The Complete ASP.NET Core API Tutorial

A Practical Step-by-Step Guide.

Les Jackson

The Complete ASP.NET Core API Tutorial

A practical Step-by-step guide on everything needed to: build, test and deploy an ASP.NET Core API to Azure

[June 2019 Beta Edition]

Les Jackson

Copyright

Warranty

While every effort has been made by the author to ensure that the information in this book is true and accurate at the date of publication, the author cannot accept any legal responsibility for any errors or omissions that may be made. The author makes no warranty, express or implied, with respect to the material contained herein, including code samples which are used at the readers own risk.

For Quynh

CONTENTS

About the Author

Les is originally from Glasgow, (Scotland's largest city), but has lived and worked in Melbourne, Australia since 2009.

Since completing his Computer Science degree in 1998, Les has always worked in IT, primarily in the telecommunications industry, most usually with the incumbent national telecom providers, (you can imagine it's a laugh-riot).

Les holds several industry accreditations, most recently re-acquiring a Microsoft Certified Solution Developer certification, although he still believes there's no substitute for experience and passion – beware of people touting certifications!

Aside from his day job, Les enjoys producing content for his YouTube channel and blog, where he hopes to grow is wonderful audience over the coming years.

In his downtime he likes cycling, trying to grow vegetables, making, (and drinking), beer and traveling with his partner.

Intentionally blank

CHAPTER 1 – INTRODUCTION

WHY I WROTE THIS BOOK

Aside from the fact that everyone is supposed to have "at least 1 book in them", the main reason I wrote this book was for you – the reader. Yes, that's right, I wanted to write a no-nonsense, no-fluff / filler book that would enable the *general reader*[1] to follow along and build, test and deploy an ASP.NET Core API to Azure. I wanted it to be a practical, straightforward text, producing a tangible, valuable outcome for the reader.

Of course, you will be the judge on whether I succeeded, (or not)!

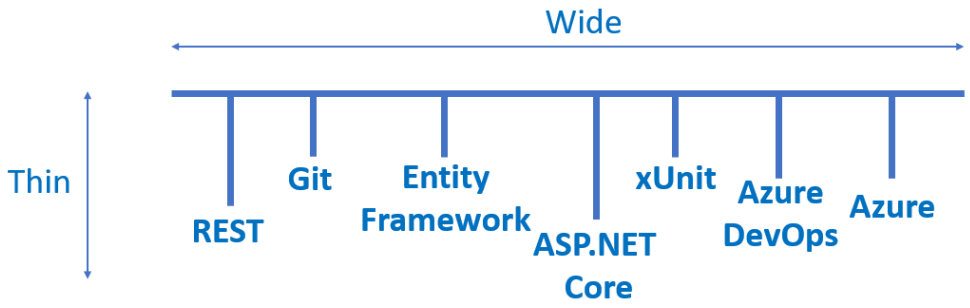
BETA EDITION

This is the Beta Edition of the book. I've taken a Lean Start Up approach, in that I'm releasing the book to the world free of charge and in return I expect you, the reader, to *beta test* if for me and feedback defects and feature improvements.

THE APPROACH OF THIS BOOK

I've taken a "thin and wide" approach with this book, meaning that I wanted to cover a lot of material from the different stages in the development of an API, (wide), without delving into extraneous detail or theory for each, (thin). We will however cover all the areas in enough practical detail, in order that you gain a decent understanding of each – i.e. we won't skip anything important!

I like to think of it like a *tasting menu*. You'll get to try a little bit of everything, so that by the end of the meal you'll have an appreciation of what you'd like to eat more of at some other time, you should also feel suitably satisfied!



Les's Personal Anecdote: The first time I tried, (or even heard of), a tasting menu was in a Las Vegas casino, (I think the MGM Grand), in the early 2000's. In addition to trying the 8 items on the menu, we also went with the "wine pairing" option – which as the name suggests meant you got a different glass of wine with each course, specifically selected to compliment the dish…

I think this is the reason why I can't remember the name of the casino.

[1] Fans of Peep Show I took this term from one of my favourite episodes of season 9: https://www.imdb.com/title/tt2128665/?ref_=ttep_ep4

WHERE CAN YOU GET THE CODE?

While I think you'll get more value by following along throughout the book and typing in the code yourself, (the book has been written so you can follow along step – by step), you may of course prefer to download the code and use that as a reference. Indeed as there may be errata, (heaven forbid!), it's prudent that I provide a repository for you, so you can just head over to GitHub and get the code there:

https://github.com/binarythistle/Complete-ASP.NET-Core-API-Tutorial

CONVENTIONS USED IN THIS BOOK

The following style conventions are use in this book:

Files & Folders

Calibri 10pt Bold & Italic

Class / Object Names

`Courier New 10pt`

Code

`Courier New 9pt / **Courier new 9pt Bold**`

Command Line

```
Courier New 10pt – Highlighted Grey
```

General *additional* information for the reader on top of the main narrative, hint or tip

Warning! Some point of notice so the reader should proceed with caution.

Learning Opportunity: Self-directed learning opportunity. Something the reader can do on their own to facilitate learning & understanding.

Celebration Check Point: Good job, milestone, worth calling out. Allows you to reflect and check learning.

Les's Personal Anecdote: Personal story to add context to a point I'm making. I'll usually try to be humorous here – so be warned. Not required reading to complete working through the book!

Version of the .Net Core Framework

At the time of writing, (May 2019), I'm using version 2.2.101 of the .Net Core Framework.

Contacting the Author

You can contact me through the following channels:

- betabook@dotnetplaybook.com
- https://dotnetplaybook.com/
- https://www.youtube.com/binarythistle

While I'll do my best to reply to you, I'm unlikely to be able to respond to detailed, lengthy technical questions…

Defects & Feature Improvements

Defects, (Errata), and suggestions for improvement should be sent to : betabook@dotnetplaybook.com

- I'll publish the errata on my blog which can be found at: https://dotnetplaybook.com/

Chapter 2 – Setting up your development environment

Chapter Summary

In this chapter we detail the tools and set up you'll require to follow the examples in this book.

When Done, You Will

- Understand what tools you'll need to install
- Have installed those tools & configured your environment ready for development

The 3 Amigos : Windows, Mac & Linux

One of the benefits of the .Net Core Framework, (when compared with the original .Net Framework), is that it's truly cross platform[2], meaning that you can develop and run the same apps on Windows, OSX (Mac) or Linux. For the vast majority of this book the OS that you run on should make little difference in following along with the examples, so the choice of OS is almost irrelevant and of course entirely up to you.

The only callout I'd make at this point is that I use SQL Server[3] as the Database backend for our API example, so non-Windows users may have some additional set up / config in that area.

I list the additional software that you need to follow along with the book below, but have decided not to go into step by step detail about how to install them, for the following reasons:

- The book would become way too bloated if I provided instructions for all 3 OS's (remember: no filler content!)
- My instructions would go out of date quickly and would possibly confuse more than help
- The various vendors typically provide perfectly decent install guides that they maintain and keep up to date, (if not I'll provide them!)

Note: If there's any additional *non-standard* config / set up required I will of course cover that.

Your Ingredients

I'm going to assume you have the absolute basic things like a PC or Mac, a web browser and an internet connection, (if not you'll have to get all of those!), so the software I've listed below is the extra stuff you'll likely need to follow along:

Ingredient	What is it?	Cost	Required For	Platform
VS Code [Get it here]	Cross-platform, fully-featured text editor.	Fee	Writing code! **Note:** This just my personal preference, you	Cross-platform

[2] Yes, there were things like "Mono", but overall I'd say the original .Net Framework was Microsoft Windows centric…

[3] SQL Server is available on Windows, Linux and as a Docker image

			can of course choose an editor that you are more comfortable with.	
.Net Core SDK [Get it here]	.Net Core Runtime and SDK	Free	It's the framework we'll be building our API on. As mentioned in the opening we'll use 2.2 in this book.	Cross-platform
Git [Get it here]	Local Source Code Control	Free	Local source control and pushing our code to GitHub for eventual publishing to Azure	Cross-platform
SQL Server Express [Get it here]	Local SQL Database	Free	We'll use this as our local development / test database.	Windows, Linux or Docker image
SQL Server Management Studio [Get it here]	Management tool suite for SQL Server	Free	Writing and executing SQL queries, setting up DB users etc.	Windows only
Postman [Get it here]	API Testing Tool	Free	[Optional] You can opt to use a web browser to test our API, Postman just gives us more options.	Cross-platform
GitHub.com	Cloud-based git repository used for team collaboration	Free	Used as the code repository component of our Continuous Integration / Continuous Delivery, (CI/CD), pipeline.	N/a – Browser Based
Azure	The Microsoft cloud services offering.	Free[4]	We'll use Azure to host our production API as well as our production SQL Server Database	N/a – Browser Based
Azure DevOps	Cloud-based Build / Test / Deployment platform.	Free	We use Azure DevOps primarily as the vehicle to publish our API to Azure. We will also leverage it's centralised build / test features.	N/a – Browser Based

Install VS Code

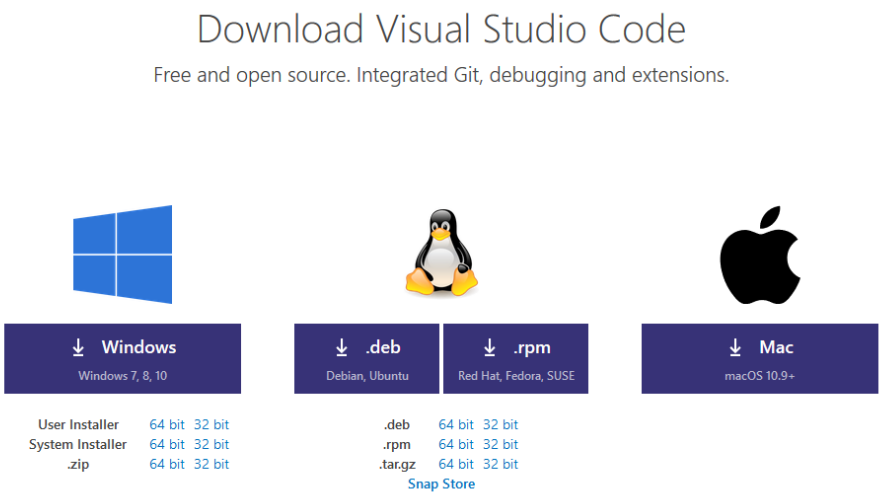
I'm suggesting Visual Studio Code, (referred to now on only as VS Code), as the text editor of choice for following this book as it has some nice features, e.g.: IntelliSense code-completion, syntax highlighting, integrated command / terminal, git integration, debug support etc…

It's also cross-platform, so no matter if you're using Windows, OSX or Linux the experience is pretty much the same, (which is beneficial for someone writing a book. 😉)

[4] At the time of writing new sign ups get $280USD credit, (to use within 1st 30 days), with an additional 12 months of "popular" services free. Other charges may be applicable though, please check the Azure website for the latest offer: https://azure.microsoft.com/

You do of course have other options, most notably Visual Studio[5], which is a fully Integrated Development Environment, (IDE), available on Windows and now OSX, as well as a range of other text editors, e.g. Notepad ++ on Windows or TextMate on OSX etc.

Anyway, to install VS Code, go to: https://code.visualstudio.com/download select your OS and follow the provided instructions for your OS:



Once installed start it up and we'll install a few useful extensions…

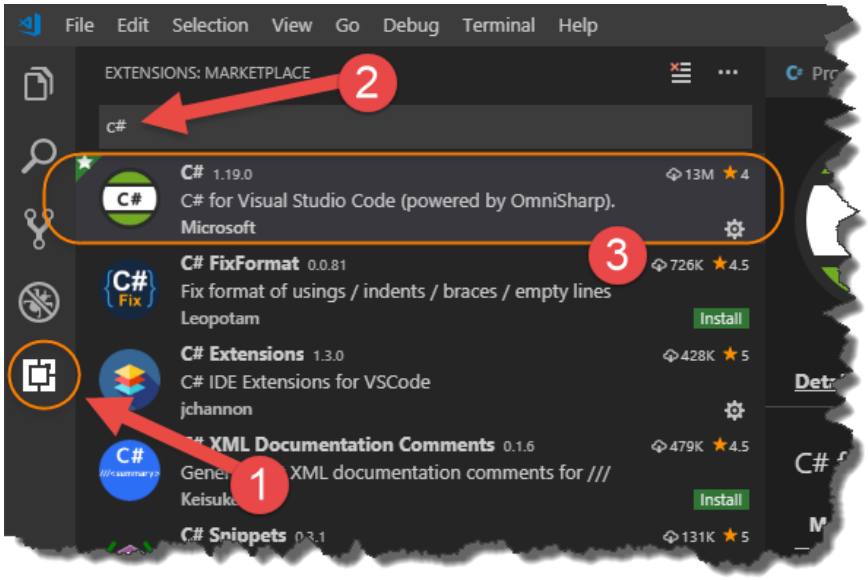
C# for Visual Studio Code

Like a lot of other text editors, VS Code allows you to install Microsoft or 3rd party provided "extensions", (or plugins if you prefer), that extend the functionality of VS Code to meet your specific development requirements. For this project possibly the most important extension is *C# For Visual Studio Code*. It gives us C# support for syntax highlighting and IntelliSense code completion amongst other things, to be honest I'd be quite lost without it.

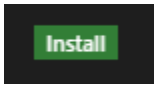
Anyway, to install this extension, (and any others if you wish):

1. Click on the "Extensions" icon in the left-hand tool-bar of VS Code
2. Type all or part of the name of the extension you want, e.g. C#
3. Click the name of the extension you'd like

[5] The "free" version of Visual Studio is called the "Community Edition", just Google it for the download site.



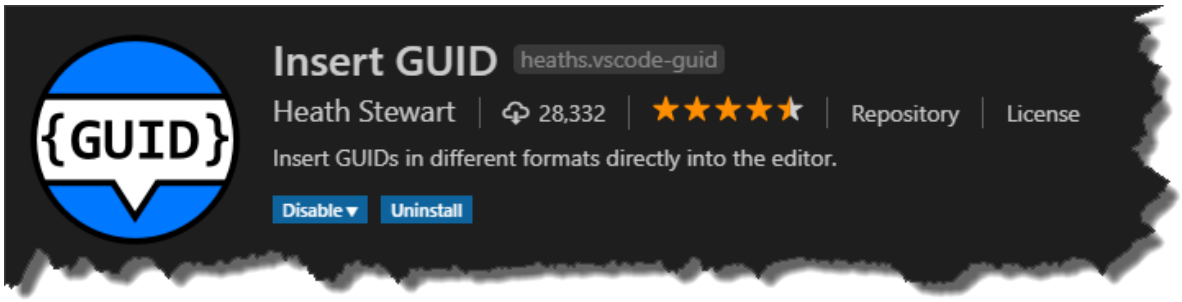
Upon clicking the desired extension you'll get a detail page explaining a bit about the extension, (along with the number of downloads and a review / rating). To install, simply click the "Install" button:



That's it!

INSERT GUID

We'll be using "GUIDs" later in the tutorial, so we may as well install the "Insert GUID" extension too, see extension details below:



Learning Opportunity: Install the "Insert GUID" VS Code extension yourself – it's not hard!

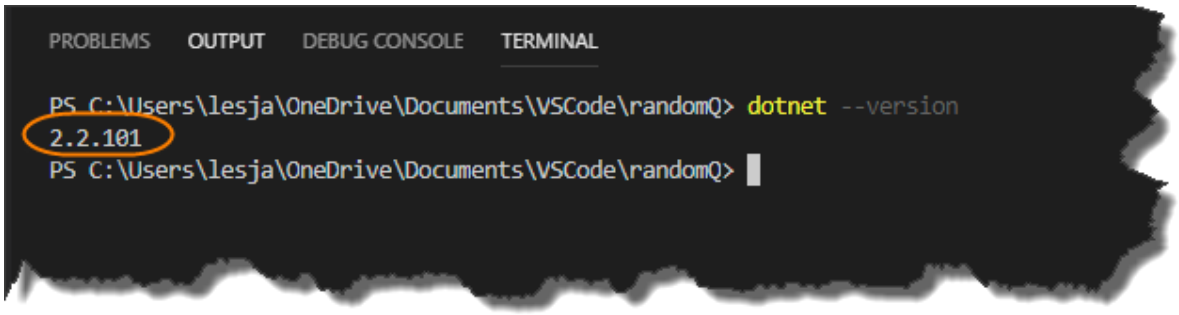
Ok we're done with VS Code set up for now so let's move onto the next install…

INSTALL .NET CORE SDK

You can check to see if you already have .Net Core installed by opening a command prompt, and typing:

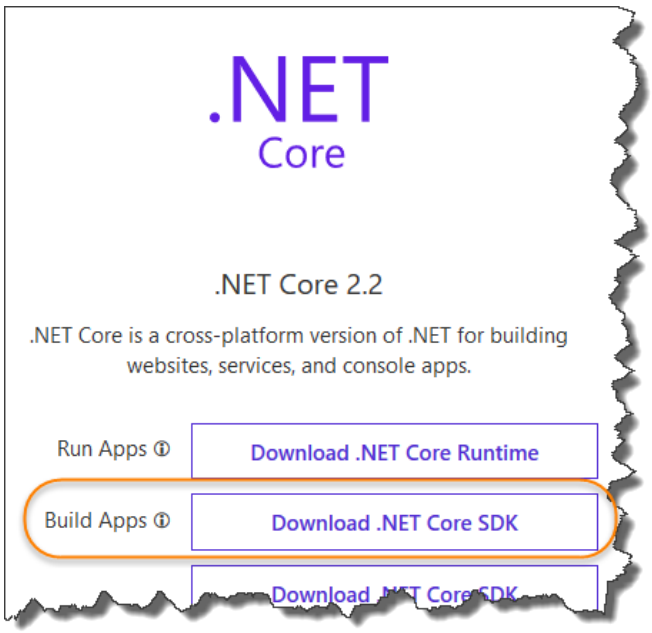
```
dotnet --version
```

If installed, you should see something like this:



Even if it is installed it's probably worth checking to see what the latest version is to make sure that you're not too far behind. At the time of writing this, I was using version 2.2 as shown above.

So if it's not installed, (or you want to update your version), pop over to https://dotnet.microsoft.com/download and select "Download .Net Core SDK", as shown below:



It's important to select the "SDK", (software development toolkit), option as opposed to the "Runtime" option for what I think are quite obvious reasons. (The runtime version is just that, it provides only the necessary resources to ***run*** .NET Core apps. The SDK Version allows us to ***build and run*** apps, it includes everything in the Runtime package.)

As usual follow the respective install procedures for your OS, once completed, you should now be able to run the same `dotnet -version` command as shown above, resulting in the latest version being returned – huzzah!

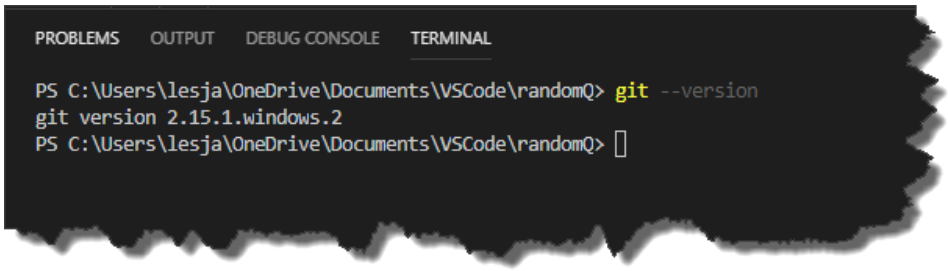
INSTALL GIT

As with .NET Core, you may already have Git installed, (indeed there's probably a much greater chance that it is given its ubiquity).

At a command prompt / terminal type:

```
git --version
```

If already installed you'll see something similar to that shown below:



I'm using the integrated terminal in VS Code running on Windows, depending on your set up it may look slightly different, (you should still see a version number returned if installed though.)

If not installed, or the version you are running is somewhat out of date, go over to https://git-scm.com/downloads and follow the download and install options for your OS.

There are no additional set up instructions for git at this stage… We'll cover setting up and using git repositories later in the book. For now though, we're done!

INSTALL SQL EXPRESS

This is probably the "thorniest" part of this chapter, and possibly the book…

So why SQL Server? Well quite simply – I like it & I use it a lot!

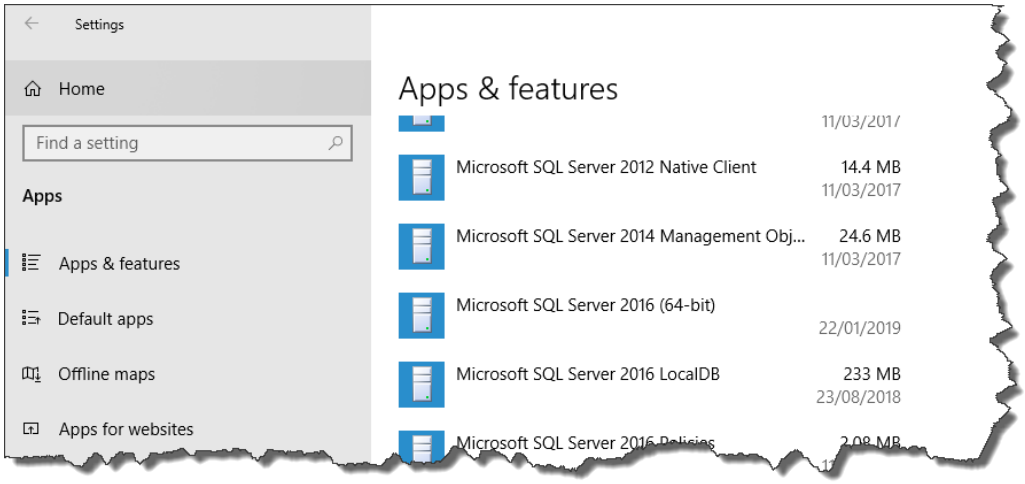
I did deliberate whether to use something like PostgreSQL, which is arguably more platform agnostic than SQL Server, (which traditionally has been Windows-centric), but decided against it. *For now*…

So again, like for the other sections, I'm not going to detail step by step instructions for your chosen OS, (it would take up too much of the book, and probably be subject to a huge amount of errors), but I am going to give a few suggestions on the install for each OS.

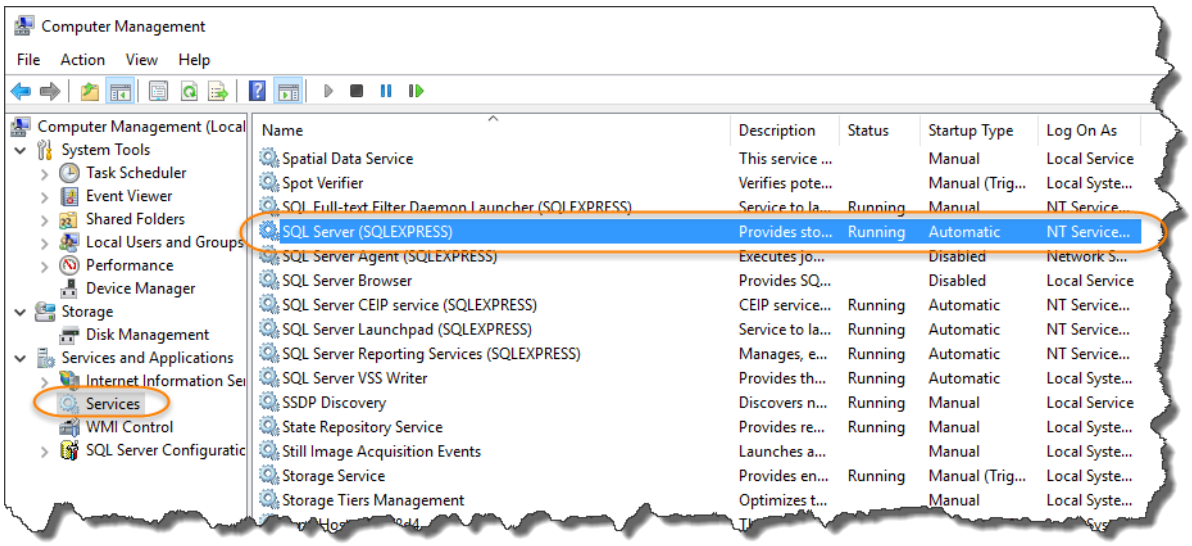
WINDOWS USERS

First off, you may want to check if SQL Server Express is already running. I suspect for most users if it is, you already know it is… To check though, either:

- Go into Settings -> Apps and Features, and locate "Microsoft SQL Server"



- Or, Open "Computer Management", (right click Windows button on Windows 10 and select from menu), then expand "Services" under the "Services and Applications" node. Search down the list of running services for SQL Server (Express)



If it's not there, you'll want to install, (if it is you may want to update – but I'll leave that decision to you).

I'm running SQL Server 2016, however at the time of writing 2017 was available. I choose *not to* upgrade at this stage..

Go to: https://www.microsoft.com/en-us/sql-server/sql-server-editions-express and follow the download and install instructions, when asked choose the following options:

- Choose "Mixed Mode Authentication", (or include SQL Server Authentication). This just means that in addition to integrated Windows user accounts, you can create stand-alone SQL Server accounts
- Choose to install SQL Server Management Studio (Tools) as part of your install. You can do this separately, but why create the extra work for yourself?

Other than that, the install is straightforward. Next, next, next… thank Microsoft!

LINUX USERS

Again unless you've explicitly installed it, SQL Server will not be running, I'd therefore say with an almost 99.99% degree of confidence that it's not running if you're a Linux user…

So to install you have 2 options:

1. Natively install on your OS
2. Install Docker and use the SQL Server Docker image – see section below

To install directly on Linux, again jump over to the Microsoft web site: https://www.microsoft.com/en-au/sql-server/sql-server-downloads and choose Linux Set up.

OSX (Mac) / Docker

If you run a Mac you'll need to use Docker to run the SQL Server image, (this option is available to Linux and Windows users as well actually). Again, Microsoft has done a good job of documenting this, (better than I could anyway), https://www.microsoft.com/en-au/sql-server/sql-server-downloads.

Install SQL Server Management Studio

If you're a **Windows user** and followed my suggestion above this will have already been installed as part of your SQL Server Express install. You can install it standalone though, browse to the Microsoft site here, download and install: https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017

SQL Server Management Studio, (SSMS), is not currently available on either Linux of OSX, so if you want to use its features, you'll need to get access to a Windows environment.

Don't worry – it's not mandatory for you to follow along with the book, there are command line tools we can use instead. I detail these where relevant.

Install Postman

This is totally optional, and up to you if you want to install - but I highly recommend it. I'll be using it at various points throughout the book and given that it's both free and excellent, I don't see why you wouldn't. It's available as both a browser plugin or as a standalone client. For more details on how to install and download go over to: https://www.getpostman.com/downloads/ and take a look.

No further configuration is required at this point.

Wrapping It Up

All the other required components are web-based and only require:

- Web Browser
- Internet connection
- User Account.

I won't insult your intelligence by detailing how to create an account on those services – it's easy. When we come on to the later sections I will cover the set-up and configuration for each where required– so don't worry. For now, all you need is an account on each of the following:

- GitHub
- Azure

- Azure DevOps

All of which, (at least initially!), are free…

CHAPTER 3 – OVERVIEW OF OUR API

CHAPTER SUMMARY

In this, (very short!), chapter I'll take you through the API that you're going to build and the problem it's attempting to solve. We'll also cover the REST API pattern at a high level.

WHEN DONE, YOU WILL

- Understand a bit more about the REST pattern
- Understand what you are going to build throughout the rest of this book
- Understand why you are going to be building this solution
- Have an appreciation of JavaScript Object Notation (JSON)

WHAT IS A REST API?

REST API's will eventually cure world hunger, bring about lasting peace and enable mankind to explore the universe together, forever, in harmony[6]. Or so some people, (usually Salesmen-types), would have you believe. I of course don't believe that and am being somewhat facetious.

REST, (or Representational State Transfer if you prefer), is an architectural style defined by Roy Fielding in 2000, that is used for creating web services. Ok Yes but *what does that* mean? In short REST, or *RESTful* API's, are a lightweight way to transfer textual representations of "resources", e.g. Books, Authors, Cars etc. They are usually, (although don't need to be), built around the HTTP protocol and the standard set of HTTP verbs, e.g. GET, POST, PUT etc.

In recent years REST APIs, have gained favour over other web-services design patterns, e.g. SOAP, as they are considered simpler & quicker to develop, as well as lending themselves to the concept of interoperability more than other approaches. ASP.NET Core APIs have a RESTful approach built-in, which we see as we start to build out our example.

For me personally, actually building out the API is going to help you understand "REST" more fully than if I were to continue writing about it here, so we'll leave the theory there for now…

Learning Opportunity: If you're not comfortable with my description of REST, there are loads of resources already produced on this topic, so if you'd like more info, I'd suggest you do some Googling!

Again though, I think you'll learn more about REST APIs when you come to building them.

OUR API

The API we are going to develop is simple but useful one, (well useful for me anyway!). With my ever-advancing years and worsening state of decrepitude, I wanted to write an API that would store "command line snippets", (e.g. `dotnet new web -n`), as I'm finding it harder and a harder to recall them when needed. In essence it'll become a command line repository that you can query should the need arise.

[6] Credit to the late, great Bill Hicks, whom I'm paraphrasing.

Each “resource” will have the following attributes:

- Howto: Description of what the prompt will do, e.g. add a fire wall exception, run unit tests etc.
- Platform: Application or platform domain, e.g. Ubuntu Linux, Dot Net Core etc.
- Commandline: The actual command line snippet, e.g. dotnet build

Here’s a list of some snippets, (aka “resources”), as an example:

HowTo	Platfrom	Commandline
How to genrate a migration in EF Core	.Net Core EF	dotnet ef migrations add <Name of Migration>
How to update the database (run migration)	.Net Core EF	dotnet ef database update
How to List all active migrations	.Net Core EF	dotnet ef migrations list
Roll back a migration	.Net Core EF	dotnet ed migrations remove
Create a Solutuon File	.Net Core	dotnet new sln --name <Name of Solution>
Add a Project Reference to another project	.Net Core	dotnet add <path to "host" proejct> reference <path to referenced project>
Add Projects to Solution File	.Net Core	dotnet sln <Solution File> add <project1 .csproj file> <projectn .csproj file>

Our API will follow the standard set of Create, Read, Update and Delete, (CRUD), operations common to most REST APIs, as described in the table below:

Verb	URI	Operation	Description
GET	/api/commands	Read	Read all command resources
GET	/api/commands/{Id}	Read	Read a single resource, (by Id)
POST	/api/commands	Create	Create a new resource
PUT	/api/commands/{Id}	Update	Update a single resource, (by Id)
DELETE	/api/commands/{Id}	Delete	Delete a single resource, (by Id)

PAYLOADS

As mentioned above, REST API’s are: “a lightweight way to transfer *textual* representations of resources…”. What do we mean by this?

Well for example, when you make a call to retrieve data from a REST API, the data will be returned to you in some serialised, textual format, e.g.:

- Javascript Object Notation, (JSON)
- Extensible Mark-up Language, (XML)
- Hyper-Text Transfer Language, (HTML)
- Yet Another Mark-up Language, (YAML)

And so on…

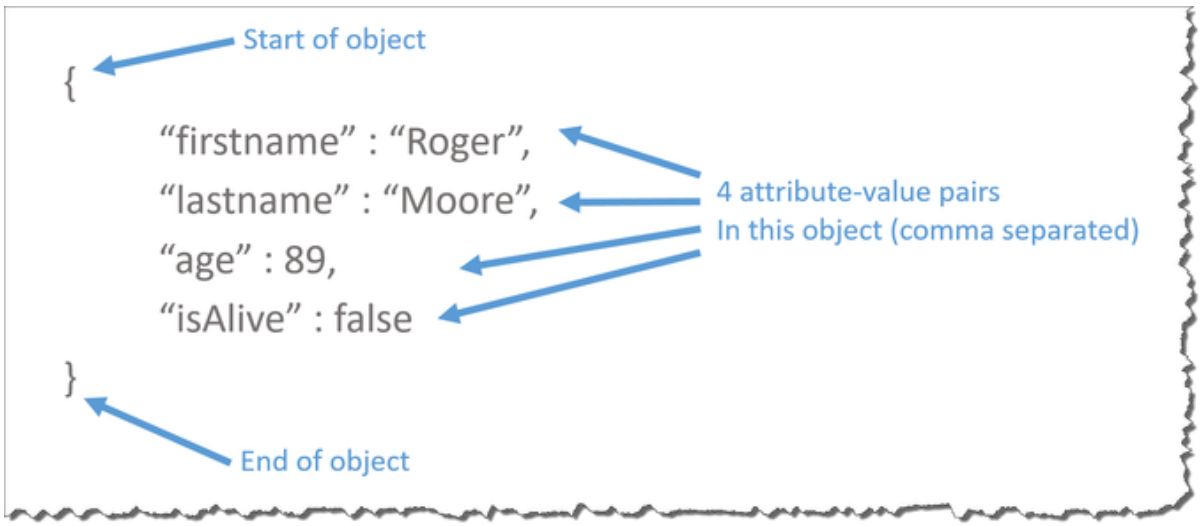
Upon receiving that serialised string payload, you’ll then do something with it, most likely some kind of deserialization operation so you can use the resource or object within the consuming application. With regard REST API’s there is no prescribed payload format, although most usually JSON will be used and returned. We will be using JSON as our payload format in this book, given its lightweight nature and ubiquity in industry.

5 MINUTES ON JSON

What is JSON?

- Stands for: “JavaScript Object Notation”
- Open format used for the transmission of “object” data, (primarily), over the web.
- It consists of attribute-value pairs, (see examples below)
- A JSON Object can contain other “nested” objects

ANATOMY OF A SIMPLE JSON OBJECT



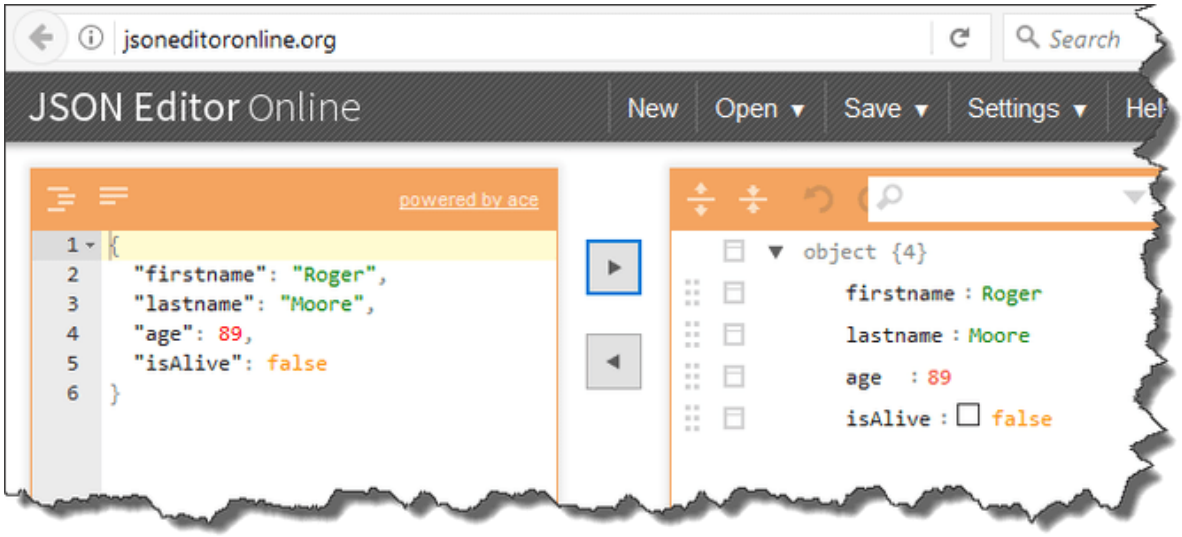
In the above example we have a “Person” object with 4 attributes:

- `firstname`
- `lastname`
- `age`
- `isAlive`

With the following respective values:

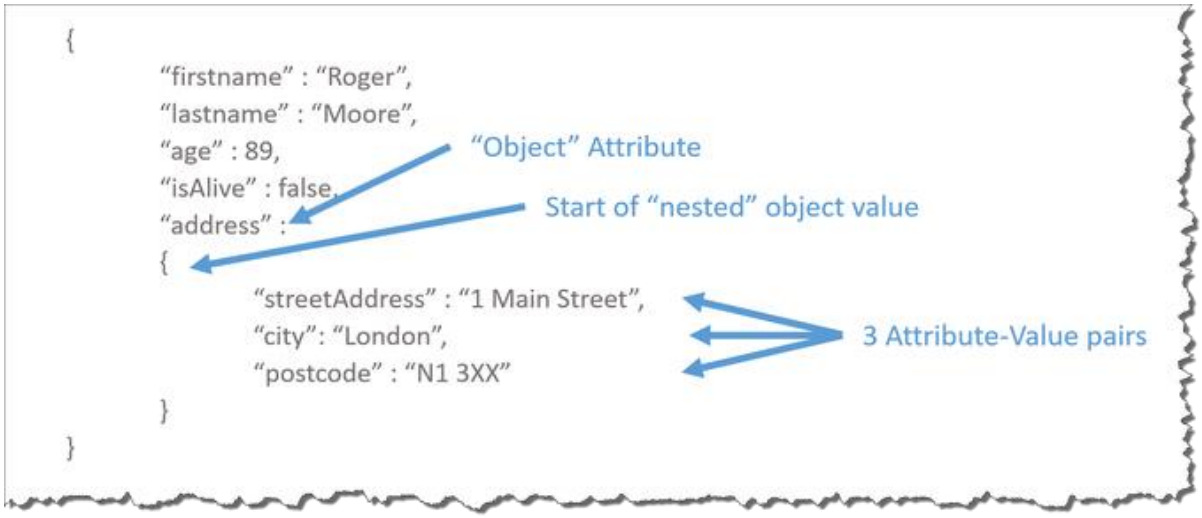
- `Roger` [This is a string data-type and is therefore delineated by double quotes ‘ ” ‘]
- `Moore` [Again this is a string and needs double quotes
- `89` [Number value that does not need quotes]
- `false` [Boolean value, again does not need the double quotes]

Paste this JSON into something like jsoneditoronline.org, and you can interrogate its structure some more:



A, (Slightly), More Complex Example

As mentioned in the overview of JSON, an object can contain "nested" objects, observe our person example above with a nested address object:

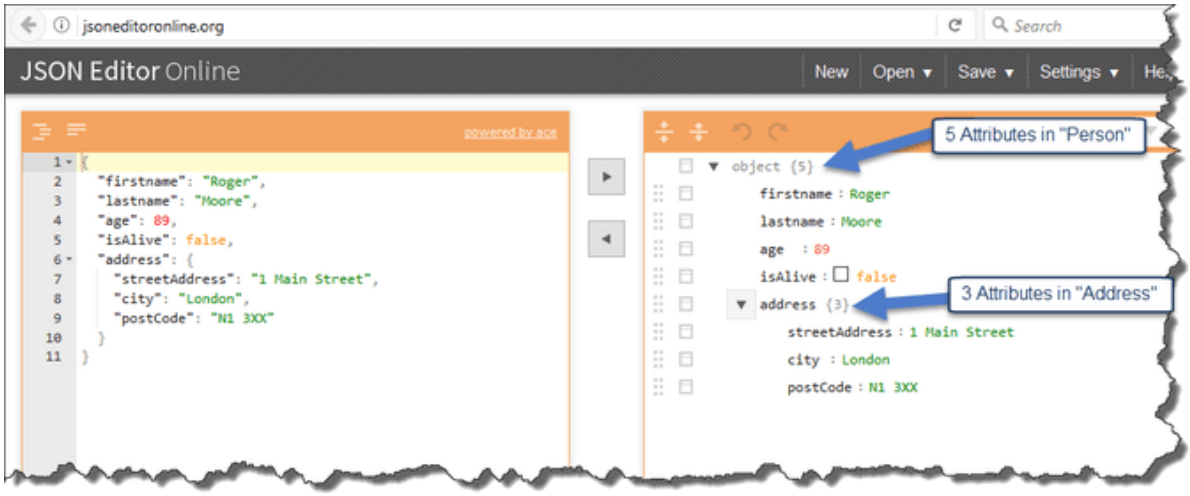


Here we can see that we have a 5th Person object attribute, `address`, which does not have a standard value like the others, but in fact contains another object with 3 attributes:

- `streetAddress`
- `city`
- `postcode`

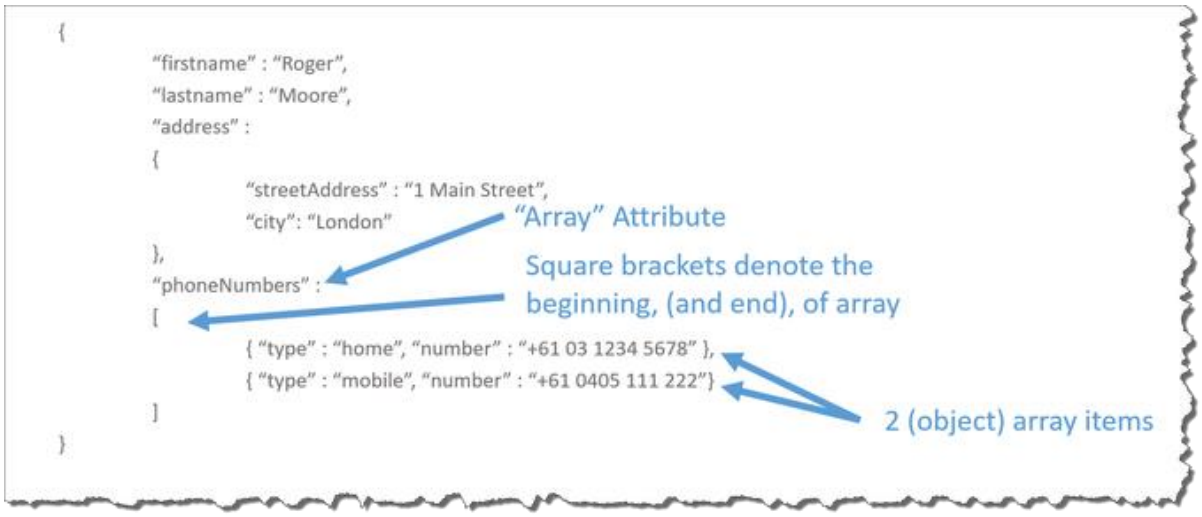
The values of all these attributes contains strings, so no need to labour that point further! This nesting can continue ad-nauseum…

Again, posting this JSON into our online editor yields a slightly more interesting structure:



A Final Example

Onto our last example which this time includes an array of phone number objects:




```
{
        "firstname" : "Roger",
        "lastname" : "Moore",
        "address" :
        {
                "streetAddress" : "1 Main Street",
                "city": "London"
        },
        "phoneNumbers" :
        [
                { "type" : "home", "number" : "+61 03 1234 5678" },
                { "type" : "mobile", "number" : "+61 0405 111 222"}
        ]
}
```

Note: I removed: "`age`" and "`isAlive`" attributes from the person object as well as the "`postcode`" attribute from the address object purely for brevity.

You'll observe that we added an additional attribute to our Person object, "`phonenNumbers`", and unlike the "`address`" attribute it contains an array of other objects as opposed to just a single nested object.

So why did I detail these JSON examples, we'll firstly it was to get you familiar with JSON and some of its core constructs, specifically:

- The start and end of an object: ' { } '
- Attribute-Value pairs
- Nested Objects, (or objects as attribute values)
- Array of Objects

Personally, on my JSON travels these constructs are the main one's you'll come across, and as far as an introduction goes, should give you pretty good coverage of most scenarios - certainly with regard the API we're building, which will both return, and accept, simple JSON objects.

CHAPTER 4 – SCAFFOLD OUR API SOLUTION

CHAPTER SUMMARY

In this chapter we will "scaffold" our 2 projects and place them within a *solution*. We'll also talk about the "bare-bones" contents of a typical ASP.NET Core application and introduce you to 2 key classes: `Program` & `Startup`.

WHEN DONE, YOU WILL

- Have created our main API Project
- Have created our Unit Test Project
- Place both projects within a solution
- Have a solid understanding of the anatomy of an ASP.NET Core project
- Get introduced to the `Program` and `Startup` classes in an ASP.NET Core project

SOLUTION OVERVIEW

Before we start creating projects I just wanted to give you an overview of what we'll end up with at the end of this chapter, (I don't know about you but it helps me if I know the end goal I'm working towards). First off a bit about our "solution hierarchy":

Component	What is it?	Main Config. File	Relationships
Solution	Primary container, holds 1 or more related Projects	.sln	Projects are Children
Project	Self-contained "project" of related functionality	.csproj	Solution is Parent Projects are siblings

A "Solution" is really nothing more than a container for 1 or more related projects, projects in turn contain the code and other resources to do useful stuff. Therefore you would not put code directly into a Solution.

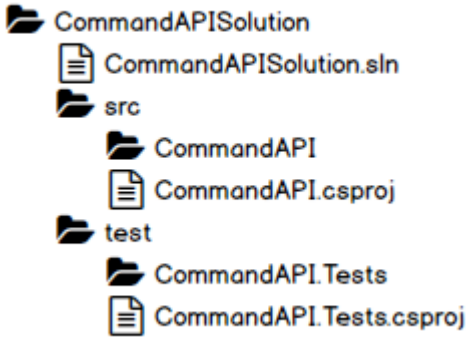
Projects can of course exist without a parent Solution, going further, Projects can reference one and other without the need for a Solution. So why bother with a solution? Great question, it boils down to:

- Personal preference on how you want to "group" related projects
- If you're using Visual Studio, (this always usually creates a solution for you)
- Whether you want to "build" all projects within a solution together

We will use a Solution as we are going to have 2 interrelated Projects:

- Source Code Project (Our API)
- Unit Test Project (Unit Tests for our API)

The overall layout for our solution is detailed below:



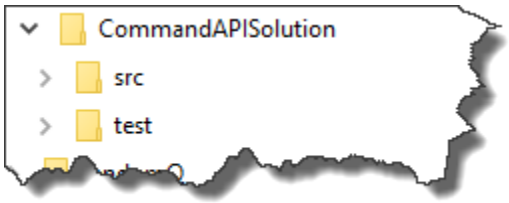
You'll see that we have sub folders within the main solution folder to segregate source code, (***src***), and unit test projects, (***test***). Ok so let's start creating our solution and projects!

Scaffold Our Solution Components

Move to your working directory, (basically where you like to store the solution and projects), and create the following folders:

- Create main "solution" folder called ***CommandAPISolution***
- Create 2 sub directories called in solution folder called ***src*** and ***test***

You should have something like:



- Open a terminal window, (if you haven't already), and navigate to "inside" the ***src*** folder you just created.

.Net Core provides a number of "templates" we can use when creating a new project, selecting a particular template will impact any additional "scaffold" code automatically generated.

To see a list of the templates available, type:

```
dotnet new
```

You should see something like the following:

```
Razor Page                                      page            [C#]
MVC ViewImports                                 viewimports     [C#]
MVC ViewStart                                   viewstart       [C#]
ASP.NET Core Empty                              web             [C#], F#
ASP.NET Core Web App (Model-View-Controller)    mvc             [C#], F#
ASP.NET Core Web App                            webapp          [C#]
ASP.NET Core with Angular                       angular         [C#]
ASP.NET Core with React.js                      react           [C#]
ASP.NET Core with React.js and Redux            reactredux      [C#]
Razor Class Library                             razorclasslib   [C#]
ASP.NET Core Web API                            webapi          [C#], F#
global.json file                                globaljson
NuGet Config                                    nugetconfig
```

You'll notice that there's a template called "webapi" that we could use to generate this project… However I felt that as most of the auto-generated scaffold code is important, that we create this ourselves. Therefore for this tutorial we'll be using the "web" template, which effectively is the simplest, empty, ASP .Net template.

So, to generate our new "API" project, type:

```
dotnet new web -n CommandAPI
```

Where:

- `web` is our template type
- `-n CommandAPI`, names our project and creates our project & folder

You should see something like:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src> dotnet new web -n CommandAPI
The template "ASP.NET Core Empty" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on CommandAPI\CommandAPI.csproj...
  Restoring packages for C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src\CommandAPI\CommandAPI.csproj...
  Generating MSBuild file C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src\CommandAPI\obj\CommandAPI.csproj.nuget.g.props.
  Generating MSBuild file C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src\CommandAPI\obj\CommandAPI.csproj.nuget.g.targets.
  Restore completed in 819.55 ms for C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src\CommandAPI\CommandAPI.csproj.

Restore succeeded.
```

As per our layout above, a folder called ***CommandAPI*** should have been created in ***src***, change into this folder and listing the contents you should see:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src> cd CommandApi
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src\CommandApi> ls


    Directory: C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src\CommandApi


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
da---l        16/05/2019   7:26 PM                obj
da---l        16/05/2019   7:26 PM                Properties
-a---l        16/05/2019   7:26 PM            146 appsettings.Development.json
-a---l        16/05/2019   7:26 PM            105 appsettings.json
-a---l        16/05/2019   7:26 PM            412 CommandAPI.csproj
-a---l        16/05/2019   7:26 PM            632 Program.cs
-a---l        16/05/2019   7:26 PM           1120 Startup.cs
```

If you're not familiar with navigating folders in a terminal or command line it may be worth Googling some basic commands. As I'm using a "PowerShell" terminal, the commands I used above are similar to those you'd find on a Unix / Linux system. If your using a Windows Command Prompt, you'd type: `cd <name of directory>` followed by `dir` . The `dir` command is similar to `ls` here in that it lists the content of the current directory.

Ok we're done scaffolding our *API project*, now we need to repeat for our Unit Test project.

- Navigate into the ***test*** folder[7] contained in the main solution directory ***CommandAPISolution***
- At the command line, type:

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

You should see the following output:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\test> dotnet new xunit -n CommandAPi.Tests
The template "xUnit Test Project" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on CommandAPi.Tests\CommandAPi.Tests.csproj...
  Restoring packages for C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\test\CommandAPi.Tests\CommandAPi.Tests.csproj...
  Generating MSBuild file C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\test\CommandAPi.Tests\obj\CommandAPi.Tests.csproj.nuget.g.props.
  Generating MSBuild file C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\test\CommandAPi.Tests\obj\CommandAPi.Tests.csproj.nuget.g.targets.
  Restore completed in 739.51 ms for C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\test\CommandAPi.Tests\CommandAPi.Tests.csproj.

Restore succeeded.

PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\test>
```

Learning Opportunity: What is xUnit? Remember the command we typed to get a list of all available templates? Try that again to see what the xUnit template is. Can you see any templates that look similar, maybe with a similar name component? Perhaps do some research into what they are too…

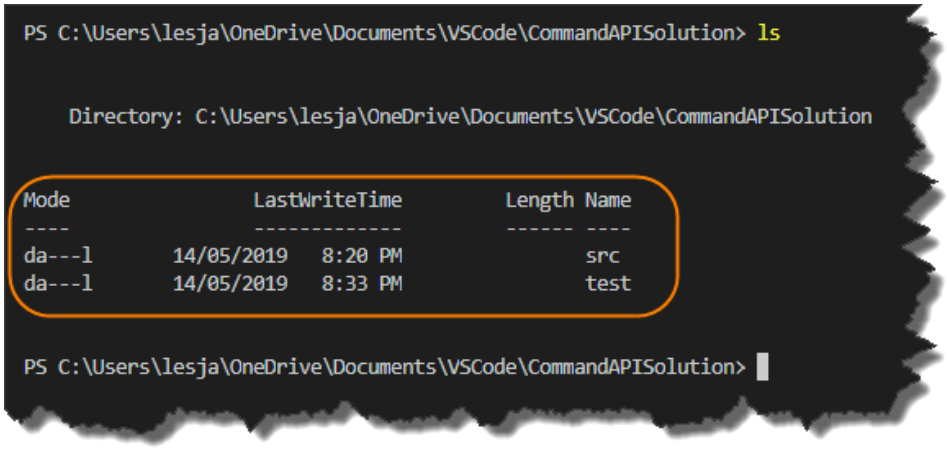
Creating Solution and Project Associations

Ok, so we've created our 2 projects, but now we need to:

- Create a Solution File that links both projects to the overall solution
- Reference Our API Project in our Unit Test Project

Back at our terminal / command line, change back into the main Solution folder, to check you're in the right place perform a directory listing, and you should see something like this:

[7] Hint: `cd ..` moves you up a directory (note there is a space between cd and the two periods, `cd_..`)



You should see the 2 directories: ***src*** & ***test***

Now issue the following command to create our solution, (.src), file:

```
dotnet new sln --name CommandAPISolution
```

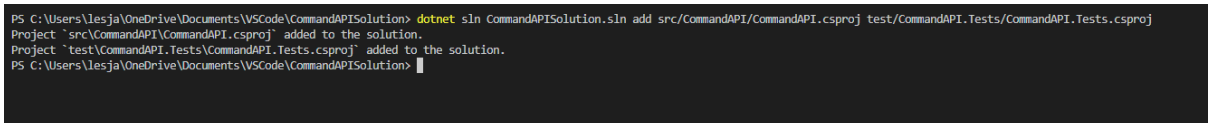
This should create our *empty solution* file, as shown below:

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

```
dotnet sln CommandAPISolution.sln add src/CommandAPI/CommandAPI.csproj
test/CommandAPI.Tests/CommandAPI.Tests.csproj
```

Note: the above command is all one line.

You should see that both projects are added to the solution file:



If you get an error, double check that you have typed the full path correctly. It's quite long so the opportunity to make a mistake is quite great. Believe me – I have spent many a time rectifying typos of this sort!

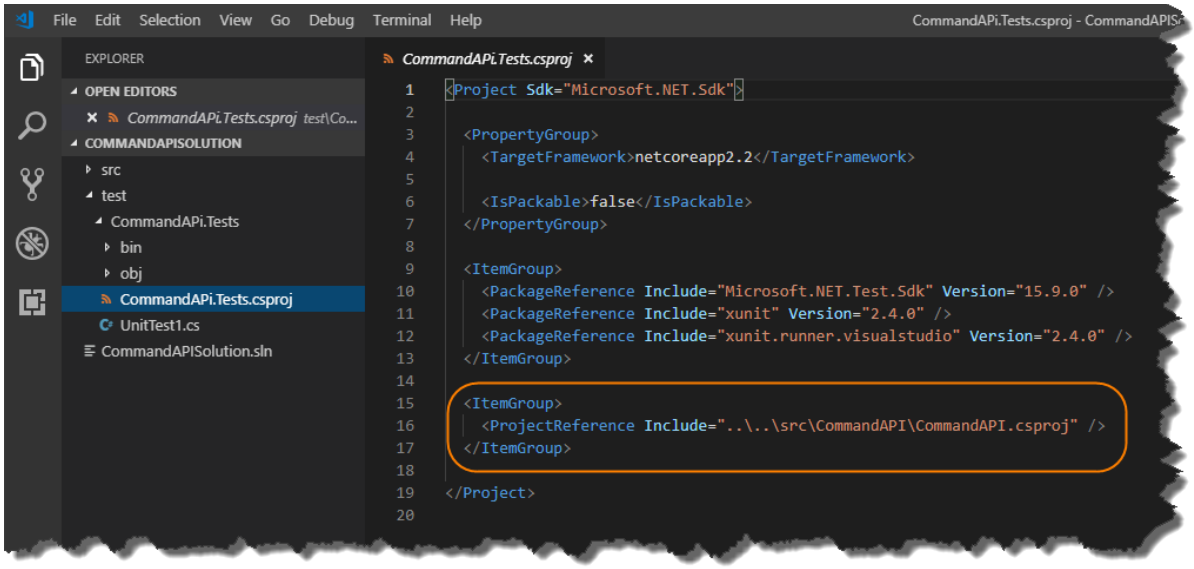
All this really does is tell our solution that it has 2 projects. The projects themselves are unaware of each other. It's kind of like a parent knowing it has 2 children, but the children are *unaware* of each other – we're going to rectify that now. Well for one of the siblings anyway…

We need to place a "reference" to our *CommandAPI* project ***in*** our *CommandAPI.Tests* project, this will enable us to reference the *CommandAPI* project and "test" it from our *CommandAPI.Tests* project. You can either manually edit the *CommandAPI.Tests.csproj* file, or type the following command:

```
dotnet add test/CommandAPI.Tests/CommandAPI.Tests.csproj reference
src/CommandAPI/CommandAPI.csproj
```

You should get something like the following:

Open VS Code, (or whatever editor you chose), and open the ***CommandAPISolution*** folder[8], find the ***CommandAPI.Tests.csproj*** file and open it – you should see a reference, (as well as other things), to the CommandAPI project:



Learning Opportunity: Why do we only place a reference this way? Why don't we place a reference to our unit test project in our API projects .csproj file?

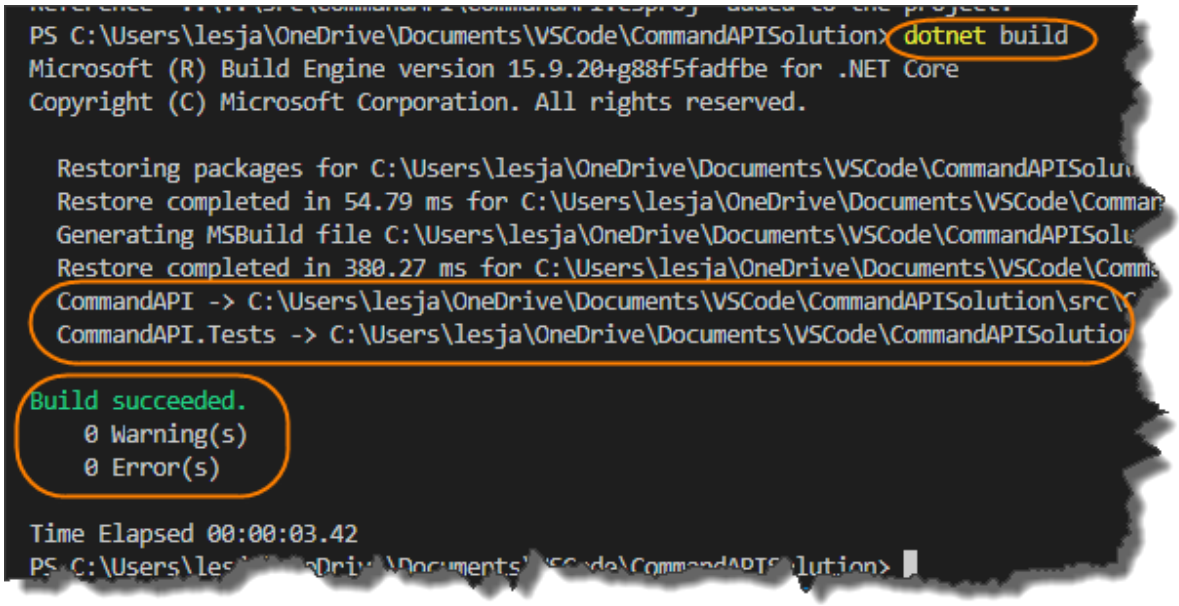
You can now build both projects, (ensure you are still in the root solution folder), by issuing:

```
dotnet build
```

Note: This is one of the advantages of using a solution file, (you can build both projects from here).

[8] In VS Code got to: File -> Open Folder… and select your solution or project folder.

Assuming all is well, the *solution build* should succeed, which comprises our 2 projects:



Celebration Check Point: Good Job! You've reached your first milestone, our app is scaffolded up and ready to rock and roll, (that means coding).

But, (there's always a but isn't there?), before we move onto the next chapter, I think a little bit about the anatomy of a ASP.NET Core app is probably appropriate. The more familiar you are with this, the easier you'll find the rest of the tutorial.

Anatomy Of An ASP.NET Core App

The table below describes the *core*[9] files and folders that you will typically encounter when you create an ASP.NET Core project. Just be aware that depending on the project, (or scaffold template), type you select, you may have additional files and folders – however the one's described below are common to most project types.

File / Folder	What is it?
.VS Code	This folder stores your VS Code workspace settings, so it's not really anything to do with the actual project. In fact if you've chosen to dev this in something other than VS Code you won't have this file, (you may have something else…)
bin (folder)	Location where final output binaries along with any dependencies and or other deployable files will be written to.
obj (folder)	Used to house intermediate object files and other transient data files that are generated by the compiler during a build.
Properties (folder) ***launchSettings.json***	Contains the ***launchSettings.json*** file. This file can be used to configure application environment variables, e.g. (Development). It is also used to configure how the webserver running your app will operate, e.g. which port it will listen on etc.
appsettings.json ***appsettings.Development.json***	File used to hold, surprise-surprise, "application settings". In the sections that follow we'll store the connection string to our database here.

[9] Core in this sense pertaining to *"part of something that is central to its existence or character"*, not .NET Core…

	Also, environment specific settings can be contained in additional settings files, (e.g. Development), as shown by the ***appsettings.Development.json*** file.
CommandAPI.csproj	The configuration for the project, principally tells us the .Net Core Framework version we're using along with other Nuget packages, (see info box below), that the application will reference and use. Also as you've seen above this is where we can place references to other projects that we need to be aware of.
Program.cs	It all starts here… This class configures the "hosting" platform, along with the "Main()" entry point method for the entire app.
Startup.cs	This class is used to configure the application services and the request pipeline.

Nuget is a package management platform that allows developers to reference and consume external, pre-packaged code that they can use in their apps. We'll add different packages to our project files as we move through the book and require extra functionality.

In short

- ***launchSettings.json***
- ***appsettings.json*** (and other environment specific settings files)
- ***CommandAPI.csproj***
- ***Program.cs***
- ***Startup.cs***

All work in symbiotic bliss with each other to get the application up and running and working according to the runtime environment. As we go through the book, we'll cover off more and more of the functions and features of each of the above when they become relevant.

However, as they are so foundational to every ASP.NET Core solution, we're going to talk briefly about both the `Program` and `Startup` classes here.

The Program & Starup Classes

The Program Class

As mentioned above this is the main entry point for the entire app and is used to configure the "hosting" environment. It then goes on to use the `Startup` class to finalise the configuration of the app.

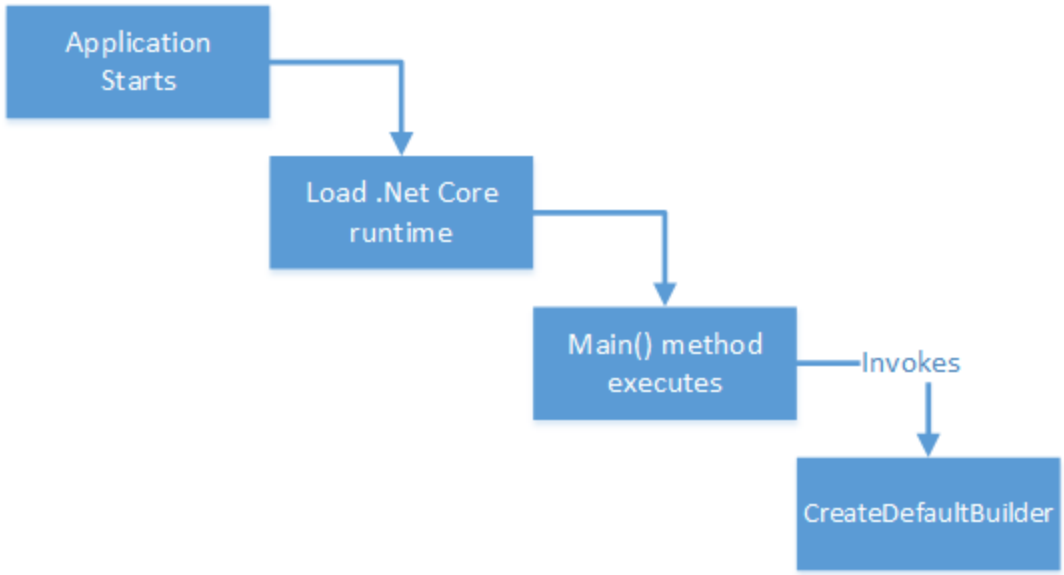
Let's take a quick look at the templated code, (that we're *not actually* going to change!), and see what it does…

Unless otherwise stated when we're working with a project it's going to be our main "API Project", (and not the unit test project). So for the examples coming up, and elsewhere in the book, reference this project first.

I'll explicitly state when we need to use the unit test project.

```
namespace CommandAPI
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

The execution sequence is as follows:



The `CreateDefaultBuilder` method uses the default builder pattern to create a web host, which can specify things like the webserver to use, config sources, as well as selecting the class we use to complete the configuration of the app services. In this case we use the default `Startup` class for this, indeed since the default contents are sufficient for our needs we'll move on in the interests of brevity.

Note: we do cover .NET Core Configuration in more detail later in the book.

The Startup Class

The `Program` class is the entry point for the app, but most of the interesting start up stuff is done in the `Startup` class. The `Startup` class contains 2 methods that we should look further at:

- `ConfigureServices`
- `Configure`

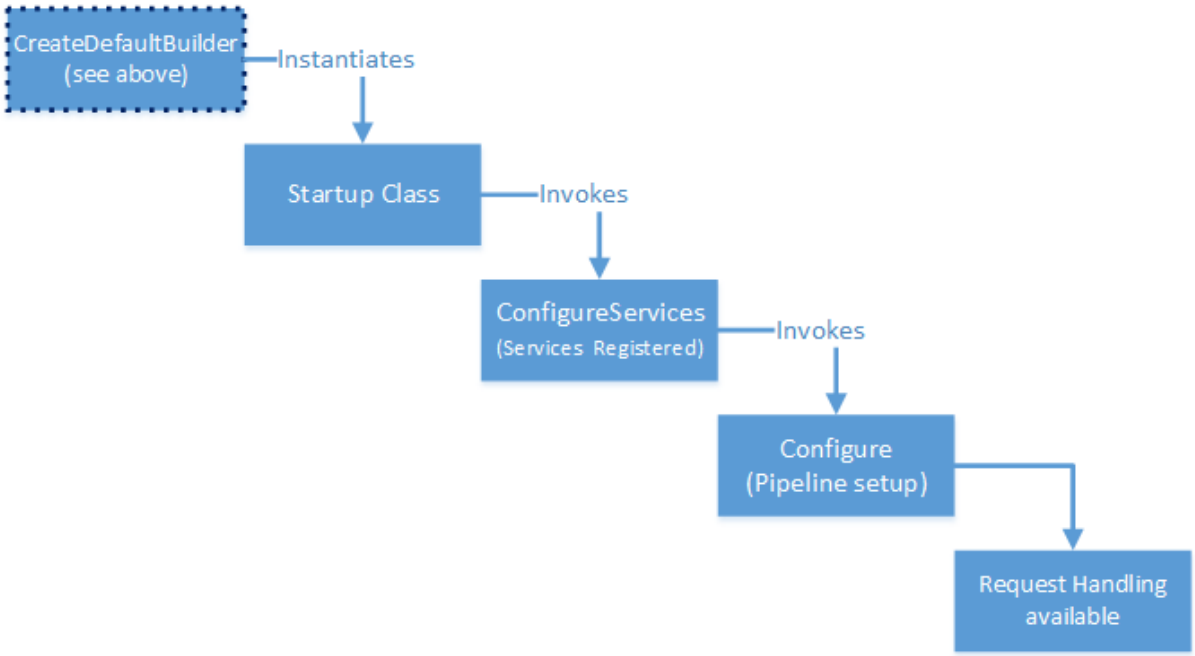
ConfigureServices

In ASP.NET Core we have the concept of "services", which are just objects that provide functionality to other parts of the application. For those of you familiar with the concept of *dependency injection*[10], this is where dependencies are registered inside the default Inversion of Control, (IoC), container provided by .NET Core.

Configure

Once services have been registered, `Configure` is then called to set up the request pipeline. The request pipeline can be built up of multiple middleware components that take (in this case http), requests and perform some operation on them.

Depending on how the multiple middleware components are created, will affect at what stage they get involved with the request and what, (if anything), they do to impact it. Just for completeness, here's another diagram, (following on from the one above).



I'm sensing that everyone has had enough of the theory – the next chapter we move into coding!

[10] This can be a tricky subject, we'll touch on it as we move through the book.

CHAPTER 5 – THE 'C' IN MVC

CHAPTER SUMMARY

In this chapter we'll go over some high-level theory on the MVC pattern, detail out our API application architecture and start to code up our API controller class.

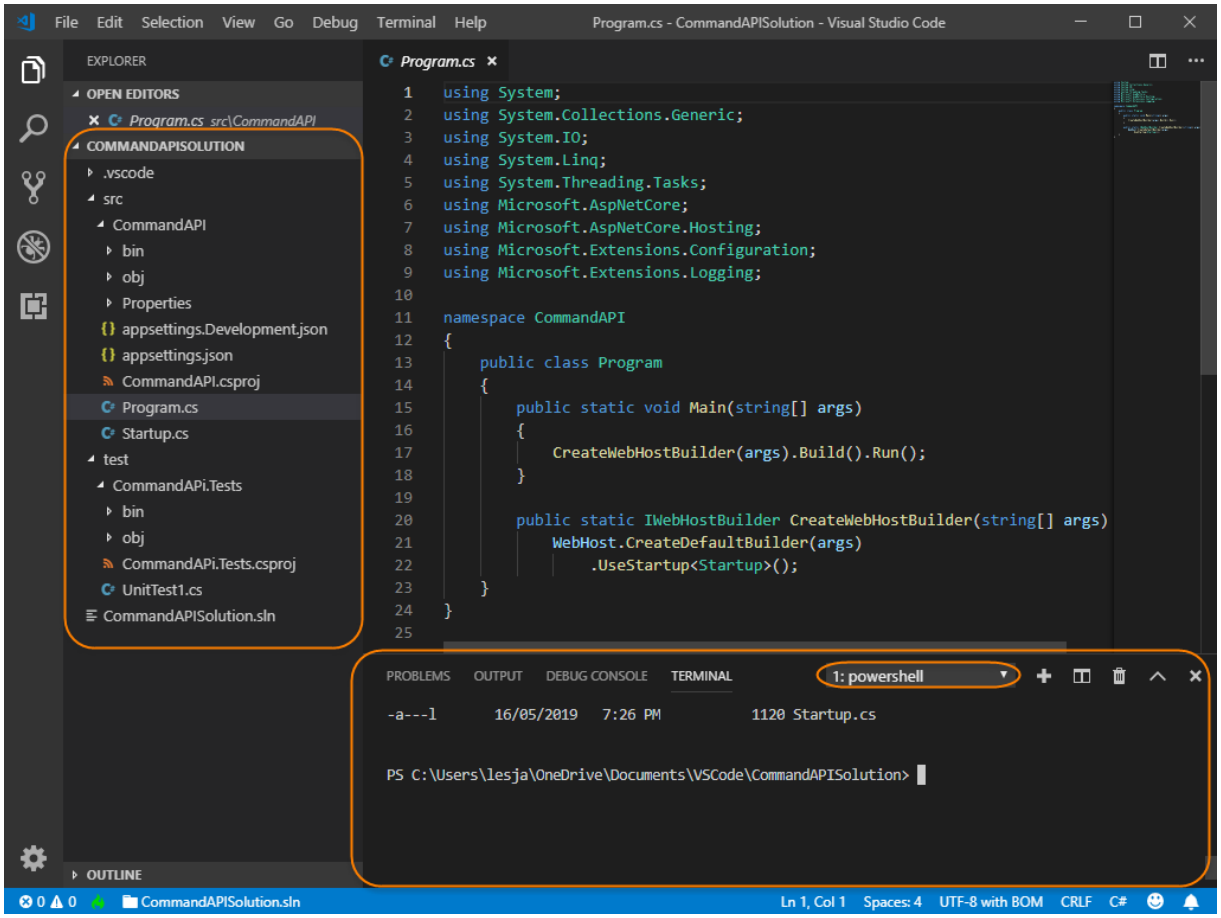
WHEN DONE, YOU WILL

- Understand what the MVC pattern is
- Understand out API Application Architecture
- Add a controller class to our API project
- Create a "hard-coded" action in our controller
- Place our solution under source control

QUICK WORD ON MY DEV SET UP

I just want to level-set here on the current state of my development set up I'm using going forward:

- I have VS Code open and running
- In VS Code I have opened the ***CommandAPISolution*** solution *folder*
- This displays my folder & file tree down the left-hand side (containing both our *projects*)
- I'm also using the integrated terminal within VS Code to run my commands
- The integrated terminal I'm using is "PowerShell" – you can change this see info box below



You can change the terminal / shell / command line type within VS Code quite easily.

1. In VS Code hit 'F1' (this opens the "command palette" in VS Code)
2. Type *shell* at the resulting prompt, and select "*Terminal: Select Default Shell*"
3. You can then select from the Terminals that you have installed

START CODING!

First let's just check that everything is set up and working ok from a very basic start up perspective. To do this from a command line type, (ensure that you're "in" the API project directory: ***CommndAPI***):

```
dotnet run
```

You should see the webserver start with output similar to the following:

```
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src\CommandAPI> dotnet run
Hosting environment: Development
Content root path: C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src\CommandAPI
Now listening on: https://localhost:5001
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

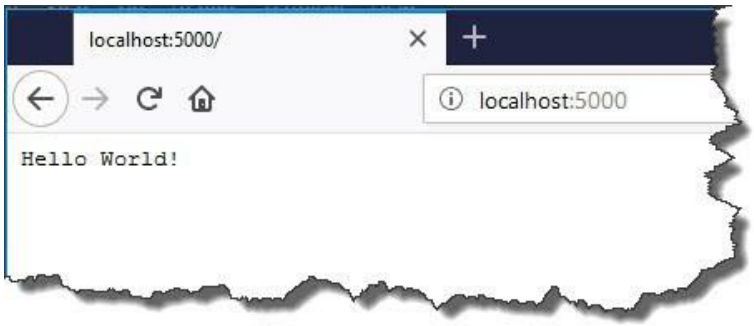
You can see that the webserver host has started and is listening on ports 5000 and 5001 for http and https respectively.

To change that port allocation you can edit the ***launchSettings.json*** file in the Properties folder, for now though there would be no benefit to that. We'll talk more about this file when we come to our discussion on setting the runtime environment in Chapter 7.

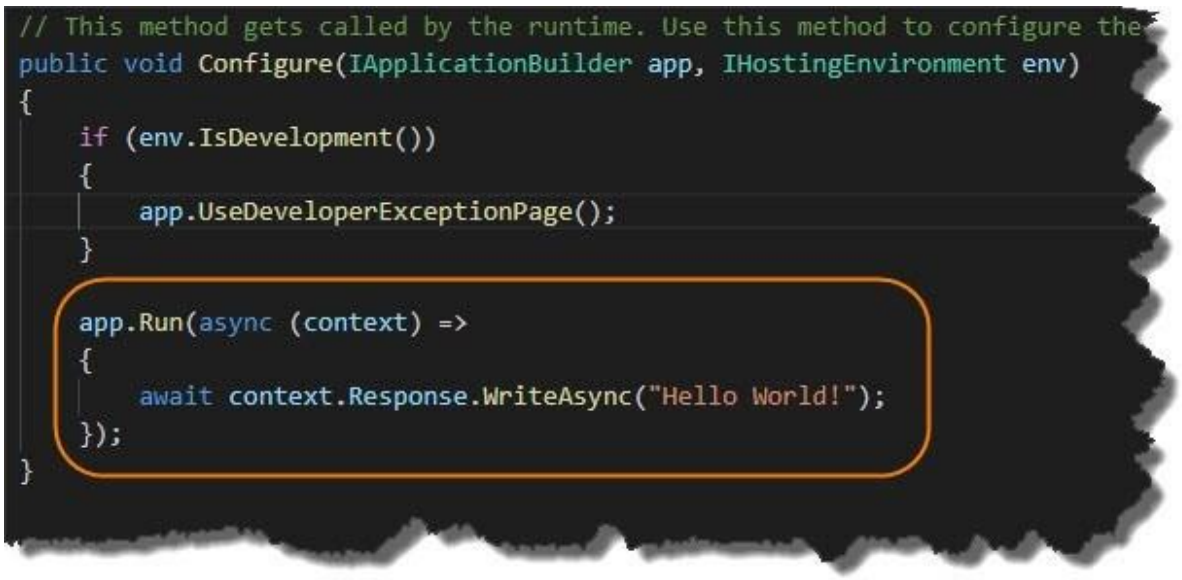
If you go to a web browser and navigate to:

http://localhost:5000

You'll see:



Not hugely useful, but it does tell us that everything is wired up correctly. Looking in the `Configure` method of our `Startup` class we can see where this response is coming from:



```
// This method gets called by the runtime. Use this method to configure the
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

Stop our host from listening, (Ctrl+C on windows – think the same for Linux / OSX), and *remove* the highlighted section of code, (shown above), from our `Configure` method. And add the highlighted code to our `Startup` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Mvc;   //SECTION 1


namespace CommandAPI
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            //SECTION 2
            services.AddMvc().
            SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            //SECTION 3
            app.UseMvc();
        }
    }
}
```

What does this code do?

1. A “using” directive to ensure we have access to the MVC namespace.

2. Adds MVC as a shared usable service, we can optionally set the compatibility version, (in this case we're using .Net Core Framework 2.2)
3. Finally we use the MVC service we registered above as part of our Request Pipeline, this will allow us to take advantage of MVC's features to build our API.

The code for the entire project can be found here on GitHub:

https://github.com/binarythistle/Complete-ASP.NET-Core-API-Tutorial

As before run the project, (ensure you save the file before doing this[11]):

```
dotnet run
```

Now navigate to the same url in a web browser, (http://localhost:5000), and we should get "nothing".

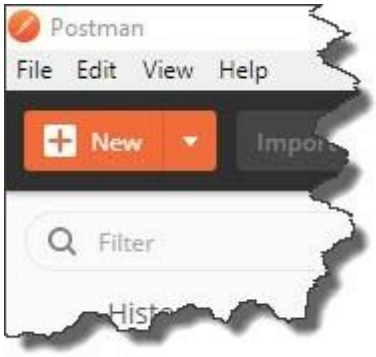
CALL THE POSTMAN

Now is probably a good time to get Postman up and running as it's a useful tool that allows you to have a more detailed look at what's going on.

So if you've not done so already, go to the Postman web site HERE and download the version most suitable for your environment, (I use the Windows desktop client, but there's a Chrome plug-in along with desktop versions for other operating systems).

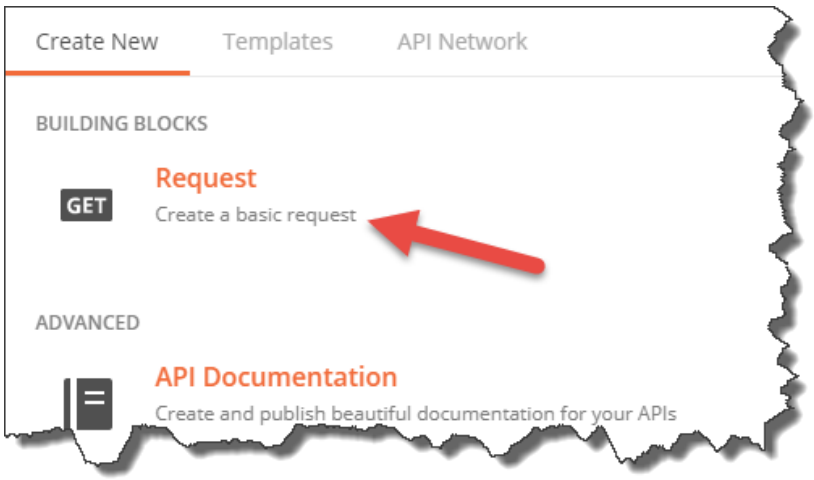
Once it's up and running,

Click "New"



The select "request"

[11] Plenty of times I've run code after making changes, and the changes were not reflected. Yes that's right – hadn't saved the file…



Give the request a simple name. e.g. “Test Request”:

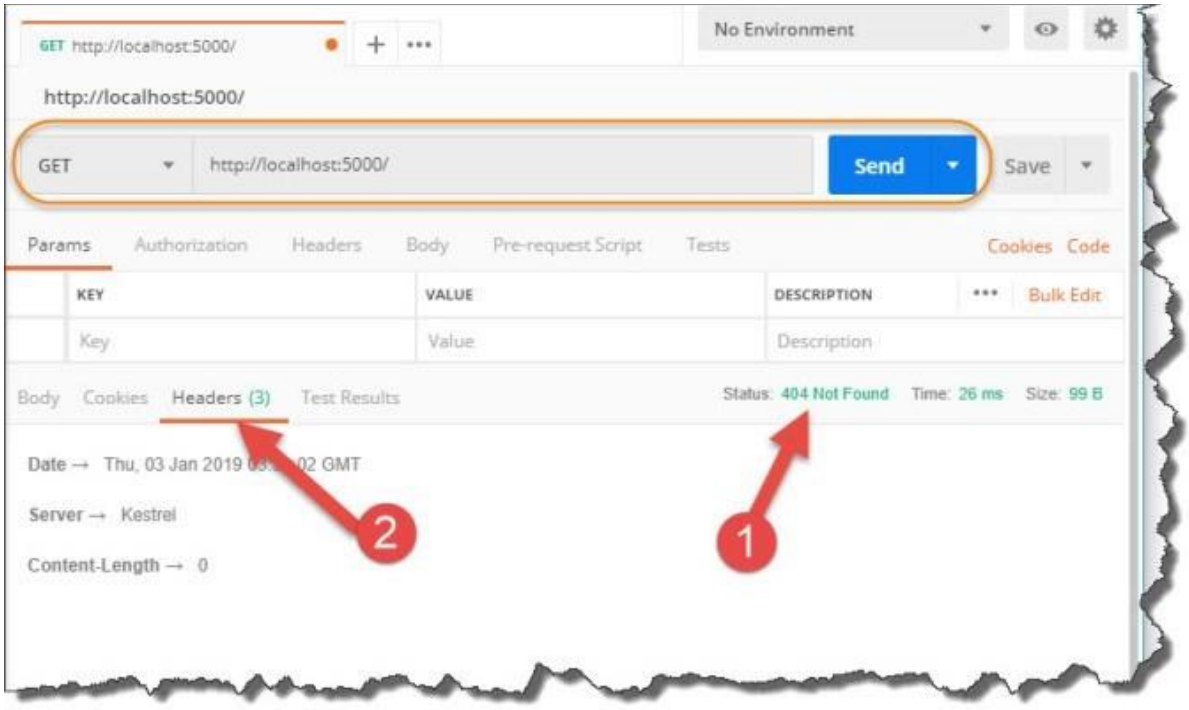
You’ll also need to create a “Collection” to house the various API requests you want to create, (e.g. GET, POST etc.):

1. Click “+ Create Collection”
2. Give it a name, e.g. “Command API”
3. Select ok, (the tick)
4. Ensure you select your newly created collection (not shown)
5. Click Save to Command API

You should then have a new tab available to populate with the details of your request. Simply type:

```
http://localhost:5000
```

into the “Enter request URL” text box, ensure “GET” is selected from the drop down next to it, and hit SEND, it should look something like:



If you’ve clicked Send then you should see a response of “404 Not Found”, clicking on the headers tab, you can see the headers returned.

We’ll return to Postman a bit later, but it’s just useful to get it up, running and tested now.

What have we broken?

We’ve not actually broken anything, but we have taken the first steps in setting up our application to use the MVC pattern to provide our API endpoint…

What is MVC?

I'm guessing if you're here you probably have some idea of what the MVC, (Model View Controller), pattern is. If not I provide a brief explanation below, but as:

1. There are already 1000's of articles on MVC
2. MVC theory is not the primary focus of this tutorial

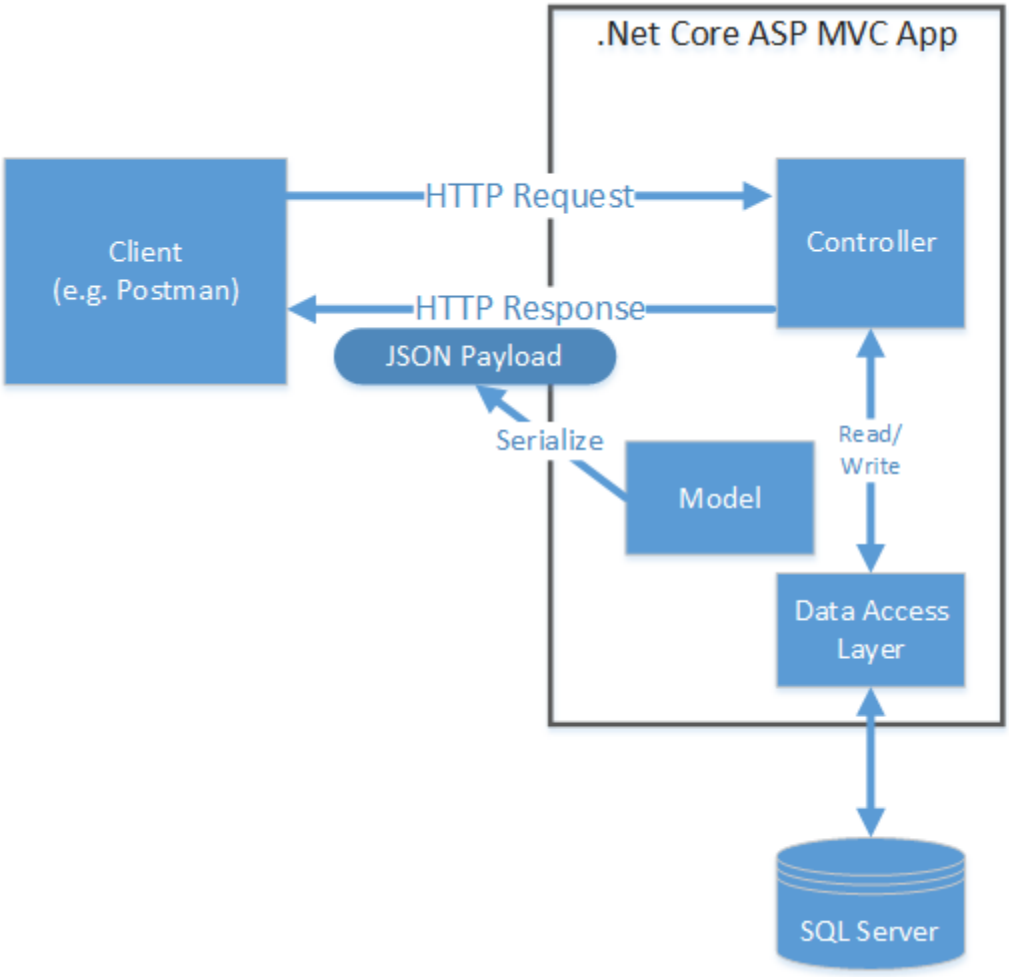
I won't go into *too* much detail.

Model, View, Controller

Put simply the MVC pattern allows us to separate the concerns of different parts of our application:

- **M**odel (our Data)
- **V**iew (User Interface)
- **C**ontroller (Requests & Actions)

In fact to make things even simpler, as we're developing an API, we won't even have any **V**iews. A high-level representation of this architecture is shown below:



Model Vs Data Access?

One area that often confused me was the difference between the Model and the Data Access Layer… While the 2 are interdependent, they are different. As simply as I can put it:

- The Model represents the data objects our application works with.
- The Data Access Layer uses the Models to mediate that data to a persistent layer*, (e.g. SQL Server DB).

Without the Data Access Layer, (and importantly the DB), while we could model data in our app, the data would be lost if for example we had a power cut, (or heaven forbid the app crashed).

* You can have a data access layer that uses an "in memory db" or non-persistent data store so theoretically you can have a data access layer that still doesn't persist data… For the most part though I think of a data access layer as being the mediator between the application and the, (persistent), data store…

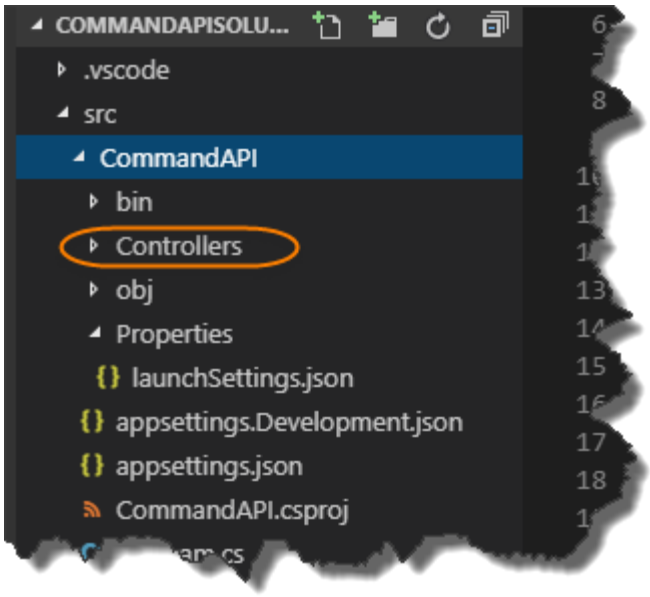
Going forward we'll leverage MVC to:

1. Create a ***C**ontroller* to manage all our API requests, (see our CRUD actions in Chapter 3)
2. Create a ***M**odel* to represent our resources (in this case our library of command line prompts)
3. Create our Data Access Layer – well use Entity Framework Core and a DB Context object to achieve this.

So what are we waiting for!?

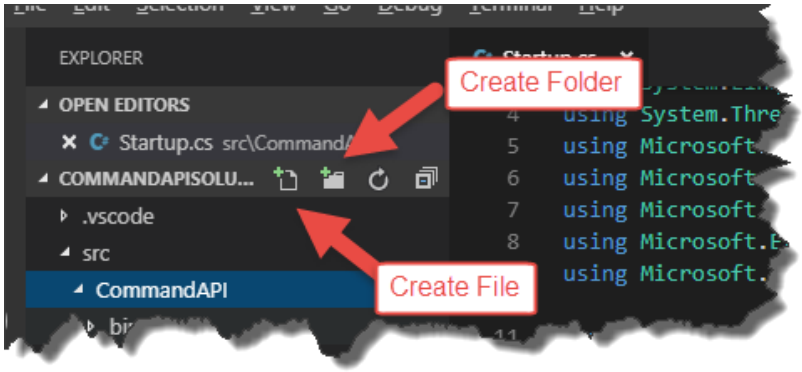
Our Controller

Making sure that you are in the main API project directory, (***CommandAPI***), create a folder named "***Controllers***":

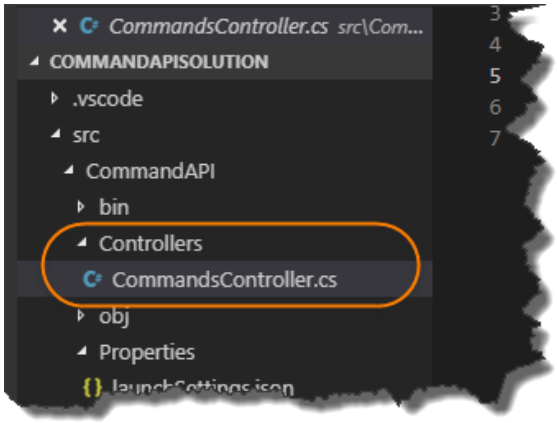


Inside the ***Controllers*** folder you just created, create a file called ***CommandsController.cs***

Quick tip: If you're using VS Code you can create both folders and files from within the VS Code directory explorer. Just make sure when you're creating either that you have the correct "parent" folder selected.



Your directory structure should now look like this:



Ensure that you postfix the ***CommandsController*** file with a ".cs" extension.

Both the folder and naming convention of our controller file follow a standard, conventional approach, this makes our applications more readable to other developers, it also allows us to leverage from the principles of "Convention over Configuration".

Now, to begin with we're just going to create a simple "action" in our Controller that will return some hard-coded JSON, (as opposed to serializing data from our db). Again this just makes sure we have everything wired up correctly.

A controller "Action" essentially maps to our API CRUD operations as listed in chapter 3, our first action though will just return a simple hard-coded string…

The code in your `CommandsController` class should now look like this:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;

namespace CommandAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CommandsController : ControllerBase
    {
        [HttpGet]
        public ActionResult<IEnumerable<string>> Get()
        {
```

```
            return new string[] {"this", "is", "hard", "coded"};
        }
    }
}
```

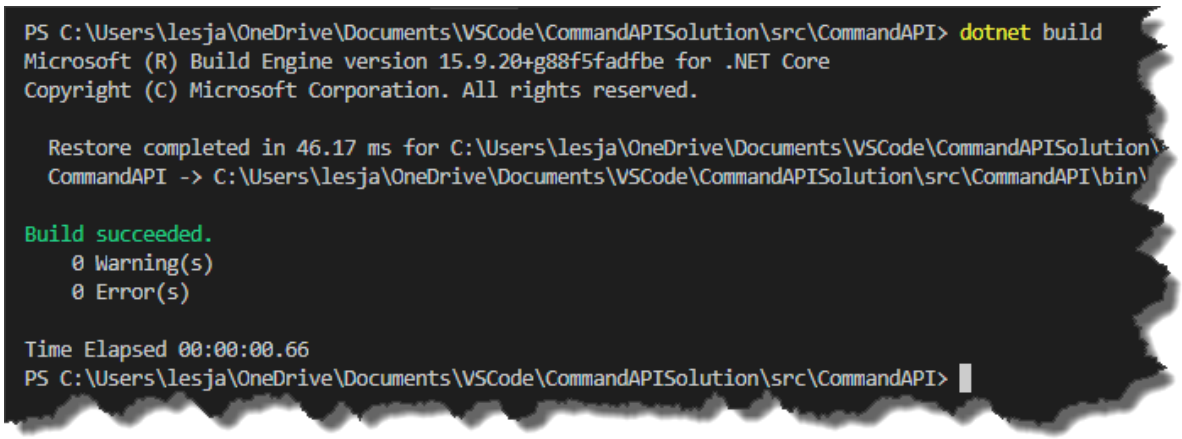
Again, if you don't fancy typing this in, the code is [available on GitHub.]

We'll come onto what all this means below, but first lets' build it…

Ensure that you don't have the server running from our example above, (Ctrl+c to terminate), save the file, then type:

```
dotnet build
```

This command just compiles, (or builds), the code. If you have any errors it'll call them out here, assuming all's well, (which is should be), you should see:



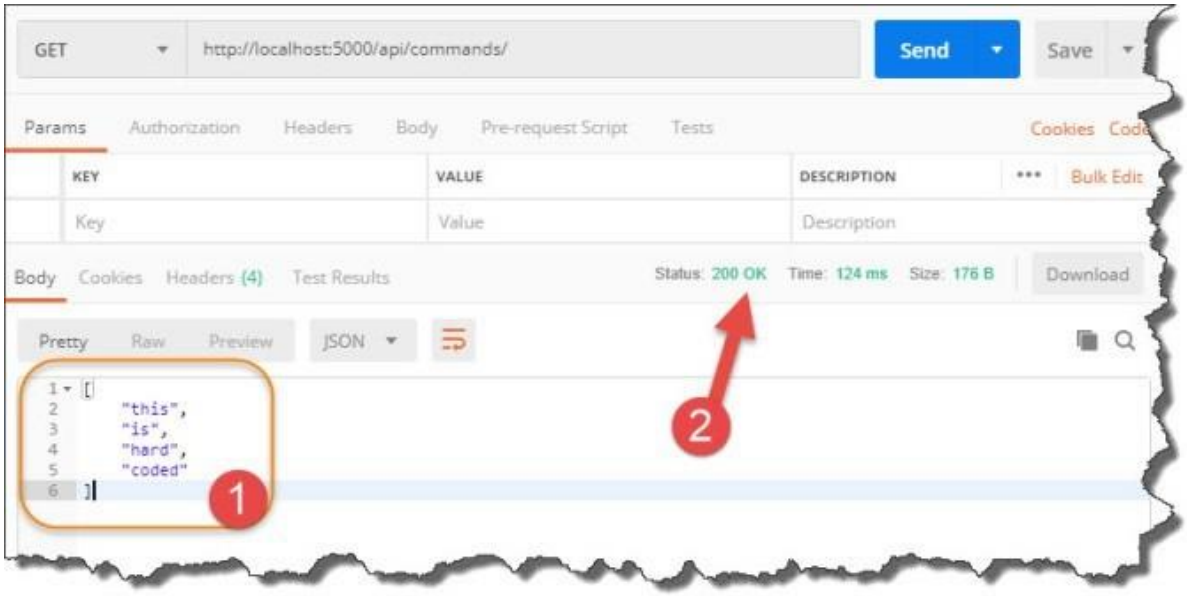
Now ***run*** the app.

Learning Opportunity: I'm deliberately not going to detail *that* command going forward now, you should be picking this stuff up as we move on. If in doubt, refer to earlier in the chapter on how to *run* your code, (as opposed to *building* it as we've just done).

Got to Postman, (or a web browser if you like), and in the URL box type:

```
http://localhost:5000/api/commands
```

Again ensure that "Get" is selected in the drop down, (in Postman), then click "Send", you should see something like:

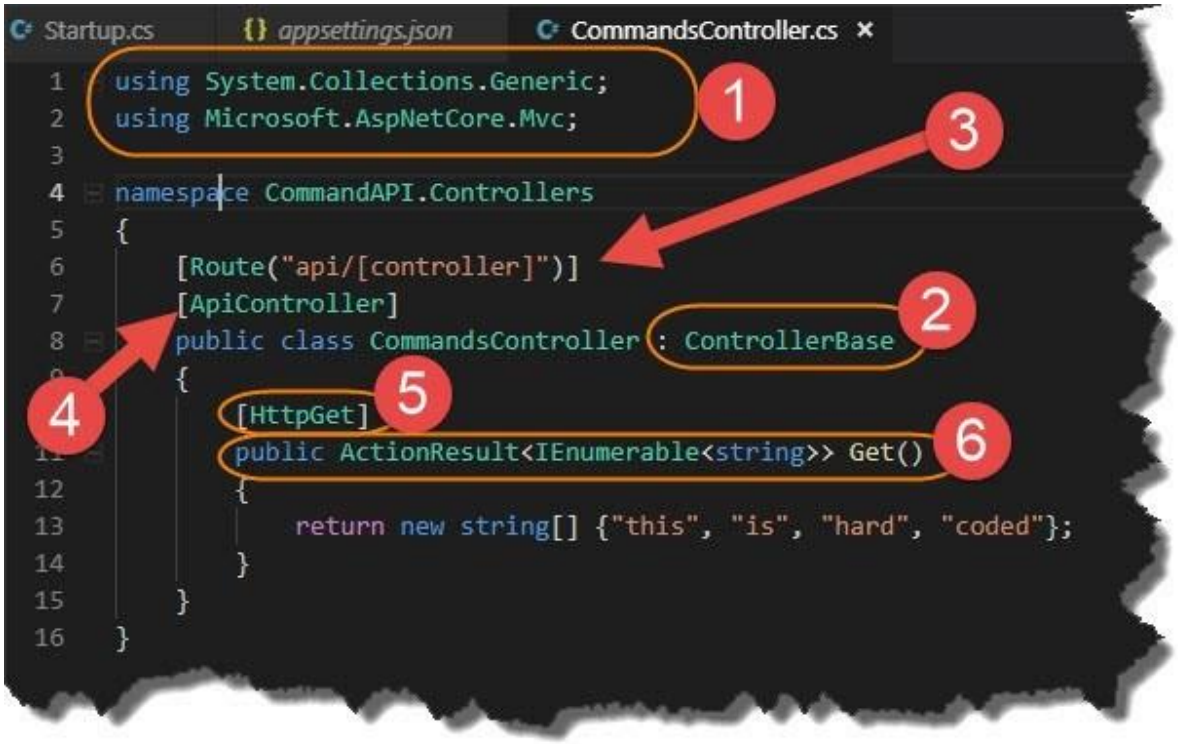


1. This is the hard-coded json string returned
2. We have a 200 OK HTTP response, (basically everything is good)

I guess technically you could say that we have implemented an API that services a simple "GET" request! Hooray, but I'm sure most of you want to take the example a little further…

Back to Our Code

Ok so that's great, but what did we actually do? Let's go back to our code and pick it apart:



1. Using Directives

We included 2 using directives here:

- `System.Collections.Generic` (supports IEnumerable)
- `Microsoft.AspNetCore.Mvc` (supports pretty much everything else detailed below)

2. Inherit from Controller Base

Our Controller class inherits from `ControllerBase` (does not provide View support which we don't need). You can inherit from `Controller` also if you like but as you can probably guess this provides additional support for Views that we just don't need.

`ControllerBase` is further detailed here on MSDN.

3. Set Up Routing

As you will have seen when you used Postman to issue a GET request to our API, you had to navigate to:

`http://localhost:5000/`**`api/commands`**

The URI convention for an API controllers is:

`http://<server_addrtess>`**`/api/<controller_name>`**

Where we use the pattern `/api/<controller_name>` following the main part of the URI.

To enable this we have "decorated" our `CommandsController` class with a route attribute:

```
[Route("api/[controller]")]
```

You'll notice that when we talk about the name of our controller from a *route perspective*, we use "`commands`" as opposed to "`CommandsController`" as we have with our class definition.

Indeed the name of our controller really is "commands", the use of the "Controller" post-fix in our class definition is another example of configuration over convention. Basically, it makes the code easier to read if we use this convention, i.e. we know it's a controller class.

4. APIController Attribute

In short decorating our class with this attribute indicates that the controller responds to web API requests, more detail on why you should use this it outlined here.

5. HttpGet Attribute

Cast your mind back to the start of the tutorial, and you'll remember that we specified our standard CRUD actions for our API, and that each of those actions aligns to a particular http verb, e.g. GET, POST, PUT etc.

Decorating our 1st simple action method with [HttpGet], is really just specifying which verb our action responds to.

You can test this, by changing the verb type in Postman to "POST" and calling our API again. As we have no defined action in our API Controller that responds to POST we'll receive a 4xx HTTP error response.

6. Our Action Result

This is quite an expansive area, and there are multiple ways you can write your "ActionResults". I've just opted for the "ActionResult" return type which was introduced as part of .Net Core 2.1.

A decent discussion on why you'd use this over IActionResult is detailed by Microsoft here.

In short you'll have an ActionResult return type for each API CRUD action.

Synchronous Vs Asynchronous?

In addition, you can also specify these actions in an Asynchronous way, (I've opted for the arguably simpler Synchronous methodology). Again, discussion of this is outside the scope of this tutorial.

Lambda Expressions

I could have written the action method in the following way to provide the same functionality:

```
[HttpGet]
public IEnumerable Get() => new string[] {"this", "is", "hard", "Coded"};
```

Again a discussion on this is outside the scope of what I want to cover, but just wanted to bring your attention to the fact you may see API Controller Actions written in this way.

If you're interested in more detail there's an article here on MSDN.

Source Control

OK this has been quite a long chapter, and we've covered a lot of ground. Before we wrap it up, I want to introduce the concept of *source control*.

What is Source Control?

Source control is really about the following 2 concepts:

1. Tracking, (and rolling back), changes in code
2. Co-ordinating those changes when there are multiple developers / contributors to the code

The general idea is that throughout a code-projects life cycle, many changes will be made to the source code, and we really need a way to track those changes, for reasons including but not limited to:

- Requirements traceability: Ensuring that the changes relate back to a requested feature / bug fix
- Release Notes: wrapping up our changes so we can publish new release notes for our app
- Rolling back: If we know what changed, (and we broke something), we can either a) fix it or b) roll back the change – a source control system allows us to do that

On top of tracking changes, the other primary reason for using a source control solution is to co-ordinate the changes to the codebase when multiple developers are working on it. If you're the only person working on your code you're not going to really conflict with yourself, (well not usually anyway)… What about when you have more that one person making changes to the same code base? How can that happen without things like over-writing each other's changes? Again, this is where a source control solution comes in to play – it co-ordinates those changes and manages or calls out conflicts should they arise.

Now we're not going to delve too deep into the workings of source control, but we are going to put our project "under source control" for two reasons:

1. To introduce you to the concept
2. So we can automatically deploy our app to production via a CI/CD[12] pipeline – more in chapter 9

GIT & GITHUB

Now there are various source control 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.

WHAT'S THE DIFFERENCE?

Git is the actual source control system that:

- You can have running on your local machine to track local code changes
- You can have running on a server to manage parallel, distributed team changes

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)
- Bitbucket, (from Atlassian – the makers of Jira and Confluence)
- Gitlabs

We're going to use both Git, (locally on our machine), and GitHub as part of this tutorial, (as mentioned in Chapter 2).

SETTING UP YOUR LOCAL GIT REPO

If you followed along in Chapter 2 you should already have Git up and running locally, if not, or you're unsure pop back to Chapter 2 and take a look.

At a terminal / command line *in* the main solution directory, (***CommandAPISolution***), type:

```
git init
```

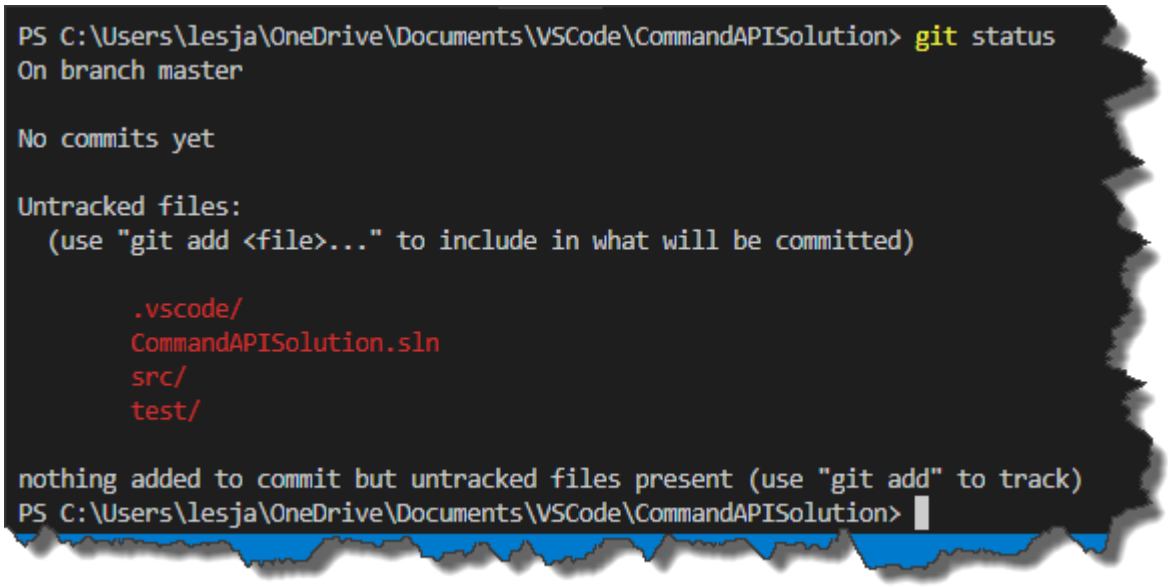
This should initialize a local Git repository in the solution directory that will track the code changes in a hidden folder called: ***.git*** (note the period '.' prefixing 'git')

Now type:

[12] Continuous Integration / Continuous Delivery (or Deployment)

```
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:



.GITIGNORE FILE

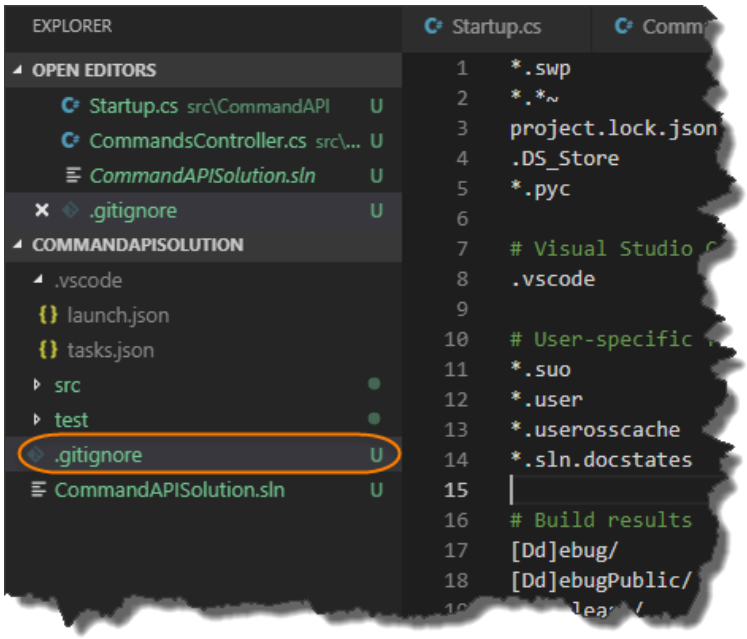
Before we start to track our solution files, (and bring them under source control), there are certain files that you shouldn’t bring under source control, in particular files that are “generated” as the result of a build, primarily as they are surplus to requirements… (and they’re not “source” files’!)

In order to “ignore” these file types you create a file in your “root” solution directory called: ***.gitignore***, (again 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!):

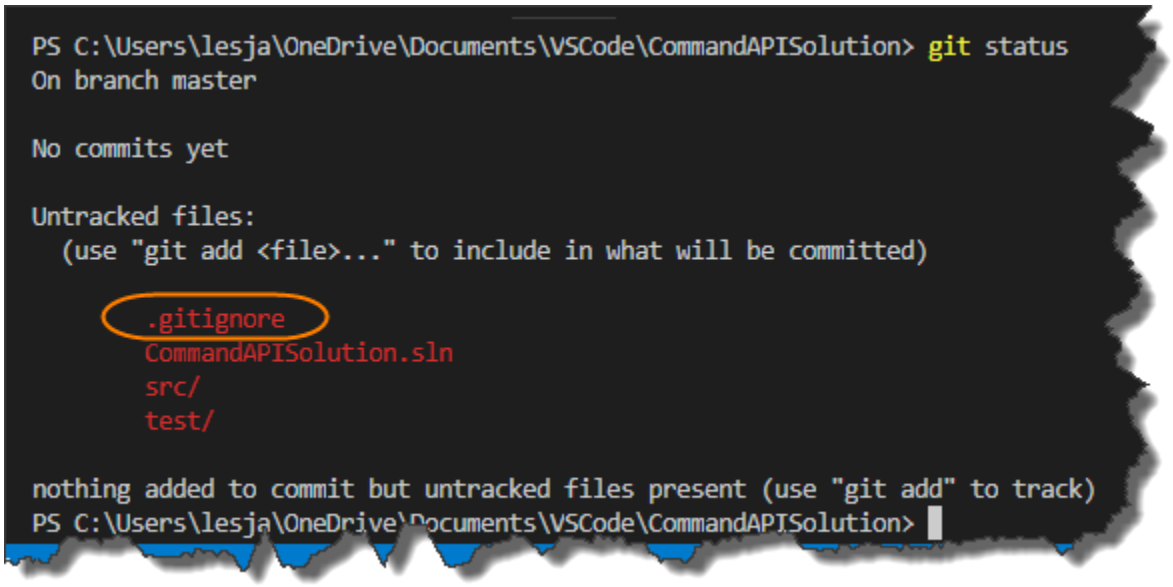
```
*.swp
*.*~
project.lock.json
.DS_Store
*.pyc

# Visual Studio Code
.VS Code

# 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 VS Code):



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



Track and Commit Your Files

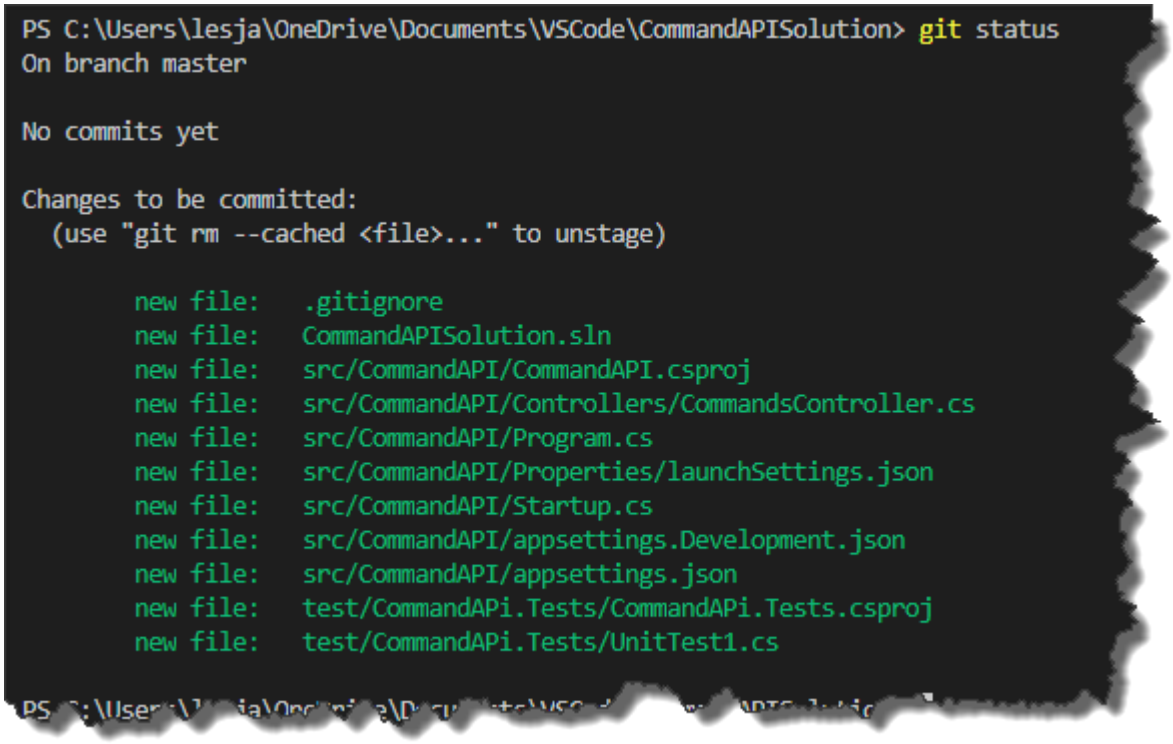
Ok we want to track “everything”, (except those files ignored!), to so type:

```
git add .
```

Followed by:

```
git status
```

You should see:



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

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

```
git commit -m "Initial Commit"
```

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\CommandAPISolution> git commit -m "Initial Commit"
[master (root-commit) c3a468b] Initial Commit
 11 files changed, 250 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 CommandAPISolution.sln
 create mode 100644 src/CommandAPI/CommandAPI.csproj
 create mode 100644 src/CommandAPI/Controllers/CommandsController.cs
 create mode 100644 src/CommandAPI/Program.cs
 create mode 100644 src/CommandAPI/Properties/launchSettings.json
 create mode 100644 src/CommandAPI/Startup.cs
 create mode 100644 src/CommandAPI/appsettings.Development.json
 create mode 100644 src/CommandAPI/appsettings.json
 create mode 100644 test/CommandAPi.Tests/CommandAPi.Tests.csproj
 create mode 100644 test/CommandAPi.Tests/UnitTest1.cs
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution>
```

A quick additional `git status` and you should see:

```
 create mode 100644 src/CommandAPI/appsettings.json
 create mode 100644 test/CommandAPi.Tests/CommandAPi.Tests.csproj
 create mode 100644 test/CommandAPi.Tests/UnitTest1.cs
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution> git status
On branch master
nothing to commit, working tree clean
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution>
```

Celebration Check Point: Good Job! We have basically placed our solution under *local* source control and have committed all our "changes" to our master branch in our 1st commit.

If this is the first time you've seen or used Git, I'd suggest you pause reading here, and do a bit of Googling to find some additional resources. It's a fairly big subject on its own and I don't want to cover it in depth here, mainly because I'd be repeating non-core content.

I will of course cover the necessary amount of Git to get the job done!

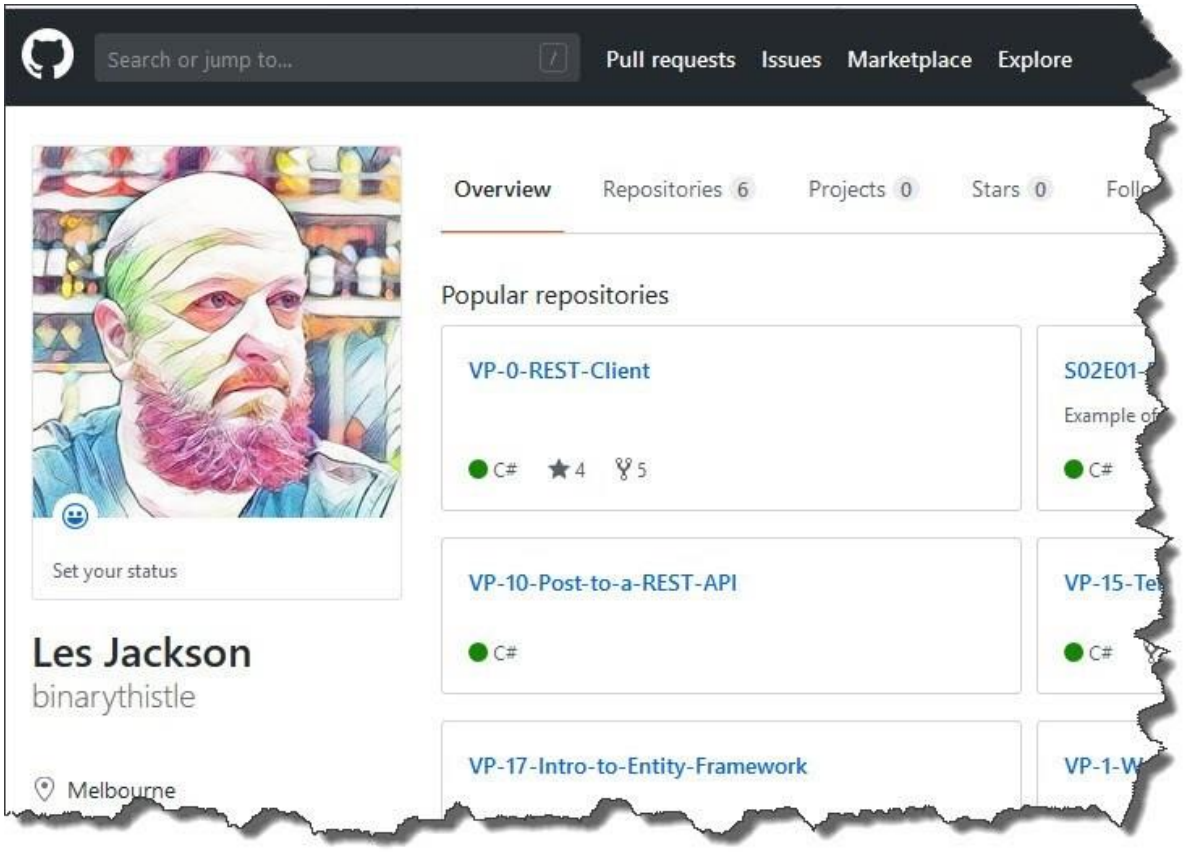
The Git web site also allows you to download the full "Pro Git" eBook, you can find that here: https://git-scm.com/book/en/v2

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 will in Chapters 9&10), we need to configure a "remote Git repository" that we will:

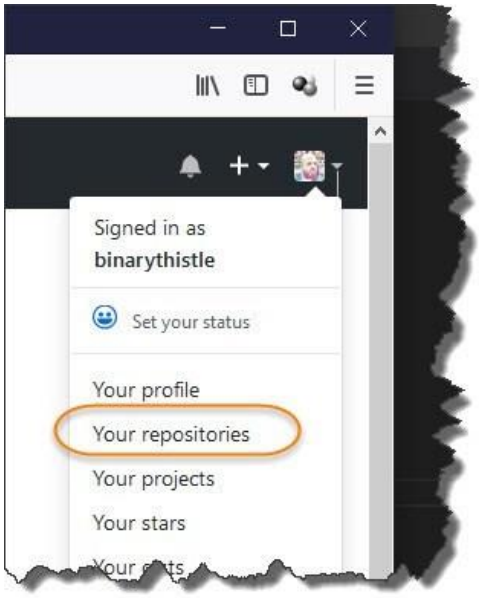
- Push to from our local machine
- Link to an Azure DevOps Build Pipeline to kick off the build process

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:

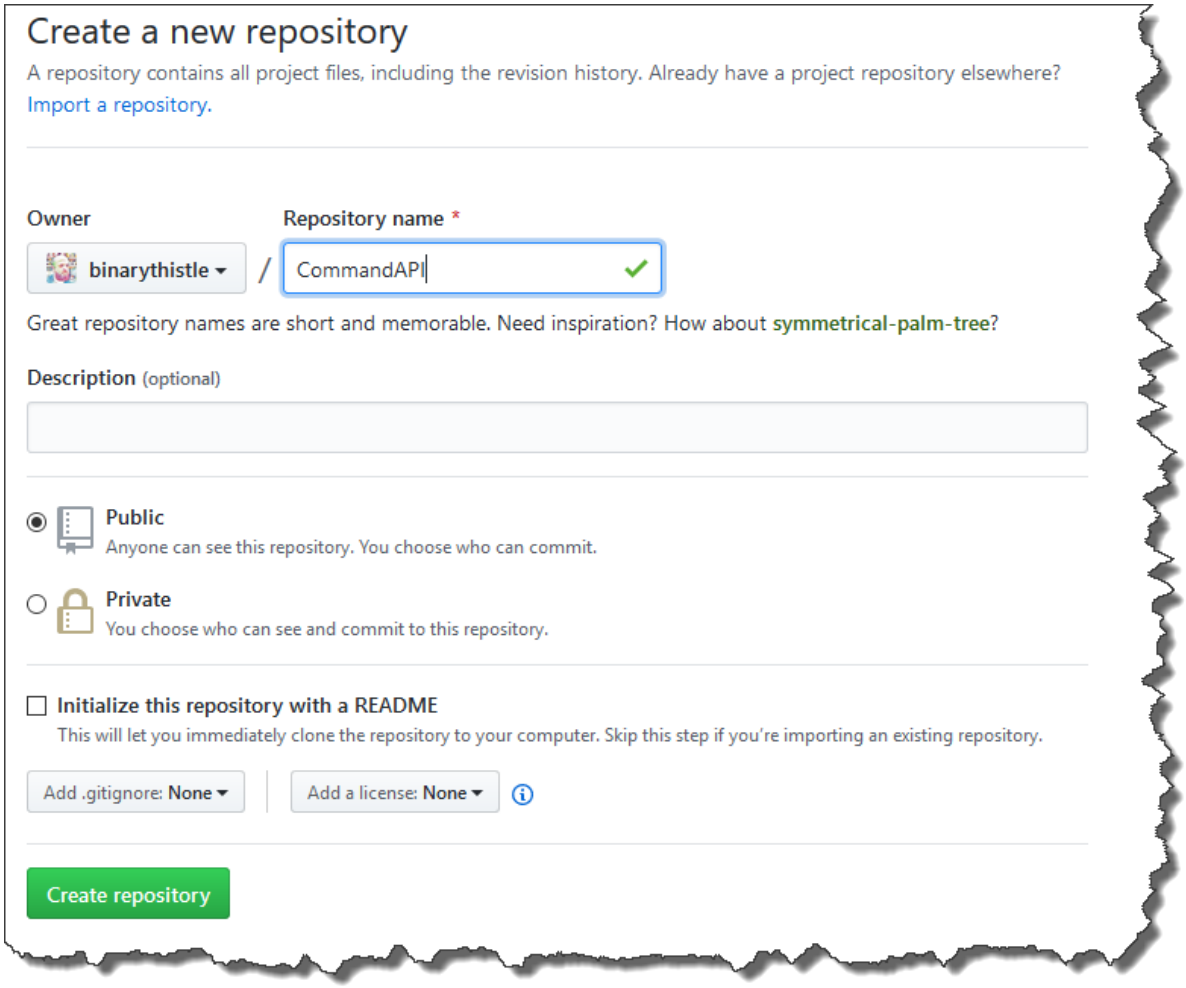


CREATE A GITHUB REPOSITORY

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":

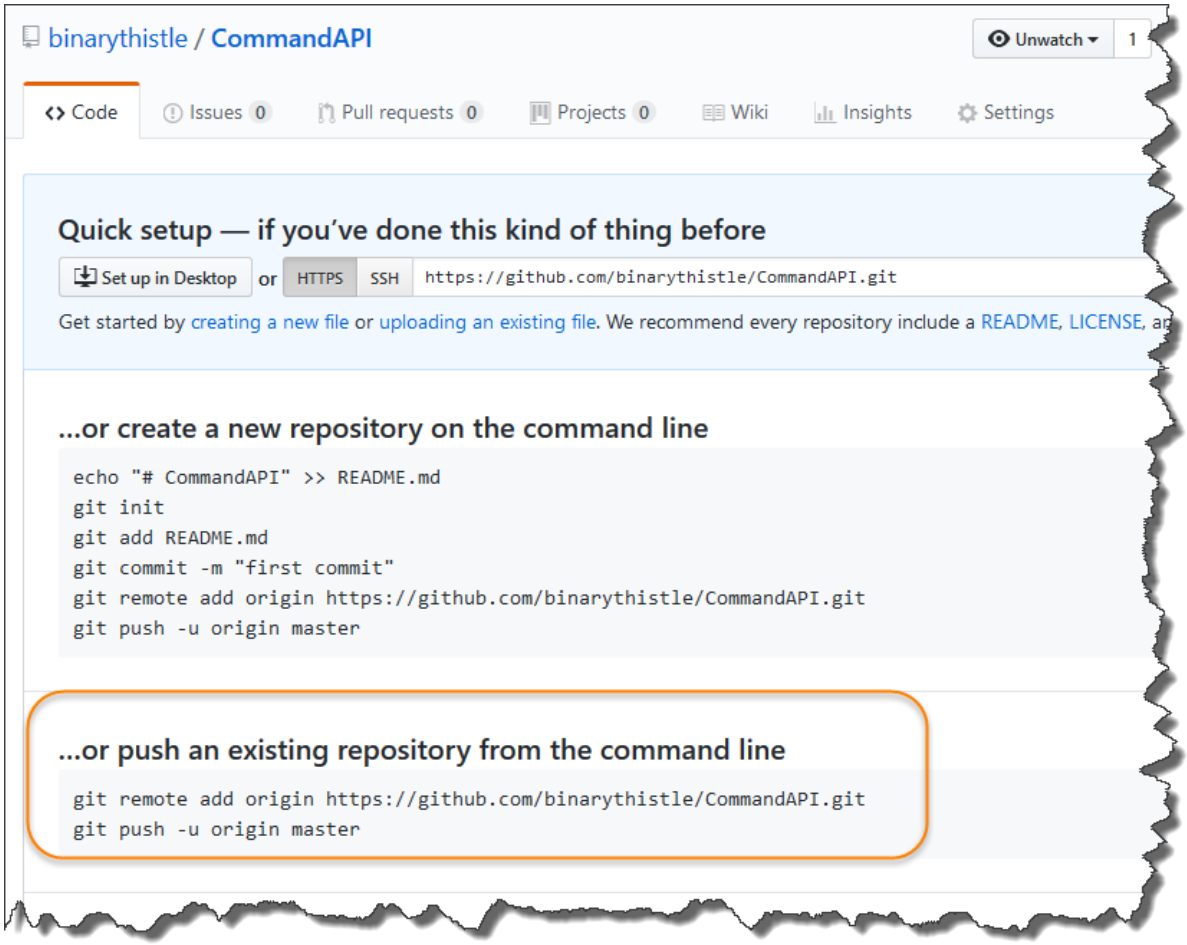


The Click "New" and you should see the "Create a new repository" screen:



Give the repository a name, (I just called mine ***CommandAPI***, 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.

Then click "Create Repository", you should see:



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\CommandAPISolution> git remote add origin https://github.com/binarythistle/CommandAPI.git
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution> 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), 3.63 KiB | 1.21 MiB/s, done.
Total 19 (delta 0), reused 0 (delta 0)
To https://github.com/binarythistle/CommandAPI.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution>
```

Les's Personal Anecdote: You may get asked to authenticate to GitHub when you issue the second command: `git push -u origin master`

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!

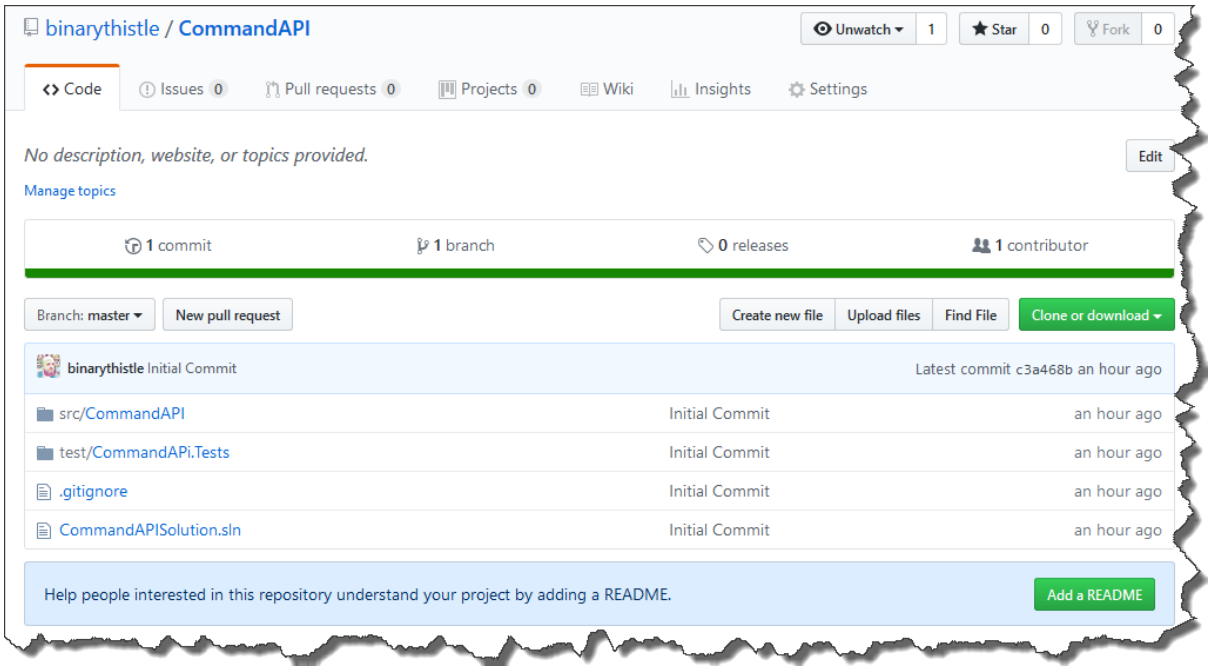
So What Just Happened?

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!



You’ll notice “Initial Commit” as a comment against every file and folder – seem familiar?

Well that’s it for this Chapter – Great Job!

CHAPTER 6 – OUR MODEL

CHAPTER SUMMARY

In this chapter we're essentially going to introduce "data" to our API, (our Model & Data Access classes), which barring a few additional changes, completes the code for our API.

WHEN DONE, YOU WILL

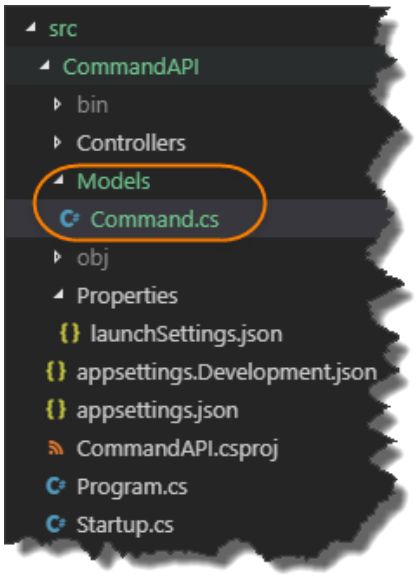
- Understand what a "Model" class is and code one up
- Use Entity Framework Core to create a Data Access (DBContext) class
- Use our Model and DBContext classes to store data in SQL Server Database

OUR MODEL

Ok so we've done the "Controller" part of the MVC pattern, (well a bit of it, it's still not fully complete – but the groundwork is in), so let's turn our attention quickly to the Model part of the equation.

Just like with our Controller, the first thing we want to do is create a ***Models*** folder in our main project directory.

Once you've done that, create a file in that folder and name it ***Command.cs***, your directory and file structure should look like this:



Once created, lets code up our "Command" model – it's super simple and when done should look like this:

```
namespace CommandAPI.Models
{
    public class Command
    {
        public int Id {get; set;}
        public string HowTo {get; set;}
        public string Platform {get; set;}
        public string CommandLine {get; set;}
    }
}
```

As promised, very simple, just be sure that you've specified the correct namespace:

```
CommandAPI.Models
```

The rest of the class is a simplistic model that we'll use to "model" our command line snippets. Possibly the only thing really of note is the Id attribute.

This will form the Primary Key when we eventually create a table in our SQL Server Db, (noting this is required by Entity Framework Core.)

Additionally, it conforms to the concept of "Convention over Configuration". I.e. we could have named this attribute differently, but it would potentially require further configuration so that Entity Framework could work with it as a primary key attribute. Naming it this way, however, means that we don't need to do this.

As we have made a simple, yet significant change to our code, let's add the file to source control, commit it, then push up to GitHub, to do so, issue the following commands in order:

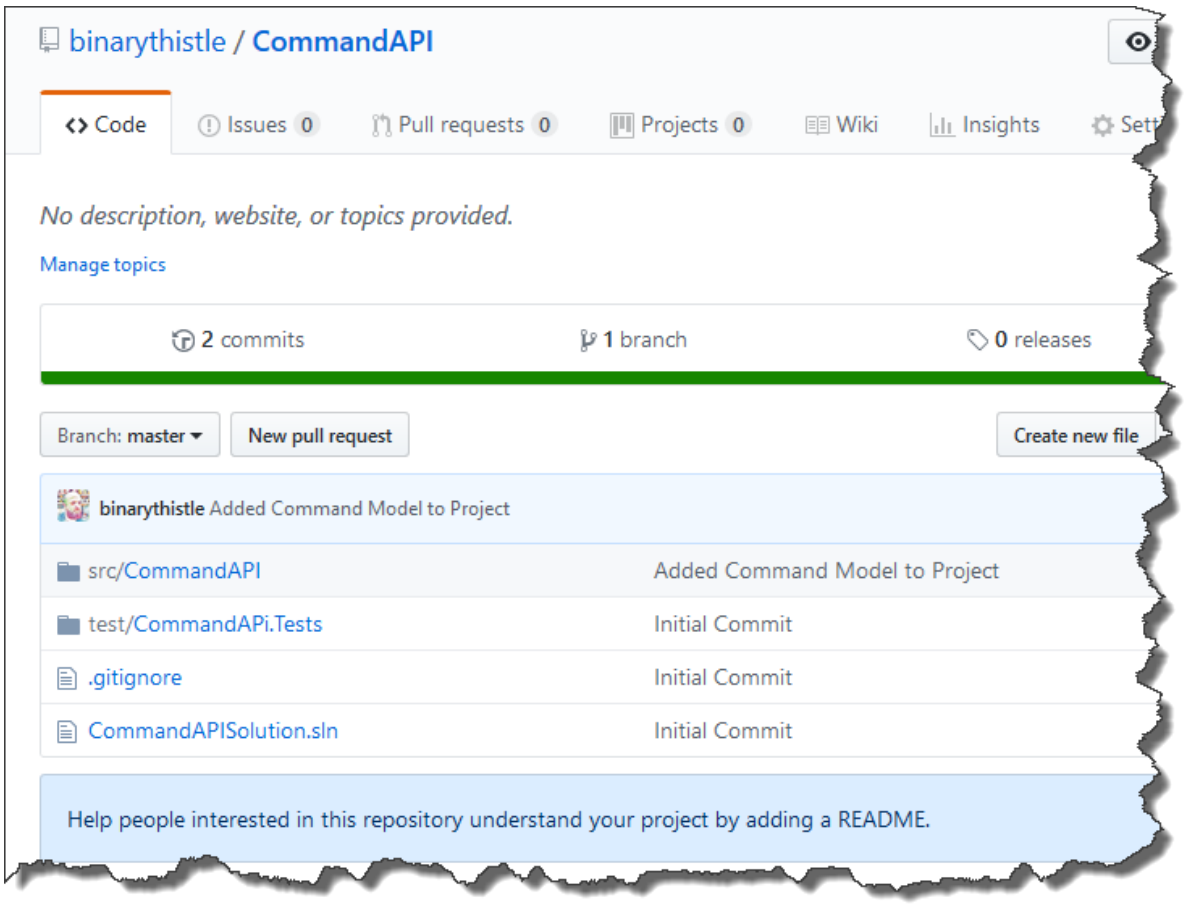
```
git add .

git commit -m "Added Command Model to API Project"

git push origin master
```

You have used these all before, but to reiterate:

- 1st command adds all files to our local Git repo, (this means our new ***Command.cs*** file)
- 2nd command commits the code with a message
- 3rd command pushes the commit up to GitHub

If all worked correctly, you should see:

- VS Code represents the commit by colouring the Command.cs file as white
- The commit has been pushed up to GitHub, see below:



Learning Opportunity: Looking at the GitHub page above, how can you tell which parts of our solution we included in the last commit and which were only included in the *initial commit*?

Tying it Together

Ok so we have:

- A Controller (it only returns hard-coded data)
- A Model (doesn't do much at the moment)

It all seems rather dis-jointed…

So for me the component that ties all this together is the *data access component* of the solution, so there are a few steps we need to take in order to be successful here.

Database

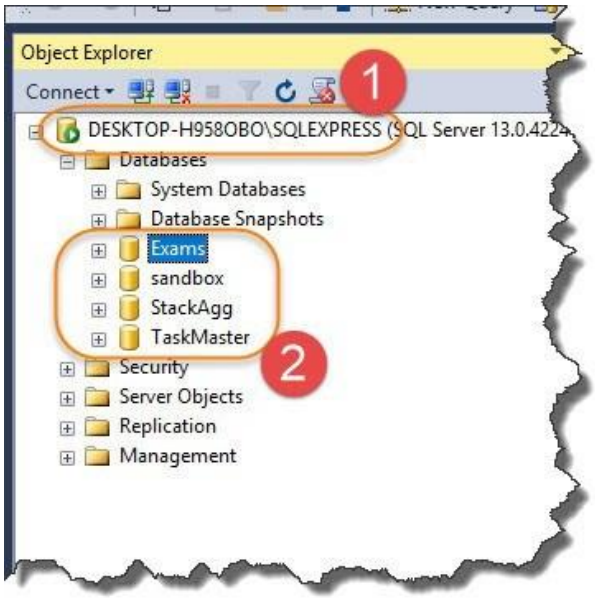
Now as alluded to in Chapter 2, we don't really need a persistent data store to get an API demo working, but for me if I didn't provide that as part of an example, I'd feel the book was incomplete.

Anyway, I decided to use Microsoft SQL Server for our repository basically because:

1. I like SQL Server
2. I know SQL Server
3. I already have it installed

So I'm now going to assume you have SQL Server Installed, (I also installed the management tool suite – currently this is only available on Windows though).

Firing up SQL Server Management Studio, after connecting and authenticating to our server you should see something similar, (if this is a new install for you, you won't yet have any user-created databases):



1. Our SQL Server Instance
2. Databases I'd previously created for other projects

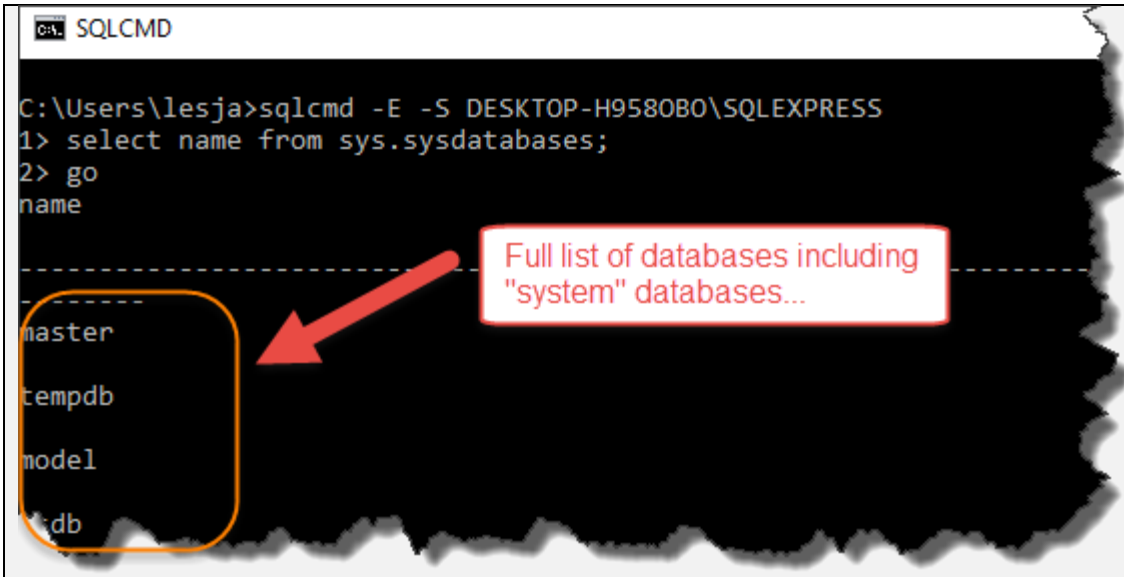
SQL COMMAND LINE

If you don't have SQL Server Management Studio installed, (Linux, OSX or lazy Windows users), you can use the Command Line tool to connect to SQL Server and run queries.

This is not my preferred approach but it does work. In the above example, to connect to SQL Server and list the databases there using the SQLCMD tool, you'd do the following.

- Open a Command Line
- Type: `sqlcmd -S <Server Name> -U <user id> -P <password>`
- Type: `select name from sys.sysdatabases;`
- Type: `GO`

An example is shown below, (I'm using Windows Integrated security so I don't need to supply a User ID and Password – more about that later):



Again, I don't really want to go into too much detail about SQLCMD, other than to point it out as an alternative to those of you that don't have access to SSMS. Where relevant I'll make reference to the SQLCMD alternative if it's required to follow the tutorial.

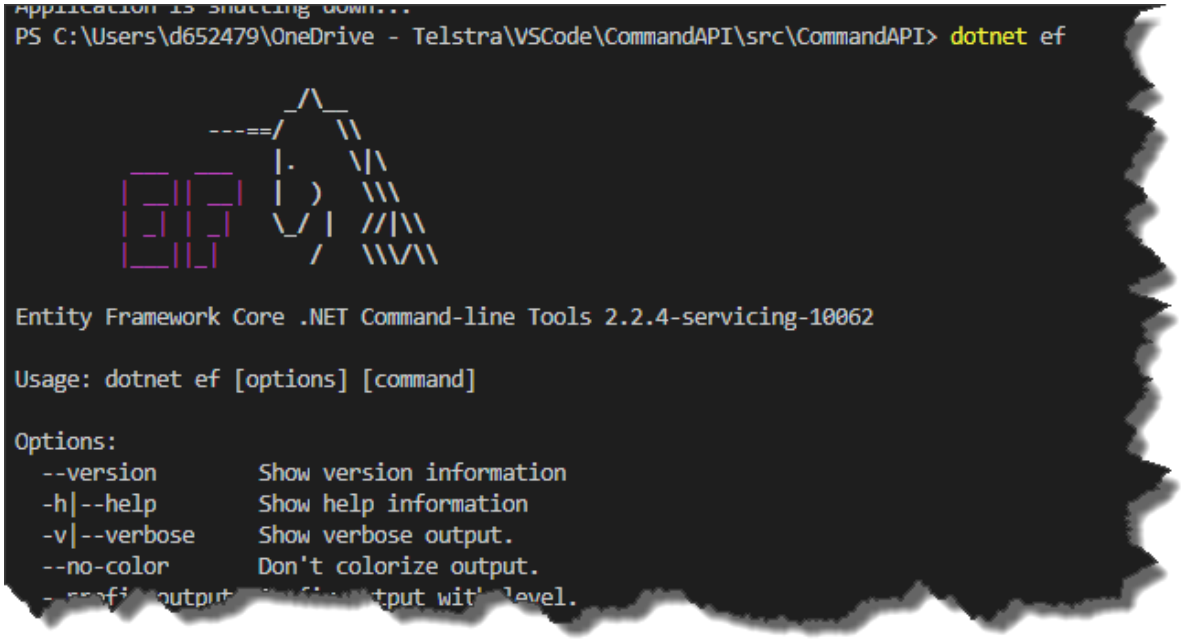
Entity Framework Command Line Tools

We're going to make use of what's called the Entity Framework Core Command Line tools, (they basically allow you to create migrations, update the database etc, don't worry if you don't know what that means yet!). Just trust me, we need the tools!

First check if you already have them installed, to do so type:

```
dotnet ef
```

You should see output similar to the following if you do:



If you don't see that, you'll need to add them, to do so open the "CommandAPI.csproj" file, you should find this in the project root. Remembering the "Anatomy of .Net Core Web App" section above, the ".csproj" file is where we add any references to additional Nuget packages we want to use…

Opening that file you should see something similar to the following:

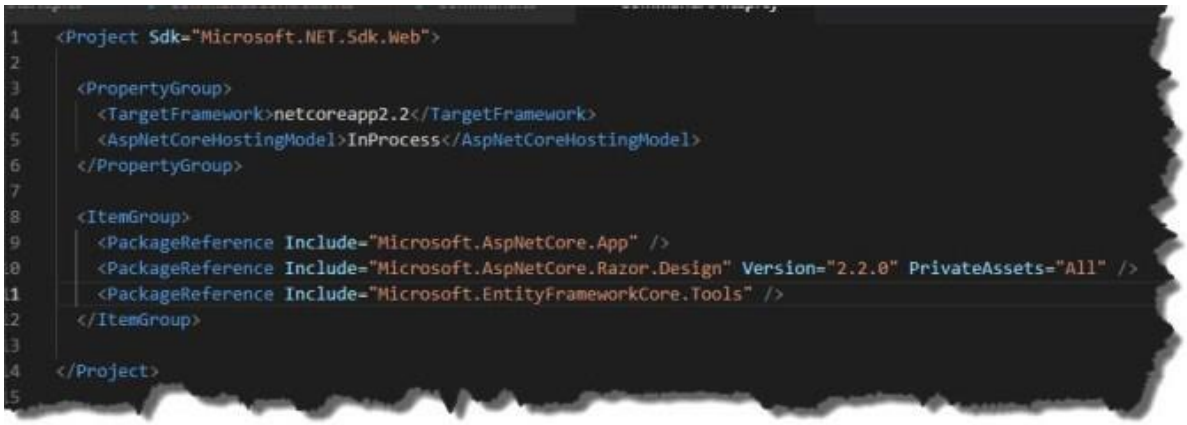

```
<Project Sdk="Microsoft.NET.Sdk.Web">

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

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0" PrivateAssets="All" />
  </ItemGroup>

</Project>
```

1. Tells us the Target .Net Core Framework we're using, in this case it's 2.2
2. Lists the existing packages we've referenced (we need to add one here)

In the `<ItemGroup>` section, add the following entry:

```
<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" />
```

So your .csproj file should look like:

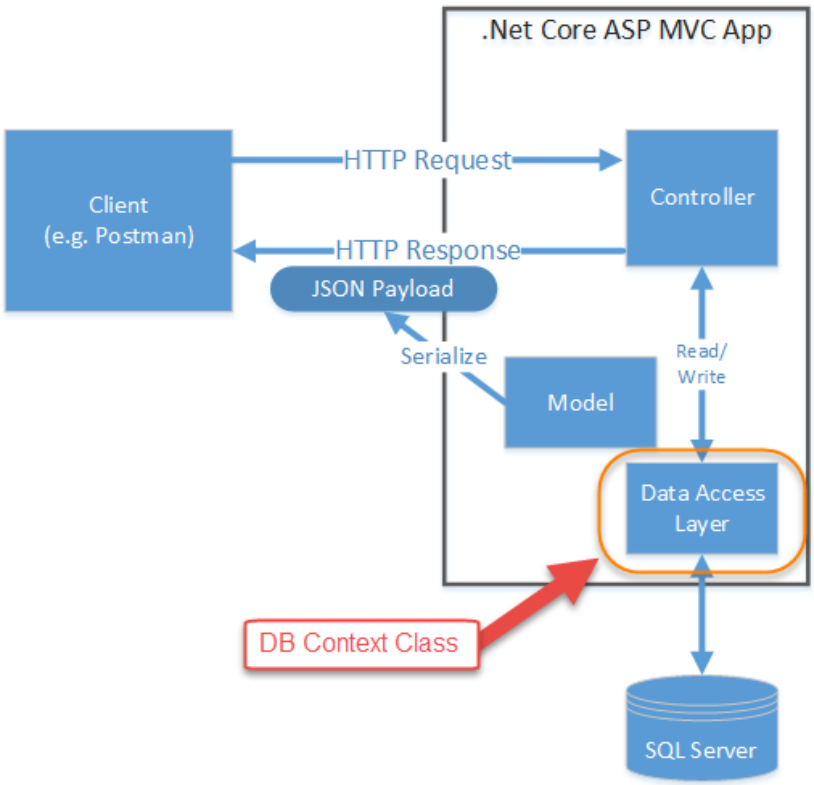


```
<Project Sdk="Microsoft.NET.Sdk.Web">

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

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0" PrivateAssets="All" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" />
  </ItemGroup>

</Project>
```

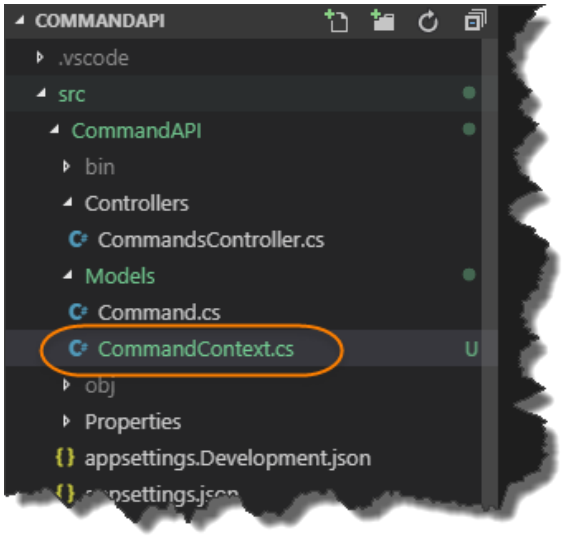
Ensure you save the file, (VS Code will load the dependencies), and you should now be able to issue the `dotnet ef` command successfully this time.

Create Our DB Context

The next step in producing the data access layer via Entity Framework Core, (EFC), is to create a Database Context Class. In short though the DBContext class acts as a representation of the Database and mediates between our data Models and their existence in the DB, as shown below:



We'll create the DBContext class in the "Models" folder, so create a new file called: ***CommandContext.cs*** and place it in the Models folder, it should look like this:



Now update the code in the ***CommandContext.cs*** file to mirror the following, be sure to include the "using" directive:

```
using Microsoft.EntityFrameworkCore;

namespace CommandAPI.Models
{
    public class CommandContext : DbContext
    {
        public CommandContext(DbContextOptions<CommandContext> options) :
base(options)
```

```
        {

        }

        public DbSet<Command> CommandItems {get; set;}
    }
}
```

Some points of note:

- Ensure you have the `EntityFrameworkCore` using statement.
- Our class inherits from `DBContext`
- Really important we create a `DbSet` of Command objects

While you can think of the `DBContext` class as a representation of the Database, you could think of a `DbSet` as a representation of a table in the Database. I.e. we are telling our `DbContext` class that we want to "model" our Commands in the Database, (so we can persistently store them as a table).

This means that we can choose which classes, (model classes), we want to put under `DBContext` control and hence represent in the DB.

Save the file and perform a `dotnet build` to ensure there are no compilation errors. As we've added a new class it's probably worth performing the "trifecta" of Git commands to:

- Place the new untracked file under source control
- Commit the class to the repository (with a message)
- Push the code up to GitHub

Learning Opportunity: Try and remember the git commands that you need to issue in order to achieve the above – I'm not going to detail them again.

If you can't remember, refer to Chapter 5.

UPDATE APPSETTINGS.JSON

Ok so that's all well and good, but there is still a disconnect between the physical SQL Server DB and our application, (specifically our `CommandContext` class).

For those of you that have done a bit of programming before, you won't be surprised to hear that we have to provide a "Connection String" to our application that essentially tells it how to connect to our database server.

We'll place out DB connection string in our ***appsettings.json*** file.

Before we do this though, we want to create a SQL Server Login that we can use to authenticate to the DB with.

I outline 2 approaches for this:

1. Using SQLCMD
2. Using SSMS (Windows Users Only)

Creating a SQL Server Login – SQLCMD

To create a SQL Server Login using the SQL Command line, open a separate command window and type:

```
sqlcmd -S <Server Name> -U <user id> -P <password>
```

Note: if your using Windows and you installed SQL Server on your local machine, you won't need to supply a user id and password. Linux / OSX user will need to use the 'SA' account you created at install time.

Once you're at the interactive sqlcmd command prompt type:

```
create login <loginame> with password = '<password>'
```

E.g.:

```
Create login cmdAPIDBLogin with password = 'ABadPassword1234!';
```

Hit return then type: `GO`

This creates a database login on our SQL Server.

We then need to assign permissions for this user to create and administer the database we have yet to create, (we'll do this via the Entity Framework Command Line).

The permission I assign to this login is probably more that you'd want to give it. To keep our example flowing we'll go with it.

At the SQLCMD prompt type:

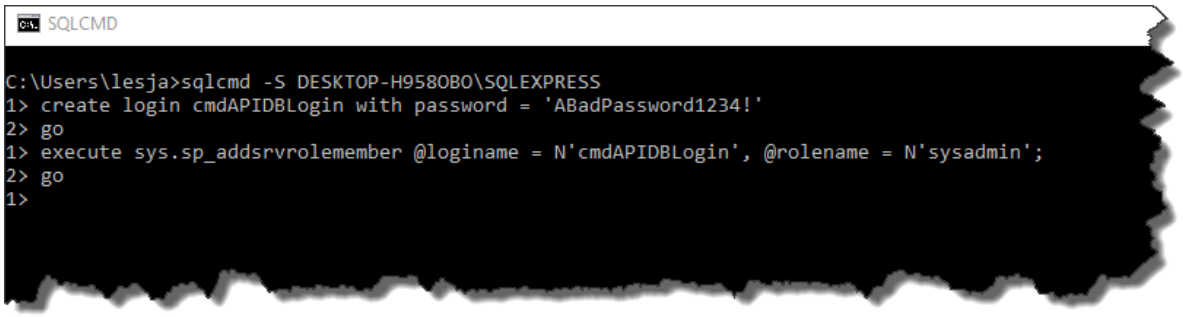
```
execute sys.sp_addsrvrolemember @loginame = N'cmdAPIDBLogin', @rolename= N'sysadmin';
```

This assigns the 'sysadmin' role to the cmdAPIDBLogin login, which pretty much gives us god rights, (again you'd want to pair this down).

Warning! Be careful if you're copy /pasting these commands into a terminal, I've had issues with the single quotes ['] being copied over correctly. If you have an issue, re-type the single charters at the command line. Refer to the screen shot below if in doubt.

Hit return then type: `GO`

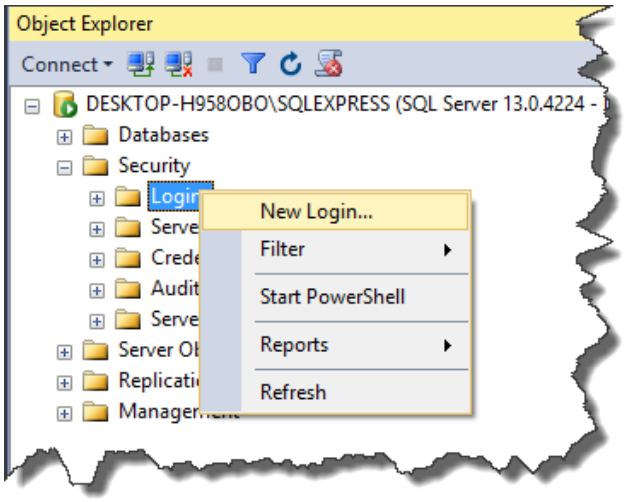
For clarity, these steps are detailed below:



Creating a SQL Server Login – SSMS

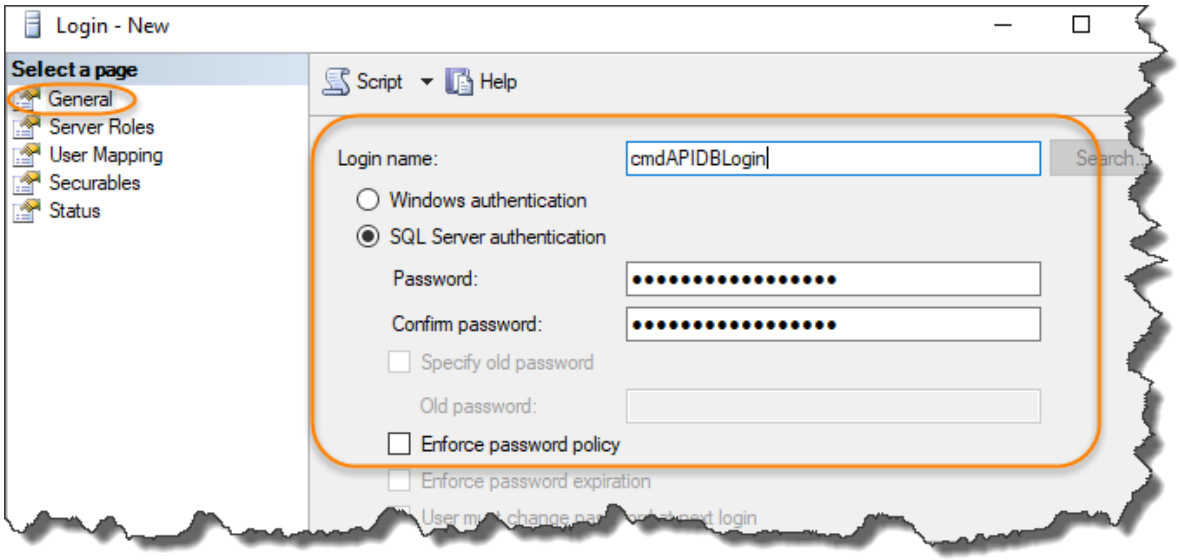
If you're a Windows user you can of course use SSMS to create the login, to do so:

- Start SSMS & Logon
- Expand the Security Folder
- Right Click the Logins folder and select "New Login…"



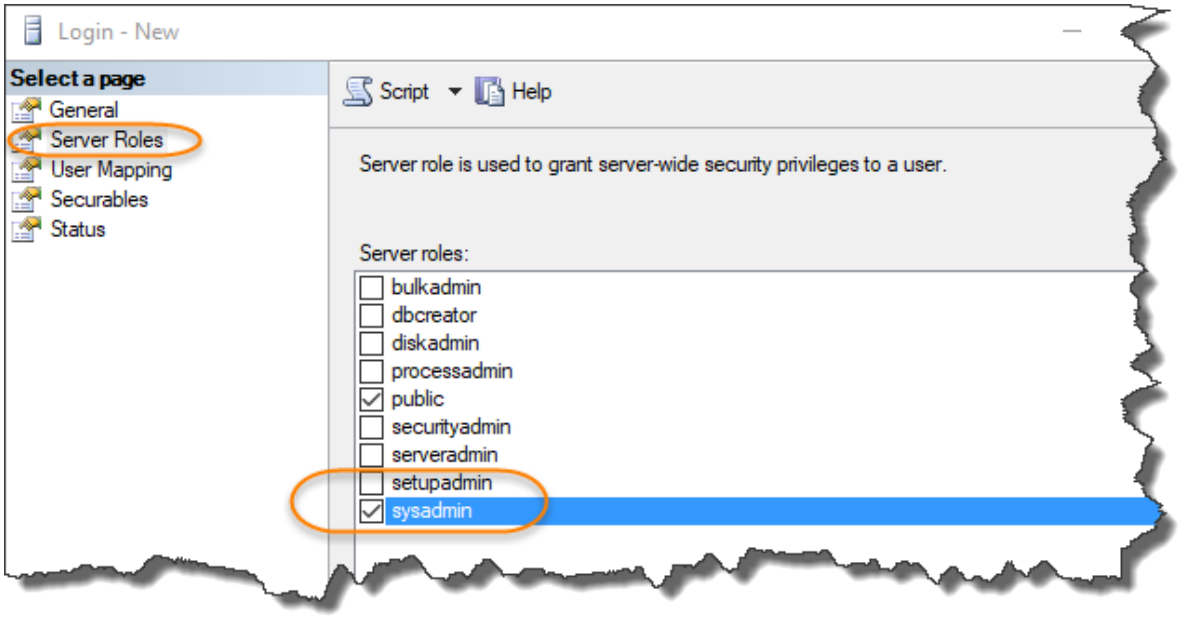
In the "Login – New" window that appears, in the "General" page:

- Give it a Login name
- Select SQL Server authentication
- Type in a password
- Untick "Enforce password policy" (not advised – but again it gets us up and running)



Then on the "Server Roles" page:

- Select the server role that you want



Click OK and your new login will be created – we will use this in our connection string.

Open ***appsettings.json*** and add the following json string to the file:

```
"ConnectionStrings":
 {
  "CommandAPISQLConection":"Server=DESKTOP-H958OBO\\SQLEXPRESS;Initial
Catalog=CommandAPIDB;User ID=cmdAPIDBLogin;Password=ABadPassword1234!;"
 }
```

So your file should look something like this[13]:

Some points to note about the connection string:

[13] If you get errors copying and pasting, check the double quote characters and ensure the connection string value is on one line.

- The "name" of the connection string is: `commandAPISQLConnection`
- The actual connection string is made up of the following components, separated by a semi-colon:
    - `Server`: The name of our SQL Server, note the use of the double back slashes, (first one is used to "escape" the actual back-slash in the string
    - `Initial Catalog`: This is our database (or will be our database – it does not exist yet)
    - `User ID`: The login for our SQL Server (we created this in the last section)
    - `Password`: The password for our login – stored in pain text – not very secure[14]!

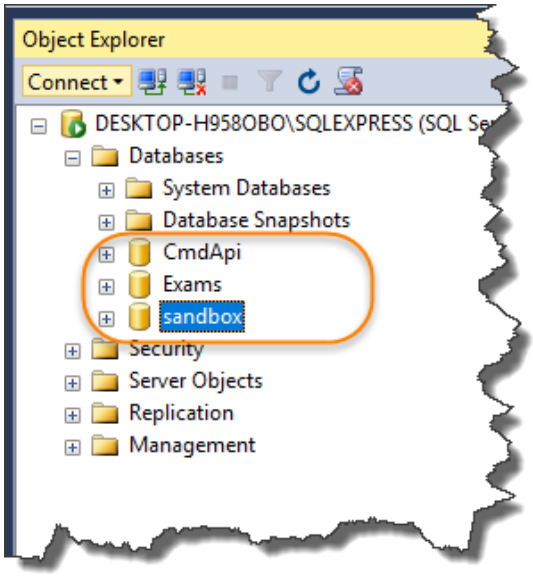
Learning Opportunity: If you've never use Javascript Object Notation, (JSON), before it's basically a way to represent nested key / value pair data as well as array data. Its power lies in its simplicity, I have an YouTube video that takes you through the fundamentals.

If you want to check the validity any json, (including the contents of the entire ***appsettings.json*** file), you can paste you JSON into something like https://jsoneditoronline.org/ which will check the syntax for you.

Where's Our Database?

As mentioned above, we have specified the name of our database in our connection string, (**CommandAPIDB**), but the actual database does not yet exist on our sever, a quick look at the databases in SSMS will confirm that:

[14] We will remedy this in the next chapter



I have 3 other databases here that I have used for other projects, but as yet the **CommandAPIDB** is not there. That is because our database will be created when we perform out first Entity Framework "migration". I explain what this is later in this section.

Revisit the Startup Class

So to recap we have:

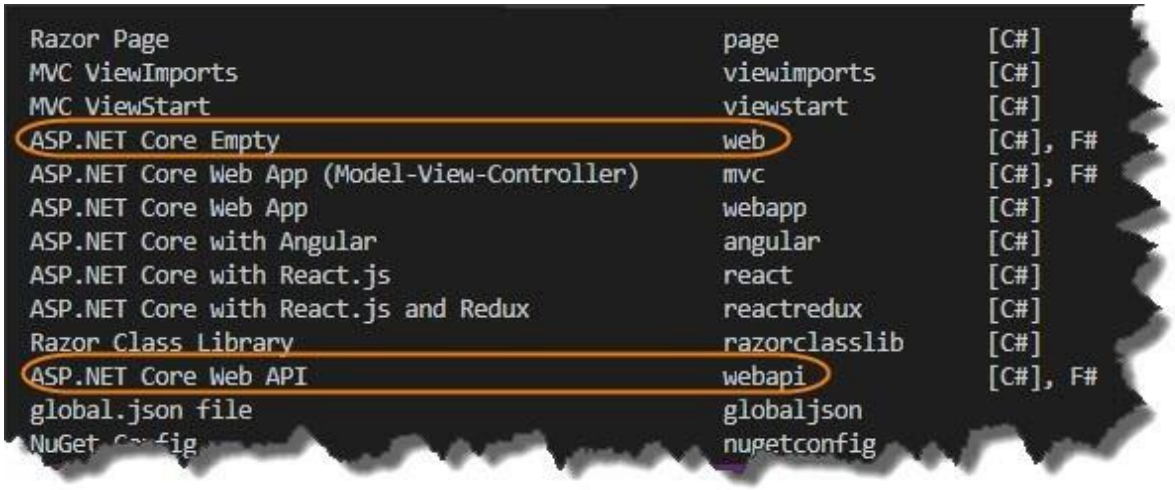
- A Database Server, (but actually no CommandAPIDB "database" as yet!)
- A Model (`Command`)
- DBContext (`CommandContext`)
- DBSet (`CommandItems`)
- Connection String to our database server

The last few things we have to do are:

- Point our `DBContext` class to the connection string (currently it's not aware of it)
- "Register" our `DBContext` class in `Starup->ConfigureServices` so that it can be used throughout our application as a "service".

In order to supply our connection string, (currently in ***appsettings.json***), to our `DBContext` class we have to update our Startup class to provide a "Configuration" object for use, (we use this configuration object to access the connection string).

Side note: Casting your mind back to the start of the tutorial, when we had a choice if project templates?

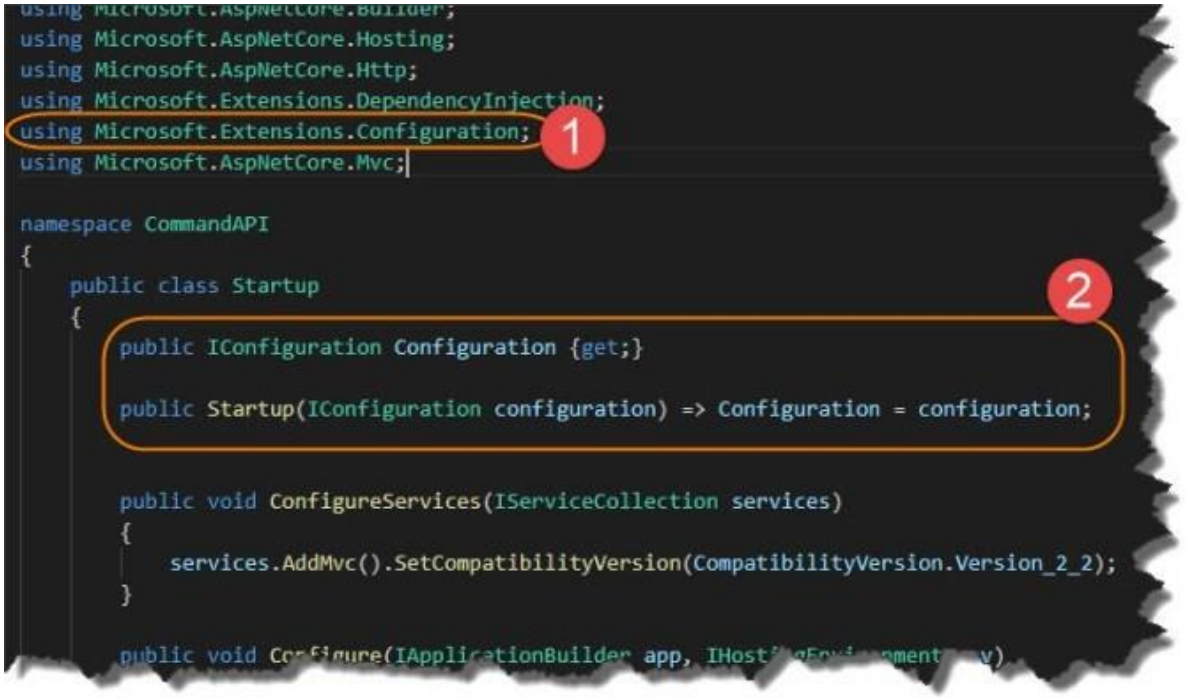


We chose "web" to provide us with an empty shell project. Well if you had chosen "webapi", the "Configuration" code we're about to introduce below, would have been provided as part of that project template. I deliberately choose not to do that so we have to manually add the following code – as I think it will help you learn the core concepts more fully.

Ok so add the following code, (shown in **bold**), to our `startup` class:

```
.
.
.
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Mvc;

namespace CommandAPI
{
    public class Startup
    {
        public IConfiguration Configuration {get;}
        public Startup(IConfiguration configuration) =>
          Configuration = configuration;

        public void ConfigureServices(IServiceCollection services)...
.
.
.
```

I've shown the new sections in context of the whole file below:



1. Add a new using directive: `Microsoft.Extensions.Configuration`
2. Create an `IConfiguration` interface and set up in the constructor

As this code is using Dependency Injection, (a tutorial for another time!), I'm not going to go into more detail of how this works, effectively though it's providing us with the fabric to read config data from ***appsettings.json***, you'll see this next.

For more information on the `IConfiguration` interface the MSDN docs are here.

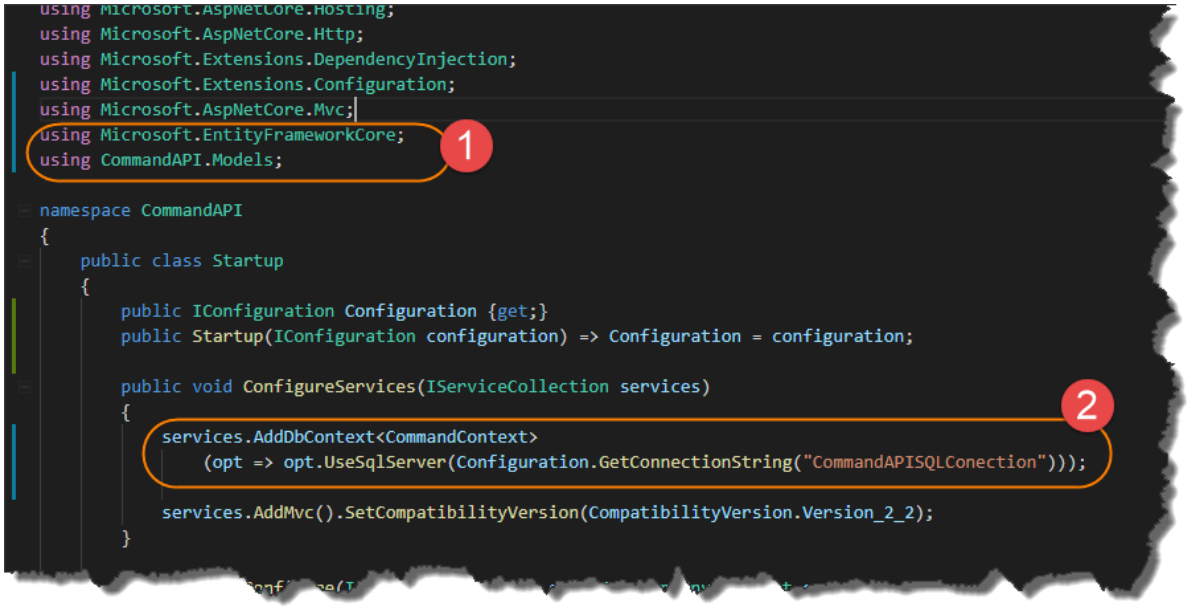
The last thing we have to do is register our `DBContext` in the `ConfigureServices` method and pass it the connection string, (via a configuration interface). So add the following `using` directives to your Startup class:

- `using Microsoft.EntityFrameworkCore;`
- `using CommandAPI.Models;`

And add the following, (**bold**), lines of code to the `ConfigureServices` method in your `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<CommandContext>(opt => opt.UseSqlServer
      (Configuration.GetConnectionString("CommandAPISQLConection")));


services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

To put those changes in context, they are shown below:



1. Our 2 new using directives
2. We register our `CommandContext` class as a solution-wide DBContext and we point it to the connection string, (`CommandAPISQLConnection`), that is contained in our ***appsesstings.json*** file. This is accessed via our `Configuration` object.

Phew! Quite a bit of coding there to wire up everything, we're almost done, but now we need to move on to "migrating" our model from the app to the DB…

Create & Apply Migrations

So we should have everything in place to to create our database and the table containing our Command Objects.

Code First Vs Database First

Just another side note, you may hear about "Code First" and "Database First" approaches when it comes to Entity Framework - in short it speaks to whether:

- We write "code first" then "push" or "migrate" that code to create our database and tables, or:
- We create out Database and tables first and "import" or "generate" code (models), from the DB

Here we are using "code first", (we've already created our command model), so we now have to "migrate" that to our database, we do this via something called, drum roll… Migrations!

So go to your command line, and ensure that you are "in" the API project folder, (***CommandAPI***), and type the following, (hitting enter when you're done):

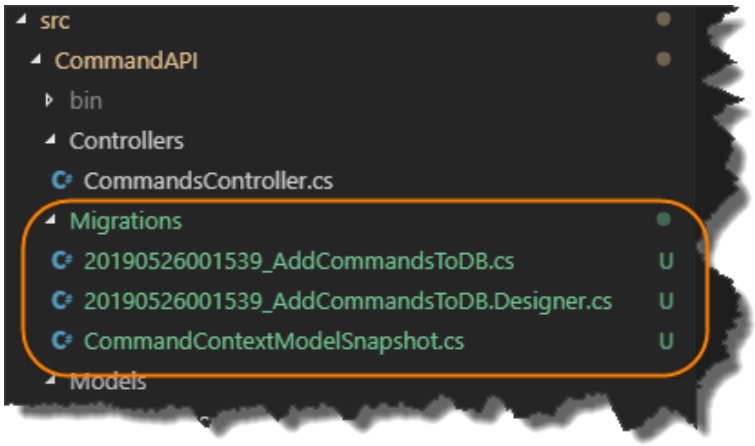
```
dotnet ef migrations add AddCommandsToDB
```

Now all being well a number of magical things should have happened here…

First off, your command line should report something along the lines of:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src\CommandAPI> dotnet ef migrations add AddCommandsToDB
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 2.2.0-rtm-35687 initialized 'CommandContext' using provider 'Microsoft.EntityFrameworkCore.Sql
Done. To undo this action, use 'ef migrations remove'
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\src\CommandAPI>
```

Next you should see an new folder appear in our project structure, called “Migrations”:



Specifically you should make note a new file called: date time stamp + migration name_.cs e.g.:

20190526001539_AddCommandsToDB.cs

It is the contents of this file that when applied to the database will create our new table, (and as it’s the first time our actual database will be created too). A quick look in the file and you’ll see:

```
public partial class AddComm  ToDB : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "CommandItems",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn),
                Howto = table.Column<string>(nullable: true),
                Platform = table.Column<string>(nullable: true),
                Commandline = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_CommandItems", x => x.Id);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "CommandItems");
    }
}
```

1. An “Up” method. Basically this method is called to create new stuff
2. A “Down” method. Used to roll back the changes made in the Up method
3. The creation of a table and it’s columns, noticing the nature of the “Id” column

Entity Framework is a huge area so I’m not going to go into any more detail than that here.

Note: at this stage we still do not have the **CommandAPIDB** database created, that comes next…

Finally all that’s left to is “update the database” to apply our changes to do this type:

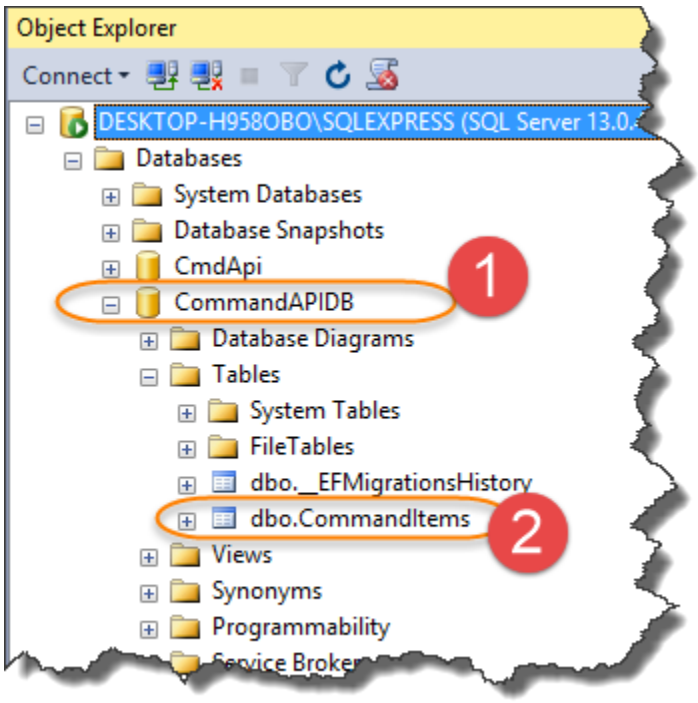
```
dotnet ef database update
```

Our migration is run, as reflected in the following output:

A number of things happen here:

1. Our database update command
2. Our database (**CommandAPIDB**), is created as it did not yet exist on the target server
3. A table called _EFMigrationsHistory is created, this just stores the ID’s of the migrations that have been run and allows Entity Framework to both roll back migrations to a certain point or correctly run migrations on a new end-point server, (and hence recreating the database).
4. Our CommandItems table is created which is the persistent equivalent of our `Command` model(s).

If we also take a look at our SQL Server instance this is reflected by the fact we have both our **CommandAPIDB** database and our **CommandItems** table:



1. CommandAPIDB database
2. CommandItems table

Add Some Data

Just to close the loop on this rather long chapter, we're going to:

- Add some data to our database
- Update our single API Action to return that data, (as opposed to the existing hard-coded string)

You can add data a number of ways, (including fully scripting this to import a lot of test data – we're not covering that today), but by far the simplest and most ubiquitous way to do that is via an `SQL INSERT` command that we can run from inside the SSMS query window, or from the SQLCMD interactive prompt.

Learning Opportunity: As I've already shown you how to execute queries using SQLCMD, I won't detail those steps again here. The SQL query I use as part of the SSMS steps below will work with SQLCMD, so the learning opportunity is to take that and run it in SQLCMD.

In terms of the data we should put in, I'd like to circle back to the creation and updating of the DB above, we used the following 2 commands:

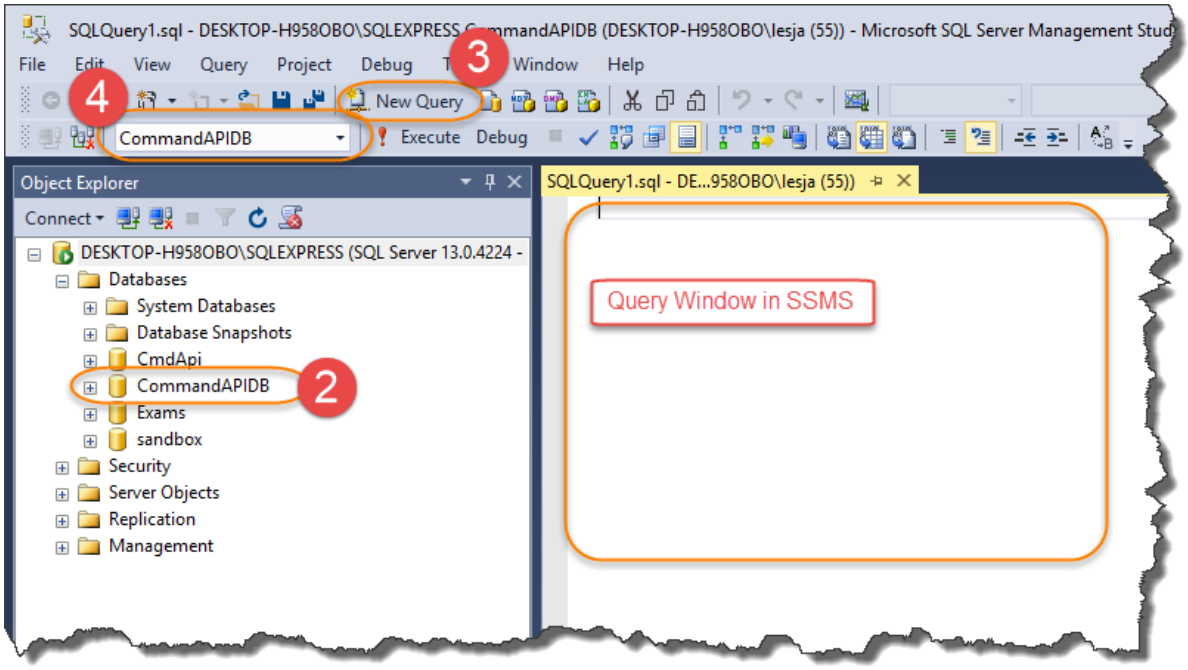
- dotnet ef migrations add
- dotnet ef database update

Therefore if we wanted to store this data in our table we'd add the following data:

ID[15]	HowTo	Platform	CommandLine
1	Create an EF Migration	Entity Framework Core Command Line	dotnet ef migrations add
2	Apply Migrations to DB	Entity Framework Core Command Line	dotnet ef database update

To add this data via a SQL INSERT command in SSMS:

1. Open SSMS & connect to the server
2. Expand Databases & select our **CommandAPIDB** database
3. Click on "New Query" to open, (surprise-surprise) a query window
4. Ensure that our **CommandAPIDB** is "active"



In the query window type the following SQL to insert both of our command line snippets into the database:

```
USE CommandAPIDB;

INSERT INTO CommandItems (HowTo, [Platform], Commandline)
VALUES ('Create an EF migration', 'Entity Framework Core Command Line', 'dotnet ef
migrations add' );

INSERT INTO CommandItems (HowTo, [Platform], Commandline)
VALUES ('Apply Migrations to DB', 'Entity Framework Core Command Line', 'dotnet ef
database update' );
```

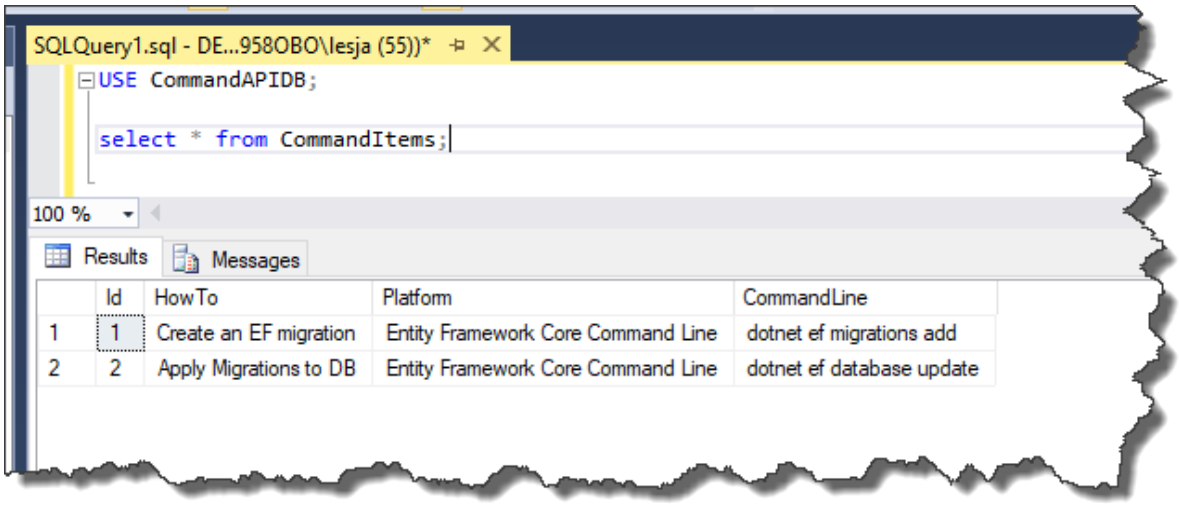
Note: as "Platform" is a reserved key work in SQL Server, we encase it in [square brackets] to fully escape it.

[15] You do not need to provide a value for ID when you add data to the database, this is auto-created by SQL Server for us.

After that hit “F5”, or click “Execute” to run the SQL – this should insert the lines into our database. To check this, clear you the SQL from the window, (otherwise if you execute it again it’ll insert 2 more rows, effectively duplicating the data), and type:

```
select * from CommandItems;
```

This should return something like:



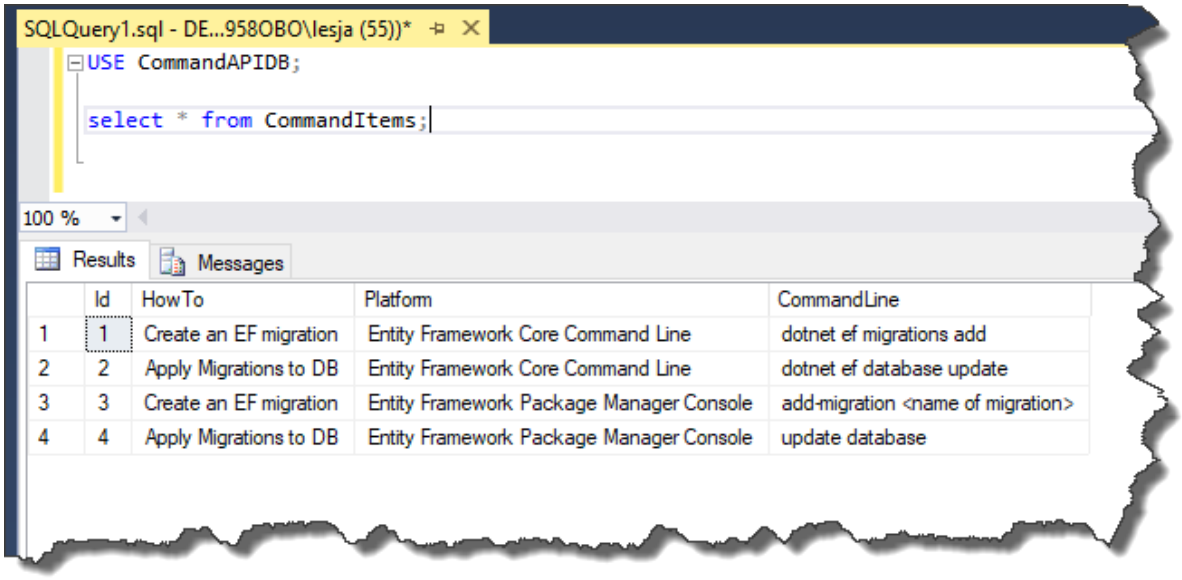
If you read my [blog post on Entity Framework], you’ll have noticed by now that the commands used in that tutorial are different to those used here. That’s because in that tutorial, we’re using the “Package Manager Console” in Visual Studio to issue commands for Entity Framework, (not Entity Framework Core / and the .Net Core Command line) – quite confusing I know!

I think therefore just to labour that point let’s add 2 new command line prompts in our DB:

HowTo	Platform	CommandLine
Create an EF Migration	Entity Framework Package Manager Console	add-migration <name of migration>
Apply Migrations to DB	Entity Framework Package Manager Console	update database

Learning Opportunity: You’ll need to write the SQL to insert these additional command snippets to our DB!

After executing the `SQL INSERT` commands, perform another `SELECT “all”`, (i.e. `SELECT *` …), and you should see:



	Id	HowTo	Platform	CommandLine
1	1	Create an EF migration	Entity Framework Core Command Line	dotnet ef migrations add
2	2	Apply Migrations to DB	Entity Framework Core Command Line	dotnet ef database update
3	3	Create an EF migration	Entity Framework Package Manager Console	add-migration <name of migration>
4	4	Apply Migrations to DB	Entity Framework Package Manager Console	update database

Hopefully you can see that as you build out the data in our table that this API will become useful, if like me, your memory is not as good as it once was!

To round out this chapter let's update our existing API Action to return this data!

Return "Real" Data

Ok , it's been a while so this is where we left our controller:

```
1   using System.Collections.Generic;
2   using Microsoft.AspNetCore.Mvc;
3
4   namespace CommandAPI.Controllers
5   {
6       [Route("api/[controller]")]
7       [ApiController]
8       public class CommandsController : ControllerBase
9       {
10          [HttpGet]
11          public ActionResult<IEnumerable<string>> Get()
12          {
13              return new string[] {"this", "is", "hard", "coded"};
14          }
15      }
16  }
```

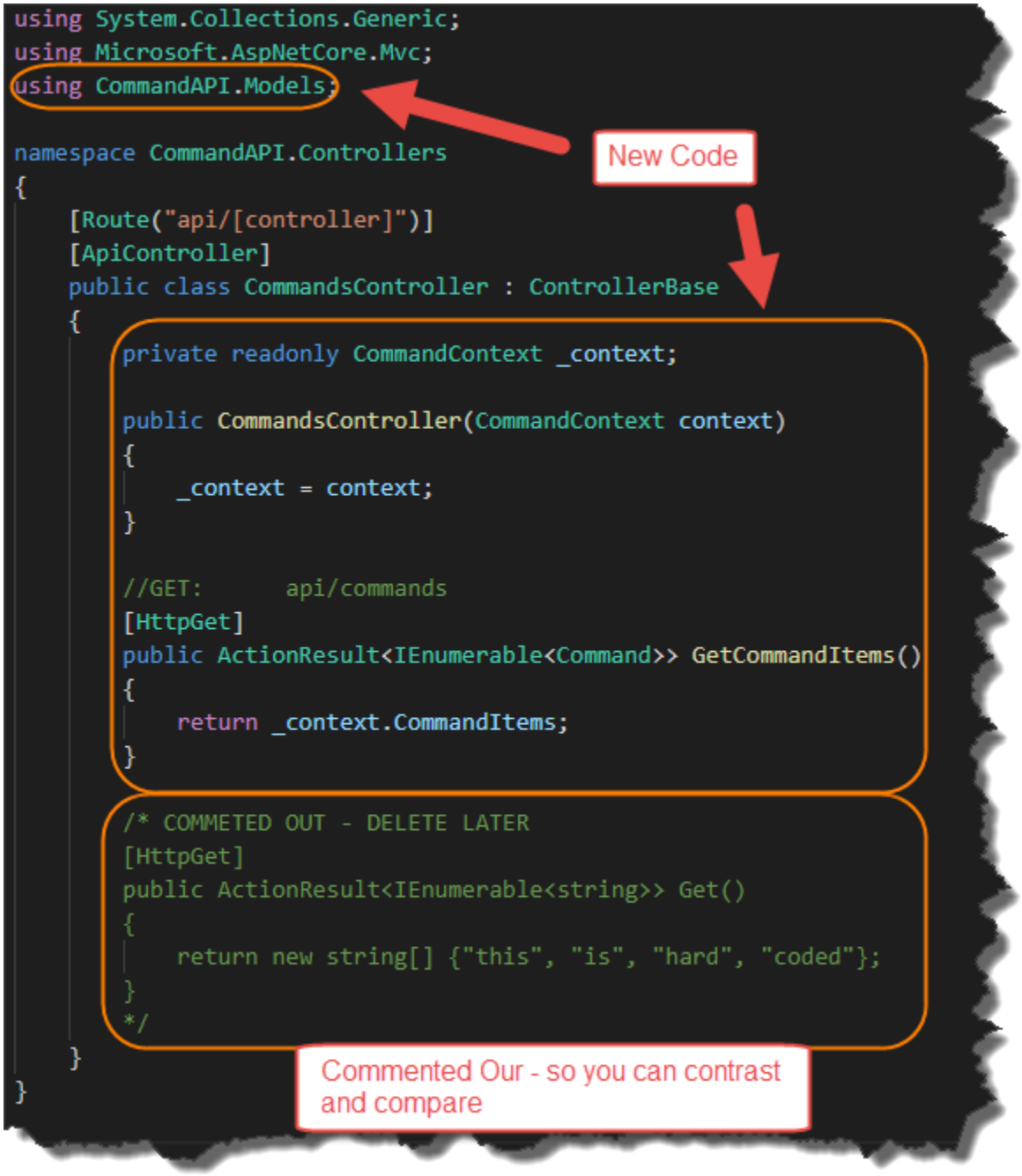
We had 1 HttpGet Action that returned a hard-coded string enumeration. Let's alter this action so that it uses our DBContext class, (`CommandContext`), to go to the database and pull back our "live" data!

Add the following lines to you controller (highlighted in bold):

```
using System.Collections.Generic;
```

```
using Microsoft.AspNetCore.Mvc;
using CommandAPI.Models;

namespace CommandAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CommandsController : ControllerBase
    {
        private readonly CommandContext _context;

        public CommandsController(CommandContext context)
        {
            _context = context;
        }

        //GET:      api/commands
        [HttpGet]
        public ActionResult<IEnumerable<Command>> GetCommandItems()
        {
            return _context.CommandItems;
        }
.
.
.
```

To put the changes in context, and to ensure that you 'comment out' our old Action Result, here's what your code should look like:



So what's going on? The first important change is the fact that we create a private instance of a `CommandContext` class (`_context`). We then create a constructor for our controller and using *Constructor Dependency Injection* we assign an instance of our DBContext to our private instance:

I appreciate this can be confusing, (i get tied up in knots with this stuff), so it's why it's important to understand the role of the `Startup` class and specifically the `ConfigureServices` method where the registration of services occurs.

In short, our private `_context` instance is a representation of the `CommandContext` class that we can then use to access the database.

The next code addition, is somewhat more straightforward, I've deliberately commented out our original, hard-coded action to see how similar they are:

```
//GET:      api/commands
[HttpGet]
public ActionResult<IEnumerable<Command>> GetCommandItems()
{
    return _context.CommandItems;
}

/* public ActionResult<IEnumerable<string>> Get()
{
    return new string[] {"this", "is", "hard", "coded"};
}
*/
```

Instead of expecting a return type of string, we're now expecting a return type of `Command`. I've also changed the name of the method from Get to `GetCommandItems`.

Finally, use our DB context to return the contents of our DbSet: `CommandItems` (essentially an enumeration of `Commands`).

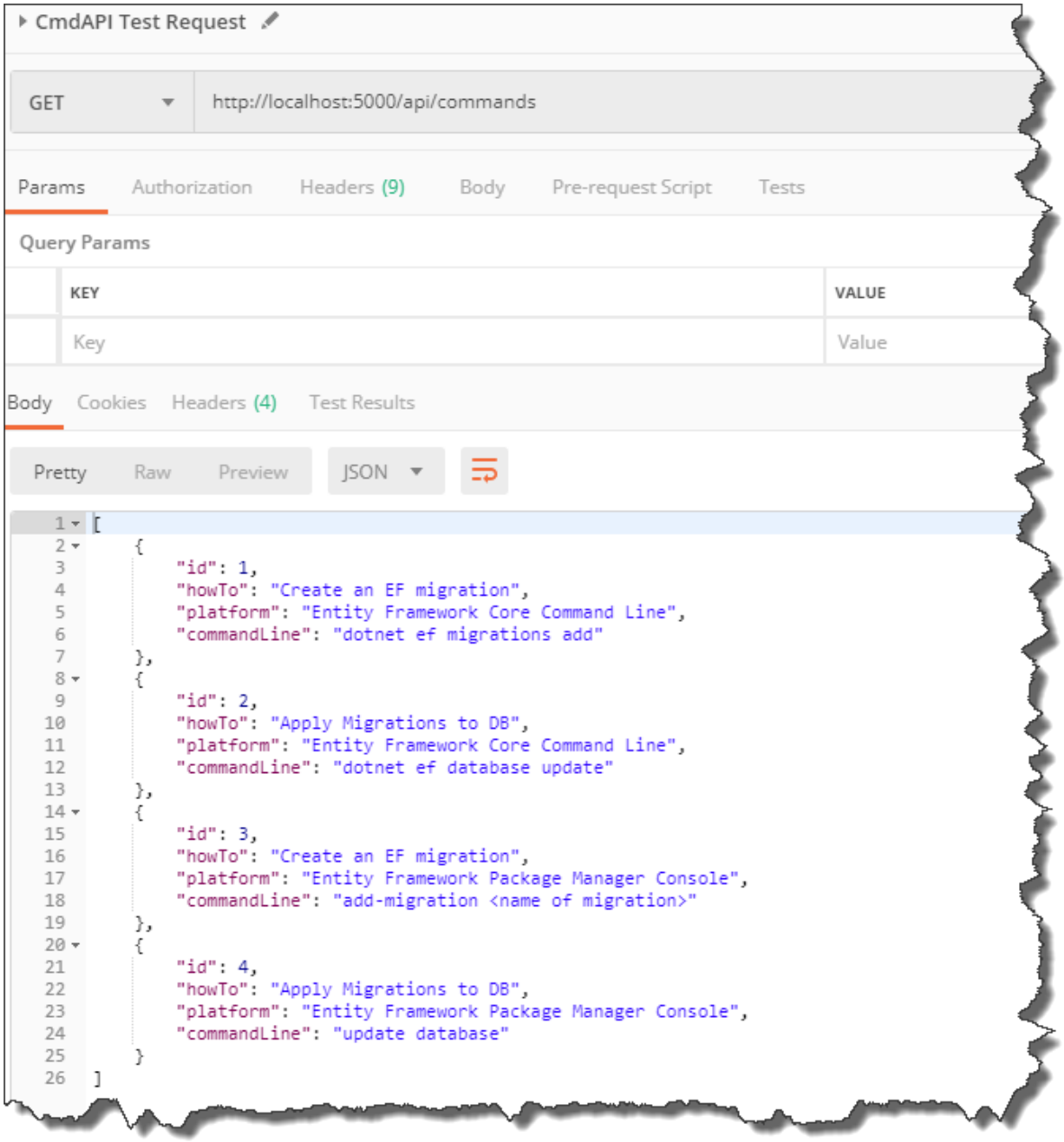
Let's save the file, and build our project to test for errors:

```
dotnet build
```

Assuming all is well lets run:

```
dotnet run
```

And finally trigger a call via Postman, (exactly the same way as before):



Celebration Check Point: Possibly the most significant celebration in the whole book – well done! You've basically build a data-drive API in .NET Core!

We've covered a lot of material in this chapter. To be honest I was going to try and make it smaller, but then I felt the flow would not be a good.

Wrapping Up The Chapter

As we have our code under source control, we want to:

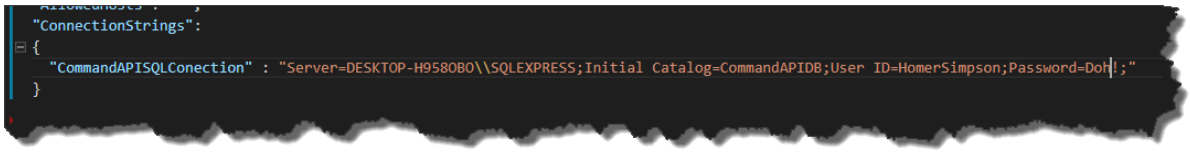
- Add untracked, (aka 'new') files to source control / Git
- Commit those changes
- Push our code up to GitHub **[WARNING before you do this!!!!]**

Why am I warning you about pushing our code up to our public GitHub repository?

That's right we have placed the user login and password to our database in the **appsettings.json** file – this will become publicly available if we push our code…

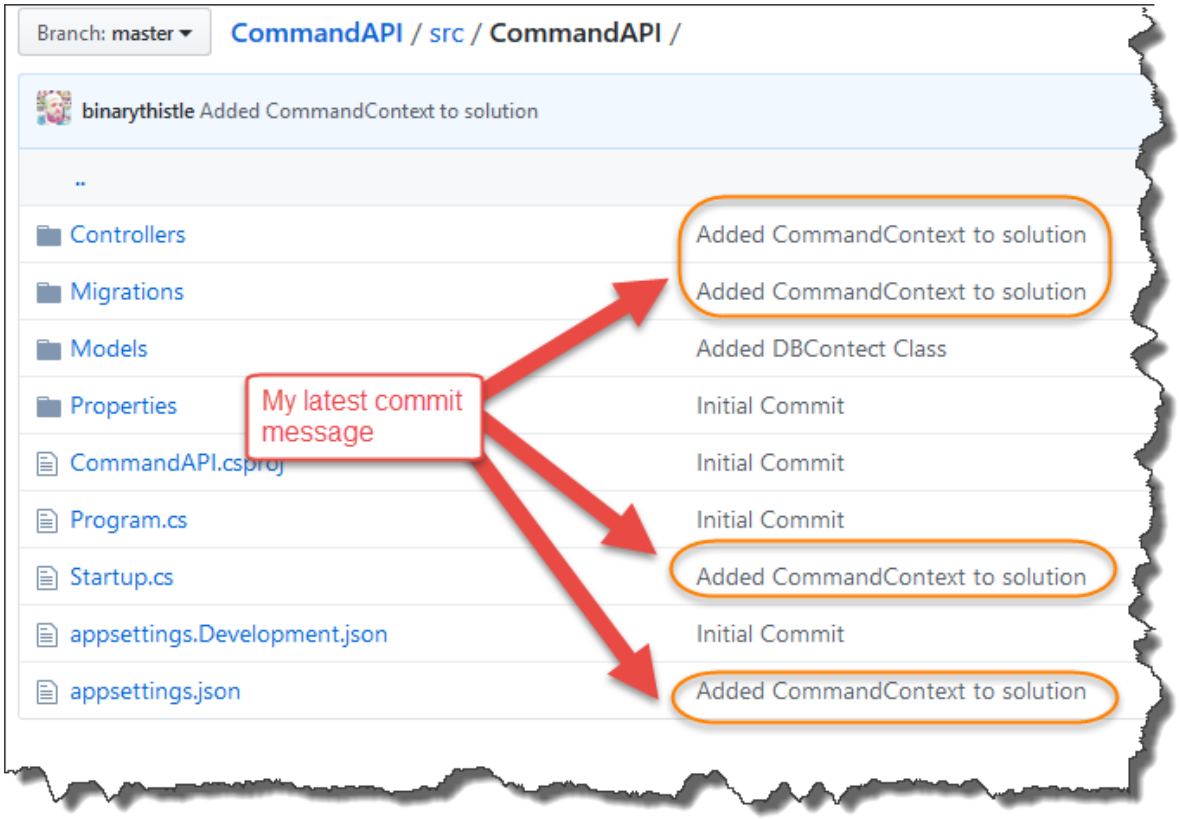
Redact Our Login and Password

Ok if your API is still running, stop it: (Ctrl + C), and edit the connection string in your ***appsettings.json*** file, redacting or changing the values for User ID and Password to something non-sensical, (note if you run the API again it will fail when we come to retrieve data!), e.g.:

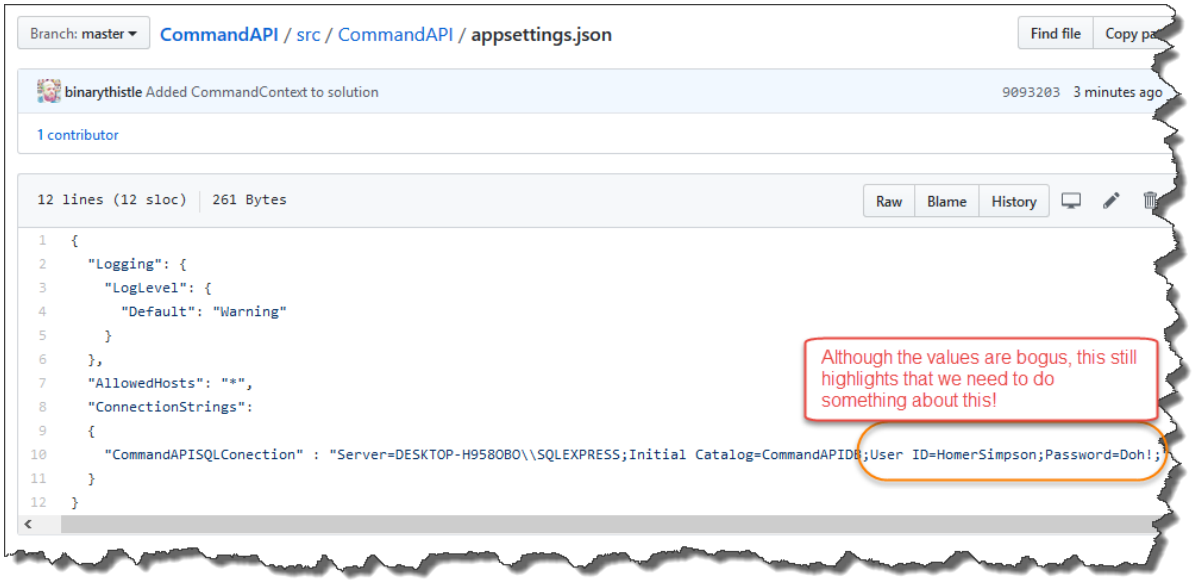


Save the file then, perform the 3 steps to: Add/Track, Commit and Push your code to GitHub.

Go over to GitHub, and look at the ***appsetting.json*** file there:



The ***appsettings.json*** file as it exists publicly on GitHub:



We have 2 major problems now:

- It's terribly insecure (even if we have put in temporary "fake" values)
- Our code does not work now! (We'll get authentication errors)

Clearly this we can't publish user id's and password to GitHub, even if we made the GitHub repository private, this is still terrible practice. We need away of keeping these details secret…

CHAPTER 7 – ENVIRONMENT VARIABLES & USER SECRETS

CHAPTER SUMMARY

In this chapter we discuss what runtime environments are and how to configure them, we'll then discuss what user secrets are and how to use them.

WHEN DONE, YOU WILL

- Understand what runtime environments are
- How to set them via the `ASPNETCORE_ENVIRONMENT` variable
- Understand the role of ***launchSettings.json*** and ***appsettings.json*** files
- What *user secrets* are
- How to use user secrets to solve the problem we had at the end of the last chapter.

ENVIRONMENTS

When developing anything, you typically want the freedom to try new code, refactor existing code, and basically feel free to fail without impacting the end user. Imagine if you had to make code changes directly to a live customer environment? That would be:

- Stressful for you as a developer
- Showing great irresponsibility as an application owner
- Potentially impactful to the end user

Therefore to avoid such a scenario, most, if not all organisations will have some kind of "Development" environment where developers can roam free and go for it, without fear of screwing up.

Les's Personal Anecdote: If you've ever worked as part of a development team you'll know the above statement is not quite true… Yes you can break things in the development environment without fear of impacting customers, but if you break the build you will have the wrath of the other members of your team to deal with!

I know this from bitter experience…

Anyway, you'll almost always have a "Development" environment, but what other "environments" can you have? Well, jumping to the other end of the spectrum, you'll always have a "Production" environment. This is where the live, "production", code sits and runs as the actual application, be it a customer facing web site, or in our case an API available for use by other applications.

You will typically never make code changes directly in production, indeed deployments and changes to production should be done, where possible, in as automated a way as possible, where the "human hand" doesn't intervene.

So are they the only 2 environments you can have? Of course not, and this is where you'll find the most differences in the real world. Most usually you will have some kind of "intermediate" environment, (or environments!), that sits in between Development and Production, it's primary use is to "stage" the build in as close to a Production environment as possible to allow for integration and even user testing. Names for this this environment vary, but you'll hear Microsoft refer to it as the "Staging" environment, I've also heard it called PR or "Production Replica".

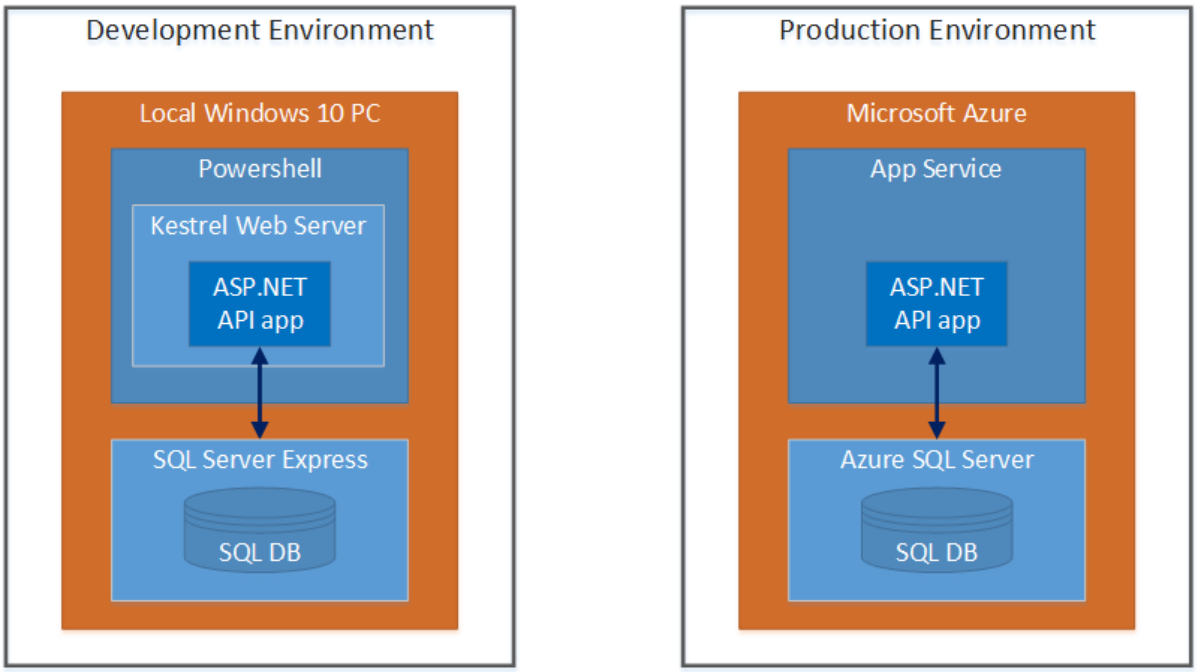
Les's Personal Anecdote: Replicating a Production environment accurately can be tricky, (and expensive), especially if you work in a large corporate environment with lots of "legacy" systems that are maintained by different 3rd party vendors – coordinating this can be a nightmare.

There are of course ways to simulate these legacy systems, but again, there is really no substitute for the real thing… If you're not simulating the legacy systems *your app* is interacting with precisely, that's when you find those lovely bugs in production.

I remember being caught out with SQL case-sensitivity on an Oracle DB while on site at a customer deployment. An easy fix when I realised the issue, but something as simple as that can be stressful and also damaging to your own reputation!

Our Environment Set Up

We are going to dispense with the Staging or Production Replica environment and use only: Development and Production – this is more than sufficient to demonstrate the necessary concepts we need to cover. Refer to the diagram below to see my environmental set up, (yours should mirror this to a large extent):



As you can see the "components" that are there are effectively the same, it's really only the underlying platform that is different, (a local Windows PC Vs Microsoft Azure).

We'll park further discussion on the Production Environment for now and come back to that in later chapters, for now we'll focus on our Development environment.

The Development Environment

How does our app know which environment it's in? Quite simply – we tell it!

This is where "Environment Variables" come into play, specifically the ASPNETCORE_ENVIRONMENT variable. Environment variables can be specified, or set, in a number of different ways depending on the physical environment, (Windows, OSX, Linux, Azure etc.). So while they can be set at the OS level, our discussion will focus setting them in the ***launchSettings.json*** file, for now…

Environment variables set in the ***launchSettings.json*** file will override environment variables set at the OS layer, that is why for the purposes of our discussion we'll just focus on setting out values in the ***launchSettings.json*** file.

A fuller discussion on this can be found on here.

Opening the ***launchSettings.json*** file in the API project, you should see something similar to the following:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:50677",
      "sslPort": 44331
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "CommandAPI": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

When you issue `dotnet run` the first *profile* with `"commandName" : "Project"` is used. The value of `commandName` specifies the webserver to launch. `commandName` can be any one of the following:

- `IISExpress`
- `IIS`
- `Project` (which launches the Kestrel web server)

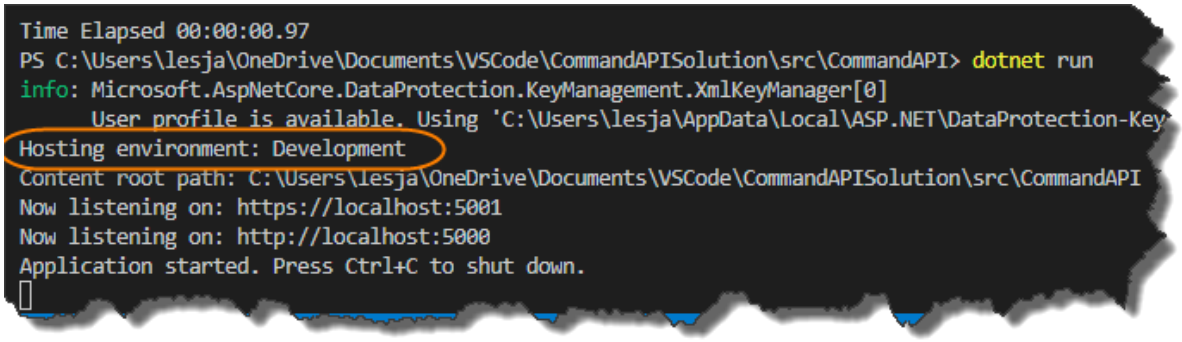
In the highlighted profile section above there are also additional details that are specified including the "`applicationUrl`" for both http and https and well as our `environmentVariables`, in this instance we only have one: `ASPNETCORE_ENVIRONMENT`, set to: `Development`.

So when an application is launched, (via `dotnet run`):

- ***launchSettings.json*** is read (if available)

- `environmentVariables` settings override system / OS defined environment variables
- The hosting environment is displayed

E.g:



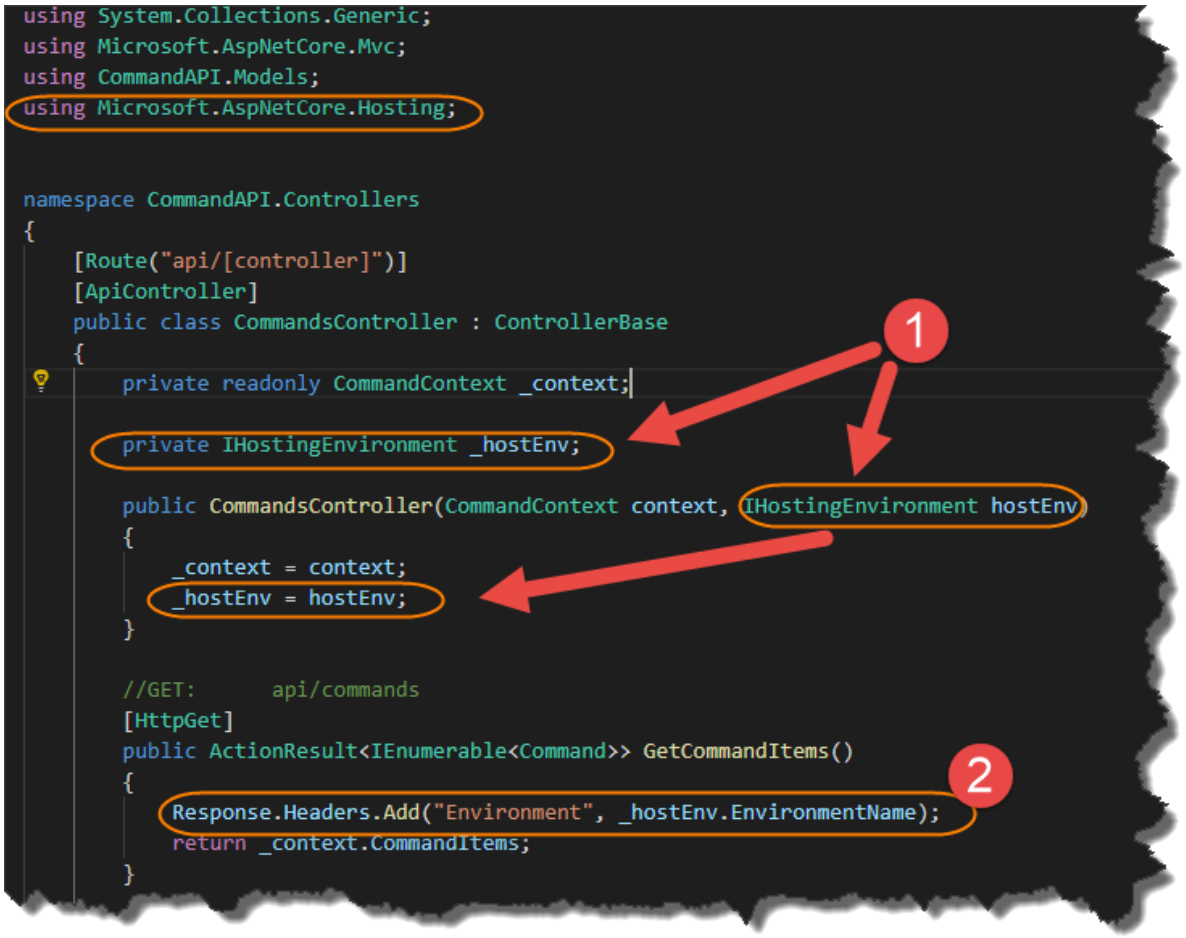
Now I want to take this a little further and pass back the Hosting Environment as an additional header in our API response, so to do this:

- Revert your `CommandAPISQLConnection` connection string in ***appsettings.json*** to use the correct User ID and Password values, (remember we redacted/changed these in the last chapter).

Open the ***CommandsController.cs*** file and add the **highlighted code** as shown below:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using CommandAPI.Models;
using Microsoft.AspNetCore.Hosting;


namespace CommandAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CommandsController : ControllerBase
    {
        private readonly CommandContext _context;

        private IHostingEnvironment _hostEnv;

        public CommandsController(CommandContext context,
                                  IHostingEnvironment hostEnv)
        {
            _context = context;
            _hostEnv = hostEnv;
        }

        //GET:      api/commands
        [HttpGet]
        public ActionResult<IEnumerable<Command>> GetCommandItems()
        {
            Response.Headers.Add("Environment",
                                 _hostEnv.EnvironmentName);
            return _context.CommandItems;
        }
.
.
.
```

For clarity I've highlighted the additional code below:

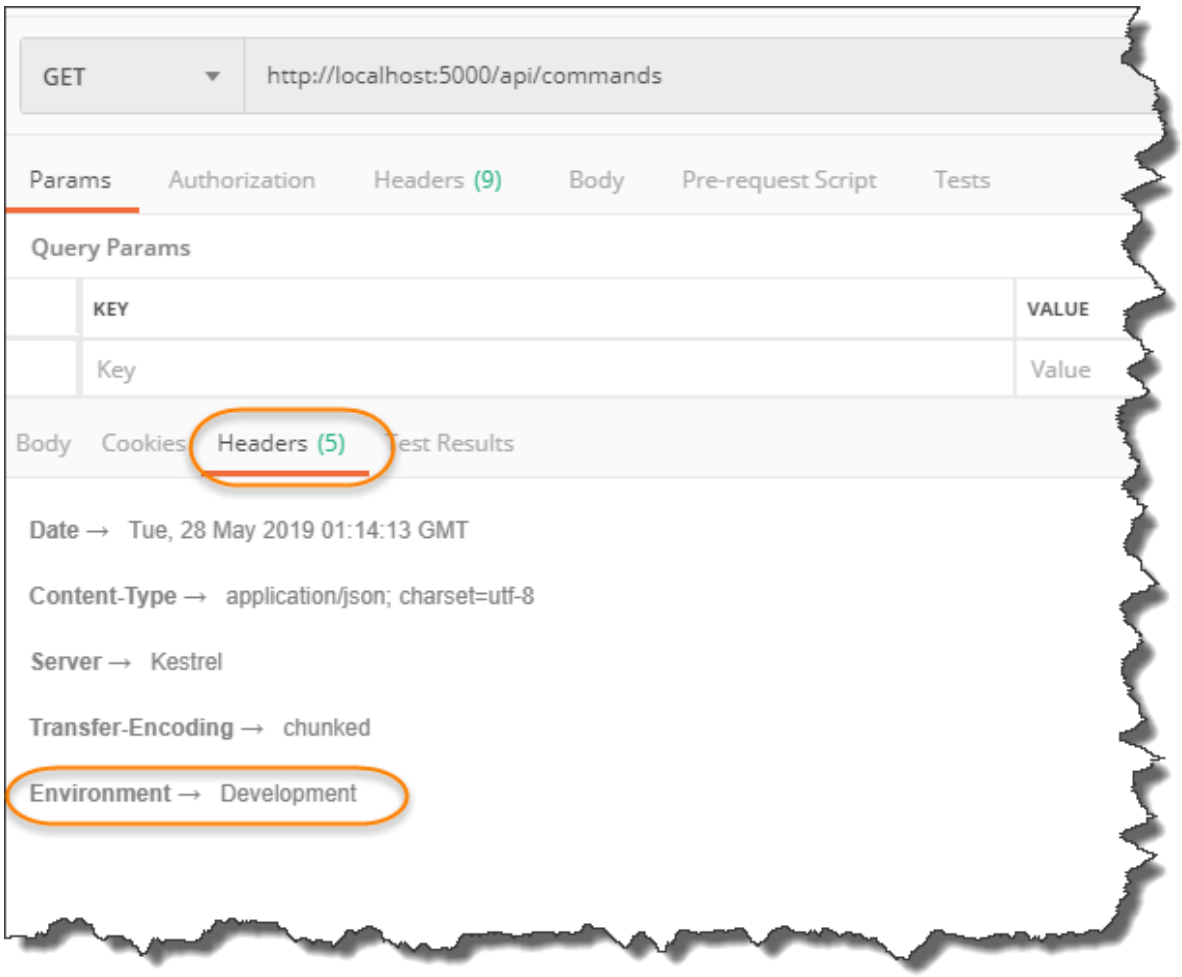


You will note:

1. We use a similar *constructor dependency injection* pattern to "inject" a `IHostingEnvironment` member into our Controller constructor for use later, (see item 2).
2. We make use of `_hostEnv` To retrieve the `EnvironmentName` and add it to our API `Response.Headers`

Save the file then:

- `dotnet build`
- `dotnet run`
- Fire up Postman and perform another GET request as before:



Ensure the "Headers" tab is selected in the response and you'll see our additional header containing the value of our Environment – this will come in useful later when we eventually deploy to Azure.

So What?

At this stage I hear you all saying: *"Yeah that's great and everything, but so what?"*

Good Question, I'm glad you asked that question[16]!

Looking back at our simple environment set up, we need to connect to our Development database and eventually our Production database, and an in almost all instances they will be different, with different:

- End Points, (e.g. Server Name / IP address etc)
- Different Login Credentials, etc.

Therefore depending on our *environment*, we'll want to change our *configuration*.

I'm using the database connection string as an example here, but there are many other configurations that will change depending on the environment. That is why it is so important we are aware of our environment.

Make the Distinction

[16] Beware when you get this response from either Salesman, an Executive or Politician – it usually means that don't know the answer and will either deflect the question somewhere else, or lay on some major bull sh!t.

Ok, so what approach should you take within your application to make determinations on configuration based on the development environment, (e.g. *use this* connection string for Development and *this one* for Production), well there are a number of different answers to that, to my mind there are 2 broad approaches:

1. "Manually" determine the environment in your code and take the necessary action
2. Leverage the power & behaviour of the .Net Core *Configuration API*

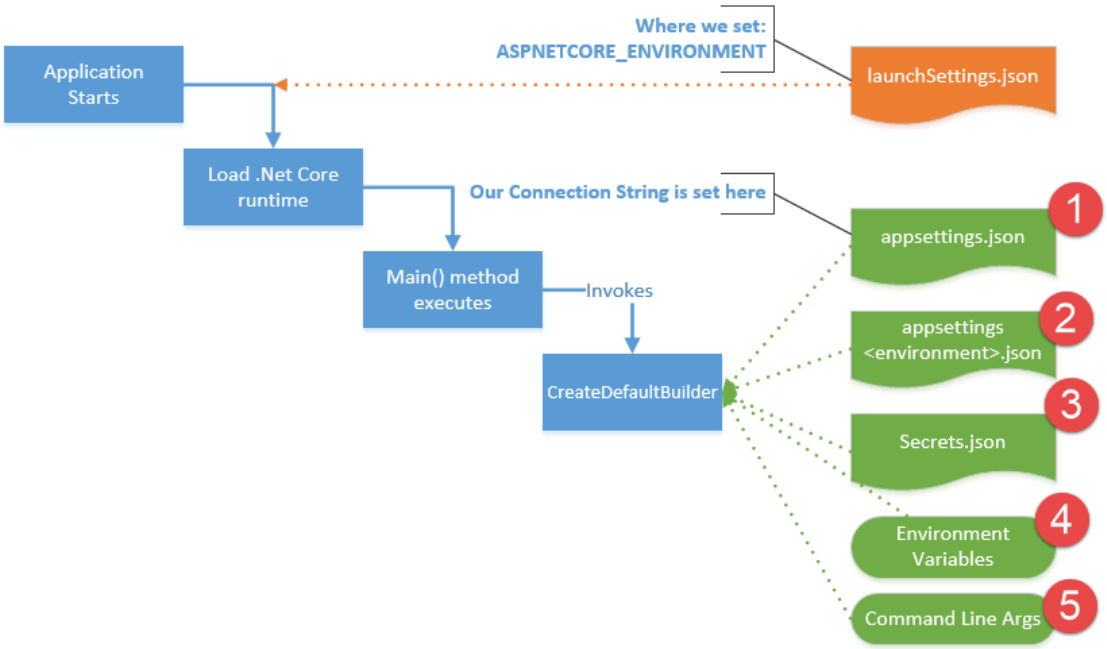
We're going to go with Option 2. While Option 1 is a possibility, (indeed this pattern is used in many of the default .Net Core Projects – see example below), I personally prefer to decouple code from configuration where possible, although it's not always possible – that is why we'll go with Option 2.

The above snippet is taken from our very own `Startup` class, where the default project template uses the `IsDevelopment` parameter to determine which exception page to use.

Order of Precedence

Ok so we're going to leverage from the behaviour of the .Net Core *Configuration API* to change the config as required for our 2 different environments.

Let's quickly revisit the `Program` Class start-up sequence for our app as covered in Chapter 4:



You'll see I've added some extra detail:

- The ***launchSettings.json*** file is loaded when we issue the `dotnet run` command and sets the value for ASPNETCORE_ENVIRONMENT
- A number of configuration sources that are used by the `CreateDefaultBuilder` method.
- By default these sources are loaded in the precedence order specified above, so ***appsettings.json*** is loaded first, followed by ***appsettings.Development.json*** and so on…

It is really important to note here that: **The Last Key Loaded Wins.**

What this means, (and we'll demonstrate this below), is that if we have a 2 configuration items with the same name, e.g. our connection string, `CommandAPISQLConection`, that appears in different configuration sources, e.g. ***appsettings.json*** and ***appsettings.Development.json***, the value contained in ***appsettings.Development.json*** will be used.

So you'll notice here that *Environment Variables* will take precedence over the values in ***appsettings.json***. This is the *opposite* of how this works when we talk about ***launchSettings.json***, above. As previously mentioned the contents of ***launchSettings.json*** take precedence *over* our system defined environment variables…

So be careful!

There's a [great article here] with more information on this.

It's Time to Move

Ok let's put a bit of this theory into practice, and demonstrate what we mean.

- Go into your ***appsettings.json*** file and *copy* the `ConnectionStrings` key-value pair that contains our `CommandAPISQLConnection` connection string
- Make sure you have the correct values for User ID and Password
- Insert this json segment into the ***appsettings.Development.json*** file – see below

This means we will have the *same configuration* element in *both*: ***appsettings.json*** and ***appsettings.Development.json***.

Again if you're unsure that your json is well-formed, use something like http://jsoneditoronline.org/ to check.

Save the files you've made any changes to, run your API and make the same call – it all still works as usual.

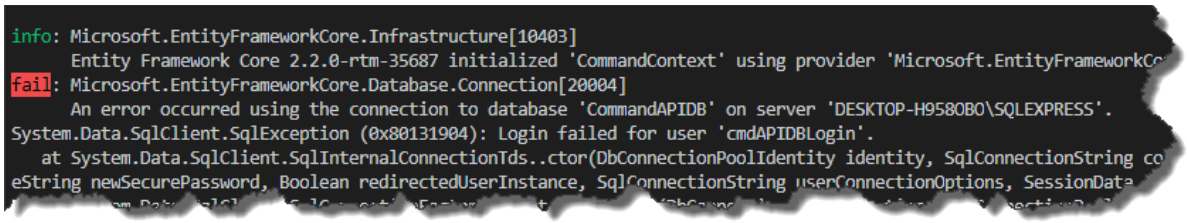
Let's Break It

Ok so to prove the point we were making above.

- Stop your API from running (Ctrl +c)
- Go back into ***appsettings.Development.json*** file and edit the `Password` parameter in the connection string, so that authentication to the SQL Server will fail – see example below
- Save your file

Ok now run the app again, and try to make the API Call…

Looking at the terminal output you'll see you get a database connection error, this is because the last value for our connection string was invalid.



Fix It Up

Ok so let's fix this.

- Edit your ***appsettings.Development.json*** file and correct the value for the `Password` parameter
- *Delete* the `ConnectionStrings` json from the ***appsettings.json*** file

This means that *only* our ***appsettings.Development.json*** file now contains our connection string, your ***appsettings.json*** file should now look like:

This means that currently we only have a valid source for our connection string when running in a Development environment.

Learning Opportunity: What will happen if you edit the ***launchSettings.json*** file and change the value of `ASPNETCORE_ENVIRONMENT` to "Production"?

Do this, run your app and explain why you get this result.

We will cover our Production connection string in the Chapter 10 – Deploying to Azure

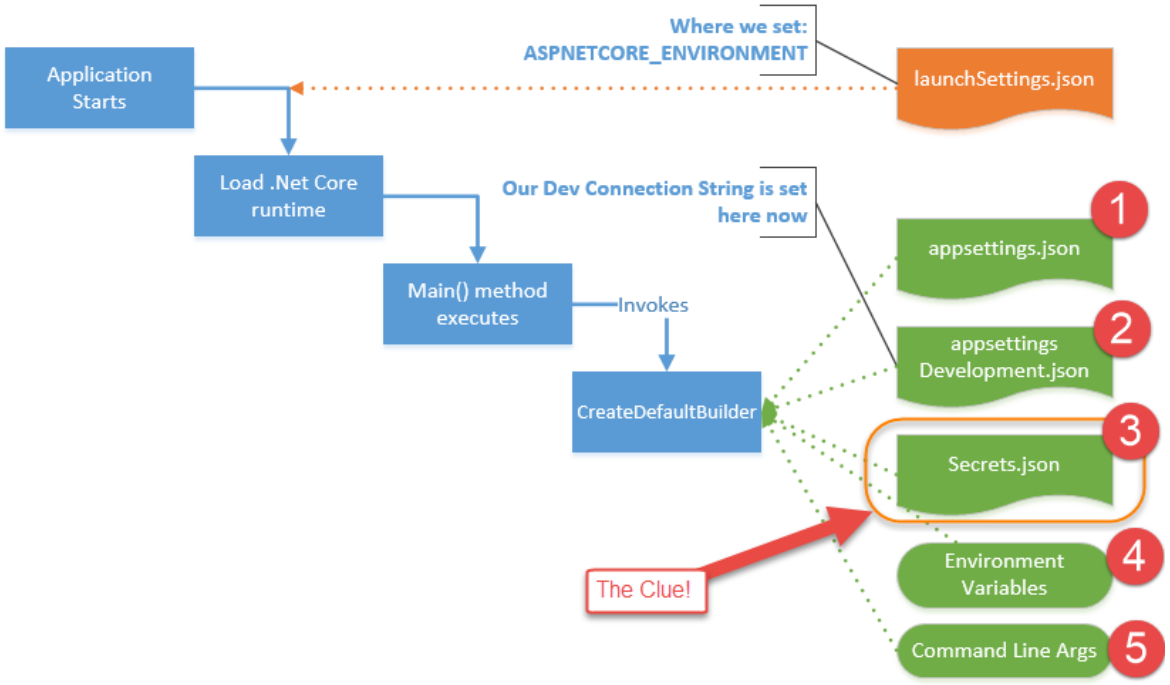
User Secrets

So we've covered the different environments you can have, why you have them, and have even reconfigured our app to have a *development environment only* connection string. But we still have not solved the issue we were left with at the end of the previous chapter- that being that our User ID and Password are still in plain-text and are therefore available to anyone who has access to our source code – e.g. someone looking at our repo in GitHub.

We solve that here.

What Are User Secrets?

Well I gave you a bit of a clue in this chapter already:



In short they are another location where you can store configuration elements, some points to note:

- User Secrets are "tied" to the individual developer
- They are abstracted away from our source code and are not checked into any code repository
- They are stored in the "***secrets.json***" file
- The ***secrets.json*** file is *unencrypted* but is stored in a file-system protected user profile folder on the local dev machine.

In short this means that individual users can store, (amongst other things), the credentials that they use to connect to a database. As the file is secured by the local file system, they remain secure, (assuming no one has login access to your PC).

In terms of what you can store, this can be anything, it's just string data. We're now going to set up User Secrets for our *development connection* string.

Setting Up User Secrets

We need to make use of something called *The Secret Manager Tool* in order to make use of user secrets, this tool works on a project by project basis and therefore needs a way to uniquely identify each project. For this we need to make use of GUID's…

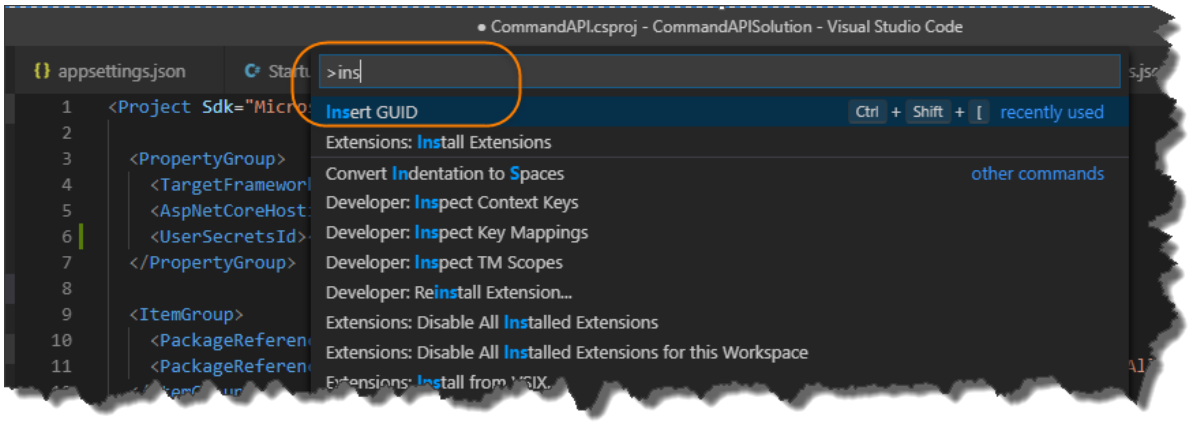
Learning Opportunity: Find out what GUID stands for and do a little bit of reading on what they are and where they can be used, (assuming you don't know this already!)

Cast your mind back to Chapter 2 where we set up our development lab, and one of the extensions we suggested for VS Code was *Insert GUID* – well now we get to use it!

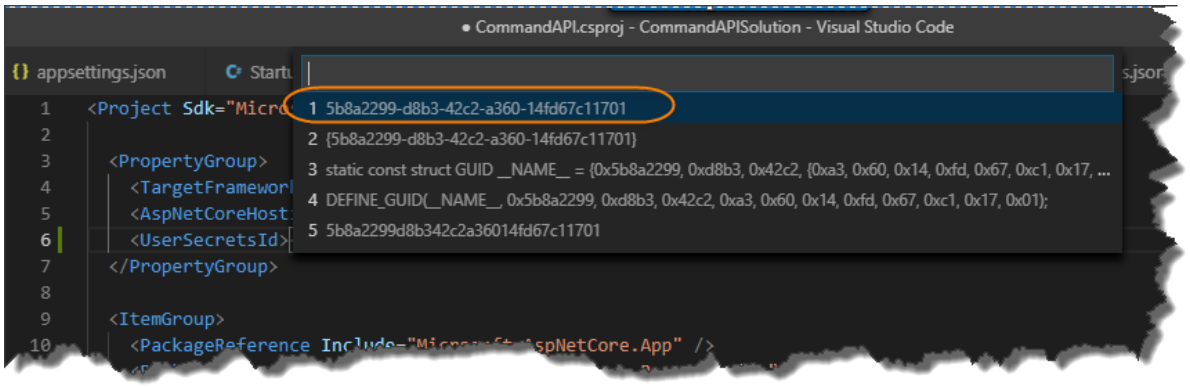
In VS Code open your ***CommandAPI.csproj*** file, and in the `<PropertyGroup>` xml element, place the xml highlighted below:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
    <UserSecretsId></UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0"
PrivateAssets="All" />
  </ItemGroup>

</Project>
```

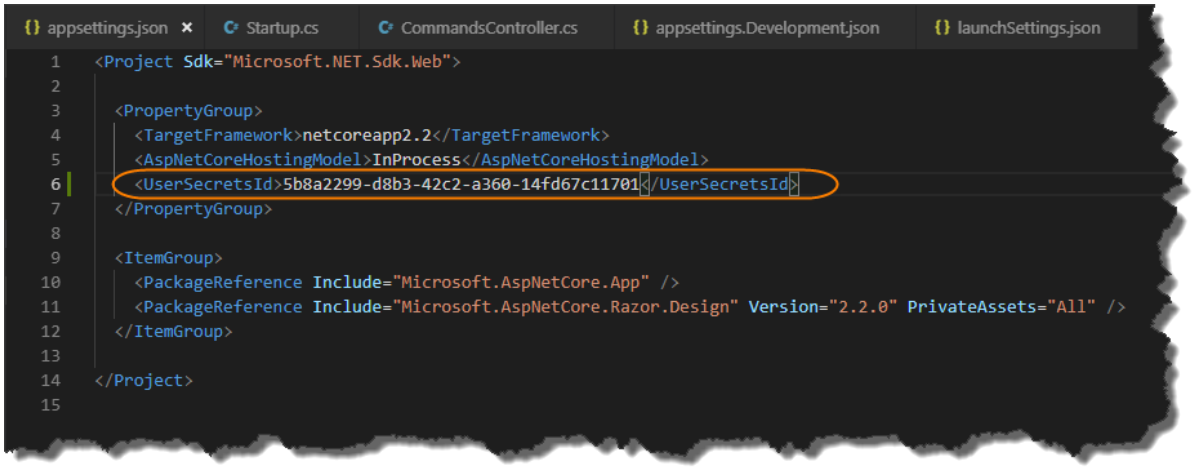
- Place your cursor in between the opening `<UserSecretsId>` and the closing `</UserSecretsId>` elements.
- Open the VS Code “Command Palette”
  - Hit: F1
  - Or: Ctrl + Shift + P
  - Or: View -> Command Palette…
- Type “Insert”



- Insert GUID should appear, select it and select the 1st GUID Option:



- This should place the auto-generated GUID into the xml elements specified, see example below:



Now save your file.

Deciding Your Secrets

Now we come to actually adding our secrets via The Secret Manager Tool, which will generate a ***secrets.json*** file.

Before we do that though, we have a decision to make in regard our connection string… Do we:

1. Want to store our entire connection string as a single secret
2. Store our User Id and Password as individual secrets and retain the remainder of the connection string in the ***appsettings.Developent.json*** file

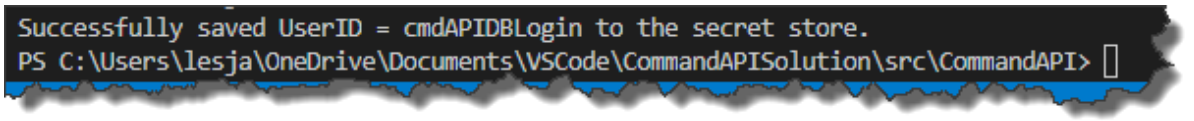
Either will work, but I'm going to go with Option 2 where we will store the individual components as "secrets". I've taken this approach primarily when we come on to talking about Azure Key Vault in later chapters.

So to add our two secrets:

- Ensure you have generated the GUID as described above and save the .csproj file
- At a terminal command, (and make sure you're "inside" the ***CommandAPI*** project folder), type:

```
dotnet user-secrets set "UserID" "cmdAPIDBLogin"
```

You should get a: "Successfully saved UserID…" message:



Repeat the same step and add the "Password" secret:

```
dotnet user-secrets set "Password" "ABadPassword1234!"
```

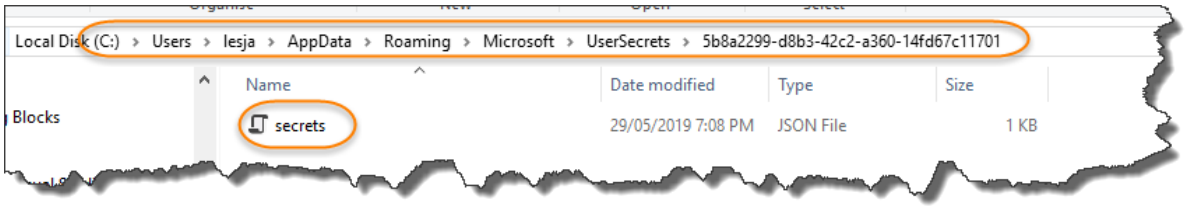
Again you should get a similar success message.

Where Are They?

So where did our secrets end up? That's right, in our ***secrets.json*** file. You can find this file in a system-protected user profile folder on your local machine at the following location:

- Windows: `%APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json`
- Linux/OSX: `~/.microsoft/usersecrets/<user_secrets_id>/secrets.json`

So on my machine, it can be found here:



Open this file, and have a look at the contents:

```
{
  "UserID": "cmdAPIDBLogin",
  "Password": "ABadPassword1234!"
}
```

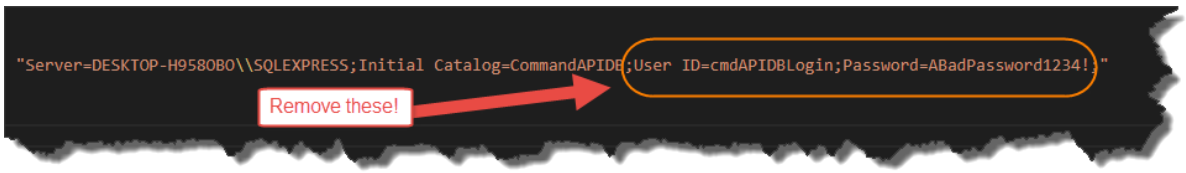
It's just simple, non-encrypted json.

Code it Up

Ok so now the really exciting bit where we'll actually use these secrets to build out our full connection string.

Step 1: Remove User ID and Password

We want to remove the "offending articles" from our existing connection string in our ***appsettings.Development.json*** file:



So our ***appsettings.Development.json*** file should now contain only:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "ConnectionStrings":
  {
    "CommandAPISQLConection" : "Server=DESKTOP-H958OBO\\SQLEXPRESS;
       Initial Catalog=CommandAPIDB;"
  }
}
```

Note: your server name will be different to mine!

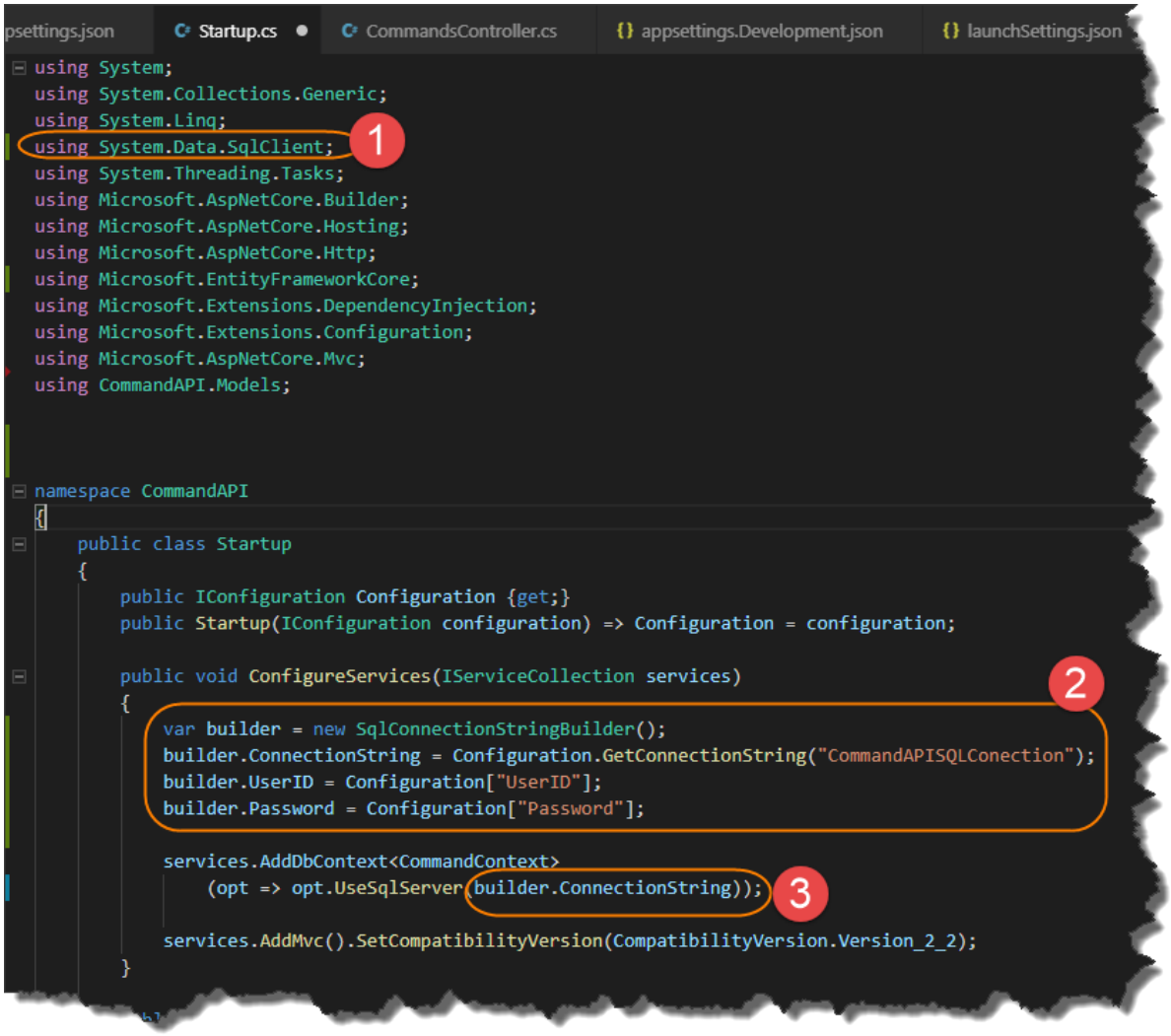
Make sure you save your file.

STEP 2: BUILD OUR CONNECTION STRING

Move over into our `Startup` class and add the following code to the `ConfigureServices` method:

```
.
.
.
using System.Data.SqlClient;

namespace CommandAPI
{
    public class Startup
    {
        public IConfiguration Configuration {get;}
        public Startup(IConfiguration configuration) => Configuration =
configuration;

        public void ConfigureServices(IServiceCollection services)
        {
            var builder = new SqlConnectionStringBuilder();
            builder.ConnectionString =
              Configuration.GetConnectionString("CommandAPISQLConection");
            builder.UserID = Configuration["UserID"];
            builder.Password = Configuration["Password"];

            services.AddDbContext<CommandContext>
                (opt => opt.UseSqlServer(builder.ConnectionString));

            services.AddMvc()
             .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
        }
.
.
.
```

Again for clarity I've circled the new / updated sections below:

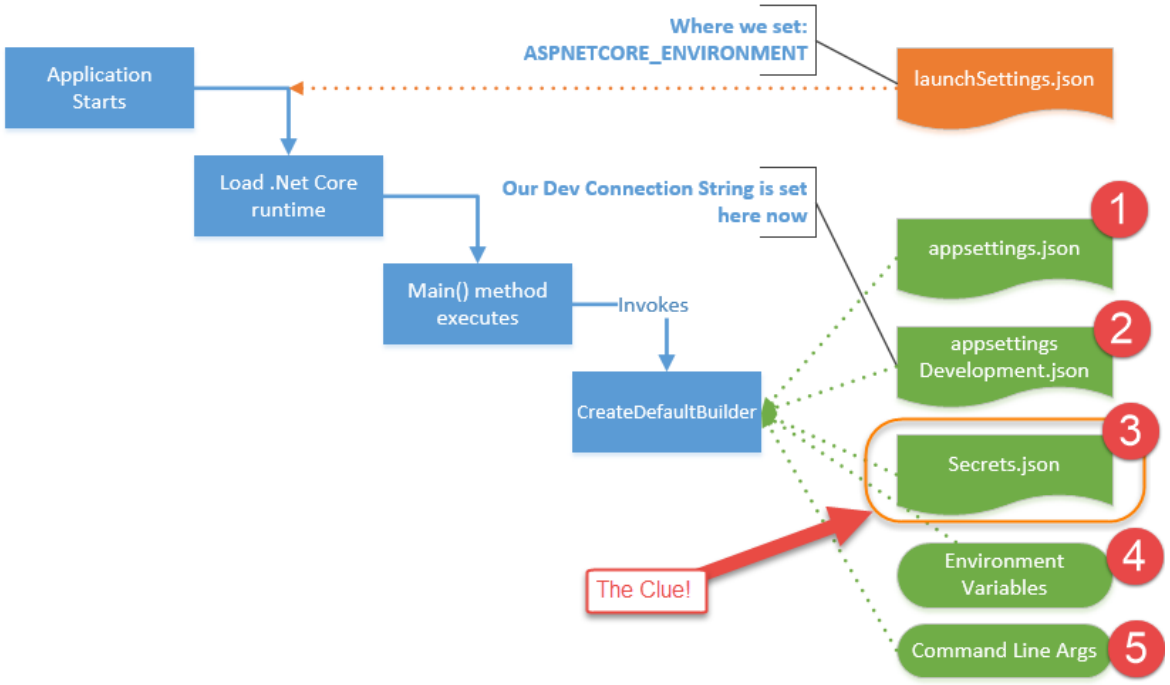


1. We need to add a reference to `System.Data.SqlClient` in order to use `SqlConnectionStringBuilder`
2. This is where we:
    a. Create a `SqlConnectionStringBuilder` object and pass in our "base" connection string: `CommandAPISQLConnection` from our ***appsettings.Development.json*** file.
    b. Continue to "build" the string by passing in both our `UserID` and `Password` secret from our ***secrets.json*** file.
3. Replace the original connection string with the newly constructed string using our `builder` object.

Save your work, build it, then run it. Fire up Postman and issue our GET request to our api… You should get a success!

Celebration Check Point: You have now dynamically created a connection string using a combination of configuration sources, one of which is User Secrets from our ***secrets.json*** file!

Just cast your mind back to the following diagram:



The .NET Configuration layer by default provides us access to the configuration sources as shown above, in this case we used a combination of 2 + 3.

Wrap It Up

Again we covered a lot in this chapter, the main points are:

- We moved our connection string to a development only config file: ***appsetting.Development.json***
- We removed the sensitive items from our connection string
- We moved the sensitive items, (`UserID` & `Password`), to ***secrets.json*** via The Secret Manager Tool
- We constructed a fully working connection string using a combination of configuration sources.

All that's left to do is commit all our changes to Git then push up to GitHub!

Moving over to our repository and taking a look in the **appsettings.Development.json** file, we see an innocent connection string without user credentials, (the ***secrets.json*** file is not added to source control)!

Branch: master ▾ CommandAPI / src / CommandAPI / appsettings.Development.json

binarythistle Secured our connection string

1 contributor

14 lines (13 sloc) | 267 Bytes Raw Blame

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "ConnectionStrings":
  {
    "CommandAPISQLConection" : "Server=DESKTOP-H958OBO\\SQLEXPRESS;Initial Catalog=CommandAPIDB;"
  }
}
```

Chapter 8 – Unit Testing & Completing Our API

Chapter Summary

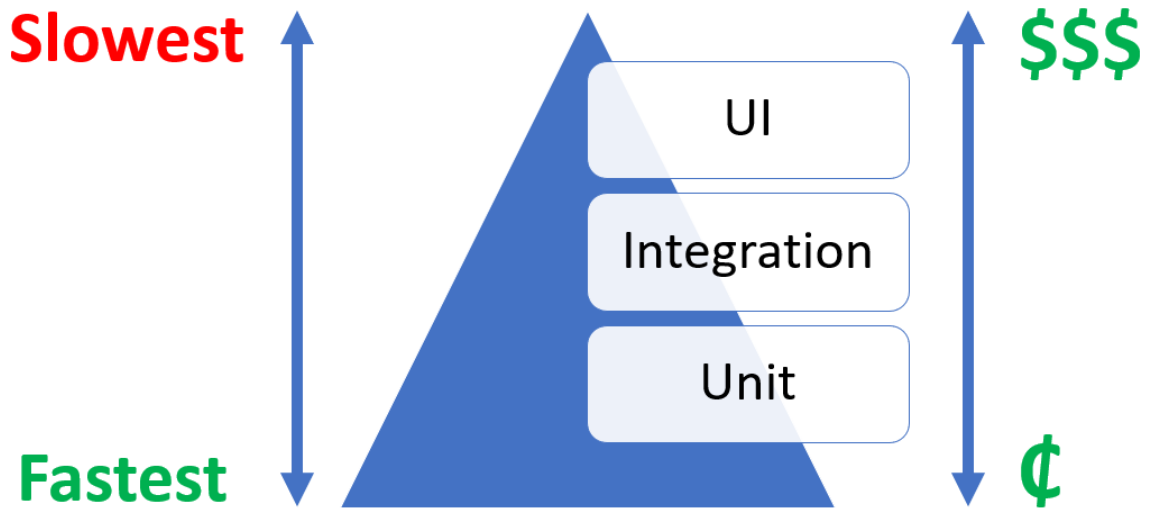
In this chapter we'll introduce you to Unit Testing, what it is, and why you'd use it. We'll then use unit testing to complete the development of our API, in an approach called *Test Driven Development*.

When Done, You Will

- Understand what Unit Testing is
- Write a Unit Test using *xUnit* to test our *existing* API functionality
- Get introduced to Mocking Framework: Moq
- Use Test Driven Development to complete the development of our API

What Is Unit Testing

Probably the best way to describe what Unit Testing is, is to put it in context of the other general types of "testing" you will encounter, so I refer you to the "Testing Pyramid" below:



So unit tests are:

- Abundant: there should be more of them than other types of test.
- Small: they should test 1 thing only, i.e. a "unit", (as opposed to full end to end "scenarios" or use cases)
- Cheap: they are both written and executed first. This means any errors they catch should be easier to rectify when compared to those you catch much later in the development lifecycle.
- Quick to both write and execute

Unit tests are written by the developer, (as opposed to a tester or business analyst), so that is why we'll be using them here to test our own code.

Ok so aside from the fact that they are quick and cheap, what other advantages do you have in using them?

Protection Against Regression

Because you'll have a suite of unit tests that are built up over time, you can run them again every time you introduce new functionality, (that you should also build tests for). This means that you can check to see if your new code had introduced errors to the existing code base, (these are called *regression defects*). Unit testing therefore gives you confidence that you've not introduced errors, or if you have, give you an early heads up so you can rectify.

Executable Documentation

When we come to write some unit tests, you'll see that the way we name them is descriptive and speaks to what is being tested and the expected outcome. Therefore, assuming you take this approach, your unit test suite essentially becomes documentation for your code.

When naming your unit test methods, they should follow a construct similar to:

```
<method name>_<expected result>_<condition>
```

E.g.

```
GetCommandItem_Returns200OK_WhenSuppliedIDIsValid
```

Note: there are variants on the above convention, so find the one the one that works best for you.

Characteristics of a Good Unit Test

I've taken the following list of unit test characteristics from this Unit Testing Best Practices guide by Microsoft; it's well worth a read.

- **Fast.** Individual tests should execute quickly, (required as we can have 1000's of them), and when we say quick, we're talking in the region of milliseconds.
- **Isolated.** Unit tests should not be dependent on external factors, e.g. databases, network connections etc.
- **Repeatable.** The same test should yield the same result between runs, (assuming you don't change anything between runs).
- **Self-checking.** Should not require human intervention to determine whether it has passed or failed.
- **Timely.** The unit test should not take a disproportionately long time to run compared with the code being tested.

I'd also add:

- **Focused.** A unit test, (as the name suggests, and as mentioned above), should test only 1 thing.

We'll use these factors as a touchstone when we come to writing our own tests.

What to Test?

Ok so we know what they are, why we have them and even the characteristics of a "good" test, but the $64000 question is what should we actually test? The characteristics should help drive this choice, but ultimately it comes down to the individual developer and what they are happy with.

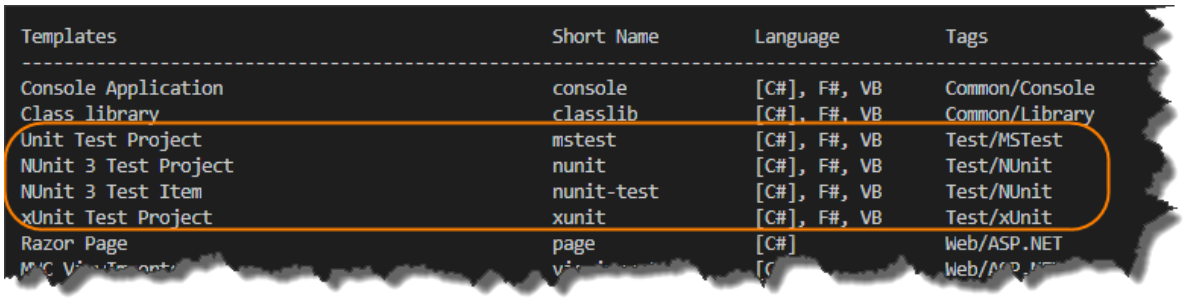
Some developers may only write a small number of unit tests that only test really novel code, others may write many more, that test more standard, trivial functionality… As our API is simple, we'll be writing tests that are pretty basic, and test quite obvious functionality. I've taken this approach to get you used to unit testing more than anything else.

Note: You would generally not test functionality that is inherent in the programming language: e.g. you would not write unit tests to check basic arithmetic operations for example – that would be over kill and not terribly useful.

Unit Testing Frameworks

I asked a question at the start of the book about what is xUnit? Well xUnit is simply a unit testing framework, it's open source and was used heavily in in the creation of .NET Core, so it seems like a pretty good choice for us!

There are alternatives of course that do pretty much the same thing, performing a `dotnet new` at the command line you'll see the unit test projects available to us:



The others we could have used are:

- MSTest
- NUnit

We'll be sticking with xUnit though so if you want to find out about the others, you'll need to do your own reading.

Arrange, Act & Assert

Irrespective of your choice of framework, all unit tests follow the same pattern, (xUnit is no exception):

Arrange

This is where you perform the "set up" of your test. E.g. you may set up some objects and configure data used to drive the test.

Act

This is where you execute the test to generate the result.

Assert

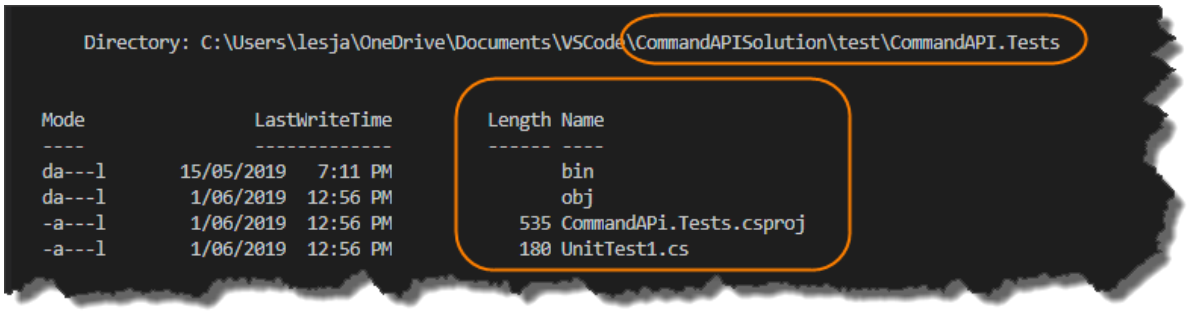
This is where you "check" the *actual result* against the *expected result*. Dependant on how that assertion goes, will depend on whether your test passes or fails.

Going back to the characteristics of a good unit test, the "focused" characteristic comes in to play here, meaning that we should really have only *1 assertion per test*. If you assert multiple conditions, the unit tests becomes diluted and confusing – what are you testing again?

So enough theory – let's practice!

Write Our First Tests

Ok so we now want to move away from our API project and into our unit test project. So in your terminal, navigate into the ***Command.Tests*** folder, listing the contents of that folder you should see:

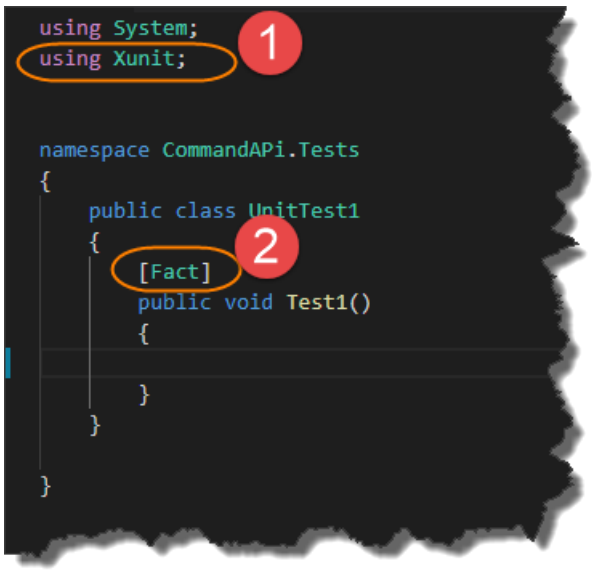


We have:

- ***bin*** folder
- ***obj*** folder
- ***CommandAPI.Tests.csproj*** project file
- ***UnitTest1.cs*** default class

You should be familiar with the 1st 3 of these, as they are the same artefacts we had in our API project. With regard the project file: ***CommandAPI.Tests.csproj***, you'll recall we added a reference to our API project in here so we can "test" it.

The 4th and final artefact here is a default class set up for us when we created the project, open it and take a look:



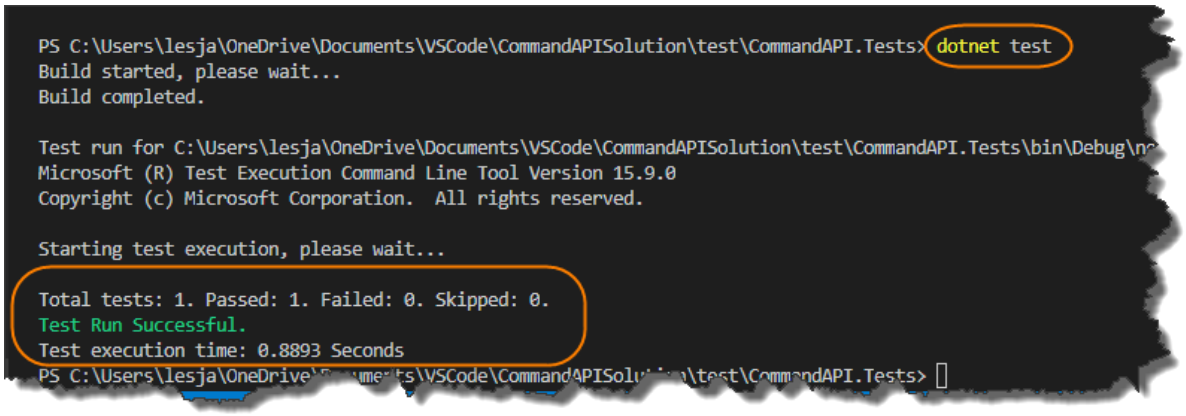
This is just a standard class definition, with only 2 points of note:

1. A reference to xUnit
2. Our class method `Test1` is decorated with the `[Fact]` attribute. This tells the xUnit test runner that this method is a test.

You'll see at this stage our `Test1` method is empty, but we can still run it nonetheless, to do so, return to your terminal, (ensure you're in the ***CommandAPI.Tests*** folder) and type:

```
dotnet test
```

This will run our test which should "pass", although it's empty and not really doing anything:



Ok, we know our testing set up is good to go, so let's start writing some tests.

Testing Our Model

Our first test is really at the trivial end of the spectrum to such an extent you probably wouldn't unit test this outside the scope of a learning exercise. However, *this is a learning exercise*, and even though it is a simple test, it covers all the necessary mechanics to get a unit test up and running.

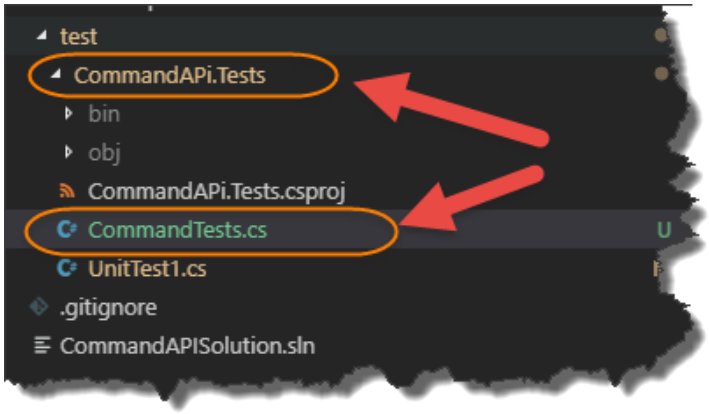
Thinking about our model what would we want to test? As a refresher here's the model class in our API project:

```
1   namespace CommandAPI.Models
2   {
3       public class Command
4       {
5           public int Id {get; set;}
6           public string HowTo {get; set;}
7           public string Platform {get; set;}
8           public string CommandLine {get; set;}
9       }
10  }
11
```

How about: *We can change the value of each of the class attributes?*

There are probably others we could think of, but let's keep it simple to start with. To set this up we're going to create a new class that will contain tests only for our `Command` model, so:

- Create a new file called ***CommandTests.cs*** in our ***CommandAPI.Tests*** Project



Add the following code to this class:

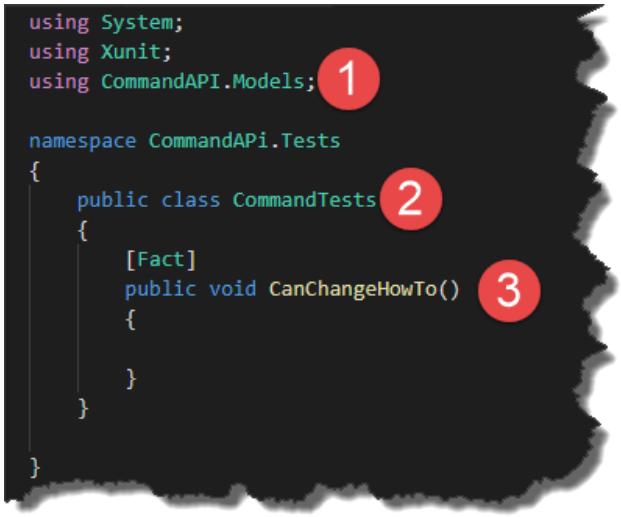
```
using System;
using Xunit;
using CommandAPI.Models;

namespace CommandAPI.Tests
{
    public class CommandTests
    {
        [Fact]
        public void CanChangeHowTo()
        {

        }
    }
}
```

This is such a trivial test, (we're not even testing as method), we can't really use the unit test naming convention mentioned above:

`<method name>_<expected result>_<condition>`

So in this instance we're going with something more basic.

The following sections are of note:



1. We have a reference to our Models in the ***CommandAPI*** project
2. Our Class is named after what we are testing (i.e. our `Command` model).
3. The naming convention of our test method is such that it tells use what the test is testing for.

Ok so now time to write our Arrange, Act and Asset code, add the following highlighted code to the `CanChangeHowTo` test method:

```
Fact]
public void CanChangeHowTo()
{
  //Arrange
  var testCommand = new Command
  {
    HowTo = "Do something awesome",
    Platform = "xUnit",
    CommandLine = "dotnet test"
  };

  //Act
  testCommand.HowTo = "Execute Unit Tests";

  //Assert
  Assert.Equal("Execute Unit Tests", testCommand.HowTo);
}
```

The sections we added are highlighted below:

1. Arrange: Create a `testCommand` and populate with initial values
2. Act: Perform the action we want to test, i.e. change the value of `HowTo`
3. Assert: Check that the value of `HowTo` matches what we expect

Steps 1 & 2 are straightforward, so it's really step 3, and the use of the `xUnit Assert` class to perform the "`Equal`" operation that is possibly new to you. Whether this step is true or false determines whether the test passes or fails.

So let's run our very simple test to see if it passes or fails:

- Ensure you save your ***CommandTests.cs*** file
- `dotnet build` - this will just check your tests are syntactically correct
- `dotnet test` – will run our test suite

The test should pass and you'll see something like:

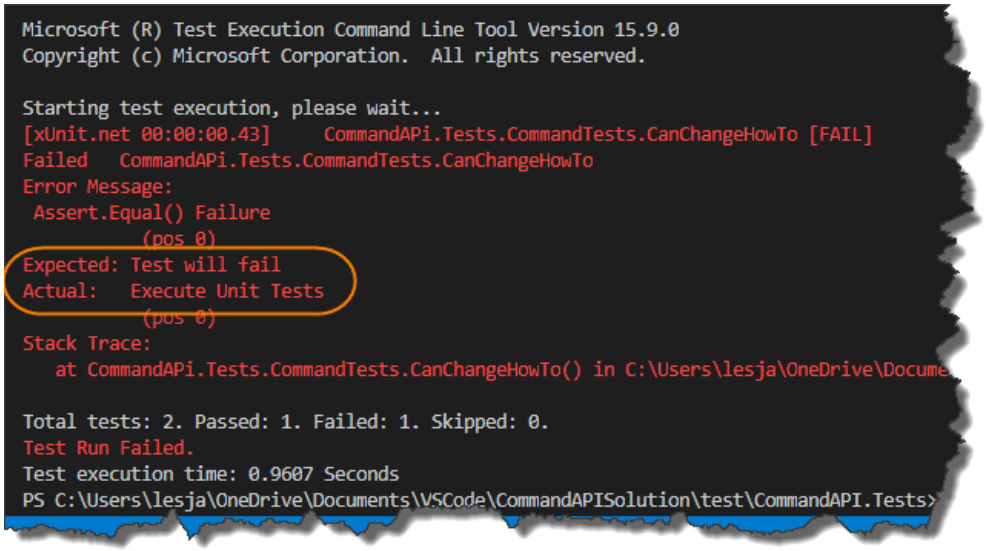
```
Time Elapsed 00:00:00.91
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\test\CommandAPI.Tests> dotnet test
Build started, please wait...
Build completed.

Test run for C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\test\CommandAPI.Tests\bin\Debug\
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: 0.8909 Seconds
PS C:\Users\lesja\OneDrive\Documents\VSCode\CommandAPISolution\test\CommandAPI.Tests>
```

It says 2 tests have passed? Where is the other test? That's right we still have our original `UnitTest1` class with an empty test method, so that's where the 2nd test is being picked up. Before we continue, lets' delete that class.

We can also "force" this test to fail. To do so change the "expected" value in our `Assert.Equal` operation to something random, e.g.

```
[Fact]
public void CanChangeHowTo()
{
    //Arrange
    var testCommand = new Command
    {
        HowTo = "Do something awesome",
        Platform = "xUnit",
        CommandLine = "dotnet test"
    };

    //Act
    testCommand.HowTo = "Execute Unit Tests";

    //Assert
    Assert.Equal("Test will fail", testCommand.HowTo);

}
```

Save the file and re-run your tests, you'll get a failure response with some verbose messaging:



Here you can see the test has failed and we even get the reasoning for the failure… Revert the expected string back to a passing value before we continue.

Learning Opportunity: We have 2 other attributes in our Command class that we should be testing for: `Platform` and `CommandLine`, (the Id attribute is auto-managed so we shouldn't bother with this for now).

Write 2 additional tests to test that we can change these values too.

Don't Repeat Yourself

Ok so assuming that you completed the last Learning Opportunity, you should now have 3 test methods in your `CommandTests` class, with 3 passing tests. If you didn't complete that, I'd suggest you do it, or if you really don't want to – refer to the code on GitHub.

One thing you'll notice is that the Arrange component for each of the 3 tests is identical, and therefore a bit wasteful. When you have a scenario like this – i.e. you need to perform some standard set up that multiple tests use, xUnit allows for that.

The xUnit documentation describes this concept as *Shared Context* between tests and specifies 3 approaches to achieve this:

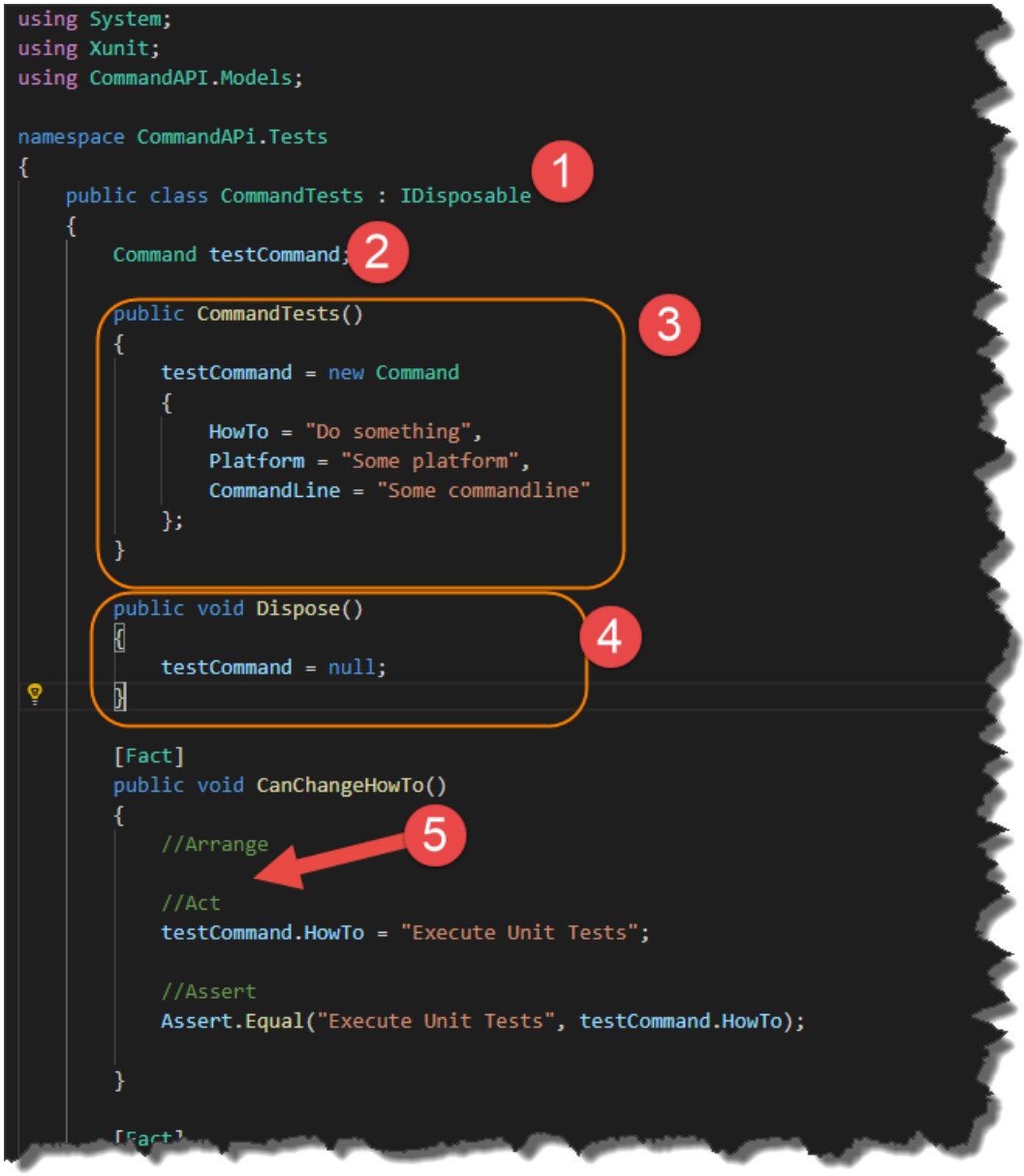
- Constructor and Dispose (shared setup/clean-up code without sharing object instances)
- Class Fixtures (shared object instance across tests in a *single class*)
- Collection Fixtures (shared object instances across multiple test classes)

We are going to use the first approach, which will set up a new instance of the `testCommand` object for each of our tests, you can alter your `CommandsTests` class to the following:

```
using System;
using Xunit;
using CommandAPI.Models;

namespace CommandAPI.Tests
{
```

```
    public class CommandTests : IDisposable
    {
        Command testCommand;

        public CommandTests()
        {
            testCommand = new Command
            {
                HowTo = "Do something",
                Platform = "Some platform",
                CommandLine = "Some commandline"
            };
        }

        public void Dispose()
        {
            testCommand = null;
        }

        [Fact]
        public void CanChangeHowTo()
        {
            //Arrange

            //Act
            testCommand.HowTo = "Execute Unit Tests";

            //Assert
            Assert.Equal("Execute Unit Tests", testCommand.HowTo);

        }

        [Fact]
        public void CanChangePlatform()
        {
            //Arrange

            //Act
            testCommand.Platform = "xUnit";

            //Assert
            Assert.Equal("xUnit", testCommand.Platform);

        }

        [Fact]
        public void CanChangeCommandLine()
        {
            //Arrange

            //Act
            testCommand.CommandLine = "dotnet test";

            //Assert
            Assert.Equal("dotnet test", testCommand.CommandLine);

        }
    }
}
```

For clarity the sections we have added are:



1. We inherit the `IDisposable` interface, (used for code clean up)
2. Create a "global" instance of our `Command` class
3. Create a Class Constructor where we perform the set up of our `testCommand` object instance
4. Implement a Dispose method, to clean up our code
5. You'll notice that the Arrange section for each test is now empty, the class constructor will be called for every test (I've only shown 1 test here for brevity)

For more information refer to the xUnit documentation.

Test Our Existing Controller Action

Ok, so testing our model was just an *amuse-bouche*[17] for what's about to come next: testing our Controller. We up the ante here as it's a decidedly more complex affair, although the concepts you learned in the last section still hold true, we just expand upon that here.

Referring back to our `CommandsController` class:

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using CommandAPI.Models;
using Microsoft.AspNetCore.Hosting;


namespace CommandAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CommandsController : ControllerBase
    {
        private readonly CommandContext _context;
        private IHostingEnvironment _hostEnv;

        public CommandsController(CommandContext context, IHostingEnvironment hostEnv)
        {
            _context = context;
            _hostEnv = hostEnv;
        }


        //GET:      api/commands
        [HttpGet]
        public ActionResult<IEnumerable<Command>> GetCommandItems()
        {
            Response.Headers.Add("Environment", _hostEnv.EnvironmentName);

            return _context.CommandItems;
        }


    }
}
```

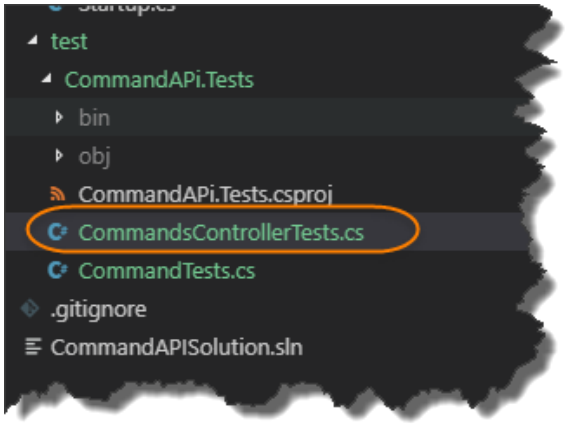
We had 1 `ActionResult, (GetCommandItems)`, that returned a list of `Command` objects as a serialized JSON string:. So, we want to test that method via unit testing. So what sort of things should we be testing for? How about:

Test ID	Condition	Expected Result
Test 1.1	Request Objects when none exist	Return "nothing"
Test 1.2	Request Objects when 1 exists	Return Single Object
Test 1.3	Request Objects when *n* exist	Return Count of *n* Objects
Test 1.4	Request Objects	Return the correct "type"

[17] Bite-sized hors d'oeuvre, literally means "mouth amuser" in French. They differ from appetizers in that they are not order from a menu by patrons but are served free and according to the chef's selection alone.

You can begin to see that what you want to test is somewhat subjective, but you want to try and capture cases that if false would comprise the validity of your API.

As with our `Command` model, we want to create a separate test class in our unit test project to hold our tests, so create a class called: ***CommandsControllerTests.cs***, as shown below:

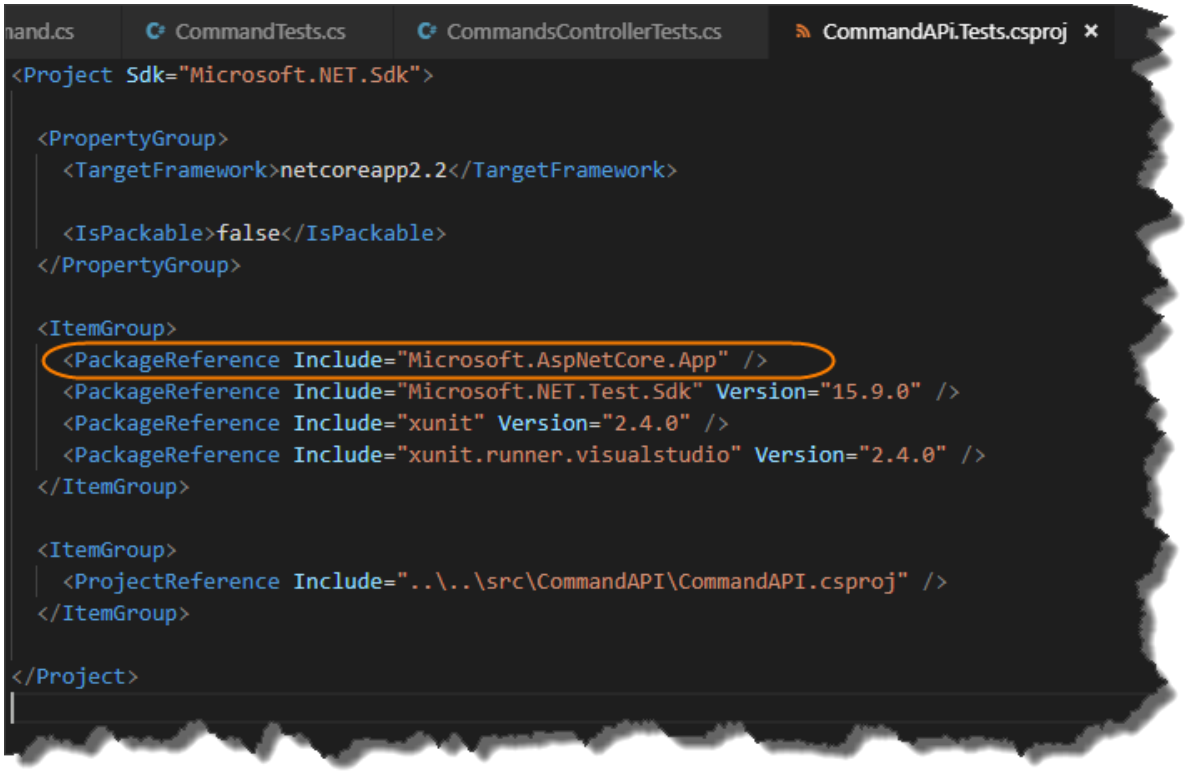


The content of this class to start with should be:

```
using System;
using Xunit;
using Microsoft.EntityFrameworkCore;
using CommandAPI.Controllers;
using CommandAPI.Models;

namespace CommandAPI.Tests
{
    public class CommandsControllerTests
    {

    }
}
```

You'll see that we have using directives for both our `CommandAPI.Controllers` and `CommandAPI.Models`, as we need to reference classes from both these to run our tests.

Additionally you can see that we have a using directive to `Microsoft.EntityFrameworkCore`, again this is required for setting up our tests. In order for that last `using` directive to work though we have to place a *Package Reference* to `Microsoft.AspNetCore.App` in the .csproj file of our unit test project, as shown below:



Copy the following text into the .csproj file, remembering to save it!

```
<PackageReference Include="Microsoft.AspNetCore.App" />
```

Before we write our first test: checking that we get “nothing” if we have no items in our DB, let’s quickly refer back to the characteristics of a good unit test:

- **Fast.** Individual tests should execute quickly, (required as we can have 1000’s of them), and when we say quick, we’re talking in the region of milliseconds.
- **Isolated.** Unit tests should not be dependent on external factors, e.g. databases, network connections etc.
- **Repeatable.** The same test should yield the same result between runs, (assuming you don’t change anything between runs).
- **Self-checking.** Should not require human intervention to determine whether it has passed or failed.
- **Timely.** The unit test should not take a disproportionately long time to run compared with the code being tested.
- **Focused.** A unit test, (as the name suggests, and as mentioned above), should test only 1 thing.

Also looking at our `CommandsController` class:

```
[Route("api/[controller]")]
[ApiController]
public class CommandsController : ControllerBase
{
    private readonly CommandContext _context;
    private IHostingEnvironment _hostEnv;

    public CommandsController(CommandContext context, IHostingEnvironment hostEnv)
    {
        _context = context;
        _hostEnv = hostEnv;
    }


    //GET:      api/commands
    [HttpGet]
    public ActionResult<IEnumerable<Command>> GetCommandItems()
    {
        Response.Headers.Add("Environment", _hostEnv.EnvironmentName);

        return _context.CommandItems;
    }


}
```

Is there anything here that should be cause for concern in relation to our unit test characteristics? The one that jumps out at me is the **Isolation** characteristic: that being unit tests should not be reliant on external factors to run. When creating an instance of our controller, we need to pass in both a `DBContext` and a `HostingEnvironment` instance, we're going to have to solve that dependency problem then…

There are many ways you can deal with external components when Unit testing, and I'm going to cover 2 different approaches with regard our `DBContext` and `HostingEnvironment`.

Now again, different developers may take a different, and equally valid approaches, the methods I'm showing you just made sense for me and this book.

DBContext

For our DBContext the main thing we want to do is not be dependent on an external database, (i.e. the SQL Server DB we have set up in our Development environment), as this would totally break the **Isolation** principle, and possibly also the **Fast** principle too…

So what we're going to do here is use our *concrete*[18] DBContext class implementation and use it with an "In Memory Database", thus removing the external dependency on a SQL Server.

Warning: The In Memory Database does not model the behaviour of the SQL Server exactly. However as we are *unit testing*, (and not *integration testing*), this is not a huge deal – worth bearing in mind though…

[18] As opposed to an Interface

We'll set this up in the Arrange section of our 1st Test method in our `CommandsControllerTests` class. First create a new test method called `ReturnsZeroItemsWhenDBIsEmpty`, and add the following code to *arrange* the test:

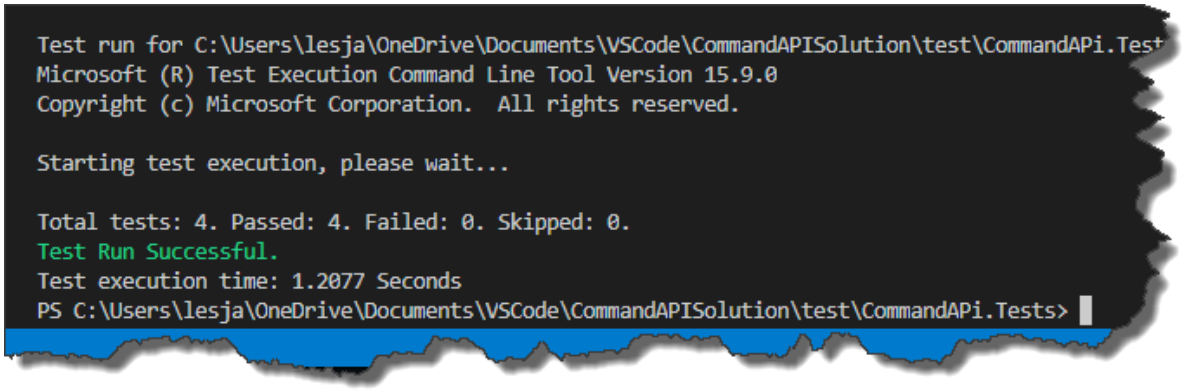
```
using System;
using Xunit;
using Microsoft.EntityFrameworkCore;
using CommandAPI.Controllers;
using CommandAPI.Models;

namespace CommandAPI.Tests
{
    public class CommandsControllerTests
    {
        [Fact]
        public void GetCommandItems_ReturnsZeroItems_WhenDBIsEmpty()
        {
            //Arrange
            //DBContext
            var optionsBuilder = new DbContextOptionsBuilder<CommandContext>();
            optionsBuilder.UseInMemoryDatabase("UnitTestInMemBD");
            var dbContext = new CommandContext(optionsBuilder.Options);
        }
    }
}
```

A description of what's going on in the following section is describe below:

```
namespace CommandAPi.Tests
{
    public class CommandsControllerTests
    {
        [Fact]
        public void ReturnsZeroItemsWhenDBIsEmpty()
        {
            //Arrange
            //DBContext
            var optionsBuilder = new DbContextOptionsBuilder<CommandContext>();
            optionsBuilder.UseInMemoryDatabase("UnitTestInMemBD");
            var dbContext = new CommandContext(optionsBuilder.Options);
```

- We create a `DBContextOptionsBuilder` with our `CommandContext` class
- We specify that we want to use an In Memory Database, removing any external dependencies
    - The name "UnitTestInMemDB" is arbitrary, naming it allows us to reference it elsewhere if we require
- We create the `dbContext` object we'll use when creating our controller

While we can't test anything yet it's prudent to do a build and run our tests just to make sure everything is wired up correctly. Indeed, you'll see in the tests results that you have 4 tests now, all of which are passing. (As mentioned our new test isn't testing anything yet).



Ok, good we've set up our `DBContext` object, now onto the `HostingEnvironment`.

IHostingEnvironment

Now you recall the only reason we pass `IHostingEnvironment` into our controller constructor is so that we can reference the run-time environment and attach it to our response object. To be honest, this isn't something that I'd do in a production environment, but we'll keep it here for now as it will pay off when we come to deploying to Azure. It also forces us to use another "Isolation" technique from a unit testing perspective so it serves another purpose!

Now the problem we have here is that our controller expects an object instance that implements the `IHostingEnvironment` interface. This isn't a problem in our API project as we are provided one of those as part of our startup, for our unit test project we don't have that…

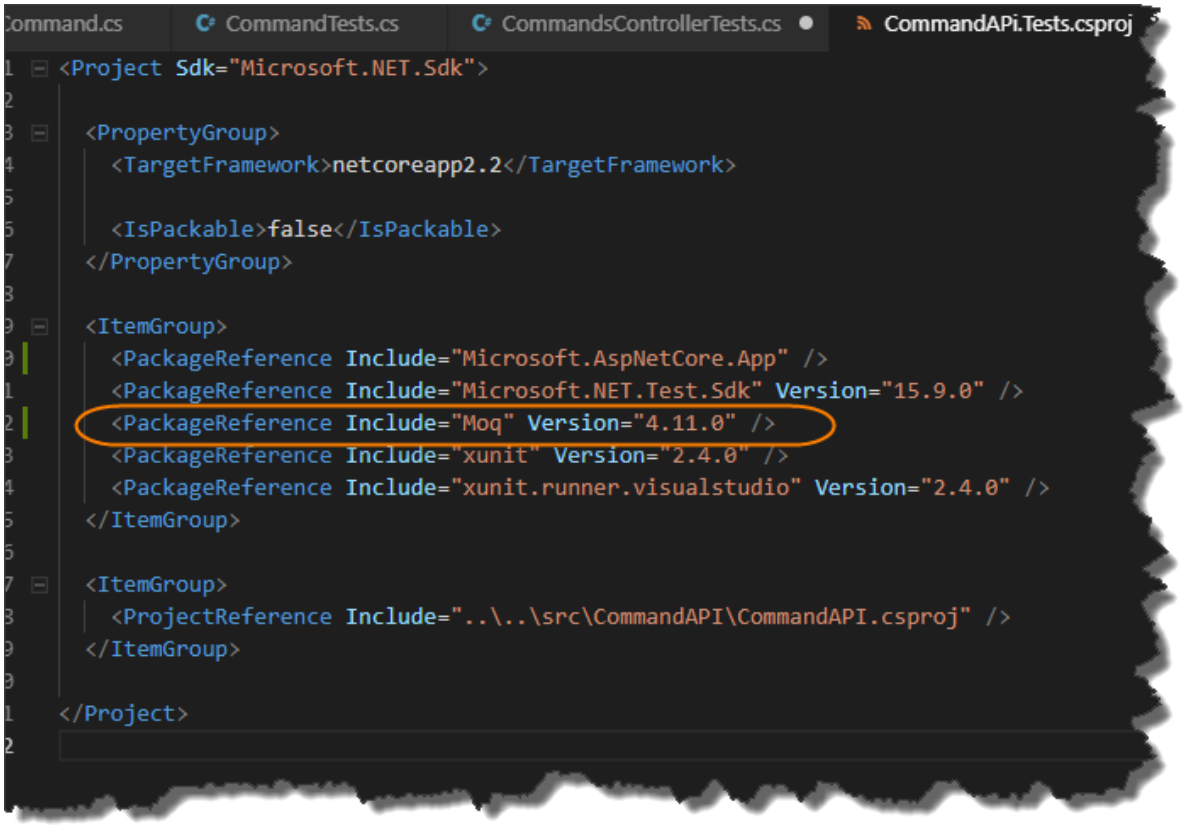
Are You Mocking Me?

Thankfully we can turn to something called "mocking", which essentially means we can create "fake", (or mock), copies of any required objects to use within our unit tests. It basically allows us to self-contain everything we need in our unit test project and adhere to the **Isolation** principle.

In this instance we are going to turn to a package aptly named Moq. In order to use it, like with all other packages, we need to add a reference inside our ***CommandAPI.Tests.csproj*** file, you can either add it manually as we did above with `Microsoft.EntityFrameworkCore` or we can use the command line. So, ensure you're "in" the ***CommandAPI.Tests*** folder and type:

```
dotnet add package Moq
```

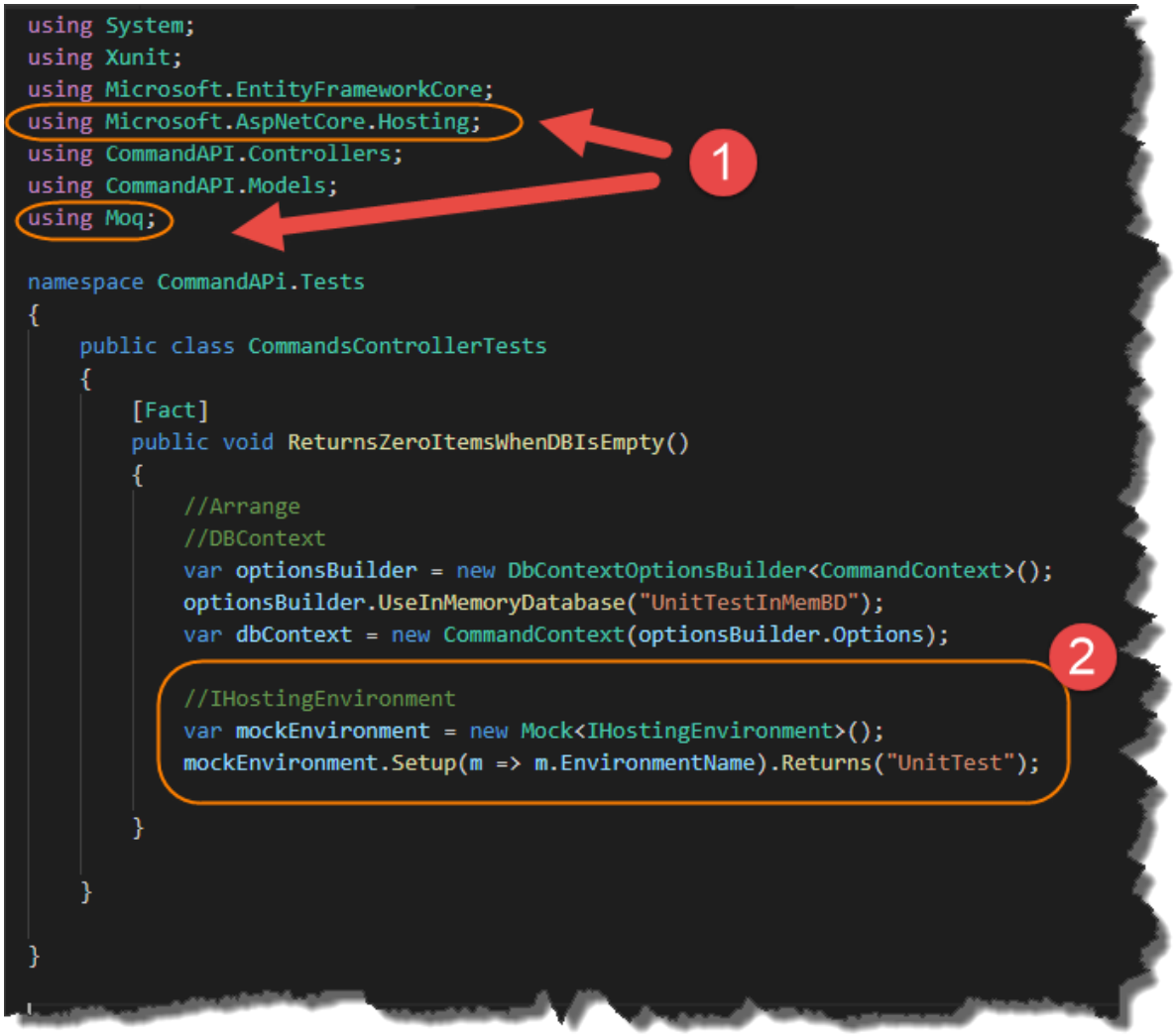
This will add the necessary reference to the ***CommandAPI.Tests.csproj*** file as follows:



Setting up a mock object that implements the IHostingEnvironment interface is then easy, we add the following code to the arrange section of our `GetCommandItems_ReturnsZeroItems_WhenDBIsEmpty` test, as shown below:

```
using System;
using Xunit;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Hosting;
using CommandAPI.Controllers;
using CommandAPI.Models;
using Moq;

namespace CommandAPI.Tests
{
    public class CommandsControllerTests
    {
        [Fact]
        public void GetCommandItems_ReturnsZeroItems_WhenDBIsEmpty()
        {
            //Arrange
            //DBContext
            var optionsBuilder = new DbContextOptionsBuilder<CommandContext>();
            optionsBuilder.UseInMemoryDatabase("UnitTestInMemBD");
            var dbContext = new CommandContext(optionsBuilder.Options);

            //IHostingEnvironment
            var mockEnvironment = new Mock<IHostingEnvironment>();
            mockEnvironment.Setup(m => m.EnvironmentName).Returns("UnitTest");
        }
    }
}
```

The sections we've added are described below:



1. We require 2 new using directives: `Moq`, (obviously as we're going to use it), and `Microsoft.AspNetCore.Hosting` – this allows to create an object based on `IHostingEnvironment`.
2. We create a "mock object": `mockEnvironement` and use Moq to create an object that implements `IHostingEnvironment`. We perform some "setup" on that mock object and set the `EnvironmentName` to something arbitrary, in this case "UnitTest";

We then have one final piece of *arranging* to do, create an instance of our troublesome controller, to do so add the following code:

```
.
.
.
[Fact]
public void GetCommandItemsReturnsZeroItemsWhenDBIsEmpty()
{
  //Arrange
  //DBContext
  var optionsBuilder = new DbContextOptionsBuilder<CommandContext>();
  optionsBuilder.UseInMemoryDatabase("UnitTestInMemBD");
  var dbContext = new CommandContext(optionsBuilder.Options);

  //IHostingEnvironment
  var mockEnvironment = new Mock<IHostingEnvironment>();
```

```
  mockEnvironment.Setup(m => m.EnvironmentName).Returns("UnitTest");

  //Controller
  var controller = new CommandsController(dbContext, mockEnvironment.Object);
}
.
.
.
```

The only thing worth noting is that in order to expose the mock object instance and use it to create out controller we need to use: `mockEnvironment.Object`

Again, even though we're not testing anything yet, build the project to ensure its wired up correctly and run your tests, again you should get 4 passing tests.

ACT & ASSERT

The Act and Assert parts of our first controller test are quite trivial in comparison to the work we were required to do in the Arrange section, they can both be found in the highlighted code section below:

```
[Fact]
public void GetCommandItemsReturnsZeroItemsWhenDBIsEmpty()
{
  //Arrange
  //DBContext
  var optionsBuilder = new DbContextOptionsBuilder<CommandContext>();
  optionsBuilder.UseInMemoryDatabase("UnitTestInMemBD");
  var dbContext = new CommandContext(optionsBuilder.Options);

  //IHostingEnvironment
  var mockEnvironment = new Mock<IHostingEnvironment>();
  mockEnvironment.Setup(m => m.EnvironmentName).Returns("UnitTest");

  //Controller
  var controller = new CommandsController(dbContext, mockEnvironment.Object);

  //Act
  var result = controller.GetCommandItems();

  //Assert
  Assert.Empty(result.Value);

}
```

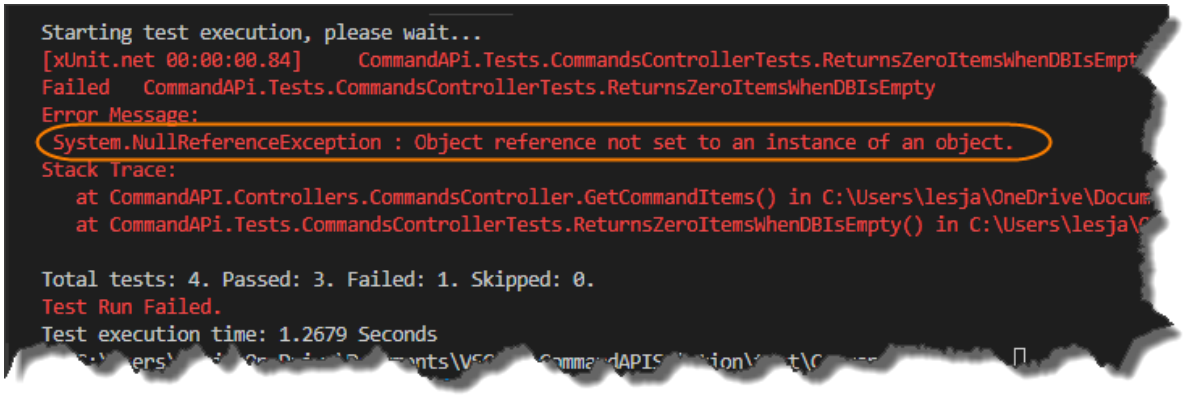
Act

- We simply call the `GetCommandItems()` method of our created controller and store the result

Assert

- We assert that the result Value is empty (as it should be)

Back to your terminal and run the tests… Oh dear…



We are getting the dreaded `NullReferenceException`!

Looking back at out controller, the "offending" object is highlighted:

Because we are calling the `GetCommandItems()` method directly on the `CommandsController` class, (and not via a HTTP GET request), there is no `Request` object, and by extension there is no `Response` object. We could call the Action Result this way, but for me this is tending away from a unit test and towards an integration test. From Microsoft, on testing controllers:

"Unit tests involve testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action are tested, not the behaviour of its dependencies or of the framework itself."

So to negate this error, we'll update our ActionResult to check if we have a `Response` object before we attempt to use it, the code looks like this:

```
//GET:    api/commands
[HttpGet]
public ActionResult<IEnumerable<Command>> GetCommandItems()
{
 if(Response != null)
   Response.Headers.Add("Environment", _hostEnv.EnvironmentName);

 return _context.CommandItems;
}
```

For clarity we just added the highlighted code:

```
//GET:      api/commands
[HttpGet]
public ActionResult<IEnumerable<Command>> GetCommandItems()
{
    if(Response != null)
        Response.Headers.Add("Environment", _hostEnv.EnvironmentName);

    return _context.CommandItems;
}
```

Run your tests again – this time they should all pass!

Finishing GetCommandItems Tests

So looking back at the unit tests we wanted to perform on our first controller `ActionResult` we said:

Test ID	Condition	Expected Result
~~**Test 1.1**~~	~~Request Objects when none exist~~	~~Return "nothing"~~
Test 1.2	Request Objects when 1 exists	Return Single Object
Test 1.3	Request Objects when *n* exist	Return Count of *n* Objects
Test 1.4	Request Objects	Return the correct "type"

We have just completed the 1st one and need to move on with the rest. Before we do though we're going to encounter a similar situation where we want to set up the same stuff in preparation for the tests, so let's refactor our existing, single controller unit test to support this. The code is shown below:

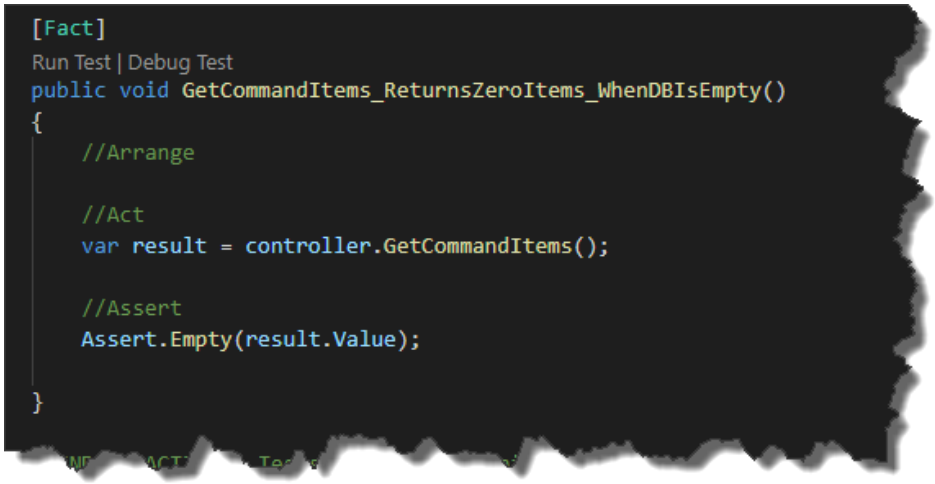
```
using System;
using System.Linq;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using CommandAPI.Controllers;
using CommandAPI.Models;
using Moq;
using Xunit;

namespace CommandAPI.Tests
{
  public class CommandsControllerTests : IDisposable
  {
    DbContextOptionsBuilder<CommandContext> optionsBuilder;
    CommandContext dbContext;
    Mock<IHostingEnvironment> mockEnvironment;
    CommandsController controller;

    public CommandsControllerTests()
    {
      optionsBuilder = new DbContextOptionsBuilder<CommandContext>();
      optionsBuilder.UseInMemoryDatabase("UnitTestInMemBD");
      dbContext = new CommandContext(optionsBuilder.Options);
      mockEnvironment = new Mock<IHostingEnvironment>();
```

```
      mockEnvironment.Setup(m => m.EnvironmentName).Returns("UnitTest");
      controller = new CommandsController(dbContext,
      mockEnvironment.Object);
    }

    public void Dispose()
    {
      optionsBuilder = null;
      foreach (var cmd in dbContext.CommandItems)
      {
        dbContext.CommandItems.Remove(cmd);
      }
      dbContext.SaveChanges();
      dbContext.Dispose();
      mockEnvironment = null;
      controller = null;
    }

    //ACTION 1 Tests: GET       /api/commands

    //TEST 1.1 REQUEST OBJECTS WHEN NONE EXIST - RETURN "NOTHING"
    [Fact]
    public void GetCommandItems_ReturnsZeroItems_WhenDBIsEmpty()
    {
      //Arrange

      //Act
      var result = controller.GetCommandItems();

      //Assert
      Assert.Empty(result.Value);

    }
  }
}
```

For clarity these are the changes with some explanations:

1. We've added some additional using directives, Linq is of particular use when we need to obtain a "count" our returned objects.
2. Our test class inherits from `IDisposable`
3. We set up our "global" objects at a class level, (not local), therefore we can't use "var", instead opting for the concrete types
4. We add a class constructor that set's up our objects
5. We have Dispose method where we clean up, ready for the next test. This includes having to remove any objects from `CommandItems` collection.

You'll also note that our single unit test is looking much cleaner!



```
[Fact]
Run Test | Debug Test
public void GetCommandItems_ReturnsZeroItems_WhenDBIsEmpty()
{
    //Arrange

    //Act
    var result = controller.GetCommandItems();

    //Assert
    Assert.Empty(result.Value);

}
```

Ok so even though we have 3 more tests to write for this `ActionResult`, because of all the work we have now done, these, (and the remaining tests for the other controller actions), should be quick!

Test 1.2: Returning a Count of 1 for a Single Command Object

The code for this is quite simple now that the majority of "arranging" is out of the way:

```
[Fact]
public void GetCommandItemsReturnsOneItemWhenDBHasOneObject()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };

  dbContext.CommandItems.Add(command);
  dbContext.SaveChanges();

  //Act
  var result = controller.GetCommandItems();

  //Assert
  Assert.Single(result.Value);
}
```

Here we:

- Create a test `Command` object and add it to our `CommandItems` collection
- Note that we have to `SaveChanges()` otherwise the change will not be reflected
- Call the controller as before
- Use the `Assert.Single` assertion to determine if we have 1 result

Now you could write the assertion as:

```
Assert.Equal(1, result.Value.Count());
```

But xUnit complains if you do! If you are only expecting a single result, it recommends you use the syntax we have in our code…

Test 3: Returning a Count of n for n Command Objects

Here we check for n number of objects being returned, we'll just use 2… The code is as follows:

```
[Fact]
public void GetCommandItemsReturnNItemsWhenDBHasNObjects()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };
  var command2 = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };
  dbContext.CommandItems.Add(command);
  dbContext.CommandItems.Add(command2);
  dbContext.SaveChanges();

  //Act
  var result = controller.GetCommandItems();

  //Assert
  Assert.Equal(2, result.Value.Count());
}
```

This is pretty much the same as our last test but we use the `Assert.Equal` assertion this time because we have multiple items being returned.

Test 4: Returns The Expected Type

Our last test is simple, it just tests to see we are getting the expected type back, the code is as follows:

```
[Fact]
public void GetCommandItemsReturnsTheCorrectType()
{
  //Arrange

  //Act
  var result = controller.GetCommandItems();

  //Assert
  Assert.IsType<ActionResult<IEnumerable<Command>>>(result);
}
```

Run all your tests and we should have 7 lovely passing tests!

Celebration Check Point: Give yourself a "pat on the back", this is a major milestone in that we have written a number of unit tests for our 1st controller action. The remainder of this chapter should be a breeze!

Test Driven Development

What Is Test Driven Development

So far we have taken the approach where we have written code, then written unit tests to test that code – seems ok? Indeed it is, it's a perfectly valid way to do things.

Test Driven Development, (TDD), reverses this approach though, where you write the unit tests first, then write the code to ensure that the test passes, (when you write the tests first and run them, they will obviously fail).

At first this can seem like a strange way to do things but when you think about it, software is usually built against requirements, so building them against tests isn't that different.

Test Driven Development grew out of the "Agile" approach to software delivery, where a decomposed, iterative approach is taken to software development, (i.e. we design, develop, test and deploy small, valuable software features – iteratively).

This paradigm lends itself more closely to TDD, (as opposed to more traditional "Waterfall" methods).

Why Would You Use TDD?

We've already talked about the advantages of unit testing, as well as the characteristics of a good unit test. So what exactly does TDD bring to the table on top of that? Or more specifically, what advantages are there in writing your tests *before* the code, (as opposed to after?).

Great question!

The fundamental benefits of unit testing remain the same irrespective of when you write them, (just writing them is the main thing!), but writing them first and putting them front of mind will generally mean that:

- You'll have more unit tests than you otherwise would, which leads to:
  - Greater protection against regression defects
  - More comprehensive "documentation"
- You'll tend to think more about the design of your code, which should lead to:
  - Less decoupling
- Closer alignment to the requirements / acceptance criteria
- You can see what code you still have to write, (you'll have failing tests otherwise!)

I've decided to finish up this chapter, and the development of our API using this approach, mainly just to introduce you to the concept. The main takeaway from this chapter should however be the *benefits of unit testing generally*, whether you eventually adopt TDD practices is up to you!

Revisit Our REST Actions

Looking back at Chapter 3 and the API actions we want to implement, we have:

Verb	URI	Operation	Description
GET	/api/commands	Read	Read all command resources
GET	/api/commands/{Id}	Read	Read a single resource, (by Id)
POST	/api/commands	Create	Create a new resource
PUT	/api/commands/{Id}	Update	Update a single resource, (by Id)
DELETE	/api/commands/{Id}	Delete	Delete a single resource, (by Id)

We have already written the first action, and associated unit tests, so the rest of this chapter will be working through each remaining actions and:

- Determining what unit tests to write to cover the required functionality
- Write the controller action to make the tests pass, (and provide that functionality)

So what are we waiting for?

Action 2: Get A Single Resource

We need to implement the following action via TDD practices:

Verb	URI	Operation	Description
GET	/api/commands/{Id}	Read	Read a single resource, (by Id)

What Should We Test

This action is ultimately about returning a single resource based on a unique Id, so we should test the following:

Test ID	Condition	Expected Result
Test 2.1	Resource ID is invalid (Does not exist in DB)	Null Object Value Result
Test 2.2	Resource ID is invalid (Does not exist in DB)	404 Not Found Return Code
Test 2.3	Resource ID is valid (Exists in the DB)	Correct Return Type
Test 2.4	Resource ID is valid (Exists in the DB)	Correct Resource Returned

Test 2.1 Invalid Resource ID – Null Object Value Result

The code for this test is presented below, place it in the `CommandsControllerTests` class:

```
[Fact]
public void GetCommandItemReturnsNullResultWhenInvalidID()
{
  //Arrange
  //DB should be empty, any ID will be invalid

  //Act
  var result = controller.GetCommandItem(0);

  //Assert
  Assert.Null(result.Value);
}
```

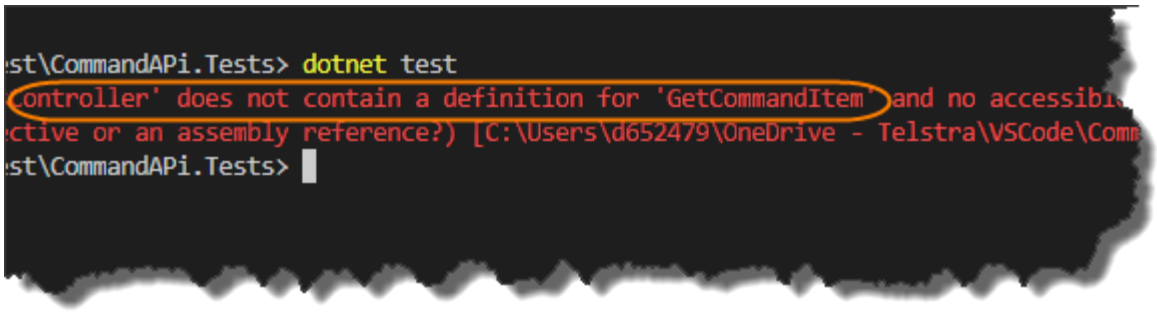
You can run all your tests again, or you can run selective tests as so:

```
dotnet test --filter DisplayName=
CommandAPI.Tests.CommandsControllerTests.ReturnsNullResultWhenInvalidID
```

Where you use the `-fillter` switch to provide the criteria to select what tests you want to run. Personally though, unless you have 1000's of tests that take a while to execute, this syntax is too clumsy for me to bother with. I also like the fact that all tests run, gives added confidence that you're not breaking anything.

If you want to find out more though, this article explains the concept further.

Irrespective you'll get an error:



For me there are 2 broad types of failure:

1. **Syntax / coding errors** that causes the test to fail, (such as the error we're getting above). These are useful for proving out code coverage considerations as well as the "correctness" of your code..
2. **Test / Assertion Failures.** The tests run "successfully", but the expected result, (the Assertion), returns false. These are more useful for proving out logic.

Our failure is due to the fact we have not yet coded up controller action, so of course it'll fail. So move over to the `CommandsContoller` class and add the following action:

```
//GET:      api/commands/Id
[HttpGet("{id}")]
public ActionResult<Command> GetCommandItem(int id)
{
  var commandItem = _context.CommandItems.Find(id);

  return commandItem;
}
```

The method is quite simple:

- It's decorated with another `[HttpGet]` attribute with an additional `{id}` element, (this will be the id passed through in the request).
- The method itself expects an `int id` as input (mapped through from the request)
- The method is expected to return a single `Command` Object (as opposed to a collection as supplied by our first action: `GetCommandItems`)
- Using the `id` we then find the required object in `CommandItems`, and return the `commandItem` (if any)

Save your code changes, then re-run your tests – success!

```
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...

Total tests: 8. Passed: 8. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 3.7111 Seconds
PS C:\Users\d652479\OneDrive - Telstra\VSCode\CommandAPI\test\CommandAPi.Tests>
```

TEST 2.2 INVALID RESOURCE ID – 404 NOT FOUND RETURN CODE

The code for this test is presented below, (back in `CommandsControllerTests` class):

```
[Fact]
public void GetCommandItemReturns404NotFoundWhenInvalidID()
{
  //Arrange
  //DB should be empty, any ID will be invalid

  //Act
  var result = controller.GetCommandItem(0);

  //Assert
  Assert.IsType<NotFoundResult>(result.Result);
}
```

Save the ***CommandsControllerTests.cs*** file, run your Tests, and…

```
Starting test execution, please wait...
[xUnit.net 00:00:02.57]     CommandAPi.Tests.CommandsControllerTests.Returns404NotFo
Failed   CommandAPi.Tests.CommandsControllerTests.Returns404NotFoundWhenInvalidID
Error Message:
 Assert.IsType() Failure
Expected: Microsoft.AspNetCore.Mvc.NotFoundResult
Actual:   (null)
Stack Trace:
   at CommandAPi.Tests.CommandsControllerTests.Returns404NotFoundWhenInvalidID() in

Total tests: 9. Passed: 8. Failed: 1. Skipped: 0.
Test Run Failed.
```

This is a prime example of the "2nd Type" of test failure, which for some reason I find more pleasing! The code is "correct" but the functionality we have specified in our test is not being satisfied, (btw returning a 404 Not Found is best practice when a single resource is not available, and therefore worth testing).

We need to turn this test green, so update your `GetCommandItem` in the `CommandsController` class with the following update:

```
//GET:      api/commands/Id
[HttpGet("{id}")]
```

```
public ActionResult<Command> GetCommandItem(int id)
{
  var commandItem = _context.CommandItems.Find(id);

  if(commandItem == null)
    return NotFound();

  return commandItem;
}
```

Here we perform an additional check to see if a `commandItem` has not been found, if so we return a `NotFound` result.

Save your code and re-run your tests and we should be good to go!

This is a good example of the process and power of TDD. We wrote our first test, it failed. We wrote the code to make it pass, it passed… We wrote our 2nd test, it failed. We added to the code to make it pass, it passed. Repeat and rinse for all your tests – so at the end you should have well-formed code that has decent unit test coverage.

However, the rest of this chapter would get *really long* if I were to document this approach for every action and all our tests, remember – no filler! For me this would be tending into filler-territory and I don't want to go there.

For the rest of the chapter I'll specify *all* the tests, then the *full code* for the corresponding action to ensure all the tests pass. I won't necessarily specify all the iterative steps you'd take to get to that end state as we did for the last example.

TEST 2.3 VALID RESOURCE ID – CHECK CORRECT RETURN TYPE

The code for this test is presented below, (place this in `CommandsControlerTests`):

```
[Fact]
public void GetCommandItemReturnsTheCorrectType()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };

  dbContext.CommandItems.Add(command);
  dbContext.SaveChanges();

  var cmdId = command.Id;

  //Act
  var result = controller.GetCommandItem(cmdId);

  //Assert
  Assert.IsType<ActionResult<Command>>(result);
}
```

Here to get a valid "Id", we need to create an object in our `CommandItems` and retrieve the id for use in our *Act*.

Running this test, our controller action passes without further alteration.

Test 2.4 Valid Resource ID – Correct Resource Returned

The code for this test is presented below, (place this in `CommandsControlerTests`):

```
[Fact]
public void GetCommandItemReturnsTheCorrectResouce()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };

  dbContext.CommandItems.Add(command);
  dbContext.SaveChanges();

  var cmdId = command.Id;

  //Act
  var result = controller.GetCommandItem(cmdId);

  //Assert
  Assert.Equal(cmdId, result.Value.Id);
}
```

In this test we use the Id from the object we created in the *Arrange* section to make our call to `GetCommandItem`, our assertion then checks that the Id's are the same, hence we have retrieved the correct object.

Running this test, our controller action passes without further alteration.

Celebration Check Point: Congratulations! You have just used TDD to develop our second controller action!

Learning Opportunity: Why don't you use Postman to check the results for yourself.

Hint: Remember when calling the action via a URL, you don't use the method name in the controller but the *route pattern*, so the URL you use to call this action is exactly the same as the one we used previously apart from one very important additional item…

Action 3: Create a New Resource

We need to implement the following action via TDD practices:

Verb	URI	Operation	Description
POST	/api/commands	Create	Create a new resource

What Should We Test

This action is ultimately about creating resource in our database, so we should test the following:

Test ID	Condition	Expected Result
Test 3.1	Valid Object Submitted for Creation	Object count increments by 1
Test 3.2	Valid Object Submitted for Creation	201 Created Return Code

Note: We'll specify all the unit test code first, followed by the final controller action code.

Test 3.1 Valid Object Submitted – Object Count Increments by 1

The code for this test is presented below, place it in the `CommandsControllerTests` class:

```
[Fact]
public void PostCommandItemObjectCountIncrementWhenValidObject()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };
  var oldCount = dbContext.CommandItems.Count();

  //Act
  var result = controller.PostCommandItem(command);

  //Assert
  Assert.Equal(oldCount + 1, dbContext.CommandItems.Count());
}
```

Test 3.2 Valid Object Submitted – 201 Created Return Code

The code for this test is presented below, place it in the `CommandsControllerTests` class:

```
[Fact]
public void PostCommandItemReturns201CreatedWhenValidObject()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };

  //Act
  var result = controller.PostCommandItem(command);

  //Assert
  Assert.IsType<CreatedAtActionResult>(result.Result);
}
```

The action we need to create in `CommandsController` to get these tests to pass:

```
//POST:     api/commands
[HttpPost]
public ActionResult<Command> PostCommandItem(Command command)
{
  _context.CommandItems.Add(command);

  try
  {
    _context.SaveChanges();
```

```
    }
    catch
    {
      return BadRequest();
    }

    return CreatedAtAction("GetCommandItem", new Command{Id = command.Id}, command);
}
```

Action 4: Update an Existing Resource

We need to implement the following action via TDD practices:

Verb	URI	Operation	Description
PUT	/api/commands/{Id}	Update	Update a single resource, (by Id)

What Should We Test

This action is about updating a resource in our database, so we should test the following:

Test ID	Condition	Expected Result
Test 4.1	Valid Object Submitted for Update	Attribute is updated
Test 4.2	Valid Object Submitted for Update	204 No Content Return Code
Test 4.3	Invalid Object Submitted for Update	400 Bad Request Return Code
Test 4.4	Invalid Object Submitted for Update	Object remains unchanged

Note: We'll specify all the unit test code first, followed by the final controller action code.

Test 4.1 Valid Object Submitted – Attribute is Updated

The code for this test is presented below, place it in the `CommandsControllerTests` class:

```
[Fact]
public void PutCommandItem_AttributeUpdated_WhenValidObject()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };
  dbContext.CommandItems.Add(command);
  dbContext.SaveChanges();

  var cmdId = command.Id;

  command.HowTo = "UPDATED";

  //Act
  controller.PutCommandItem(cmdId, command);
  var result = dbContext.CommandItems.Find(cmdId);

  //Assert
  Assert.Equal(command.HowTo, result.HowTo);
}
```

Test 4.2 Valid Object Submitted – 204 Return Code

The code for this test is presented below, place it in the `CommandsControllerTests` class:

```
[Fact]
public void PutCommandItem_Returns204_WhenValidObject()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };
  dbContext.CommandItems.Add(command);
  dbContext.SaveChanges();

  var cmdId = command.Id;

  command.HowTo = "UPDATED";

  //Act
  var result = controller.PutCommandItem(cmdId, command);

  //Assert
  Assert.IsType<NoContentResult>(result);
}
```

TEST 4.3 INVALID OBJECT SUBMITTED – 400 RETURN CODE

The code for this test is presented below, place it in the `CommandsControllerTests` class:

```
[Fact]
public void PutCommandItem_Returns400_WhenInvalidObject()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };
  dbContext.CommandItems.Add(command);
  dbContext.SaveChanges();

  var cmdId = command.Id+1;

  command.HowTo = "UPDATED";

  //Act
  var result = controller.PutCommandItem(cmdId, command);

  //Assert
  Assert.IsType<BadRequestResult>(result);
}
```

TEST 4.4 INVALID OBJECT SUBMITTED – OBJECT REMAINS UNCHANGED

The code for this test is presented below, place it in the `CommandsControllerTests` class:

```
[Fact]
public void PutCommandItem_AttributeUnchanged_WhenInvalidObject()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
```

```
  };
  dbContext.CommandItems.Add(command);
  dbContext.SaveChanges();

  var command2 = new Command
  {
    Id = command.Id,
    HowTo = "UPDATED",
    Platform = "UPDATED",
    CommandLine = "UPDATED"
  };

  //Act
  controller.PutCommandItem(command.Id + 1, command2);
  var result = dbContext.CommandItems.Find(command.Id);

  //Assert
  Assert.Equal(command.HowTo, result.HowTo);
}
```

The action we need to create in `CommandsController` to get these tests to pass:

```
using Microsoft.EntityFrameworkCore;
.
.
.


//PUT:     api/commands/n
[HttpPut("{id}")]
public ActionResult PutCommandItem(int id, Command command)
{
  if (id != command.Id)
  {
    return BadRequest();
  }

  _context.Entry(command).State = EntityState.Modified;
  _context.SaveChanges();

  return NoContent();
}
```

Action 5: Delete an Existing Resource

We need to implement the following action via TDD practices:

Verb	URI	Operation	Description
DELETE	/api/commands/{Id}	Delete	Delete a single resource, (by Id)

What Should We Test

This action is about updating a resource in our database, so we should test the following:

Test ID	Condition	Expected Result
Test 4.1	Valid Object Id Submitted for Delete	Object Count Decrements by 1
Test 4.2	Valid Object Id Submitted for Delete	200 OK Return Code
Test 4.3	Invalid Object Id Submitted for Delete	400 Bad Request Return Code
Test 4.4	Invalid Object Id Submitted for Delete	Object count remains unchanged

Note: We'll specify all the unit test code first, followed by the final controller action code.

TEST 4.1 VALID OBJECT ID SUBMITTED – OBJECT COUNT DECREMENTS BY 1

The code for this test is presented below, place it in the `CommandsControllerTests` class:

```
[Fact]
public void DeleteCommandItem_ObjectsDecrement_WhenValidObjectID()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };
  dbContext.CommandItems.Add(command);
  dbContext.SaveChanges();

  var cmdId = command.Id;
  var objCount = dbContext.CommandItems.Count();

  //Act
  controller.DeleteCommandItem(cmdId);

  //Assert
  Assert.Equal(objCount-1, dbContext.CommandItems.Count());
}
```

TEST 4.2 VALID OBJECT ID SUBMITTED – 200 OK RETURN CODE

The code for this test is presented below, place it in the `CommandsControllerTests` class:

```
[Fact]
public void DeleteCommandItem_Returns200OK_WhenValidObjectID()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };
  dbContext.CommandItems.Add(command);
  dbContext.SaveChanges();

  var cmdId = command.Id;

  //Act
  var result = controller.DeleteCommandItem(cmdId);

  //Assert
  Assert.Null(result.Result);
}
```

TEST 4.3 INVALID OBJECT ID SUBMITTED – 404 NOT FOUND RETURN CODE

The code for this test is presented below, place it in the `CommandsControllerTests` class:

```
[Fact]
public void DeleteCommandItem_Returns404NotFound_WhenValidObjectID()
{
  //Arrange
```

```
  //Act
  var result = controller.DeleteCommandItem(-1);

  //Assert
  Assert.IsType<NotFoundResult>(result.Result);
}
```

TEST 4.4 VALID OBJECT ID SUBMITTED – OBJECT COUNT REMAINS UNCHANGED

The code for this test is presented below, place it in the `CommandsControllerTests` class:

```
[Fact]
public void DeleteCommandItem_ObjectCountNotDecremented_WhenValidObjectID()
{
  //Arrange
  var command = new Command
  {
    HowTo = "Do Somethting",
    Platform = "Some Platform",
    CommandLine = "Some Command"
  };
  dbContext.CommandItems.Add(command);
  dbContext.SaveChanges();

  var cmdId = command.Id;
  var objCount = dbContext.CommandItems.Count();

  //Act
  var result = controller.DeleteCommandItem(cmdId+1);

  //Assert
  Assert.Equal(objCount, dbContext.CommandItems.Count());
}
```

The action we need to create in `CommandsController` to get these tests to pass:

```
[HttpDelete("{id}")]
public ActionResult<Command> DeleteCommandItem(int id)
{
  var commandItem = _context.CommandItems.Find(id);

  if (commandItem == null)
    return NotFound();

  _context.CommandItems.Remove(commandItem);
  _context.SaveChanges();

  return commandItem;
}
```

WRAP IT UP

Phew! We covered a lot in this chapter, and to be honest we really only scraped the surface. Hopefully though you learned enough to start to get you up to speed on unit testing. Again, it's up to you how you choose to employ it, you don't need to follow TDD, but some form of unit testing is a requirement in modern software development.

CHAPTER 9 – THE CI/CD PIPELINE

CHAPTER SUMMARY

In this chapter we bring together what we've done so far: build activity, source control and unit testing and frame it within the context of Continuous Integration / Continuous Delivery (CI/CD).

WHEN DONE, YOU WILL

- Understand what CI/CD is
- Understand what a CI/CD Pipeline is
- Set Up Azure DevOps with GitHub to act as our CI/CD pipeline
- Automatically Build, Test and Package our API solution using Azure DevOps
- Prepare for Deployment to Azure

WHAT IS CI/CD?

To talk about CI/CD, is to talk about a pipeline of work", or if you prefer another analogy: a production line, where a product, (in this instance working software), is taken from is raw form, (code[19]), and gradually transformed into working software that's usable by the end users.

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).

CI/CD OR CI/CD?

Don't worry, the heading not an typo, (we'll come on to that in a minute)…

CI is easy, that stands for *Continuous Integration*. CI is the process of taking any code changes from 1 or more developers working on the same piece of software, and 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, (as you have already been doing with Git / GitHub).

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? We'll you'll hear people using the both the following terms in reference to CD: Continuous *Delivery*, and Continuous *Deployment*.

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.

[19] 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 book though, we'll use code as the start point of the journey.

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?

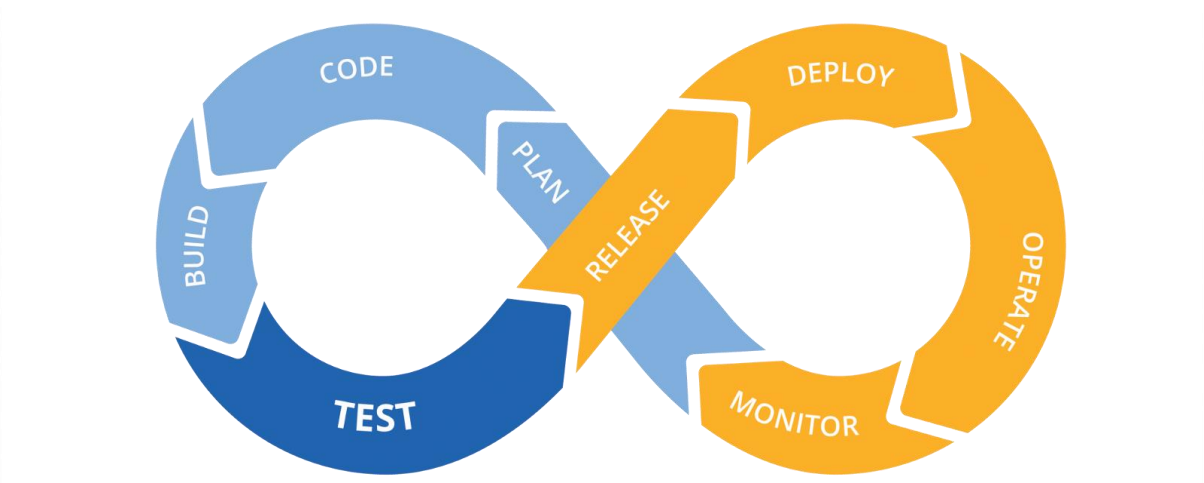
Typically, when we talk about CI/CD we talk about Continuous Integration & Continuous Delivery, although it can be dependent on the organization. Ultimately the decision to deploy software into production is a business decision, so the idea of Continuous Deployment is still overwhelming for most organizations….

In this book though we're going to go all out and practice full-on Continuous Deployment!

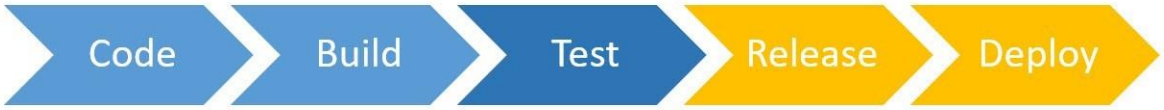
The Pipeline

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":



Coming back to the whole point of this chapter, (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:



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", so 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!)

In this chapter we are going to be focusing exclusively on the "pipeline" features it has to offer, and leave the other aspects untouched.

Alternatives

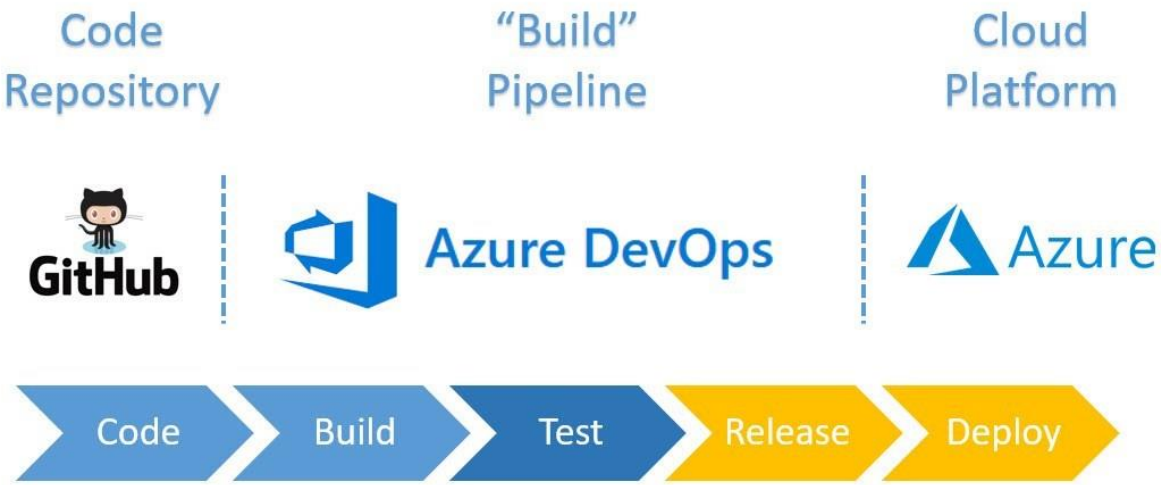
There are various on-premise and cloud-based alternatives: Jenkins is possibly the most "famous" of the on-premise solutions available, but you also have things like:

- 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!

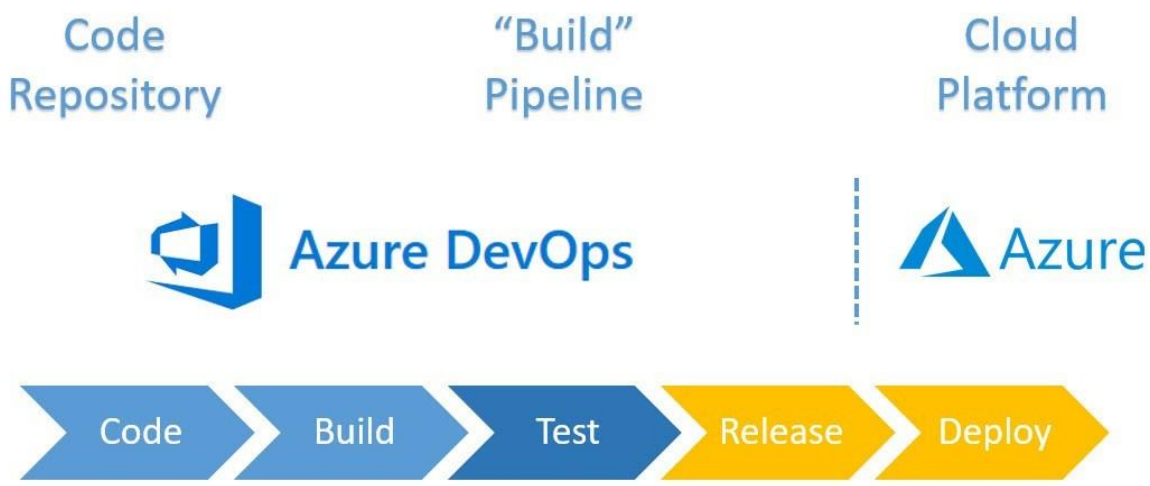
Technology In Context

Referring to our pipeline, in-terms of our technology overlay this is what we will be working with to build a CI/CD pipeline:

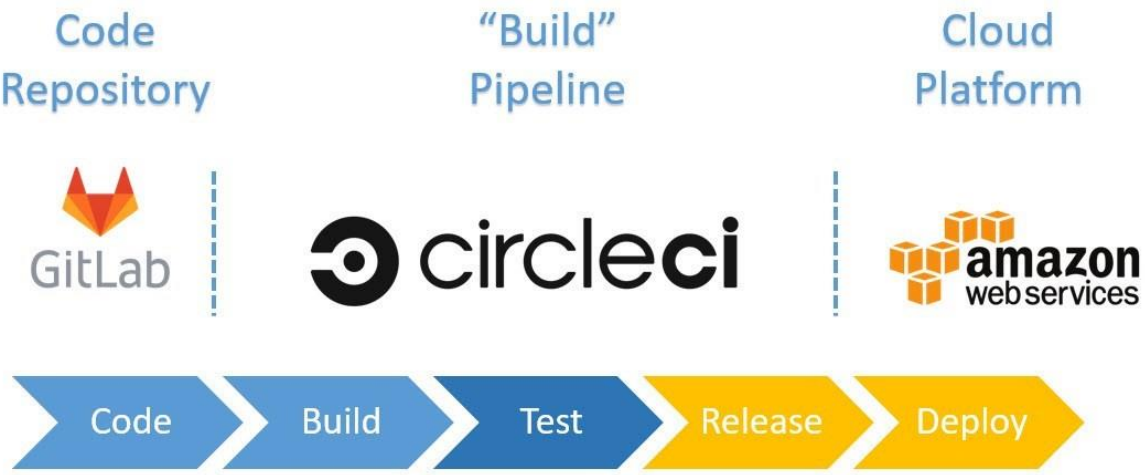


Indeed, Azure DevOps comes with its own "code repository" feature, (Azure Repos), which means we could do away with GitHub…

So our mix could look like:



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



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.

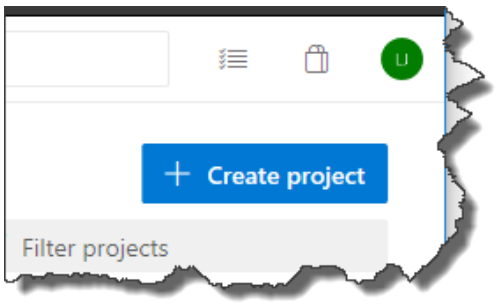
The takeaway points I wanted to make were:

1. The relevant sequencing of technologies in our example
2. Make sure you understand the importance of the code repository, (GitHub), as the start point
3. Be aware of the almost limitless choice of tech

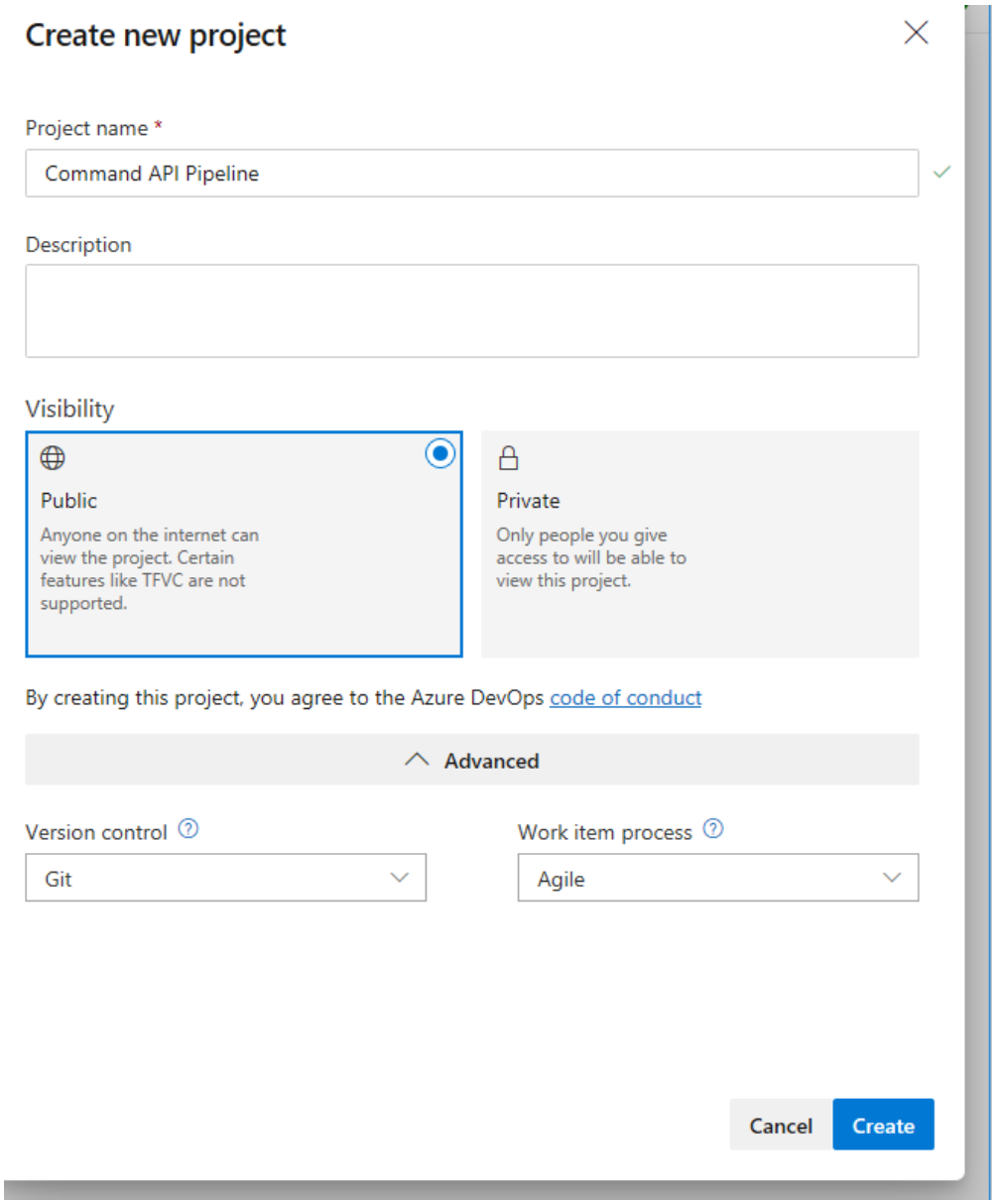
Ok enough theory, let's build our pipeline!

Create a Build Pipeline

If you've not done so already, go to the Azure DevOPs site: https://dev.azure.com and sign up for a free account. Once you have signed in / signed up, click on "Create Project":



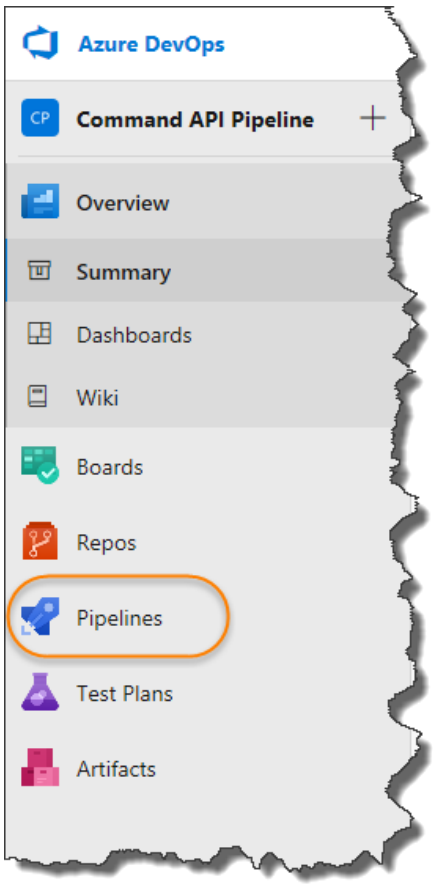
You can call it anything you like, so let's keep the theme going and call it *Command API Pipeline*:



Make sure:

- You select the same "visibility" setting that your GitHub repo has, (recommend Public for test projects)
- Version Control is set to Git – this is the default

Once you're happy – click "Create", this will create your project and take you into the landing page:

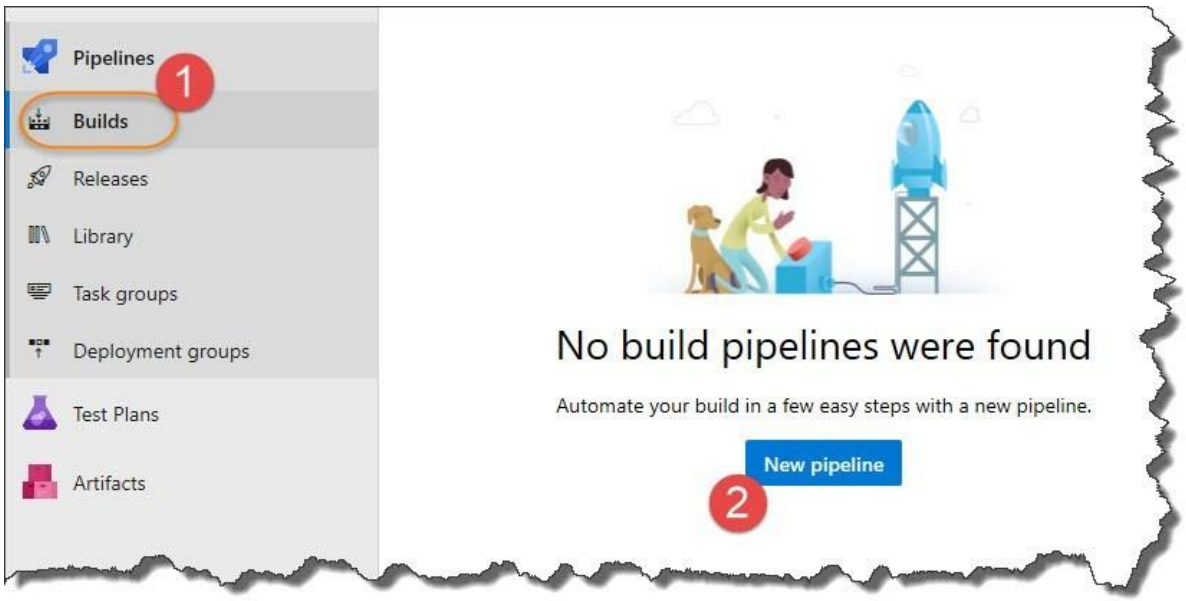


Les's Personal Anecdote: Followers of my Blog or YouTube channel will realise that the core of this chapter has been taken from a blog post & video I did on Azure DevOps. One of the things I noticed while making those was that the Azure DevOps product changes quite rapidly.

I basically had to go back and re-take some screenshots for my blog post and realised that some fundamental user interface changes had been made within the space of a few days - meaning I had to re-take all the screen shots, argh! Why am I tell you this? Well for the same reason… The screen shots that follow were correct at the time writing, (~June 2019), however they are subject to, (potentially rapid), change!

Azure DevOps has many features, but we'll just be using the "Pipelines" for now… Select Pipelines, then:

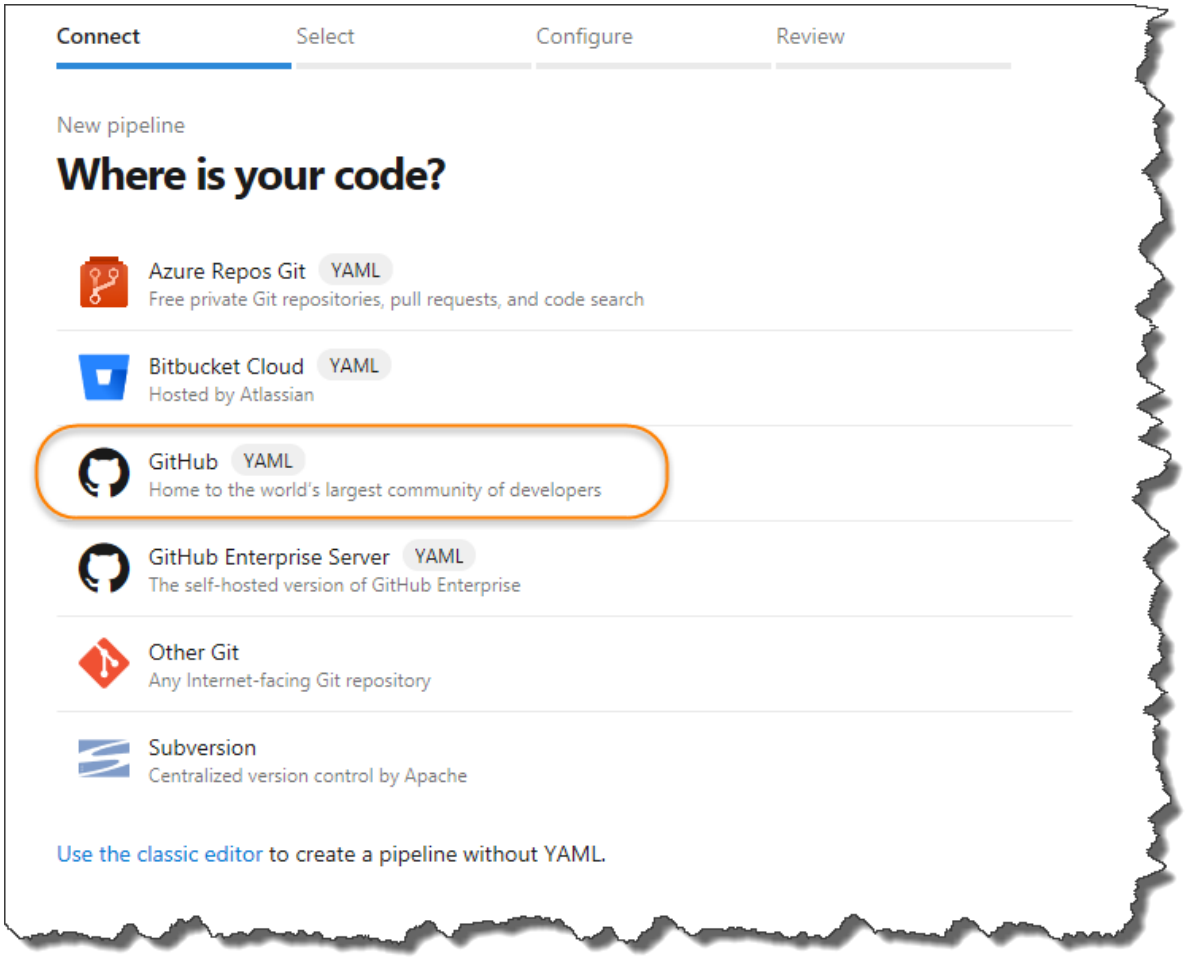
1. Builds
2. New pipeline:



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

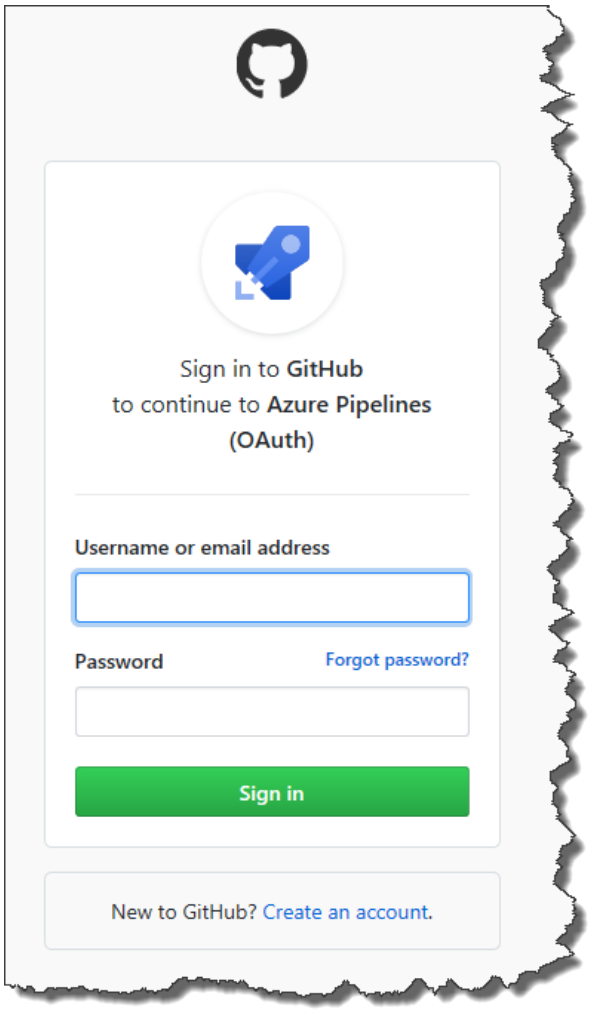
Well, where do you think?

Yeah – that’s right – in GitHub!

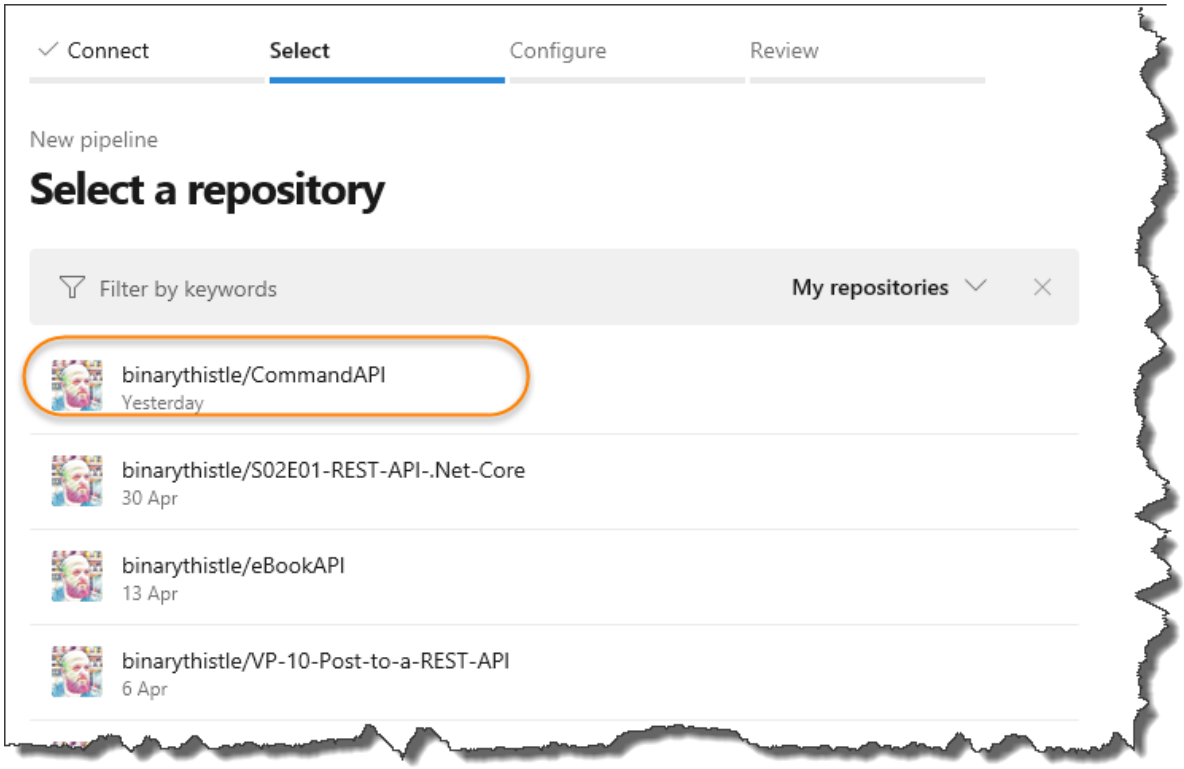


Be careful to select GitHub, as opposed to GitHub Enterprise Server, (which as the description states is the on-premise version of GitHub).

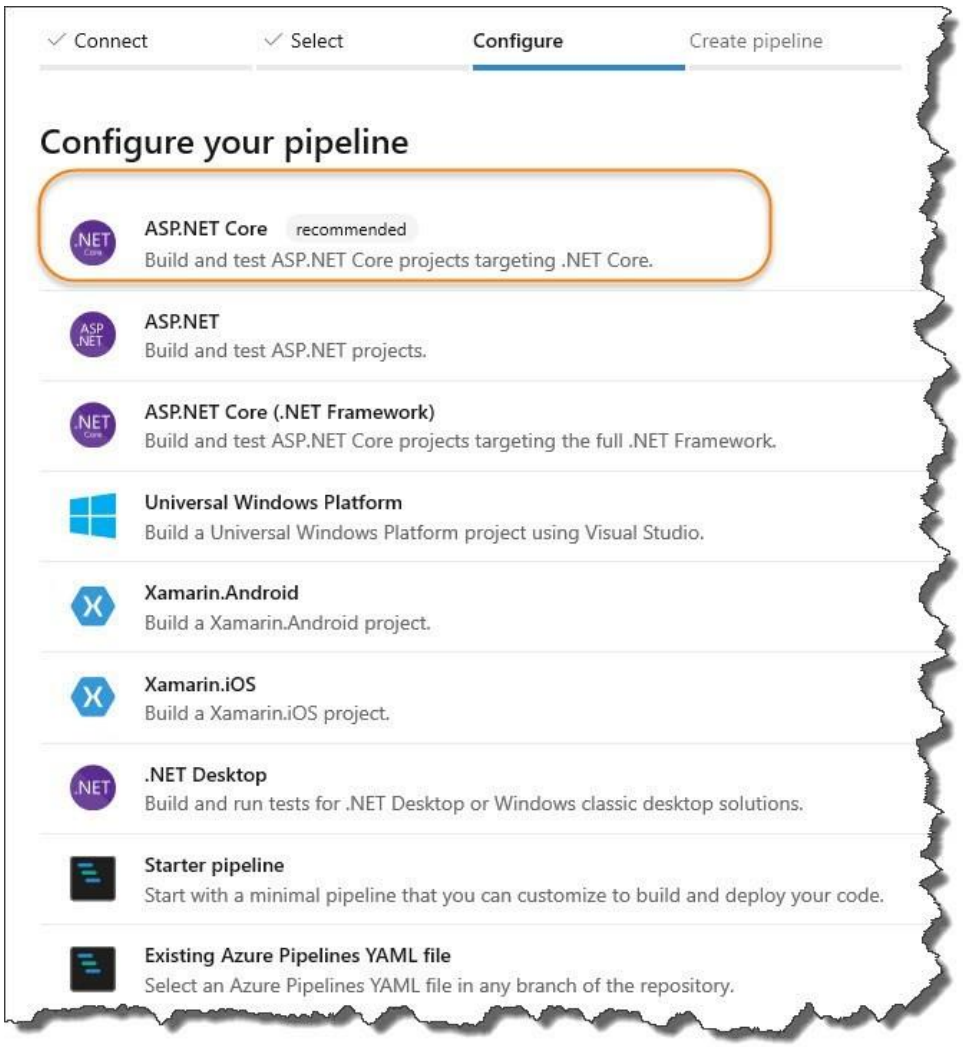
IMPORTANT: If this is the 1st time you're doing this, you'll need to give Azure DevOps permission to view your GitHub account:



Supply your GitHub account details and sign in. Once you've given Azure DevOps permission to connect to GitHub, you'll be presented with all your repositories:



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

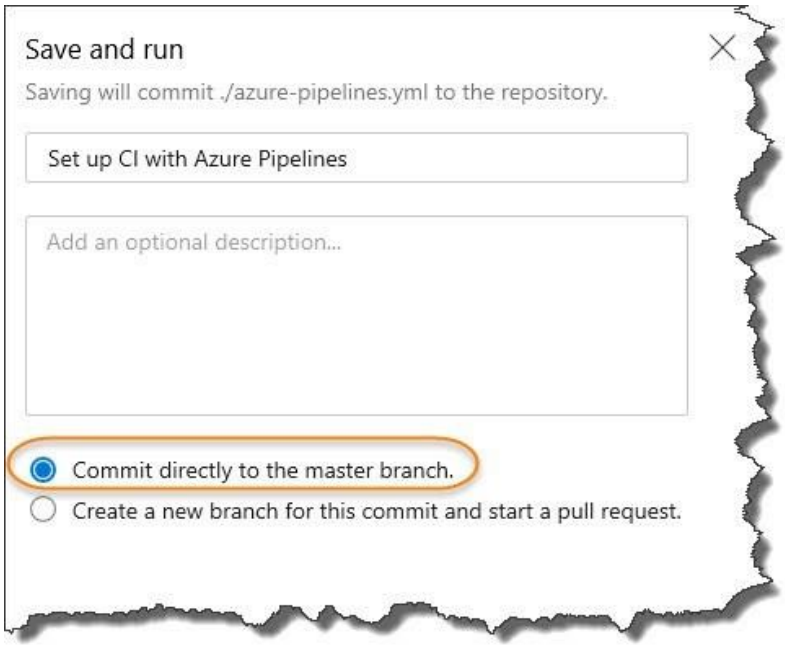


In this case go with the recommended pipeline template: ASP.NET Core, (select this if it doesn't recommend it), click it and you'll be presented with your pipeline YAML file:

azure-pipelines.yml

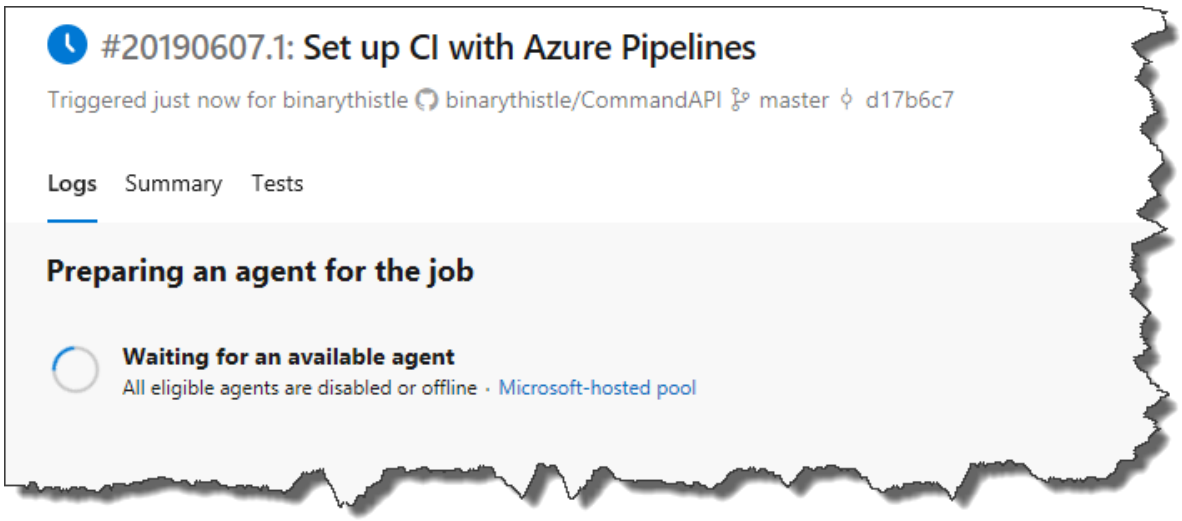
```yaml
# ASP.NET Core
# Build and test ASP.NET Core projects targeting .NET Core.
# Add steps that run tests, create a NuGet package, deploy, and more:
# https://docs.microsoft.com/azure/devops/pipelines/languages/dotnet-core

trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

variables:
  buildConfiguration: 'Release'

steps:
- script: dotnet build --configuration $(buildConfiguration)
  displayName: 'dotnet build $(buildConfiguration)'

```

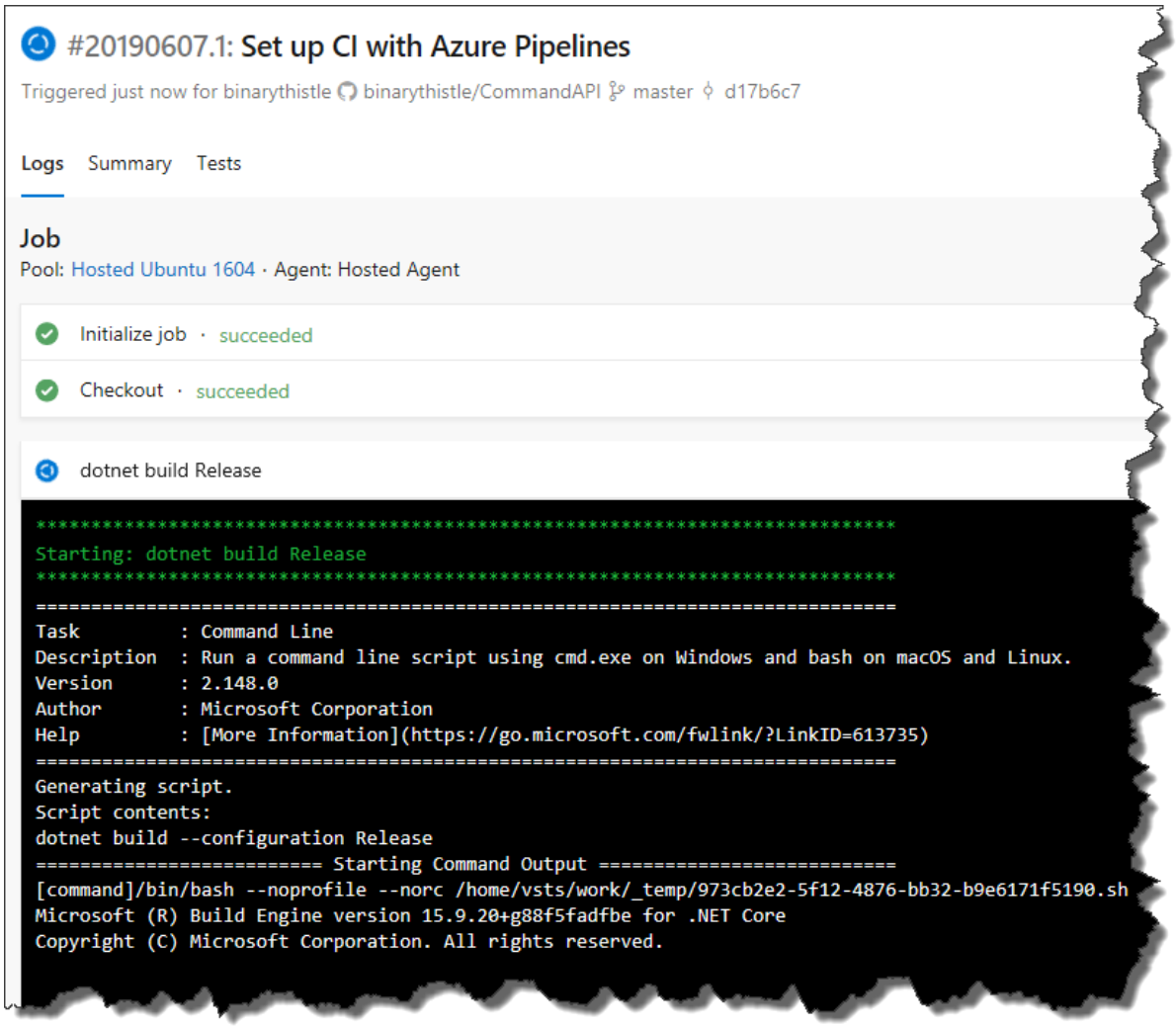
We'll go through this in detail later, suffice to say it's essentially a configurable script that dictates the steps you want to execute in your build pipeline. Click Save & Run



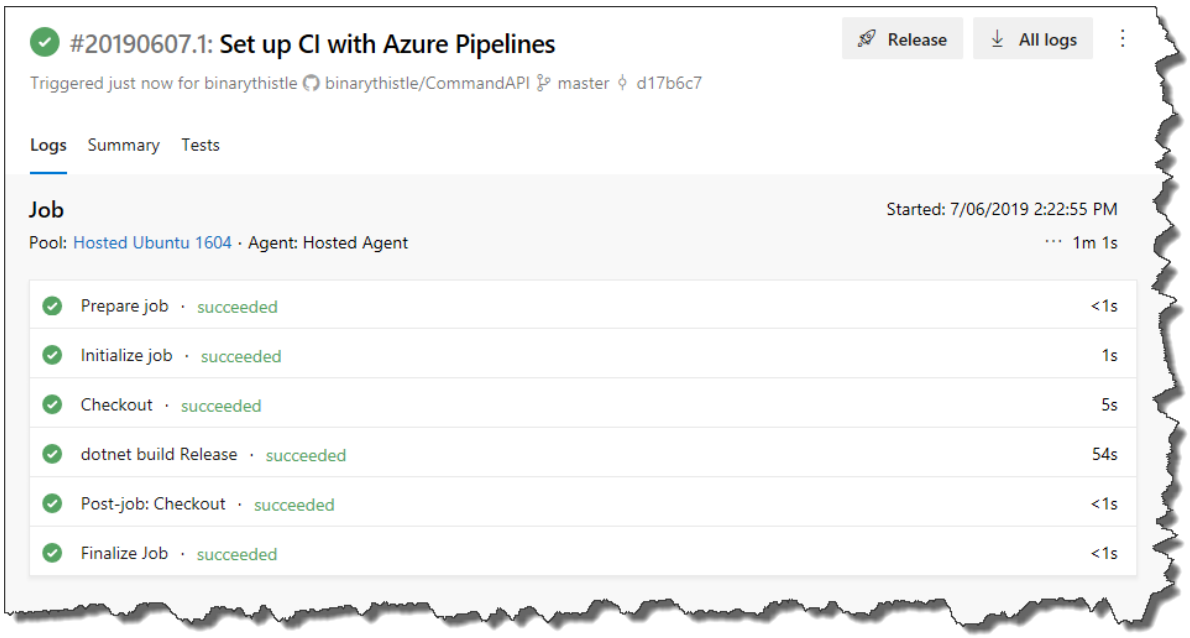
This is asking you where you want to store the ***azure-pipelines.yml*** file, in this case we want to add it directly to our GitHub repo, (remember this selection though as it comes back later!), so select this option and click Save & Run.

An "agent" is then assigned to execute the pipeline, you'll see various screens, such as:





And finally you should see the completion screen:



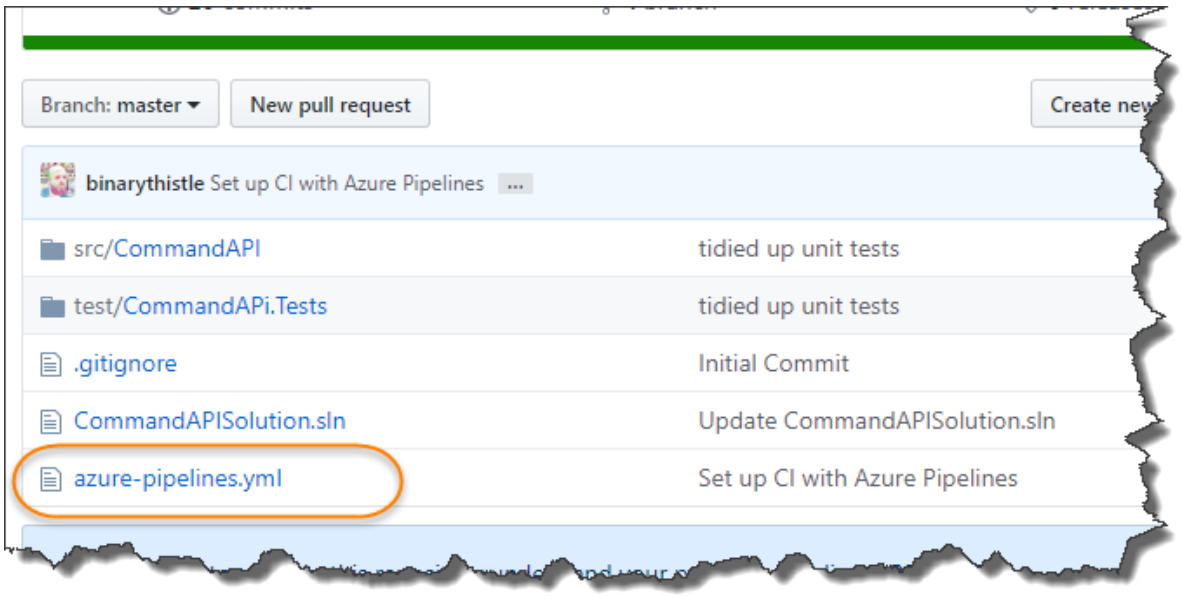
What Just Happened?

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

Let's pop back over to our GitHub repository and refresh – you should see the following:



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

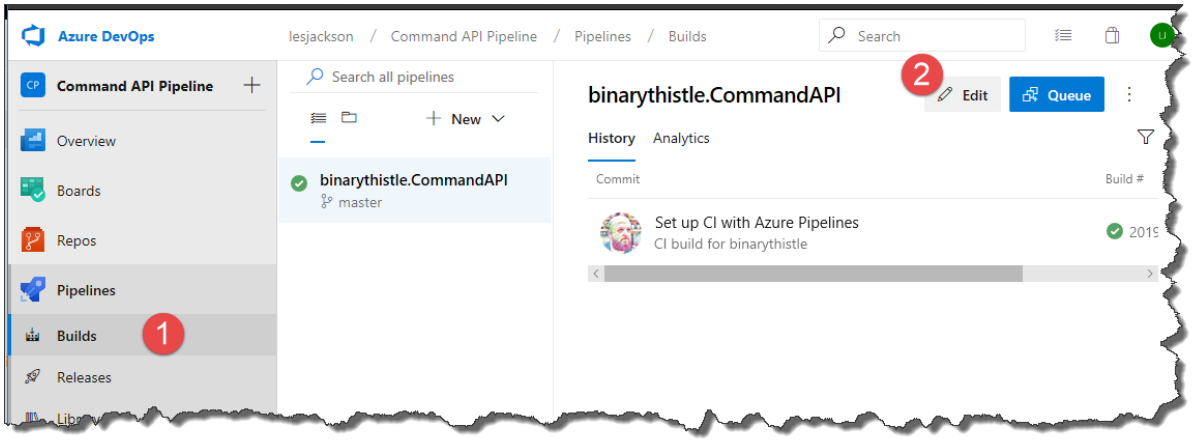
I THOUGHT WE WANTED TO AUTOMATE?

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!

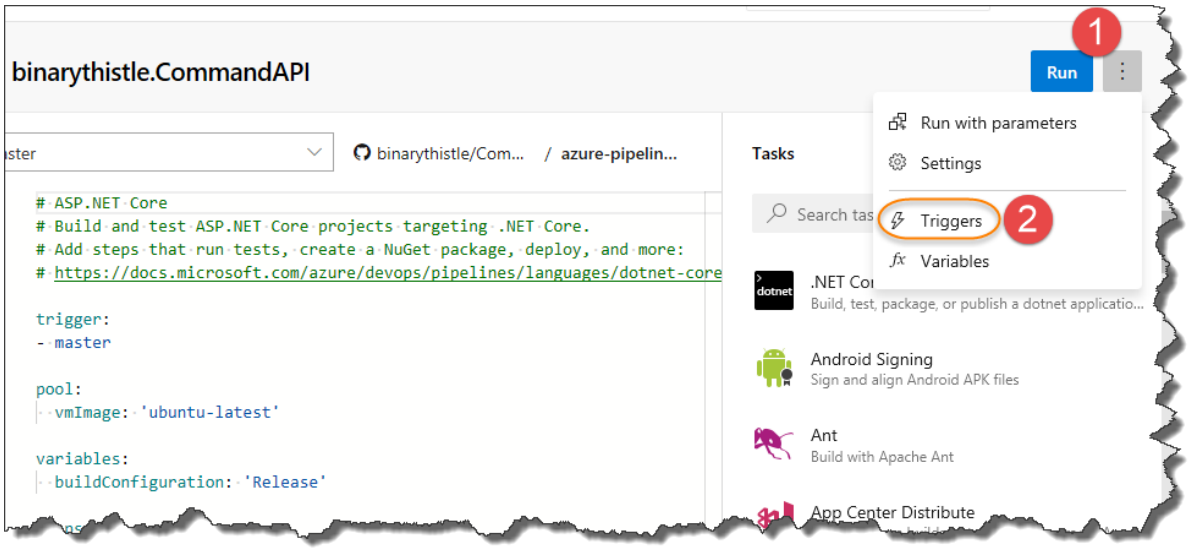
We are asked to execute when we created the pipeline that is true, but we can also set up "triggers", meaning we can configure the pipeline to execute when it receives a particular event…

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:

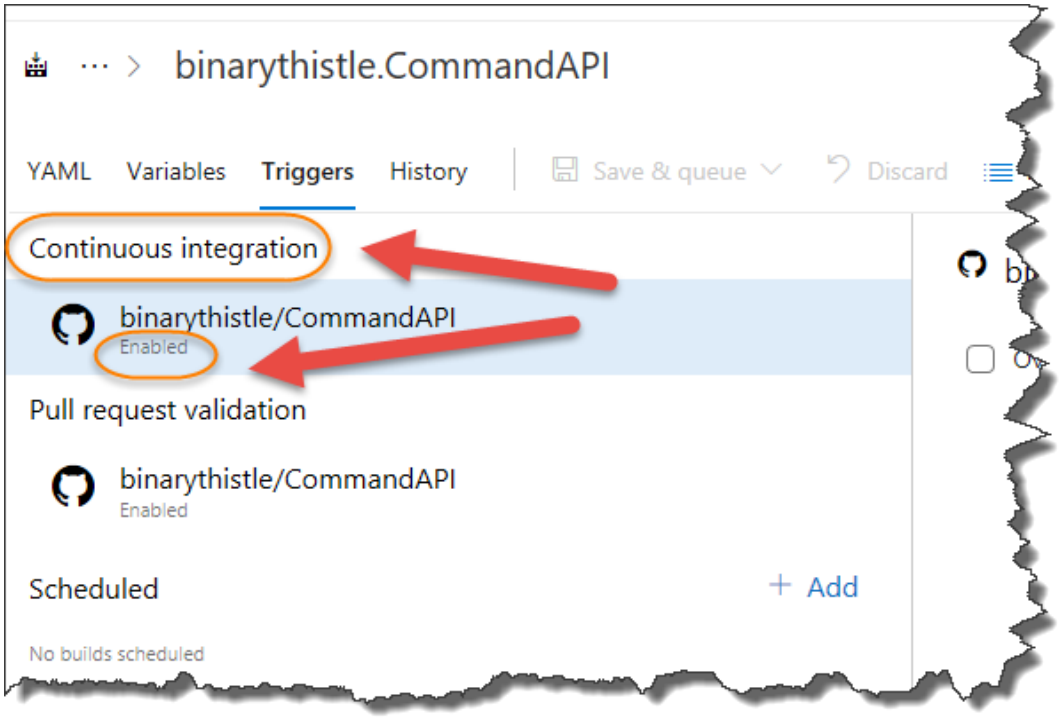


After doing that you should be returned to the ***azure-pipelines.yml*** file, (we will return here to edit it later),

1. Click the Ellipsis
2. Select Triggers



Below you can see the Continuous Integration, (CI), settings for our pipeline:



You can see that the automation trigger is enabled by default, so now let's trigger a build! But how do we do that?

Triggering a Build

Triggering a build starts with a `git push origin master` to GitHub, so really any code change, (including something trivial like adding or editing a comment), will suffice. With that in mind, back in VS Code open `CommandsController` class in the "main" ***CommandAPI*** project and put a comment in our `GetCommandItems` method, reminding us to remove this code once we move to production:

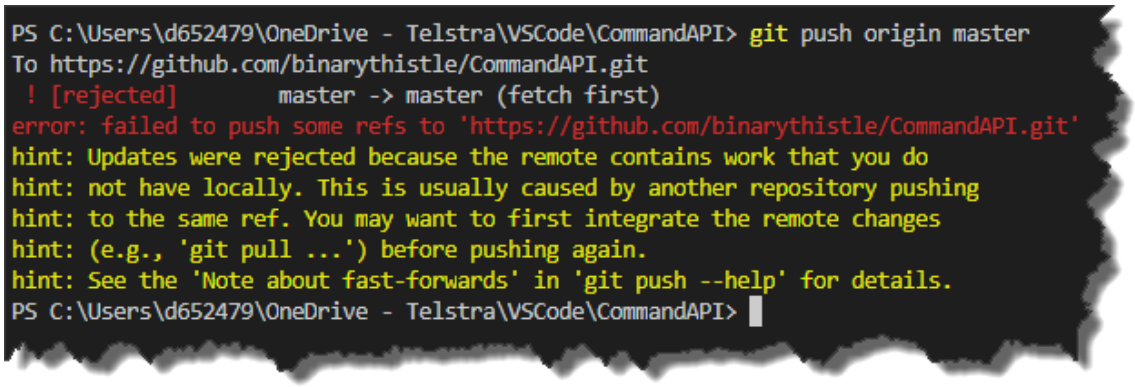
```
//GET:      api/commands
[HttpGet]
public ActionResult<IEnumerable<Command>> GetCommandItems()
{
    //We'll remove after moving to production
    if(Response != null)
        Response.Headers.Add("Environment", _hostEnv.EnvironmentName);

    return _context.CommandItems;
}
```

Save the file and perform the usual sequence of actions:

- `git add .`
- `git commit -m "Added a reminder to clean up code"`

- `git push origin master`

Everything should go as planned except when it comes to executing the final `push` command:

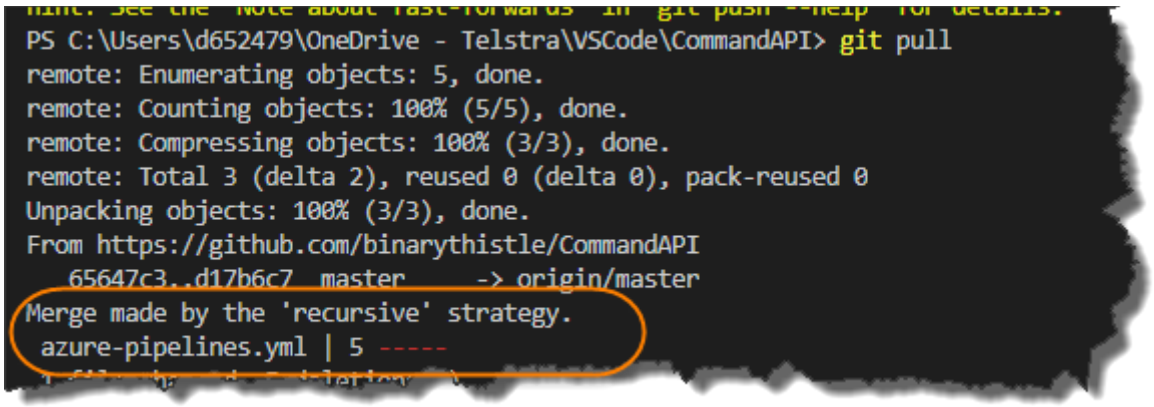


What does this mean?

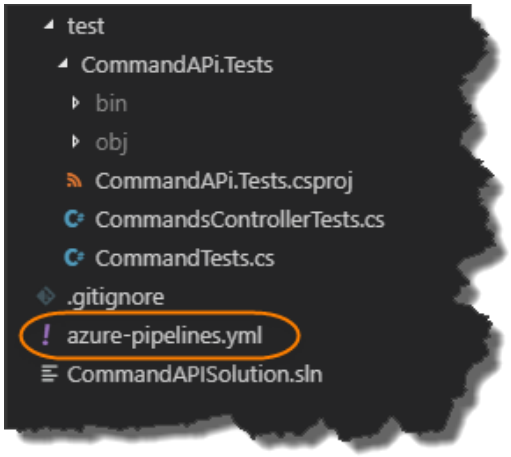
Well remember we added the ***azure-pipelines.yml*** file to the GitHub repo? Yes? Well that's the cause, essentially the local repository and the remote GitHub repository are out of sync, (the central GitHub repo has some newer changes than our local repository). To remedy this, we simply type:

```
git pull
```

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



Indeed if you look the VS Code file tree, you'll see our ***azure-pipelines.yml*** file has appeared!



Now we have synced our repositories, you can now attempt to push our combined local Git repo back up to GitHub, (this includes the comment we inserted into our `CommandsController` class). Quickly jump over to Azure DevOps and click on Pipelines -> Builds, you should see something like this:

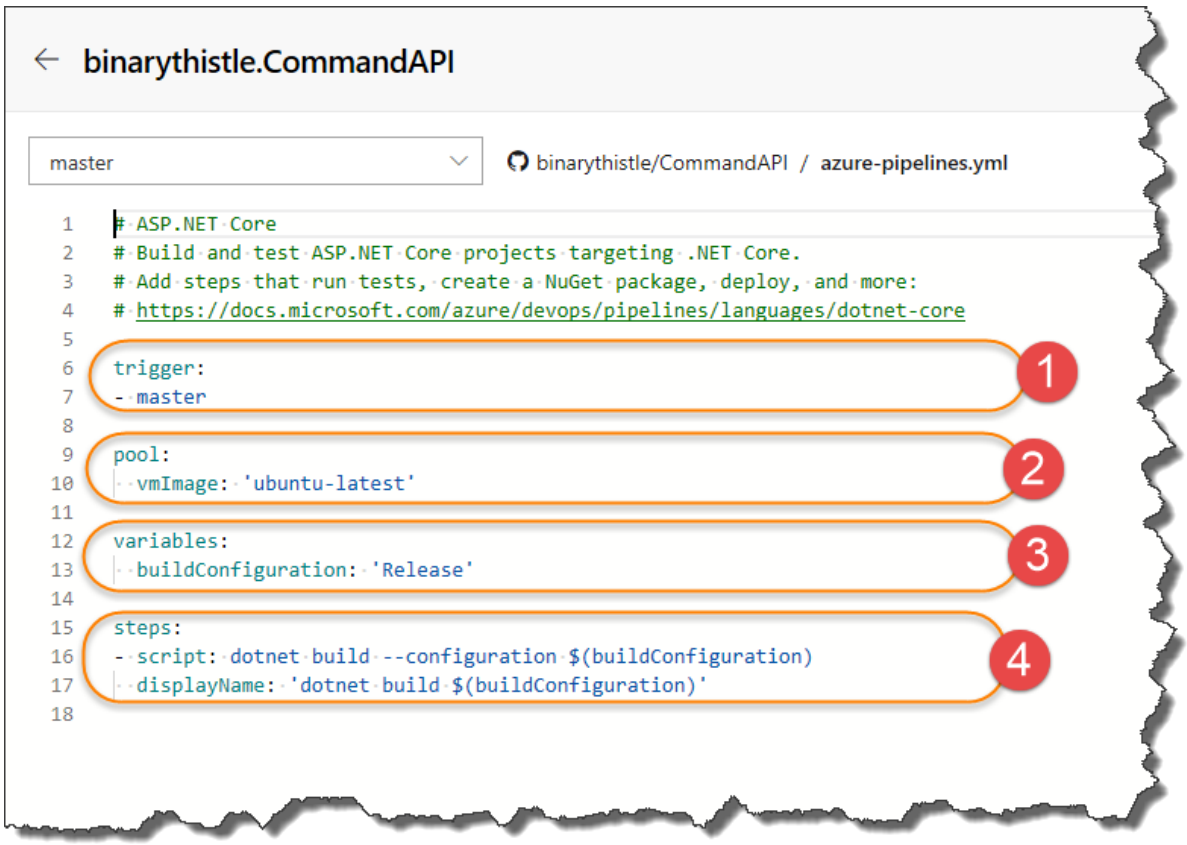
A new build has been queued to start, this time triggered by a remote commit to GitHub!

Once it starts, 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

Returning to our azure-pipelines.yml file in Azure DevOps, (click Pipelines->Builds->Edit), you should see the following:



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.

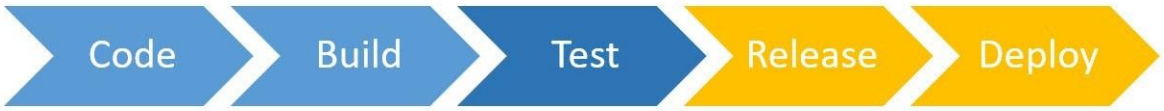
It does not:

- Run our unit tests
- Package our project for deployment

We need to add those steps to make the Pipeline valuable, (a CI/CD pipeline that does not run tests, in my mind is useless!).

Running Unit Tests

Returning to the steps in our pipeline view:



You'll see the suggested sequencing is: Build->**Test->** Release, so as we already have our Build step in our azure-pipelines.yml file we now need to add the test step after this. We can edit our YAML file either:

- Directly in the browser
- In VS Code

Editing directly in the browser window has the advantage that Azure DevOps provides IntelliSense like guidance on how to complete, which can be really useful, however we're going to do our editing in VS Code, (basically because I like to do all my coding in one place!).

So, move back to VS Code and open ***azure-pipelines.yml*** and *append* the following steps *after* the build steps, (keeping in sync with our steps as outlined above).

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

So overall the file should like this:

```
trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

variables:
  buildConfiguration: 'Release'

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

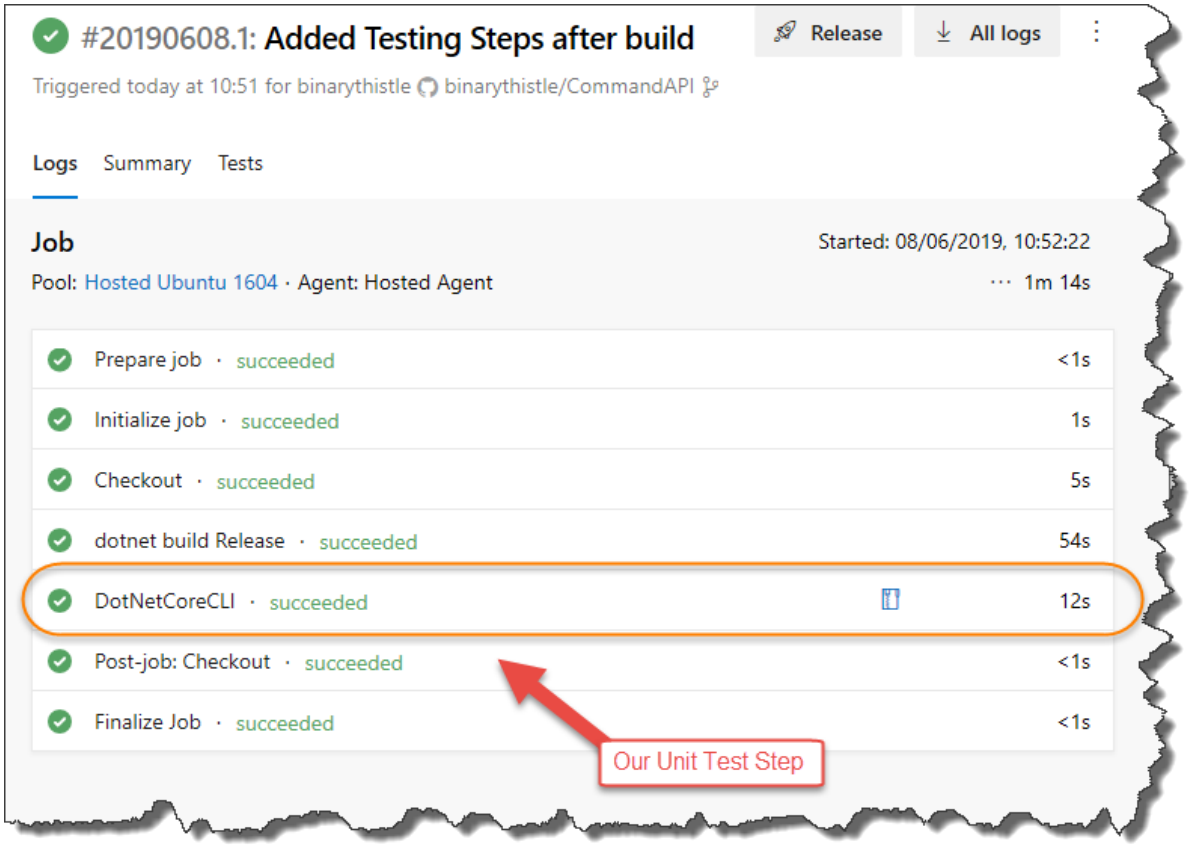
Les's Personal Anecdote: If you read my blog article on the same subject you'll realise I put the testing step *before* the build step – the reason? A mistake that I couldn't be bothered to fix in my blog article, but that I'm fixing here.

I'm in good company getting things back to front though…

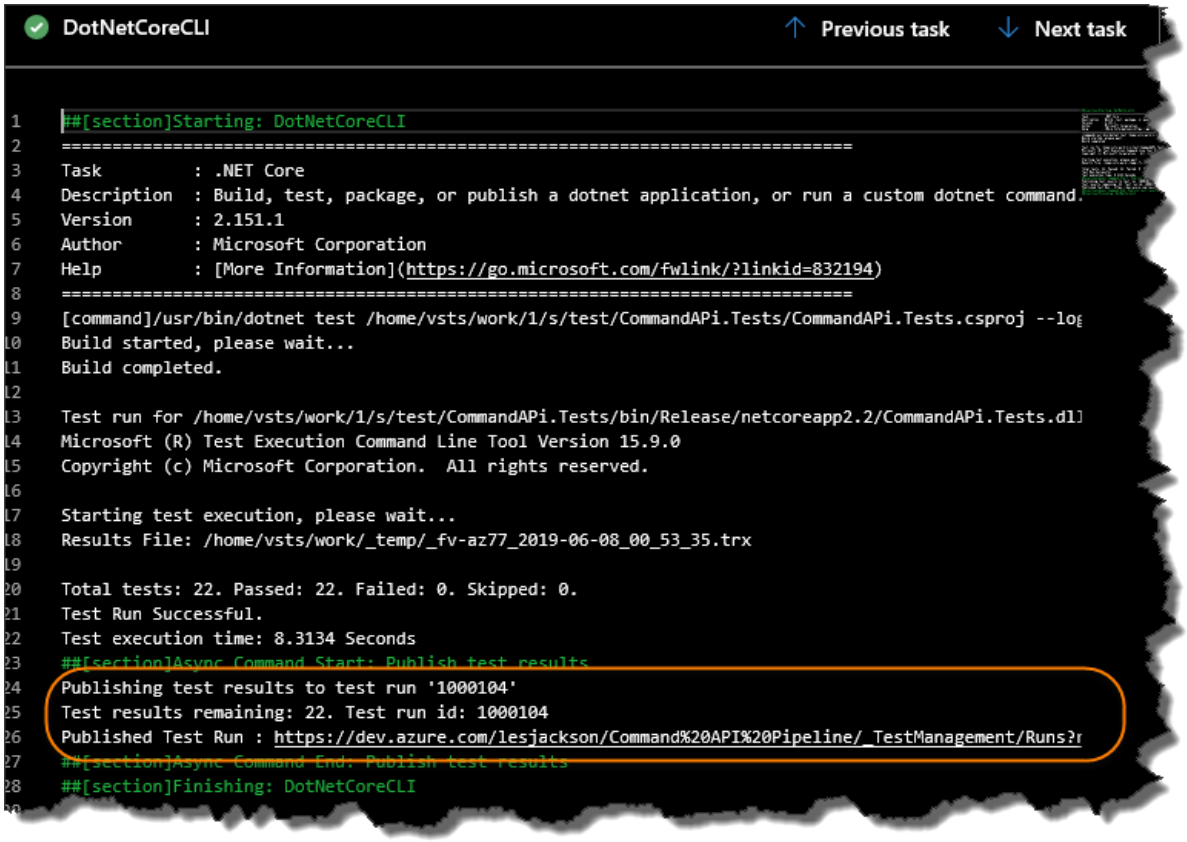
- In my home town of Glasgow, there is an urban myth that the famous Kelvingrove Art Gallery and Museum was built back to front… The Architect realising his mistake only on completion – then climbed to the highest spire and threw himself off!
- In my adopted home-town of Melbourne there's a similar urban myth that the main train station Flinders Street was built by mistake in Australia, (the design was intended for Mumbai – India, with Mumbai receiving the Australian station design!)

Warning! YAML can be "whitespace sensitive", meaning if you don't put the correct whitespace indentation in, the file will not work as intended. I have encountered this experience personally, and it's not fun – a reason I don't like YAML. Anyway be careful!

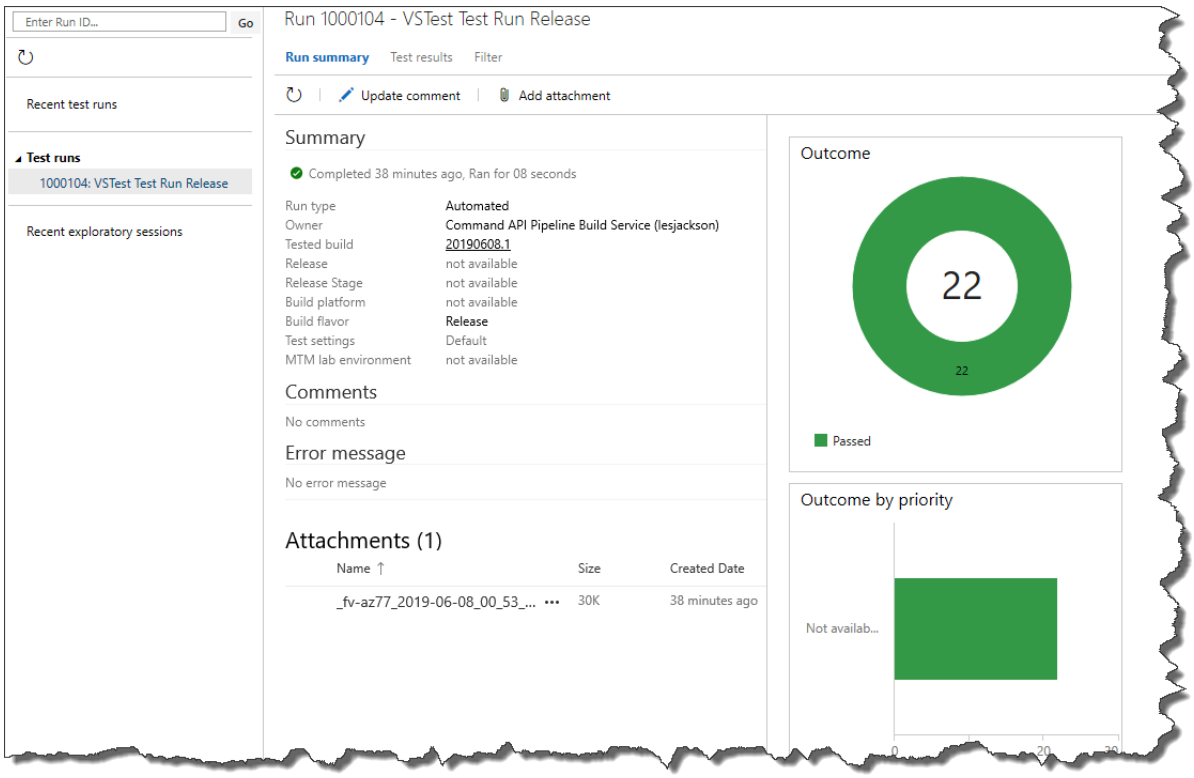
The steps are quite self-explanatory, the only point of note is that instead of a `script` step we are using the `task` step, primarily as it's more readable given the number of parameters required. Save the file in VS Code, and perform the necessary Git command line steps to commit your code and push to GitHub – this should trigger another build of our pipeline – this time the unit tests should execute too:



Click on the step as shown above to drill-down to see what's going on, you should see something like:



Clicking on the link highlighted above takes you to a realty cool test result dashboard:

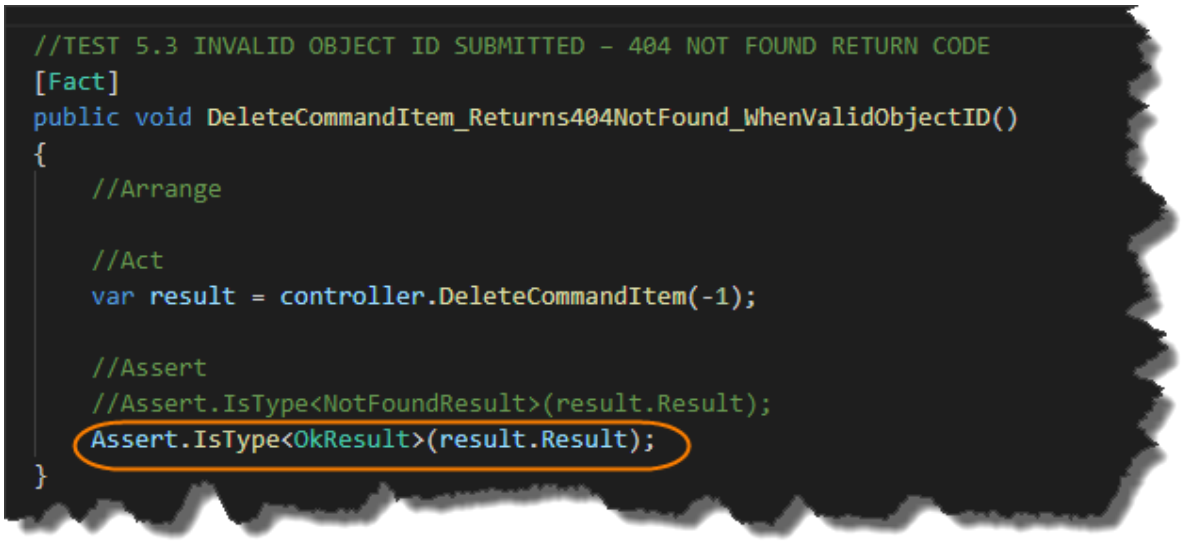


Very nice! Indeed, this is the type of *Information Radiator* that you should make highly visible when working in a team environment, as it helps everyone understand the health of the build, and if necessary take action to remediate any issues…

Breaking Our Unit Tests

Now just to labour the point of unit tests, and CI/CD pipelines let's deliberately break one of our tests…

Back in VS Code and back in our ***CommandAPI.Tests*** project, open our ***CommandsController*** tests and edit one of your tests and change the expected return type, I've chosen the test below, and swapped `NotFoundResult` with `OKResult`:



```
//TEST 5.3 INVALID OBJECT ID SUBMITTED - 404 NOT FOUND RETURN CODE
[Fact]
public void DeleteCommandItem_Returns404NotFound_WhenValidObjectID()
{
    //Arrange

    //Act
    var result = controller.DeleteCommandItem(-1);

    //Assert
    //Assert.IsType<NotFoundResult>(result.Result);
    Assert.IsType<OkResult>(result.Result);
}
```

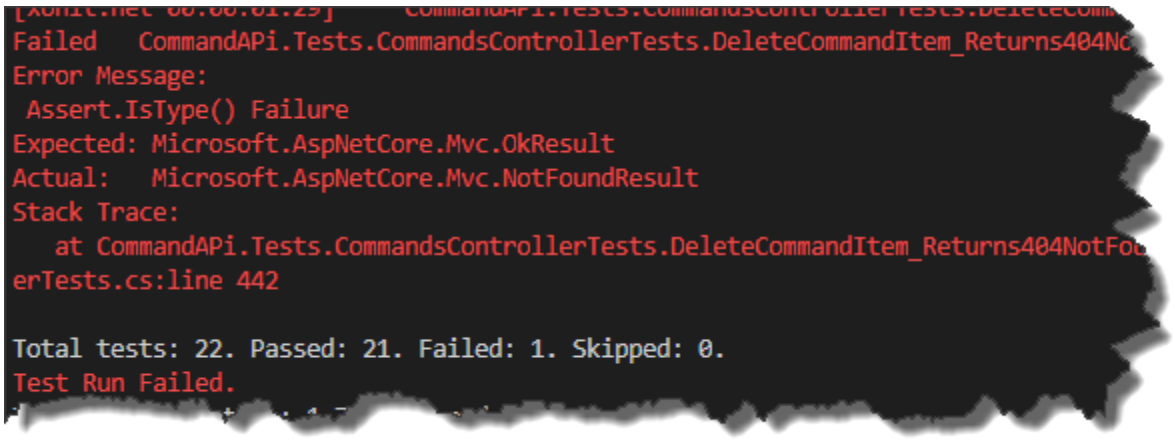
Save the file and (ensuing you're "in" the CommandAPI.Tests project), run a build:

```
dotnet build
```

The *build of the project will succeed* as there is nothing here that would cause a compile-time error. However if we try a:

```
dotnet test
```

We'll of course get a failing result:

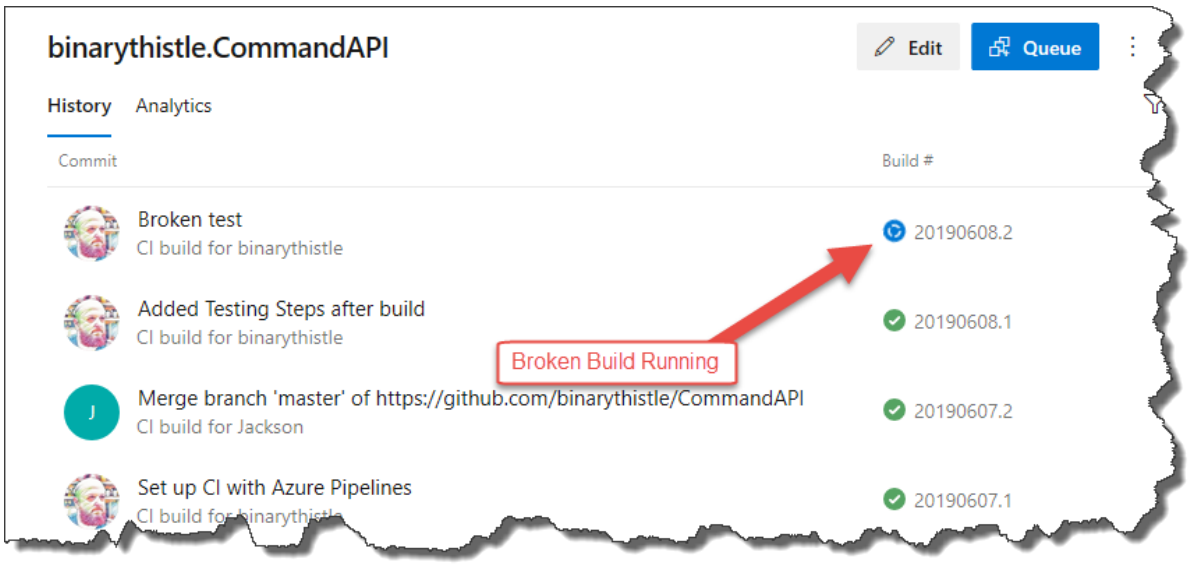


```
Failed   CommandAPi.Tests.CommandsControllerTests.DeleteCommandItem_Returns404Nc
Error Message:
 Assert.IsType() Failure
Expected: Microsoft.AspNetCore.Mvc.OkResult
Actual:   Microsoft.AspNetCore.Mvc.NotFoundResult
Stack Trace:
   at CommandAPi.Tests.CommandsControllerTests.DeleteCommandItem_Returns404NotFo
erTests.cs:line 442

Total tests: 22. Passed: 21. Failed: 1. Skipped: 0.
Test Run Failed.
```

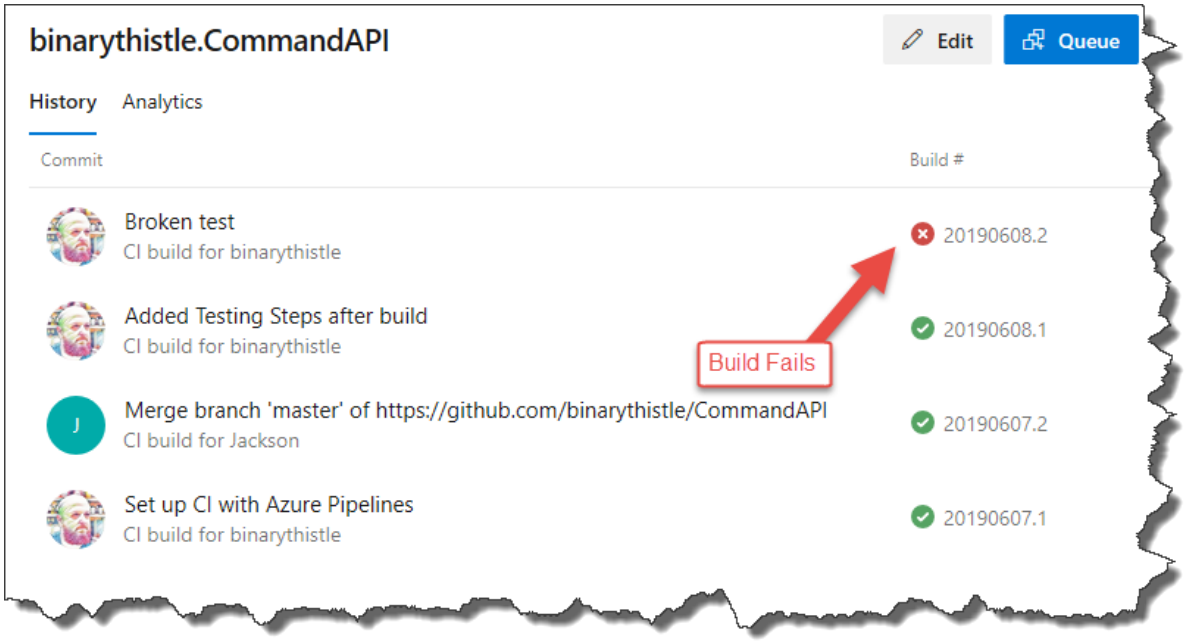
Now under normal circumstance, having just caused our unit test suite to fail locally, you **would not then commit** the changes and push them to GitHub! That is exactly what we are going to do just to prove the point that the tests will fail in the Azure DevOps build pipeline too.

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 that have done so and commit their code, again this just highlights the value in a CI/CI build pipeline.

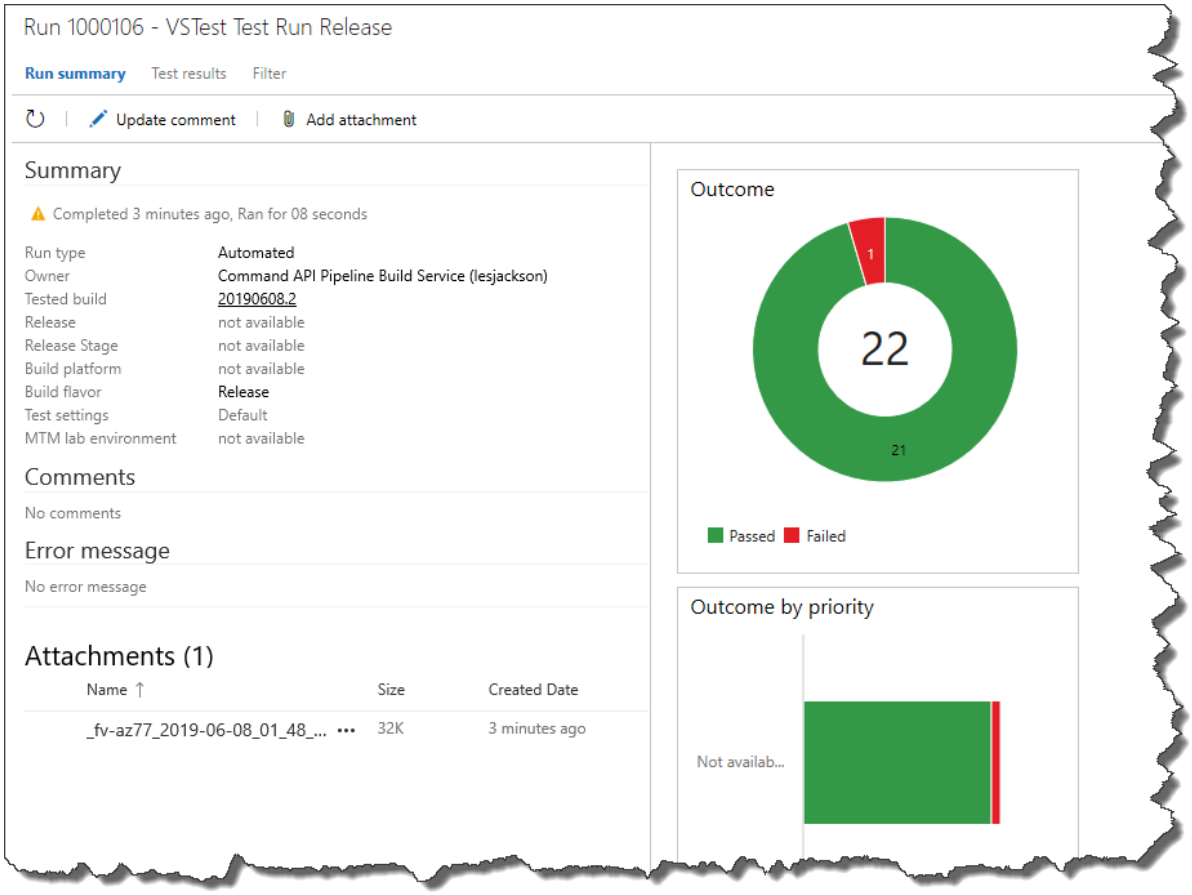
So perform the 3 "Git" steps you should be familiar with now, and once you've pushed to GitHub, move back across to Azure DevOps and observe what happens…



Then as expected, our test fails:



Again you can drill down to see what caused the error, and if for example you were displaying test results on a large LCD screen, it would be immediately apparent that there is something wrong with the build pipeline, and that remedial action needs to be taken:



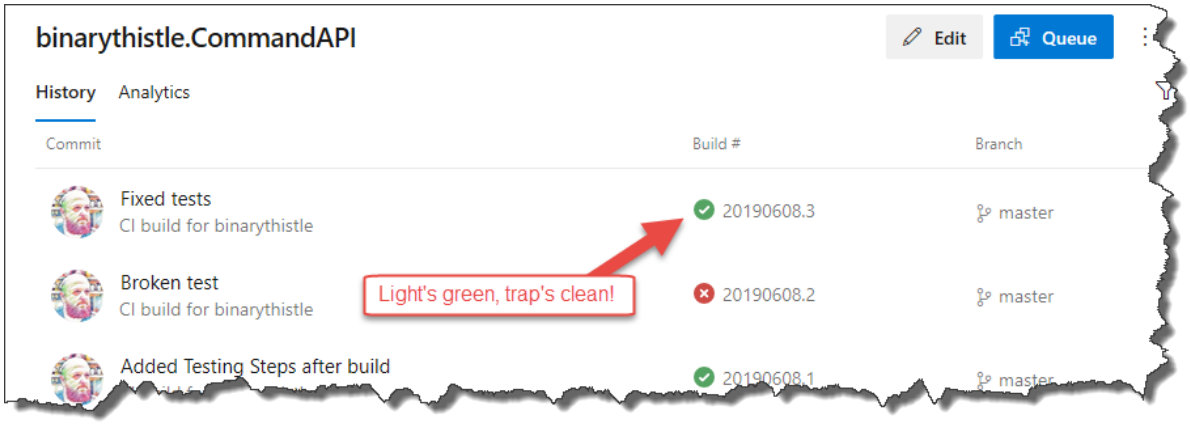
Testing – The Great Catch All?

Now this shows us the power of unit testing, 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! 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 – ours is not bad at this stage though – so we can be quite confident in moving to the next step…

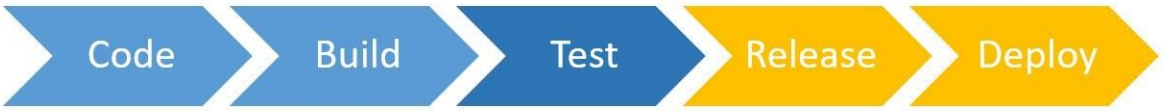
Before we do that though revert the change, we just made to ensure that all our unit tests are passing, and that our pipeline returns to a green state.

Warning! Do not progress to the next section without ensuring that all your tests are passing!



Release / Packaging

Referring to our pipeline again, we're now at the Release stage, this is where we need to package our build ready to be deployed:



So once again, move back in to VS Code and open ***azure-pipelines.yml*** file, and append the following steps:

```
- task: DotNetCoreCLI@2
  displayName: 'dotnet publish --configuration $(buildConfiguration) --output
$(Build.ArtifactStagingDirectory)'
  inputs:
    command: publish
    publishWebProjects: false
    projects: 'src/CommandAPI/CommandAPI.csproj'
    arguments: '--configuration $(BuildConfiguration) --output
$(Build.ArtifactStagingDirectory)'
    zipAfterPublish: true
- task: PublishBuildArtifacts@1
  displayName: 'publish artifacts'
```

So overall you're file should look like this, with the new code highlighted:

```yaml
trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

variables:
  buildConfiguration: 'Release'

steps:
- script: dotnet build --configuration $(buildConfiguration)
  displayName: 'dotnet build $(buildConfiguration)'
- task: DotNetCoreCLI@2
  inputs:
    command: test
    projects: '**/*Tests/*.csproj'
    arguments: '--configuration $(buildConfiguration)'
- task: DotNetCoreCLI@2
  displayName: 'dotnet publish --configuration $(buildConfiguration) --output $(Build.ArtifactStagingDirectory)'
  inputs:
    command: publish
    publishWebProjects: false
    projects: 'src/CommandAPI/CommandAPI.csproj'
    arguments: '--configuration $(BuildConfiguration) --output $(Build.ArtifactStagingDirectory)'
    zipAfterPublish: true
- task: PublishBuildArtifacts@1
  displayName: 'publish artifacts'
```

The steps are explained in more detail in this MSDN article, but in short:

- A dotnet publish command is issued for our **CommandAPI** project only [20]
- The output of that is zipped
- Finally zipped output is published

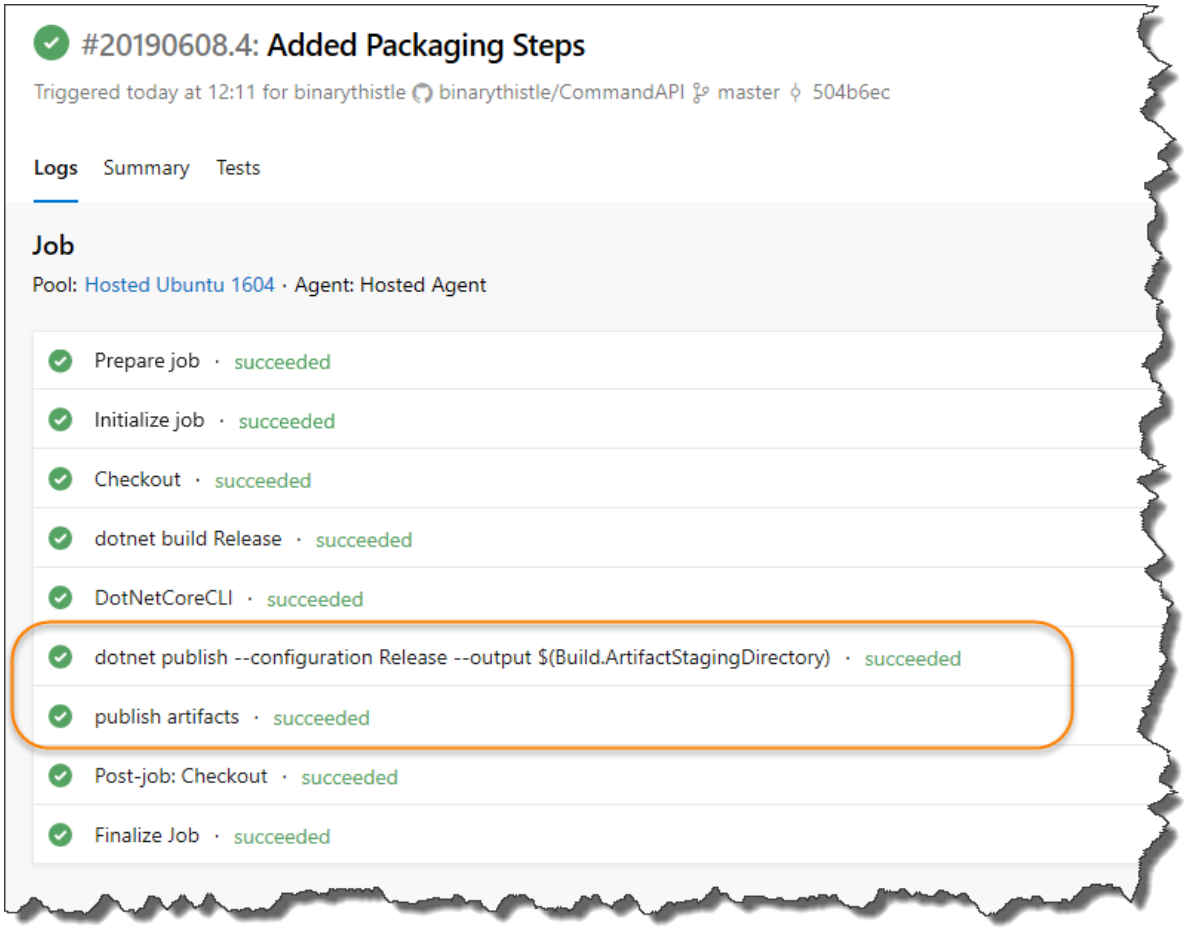
Les's Personal Anecdote: Ensure that you put in the following line:

```
publishWebProjects: false
```

When researching this, I spent about 2-3 hours trying to understand why the packaging step was not working – it was because of this! The default is `true`, so if you don't include that, the step fails…. ARGHHHH!

Save the file, and again: add, commit and push your code. The pipeline should succeed and if you drill into the successful build, you'll see our 2 additional `task` steps:

[20] We don't want to publish our tests anywhere!



Celebration Check Point: Excellent work! You have completed the: build, test and release steps of our pipeline using Azure DevOps.

Wrap It Up

A lot of ground covered here, where we:

- Set up a CI/CD pipeline on Azure DevOps
- Connected Azure DevOps to GitHub (and ensured CI triggers were enable)
- Add Test and Packaging steps to our ***azure-pipeline.yml*** file

We are now almost ready to deploy to Azure!

CHAPTER 10 – DEPLOYING TO AZURE

CHAPTER SUMMARY

In this chapter we bring deploy our API onto Azure for use in the real world. On the way we create the Azure resources we need and re-visit the discussion on runtime environments and configuration.

WHEN DONE, YOU WILL

- Know a bit more about Azure
- Have created the Azure resources we need to deploy our API
- Update our CI/CD pipeline to deploy our release to Azure
- Provide the necessary configuration to get the API working in a Production Environment

We have a lot to cover – so let's get going!

CREATING AZURE RESOURCES

Azure is a huge subject area, and could fill many books, many times over, so I'll be focusing only on the aspects we need to get our API & Database up and running in a Production environment – which should be more than enough!

In simple terms, everything in Azure is a "resource", e.g. a Database server, Virtual Machine, Web App etc. So we need to create a few resources to house our app. There are 2 main ways to create resources in Azure:

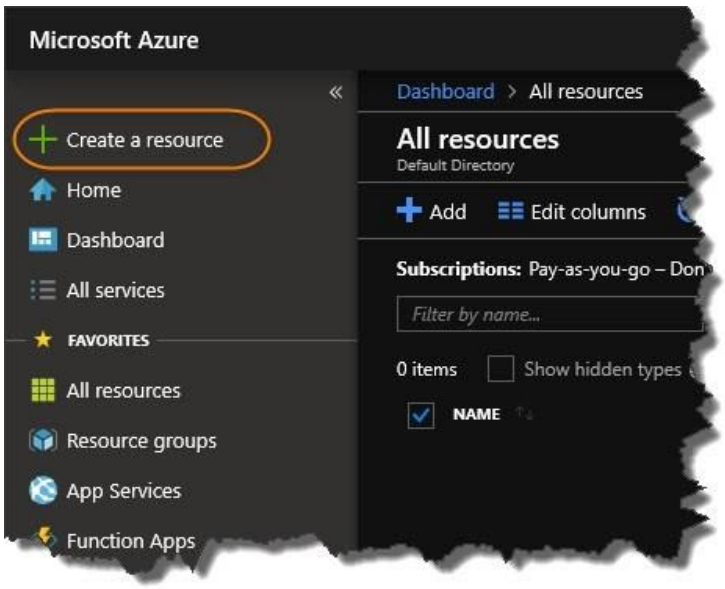
1. Create resources manually via the Azure Portal
2. Create resources automatically via Azure Resource Manager Templates

In this chapter, we'll be manually creating the resources we need as:

- It's simpler
- We only have a small number of resources
- I think it's the right approach to learning, (jumping to Resource Templates, I feel would be running before we were walking).

CREATE OUR API APP

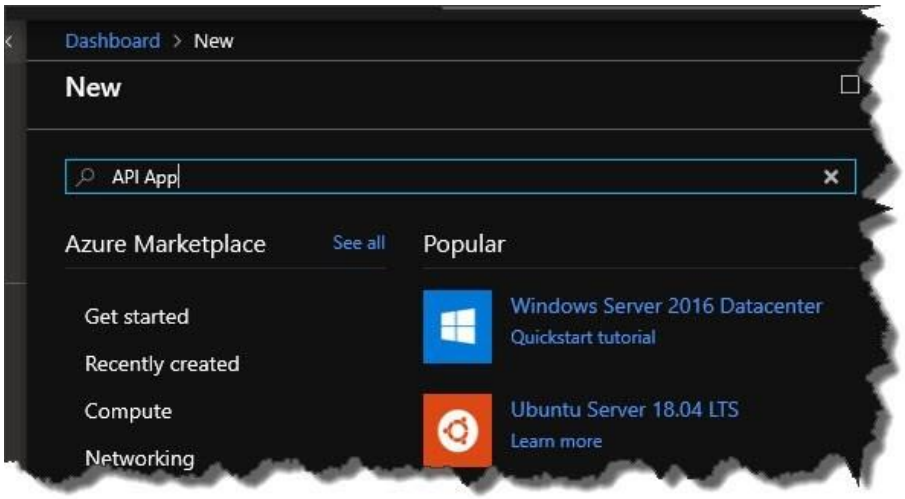
The 1st resource we are going to create is an API App, this unsurprisingly is where our API code will run! To do so, Login to Azure , (or if you don't have an account you'll need to create one), and click on "Create a resource":



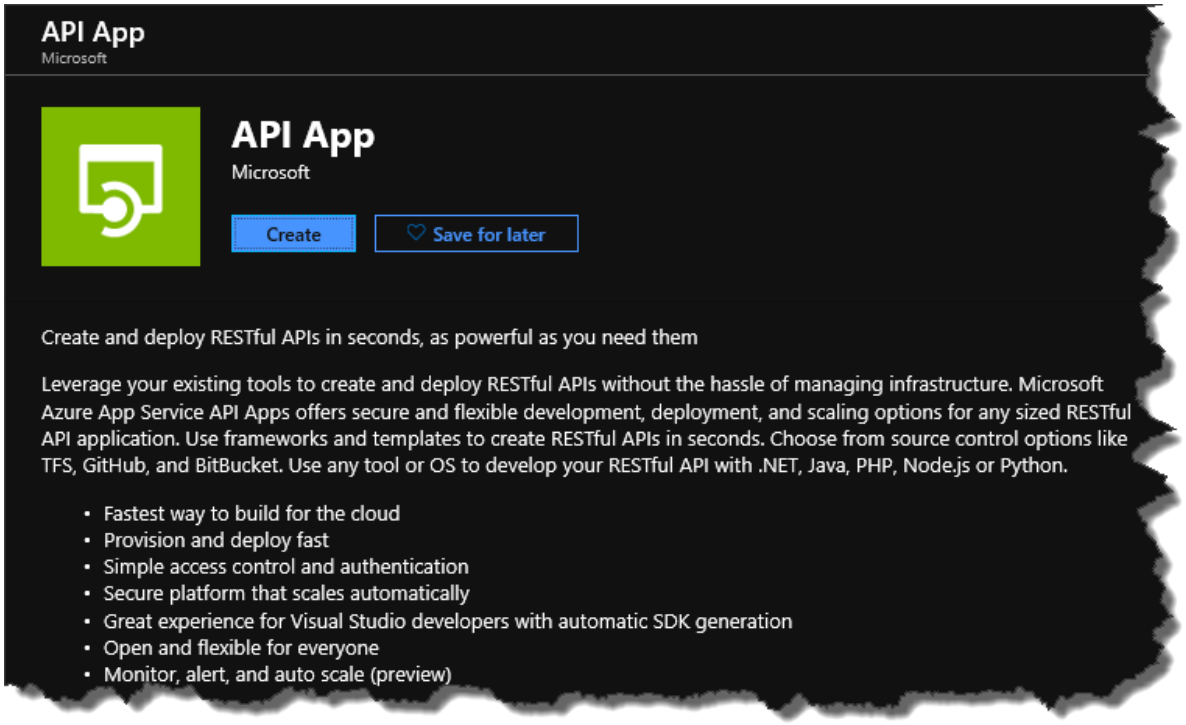
Again, I'll mention the point that the following screenshot were correct at the time of writing but given the fast pace of change in Azure they may be subject to change.

Fundamentally though, resource creation is Azure is not that difficult, so small UI changes should not stump someone as smart as yourself!

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:



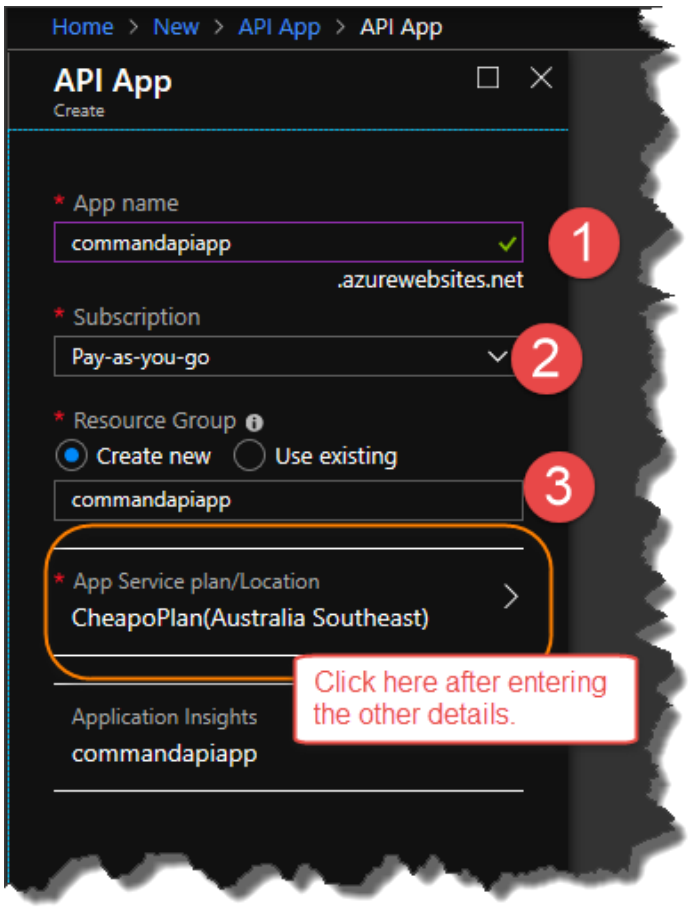
Select "API App", then click "Create":



On the Next "page", enter:

1. A name for your API App[21]
2. Select your subscription (I just have a pay-as-you-go")
3. A name for your new "Resource Group" – these are just groupings of "resources" – duh!

[21] This needs to be unique in Azure so, your name will be different to mine.



WAIT! Before you click Create, click on the App Service plan/location.

Les's Personal Anecdote: The API App resource describes what you are getting, the App Service plan & location tells you how that API App will be delivered to you…

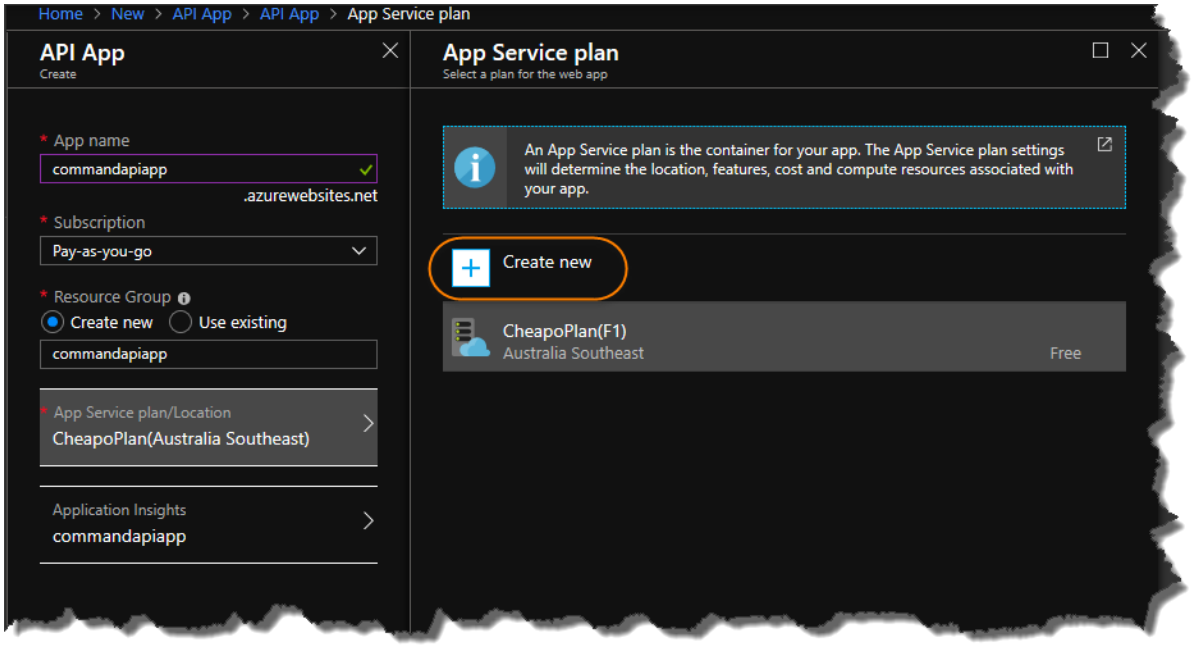
E.g. Do you want your API App:

- Hosted in the US, Western Europe, Asia etc,
- On shared or dedicated hardware
- Running on certain processor speed, etc.

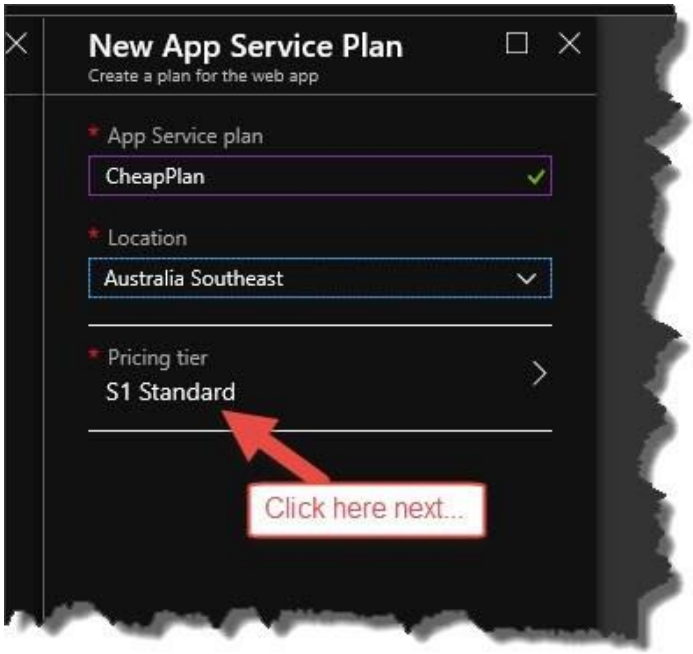
By default if you've not used Azure before you'll be placed on a Standard plan which can incur costs! (This is a personal anecdote because I did that and was shocked when my test API started costing me money!).

So be careful of the Service Plan you set up, I detail the free plan next.

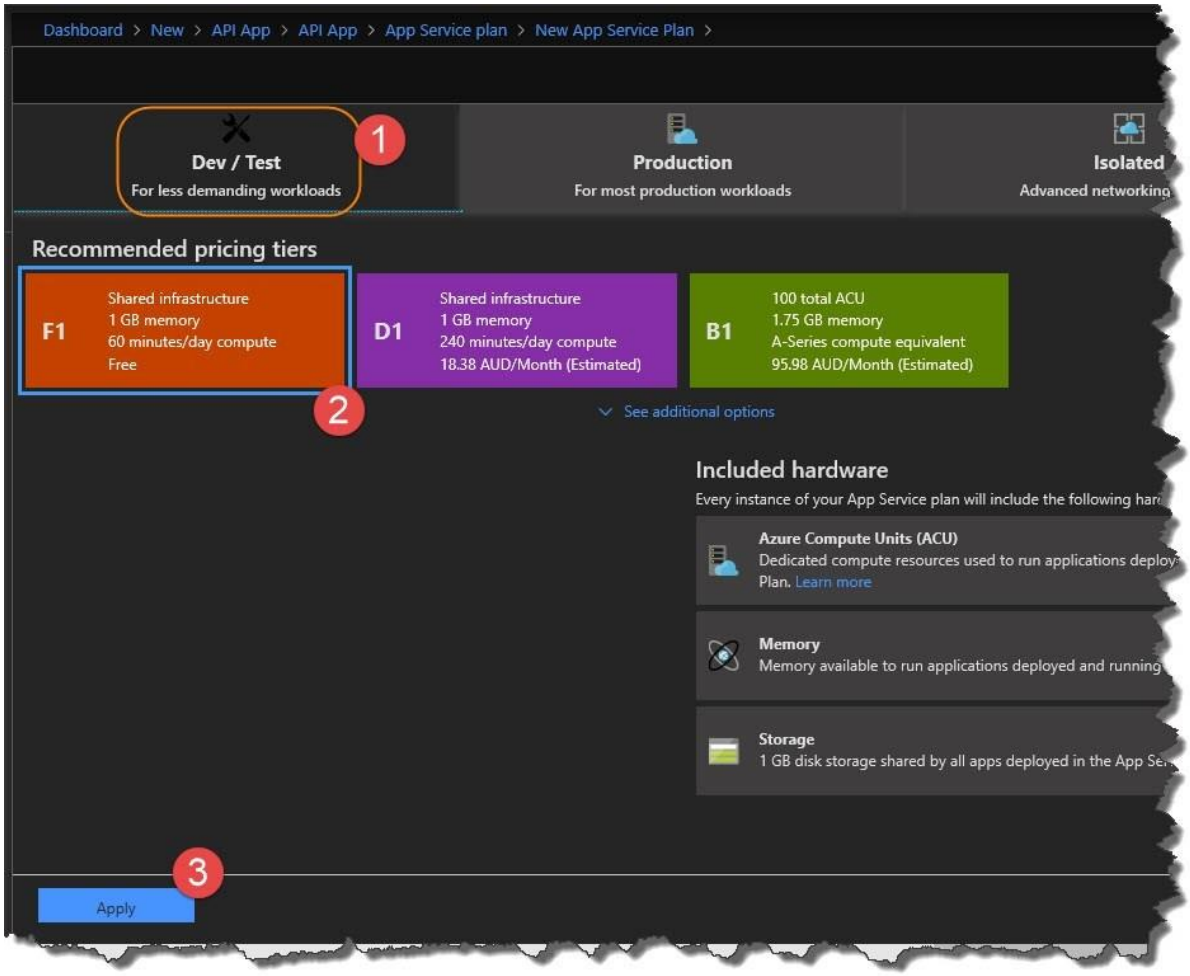
After clicking on the Service Plan, click on "Create new":



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



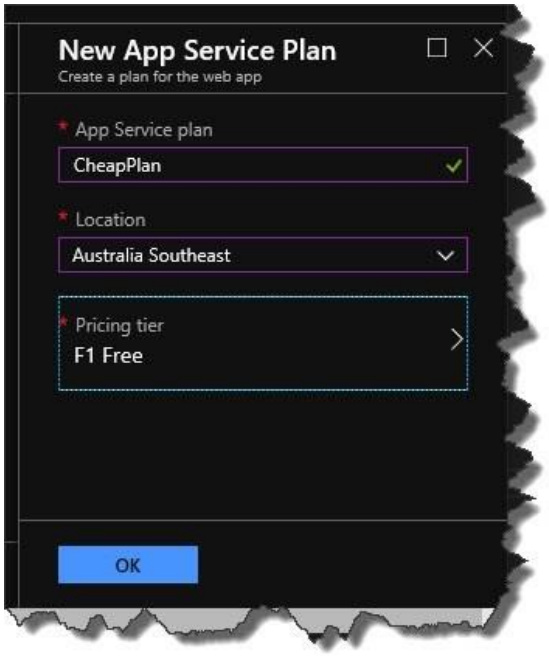
After, click on the *Pricing Tier:*



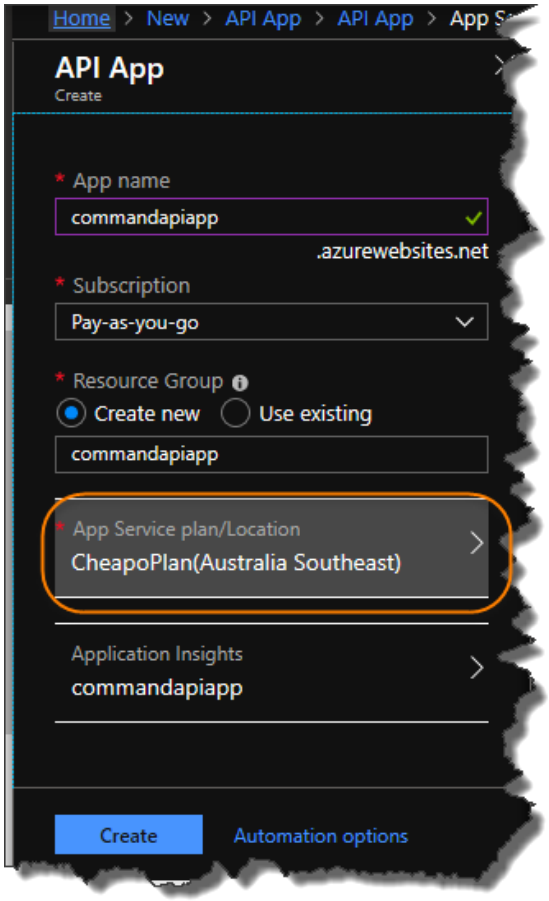
1. Select the Dev/Test Tab
2. Select the “F1” Option (Shared Infrastructure / 60 minutes compute)
3. Click Apply

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).

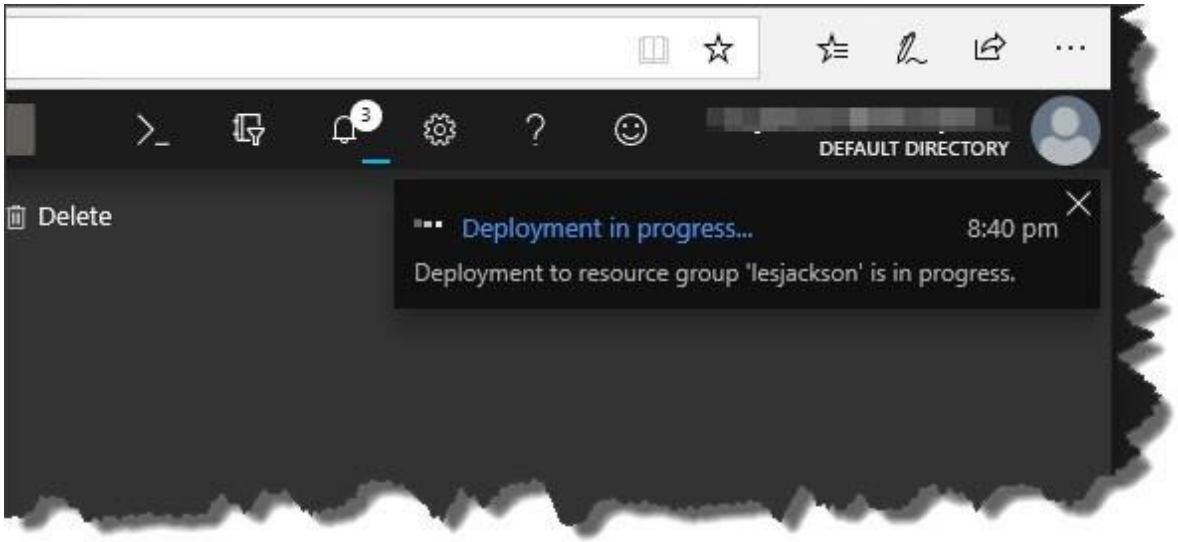
Then Click OK…



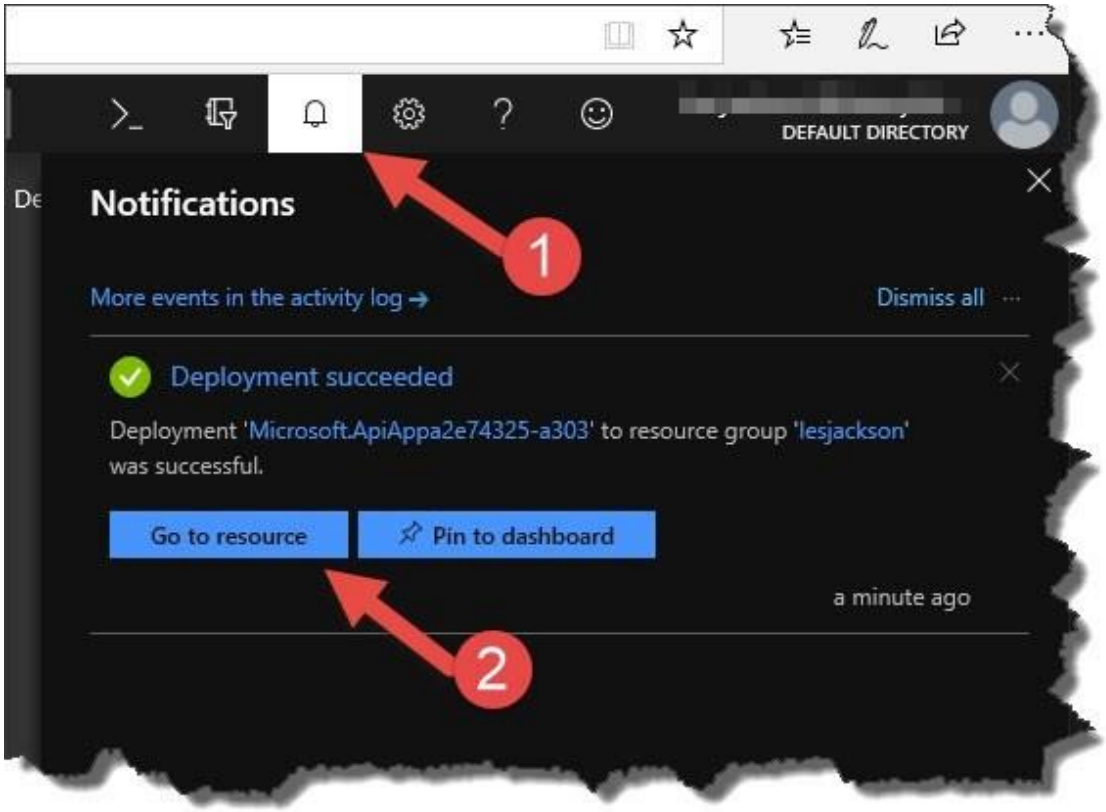
Then click “Create”, (ensure your new App Service Plan is selected):



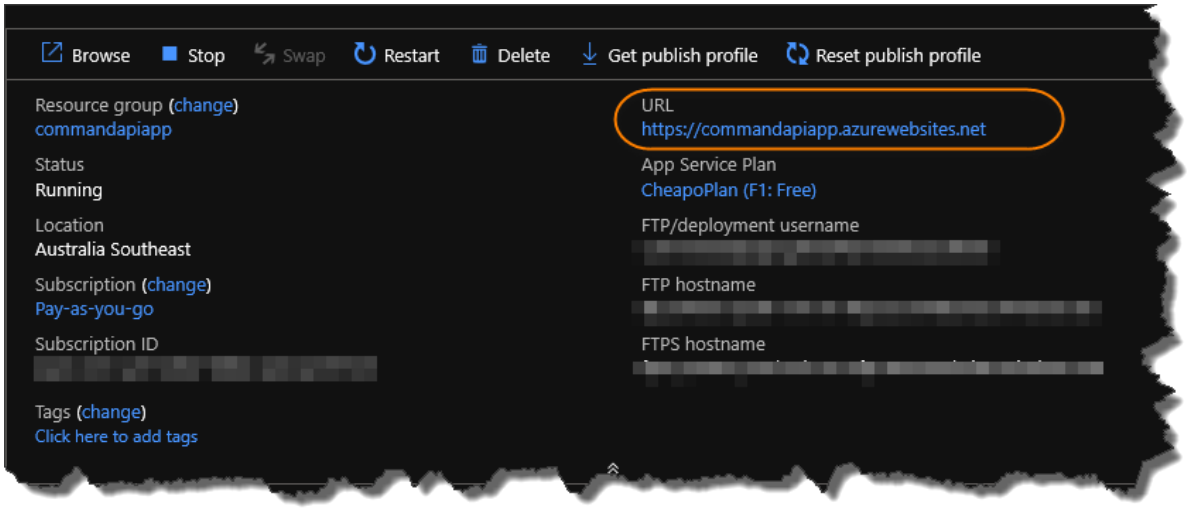
After clicking Create, Azure will go off an create the resource ready for use:



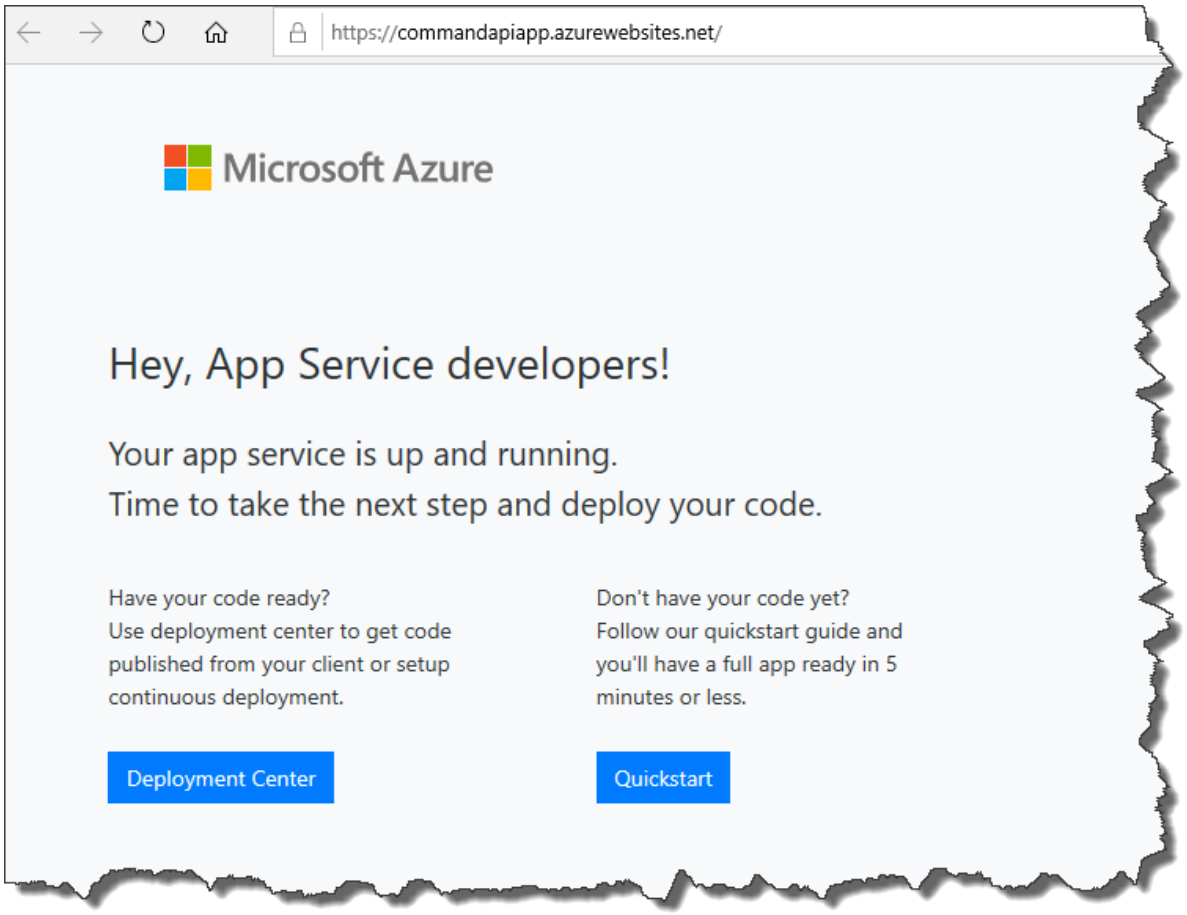
You will get notified when the resource is successfully created, if not, click the little “Alarm Bell” icon near the top right hand side of the Azure portal:



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



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:



As we have not deployed anything, you'll get a similar landing page as shown above.

Celebration Check Point: You've just created your 1st Azure resource, one of the primary components of our production solution architecture!

Create Our SQL Server & Database

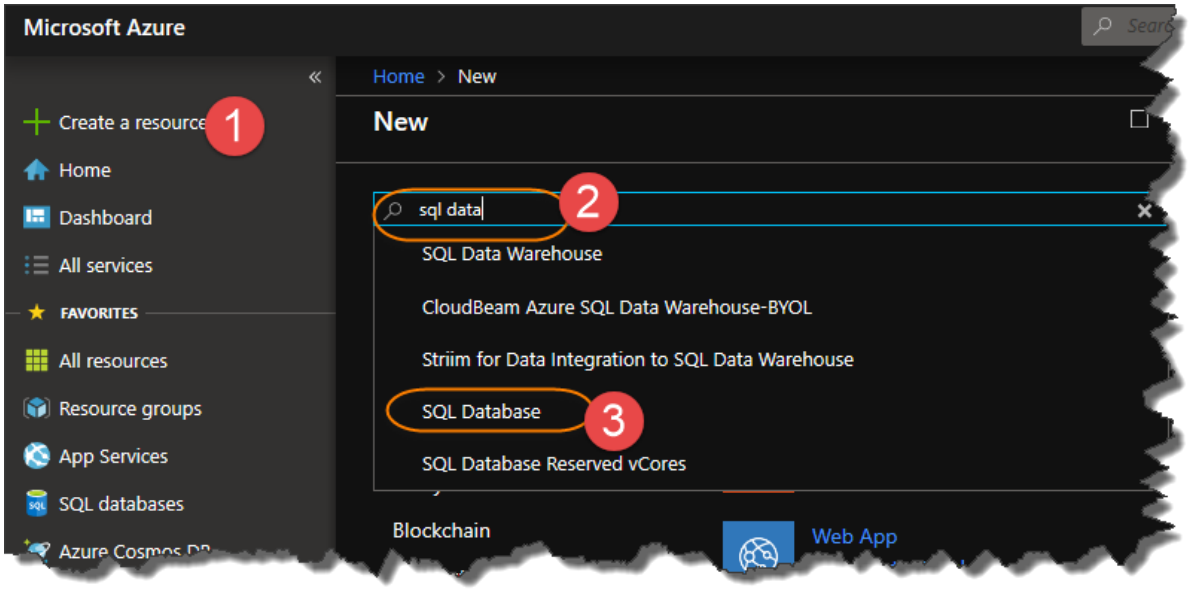
Like our local version of SQL Server, in Azure, SQL Databases have the concepts of:

- Server
- Database

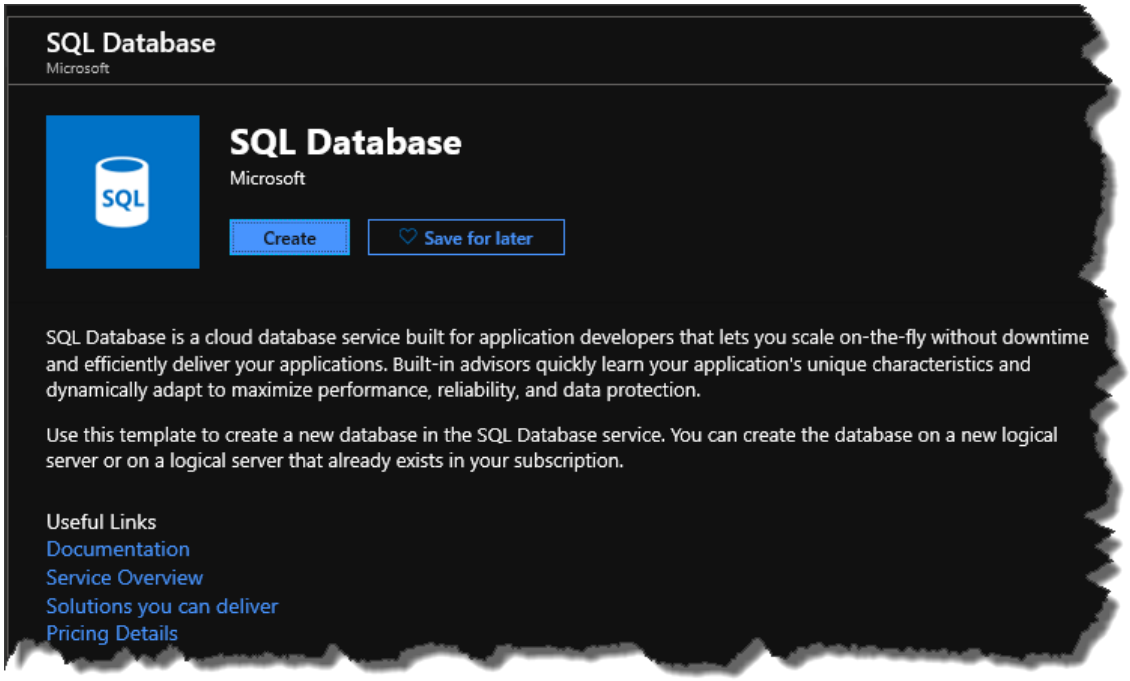
So the Server is the *platform* that hosts one or more Databases, so we're going to set that up now.

Again, on the Azure Portal homepage:

1. Click *Create a Resource*
2. Search for "SQL Database"
3. Select *SQL Database* from the drop down

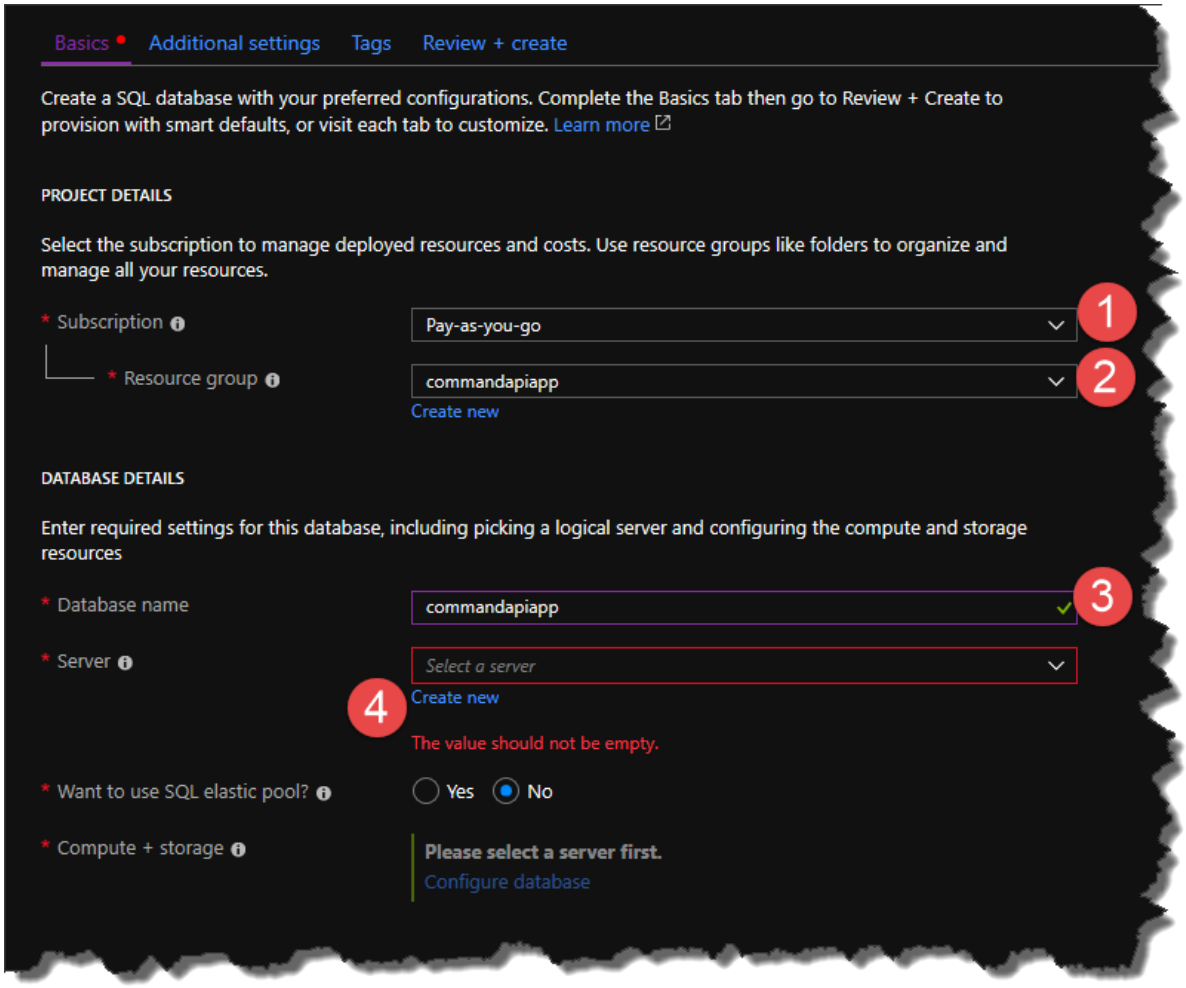


On the resulting SQL Database screen, just click *Create*:



On the Create screen enter:

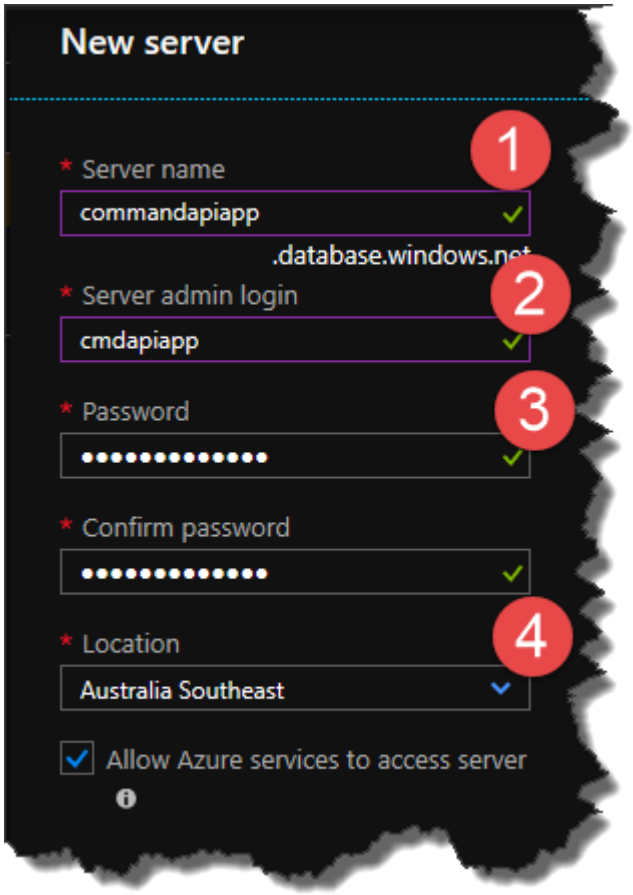
1. Select your *Subscription*
2. Select the *Resource Group*, (I'd select the same one that you placed you API App in)
3. *Database name* (FYI I used the same name that I gave my local instance – you don't have to though)



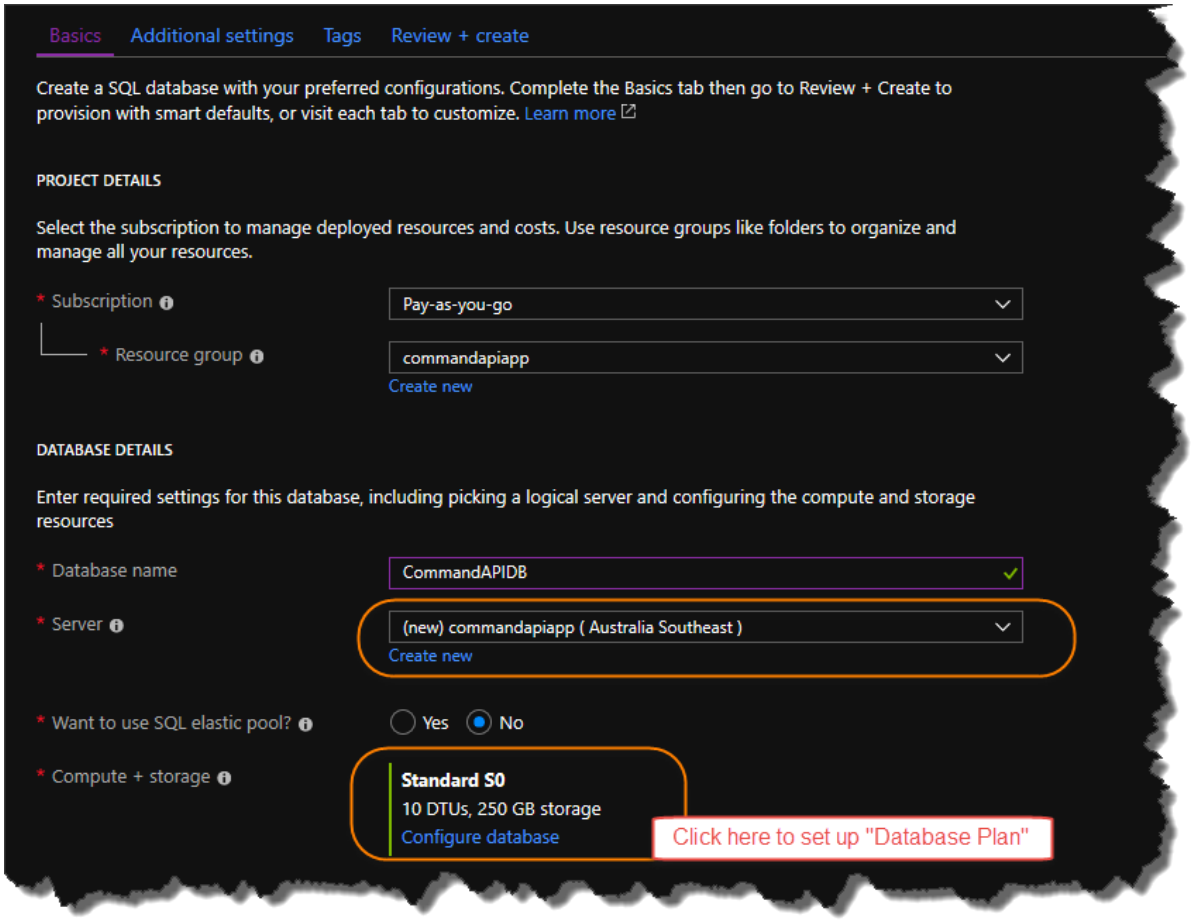
4. Click Create new to create a new Server, as shown below:

In the New Server box, enter:

1. Server Name
2. Server admin login name (make a note of this)
3. Password (again make a note of this)
4. Location – should be the same location as your API App (as specified in the *App Service Plan*)

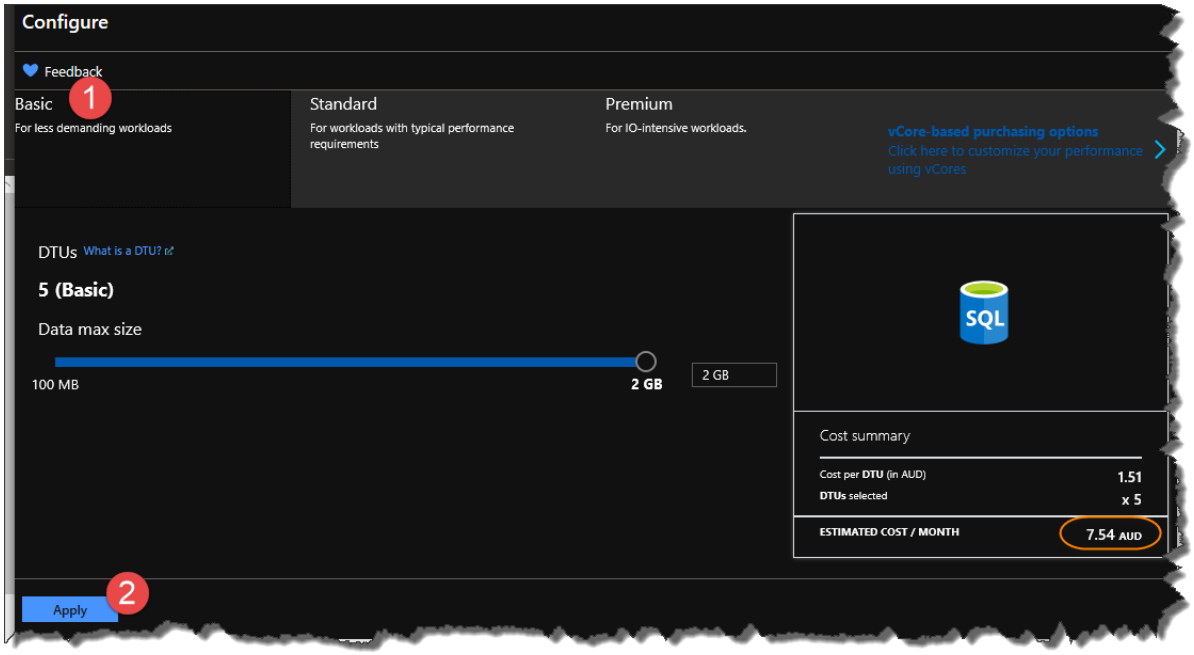


Click Select to create the Server, this should now be reflected in the main Create SQL Database form window:



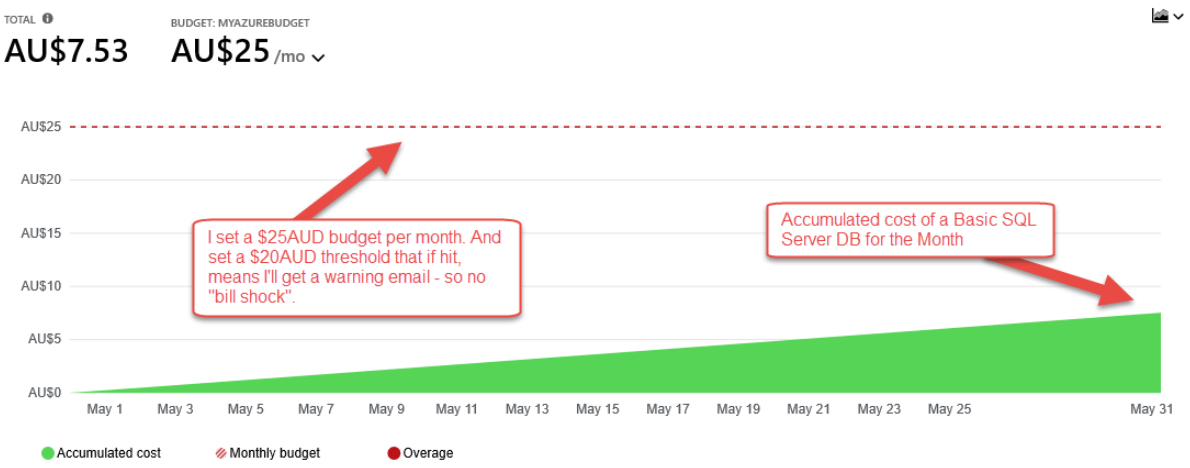
WAIT: Again, as with our API App, the Database also has a configurable "plan", you can see that by default the *Standard S0* plan is selected, we don't want that so click Configure database to change this. On the resulting screen:

1. Select *Basic*
2. Click *Apply*



You'll notice that the estimated cost for this is ~$7 ½ AUD, (Australian Dollars), per month.

Having had one of these running for a few months with little traffic I can confirm that this pricing is fairly accurate, see below:

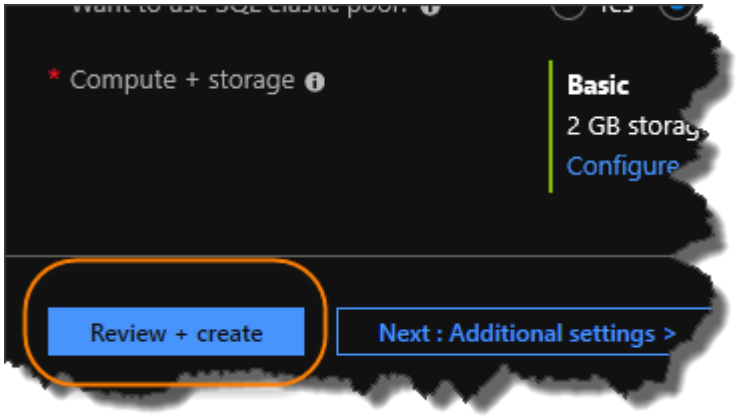


Warning! If you choose to create this SQL Database in the steps that follow that means that you may be liable for charges to your Azure account. If you are uncomfortable with that idea, don't proceed.

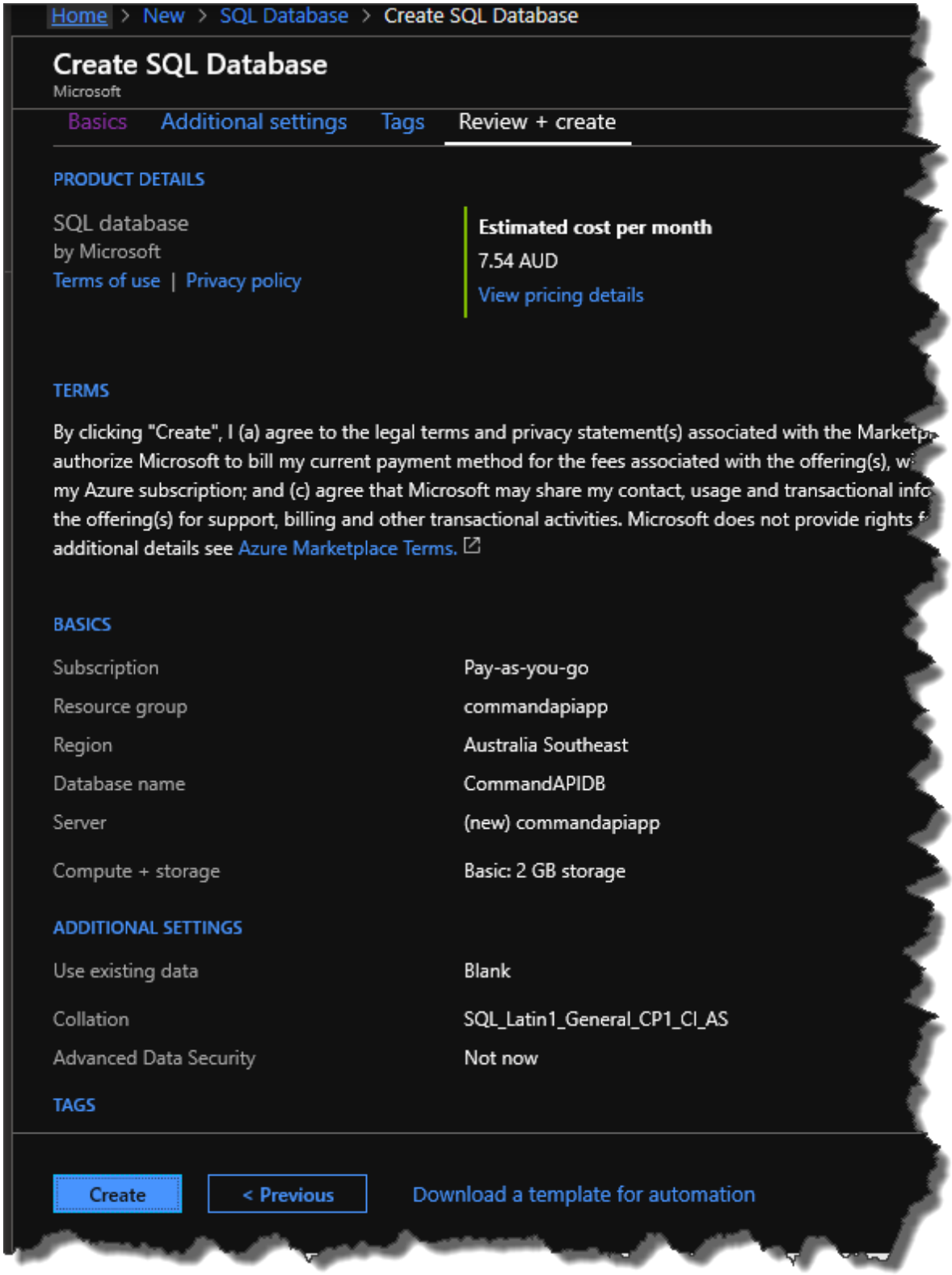
Not that I'm trying to sell Azure, but before backing out, think about the following:

- The estimated pricing is correct for low traffic (this is about the same price as 2 cups of coffee here in Melbourne – which I think is reasonable!)
- You can choose to delete these resources at any time, therefore not incurring continued costs
- Microsoft provides a number of ways to manage your budgets – I wrote an article on the options available on my blog.
- If you're a new Subscriber to Azure, you get 250GB of SQL Database free for the 1st 12 months – so you shouldn't incur costs, (within those bounds).

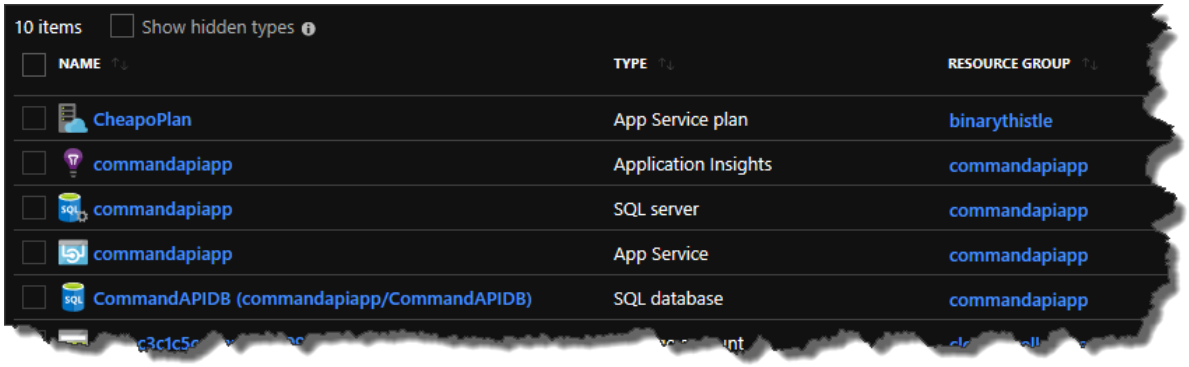
Click Review and Create back on the Create SQL Database screen:



You'll get a final summary page, if you're happy, click create and Azure will go off and provision your Database for you:

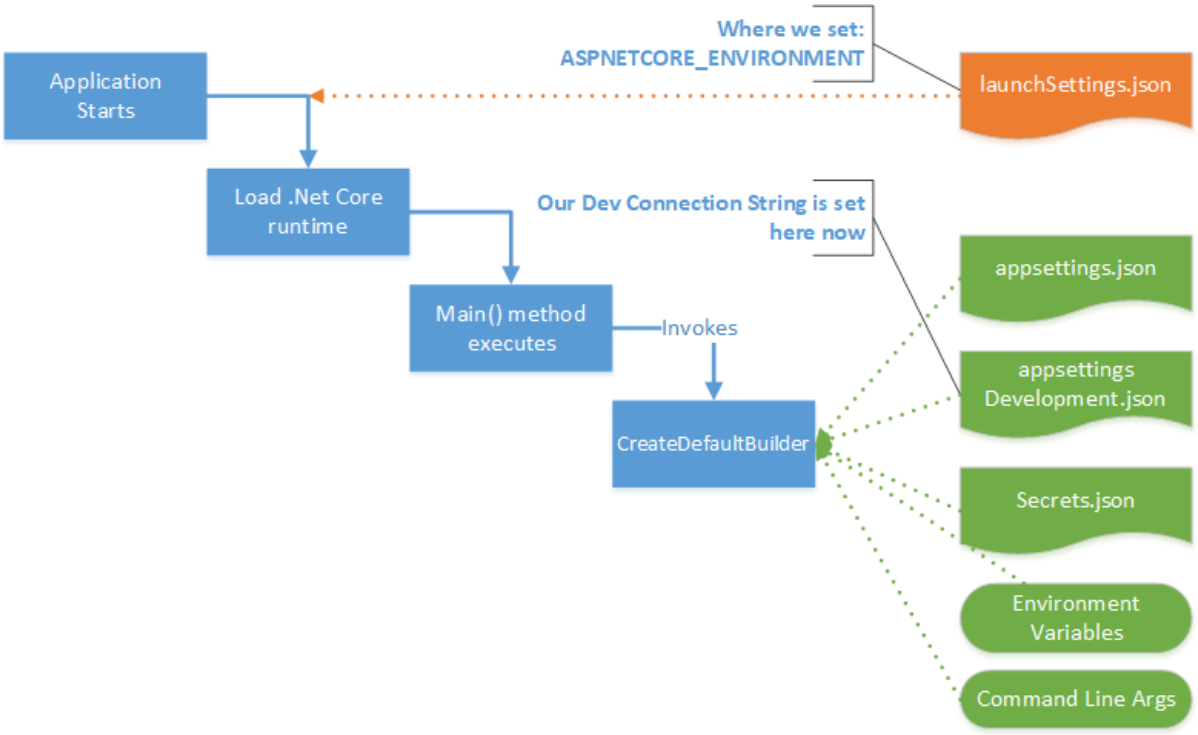


Again you'll get notified when both your resources are set up: clicking on All resources, you can see everything we have created:



Revisit Our Dev Environment

So we've covered a lot of ground since Chapter 7 – Environment Variables & User Secrets, but it's worth doing a bit of a review:



- We set our environment ***launchSettings.json*** (in the ASPNETCORE_ENVIRONEMENT variable)
- Our Connection Strings can sit in ***appsettings.json***, or the environment specific variants of that file, e.g.: ***appsettingsDevelopment.json*** This is where our Development connection string sits.
- "Secret" information, such as Database login credentials can be broken out into **Secrets.json** via The Secret Manager tool. Meaning we don't check in sensitive data to our code repository

Also remember that we chose to build our full connection string in our `Startup` class using:

- Non sensitive Connection String (Stored in ***appSettingsDevelopment.json***)
- Our User ID, stored in a User Secret called: `UserID`
- Our Password, stored in a User Secret called: `Password`

Setting Up Config In Azure

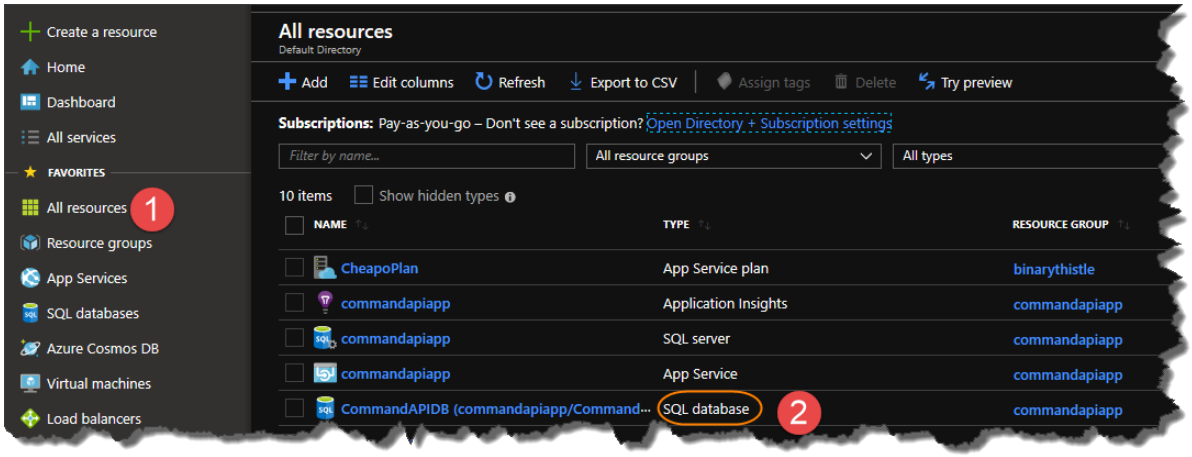
When you deploy a .NET Core app to an Azure API App, it sits on top of its configuration layer that we access via the .NET Core configuration API in exactly the same way as we have done to date. So setting up our production environment will:

- Require some simple config settings in our API App
- Require no code changes in our app, (there would be something very wrong if we needed to change our code to move into production – that should all be handled by configuration.

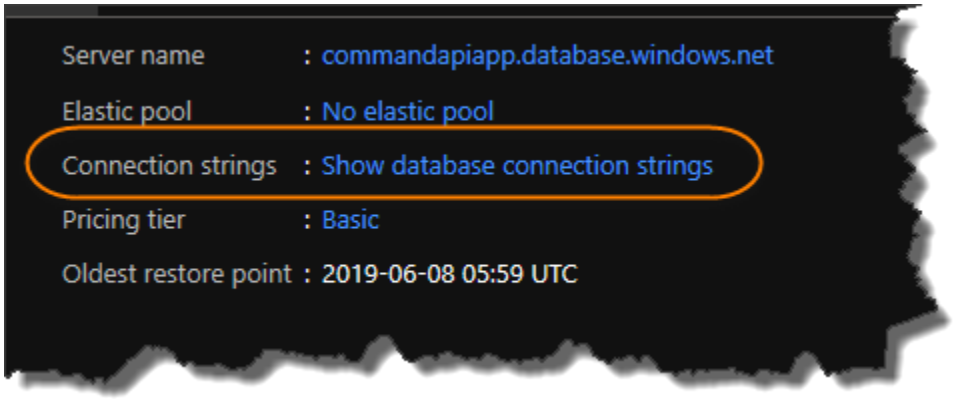
Get Our Connection String

First we want to get our connection string to our Azure SQL Database, to do so:

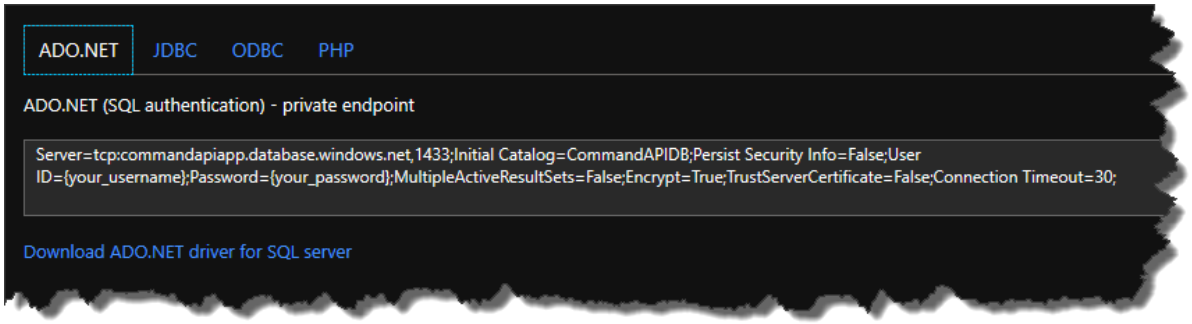
1. On the Azure Portal Home Page click *All resources*
2. In the resulting list find your SQL database, (not the SQL Server), and click it



In the resulting SQL Database screen, click on: Show database connection strings



You'll see something similar to the following, (ensure you have the ADO.NET tab selected):

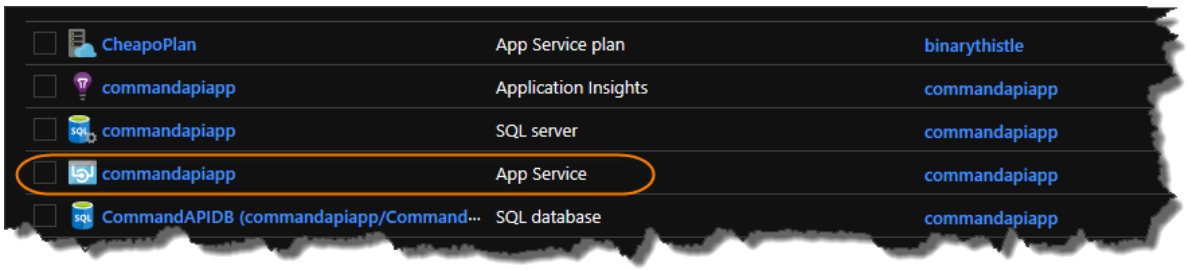


As we are going to "build" our connection string using a base, (non-sensitive), connection string as we have done previously, copy this string and remove the User ID and Password components. You should end up with something like:

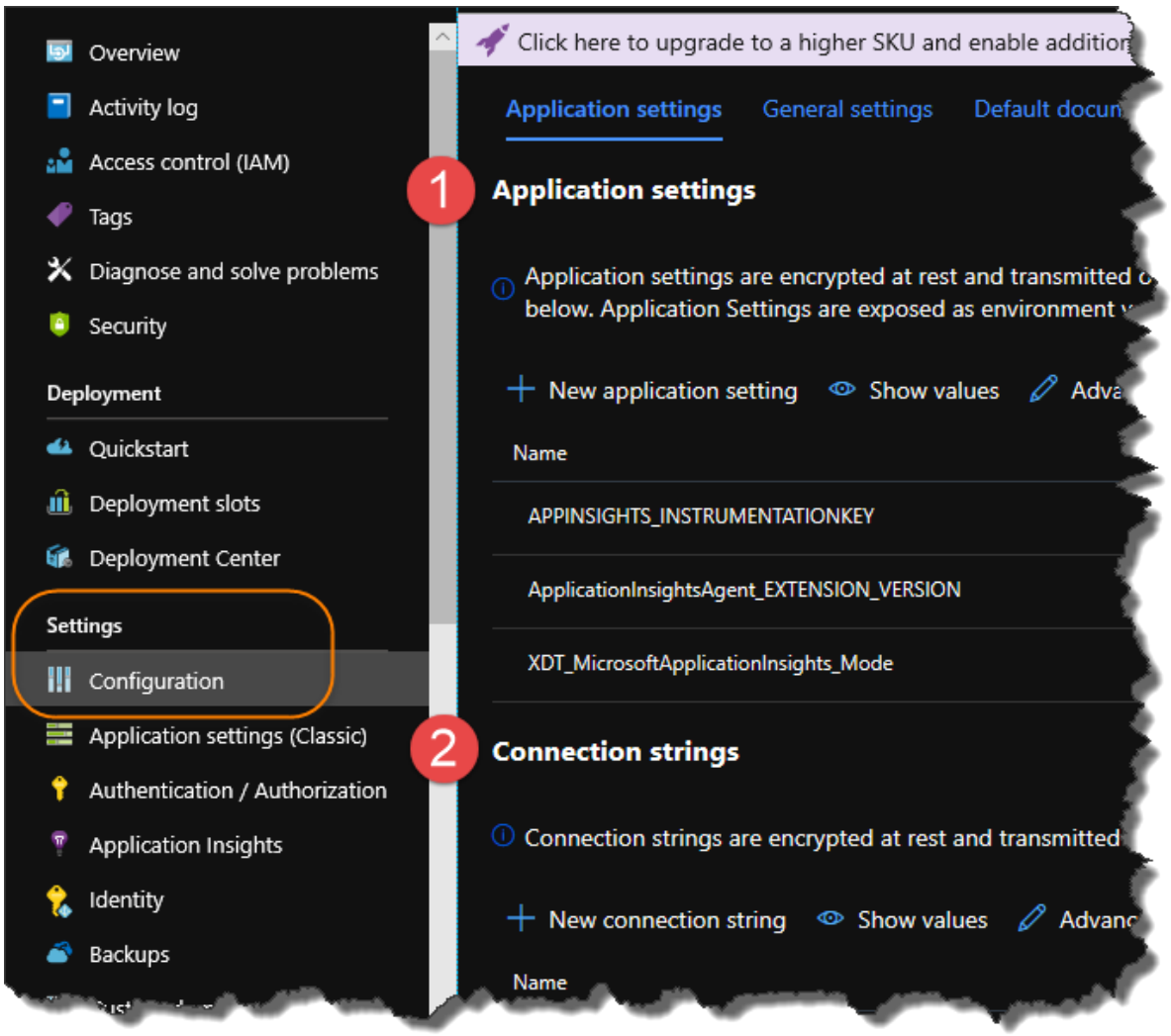
```
Server=tcp:commandapiapp.database.windows.net,1433;Initial
Catalog=CommandAPIDB;Persist Security
Info=False;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=Fa
lse;Connection Timeout=30;
```

Configure Our Connection String

Ok, so go back to your list of Azure resources and select your API App Service:



On the resulting screen, select *Configuration* in the *Settings* section:

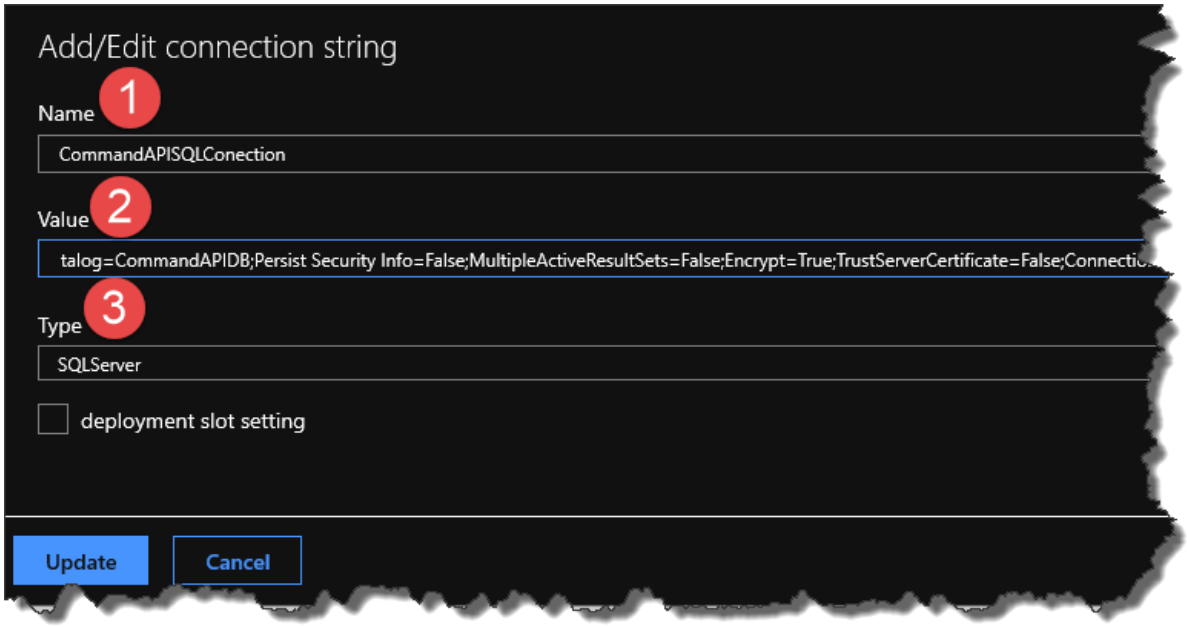


You'll see there are 2 sections here for use to play with:

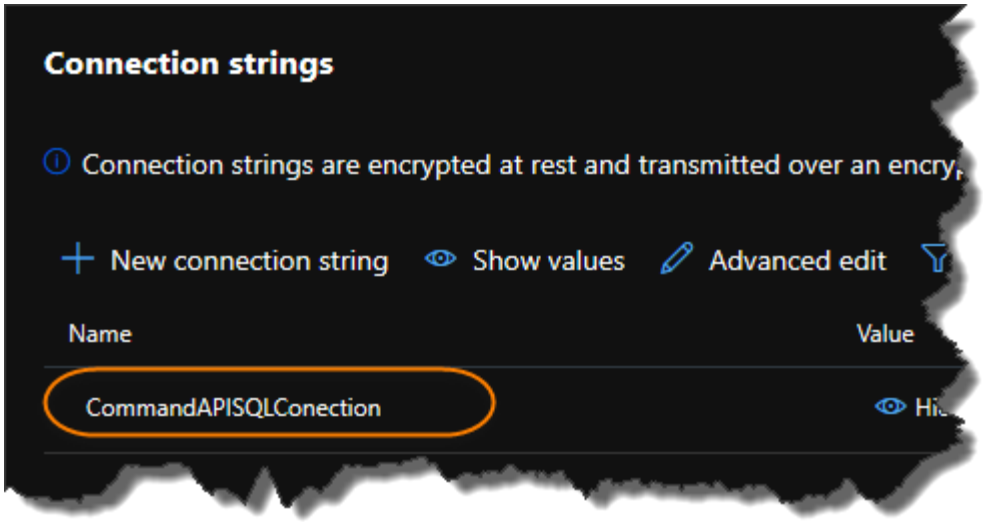
1. Application Settings
2. Connection Strings

We are going to add the connection string we obtained above to the Connection strings settings here, to do so, click: *+ New connection string*, in the resulting form enter:

1. Connection String Name, (this should be the same name as our existing connection string)
2. The connection string we obtained in the last step, (minus the User ID & Password attributes)
3. Set the type to SQL Server



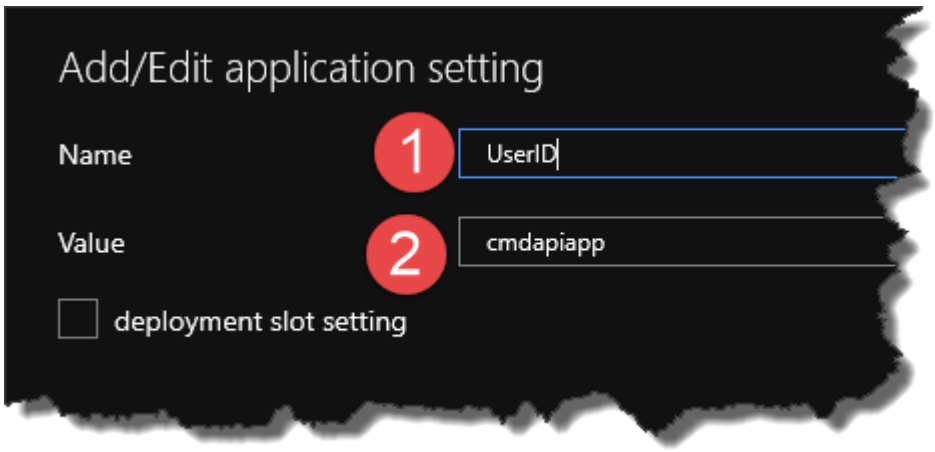
Click Update, and you'll see the connection string has been added to our collection:



Configure Our DB User Credentials

We're going to add our User ID and Password in a very similar way, except this time, we'll add these items to the *Application Settings* section of our Azure Configuration. To add our User ID, click + New application setting, in the resulting form enter:

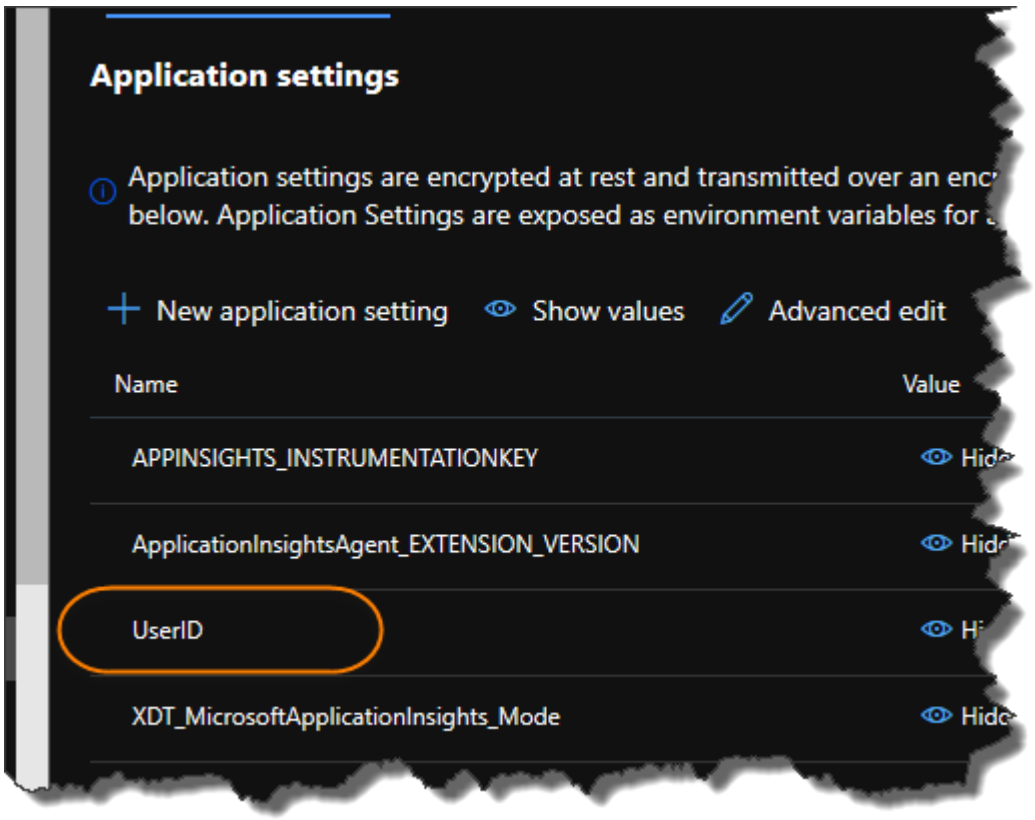
1. Name of our setting, (this should be the same as our User Secret name for User ID)
2. Value. This is the user account you created when you set up the SQL Server



A Couple of things to note:

- There is no space between User and ID (this is exactly the same as our local User Secret)
- The user account we're using (in this case cmdapiapp), is a DB admin account – usually I'd suggest creating a specific account to access our DB, but for brevity let's use this for now.

Click Update, and you'll see the new UserID application setting:



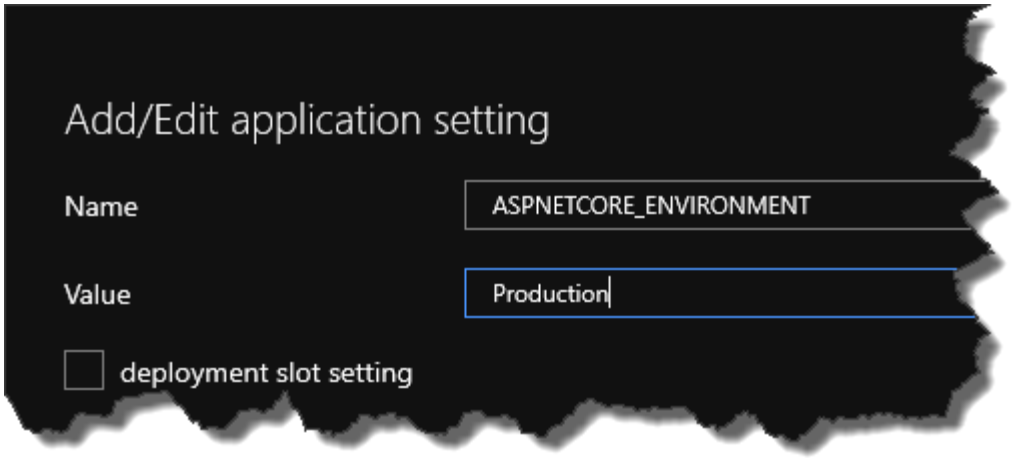
Learning Opportunity: Add a second *Application setting* for our ***Password***. This should follow the same process as UserID.

CONFIGURE OUR ENVIRONMENT

We want to set our runtime environment to “Production”, we do this simply by adding another Application setting with:

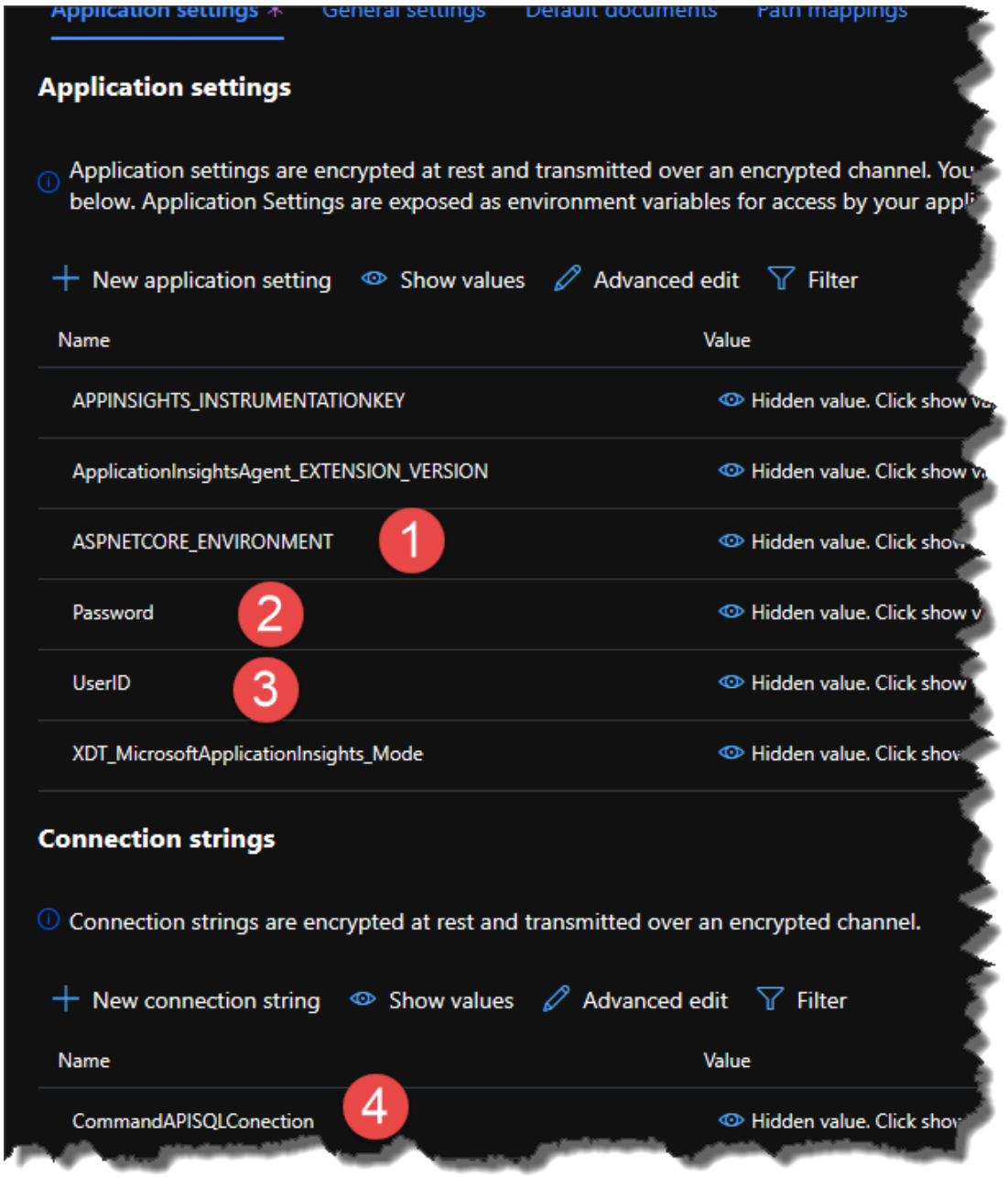
- A Name of: ASPNETCORE_ENVIRONMENT
- A Value of: Production

As shown below:

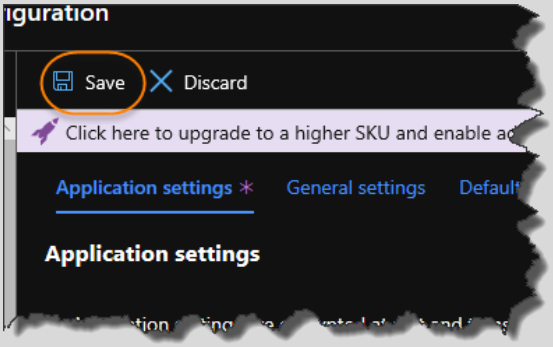


Click Update and you should have:

1. Application settings: ASPNETCORE_ENVIRONMENT
2. Application settings: Password
3. Application settings: UserID
4. Connection strings: CommandAPISQLConection (note connection is spelt wrong!)



Warning! Every time you make a configuration change you need to Save it! See below:

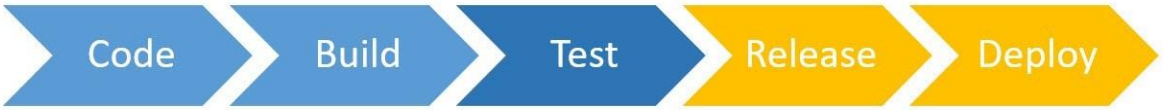


Make sure you click Save to apply your changes (when starting out with this stuff I didn't and spent a lot time trying to understand what as wrong!)

Celebration Check Point: You have just set up all your Azure Resources and have configured them ready for our deployment!

Completing Our Pipeline

At last! We create the final piece of the puzzle in our CI/CD pipeline: Deploy.



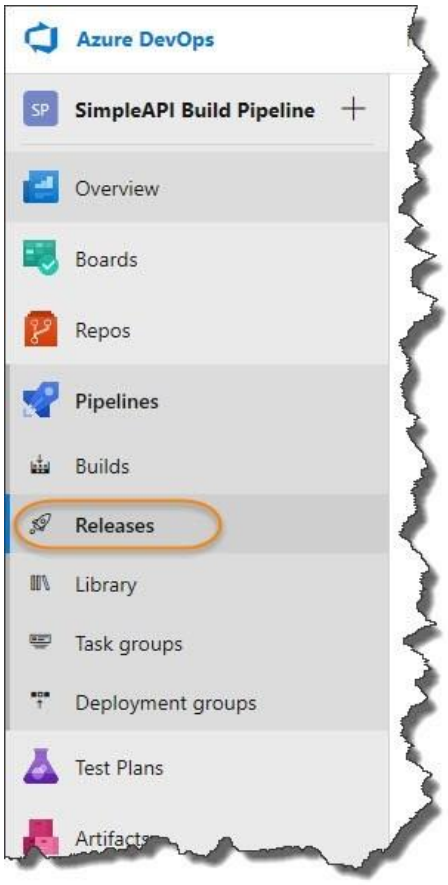
So a quick recap on our CI/CD Pipeline so far:

- We created what Azure DevOps calls a *Build Pipeline* that does the following:
  - Builds Our Projects
  - Runs our unit tests
  - Packages our release

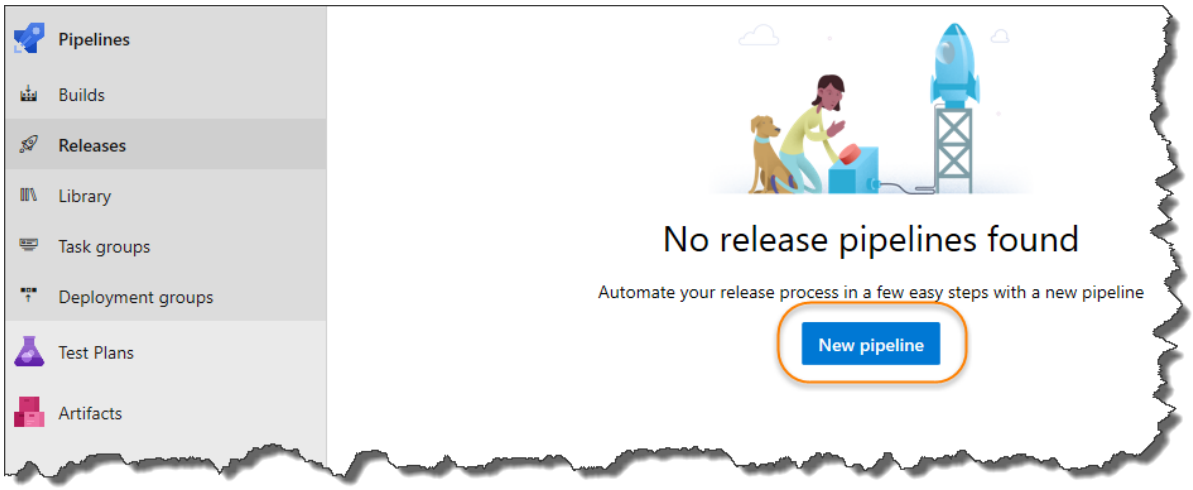
What we now need to do in Azure DevOps is create a *Release Pipeline* that takes our package release and deploys it to Azure. So basically our full CI/CD Pipeline = Azure DevOps Build Pipeline + Azure DevOps Release Pipeline.

Creating Our Azure DevOps Release Pipeline

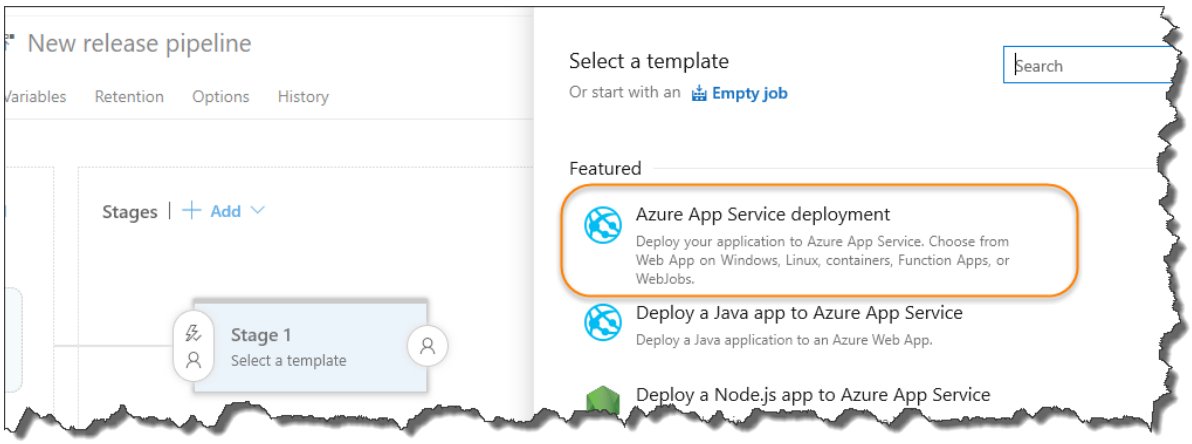
Back in Azure DevOPs, click on Pipelines -> Releases



The click on New Pipeline:

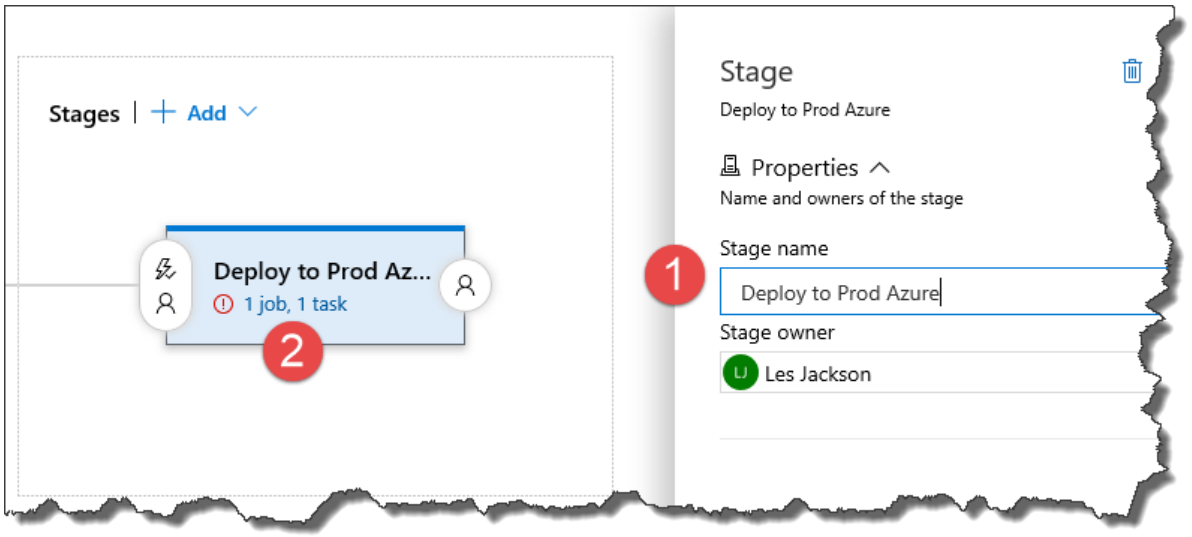


On the next screen, select and *Apply* the *Azure App Service deployment* template:



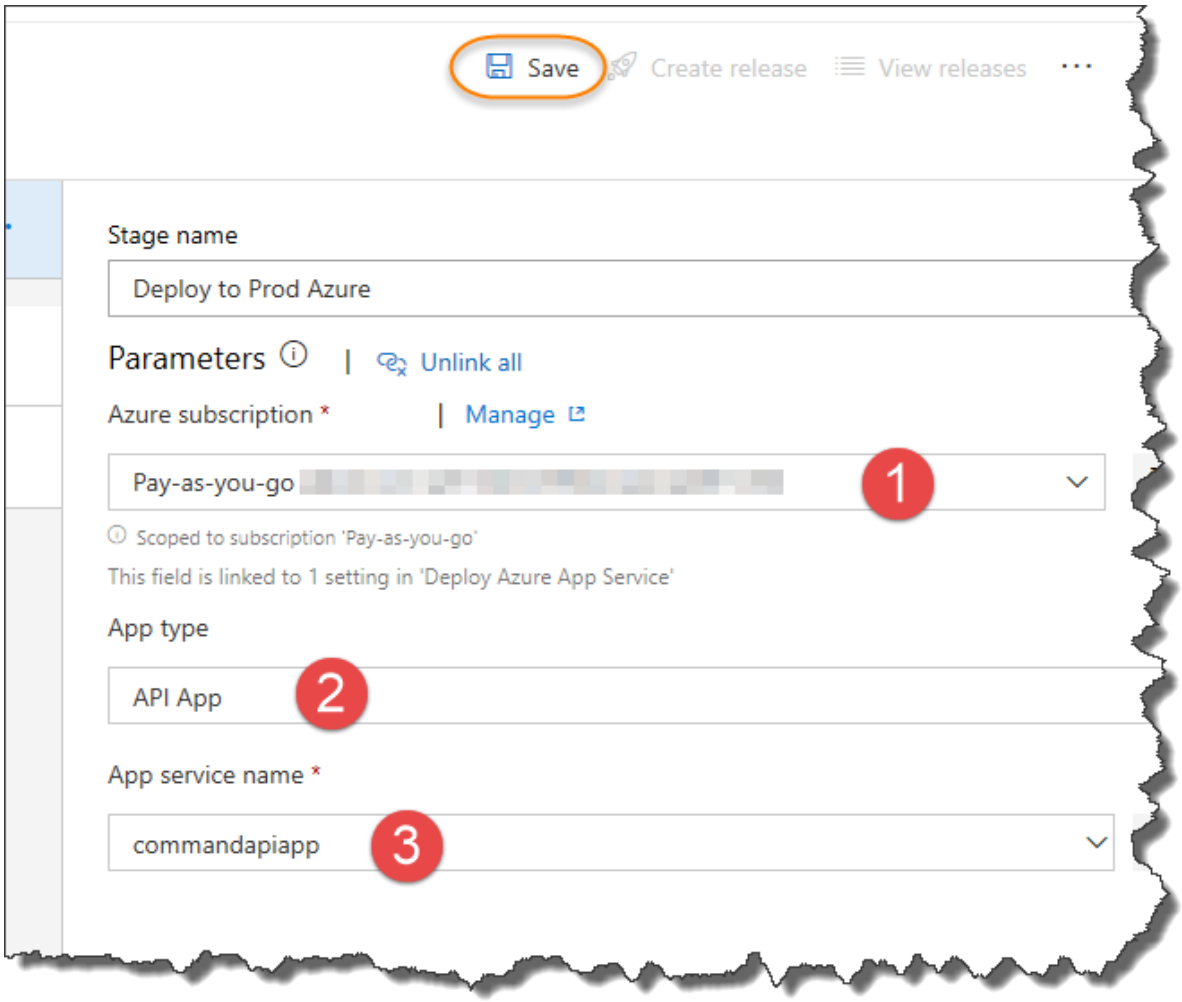
In the “Stage” widget:

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

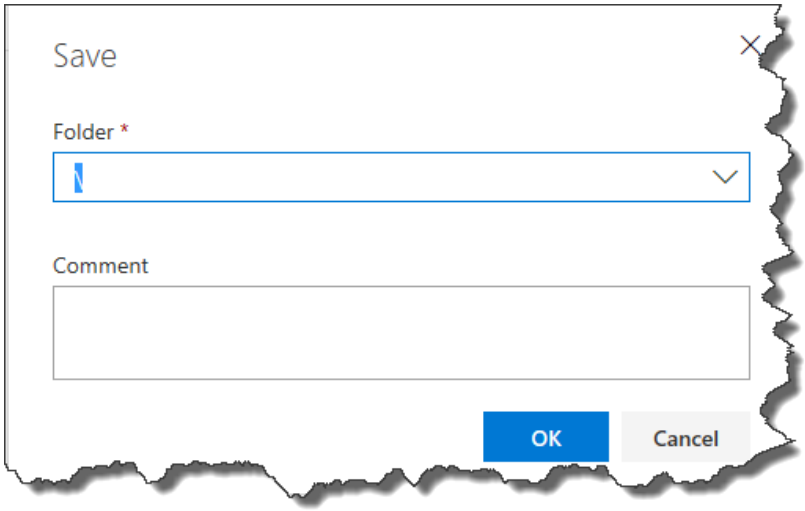


Here we need to:

1. Select Our Azure Subscription (you will need to “authorise” Azure DevOps to use Azure)
2. App Type (remember this is an API App)
3. App Service Name (All of your API Apps will be retrieved from Azure – select the one you created above)

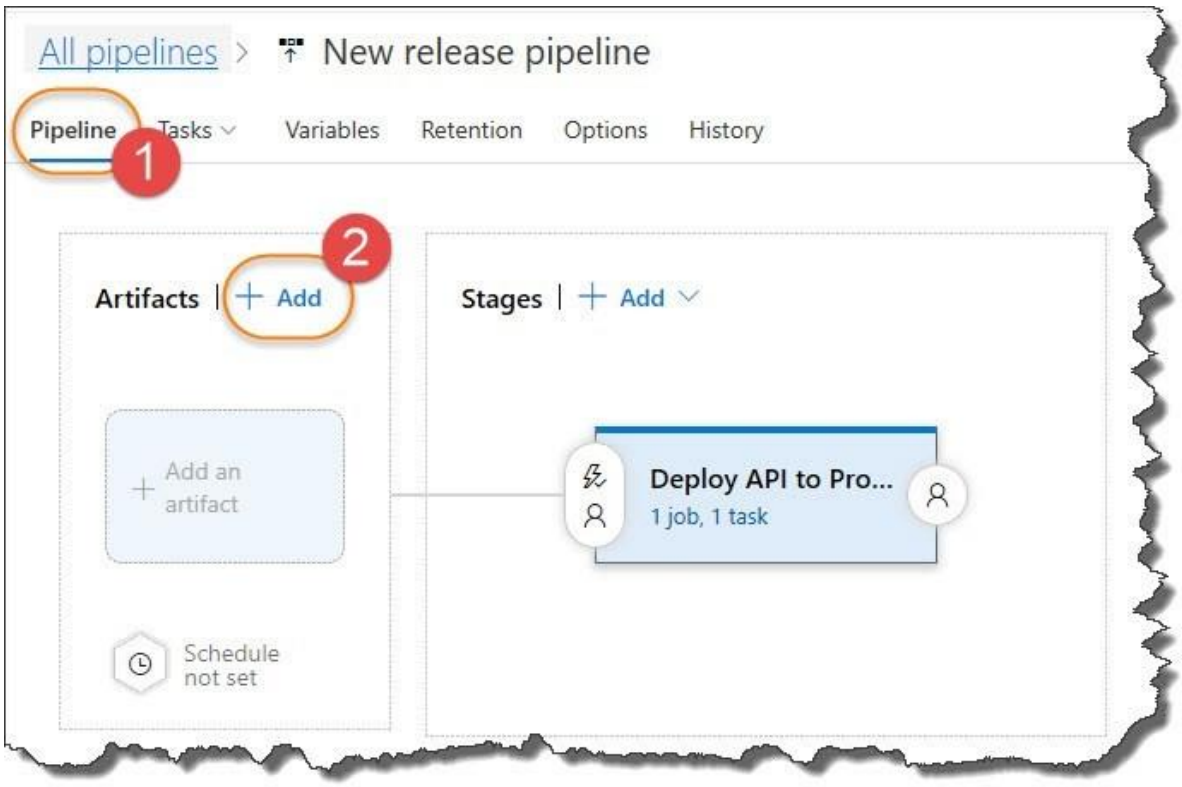


Don’t forget to *Save*. When you do you’ll be presented with:



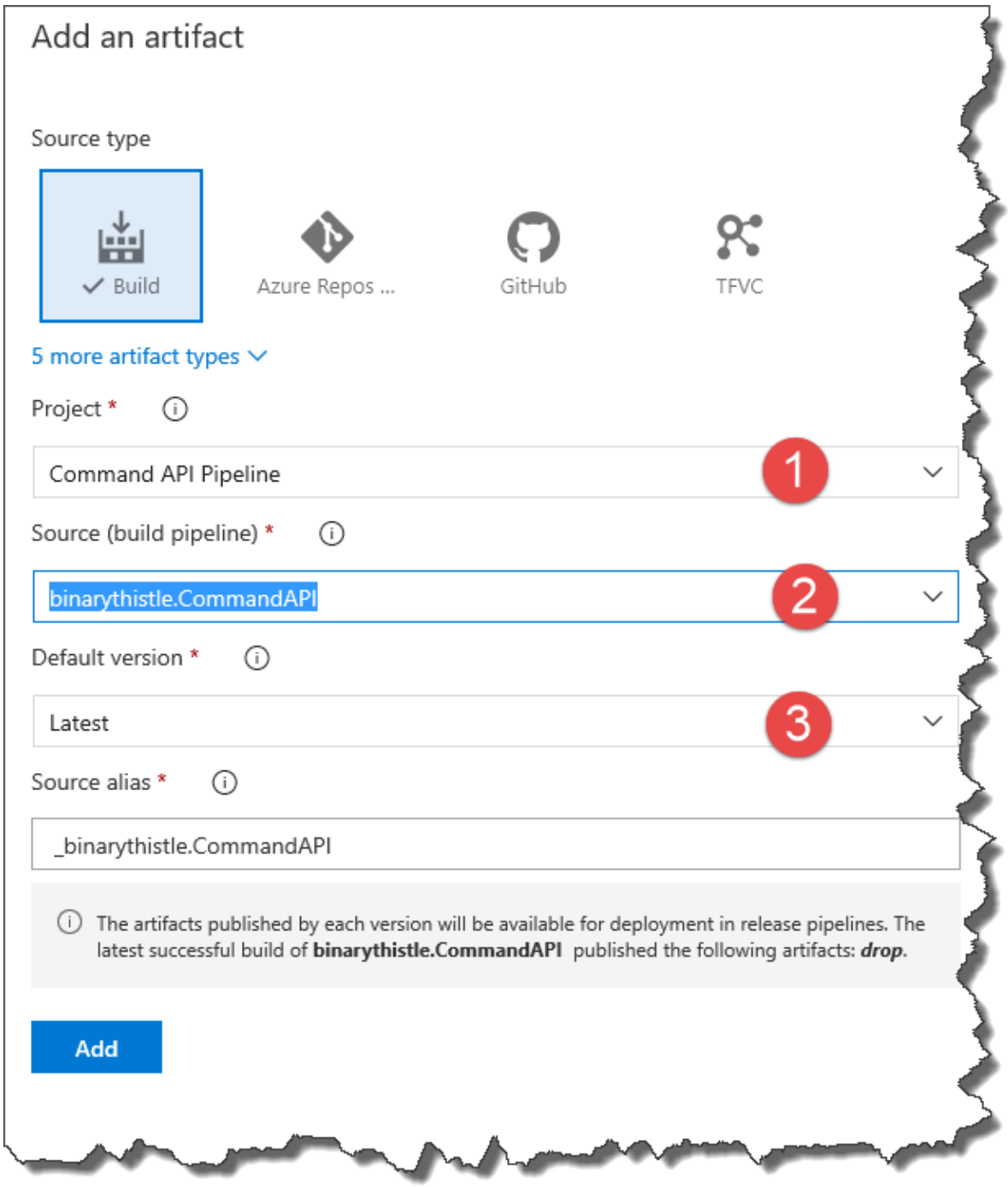
Just click OK.

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

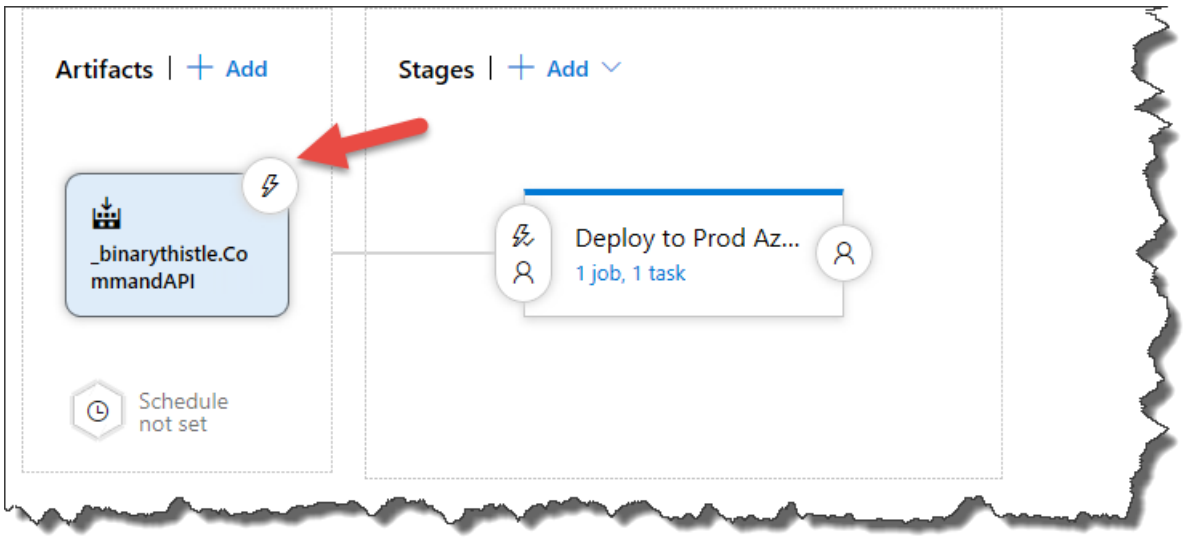


Here you will need to provide:

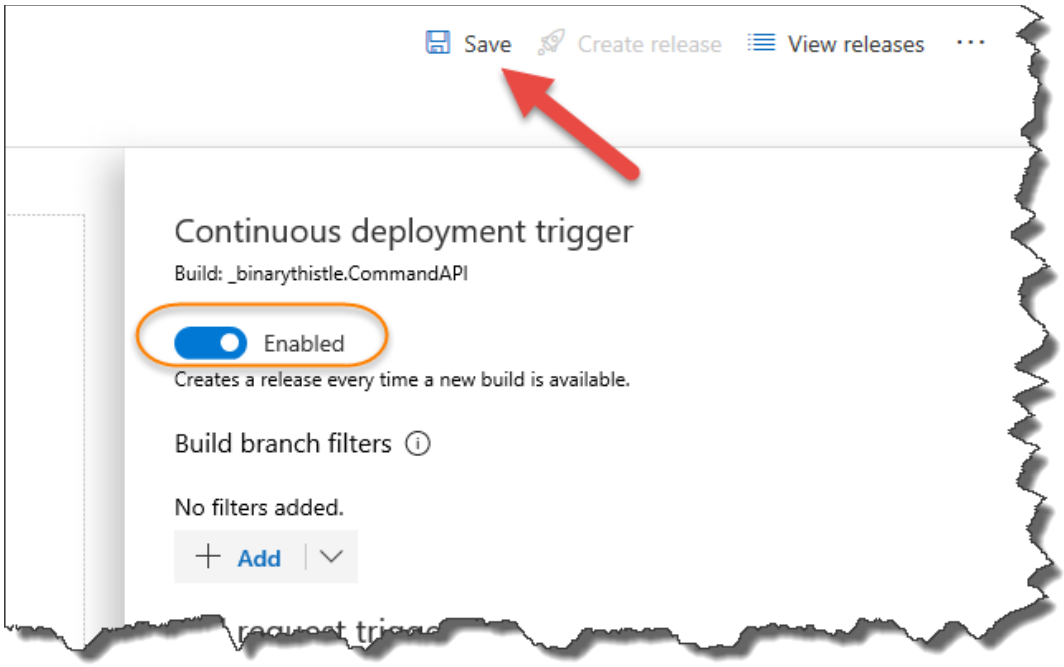
1. The Project (this should be pre-selected)
2. The Source Pipeline (this is our build pipeline we created previously)
3. Default version (select “Latest” from the drop down)



Click Add. The click on the lightening bolt on the newly created Artefact node:

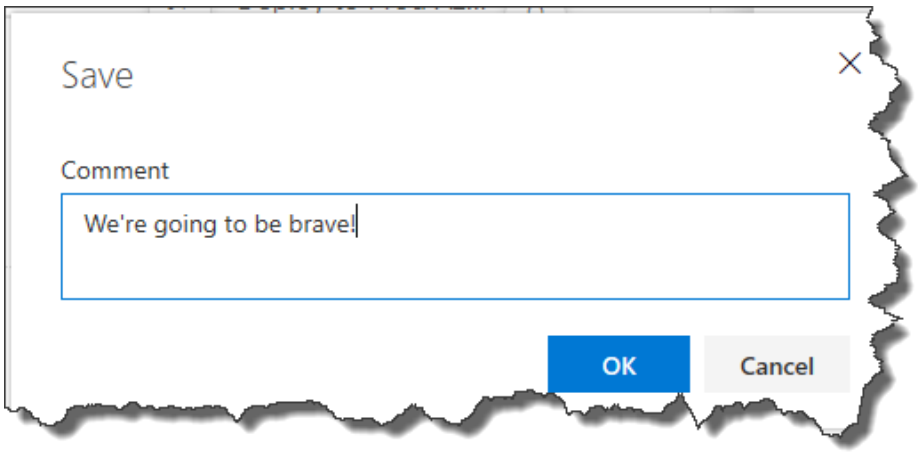


In the resulting pop up, ensure that Continuous deployment trigger is **enabled**, then click *Save*.

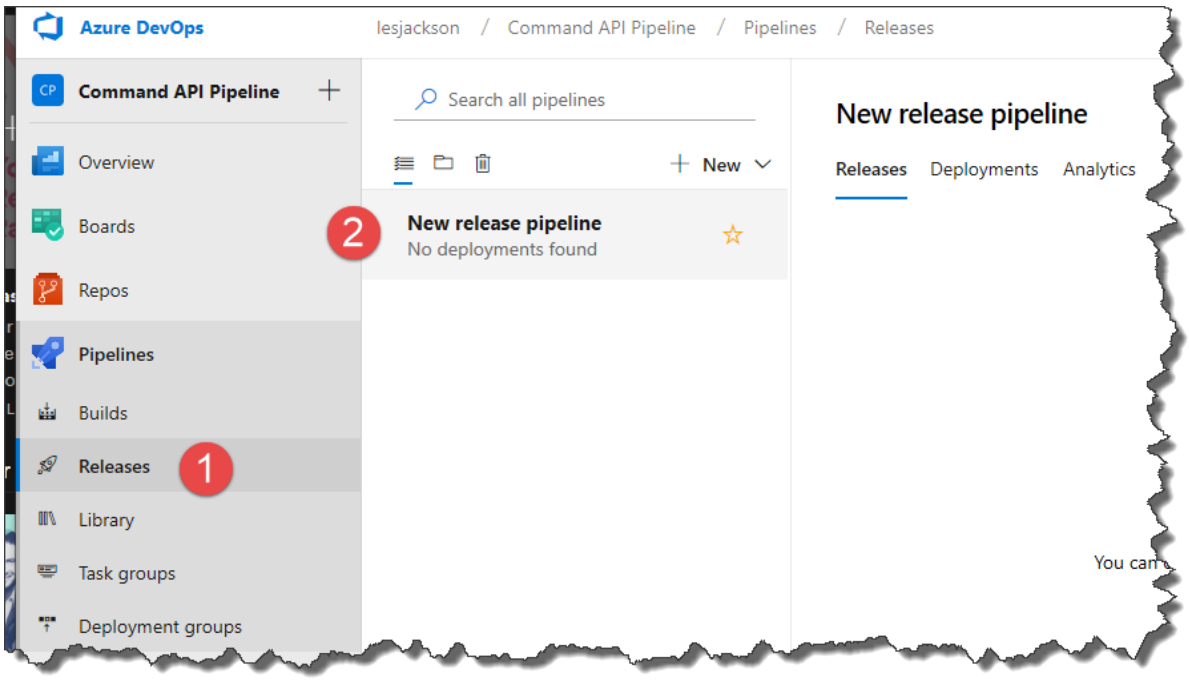


Note: it is this setting that switches us from Continuous Delivery to Continuous Deployment…

You'll get asked to supply a comment when turning this on, do so if you like:



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:



Pull The Trigger – Continuously Deploy

Ok the moment of truth. If we have set everything up correctly all we need to do now to test our entire CI/CD pipeline end to end is to perform another code commit to GitHub, which will trigger the *Build Pipeline*, then as we've just configured, the *Release Pipeline* which will deploy to Azure…

Wait! What About EF Migrations?

Just before you do that – cast your mind back to Chapter 6 where we set up our DB Context and performed a database migration at the command line:

```
dotnet ef database update
```

Nowhere in our CI/CD pipeline have we accounted for this step, where we tell Azure it has to create the necessary schema in our SQL DB. There are a few ways we can do this, but the simplest is to update the `Configure` method in out `Startup` class.

This approach means that migrations will be applied when the app is started for the 1st time.

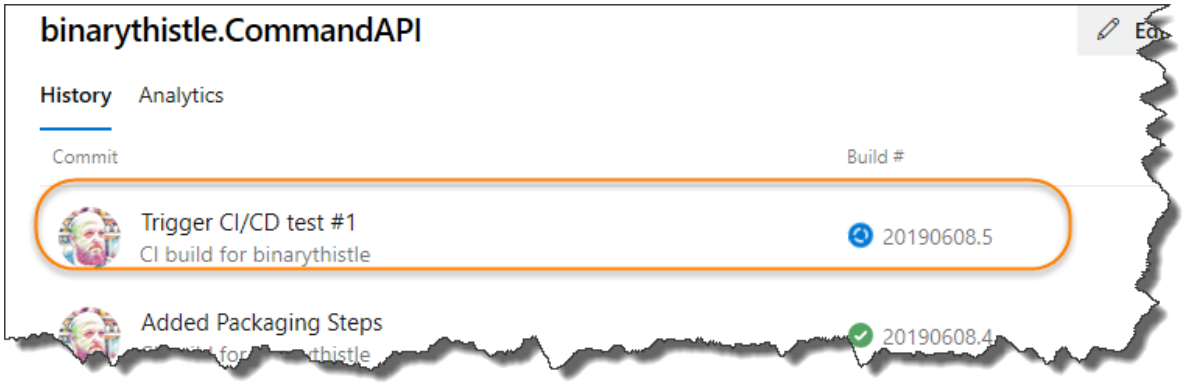
In VS Code, open the `Startup` class and make the following alterations to the `Configure` method:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
CommandContext context)
{
  context.Database.Migrate();
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }
  app.UseMvc();
}
```

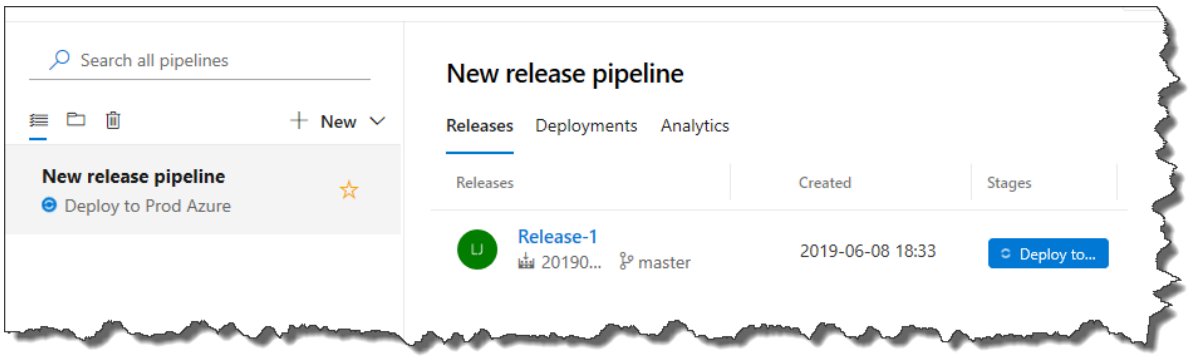
For clarity, the `Configure` method changes are highlighted below:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, CommandContext context)
{
    context.Database.Migrate();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseMvc();
}
```

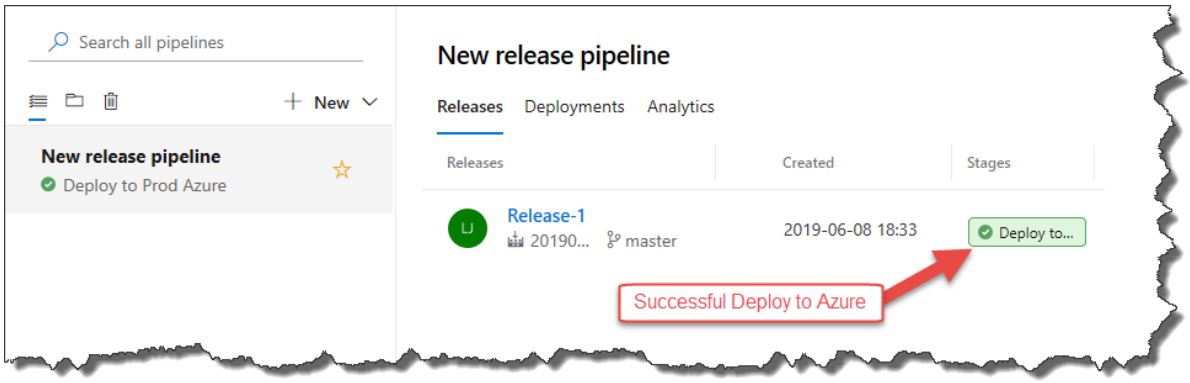
Save your changes and: Add, Commit and Push your code as usual, this should trigger the build pipeline…



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

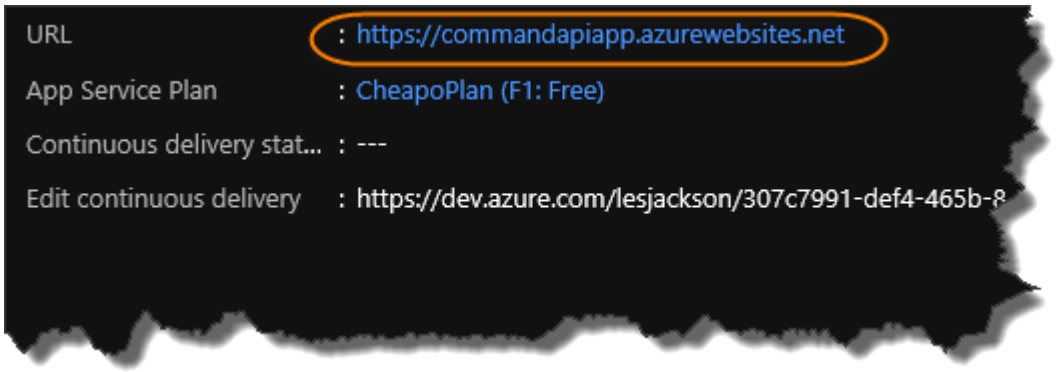


You'll see the Release Pipeline attempting to deploy to Azure… And eventually it should deploy, (you may need to navigate away from the Release Pipeline and back again):



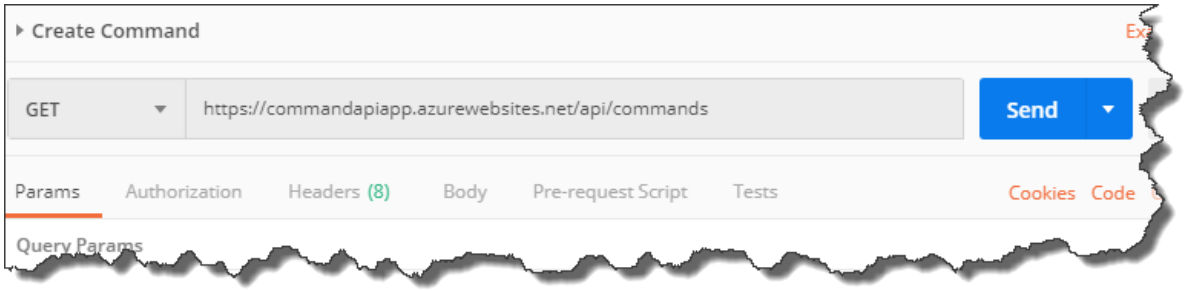
And now the moment of truth, let's see if our API is working, first obtain the base app URL from Azure:

- Click All resources
- Select you API App (App Service type)



Note: yours will be named differently…

Now fire up Postman, and prepare to make a GET request to retrieve all our commands (we won't have any yet):

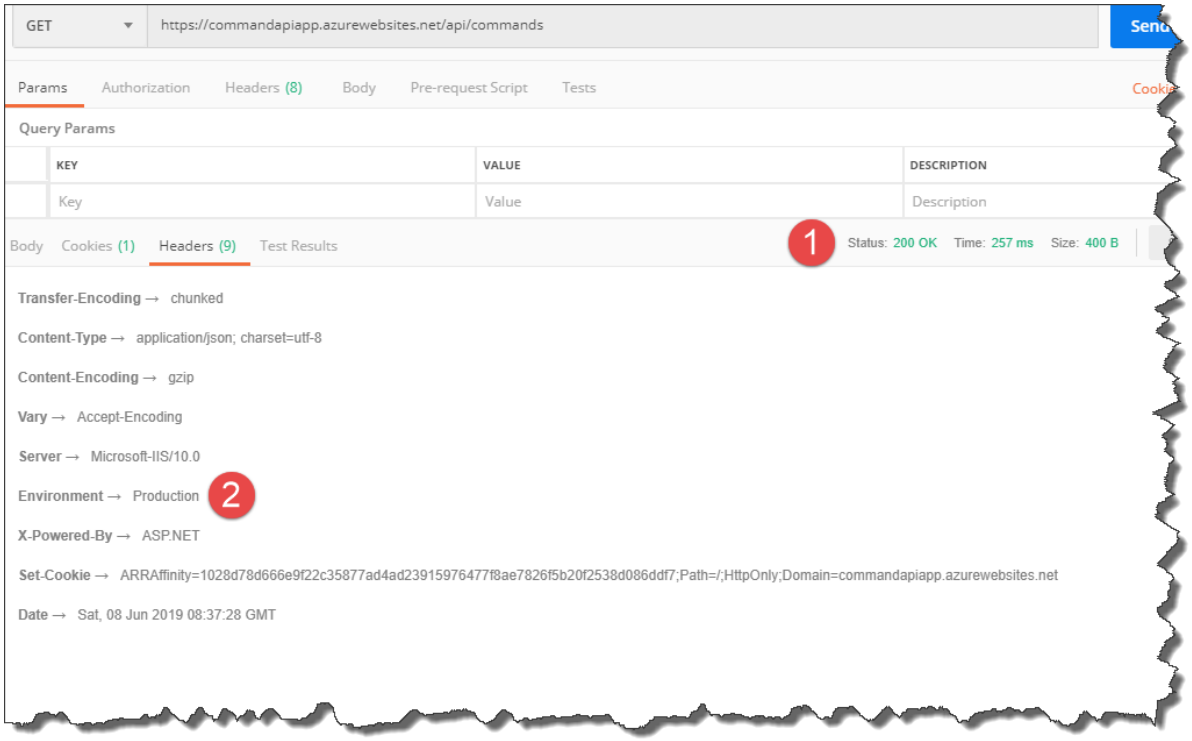


Remember to append: `/api/commands` to the base URL

Then click Send.

If the deployment and Azure configuration were successful, you'll get an empty payload response, but you should get:

- 200 OK Result
- Looking at the headers there should be our custom `Environment` header with a value of `Production`



Celebration Check Point: Rad[22]! Our API is deployed and working in our Production Azure environment, moreover it's there via process of Continuous Integration / Continuous Deployment!

Double Check

Just to double check everything, let's make a POST request to create some data.

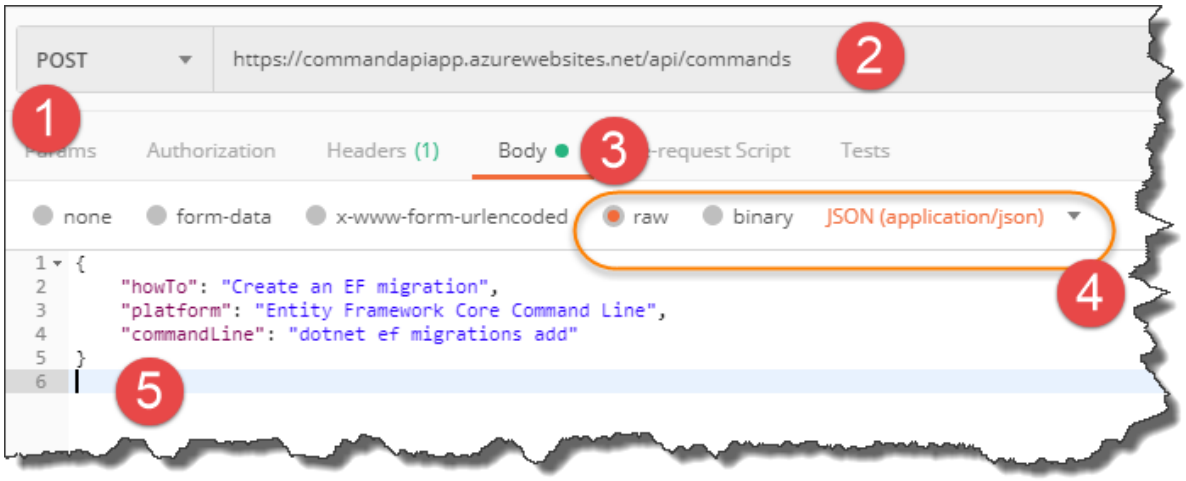
[22] Children of the 90s' will get this superlative

Using the JSON string below:

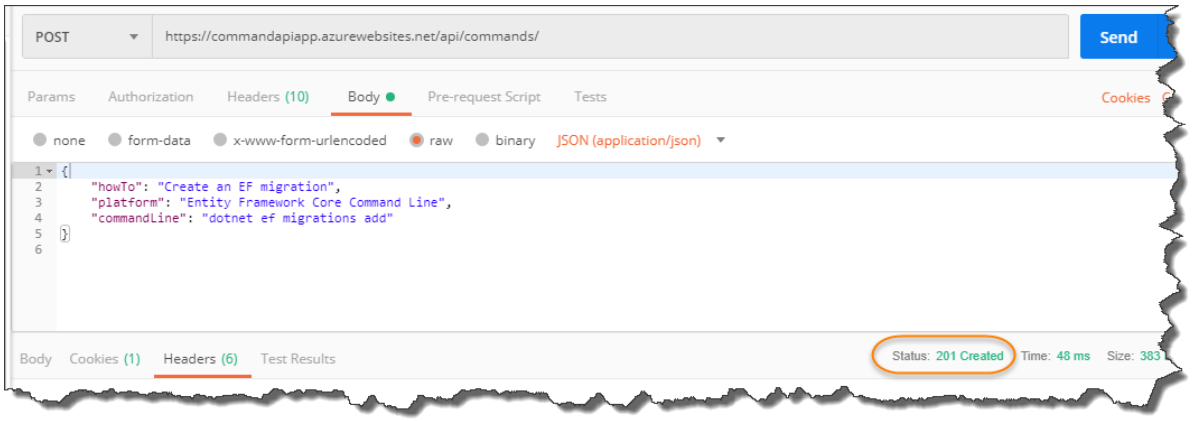
```
{
    "howTo": "Create an EF migration",
    "platform": "Entity Framework Core Command Line",
    "commandLine": "dotnet ef migrations add"
}
```

Create a new Postman request and set:

1. Request Verb to POST
2. The request URL is correct
3. Click Body
4. Select Raw and JSON(application/json) for the request body format
5. Paste the JSON into the body payload window



Finally, if you're brave enough click "Send" to make the request.



And again we have success!

EPILOGUE

Firstly, if you've made it all the way through, and followed all the steps then Well Done! I hope you found it a useful and entertaining exercise

For me, although writing has always formed a large part of my career, I've never written book before, so here are some of my thoughts on that:

- I thought taking my blog posts and other random works and tying them together in a book would take about 2 weeks. In reality it took well over 2 months…
- I am so grateful that I'm in a position where I could write a book, primarily because I was born into privilege, for which I am thankful and ashamed in equal measure. And by privilege, I don't mean that I or my family are rich, (we are not!), but that I was born healthy, to lovely parents, in a country at peace, and with the very rare privilege of a free university education.
- The are so many clever, creative people out there sharing their knowledge, that without them I'd not be able to complete such a book.
- Now I'm done, looking back there are lots of tweaks and potential additions…
- That I wouldn't write another one!
- Well, maybe I'll update this one…