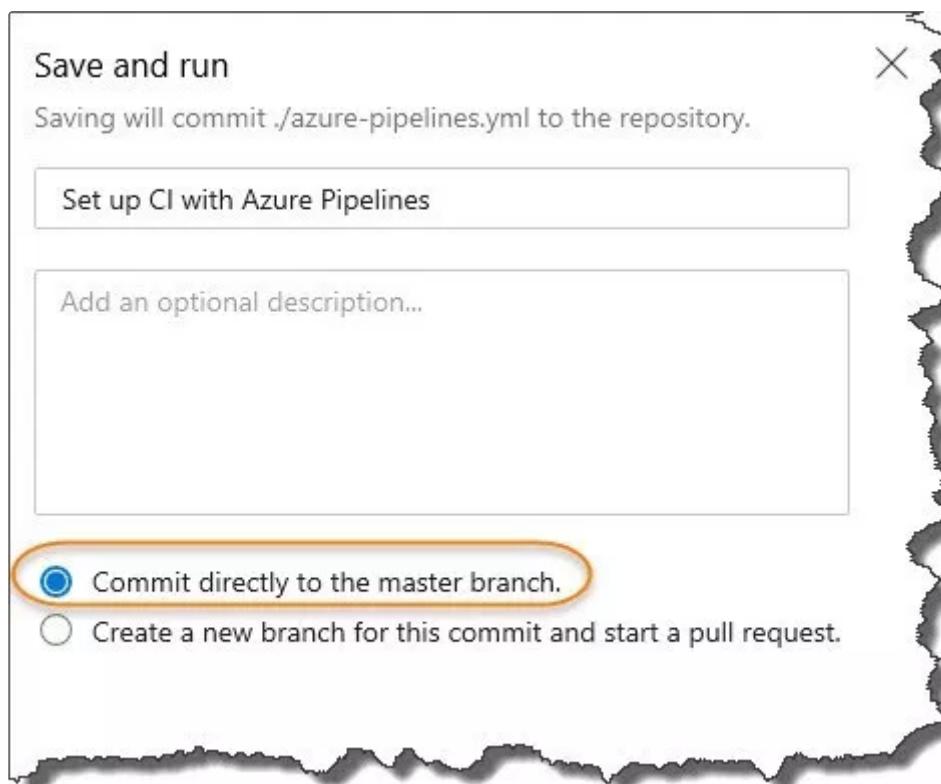
Select an Azure Pipelines YAML file in any branch of the repository.

In this case go with the recommended pipeline template: ASP.NET Core, click it and you'll be presented with your pipeline yaml file:

Dotnet Playbook

```
1 # ASP .NET Core
2 # Build and test ASP.NET Core projects targeting .NET Core.
3 # Add steps that run tests, create a NuGet package, deploy, and more:
4 # https://docs.microsoft.com/azure/devops/pipelines/languages/dotnet-core
5
6 trigger:
7 - master
8
9 pool:
10  vmImage: 'Ubuntu-16.04'
11
12 variables:
13 buildConfiguration: 'Release'
14
15 steps:
16 - script: dotnet build --configuration $(buildConfiguration)
17  displayName: 'dotnet build $(buildConfiguration)'
```

We'll go through this in detail later, suffice to say it's essentially a configurable script for what you want to happen in your build pipeline. Click Save & Run



This is asking you where you want to store the azure-pipelines.yml file, (last step), in this case we want to add it directly to our Github repo, (remember this selection though)

Dotnet Playbook



An “agent” is then assigned to execute the pipeline, you’ll see various screens, such as:

The screenshot shows a build pipeline named #20190306.1: Set up CI with Azure Pipelines. It was manually run just now by Les Jackson. The status is "Preparing an agent for the job". A progress bar indicates the status is "Waiting for the request to be queued". Below the progress bar, a message says "Request queued for agent to pick up".

The screenshot shows the same build pipeline. The "Job Job" step has started at 06/03/2019, 21:09:20. The tasks listed are: Initialize Agent (succeeded), Initialize job (succeeded), Get sources (succeeded), and dotnet build Release. The "dotnet build Release" task is currently executing, showing command-line output for generating a script and running the build command.

```
*****  
Starting: dotnet build Release  
*****  
=====  
Task      : Command Line  
Description : Run a command line script using cmd.exe on Windows and bash on macOS and Linux.  
Version   : 2.146.1  
Author    : Microsoft Corporation  
Help      : [More Information](https://go.microsoft.com/fwlink/?LinkID=613735)  
=====  
Generating script.  
Script contents:  
dotnet build --configuration Release  
[command]/bin/bash --noprofile /home/vsts/work/_temp/6d2ec961-63c9-4896-85f4-09528a494548.sh
```

Pipeline in the midst of execution

And finally you should see the completion screen:

Dotnet Playbook



Logs Summary Tests

Job Job Started: 06/03/2019, 21:09:20
Pool: Hosted Ubuntu 1604 · Agent: Hosted Agent ⋮ 1m 51s

Step	Status	Duration
Initialize Agent	succeeded	<1s
Prepare job	succeeded	<1s
Initialize job	succeeded	1s
Get sources	succeeded	7s
dotnet build Release	succeeded	1m 42s
Post-job: Get sources	succeeded	<1s
Finalize Job	succeeded	<1s

A Green Run!

What Just Happened?

Ok to recap:

- We connected Azure DevOps to Github
- We selected a repository
- We said that we wanted the pipeline configuration file (azure-pipelines.yml) to be placed in our repository
- We manually ran the pipeline
- Pipeline ran through the azure-pipelines.yml file and executed the steps

Azure-Pipelines.yml File

Lets pop back over to our Github repository and refresh – you should see the following:

Dotnet Playbook

The screenshot shows the Azure DevOps repository page for 'Dotnet Playbook'. At the top, there are navigation links for Code, Issues (0), Pull requests (0), Projects (0), Wiki, and Insights. Below the header, a message says 'No description, website, or topics provided.' with a 'Manage topics' link. A summary bar shows 2 commits, 1 branch, and 0 releases. Below this, a dropdown menu shows 'Branch: master' and a 'New pull request' button. A 'Create new...' button is also visible. The main content area lists files: 'src/SimpleAPI' (Initial Commit), 'test/SimpleAPI.Tests' (Initial Commit), '.gitignore' (Initial Commit), 'SimpleAPI.sln' (Initial Commit), and 'azure-pipelines.yml' (Set up CI with Azure Pipelines). The 'azure-pipelines.yml' file is highlighted with an orange oval. At the bottom, a blue box encourages adding a README.

You'll see that the `azure-pipelines.yml` file has been added to our repo (this is important later...)

I thought we wanted to automate?

One of the benefits of a CI/CD pipeline is the automation opportunities it affords, so why did we manually execute the pipeline?

Great Question!

We are asked to execute when we created the pipeline that is true, but we can also set up "triggers", meaning we can configure the pipeline to execute when it receives a

Dotnet Playbook

≡

In your Azure DevOps project click on “Builds” under the Pipelines section, then click the “Edit” button at the top right of the screen, as shown below:

After doing that you should be returned to the azure-pipelines.yml file, (we will return here to edit it later).

```

binarythistle.SimpleAPI
binarythistle/SimpleAPI master /azure-pipelines.yml

1 # ASP.NET Core
2 # Build and test ASP.NET Core projects targeting .NET Core.
3 # Add steps that run tests, create a NuGet package, deploy, and more:
4 # https://docs.microsoft.com/azure/devops/pipelines/languages/dotnet-core
5
6 trigger:
7   - master
8
9 pool:
10  - vmImage: 'Ubuntu-16.04'
11
12 variables:
13   buildConfiguration: 'Release'
14
15 steps:
16   - script: dotnet build --configuration $(buildConfiguration)
17   - displayName: 'dotnet build $(buildConfiguration)'
```

Click the ellipsis, and select “Pipeline Settings”

Dotnet Playbook



The screenshot shows the 'Triggers' tab in the Azure DevOps interface. At the top, there are tabs for 'YAML', 'Variables', 'Triggers', and 'History'. The 'Triggers' tab is selected, indicated by a blue underline. Below the tabs, there's a list of triggers:

- Continuous integration**: Shows a GitHub icon and the repository name 'binarythistle/SimpleAPI'. The status is 'Enabled'. There is a red arrow pointing from the 'History' tab at the top left towards this entry.
- Pull request validation**: Shows a GitHub icon and the repository name 'binarythistle/SimpleAPI'. The status is 'Enabled'.
- Scheduled**: Shows '+ Add'.
- Build completion**: Shows '+ Add'.

On the right side of the screen, there is a summary section for the repository 'binarythistle/SimpleAPI'. It includes a link to the repository, a checkbox for 'Override the YAML continuous integration from here', and some other summary details.

In essence, every time we commit to the repository, (e.g. from our local workstation), a build will be triggered! Let's test that theory...

Triggering a Build

Back at our workstation, and back in VSCode, (or whatever environment you've chosen to use), open the Startup.cs file in our SimpleAPI project and remove the following line of code, making sure to save the file:

Dotnet Playbook

The screenshot shows the Visual Studio interface with the file structure of the 'SimpleAPI' project on the left and the 'Startup.cs' code on the right. A red arrow points to the line 'app.UseHttpsRedirection();' which is highlighted with a yellow oval.

```
C ValuesController.cs src\SimpleAPI\Controllers 30
x C Startup.cs src\SimpleAPI 31
SIMPLEAPI 32
.vscode 33
src 34
SimpleAPI 35
bin 36
Controllers 37
ValuesController.cs 38
obj 39
Properties 40
{} appsettings.Development.json 41
{} appsettings.json 42
C Program.cs 43
SimpleAPI.csproj 44
C Startup.cs 45
test 46
.gitignore 47
SimpleAPI.sln 48
49
```

```
// This method gets called by the runtime.
public void Configure(IApplicationBuilder app,
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // The default HSTS value is 30 days.
        app.UseHsts();
    }
}
app.UseHttpsRedirection();
app.UseMvc();
```

Now I'm not necessarily recommending this is a change you should make in production.. I just want to make any code change so you can see how our local git repo responds, and how we can then push the change to our Github repo and trigger a Azure DevOps build.

Ensuring the deletion is made and the page is saved, go to your command line, (ensure you're in the solution root folder), and type:

```
git status
```

Assuming you've not changed any other files, you should see something like this:



Dotnet Playbook

```
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified:   src/SimpleAPI/Startup.cs
```

```
no changes added to commit (use "git add" and/or "git commit -a")  
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> []
```

Git is telling us that we have modified a file, (startup.cs), since the last commit – remember we are tracking changes on this file. Type:

```
git add .  
git status
```

This stages the file for commit:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git add .  
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git status  
On branch master  
Your branch is up to date with 'origin/master'.
```

```
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)  
  
modified:   src/SimpleAPI/Startup.cs
```

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> []
```

Finally we commit the changed file to our local repo with the following command:



Dotnet Playbook

Another git status will reveal that there is nothing left to commit:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git commit -m "Removed https redirection"
[master dd116d7] Removed https redirection
 1 file changed, 1 insertion(+), 1 deletion(-)
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI>
```

We have made a local code change, and committed it to our local git repository, now we need to push it up to Github to trigger the Build Pipeline!

Type the following at the command line:

```
git push origin master
```

What!???

Yeah that's right we get an error:



Dotnet Playbook

```
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI>
```

What does this mean?

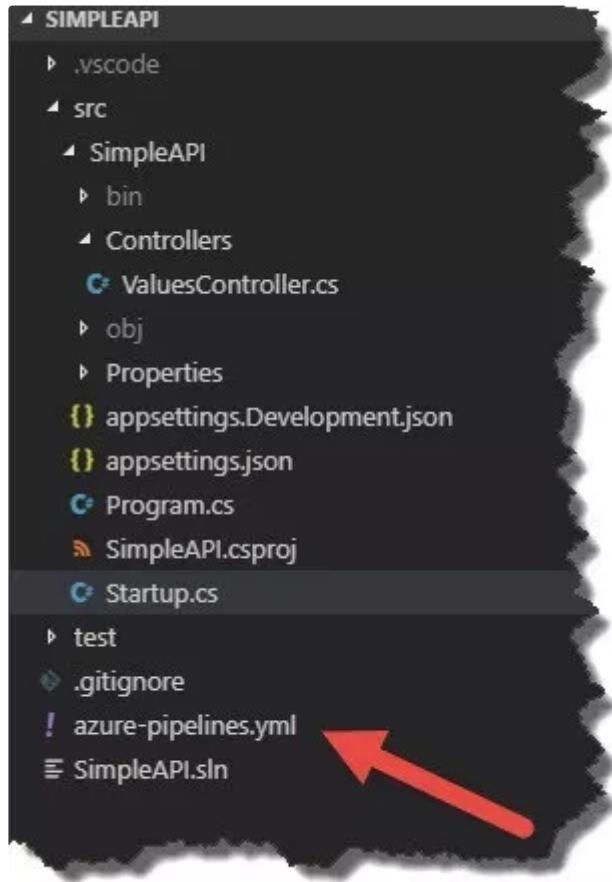
Well remember we added the *azure-pipelines.yml* file to the Github repo? Yes? Well that's the cause, essentially the local repository and the remote Github repository are out of sync. To remedy this, we simply type:

```
git pull
```

This pulls down the changes from the remote Github repository and merges them with our local one:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI> git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/binarythistle/SimpleAPI
  aaba667..a8c2fa9  master      -> origin/master
Merge made by the 'recursive' strategy.
 azure-pipelines.yml | 17 ++++++-----
 1 file changed, 17 insertions(+)
 create mode 100644 azure-pipelines.yml
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI>
```

Dotnet Playbook



Now we have synced our remote repository, we still have to push our local changes up, again issue the following command and this time it should be successful:

```
git push origin master
```

Dotnet Playbook

```
Compressing objects: 100% (6/6), done.  
Writing objects: 100% (7/7), 662 bytes | 662.00 KiB/s, done.  
Total 7 (delta 4), reused 0 (delta 0)  
remote: Resolving deltas: 100% (4/4), completed with 3 local objects.  
To https://github.com/binarythistle/SimpleAPI.git  
 a8c2fa9..a362da7 master -> master  
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI>
```

Quickly jump over to Azure DevOps and click on Pipelines -> Builds, you should see something like this:



Another build process has been kicked off, this time triggered by a remote commit to Github!

All being well this should succeed.

We are getting there, but there is still some work to do on our build pipeline before we move on to deploying – and that is ensuring that our Unit Tests are run – which currently they are not...

Revisit azure-pipelines.yml

Dotnet Playbook



binarythistle.SimpleAPI

binarythistle/SimpleAPI master /azure-pipelines.yml

```
1 # ASP.NET Core
2 # Build and test ASP.NET Core projects targeting .NET Core.
3 # Add steps that run tests, create a NuGet package, deploy, and more:
4 # https://docs.microsoft.com/azure/devops/pipelines/languages/dotnet-core
5
6 trigger:
7   - master
8
9 pool:
10  - vmImage: 'Ubuntu-16.04'
11
12 variables:
13  - buildConfiguration: 'Release'
14
15 steps:
16  - script: dotnet build --configuration $(buildConfiguration)
17  - displayName: 'dotnet build $(buildConfiguration)'
```

1

2

3

4



For brevity I'm not going to cover sections 1-3, (they're quite self explanatory anyway), the "meat" of what we're doing is contained in step 4.

Step 4 is simply executing the "dotnet build" command as you would if you were issuing it at the command line... Nothing more, nothing less.

It does not:

- Run our unit tests
- Package our project for deployment

Dotnet Playbook



Running Unit Tests

We need to edit our .yml file and we can do it either:

- Directly in the browser
- In VSCode

We're going to do the latter, so move back to VSCode and open "azure-pipelines.yml" and add the following directly AFTER "steps:", (noting to be really careful with adding the correct spacing!):

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

Overall your file should look like this:



Dotnet Playbook

```
9  pool:
10 |   vmImage: 'Ubuntu-16.04'
11
12 variables:
13 |   buildConfiguration: 'Release'
14
15 steps:
16 |   - task: DotNetCoreCLI@2
17 |     inputs:
18 |       command: test
19 |       projects: '**/*Tests/*.csproj'
20 |       arguments: '--configuration $(buildConfiguration)'
21
22 |   - script: dotnet build --configuration $(buildConfiguration)
23 |     displayName: 'dotnet build $(buildConfiguration)'
```

Added Section

It basically attempts to do a:

```
dotnet test
```

On our SimpleAPI.Tests project with an additional configuration switch.

Save the file, and issue the following commands, (note “git status” is optional, it’s just useful if starting out with Git):

```
git status
git add .
git status
```

Dotnet Playbook



git push origin master

Pop over to Azure DevOps and refresh your Build Pipeline:

The screenshot shows the build pipeline history for the repository `binarythistle.SimpleAPI`. At the top right are **Edit** and **Queue** buttons. Below them are tabs for **History** and **Analytics***, with **History** being the active tab. The history list is titled "Commit". It contains three entries:

- Updated yaml file to run unit tests** (CI build for binarythistle) - Build # 20190307.2 (indicated by a blue circle with a white gear icon)
- Merge branch 'master' of https://github.com/binarythistle/Sim...** (CI build for binarythistle) - Build # 20190307.1 (indicated by a green circle with a white checkmark icon)
- Set up CI with Azure Pipelines** (Manual build for Les Jackson) - Build # 20190306.1 (indicated by a green circle with a white checkmark icon)

Eventually it should succeed, click the build to have a look at the execution steps:

This screenshot is identical to the one above, showing the build pipeline history for `binarythistle.SimpleAPI`. A red arrow points to the first commit: "Updated yaml file to run unit tests". A red box with the word "click" is overlaid on the build number 20190307.2, indicating where to click to view the execution steps.

Success Steps:

Dotnet Playbook



Logs Summary Tests

Job Job

Started: 07/03/2019, 21:28:32

Pool: Hosted Ubuntu 1604 · Agent: Hosted Ubuntu 1604 2

... 2m 8s

✓ Initialize Agent	· succeeded	5s
✓ Prepare job	· succeeded	<1s
✓ Initialize job	· succeeded	<1s
✓ Get sources	· succeeded	7s
✓ DotNetCoreCLI	· succeeded	1m 55s
✓ dotnet build Release	· succeeded	2s
✓ Post-job: Get sources	· succeeded	<1s
✓ Finalize Job	· succeeded	<1s

[View detailed logs](#)

Select the DotNetCoreCLI step we created in the azure-pipelines.yml file to run tests and drill down, you should see something similar to the following:

Dotnet Playbook

```

2 =====
3 Task      : .NET Core
4 Description : Build, test, package, or publish a dotnet application, or run a custom dotnet command
5 Version   : 2.147.2
6 Author    : Microsoft Corporation
7 Help      : [More Information](https://go.microsoft.com/fwlink/?linkid=832194)
8 =====
9 [command]/usr/bin/dotnet test /home/vsts/work/1/s/test/SimpleAPI.Tests/SimpleAPI.Tests.csproj --log
10 Build started, please wait...
11 Build completed.
12
13 Test run for /home/vsts/work/1/s/test/SimpleAPI.Tests/bin/Release/netcoreapp2.2/SimpleAPI.Tests.dll(
14 Microsoft (R) Test Execution Command Line Tool Version 15.9.0
15 Copyright (c) Microsoft Corporation. All rights reserved.
16
17 Starting test execution, please wait...
18 Results File: /home/vsts/work/_temp/_fv-az77_2019-03-07_10_30_35.trx
19
20 Total tests: 2. Passed: 2. Failed: 0. Skipped: 0.
21 Test Run Successful.
22 Test execution time: 15.2566 Seconds
23 #[section]Async Command Start: Publish test results
24 Publishing test results to test run '1000022'
25 Test results remaining: 2. Test run id: 1000022
26 Published Test Run : https://dev.azure.com/lesjackson/SimpleAPI%20Build%20Pipeline/\_TestManagement/R
27 #[section]Async Command End: Publish test results
28 #[section]Finishing: DotNetCoreCLI
29

```

Clicking on the suggested link in the output will take you to Azure DevOps Test Dashboard – very pretty!

Name ↑	Size	Created Date	Comment
_fv-az77_2019-03-07_10_30_35.trx	4K	9 minutes ago	

Dotnet Playbook



Break Our Unit Test

Ok, so return to VSCode and to our test project: SimpleAPI.Tests, and open the UnitTests.cs file. You'll remember we have 2 unit tests:

1. An empty default test that does nothing and passes
2. A unit test we wrote to test the return value of our API controller

So we want to “break” the 2nd test from passing, so go back to the SimpleAPI project and open the ValuesController.cs file, and edit the return value of our 2nd action method to anything other than “Les Jackson”, (this will cause our test to fail), e.g. :

```
namespace SimpleAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ValuesController : ControllerBase
    {
        // GET api/values
        [HttpGet]
        public ActionResult<IEnumerable<string>> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/values/5
        [HttpGet("{id}")]
        public ActionResult<string> Get(int id)
        {
            return "Max Power";
        }

        // POST api/values
        [HttpPost]
        public void Post([FromBody] string value)
    }
}
```

The name I always wish I had...

Dotnet Playbook



`dotnet build`

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\src\SimpleAPI> dotnet build
Microsoft (R) Build Engine version 15.9.20+g88f5fadfe for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 52.91 ms for C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\src\SimpleAPI\SimpleAPI.csproj.
SimpleAPI -> C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\src\SimpleAPI\bin\Debug\netcoreapp2.2\SimpleAPI.dll

Build succeeded.
  0 Warning(s)
  0 Error(s)

Time Elapsed 00:00:00.90
PS C:\Users\lesja\OneDrive\Documents\VSCode\SimpleAPI\src\SimpleAPI>
```

Ensure you're in the *project* directory.

The **build** succeeds. I just wanted to make this point. We have changed the return value of the action method, but there is nothing wrong syntactically with the code – it's fine.

Now...

In the command line, change into the SimpleAPI.Tests project and type:

`dotnet test`

You should see the test failing:

Dotnet Playbook



```
Assert.Equal() Failure
  (pos 0)
Expected: Les Jackson
Actual:   Max Power
  (pos 0)
Stack Trace:
  at SimpleAPI.Tests.UnitTest1.GetResult() in C:\Users\jackson\OneDrive\Documents\VSCode\Simpl...
```

So the *behavior* of our method is not as we expected, but the code is ok, (i.e. the build succeeded)..

Now ordinarily, having just caused our unit test suite to fail locally, you would not then commit the changes and push them to Github! That is exactly what we are going to do just to prove the point that the tests will fail in the Azure DevOps build pipeline too.

Note: In this instance *we know* that we have broken our tests locally, but there may be circumstances where the developer may be unaware that they have done so and commit their code, again this just highlights the value in a CI/CI build pipeline.

Commit Our Code and Break the Pipeline

At the command line, make sure that you're in the *main solution directory*, and type:

```
git status
```

You should see that our ValuesController.cs file has changed.

Add the file for pre-commit then commit the change to our local git repo:

Dotnet Playbook



```
git commit -m "Changed the return type of our api/values/n method"
```



We now want to push those changes to our remote Github repository:

```
git push origin master
```

Jump over to Azure DevOps, click on Pipelines-> Builds, you should see the pipeline building...

Commit	Build #
Changed the return type of our api/values/n method CI build for binarythistle	20190309.1
Updated yml file to run unit tests CI build for binarythistle	20190307.2
Merge branch 'master' of https://github.com/binarythistle/SimpleAPI CI build for binarythistle	20190307.1
Set up CI with Azure Pipelines Manual build for Les Jackson	20190306.1

Our doomed build...

Of course if you wait long enough, the build will inevitably fail:

Dotnet Playbook



Now clicking on the failed build you can drill down as to why it failed, for brevity I won't show that here, but I'm sure you are aware why this has happened.

Testing – The Great Catch All?

Now this shows us the power of unit testing, (as well as the other forms of automated testing that we mentioned above), in that it will cause the build pipeline to fail and buggy software won't be released or even worse deployed to production! It also means we can take steps to remediate the failure – huzzah!

So conversely does this mean that if all tests pass that you won't have failed code in production? No it doesn't for the simple reason that your tests are only as good as, well your tests!

We can fix this broken test and it will pass, (and in turn all tests will pass), but our overall test coverage is poor – for example we're not testing our other API Action Result methods.

So the point that I'm making, (maybe rather depressingly), is that even if all your tests pass, the confidence you have in your code will only be as good as your test coverage.

Revert Our Change

Dotnet Playbook



multiple files in your project, however as our change was so minimal, it's easier just to change the Action Method in our API Project back to "Les Jackson", (or whatever value you're unit test is using).

So:

- Make the change
- Save the ValuesController.cs file
- Test the build (*dotnet build* in the SimpleAPI project)
- Run the unit tests (*dotnet test* in the SimpleAPI.Tests project)
- Add the ValuesController.cs for pre-commit (*git add .* in the main solution directory)
- Commit the change (*git commit -m "Reverted return value of our api/values/n method"*)
- Push the changes to Github (*git push origin master*)

Once the Azure DevOps pipeline has finished – it should be green again:

binarythistle.SimpleAPI

History Analytics

Commit

Build #

Reverted return value of our api/values/n method
CI build for binarythistle

20190309.2

Changed the return type of our api/values/n method
CI build for binarythistle

20190309.1

All is good in the world again...



Dotnet Playbook

So we have to make one more change to our azure-pipelines.yml file – but what change?

This actually caught me out until I actually read the documentation, (and in fact was one of the reasons why I decided to do this tutorial)...

So far our azure-pipelines.yml does the following:

- Builds our project
- Runs the Unit Tests project

What it does not do is produce an artifact that an Azure DevOps *Release Pipeline* can take and deploy...

So the final change we need to make to our azure-pipelines.yml file is to add some steps to package the build (assuming the build and test steps have passed)

Add the Packaging Steps

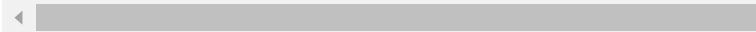
So back to VSCode and open the azure-pipelines.yml file and append the following 2 “tasks” to the end of file:

```
- task: DotNetCoreCLI@2
  displayName: 'dotnet publish --configuration $(buildConfiguration)'
  inputs:
    command: publish
    publishWebProjects: false
    projects: 'src/SimpleAPI/SimpleAPI.csproj'
```



Dotnet Playbook

```
- task: PublishBuildArtifacts@1  
  displayName: 'publish artifacts'
```



So your file should look like:

```
trigger:  
- master  
  
pool:  
| vmImage: 'Ubuntu-16.04'  
  
variables:  
| buildConfiguration: 'Release'  
  
steps:  
  
- task: DotNetCoreCLI@2  
  inputs:  
    command: test  
    projects: '**/*Tests/*.csproj'  
    arguments: '--configuration $(buildConfiguration)'  
  
- script: dotnet build --configuration $(buildConfiguration)  
  displayName: 'dotnet build $(buildConfiguration)'  
  
  Insert Here  
  
- task: DotNetCoreCLI@2  
  displayName: 'dotnet publish --configuration $(buildConfiguration) --output $(Build.ArtifactStagingDirectory)'  
  inputs:  
    command: publish  
    publishWebProjects: false  
    projects: 'src/SimpleAPI/SimpleAPI.csproj'  
    arguments: '--configuration $(BuildConfiguration) --output $(Build.ArtifactStagingDirectory)'  
    zipAfterPublish: true  
  
- task: PublishBuildArtifacts@1  
  displayName: 'publish artifacts'
```

Final changes to yml

The steps are explained in more detail in this [msdn article](#), but in short:

- A dotnet build command is issued for our SimpleAPI project, (not the SimpleAPI.Tests project)
- The output of that is zipped
- Finally zipped output is published

Dotnet Playbook



```
publishWebProjects: false
```

The default is true I guess, for if you don't include that it assumes this is a web project, and the pipeline will fail... urgh!

Commit the Change & Push

So you should be used to this process now:

- Save the file
- git add . (in the main solution directory)
- git commit -m "Updated azure-pipelines.yml to publish build"
- git push origin master

Again the push to Github, will kick off the Azure DevOps Build pipeline, it should succeed:

The screenshot shows the GitHub commit history for the repository `binarythistle.SimpleAPI`. It displays two commits:

- Updated azure-pipelines.yml to publish build** (CI build for binarythistle) - **Build # 20190309.3**
- Reverted return value of our api/values/n method** (CI build for binarythistle) - **Build # 20190309.2**

Both commits have green checkmarks indicating success. Above the commits, there are buttons for **Edit** and **Queue**. Below the commits, the text **Build Published** is visible.

Dotnet Playbook



The screenshot shows a build log for a job named "Job Job". The log details the execution of various tasks:

Task	Status	Duration
Initialize Agent	succeeded	<1s
Prepare job	succeeded	<1s
Initialize job	succeeded	1s
Checkout	succeeded	7s
DotNetCoreCLI	succeeded	1m 50s
dotnet build Release	succeeded	3s
dotnet publish --configuration Release --output \$(Build.ArtifactStagingDirectory)	succeeded	2s
publish artifacts	succeeded	3s
Post-job: Checkout	succeeded	<1s
Finalize Job	succeeded	<1s

A yellow box highlights the "dotnet publish" and "publish artifacts" tasks.

We are almost ready to deploy!

Create Our Azure Resource

Now we move over to Azure.

Azure is a MASSIVE subject area, and given the length this post has gotten, I'm not going to go into details here. Basically we need to create an "API App" on Azure that will host our production REST API, (there are alternative ways we can do this, but for me this is the most appropriate method).

There are 2 ways we can create that API App "resource", (everything in Azure is essentially a resource):

Dotnet Playbook



2. Create it via a script

For simplicity, (and brevity), we'll go with Option 1, don't worry we only have to do this once, it's not something we need to repeat with every deployment, (which would defeat the whole purpose of attempting to automate our deployment).

If you are interested in the scripting approach to create the API App, then Google "Azure Resource Manager Templates", these are also known as ARM Templates.

Create Our API App

Login to Azure, (or if you don't have an account you'll need to create one), and click on "Create a resource":

The screenshot shows the Microsoft Azure portal interface. On the left, there is a sidebar with various navigation options: Home, Dashboard, All services, FAVORITES, All resources, Resource groups, App Services, and Function Apps. The 'Create a resource' button is highlighted with an orange oval. The main content area is titled 'All resources' and shows a list of resources. At the top of this list, there is a 'Subscriptions' section showing 'Pay-as-you-go - Don'. Below this, there is a search bar labeled 'Filter by name...' and a status message '0 items'. The list is currently sorted by 'NAME'.

In the "search box" that appears in the new resource page, start to type "API App", you will be presented with the API App resource type:

Dotnet Playbook



The screenshot shows the Azure Marketplace search interface. A search bar at the top contains the text "API App". Below the search bar, there are three navigation links: "Azure Marketplace", "See all", and "Popular". Under the "Popular" section, two items are listed: "Windows Server 2016 Datacenter" (with a blue Windows logo icon) and "Ubuntu Server 18.04 LTS" (with an orange Ubuntu logo icon). Both items have "Quickstart tutorial" and "Learn more" links next to them.

Select “API App”, then click “Create”:

Dotnet Playbook



Create and deploy RESTful APIs in seconds, as powerful as you need them

Leverage your existing tools to create and deploy RESTful APIs without the hassle of managing infrastructure. Microsoft Azure App Service API Apps offers secure and flexible development, deployment, and scaling options for any sized RESTful API application. Use frameworks and templates to create RESTful APIs in seconds. Choose from source control options like TFS, GitHub, and BitBucket. Use any tool or OS to develop your RESTful API with .NET, Java, PHP, Node.js or Python.

- Fastest way to build for the cloud
- Provision and deploy fast
- Simple access control and authentication
- Secure platform that scales automatically
- Great experience for Visual Studio developers with automatic SDK generation
- Open and flexible for everyone
- Monitor, alert, and auto scale (preview)

Save for later

PUBLISHER

Microsoft

USEFUL LINKS

[Documentation](#)

[Solution Overview](#)

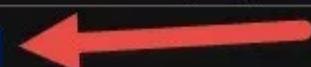
[Pricing Details](#)

Select a software plan

API App

scalable RESTful API with enterprise grade security, simple access control and auto SDK gener...

Create



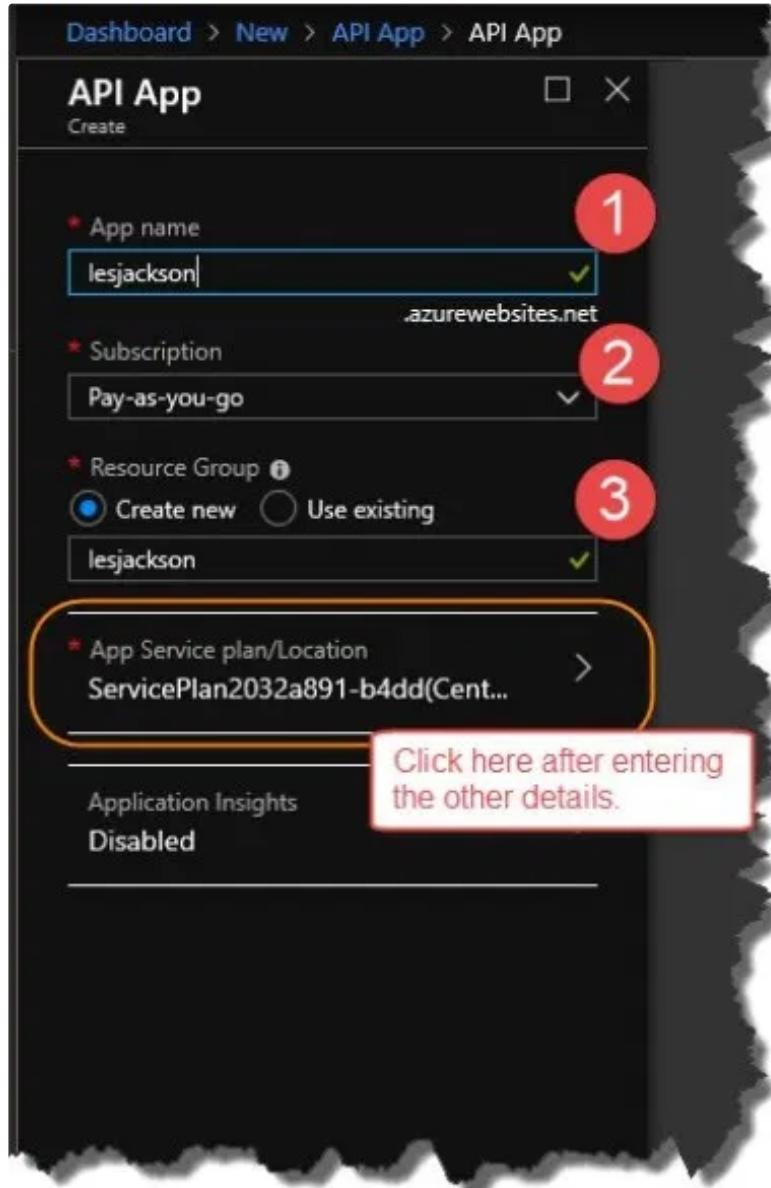
On the Next “page”, enter:

1. A name for your API App



Dotnet Playbook

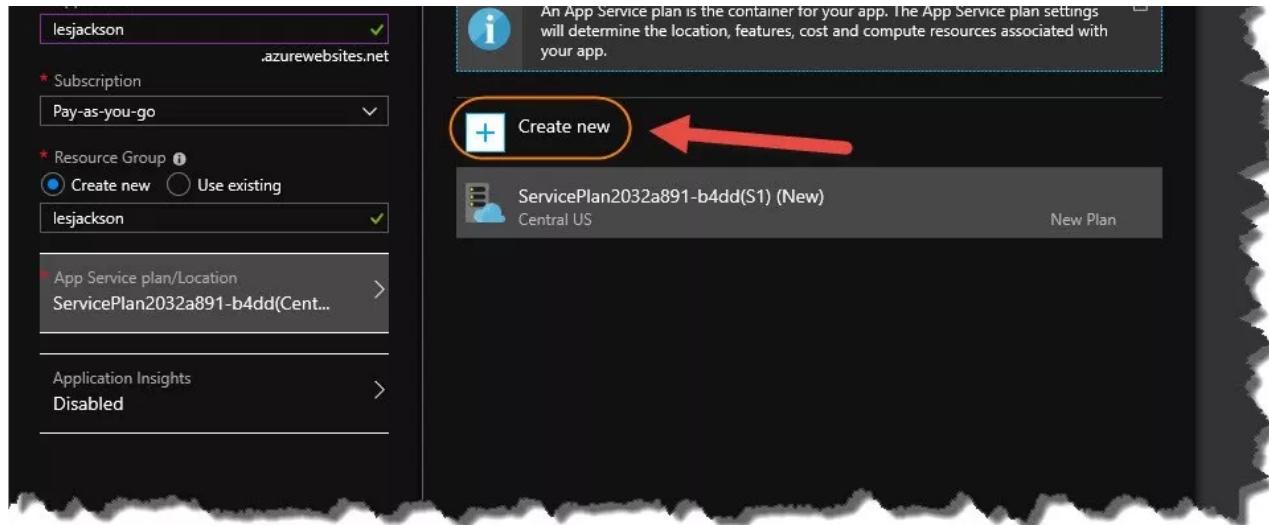
3. A name for your new “Resource Group” – these are just groupings of “resources” – duh!



WAIT! Before you click “Create” click on the “App Service plan/Location”

After clicking on the Service Plan, click on “Create new”:

Dotnet Playbook



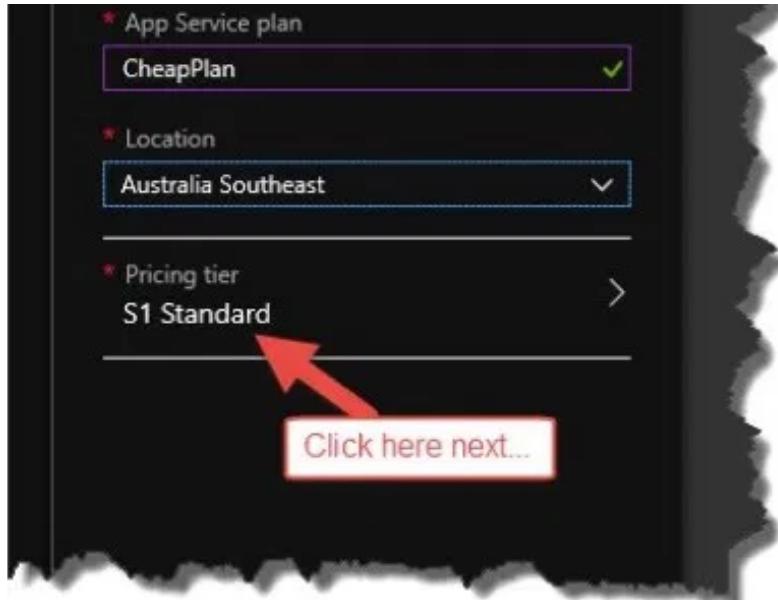
What is A Service Plan?

Basically this is where you select things like:

- Location of your API App (i.e. which Microsoft data center its hosted in)
- Pricing Tier, e.g. do you want dedicated hosting with large CPU, or do you just want a dev/ test box on shared infrastructure?

So on the “New App Service Plan” widget, enter an App Service Plan name, and pick your location, then click on the Pricing Tier...

Dotnet Playbook



After click on the “Pricing Tier”:

Dashboard > New > API App > API App > App Service plan > New App Service Plan >

Dev / Test 1
For less demanding workloads

Production
For most production workloads

Isolated
Advanced networking

Recommended pricing tiers

Tier	Description	ACU	Memory	Compute
F1	Shared infrastructure 1 GB memory 60 minutes/day compute Free	100 total ACU	1.75 GB memory	A-Series compute equivalent
D1	Shared infrastructure 1 GB memory 240 minutes/day compute 18.38 AUD/Month (Estimated)	100 total ACU	1.75 GB memory	A-Series compute equivalent
B1	100 total ACU 1.75 GB memory A-Series compute equivalent 95.98 AUD/Month (Estimated)	100 total ACU	1.75 GB memory	A-Series compute equivalent

2 See additional options

Included hardware
Every instance of your App Service plan will include the following hardware:

- Azure Compute Units (ACU)**
Dedicated compute resources used to run applications deployed to the App Service plan. [Learn more](#)
- Memory**
Memory available to run applications deployed and running in the App Service plan.
- Storage**
1 GB disk storage shared by all apps deployed in the App Service plan.

3 Apply

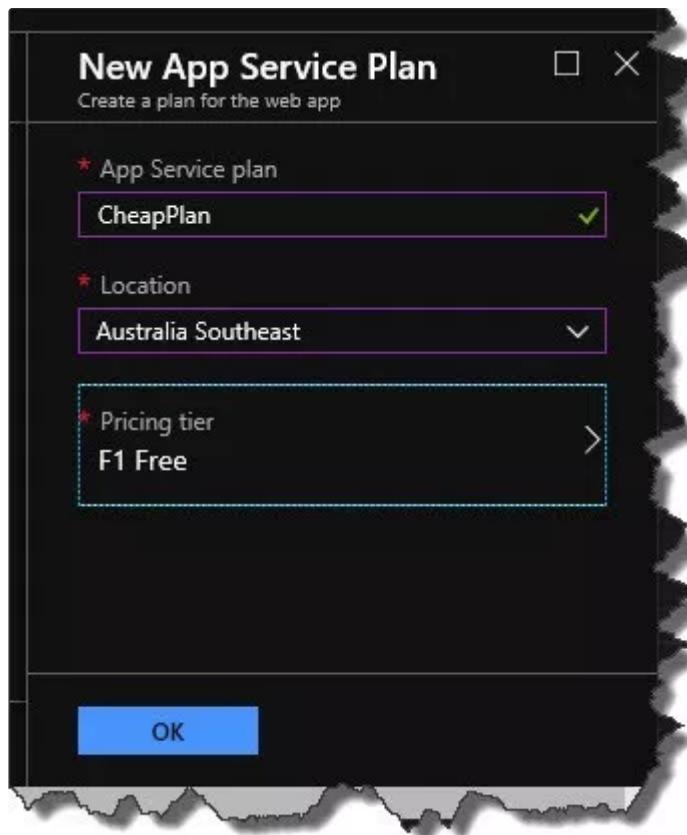
Dotnet Playbook



2. Select the “F1” Option (Shared Infrastructure / 60 minutes compute)
3. Click Apply

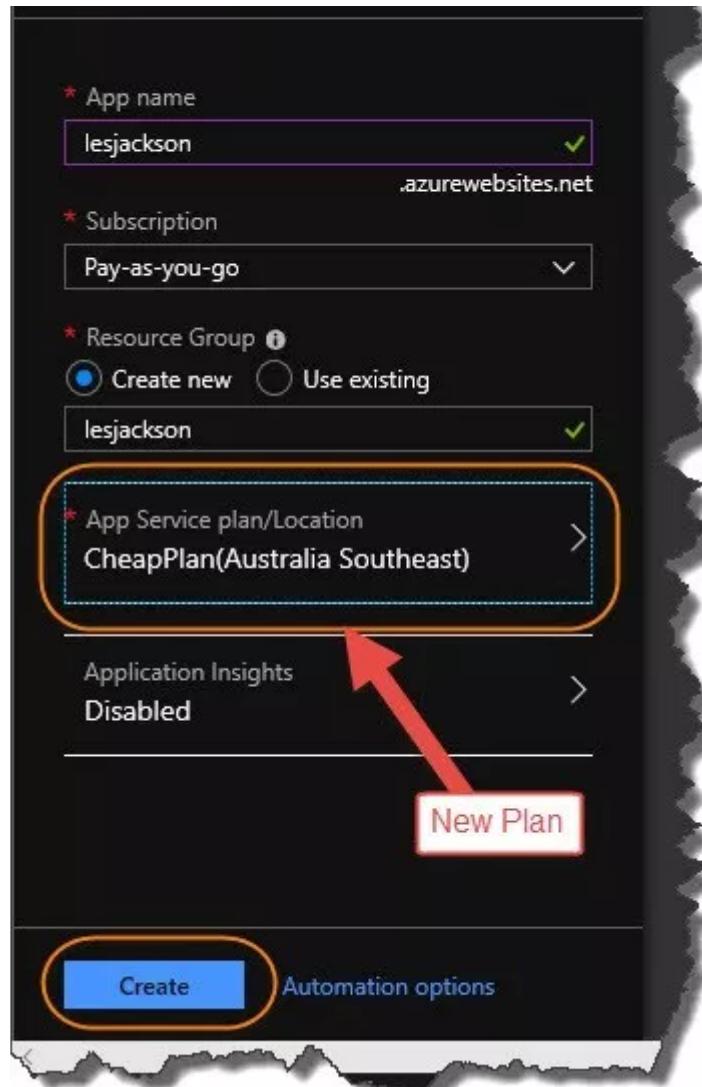
We have selected the cheapest tier with “Free Compute Minutes”, although please be aware that I cannot be held responsible for any charges on your Azure Account! (After I create and test a resource if I don’t need it – I “stop it” or delete it).

Then Click OK...

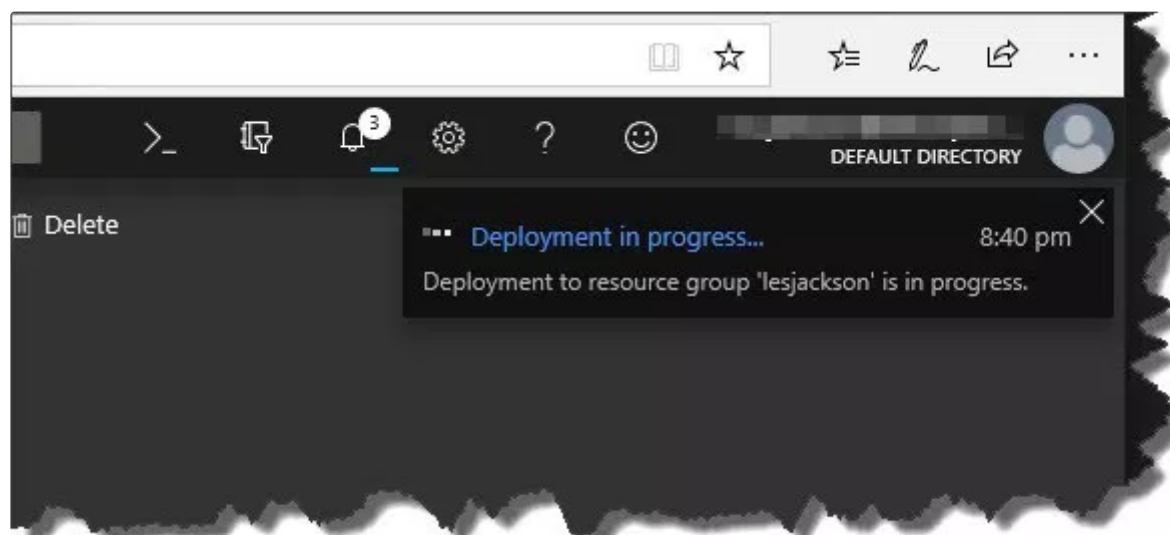


Then click “Create”, (ensure your new App Service Plan is selected):

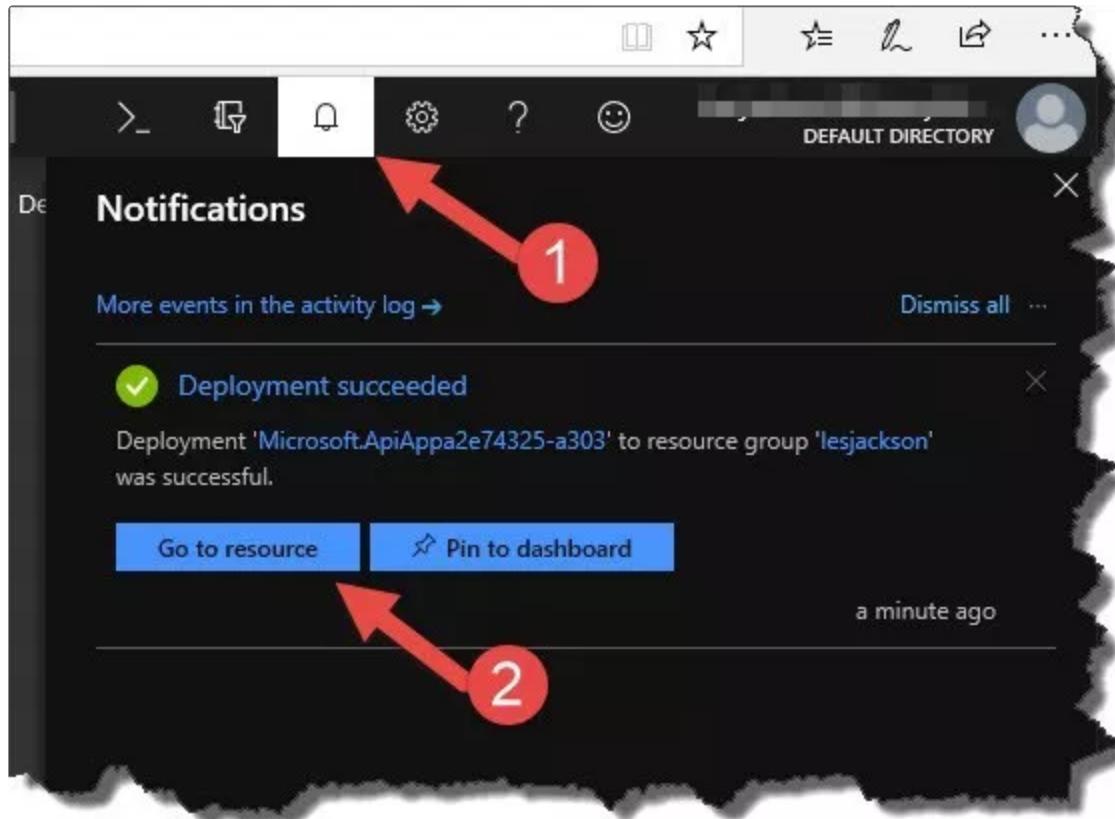
Dotnet Playbook



After clicking Create, Azure will go off and create the resource ready for use:



Dotnet Playbook



Here you can see the resource was successfully created, now click on “Go to resource”:

Dotnet Playbook



The screenshot shows the Azure DevOps Dotnet Playbook interface. At the top, it displays deployment details: Location (Australia Southeast), Subscription (Pay-as-you-go), Tags (Click here to add tags), and connection information for FTP and HTTPS. Below this, there are three cards: 'Diagnose and solve problems' (with a wrench icon), 'Application Insights' (with a purple circle icon), and 'App Service Advisor' (with a blue ribbon icon). The main area contains three monitoring charts: 'Http 5xx' (0 errors from 8 pm to 8:45 pm), 'Data In' (0 bytes from 8 pm to 8:45 pm), and 'Data Out' (0 bytes from 8 pm to 8:45 pm).

This just gives us an overview of the resource we created, and gives us the ability to stop or even delete it. You can even click on the location URL and it will take you to where the API App resides:

The screenshot shows the Azure portal's 'Reset publish profile' dialog. It lists the following settings: URL (https://lesjackson.azurewebsites.net), App Service Plan (CheapPlan (Free: 0 Small)), and other fields like FTP and deployment user which are blurred.

Our “Live” site – although we’ve not deployed our SimpleAPI here yet – we do that next!

Dotnet Playbook



 Microsoft Azure

Your App Service app is up and running

Go to your app's [Quick Start](#) guide in the Azure portal to get started or read our [deployment documentation](#).

We would love to meet you! The App Service team will be present at the following upcoming events:

Create Our Release Pipeline

At last! We create the final piece of the puzzle: The Release Pipeline...

The Release Pipeline takes our build artifact and deploys it, (in this case), to our Azure API App. So back in Azure DevOps, click on Pipelines -> Releases:

Dotnet Playbook



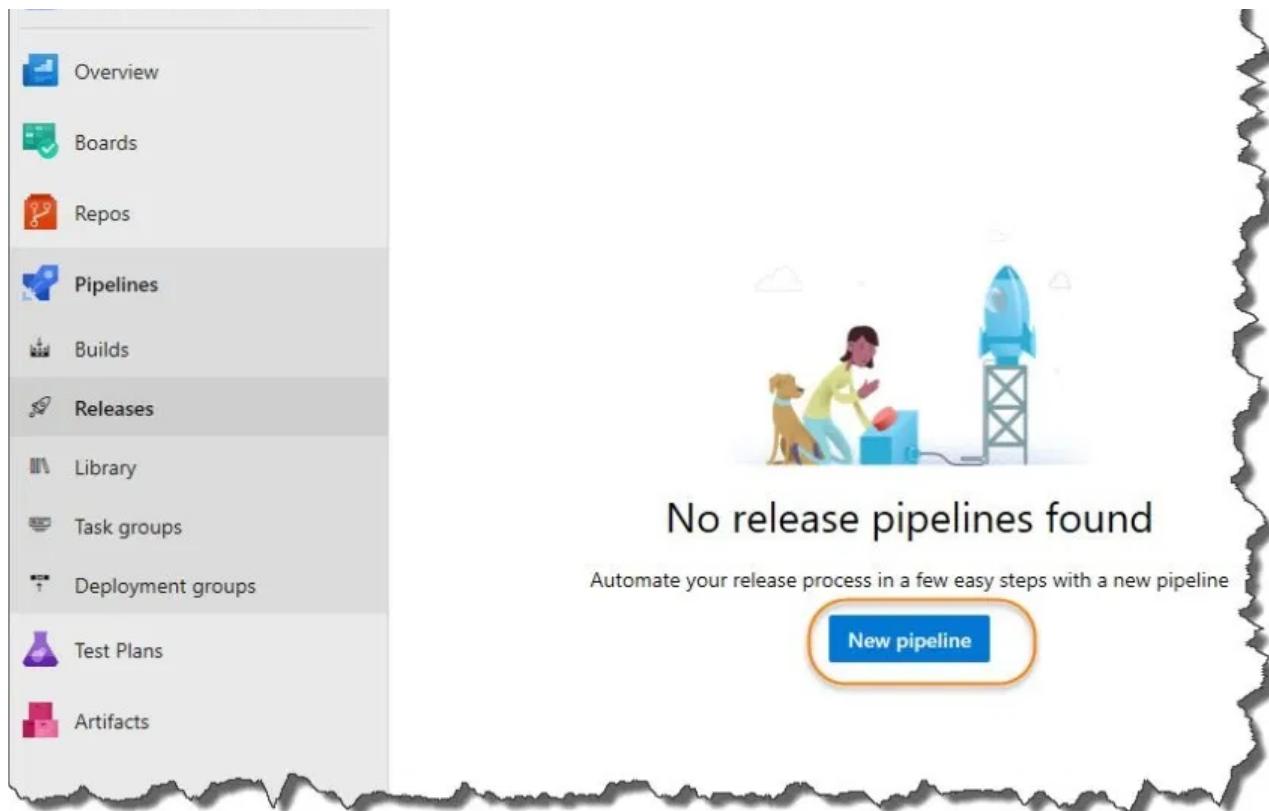
The screenshot shows the Azure DevOps interface for a project named "SimpleAPI Build Pipeline". The left sidebar contains several navigation items:

- Overview
- Boards
- Repos
- Pipelines
- Builds
- Releases** (highlighted with an orange oval)
- Library
- Task groups
- Deployment groups
- Test Plans
- Artifacts

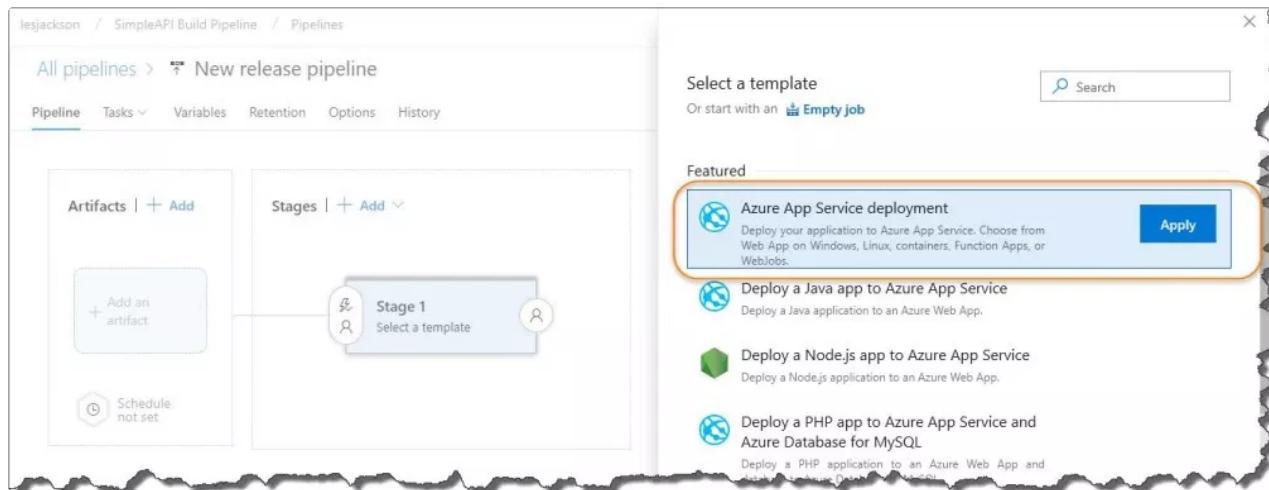
The main content area of the dashboard is currently empty.

Then click on “New pipeline”:

Dotnet Playbook



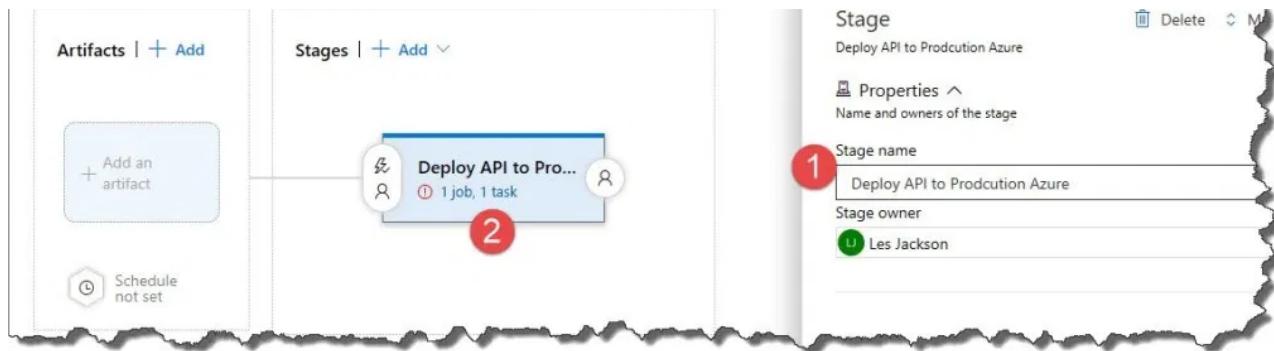
On the next screen, select & “Apply” the “Azure App Service deployment” Template:



In the “Stage” widget:

1. Change the stage name to: “Deploy API to Production Azure”
2. Click on the Job / Task link in the designer

Dotnet Playbook



Here we need to:

1. Select Our Azure Subscription
2. App Type
3. App Service Name

Dotnet Playbook



The screenshot shows the 'Deploy API to Production Azure' stage configuration. Step 1 highlights the 'Azure subscription' dropdown, which is set to 'Pay-as-you-go'. Step 2 highlights the 'App type' dropdown, which is set to 'API App'. Step 3 highlights the 'App service name' dropdown, which is set to 'lesjackson'.

For the Azure Subscription you'll be asked to provide your Azure credentials, then Authorise Azure DevOps to use this account.

The App Type is: API App.

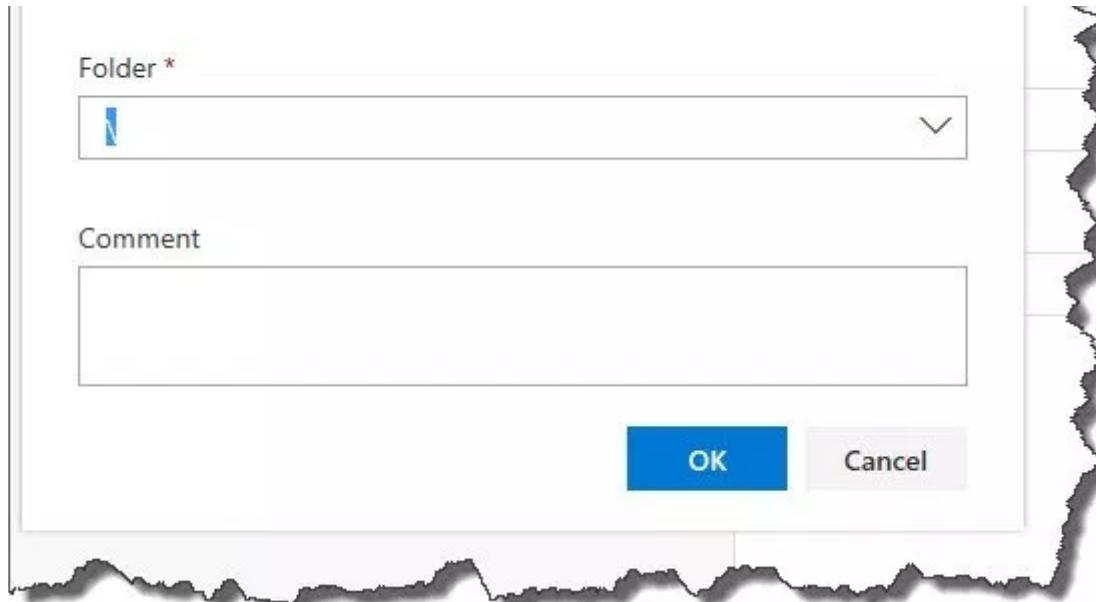
The if you click on the App Service Name drop down, a list of the API Apps that you have on the provided subscription should appear, in this case we have just one.

Finally click "Save"

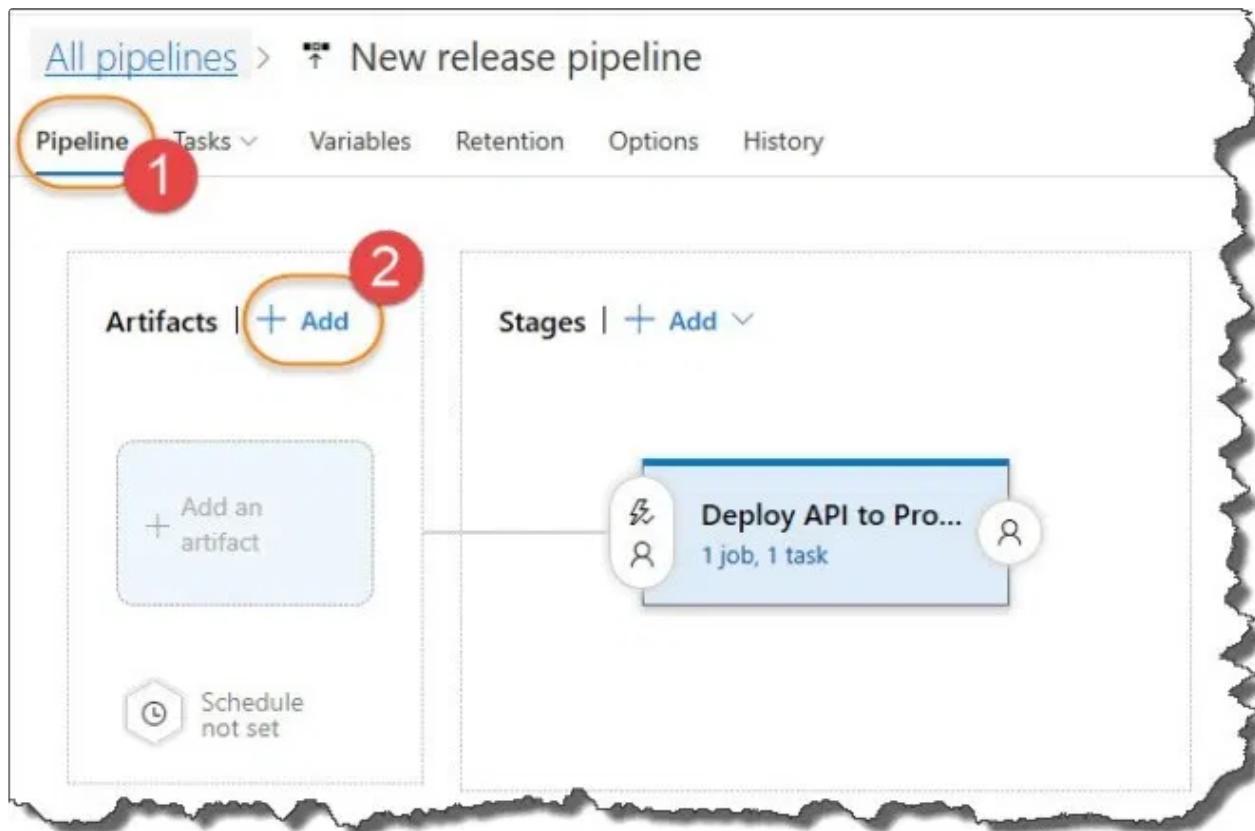
You'll be presented with a Folder pop up, just click ok:

Dotnet Playbook

≡



Click back on the “Pipeline” tab, then on Add (to add an artifact):



Here you will need to provide:

Dotnet Playbook



2. The Source Pipeline (this is our build pipeline we created previously)
3. Default version (select “Latest” from the drop down)

Add an artifact

Source type

Build Azure Repos ... GitHub TFVC

5 more artifact types ▾

Project * (i)

1 SimpleAPI Build Pipeline

Source (build pipeline) * (i)

2 binarythistle.SimpleAPI

Default version * (i)

3 Latest

Source alias * (i)

_binarythistle.SimpleAPI

(i) The artifacts published by each version will be available for deployment in release pipelines. The latest successful build of **binarythistle.SimpleAPI** published the following artifacts: **drop**.

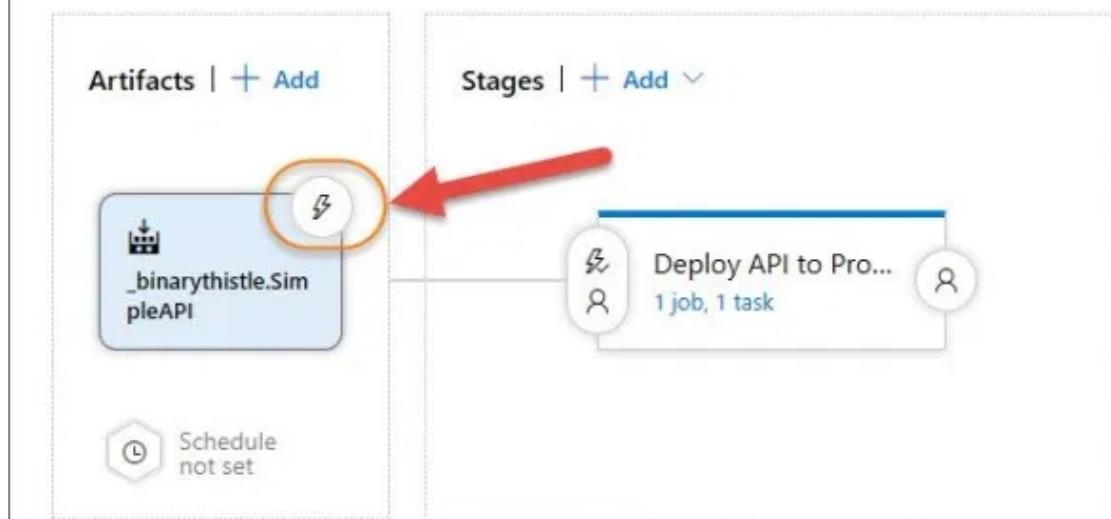
Add 4

Click “Add”, this will detail your Release Pipeline:

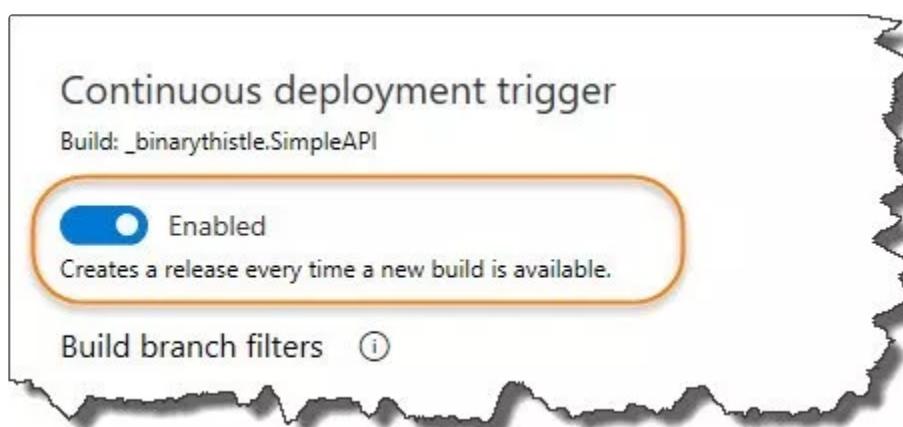
Dotnet Playbook



Pipeline Tasks Variables Retention Options History



Click on the Lightening Bolt on the Artifact node, on the resulting widget ensure that the “Continuous deployment trigger” is enabled:



The click “Save”, (you may be asked to provide a comment – do so if you please):

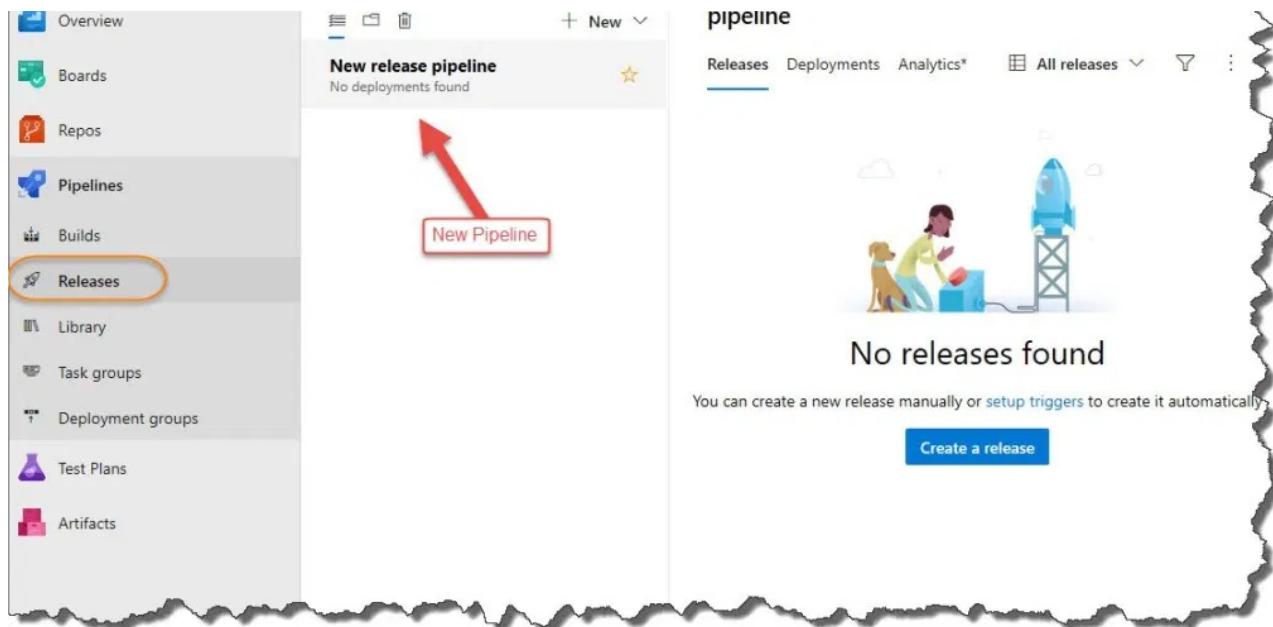
Dotnet Playbook

A screenshot of the Azure DevOps 'Dotnet Playbook' interface. At the top, there's a red arrow pointing to the 'Continuous deployment trigger' section. Below it, the 'Build branch filters' section is shown with a note that no filters have been added. A 'Pull request trigger' section is also visible at the bottom.

This now means that when the *Build Pipeline* completes a build the *Release Pipeline* is triggered too – we then have full Continuous Integration and Continuous Deployment!

Click on Releases, you'll see that we have a new pipeline but no release, this is because the pipeline has not yet been executed:

Dotnet Playbook



Bring It All Together

Finally let's test our end to end pipeline...

In VSCode, go to the ValuesController.cs file in the SimpleAPI project and change the return values of the 1st Action Method from “value1” and “value2” to something else, e.g.:



Dotnet Playbook

```
[ApiController]
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    public ActionResult<IEnumerable<string>> Get()
    {
        return new string[] { "Dot Net", "Play Book" };
    }

    // GET api/values/5
    [HttpGet("{id}")]
    public ActionResult<string> Get(int id)
    {
        return "Les Jackson";
    }
}
```

Note: As there are no associated unit tests with this method there will be no impact to our “test suite”.

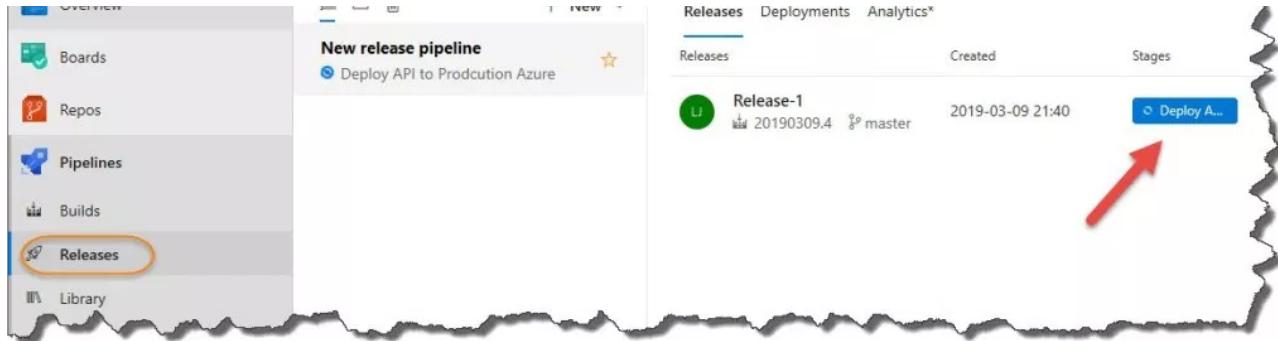
Save the file, commit to git, the push to Github as usual, (I’ll not detail the steps you should know these by now, and I’m tired of typing!).

This should trigger the build pipeline....

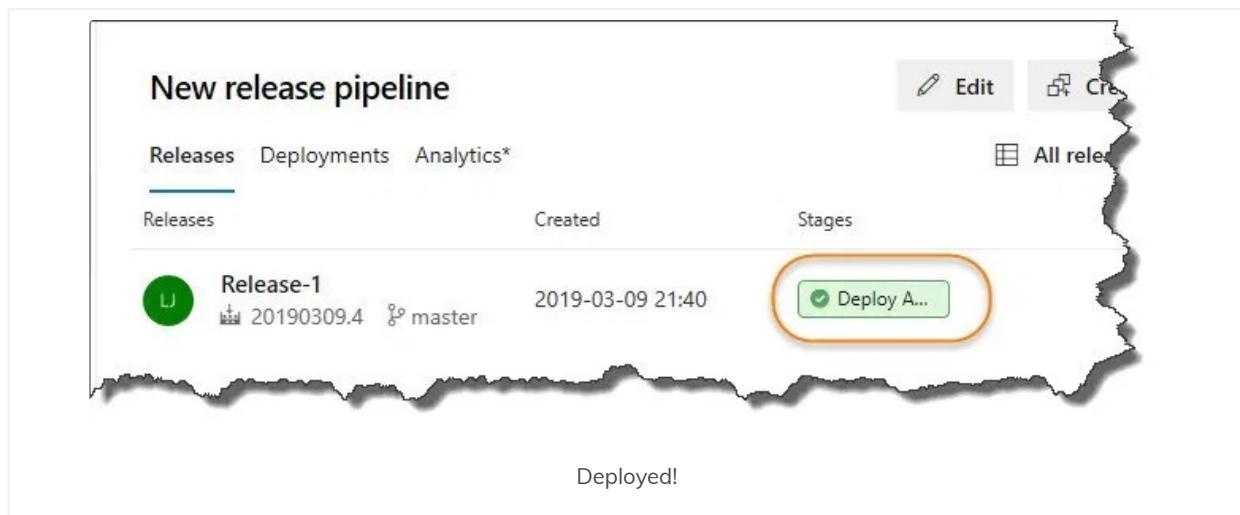
The screenshot shows the Azure DevOps Pipelines interface. On the left, a sidebar lists pipelines: 'Search all pipelines', '+ New', and 'binarythistle.SimpleAPI'. The 'binarythistle.SimpleAPI' pipeline is selected and highlighted. To its right, the pipeline details are shown: 'History' tab is active, 'Analytics' tab is available, 'Commit' section shows a recent commit from 'binarythistle' with the message 'Minor change to API CI build for binarythistle', and 'Build #' shows '20190309.4'. A callout box highlights this commit message. At the bottom of the pipeline card, it says 'Build Pipeline executing...'. The overall interface has a white background with blue and grey UI elements.

When the Build Pipeline finished executing, (successfully), click on “Releases”:

Dotnet Playbook



You'll see the Release Pipeline attempting to deploy to Azure... And eventually it should deploy, (you may need to navigate away from the Release Pipeline and back again):



Finally, go to your web browser and let's see if our API is actually deployed, (this should be the URL provided by Azure that we browsed to above followed by /api/values), e.g.:

<https://lesjackson.azurewebsites.net/api/values>

And, yes – our API has been deployed, (with our changed values!):

Dotnet Playbook



I can't believe this article is this long!

Final Thoughts

Well once again the article turned out much longer than I intended! But to cover the steps in sufficient detail – I guess that was required.

Overall I found using Azure DevOps pretty straightforward, the key to the whole thing, (for me anyway), is understanding the azure-pipelines.yml file and what's possible with it.

The only other addition I'd make, (and I may write a follow up article), is to use Azure Resource Manager, (ARM), templates to set up the API App automatically – but that's for another time...

.NET API APP AZURE AZURE DEVOPS AZURE-PIPELINES.YML BUILD
CI/CD CONTINUOUS CORE DELIVERY DEPLOY DEVOPS DOTNET GIT
GITHUB PIPELINE REST API XUNIT

SHARE: [Twitter](#) [Facebook](#) [Google+](#) [Reddit](#) [Pinterest](#) [LinkedIn](#)

Dotnet Playbook



Les enjoys understanding how things work, proving concepts then telling people about it! He lives and works in Melbourne, Australia but is originally from Glasgow, Scotland. He's just obtained an MCSD accreditation after almost a year, so now has more time for writing this blog, making YouTube videos, as well as enjoying the fantastic beer, wine, coffee and food Melbourne has to offer.



ICON RELATED ARTICLES

REST API

Consuming a REST API from

C#

ENTITY FRAMEWORK

Introduction to Entity
Framework

Dotnet Playbook



REST API

Develop a REST API with .Net

Core

PREVIOUS POST

[Develop a REST API with .Net Core](#)

NEXT POST

[How Much Does Azure Cost a Lone Developer?](#)

Dotnet Playbook

Pragmatic hints, tips, step by step tutorials on how to get the most out of the .Net Framework.

CATEGORIES

› Certification

2

Dotnet Playbook



> Entity Framework	1
> JSON	1
> Microsoft Azure	1
> REST API	4
> WebSockets	1

META

> Log in
> Entries RSS 
> Comments RSS 
> WordPress.org

Copyright © Dotnet Playbook. 2019 • All rights reserved.

