The Complete ASP.NET Core API Tutorial

A Practical Step-by-Step Guide.

2nd Edition

Les Jackson

Updated for .NET Core 3.1

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

[February 2020 2nd 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.

ISBN: 978-1-64826-791-8

For Quynh

CONTENTS

Chapter 1 – Introduction 14
Why I wrote this book 14
2nd Edition 14
The Approach of This Book 14
Where can you get the code? 15
Conventions Used In This Book 15
Version of the .net core framework 15
Contacting the Author 16
Defects & Feature Improvements 16
Chapter 2 – Setting up your development environment 17
Chapter Summary 17
When Done, You Will 17
The 3 Amigos : Windows, Mac & Linux 17
Your Ingredients 17
Install VS Code 18
C# for Visual Studio Code 19
Insert GUID 20
Install .Net Core SDK 20
Install GIT 22
Name and Email 22
Install Docker [Optional] 22
What is Docker? 23
Docker Desktop Vs Docker CE 23
Post Installation Check 23
Docker Plugin For VS Code 25
Install PostgreSQL 25
Install DBeaver CE 25
DBeaver Vs pgAdmin 26
Install Postman 26
Trust Local Host Development Certs 26
Wrapping It Up 27
Chapter 3 – Overview of our API 28
Chapter Summary 28
When Done, You Will 28
What Is a REST API? 28

Our API 28
Payloads 29
5 Minutes On JSON 29
Chapter 4 – Scaffold Our API Solution 33
Chapter Summary 33
When Done, You Will 33
Solution Overview 33
Scaffold Our Solution Components 34
Creating Solution and Project Associations 37
Anatomy Of An ASP.NET Core App 40
The Program & Starup Classes 41
Chapter 5 – The 'C' in MVC 45
Chapter Summary 45
When Done, You Will 45
Quick Word On My Dev Set Up 45
Start Coding! 46
Call the Postman 49
What is MVC? 51
Model, View, Controller 51
Our Controller 53
1. Using Directives 56
2. Inherit from Controller Base 56
3. Set Up Routing 56
4. APIController Attribute 57
5. HttpGet Attribute 57
6. Our Action Result 57
Source Control 58
Git & GitHub 59
Setting Up Your Local Git Repo 59
.gitignore file 60
Track and Commit Your Files 62
Set Up Your GitHub Repo 64
Create a GitHub Repository 64
So What Just Happened? 67
Chapter 6 – Our Model 68
Chapter Summary 68
When Done, You Will 68

Our Model 68
Tying it Together 70
PostgreSQL Database 70
Connecting with DBeaver 74
Entity Framework Command Line Tools 78
Create Our DB Context 78
Update appsettings.json 81
Revisit the Startup Class 86
Create & Apply Migrations 89
Code First Vs Database First 89
Add Some Data 92
Return “Real” Data 96
Wrapping Up The Chapter 99
Redact Our Login and Password 100
Chapter 7 – Environment Variables & User Secrets 102
Chapter Summary 102
When Done, You Will 102
Environments 102
Our Environment Set Up 103
The Development Environment 103
So What? 105
Make the Distinction 105
Order of Precedence 106
It’s Time to Move 107
User Secrets 109
What Are User Secrets? 109
Setting Up User Secrets 110
Deciding Your Secrets 112
Where Are They? 113
Code it Up 113
Wrap It Up 116
Chapter 8 – Unit Testing & Completing Our API 118
Chapter Summary 118
When Done, You Will 118
What Is Unit Testing 118
Protection Against Regression 118
Executable Documentation 119

Characteristics of a Good Unit Test 119
What to Test? 119
Unit Testing Frameworks 120
Arrange, Act & Assert 120
Arrange 120
Act 120
Assert 120
Write Our First Tests 120
Testing Our Model 122
Don't Repeat Yourself 127
Test Our Existing Controller Action 130
DBContext 132
Finishing GetCommandItems Tests 135
Test Driven Development 139
What Is Test Driven Development? 140
Why Would You Use TDD? 140
Revisit Our REST Actions 140
Action 2: Get A Single Resource 141
What Should We Test 141
Test 2.1 Invalid Resource ID – Null Object Value Result 141
Test 2.2 Invalid Resource ID – 404 Not Found Return Code 143
Test 2.3 Valid Resource ID – Check Correct Return Type 144
Test 2.4 Valid Resource ID – Correct Resource Returned 145
Action 3: Create a New Resource 145
What Should We Test 146
Test 3.1 Valid Object Submitted – Object Count Increments by 1 146
Test 3.2 Valid Object Submitted – 201 Created Return Code 146
Action 4: Update an Existing Resource 147
What Should We Test 147
Test 4.1 Valid Object Submitted – Attribute is Updated 147
Test 4.2 Valid Object Submitted – 204 Return Code 148
Test 4.3 Invalid Object Submitted – 400 Return Code 148
Test 4.4 Invalid Object Submitted – Object Remains Unchanged 148
Action 5: Delete an Existing Resource 149
What Should We Test 149
Test 5.1 Valid Object ID Submitted – Object Count Decrements by 1 150
Test 5.2 Valid Object ID Submitted – 200 OK Return Code 150

Test 5.3 Invalid Object ID Submitted – 404 Not Found Return Code 150
Test 5.4 Valid Object ID Submitted – Object Count Remains Unchanged 151
Wrap It Up 151
Chapter 9 – The CI/CD Pipeline 152
Chapter Summary 152
When Done, You Will 152
What is CI/CD? 152
CI/CD or CI/CD? 152
What's The Difference? 152
So which is it? 153
The Pipeline 153
What is "Azure DevOps"? 154
Alternatives 154
Technology In Context 154
Create a Build Pipeline 155
Update azure-pipelines.yml To Use .NET Core 3.1 162
What Just Happened? 167
Azure-Pipelines.yml File 167
Triggering a Build 169
Revisit azure-pipelines.yml 171
Another VS Code Extension 172
Running Unit Tests 173
Breaking Our Unit Tests 176
Testing – The Great Catch All? 180
Release / Packaging 181
Wrap It Up 183
Chapter 10 – Deploying to Azure 184
Chapter Summary 184
When Done, You Will 184
Creating Azure Resources 184
Create Our API App 184
Create Our PostgeSQL Server 192
Connect & Create Our DB User 200
Revisit Our Dev Environment 202
Setting Up Config In Azure 202
Configure Our Connection String 202
Configure Our DB User Credentials 204

Configure Our Environment 206
Completing Our Pipeline 208
Creating Our Azure DevOps Release Pipeline 208
Pull The Trigger – Continuously Deploy 215
Wait! What About EF Migrations? 215
Double Check 218
Chapter 11 – Securing Our API 220
Chapter Summary 220
When Done, You Will 220
What We're Building 220
Our Authentication Use Case 220
Overview of Bearer Authentication 220
Build Steps 221
Registering our API in Azure AD 222
Create a New AD? 223
Register Our API 224
Expose Our API 229
Update our Manifest 231
Add Configuration Elements 233
Update our Project Packages 234
Updating our Startup Class 234
Update Configure Services 234
Update Configure 235
Update Our Controller 236
Register Our Client App 238
Create a Client Secret 239
Configure API Permissions 241
Create Our Client app 245
Our Client Configuration 245
Add Our Package References 247
Client Configuration Class 247
Finalise Our Program Class 250
Updating for Azure 255
Client Configurations 259
Deploy Our API to Azure 259
Epilogue 262

Intentionally blank

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)!

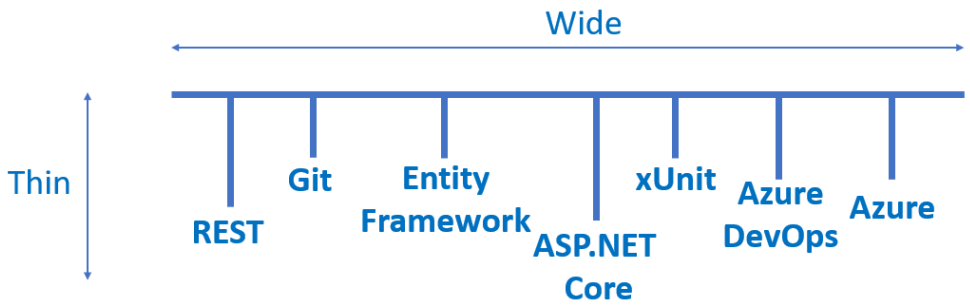
2ND EDITION

This is the 2nd Edition of the book. Having taken a Lean Start Up approach, (releasing versions as-is when they were ready), I received feedback on each of those to make each successive version better. With the release of .Net Core 3.1 in November 2019, it seemed like the perfect time to release the 2nd Edition. I've also included a much-needed chapter on Securing APIs, as well as moving the database backend from Microsoft SQL Server to PostgreSQL which I feel is much more in keeping with ethos of platform ubiquity[2].

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!



[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

[2] I still love SQL Server though!

Les's Personal Anecdote: The first time I tried, (or even heard of), a tasting menu was in a Las Vegas casino, (I think it was 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.

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:

Main Solution Repository (API & Unit Tests)

https://github.com/binarythistle/Complete-ASP.NET-Core-API-Tutorial-2nd-Edition-Net-Core-3.1

Secure Daemon Client Repository

https://github.com/binarythistle/Secure-Daemon-Client

Conventions Used In This Book

The following style conventions are use in this book:

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, (Feb 2020), I'm using version 3.1 of the .Net Core Framework.

Contacting the Author

You can contact me through the following channels:

- les@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: les@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[3], 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.

Additionally, as mentioned in the intro, for this second edition I've moved to PostgeSQL as the database backend which is available natively on Windows, Linux and OSX. I will however be running it as a Docker container, but more of that later…

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.	Free	Writing code! **Note:** This just my personal preference, you	Cross-platform

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

			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 3.1 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
PostgreSQL [Get it here natively] [Get it here for Docker]	Local Database	Free	We'll use this as our local development / test database.	Cross-platform or Docker image
DBeaver CE [Get it here]	Database independent management tool	Free	Writing and executing SQL queries, setting up DB users etc.	Cross-platform
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
Docker Desktop / Docker CE [Get it here - Desktop] [Get it here - CE]	Containerisation Platform (Run Docker containers)	Free	**[Optional]** I use Docker to quickly spin up and run a PostgeSQL database without the need to install it, (PostgreSQL), locally on my desktop.	Cross-platform: Docker Desktop - Windows & OSX Docker CE - Linux
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

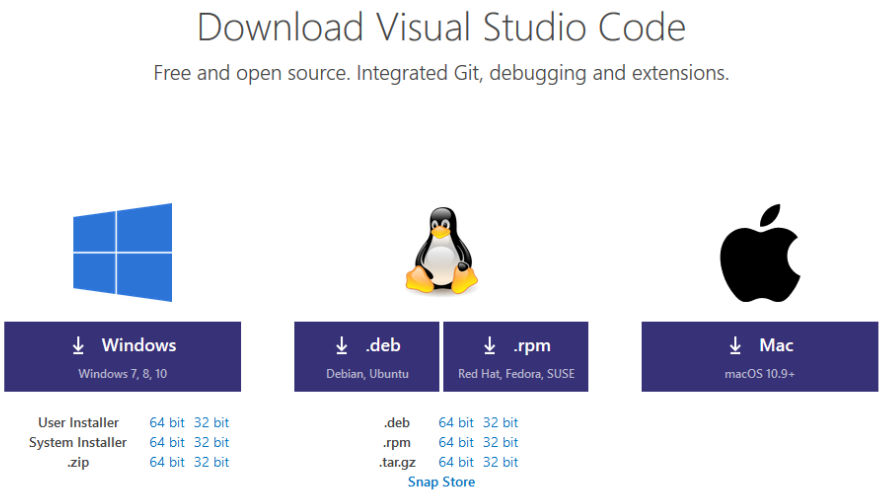
I'm suggesting Visual Studio Code, (referred to now on only as VS Code), as the text editor of choice for following this book as it has some nice features, e.g.: IntelliSense code-completion, syntax highlighting, integrated command / terminal, git integration, debug support etc…

[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/

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!)

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…

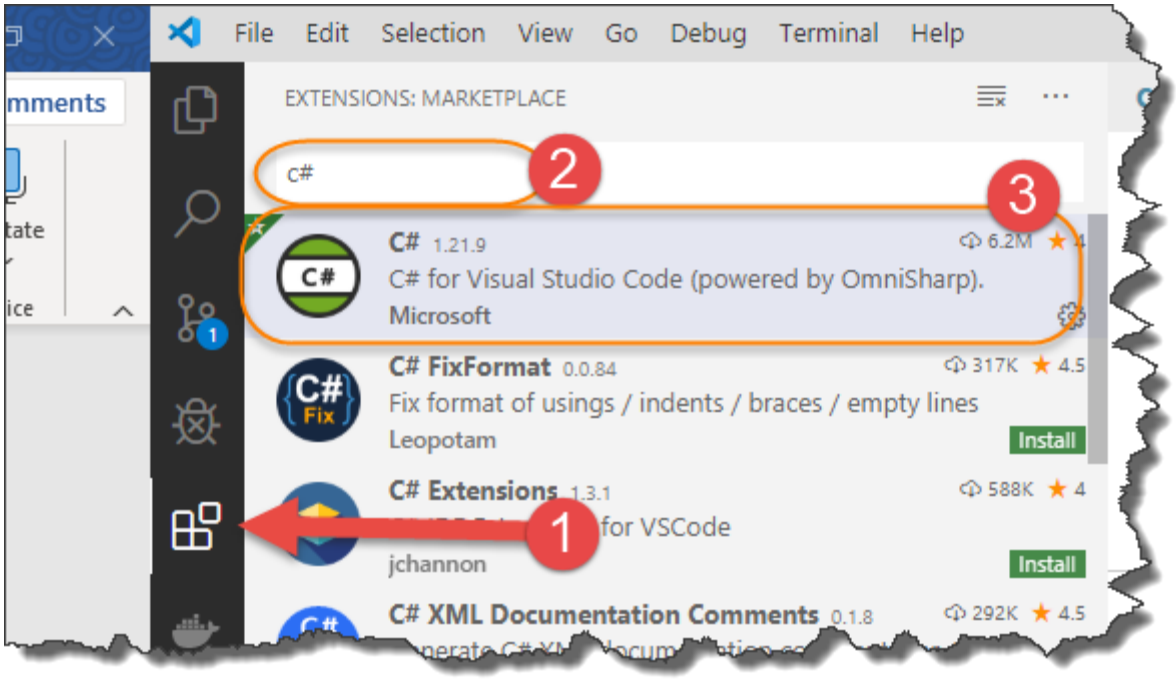
C# for Visual Studio Code

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 toolbar 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.

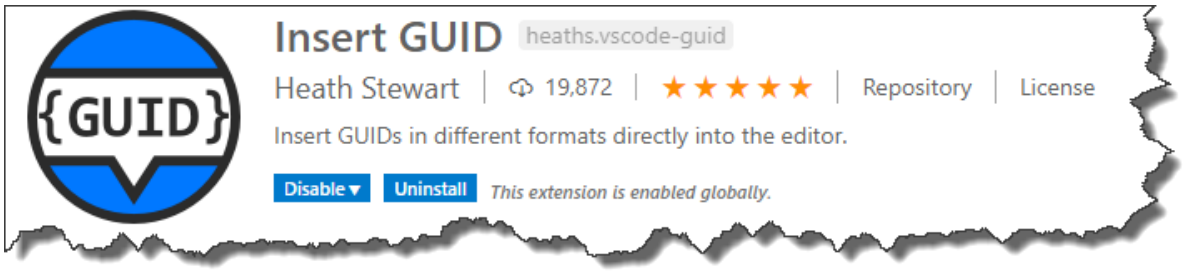


Upon clicking the desired extension you'll get a detail page explaining a bit about the extension, (along with the number of downloads and a review / rating). To install, simply click the "Install" button:

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!

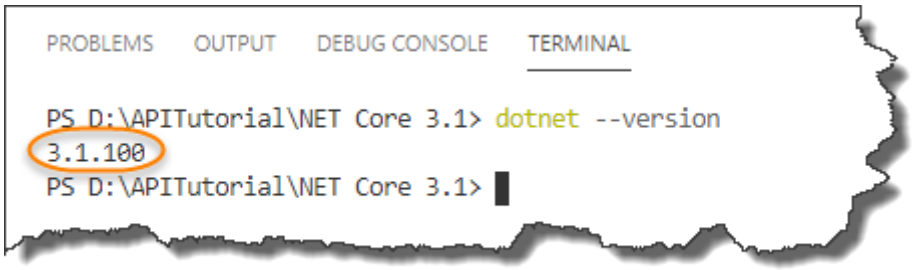
Ok we're done with VS Code set up for now so let's move onto the next install…

INSTALL .NET CORE SDK

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

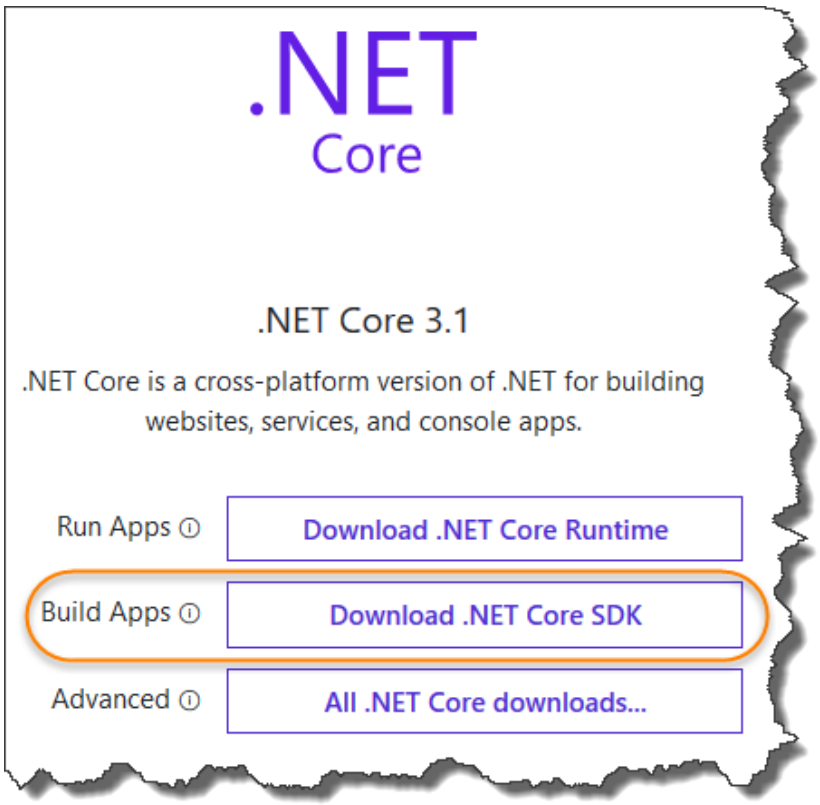
```
dotnet --version
```

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. From the screen shot above, you can see I'm running 3.1 which at the time of writing is the latest version.

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!

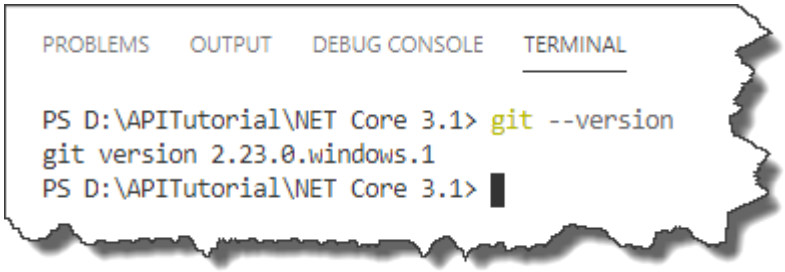
INSTALL GIT

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 (note I'm using the integrated terminal in VS Code):

```
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.

NAME AND EMAIL

Just to complete the setup of Git, we need to tell it who we are by way of a name and email address, as this information is required by Git in order for it to know *who* is making changes to the code.

To do so enter the following commands in a terminal session, replacing "you@example.com" and "Your Name" with suitable values:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

E.g.

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 DOCKER [OPTIONAL]

If you're intending to install PostgreSQL directly on your development machine, or you already have a version running somewhere that you can use, then you can skip this section if you like. However, if like me you don't like "faffing" around installing large apps on your local machine then Docker is a great option for you, (although paradoxically, Docker is quite a large application as of itself!)

What is Docker?

Docker is a containerization platform that enables you to:

- Package *your* apps as images and allow others to download and run them as containers, (on Docker).
- Obtain other developer or software vendor "images", (from a repository), and run them as containers on your machine, (so long as you've installed Docker).

The core concept of a Docker image is that they are self-contained, meaning that the image has everything it needs for it to run, avoiding complex installations, locating and installing 3rd party support libraries etc. It ultimately avoids the: "it works on my machine" argument.

There is a little bit of a learning curve to it, (not much though!), and once you master the basics, it can save you so much time and effort, that as a developer, I can't recommend it highly enough.

Docker Desktop Vs Docker CE

Confusingly, (for me at least), if you're running Windows or OSX you need to install something called *Docker Desktop*. If however you're a Linux person, then you should install *Docker Community Edition* or CE. There are probably torturously pedantic reasons for this, which I not aware of, nor would I be interested in learning about, so all you really need to know is where to get them!

- Docker Desktop Here
- Docker CE Here

Before you can download and install Docker Desktop, you need to sign up for a Docker Hub account, this is a free sign-up so nothing really to worry about. It also comes in useful if you want to upload your own images to the Docker Hub for distribution.

Warning! If you're wanting to install Docker Desktop on Windows, at the time or writing you will need Windows 10 Professional, (the Home edition does not support it).

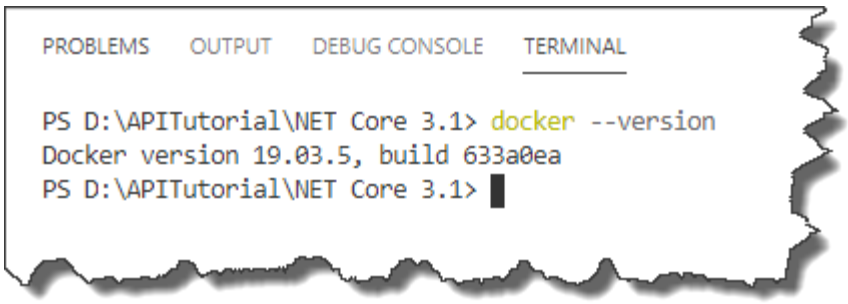
Docker Desktop installation is super-simple, for Docker CE you will need to refer to the install instructions for your specific distro – again however it's straightforward.

Post Installation Check

Irrespective of which flavour of Docker you install, post installation open a command line and type:

```
docker --version
```

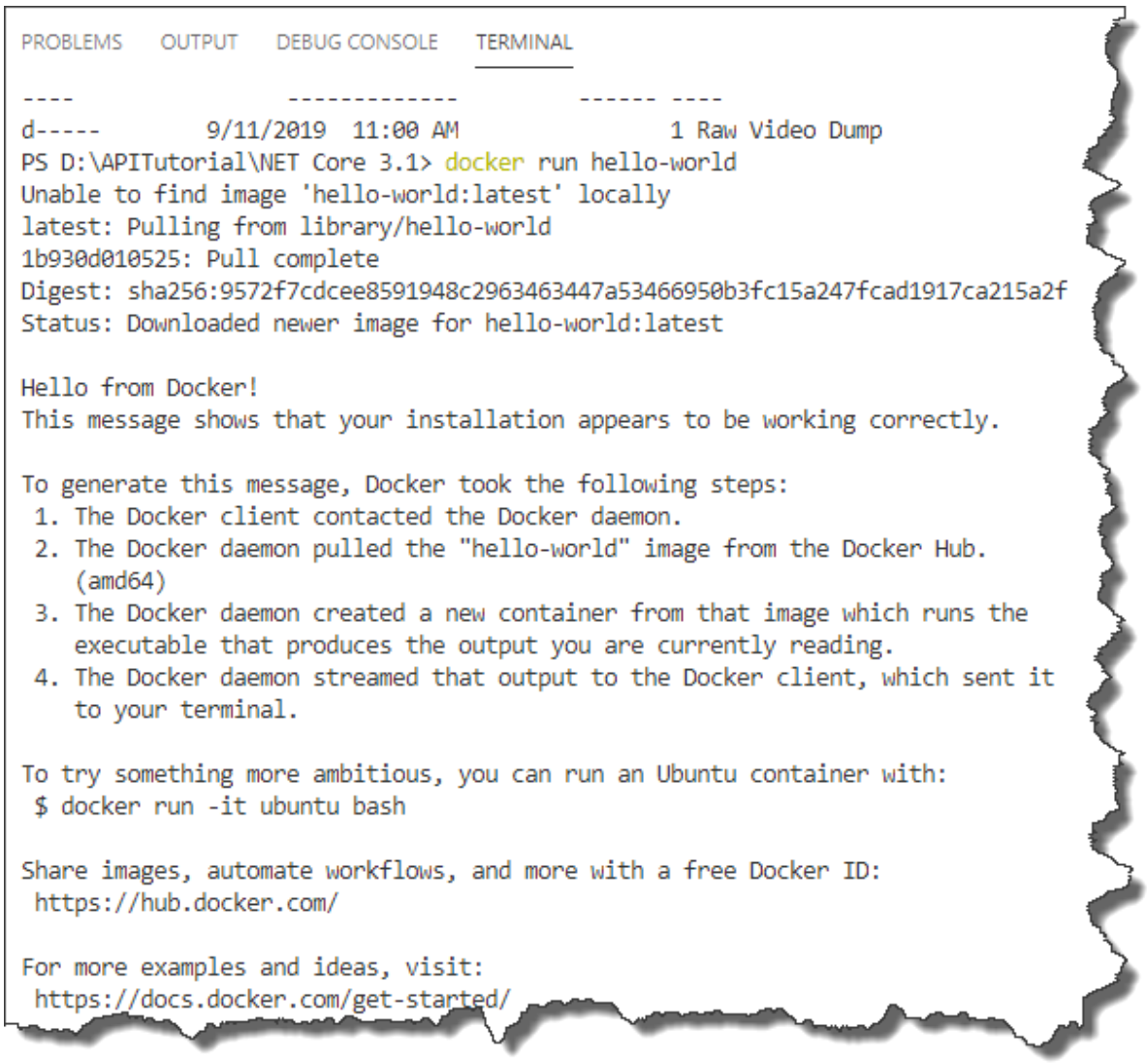
You should get something like the following:



To further test that it's fully working, type:

```
docker run hello-world
```

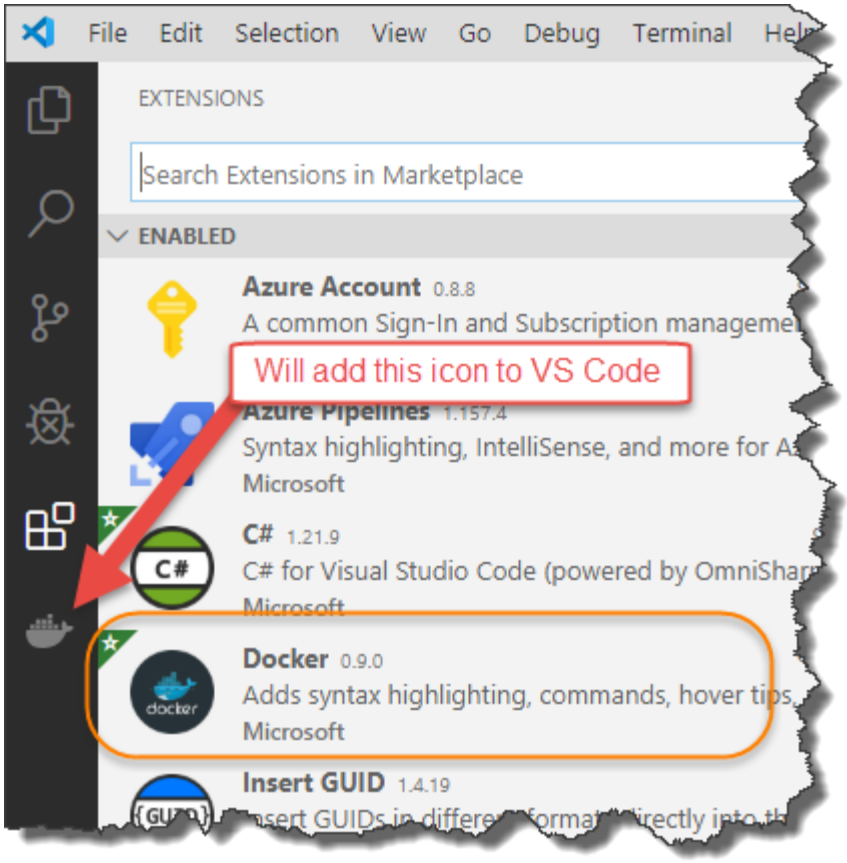
If this is the first time you've run this, Docker will go to the Docker Hub, pull down the `hello-world` image, and run it, you should see something like this:



We don't need to go into too much more detail about what's happening here, (although the output generated by `hello-world` does a pretty good job), suffice to say that Docker is set up and ready to go. I'll cover more on Docker as we move through the tutorial.

Docker Plugin For VS Code

If you're using VS Code as your development editor and you've decided to go with Docker, then I highly recommend you install the Docker extension from Microsoft. I've shown this below, but will leave it to you to install…



Install PostgreSQL

If you don't want to use Docker and want to install PostgreSQL directly on your development machine, (or on another server, virtual machine etc.), then you'll need to follow the install steps for your OS. As mentioned previously, I won't be detailing those steps in detail here as the PostgreSQL guys have done a great job of that already here: https://www.postgresql.org/download/

Warning! I've spent many hours getting PostgreSQL up and running on a Linux box and connecting in from another machine. Now this is due largely to the fact I'm not particularly great with Linux, and so those of you that are good with Linux would undoubtedly have less trouble.

None the less, struggling with the nuances of installing a DB, for me detract from the act of coding which is what I *really* want to be doing. Hence the reason why I *strongly* suggest the use of Docker.

Windows and OSX installations are, (as usual), much easier.

Install DBeaver CE

Whether you're going to use Docker or a native PostgreSQL install, either way we'll want to do some small bits of DB admin as well as write SQL queries to read and write data into our DB. You can of course use the command line options that come with the PostgreSQL install, but I like to also have a graphical environment at my disposal as the "barrier to entry" is significantly reduced when compared to the command line alternative.

Just remember: my focus in this book is *coding an API,* not being an expert PostgreSQL DB administrator.

DBeaver Vs pgAdmin

Probably the most popular admin tool for PostgreSQL is pgAdmin, and in fact this would have been the tool I'd have recommended previously…

Les's Personal Anecdote: The choice of admin tool here is a totally personal one… I have used pgAdmin in its prior iterations and it was totally fine, but since they moved it to a "web version", running in its own little webserver, I've avoided it. Can't quite put my finger on why, I think mostly it just comes across as a bloated and counter intuitive piece of software.. It's a web app that requires the install of a local webserver? Doesn't "smell" right to me…

Having looked at several graphical database management tools for PosrgrSQL, I've landed on DBeaver Community Edition - which is free. This is a database agnostic management tool that you can use to connect to and manage most of the popular RDBM[6]'s out there. It's also cross platform, which is even better – you can download your copy here: https://dbeaver.io/download/

We'll go through connecting to and setting up PostgreSQL later in the book. For now though we're done….

Les's Personal Anecdote: Just before we move on, I just wanted to say that for me the king of Database Management Tools is still SQL Server Management Studio. In my *personal view* nothing comes close to it in terms of useability, speed, features etc.

The only reason I've moved away from it in this version of the book is simply because I've decide to switch database platforms, (you can only use SQL Server Management Studio to manage MS SQL Servers – it also only runs on Windows).

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.

Trust Local Host Development Certs

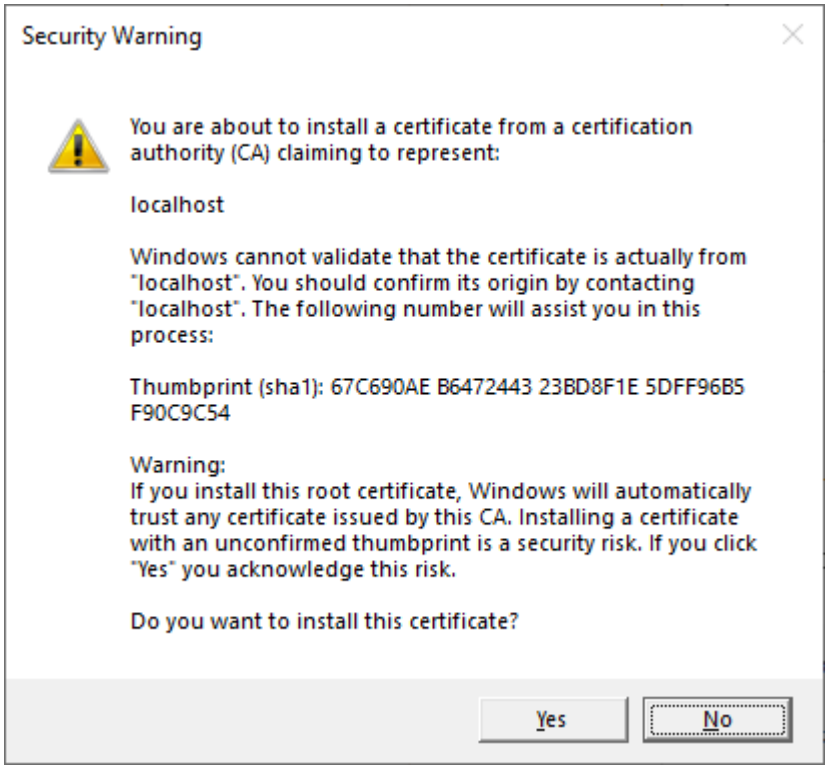
Throughout the tutorial we'll be hitting localhost end points over http and https. For those connections using https, we may encounter some errors / exceptions along the lines that that the certificate is not valid. We do not want to turn off SSL certificate validation, instead we want to trust our local development certificate.

To do that, at a command prompt type:

```
dotnet dev-certs https –trust
```

You'll get a message box similar to the following:

[6] Relational Database Management Systems



Click “Yes” to install the certificate and you should be good to go.

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[7]. 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.

[7] 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, 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…

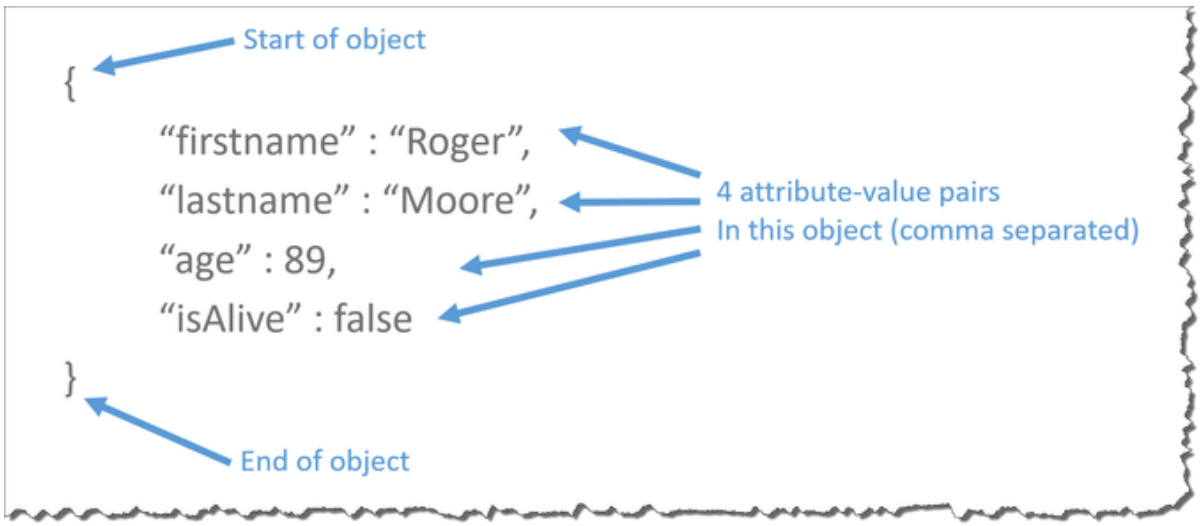
Upon receiving that serialised string payload, you’ll then do something with it, most likely some kind of deserialization operation so you can use the resource or object within the consuming application. With regard REST API’s there is no prescribed payload format, although most usually JSON will be 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



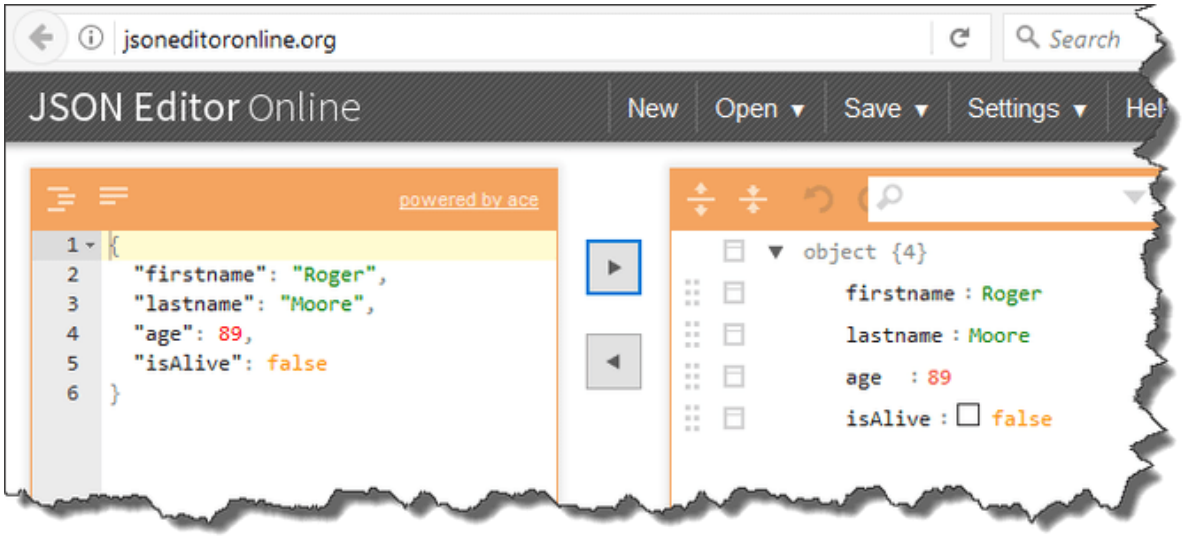
In the above example we have a “Person” object with 4 attributes:

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

With the following respective values:

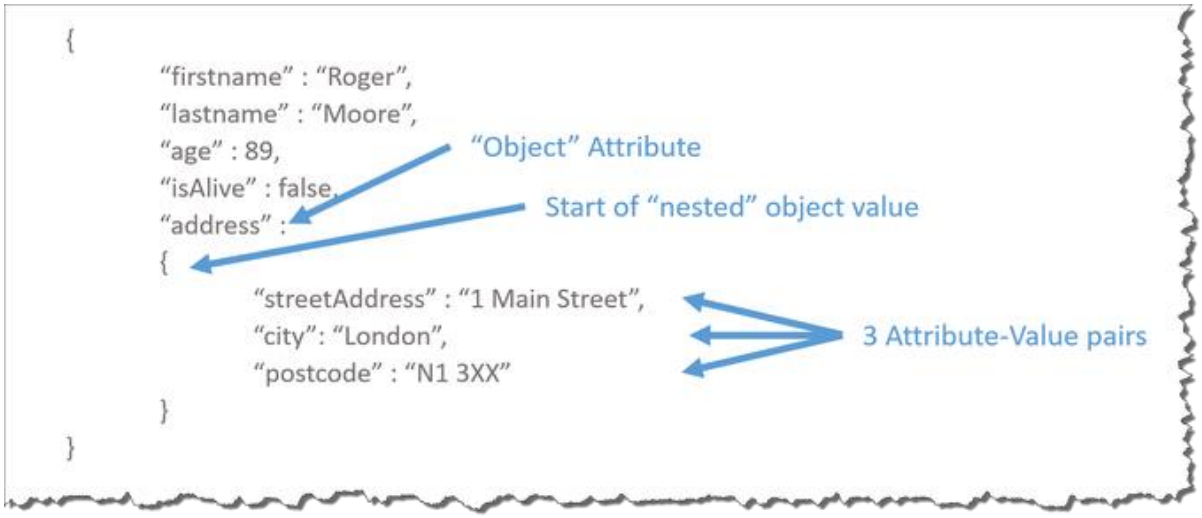
- `Roger` [This is a string data-type and is therefore delineated by double quotes ‘ ” ‘]
- `Moore` [Again this is a string and needs double quotes
- `89` [Number value that does not need quotes]
- `false` [Boolean value, again does not need the double quotes]

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:

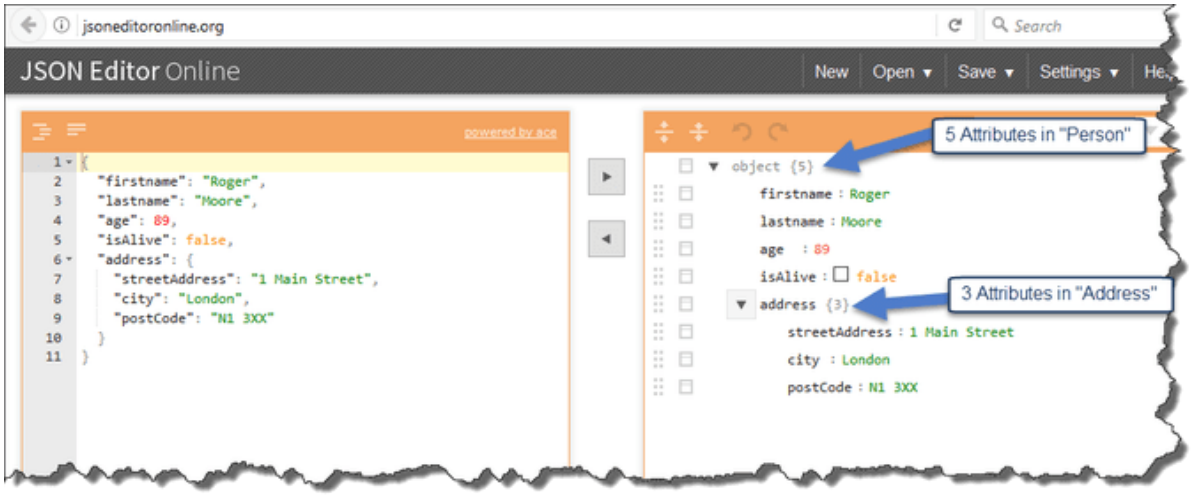


Here we can see that we have a 5th Person object attribute, `address`, which does not have a standard value like the others, but in fact contains another object with 3 attributes:

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

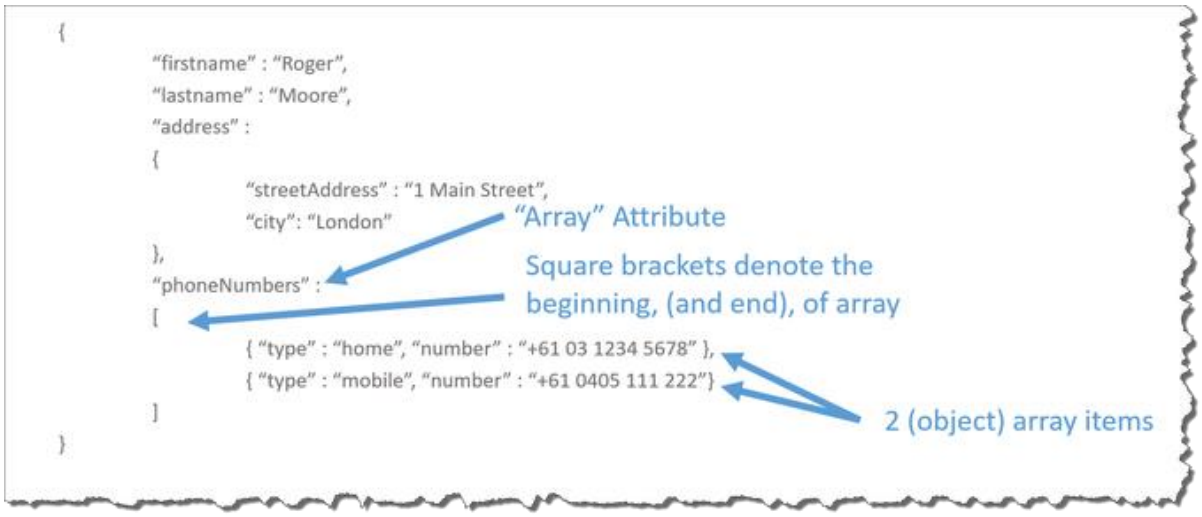
The values of all these attributes contains strings, so no need to labour that point further! This nesting can continue ad-nauseum…

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:



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

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.

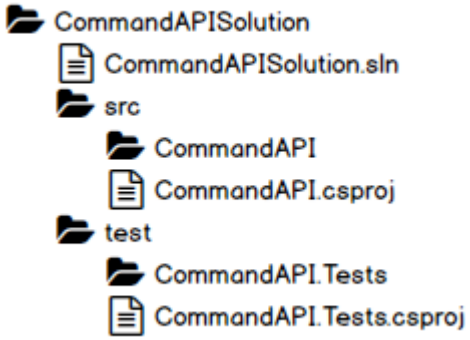
Projects can of course exist without a parent Solution, going further, Projects can reference one and other without the need for a Solution. So why bother with a solution? Great question, it boils down to:

- 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:



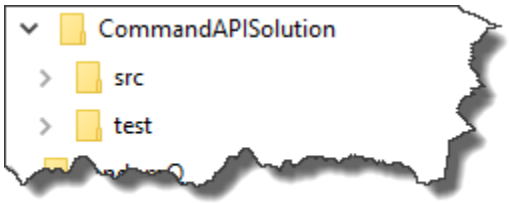
You'll see that we have sub folders within the main solution folder to segregate source code, (***src***), and unit test projects, (***test***). Ok so let's start creating our solution and projects!

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:



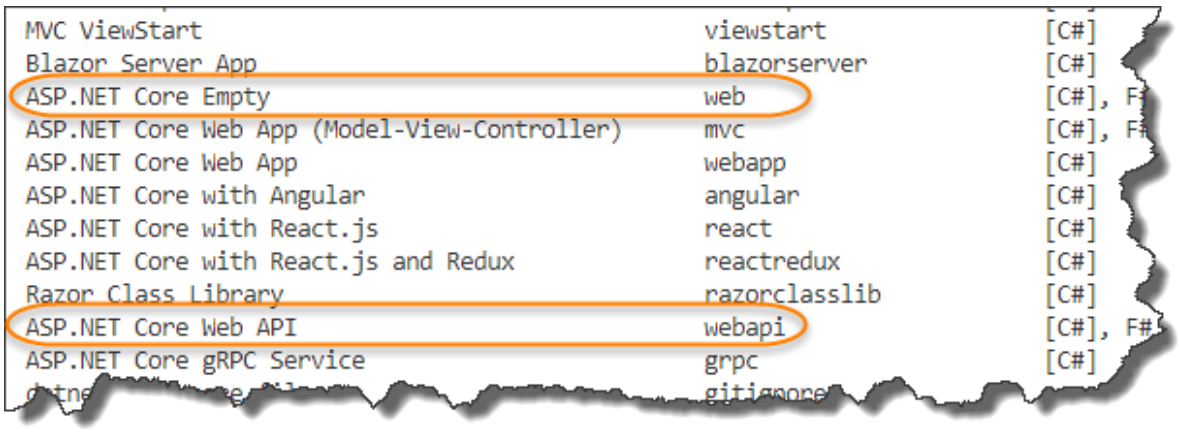
- Open a terminal window, (if you haven't already), and navigate to "inside" the ***src*** folder you just created.

.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:



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, (again ensure you are "inside" the ***src*** directory):

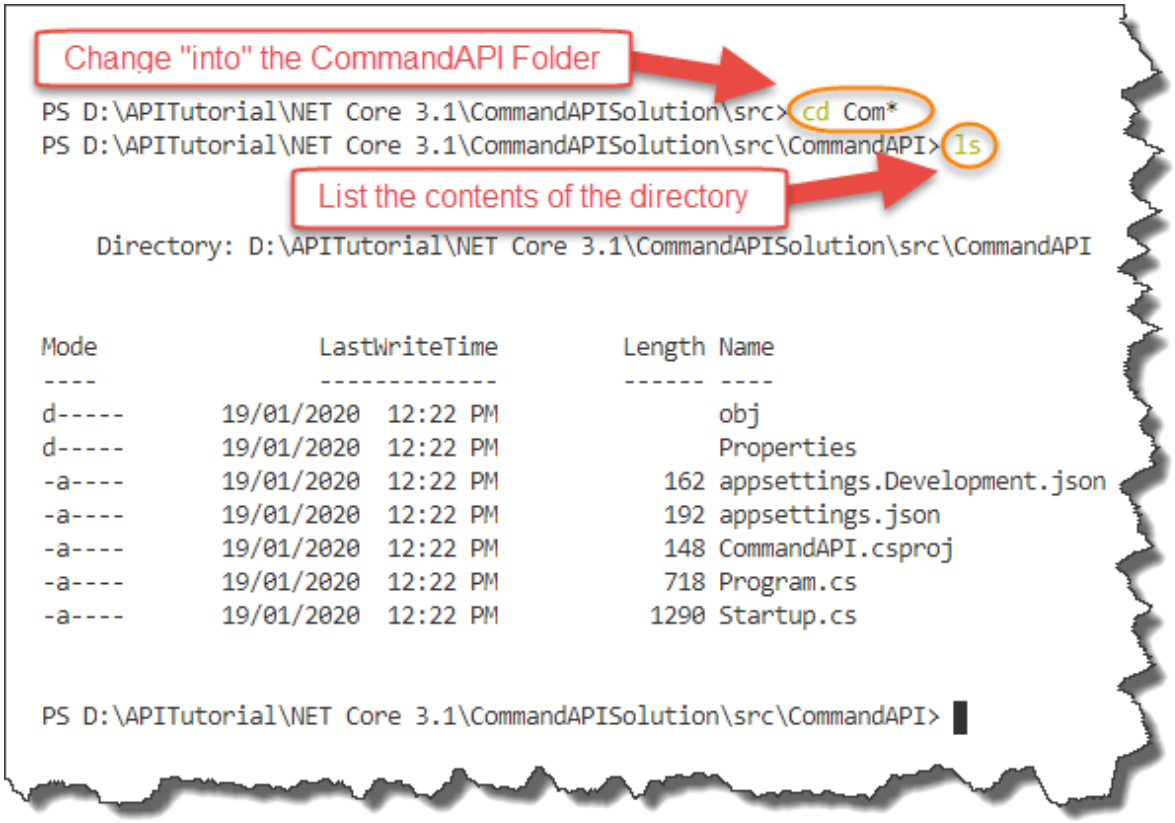
```
dotnet new web -n CommandAPI
```

Where:

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

You should see something like:

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:



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[8] contained in the main solution directory ***CommandAPISolution***
- At the command line, type:

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

You should see the following output:

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

```
PS D:\APITutorial\NET Core 3.1\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...
  Restore completed in 1.37 sec for D:\APITutorial\NET Core 3.1\CommandAPISolution\test\Comman

Restore succeeded.

PS D:\APITutorial\NET Core 3.1\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: ***CommandAPISolution***, to check you're in the right place perform a directory listing, and you should see something like this:

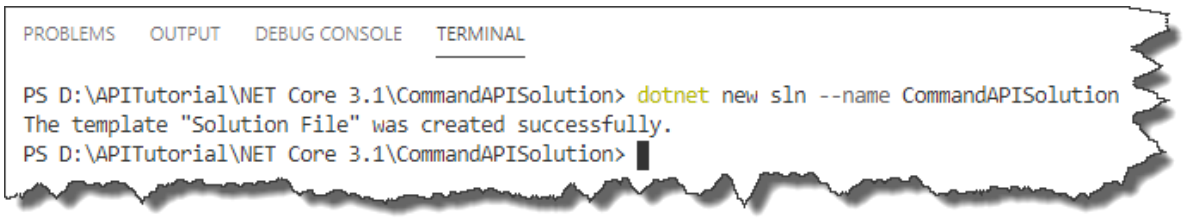
```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS D:\APITutorial\NET Core 3.1\CommandAPISolution\test> cd ..
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> ls


    Directory: D:\APITutorial\NET Core 3.1\CommandAPISolution

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        19/01/2020  12:22 PM                src
d-----        19/01/2020  12:28 PM                test


PS D:\APITutorial\NET Core 3.1\CommandAPISolution>
```

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:

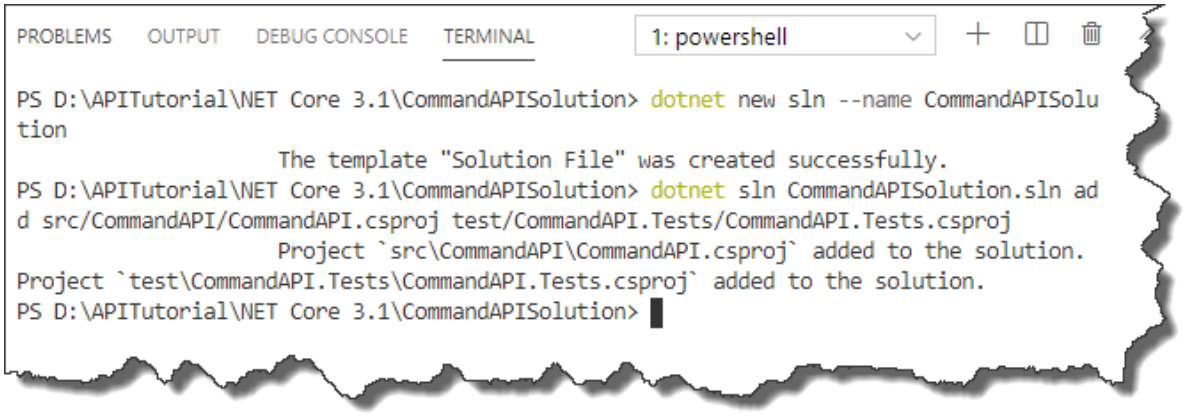


We now want to associate both our "child" projects to our solution, to do so, issue the following command:

```
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:



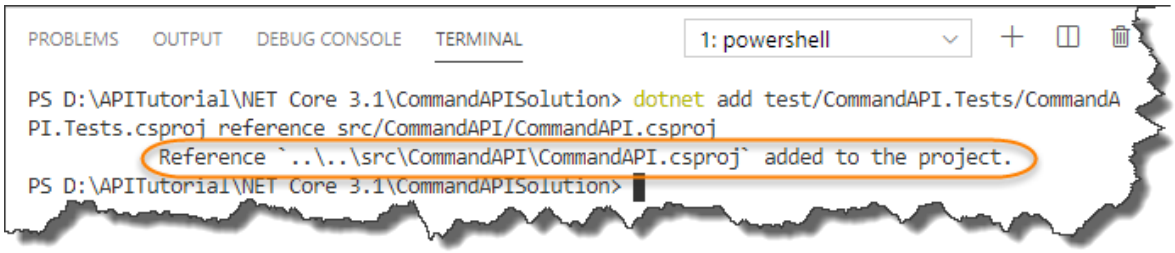
If you get an error, double check that you have typed the full path correctly. It's quite long so the opportunity to make a mistake is there. Believe me – I have spent many a time rectifying typos of this sort…

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…

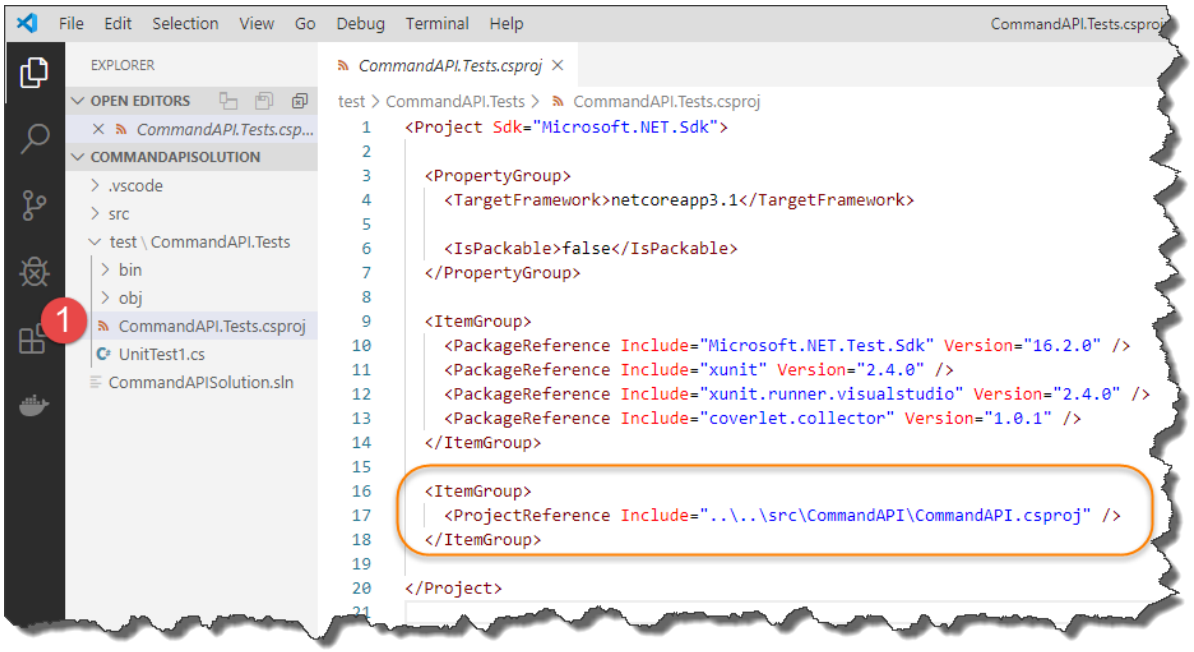
We need to place a "reference" to our ***CommandAPI*** project ***in*** our ***CommandAPI.Tests*** project, this will enable us to reference the ***CommandAPI*** project and "test" it from our ***CommandAPI.Tests*** project. You can either manually edit the ***CommandAPI.Tests.csproj*** file, or type 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[9], 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?

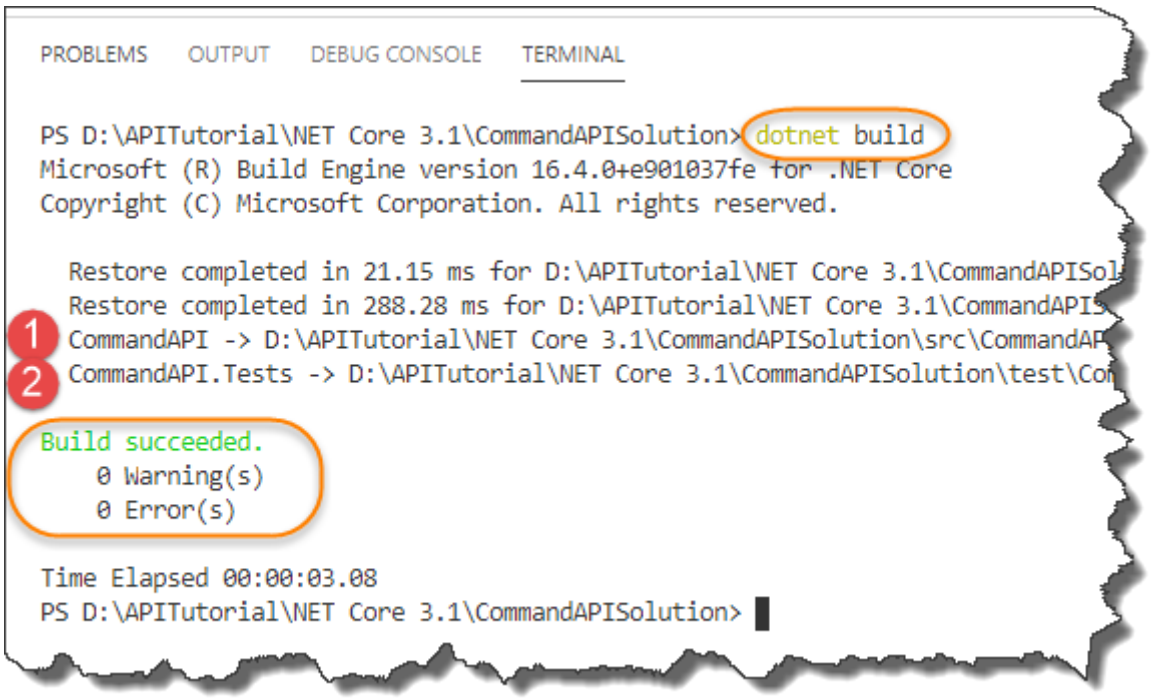
You can now build both projects, (ensure you are still in the root solution folder), by issuing:

```
dotnet build
```

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

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

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



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*[10] 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.

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

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. Also, environment specific settings can be contained in additional settings files, (e.g. Development), as shown by the ***appsettings.Development.json*** file.
<ProjectName>.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)
- ***<ProjectName>.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.

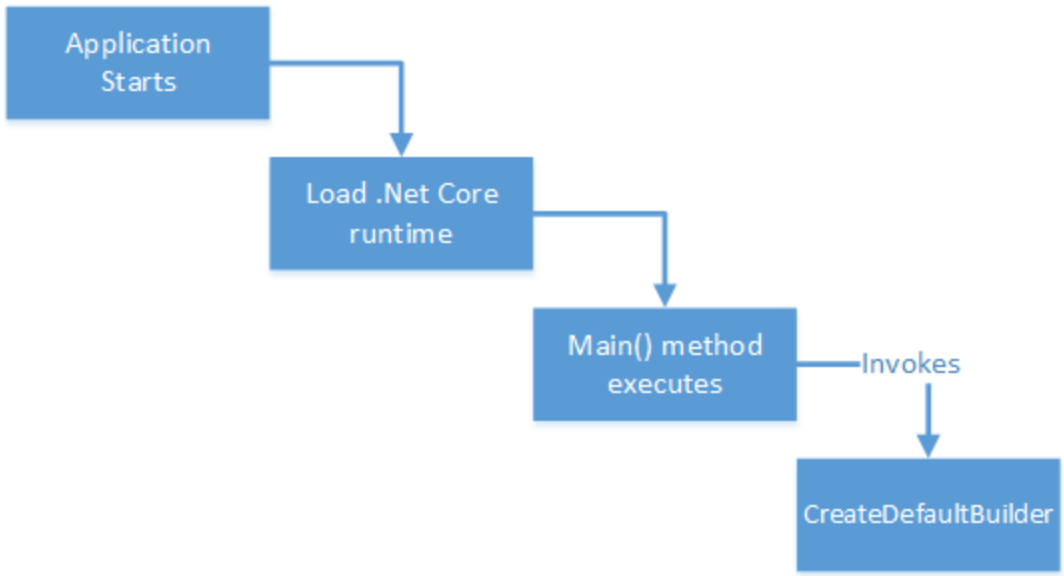
Let's take a quick look at the templated code, (that we're *not actually* going to change!), and see what it does…

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)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.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...

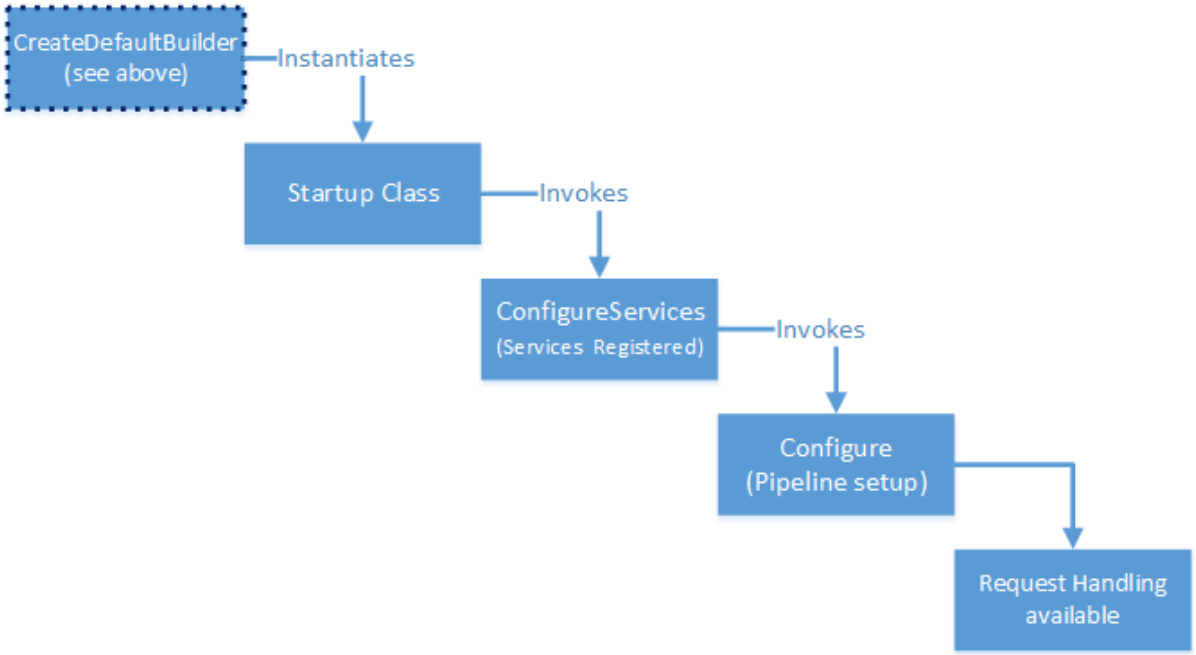
Note: we do cover .NET Core Configuration in more detail later in the book.

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`

The execution sequence is as follows:



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*[11], this is where dependencies are registered inside the default Inversion of Control, (IoC), container provided by .NET Core.

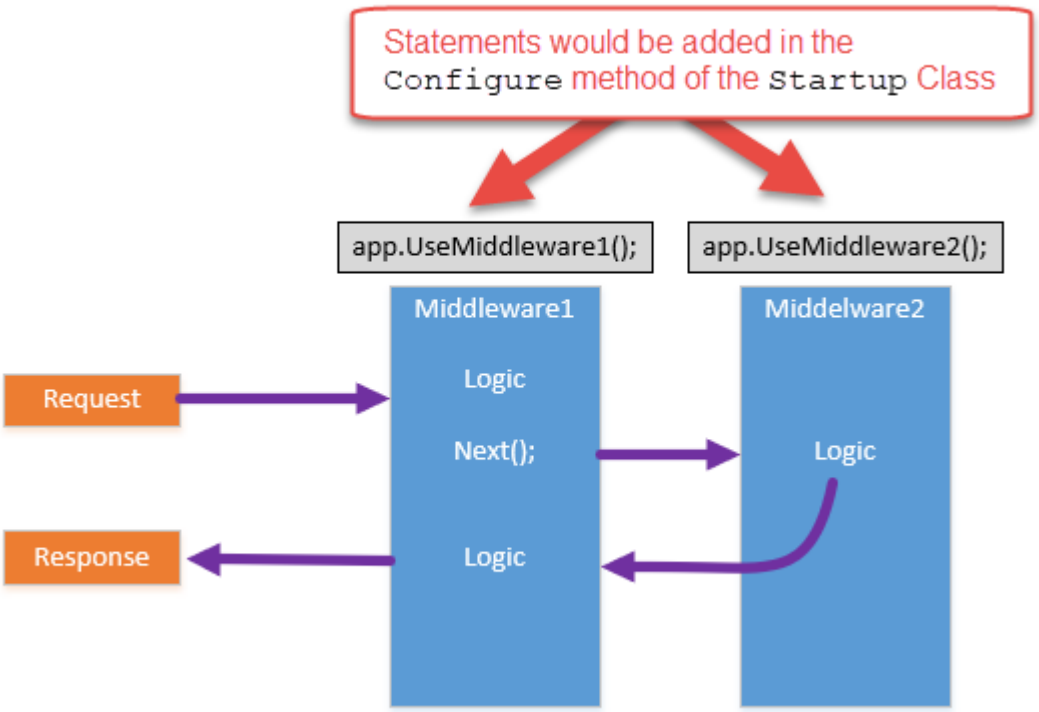
CONFIGURE

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.

In the diagram below, you can see how a request would traverse the middleware components added in the `Configure` method. The nature of the request, (e.g. is it an attempt to open a Web Socket?), and the logic in the middleware will determine what will happen to that request, with the ability to "short-circuit" traversing further middleware if required, (not shown).

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



The *Request Pipeline* and middleware in general, is an expansive area which could occupy a whole chapter of the book on its own. In keeping with the “thin and wide” approach I feel we’ve covered enough to move on and start coding!

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.

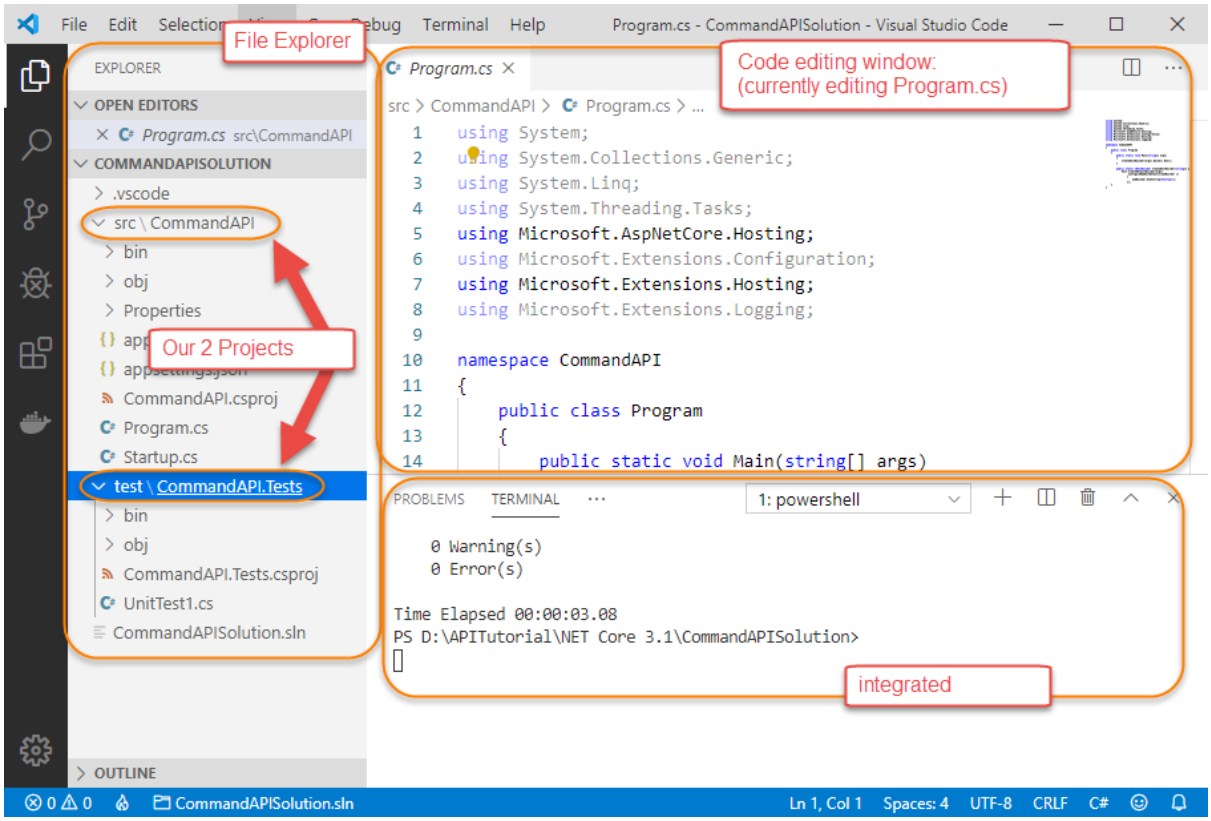
When Done, You Will

- Understand what the MVC pattern is
- Understand our 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.

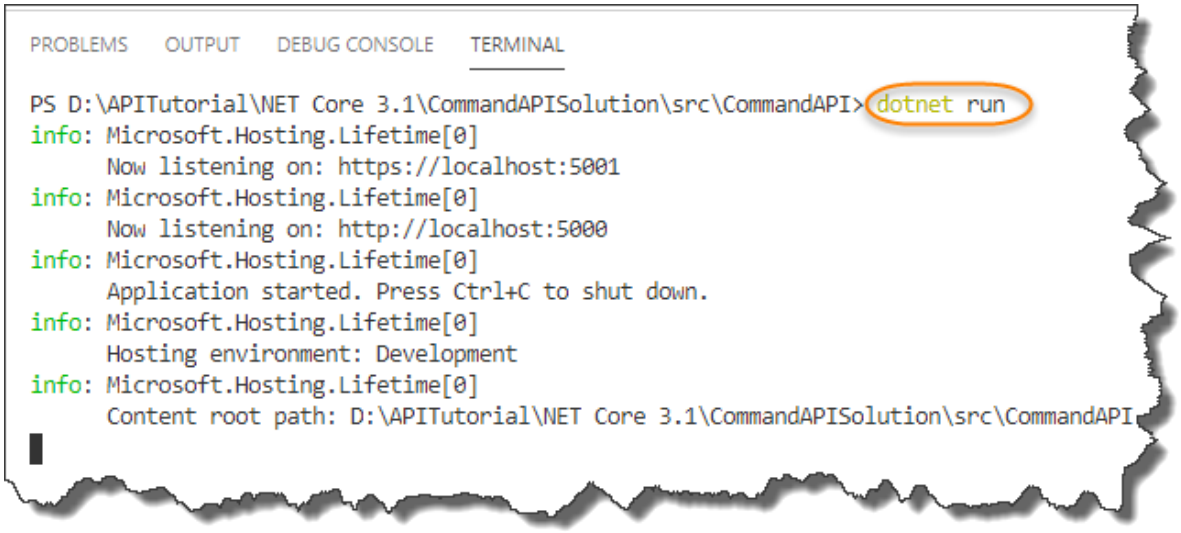
1. In VS Code hit 'F1' (this opens the "command palette" in VS Code)
2. Type *shell* at the resulting prompt, and select "*Terminal: Select Default Shell*"
3. You can then select from the Terminals that you have installed

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:



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:

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

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

For those of you that have worked with any of the 2.x versions of the .NET Core Framework, (for those of you ***that haven't*** you can ignore this), this will look slightly different to what you may have seen before. As opposed to:

- `app.USeEndPoints…`

You would have seen:

- `app.Run(async ….`

The previous version of the framework would also make use of:

- `services.AddMvc()` – in our `ConfigurerServices` method and:
- `app.UseMVC()` – in our `Configure` method

Further discussion on the differences between versions 2.x and 3.x of the .NET Core Framework can be found here: https://docs.microsoft.com/en-us/aspnet/core/migration/22-to-30?view=aspnetcore-3.0&tabs=visual-studio-code

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 Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace CommandAPI
{
    public class Startup
    {

        public void ConfigureServices(IServiceCollection services)
        {
            //SECTION 1
            services.AddControllers();
        }


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

            app.UseRouting();

            app.UseEndpoints(endpoints =>
            {
                //SECTION 2
                endpoints.MapControllers();
            });
        }
    }
}
```

What does this code do?

1. Adds services for "Controllers" to be used throughout our application. As mentioned in the info box above, in previous versions of .NET Core Framework, you would have specified: `services.AddMVC`. Don't worry we cover what the **M**odel **V**iew **C**ontroller, (MVC), pattern is below.
2. We "MapControllers" to our EndPoints. This means we make use of the Controller services, (registered in the `ConfigureServices` method), as endpoints in the *Request Pipeline*.

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

https://github.com/binarythistle/Complete-ASP.NET-Core-API-Tutorial-2nd-Edition-Net-Core-3.1

As before run the project, (ensure you save the file before doing this[12]):

[12] 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…

```
dotnet run
```

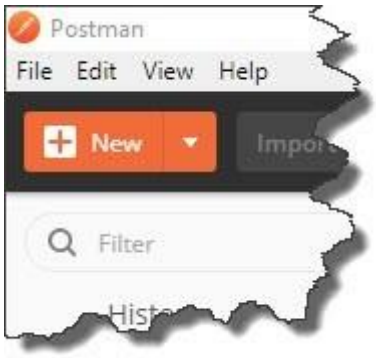
Now navigate to the same url in a web browser, (http://localhost:5000), and we should get “nothing”.

CALL THE POSTMAN

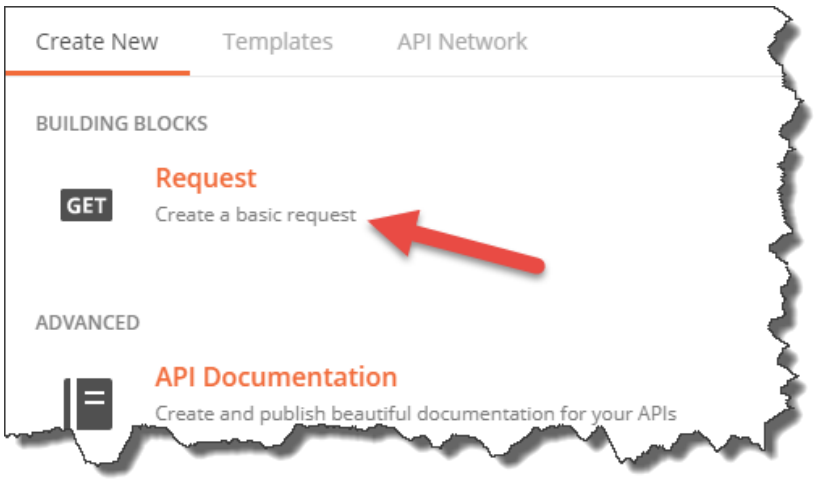
Now is probably a good time to get Postman up and running as it’s a useful tool that allows you to have a more detailed look at what’s going on.

So if you’ve not done so already, go to the Postman web site, (https://www.getpostman.com), 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).

We want to make a request to our API using Postman, so click “New”



The select “request”



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

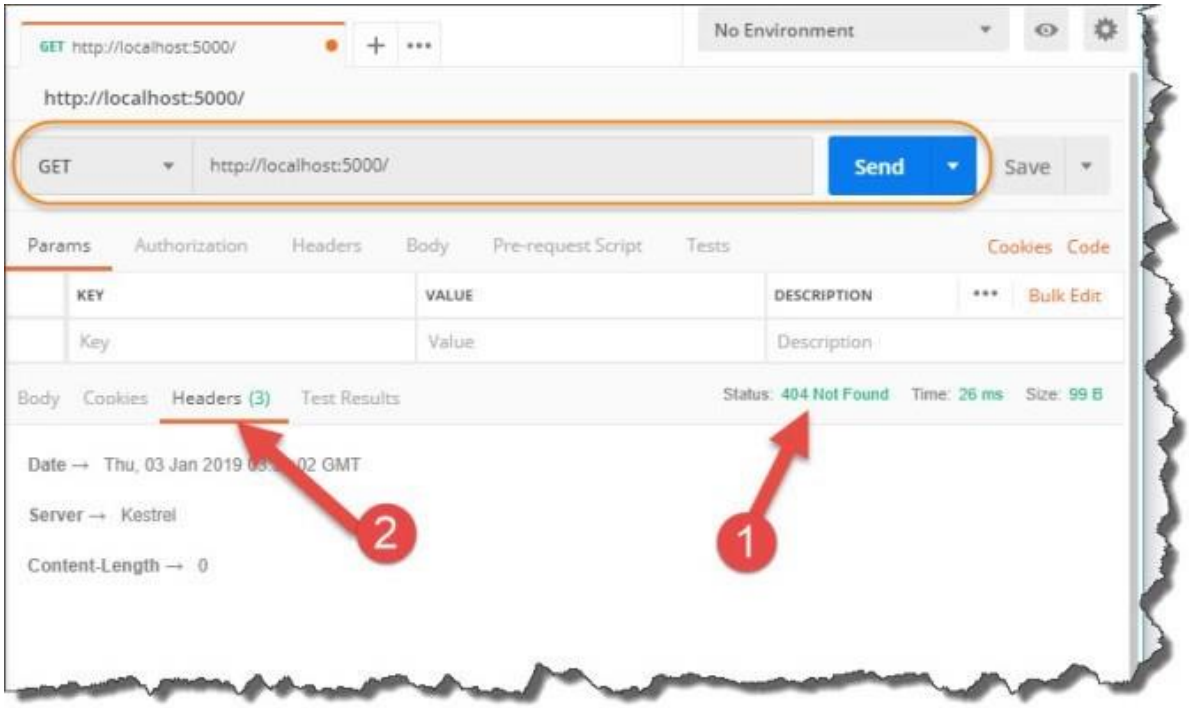
You should then have a new tab available to populate with the details of your request. Simply type:

`http://localhost:5000`

Or

`https://localhost:5001`

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

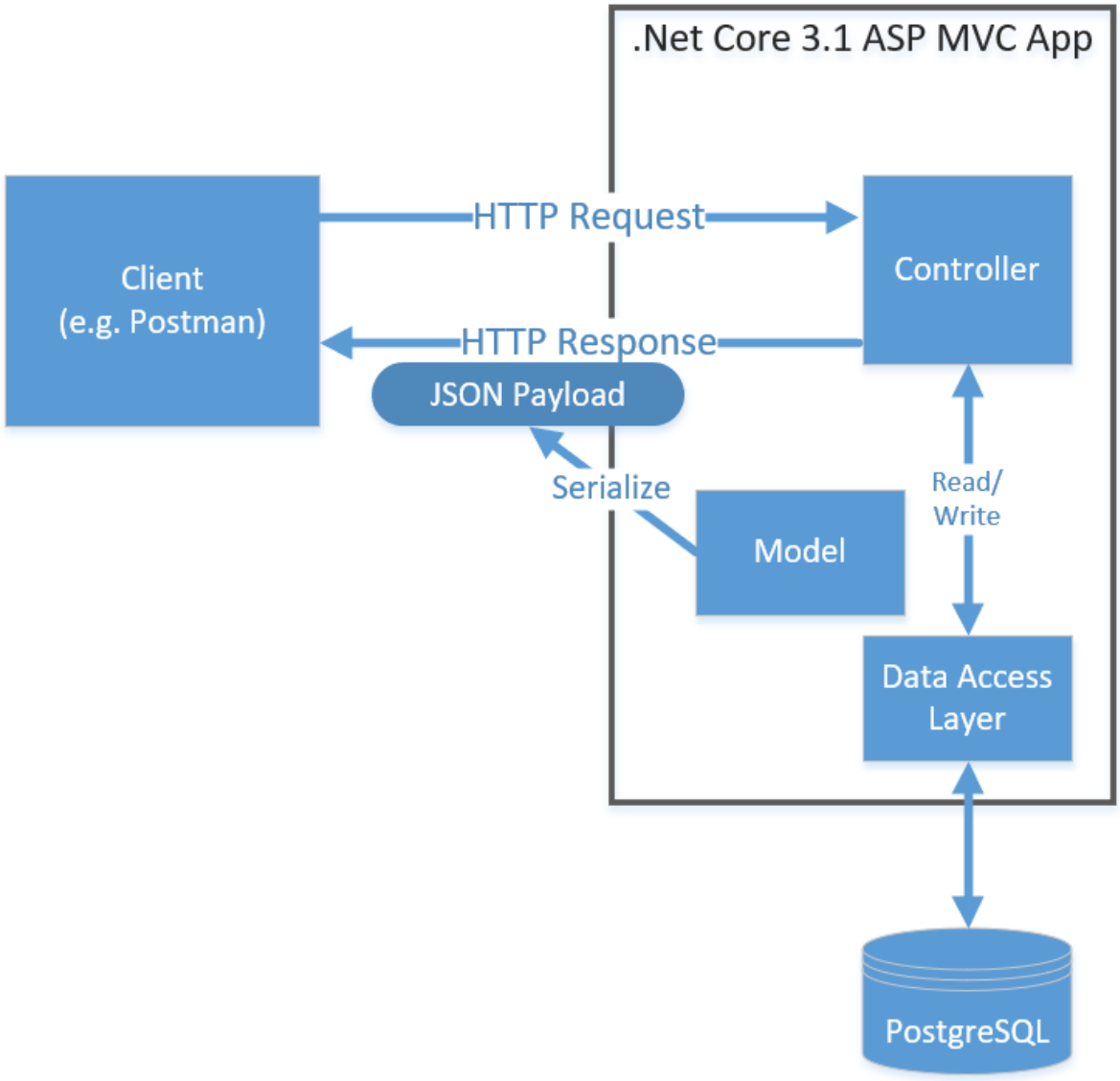
I won't go into *too* much detail.

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 for our API is shown below:



It's also worth noting, in case it wasn't clear, that the MVC pattern is just that – an application architecture *pattern* - it is agnostic from technical implementation. As this happens to be a book about a particular technology, (.NET Core), we cover how .NET Core implements MVC, however there are other implementations of the MVC pattern using different frameworks and languages.

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. PostgreSQL).

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).

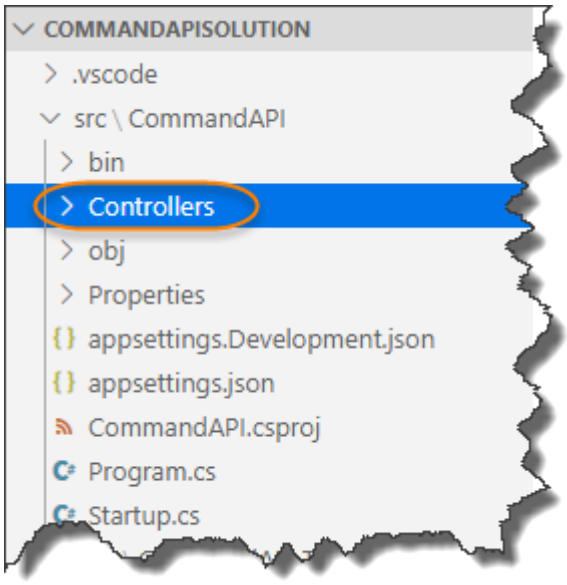
* You can have a data access layer that uses an "in memory db" or non-persistent data store so theoretically you can have a data access layer that still doesn't persist data… For the most part though I think of a data access layer as being the mediator between the application and the, (persistent), data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

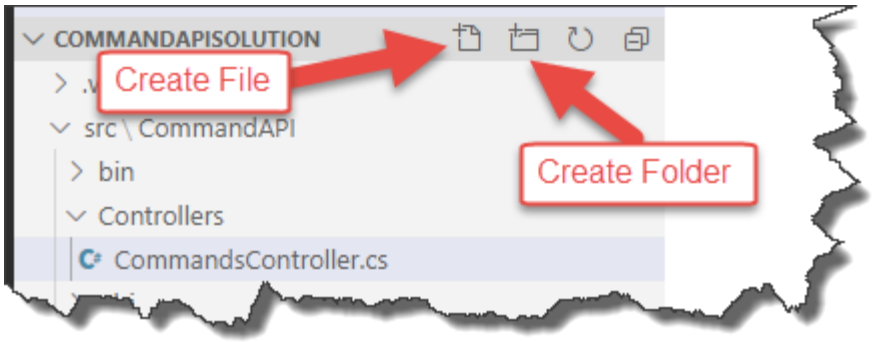
Our Controller

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

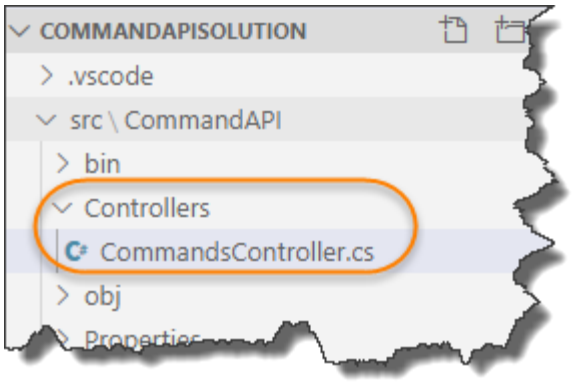


Inside the ***Controllers*** folder you just created, create a file called ***CommandsController.cs***

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"};
        }
    }
}
```

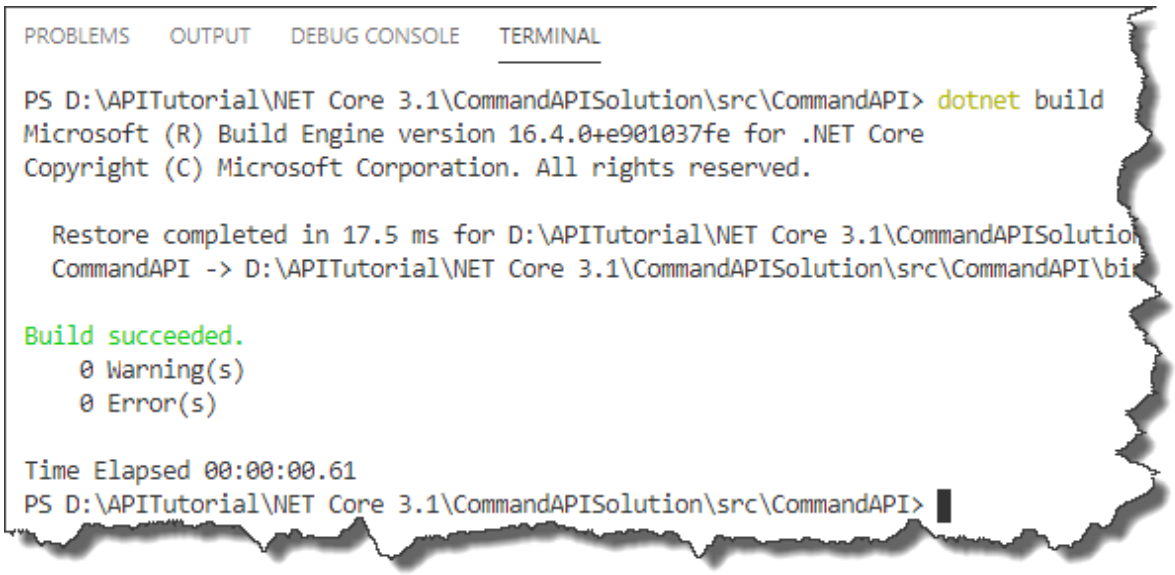
Again, if you don't fancy typing this in, the code is [available on GitHub.]

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:



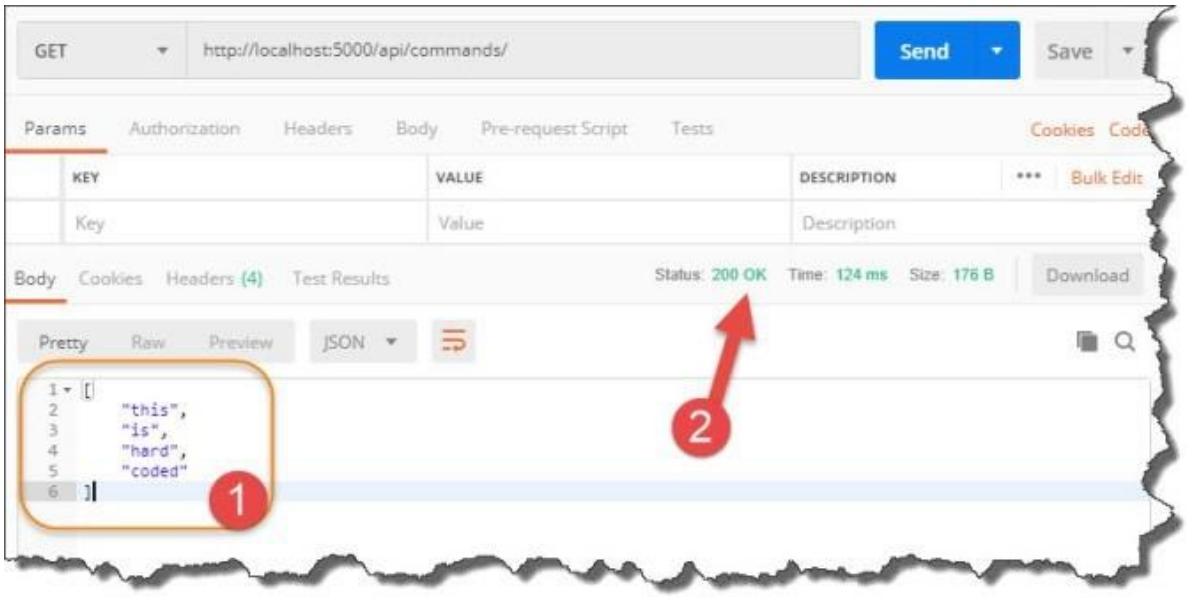
Now ***run*** the app.

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
```

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

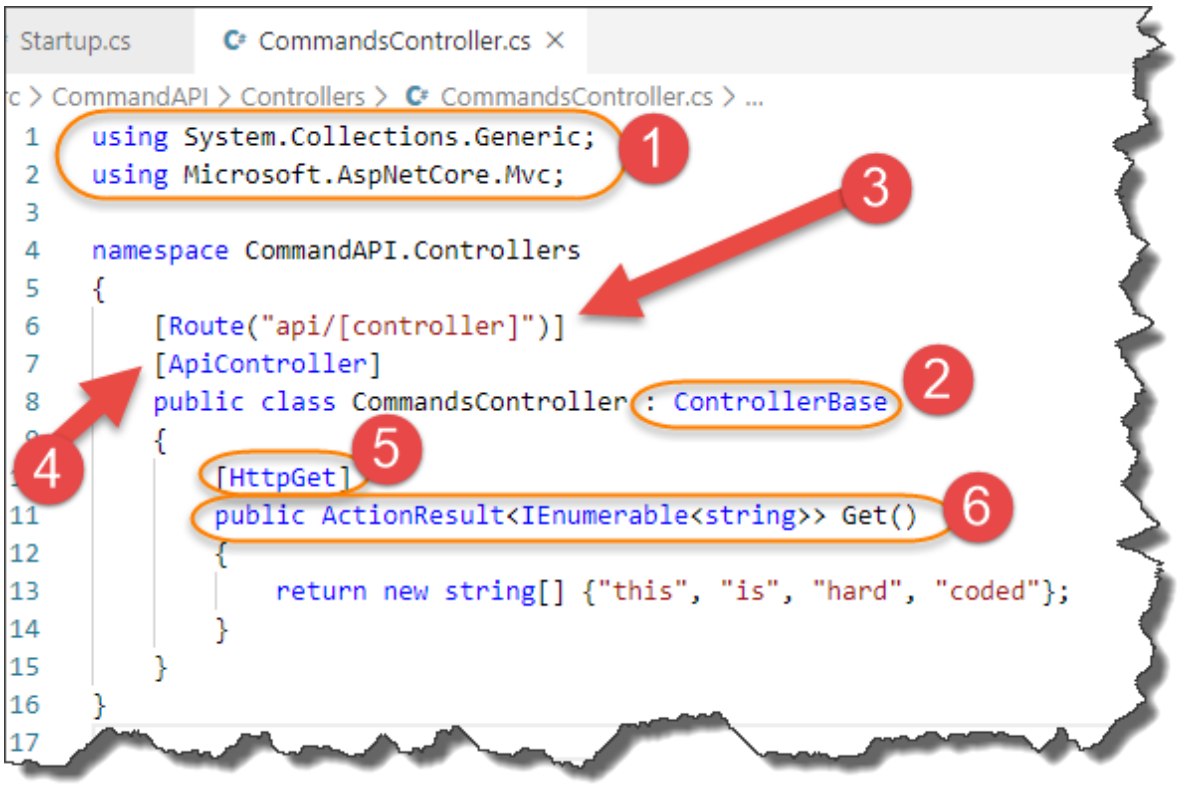


1. This is the hard-coded json string returned
2. We have a 200 OK HTTP response, (basically everything is good)

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

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 the above example, our controller action is *synchronous*, meaning that it is called in a "serial" fashion: i.e. it gets called and we *wait* for the result to be returned – this is fine for the simple hard coded scenario we have as we don't expect it to take a long time to execute

However, if you were expecting your controller actions to take a long time to respond, (when we come to introduce potentially long-running data retrieval queries of from a database), then you may want to consider making your action methods *asynchronous.*

In simple terms this just means that we kick off a call to the action method, but we *don't wait around* for it to return, instead we pass code execution back to the caller of the async method, and let it continue with anything else it can be doing *while we wait*….

When our `ActionResult` finally does return, (a result), this is then supplied for use to the code that called it in the first place.

Microsoft introduced a simplified asynchronous programming model with the release of C# 5 in 2012, via the use of the `async` and `await` key words.

In short:

- Asynchronous methods use the `async` modifier.
- Asynchronous methods typically return a `Task` (with optional type)
- `Tasks` represent ongoing work, and will eventually return with the required result, (or exception)
- Asynchronous method names will usually have "Async" as the last segment of the name
- `await` statements can be used in Async methods to designate "suspension points", i.e. we can't continue past this point until the awaited process is complete.
- The "awaited" method it's self must also be asynchronous
- While "awaiting" control returns to the caller of the Async method

If you'd like more information on the "Task Asynchronous Programming Model" then I recommend you reads the following article my Microsoft: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/task-asynchronous-programming-model

We'll return to this asynchronous model later in out tutorial when we introduce our database backend…

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"};
```

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 than 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 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[13] 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 placing "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

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

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, (if your API app is still running you may want to stop it by hitting Ctrl + c):

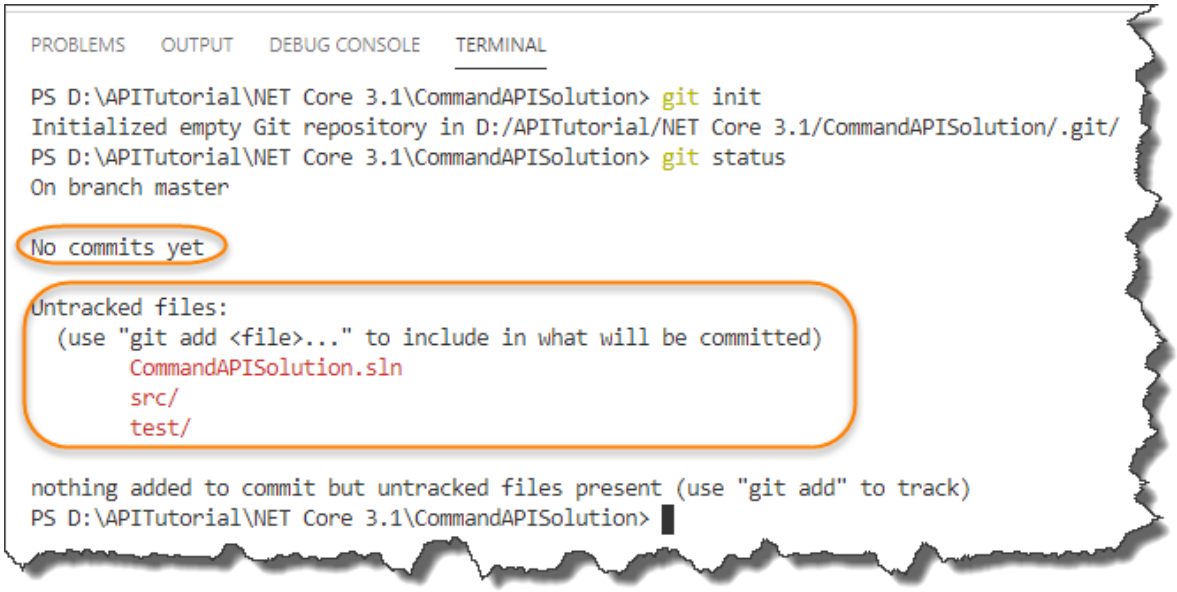
```
git init
```

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:

```
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

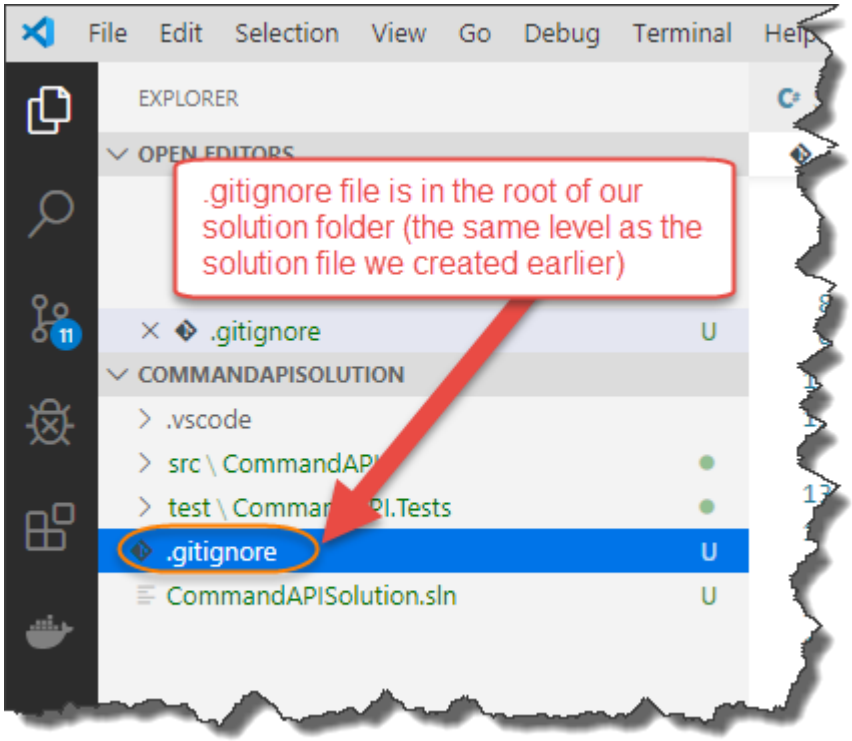
Before we start to track our solution files, (and bring them under source control), there are certain files that you shouldn't bring under source control, in particular files that are "generated" as the result of a build, primarily as they are surplus to requirements… (and they're not "source" files'!)

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, (ensure you put the trailing period '.'):

```
git add .
```

Followed by:

```
git status
```

You should see:

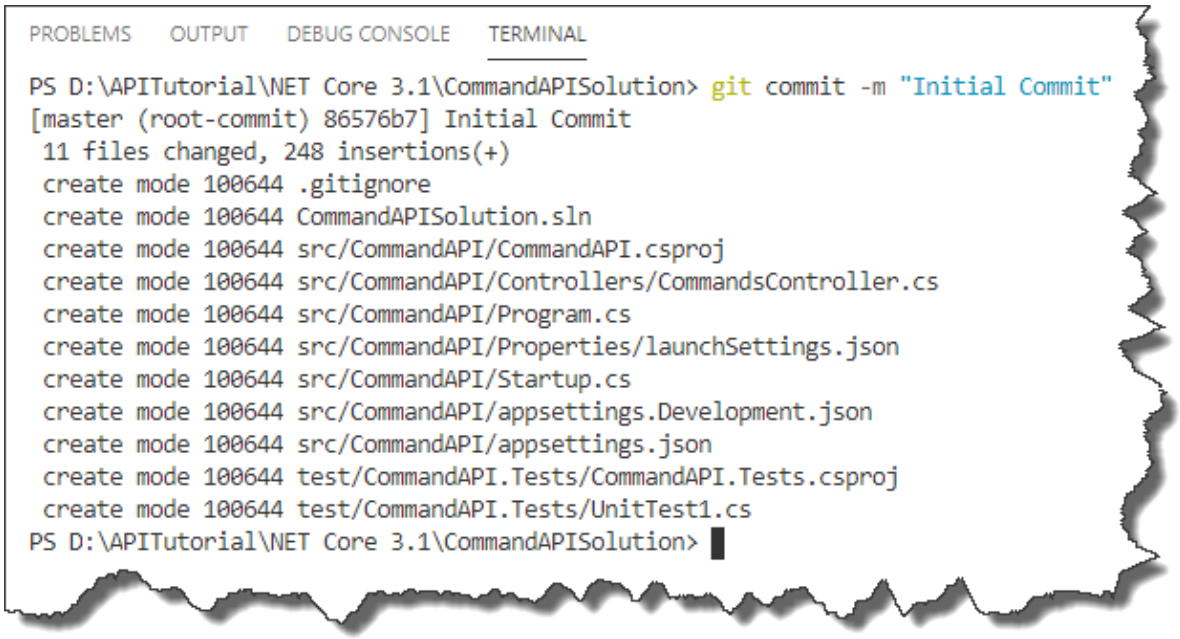
```
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> git add .
warning: LF will be replaced by CRLF in src/CommandAPI/appsettings.Development.json.
The file will have its original line endings in your working directory
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   CommandAPISolution.sln
        new file:   src/CommandAPI/CommandAPI.csproj
        new file:   src/CommandAPI/Controllers/CommandsController.cs
        new file:   src/CommandAPI/Program.cs
        new file:   src/CommandAPI/Properties/launchSettings.json
        new file:   src/CommandAPI/Startup.cs
        new file:   src/CommandAPI/appsettings.Development.json
        new file:   src/CommandAPI/appsettings.json
        new file:   test/CommandAPI.Tests/CommandAPI.Tests.csproj
        new file:   test/CommandAPI.Tests/UnitTest1.cs
```

These files are being tracked and are "staged" for commit.

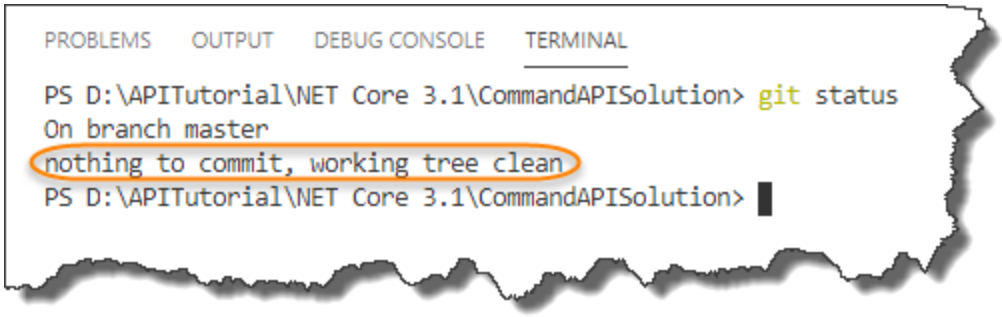
Finally, we want to “commit” the changes, (essentially lock them in), by typing:

```
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:



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



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!

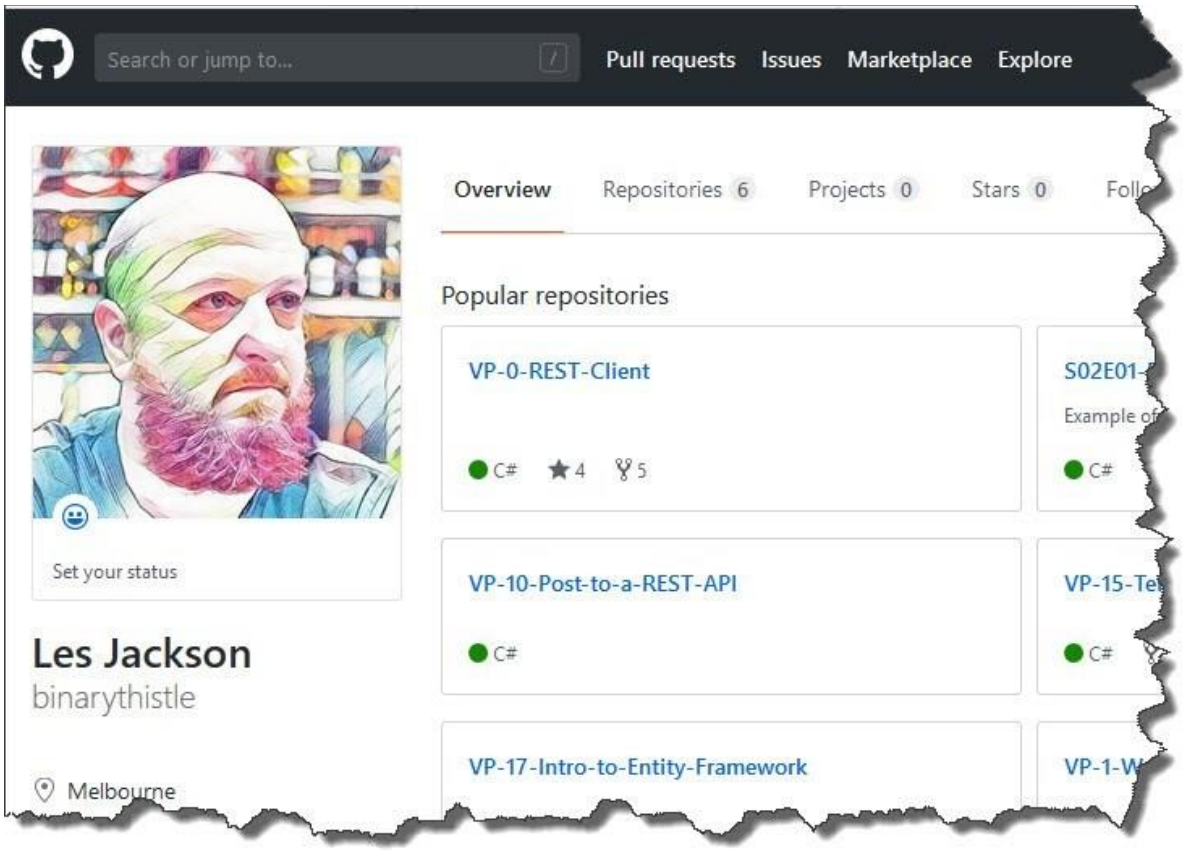
The Git web site also allows you to download the full “Pro Git” eBook, you can find that here: https://git-scm.com/book/en/v2

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 & 11), we need to configure a "remote Git repository" that we will:

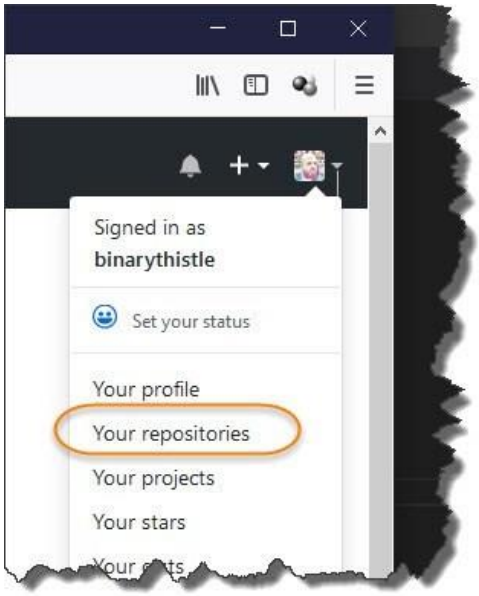
- Push to from our local machine
- Link to an Azure DevOps Build Pipeline to kick off the build process

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:

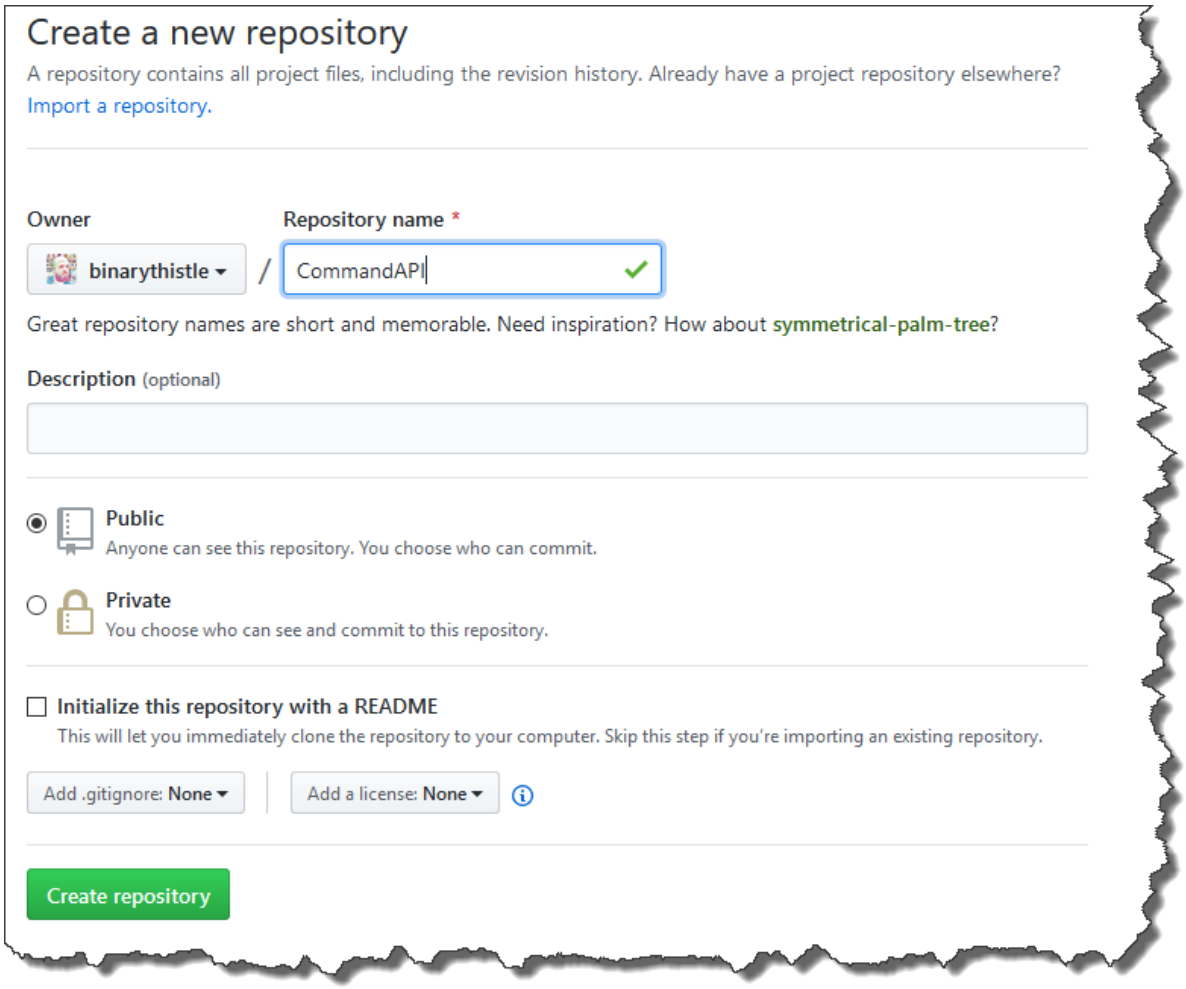


CREATE A GITHUB REPOSITORY

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":

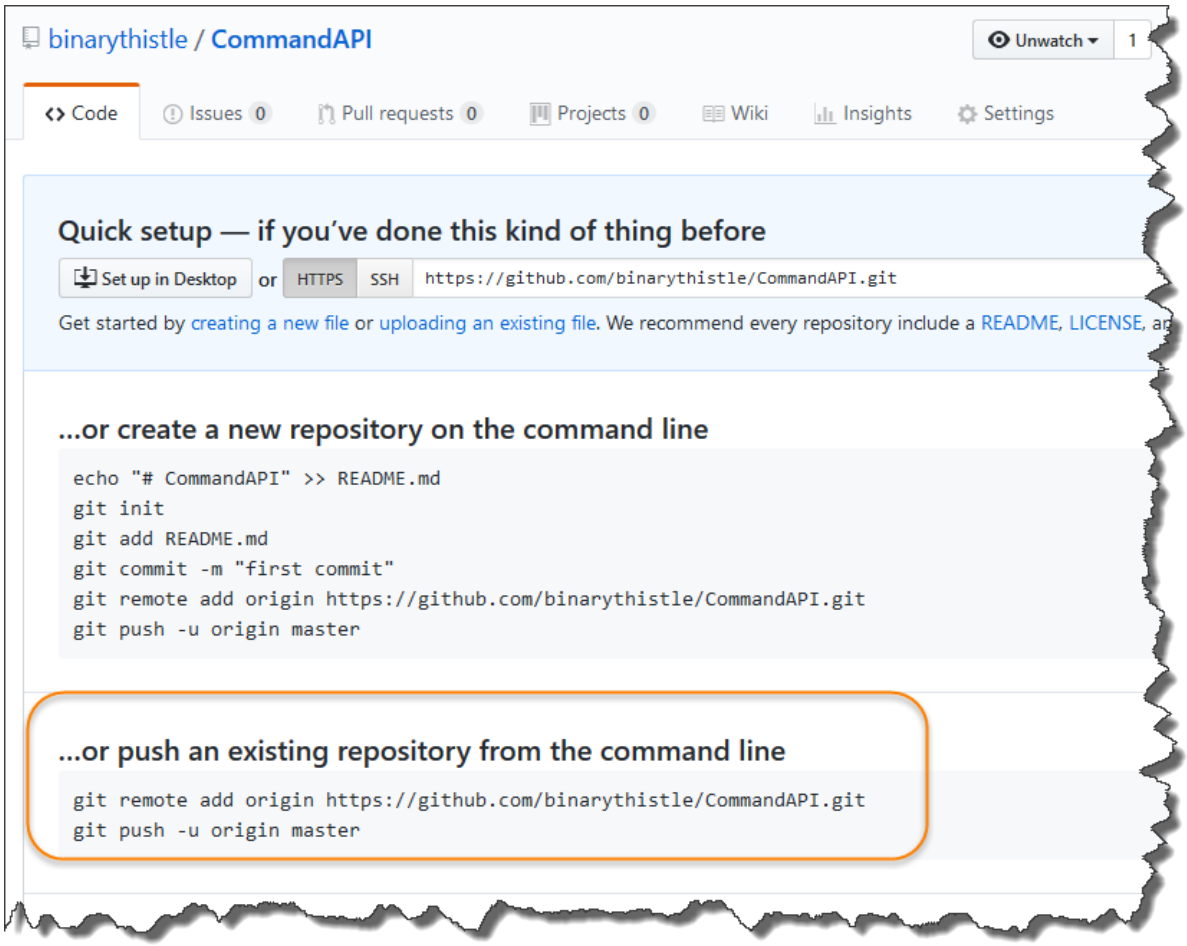


The Click “New” and you should see the “Create a new repository” screen:



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):

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!

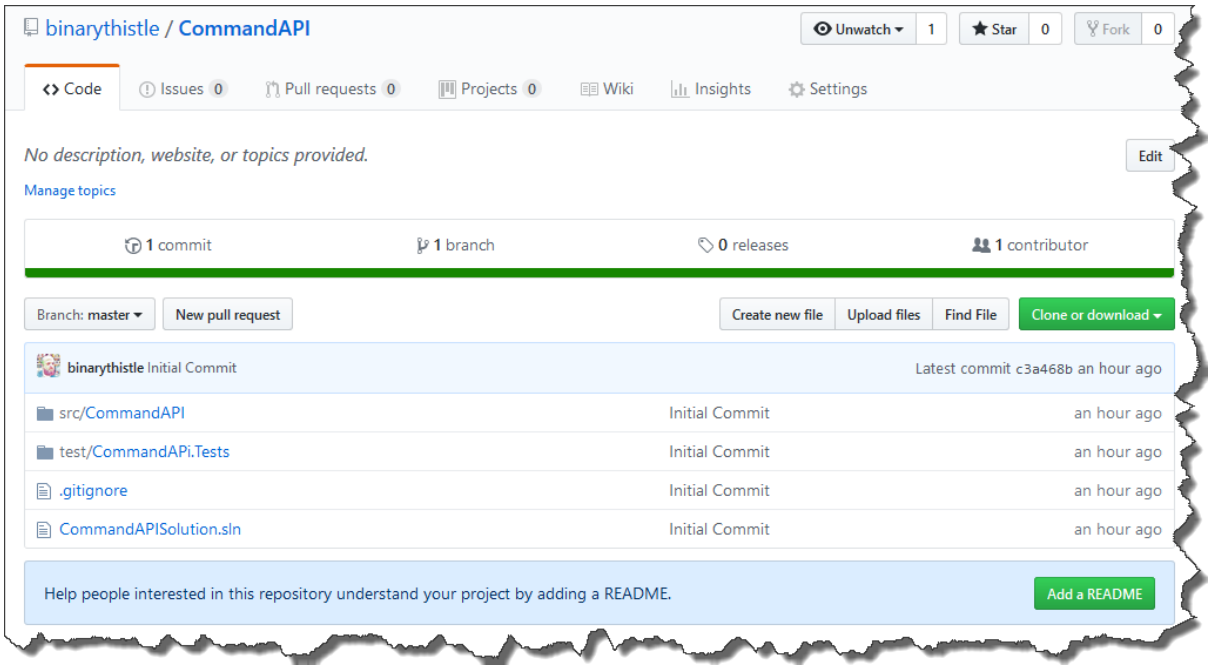
So What Just Happened?

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 going to introduce "data" to our API, meaning our Model & Data Access classes.

WHEN DONE, YOU WILL

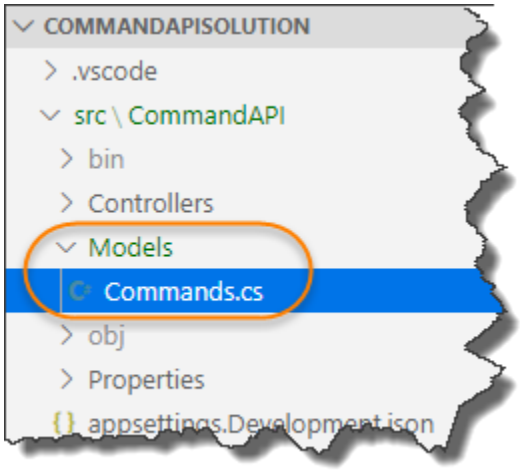
- Understand what a "Model" class is and code one up
- Use Entity Framework Core to create a Data Access (DBContext) class
- Use our Model and DBContext classes to store data in a PostgreSQL Database

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 PostgreSQL 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"
```

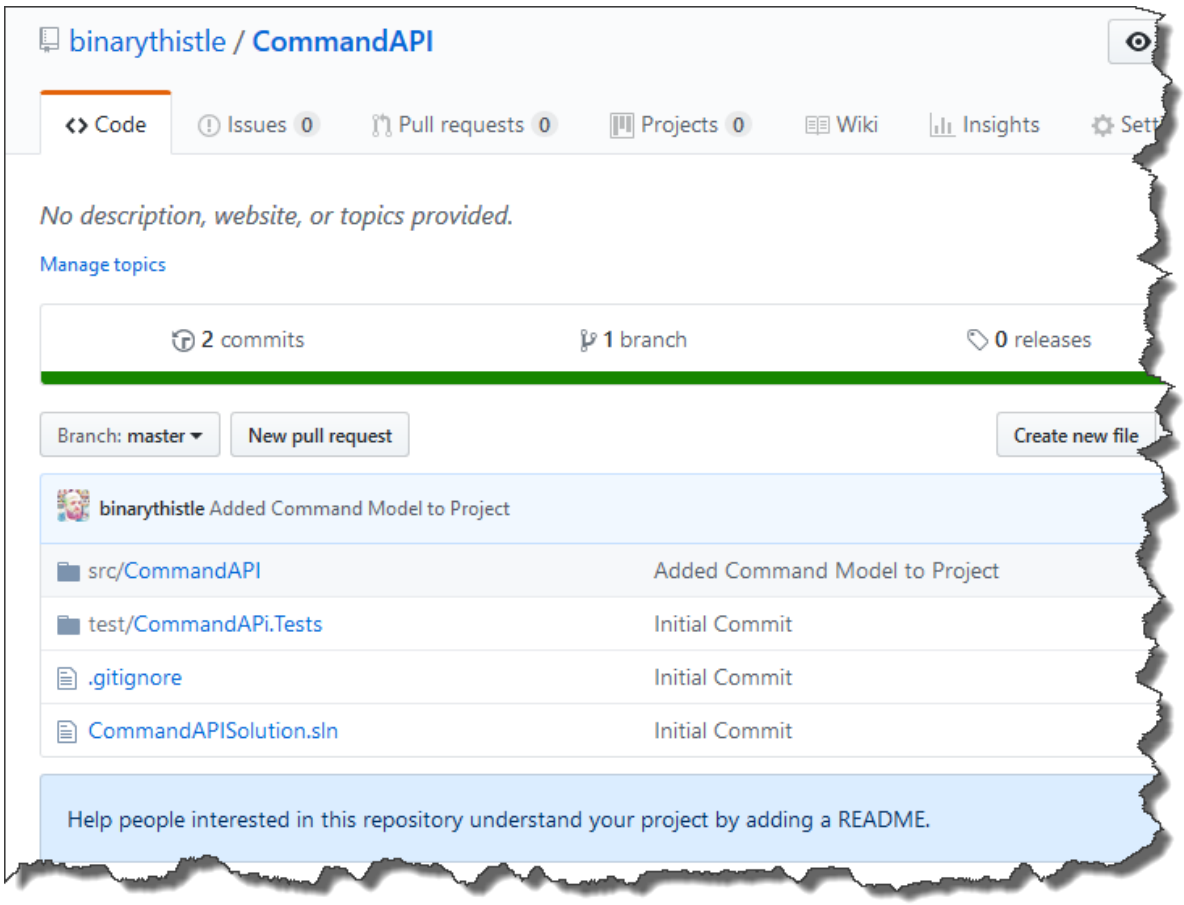
```
git push origin master
```

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.

PostgreSQL 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. In the first edition of this book I used Microsoft SQL Server as the DB of choice, but for the second edition we're moving to PostgreSQL simply because it's available on more platforms.

Using Docker

Now I'm going to use Docker to run my instance of PostgreSQL on my development machine, so if you've chosen that approach too, (or you want to see how easy it is to spin up an instance), read on. If you've already got a PostgreSQL instance running, you can skip to the Connecting with DBeaver section below.

At a commend prompt, simply type:

```
docker run --name some-postgres -e POSTGRES_PASSWORD=mysecretpassword -p
5432:5432 -d postgres
```

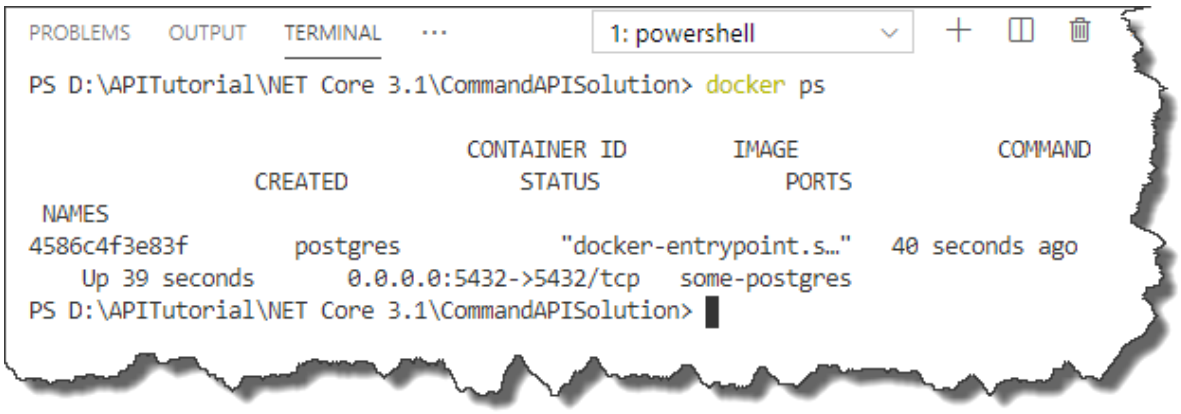
Note: this is all on one line.

Assuming you have Docker installed and it's running, (I don't like having Docker Desktop run automatically at start up so I manually start it when needed), you should see:

If this is the first time you've run this command you'll see that Docker is: "Unable to find image" locally, so it pulls it down from Docker Hub. Typing:

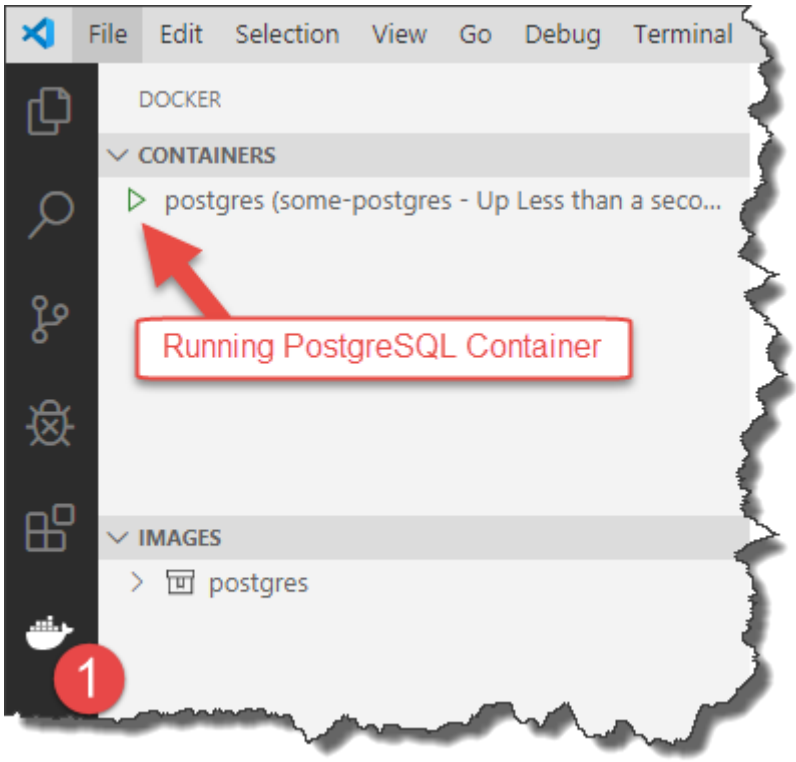
```
docker ps
```

Should show you the number of running containers:



Here you can see that we have one, which should be our PostgreSQL instance!

Just before I take you through the command we just issued in a bit more detail, if you installed the Docker plug in for VS Code, you should see something like:



From here you can stop the running container, (right click any entry in the containers box), and start it again etc. You will also see that it lists the Images you have on your machine.

When you run this command again, assuming there isn't a later version of the PostgreSQL image on Docker Hub, Docker will not attempt to download a new image, it will simply use the cached copy locally available to you.

Docker Command Prompt

Just so you understand what's happening let's just set through each of the command line arguments:

docker run

- Simple enough, this is just the primary command we use to run a container

--name

- By default, a running Docker container will just be identified by an ID, this ok, but when you come to issuing start and stop commands at the command line, these ID's can be cumbersome and prone to mistyping. The `--name` argument just allows you to "name" your container.

-e POSTGRES_PASSWORD=mysecretpassword

- The `-e` argument just means that we are supplying one or more "environment variables" into the container at start up. In this case we are setting the password for the default user: postgres

-p [internal port] : [external port]

- The `-p` argument is REALLY important - this is our port mapping. Without going into too much detail, a container will usually have an "internal" port, and we need to map an "external" port through to it in order for us to connect. Here the internal port our PostgreSQL is listening on is the standard 5432 PostgreSQL port, and we're just mapping externally to that same port number.

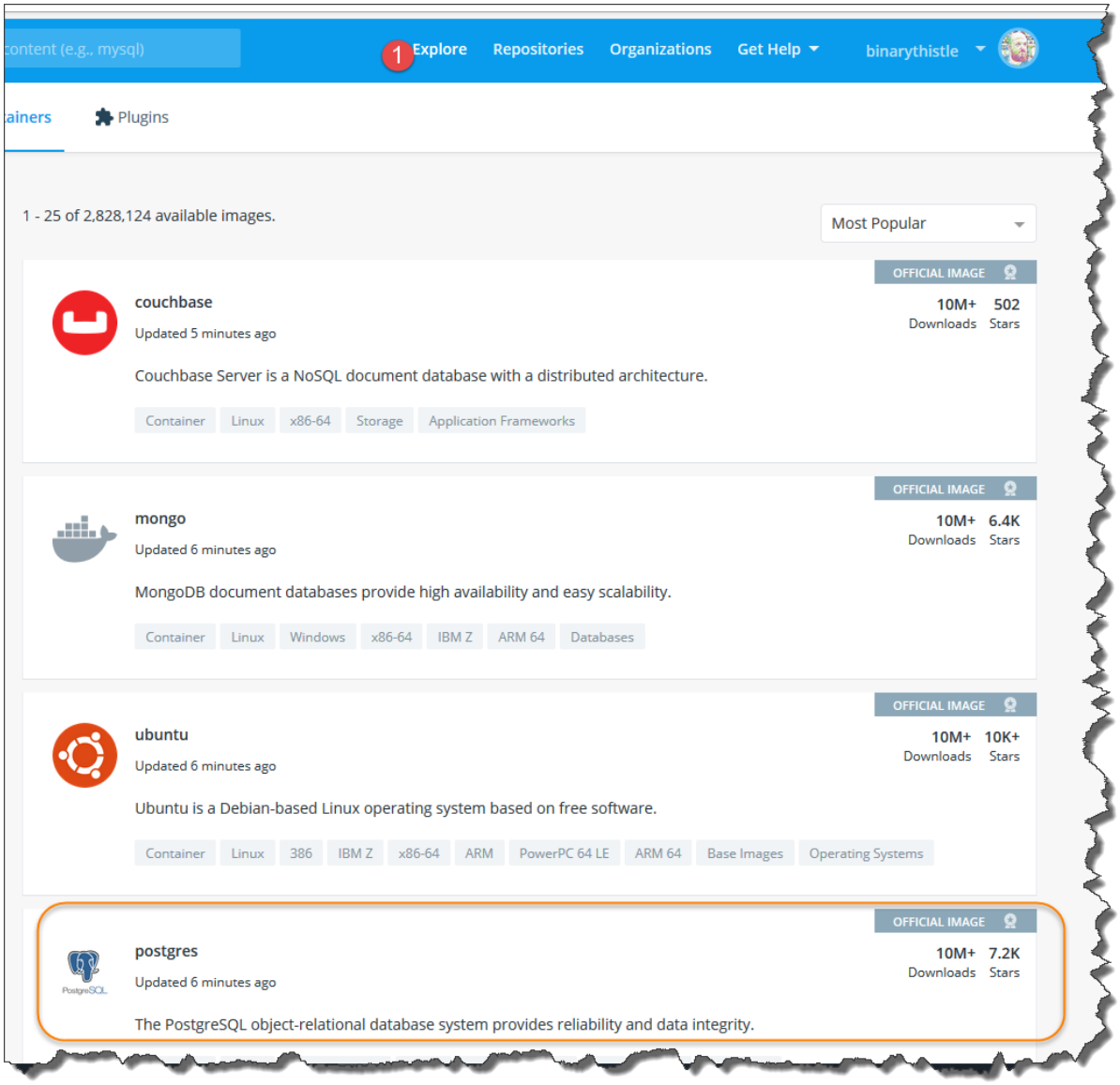
-d

- This argument just tells docker to run "detached" meaning that the command prompt is returned to us for subsequent use.

postgres

- This last argument is just the name of the image we want from Docker Hub.

If we go to https://hub.docker.com/ and click on "Explore" near the top of the screen, you'll get a list of the most popular Docker images available. These are most usually images provided by the vendor of the product in question:



In this instance you can see that postgres is #4

Connecting with DBeaver

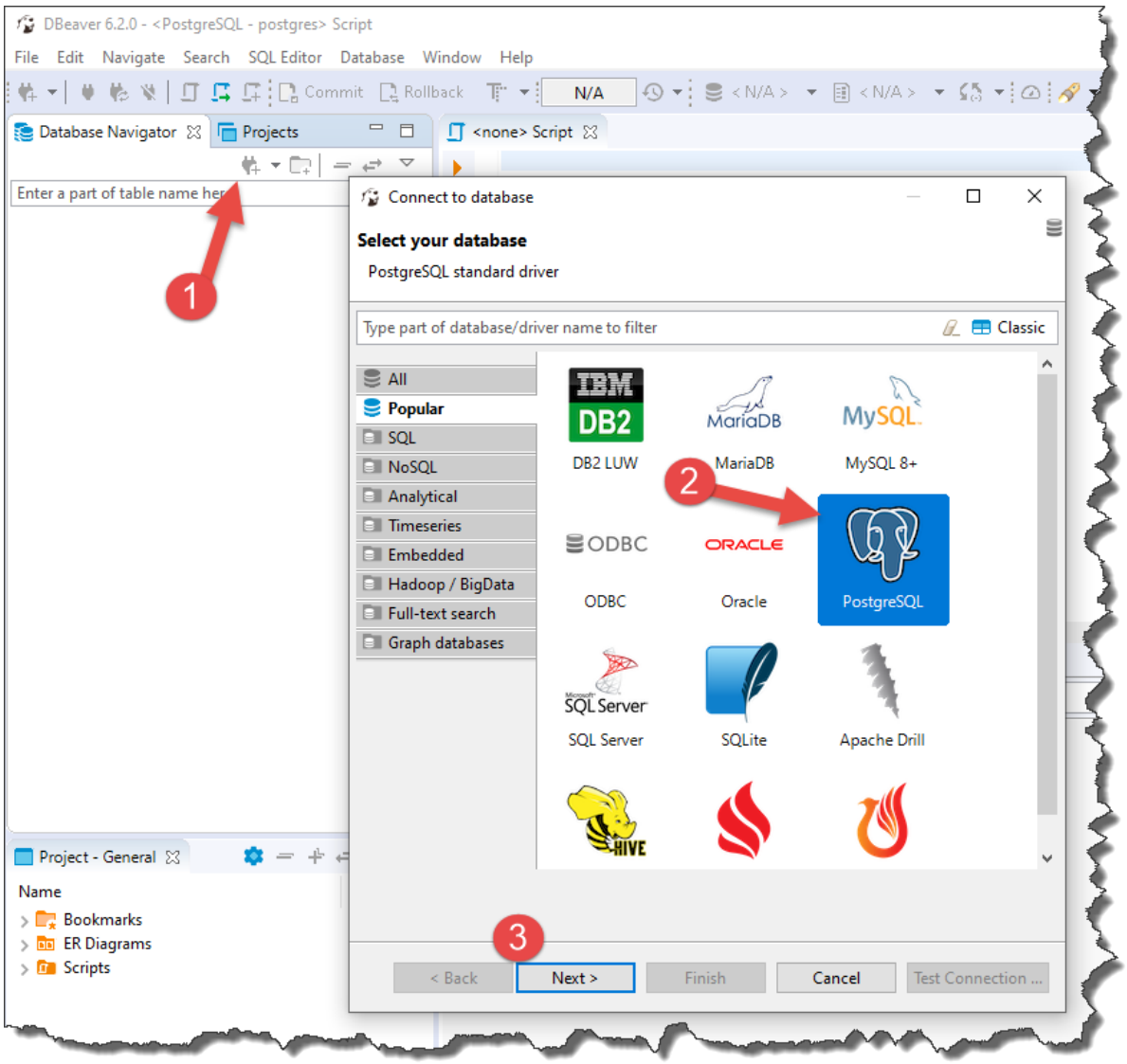
So by now you should either:

1. Have followed along with the Docker steps above and have a running PostgreSQL Docker container
2. Have a natively installed instance of PostgreSQL either locally or elsewhere

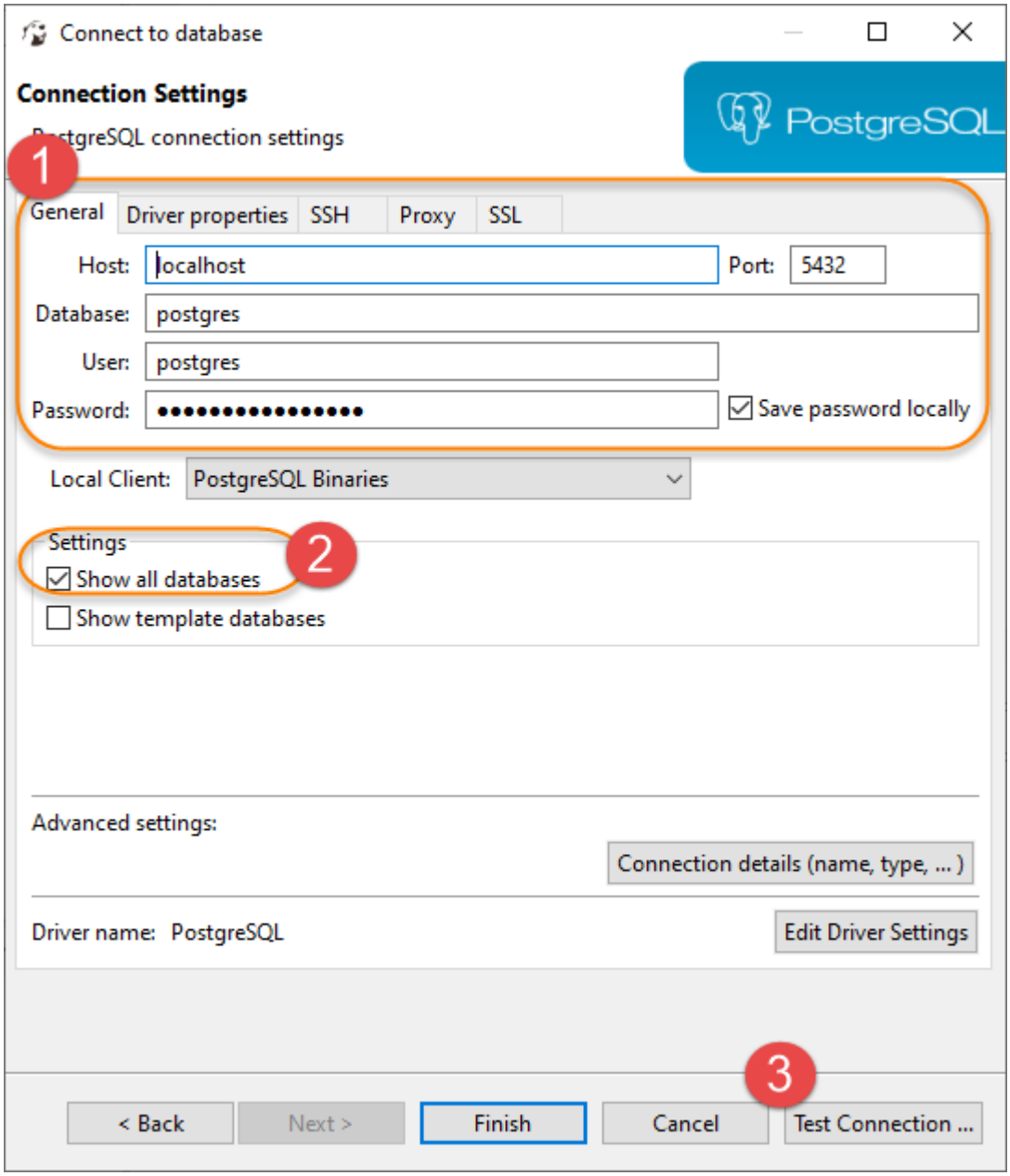
Now we want to connect in and see what we have…

Open DBeaver and:

1. Click the New Connection Icon
2. Select PostgreSQL
3. Click Next



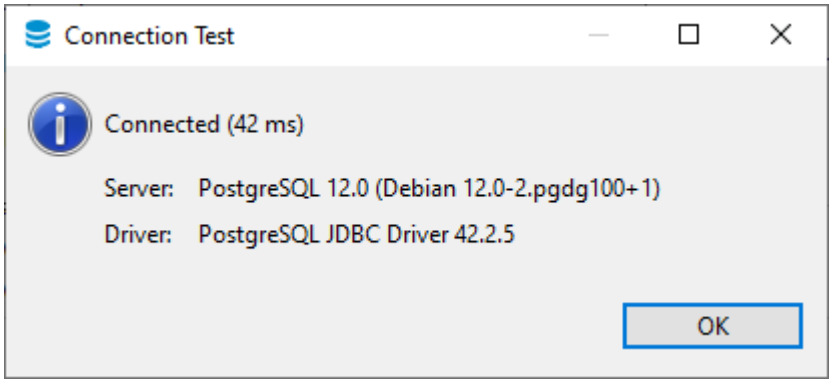
You'll then be presented with the Connection Configuration settings for PostgreSQL. Enter the details as appropriate for you, note the details I have here are good for the PostgreSQL instance I have running in Docker:



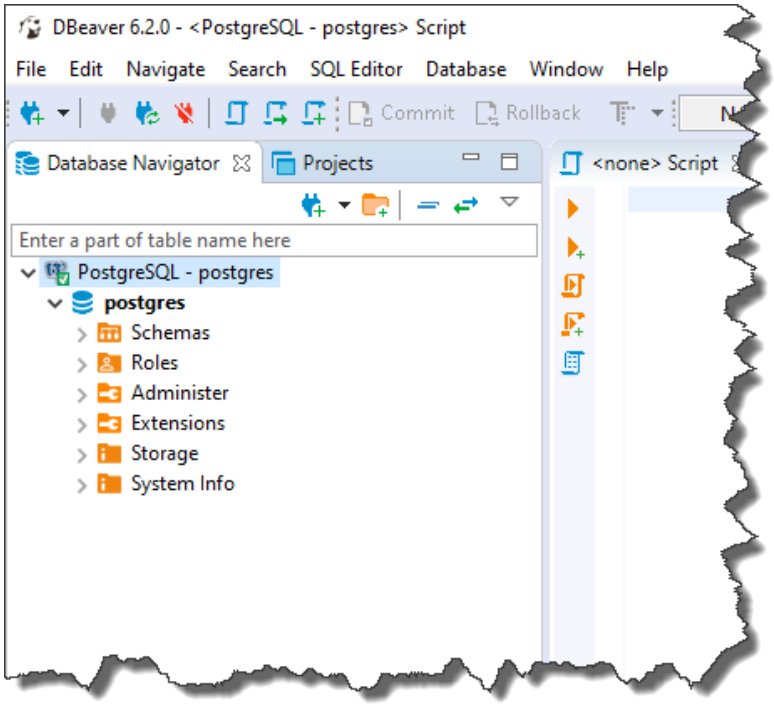
I'd also suggest:

- Tick: Show all databases
- You Click: Test Connection…

Assuming the connection is successful:



You should be OK to click “Finish”, this will add your connection to the main DBeaver environment:



Here you can see we have connected to the default database **postgres**, don’t worry we’ll be creating our own database for our API later.

Connection Issues

Connection issues will most usually be down to:

- Incorrect user credentials, (username or password)
- Incorrect / wrongly configured network attributes (e.g. Firewalls etc.)

If you’re running your PostgreSQL locally on the same machine as your code environment, you can usually avoid all the pain of a “remote” database. If you are running your database on a separate machine or even in a virtual machine, issues here are almost always due to Firewall or other network settings. I’m afraid I don’t have the space to troubleshoot that here, in fact it’s one of the reasons I recommend Docker as I’ve spent many an unhappy hour trouble shooting exactly this!

The default “super user” for PostgreSQL is, (not surprisingly), called: **postgres**. Again, depending on how you installed the server will depend on whether you set the value for this. There are articles on how to reset this

password, (assuming you have administrator / root privileges on the machine you've installed on), if you get stuck.

Assuming you have successfully connected though, we can move on.

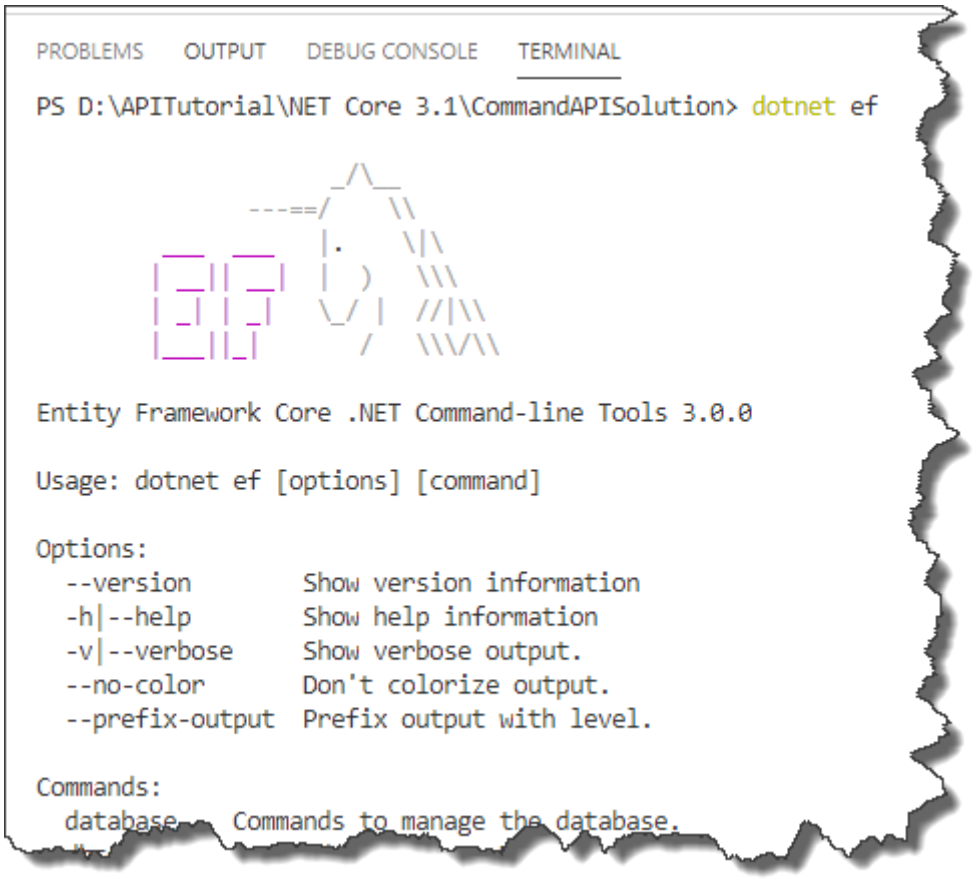
Entity Framework Command Line Tools

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:



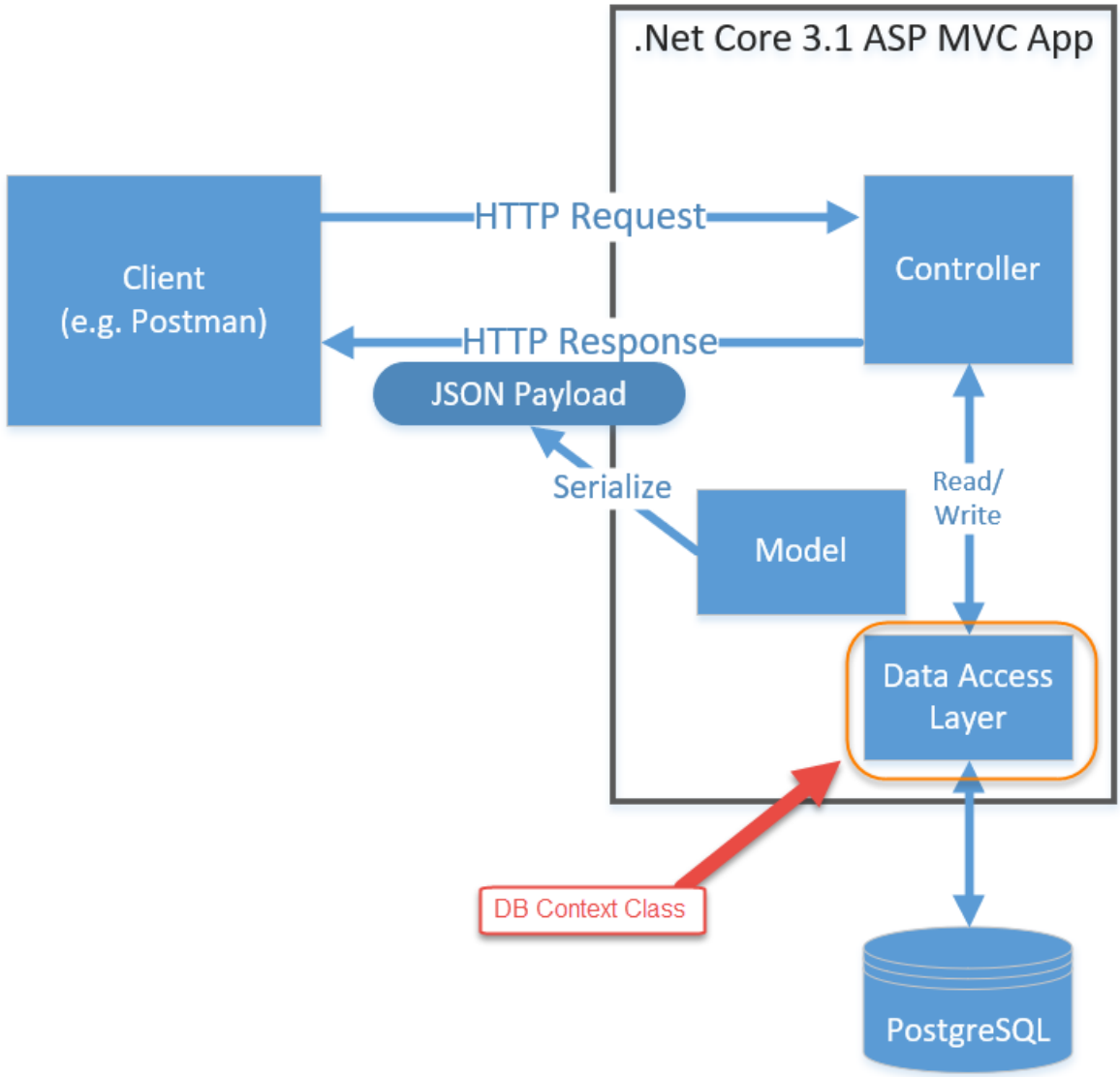
If you don't see that, simply run the following at the command line:

```
dotnet tool install --global dotnet-ef
```

And this will make the tools available to you globally.

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:



REFERENCE PACKAGES

First in order to use the features of Entity Framework core we're going to have to add reference 3 packages in our .csproj file:

- Microsoft.EntityFrameworkCore - Primary Entity Framework Core Package
- Microsoft.EntityFrameworkCore.Design - Design time components (required for migrations)
- Npgsql.EntityFrameworkCore.PostgreSQL - PosrgreSQL provider for Entity Framework Core

You can add these manually to the .csproj file, but I'd rather use the .NET Core CLI, to do so run the following commands in a terminal, (making sure you're "inside" the API Project folder: ***CommandAPI***:

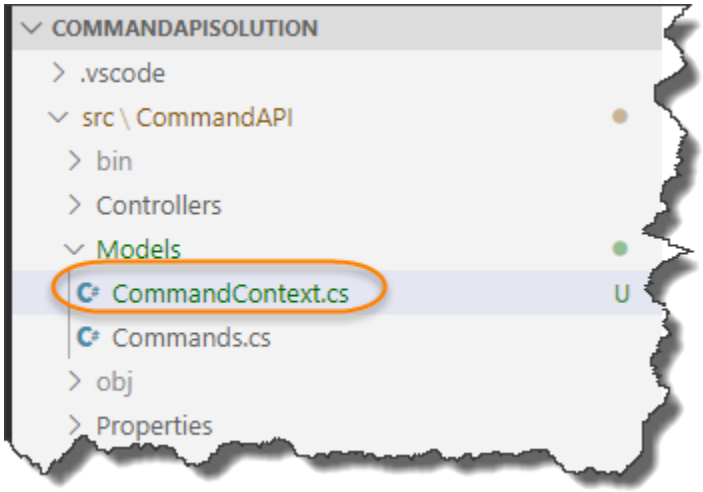
```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
```

Opening the .csproj file for our API project, you should see something like this:


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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.1">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Npgsql.EntityFrameworkCore.PostgreSQL" Version="3.1.0" />
  </ItemGroup>

</Project>
```

With the necessary packages added we can move on…

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 also:

```
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 PosrgreSQL 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 need to create a PostgreSQL Login that we can use as the "application user" of our, (as yet to be created), database. This is the account that the API will use to authenticate to the PostgreSQL server with and derive its permissions to run our Entity Framework Core "migrations", that will:

- Creating a new database
- Create or alter any tables
- Read, write and delete data

Learning Opportunity: Why should we not use the **postgres** user account that we previously used from within DBeaver to connect to our PostgreSQL server?

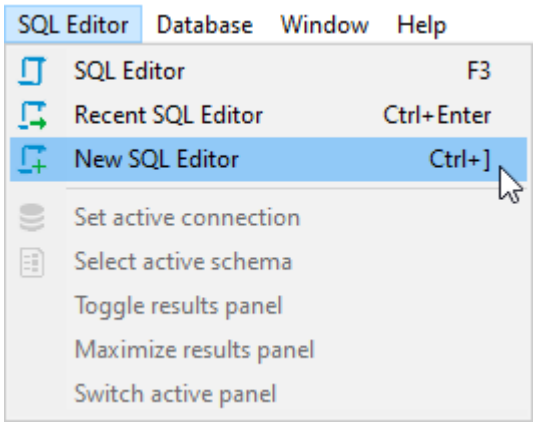
Creating a user can be done 1 of 2 ways using DBeaver:

- SQL Command
- Using the Graphical Interface

I'll show you how to do this Via SQL, once you learn that, using the Graphical UI to perform the same action should be a piece of cake.

CREATE USER – SQL

Open DBeaver and select SQL Editor-> New SQL Editor from the menu:

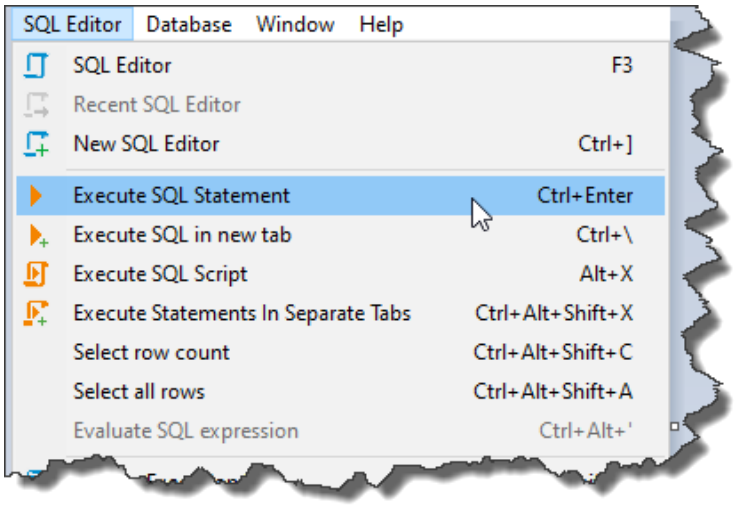


This should open up a new query widow, then simply enter the following SQL:

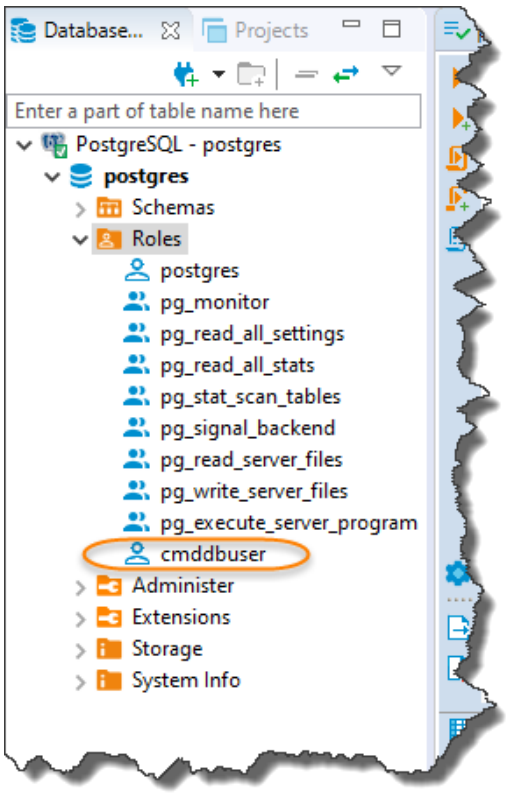
```
create user cmddbuser with encrypted password 'pa55w0rd!' createdb;
```

I've called our user: `cmddbuser` and given it a password of `pa55w0rd!`, you can of course alter these values to your own needs.

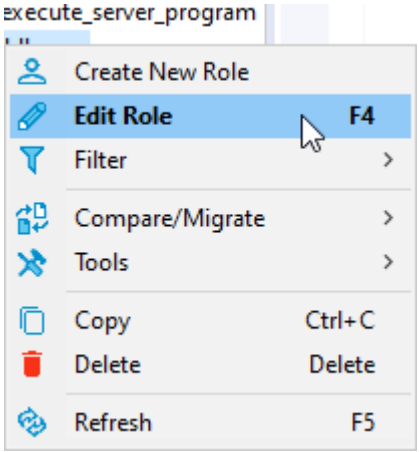
You can the hold Ctrl + Enter to execute the SQL statement or select "Execute SQL Statement" from the SQL Editor menu:



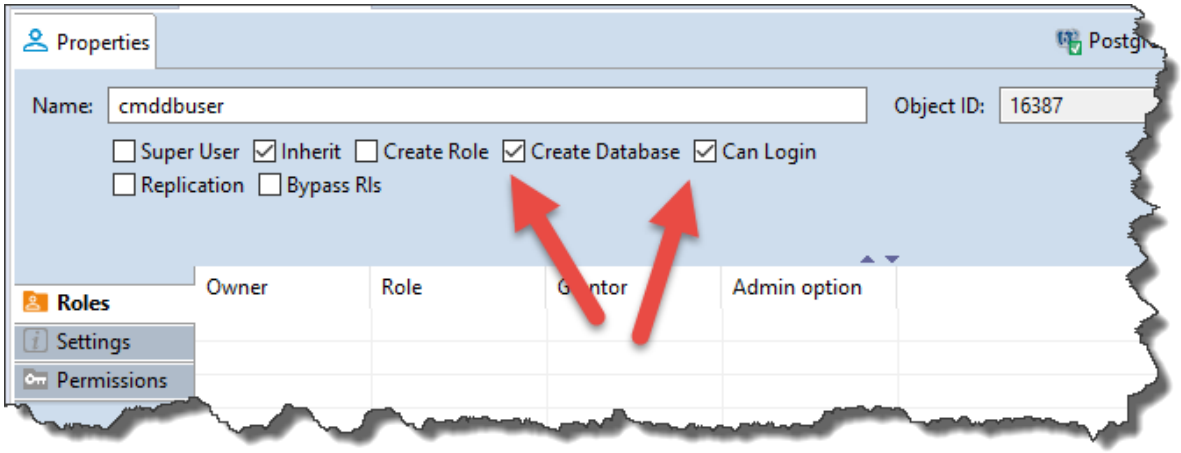
The command should execute successfully, and if you then expand: postgres-> Roles you should see your newly created user, (or Role in PostgreSQL-speak). If you don't, right click the "Roles" folder and select "Refresh":



Right click the newly created role and select “Edit Role”, (or you can just hit F4):



The resulting information should detail that our user can login and create databases which is critical when we come to running migrations.



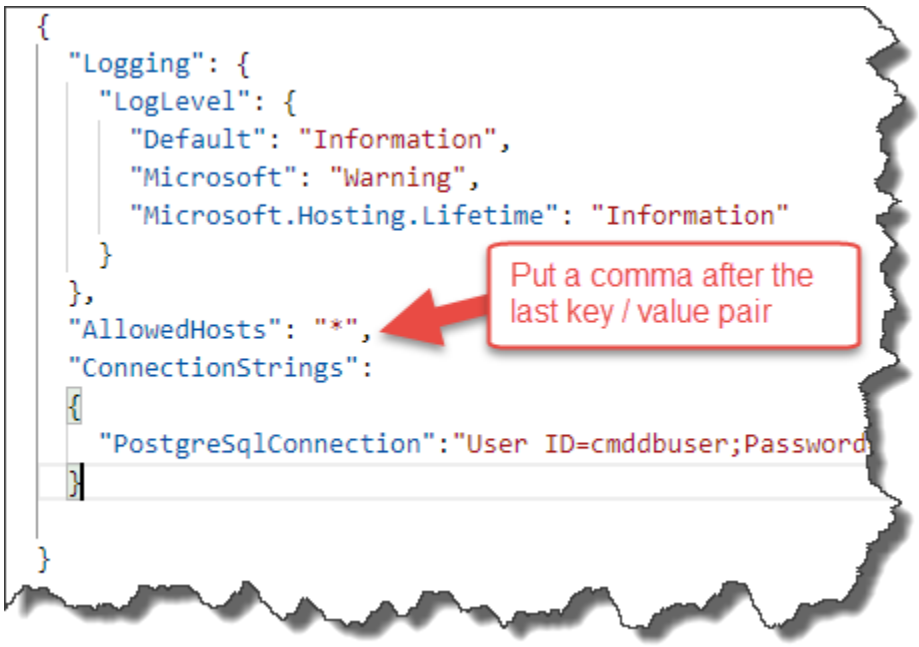
Learning Opportunity: Using the properties that we've set above as guide, see if you can use DBeaver to create a new Role using the Graphical UI & menus.

Open ***appsettings.json*** and add the following json string to the file, (again make sure you replace the User ID and password to match the user *you* created above):

```
"ConnectionStrings":
{
    "PostgreSqlConnection":"User ID=cmddbuser;
    Password=pa55w0rd!;
    Host=localhost;
    Port=5432;
    Database=CmdAPI;
    Pooling=true;"
}
```

So your file should look something like this[14]:

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



Some points to note about the connection string:

- The "name" of the connection string is: `PosrgreSqlConnection`
- The actual connection string is made up of the following components, separated by a semi-colon:
  - `User ID`: The login for our Postgres Server (we created this in the last section)
  - `Password`: The password for our login – stored in pain text – not very secure[15]!
  - `Host`: The host name of our PostgreSQL server
  - `Port`: The port our PostgreSQL server is listening on
  - `Database`: This is our database (or will be our database – it does not exist yet)
  - `Pooling`: Connection pooling, (essentially sharing), is being used

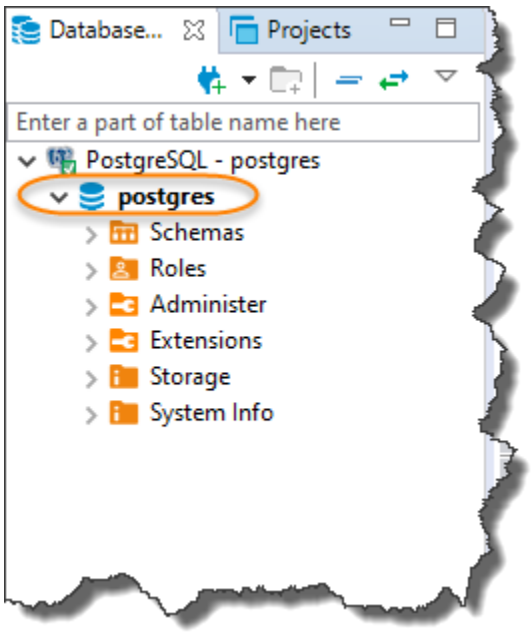
Learning Opportunity: If you've never use Javascript Object Notation, (JSON), before it's basically a way to represent nested key / value pair data as well as array data. Its power lies in its simplicity, I have an YouTube video that takes you through the fundamentals.

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, (**CmdAPI**), but the actual database does not yet exist on our sever, a quick look at the databases in DBearver will confirm that:

[15] We will remedy this in the next chapter



We only have the default **postgres** database, but as yet **CmdAPI** 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:

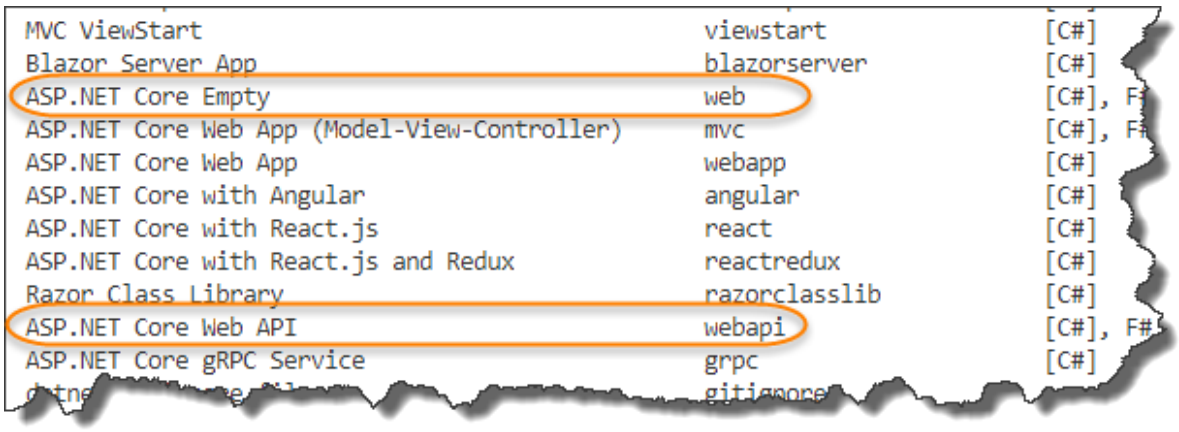
- A Database Server, (but actually no **CmdAPI** "database" as yet!)
- A Model (`Command`)
- DBContext (`CommandContext`)
- DBSet (`CommandItems`)
- Connection String to our database server

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?

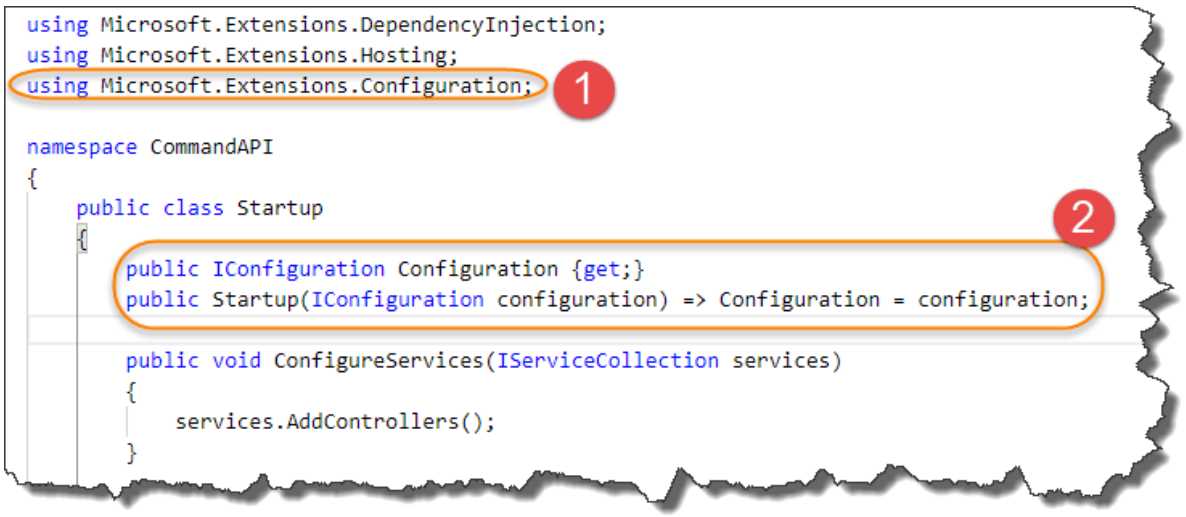


We chose “web” to provide us with an empty shell project. Well if you had chosen “webapi”, the “Configuration” code we’re about to introduce below, would have been provided as part of that project template. I deliberately choose not to do that so we have to manually add the following code – as I think 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.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;

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.

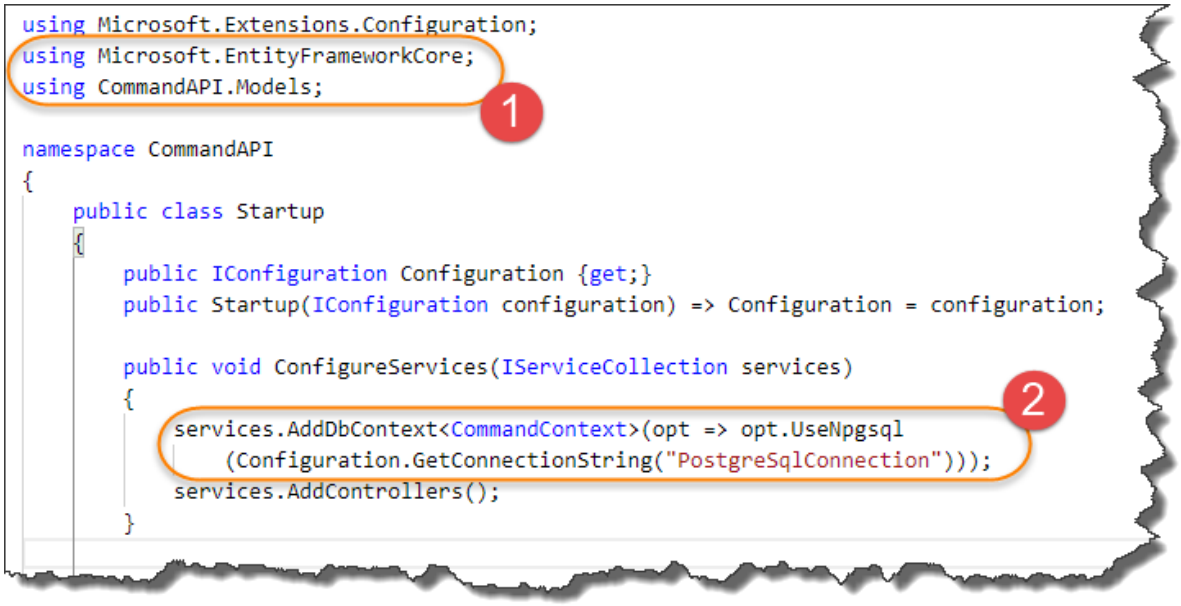
The last thing we have to do is register our `DBContext` in the `ConfigureServices` method and pass it the connection string, (via a configuration interface). So add the following `using` directives to your Startup class:

- `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.UseNpgsql
        (Configuration.GetConnectionString("PostgreSqlConnection")));

    services.AddControllers();
}
```

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, (`PostgreSqlConnection`), 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 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 - 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

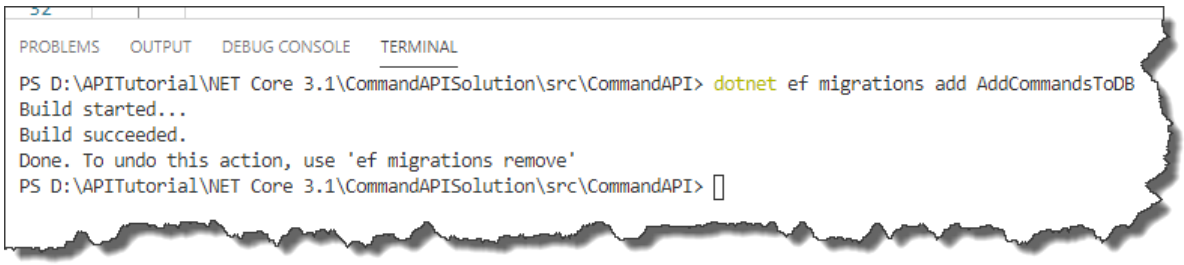
Here we are using "code first", (we've already created our command model), so we now have to "migrate" that to our database, we do this via something called, drum roll… Migrations!

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):

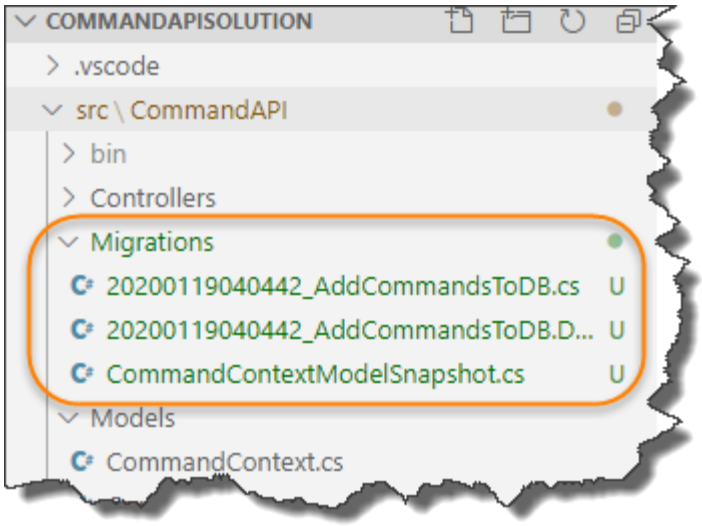
```
dotnet ef migrations add AddCommandsToDB
```

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:



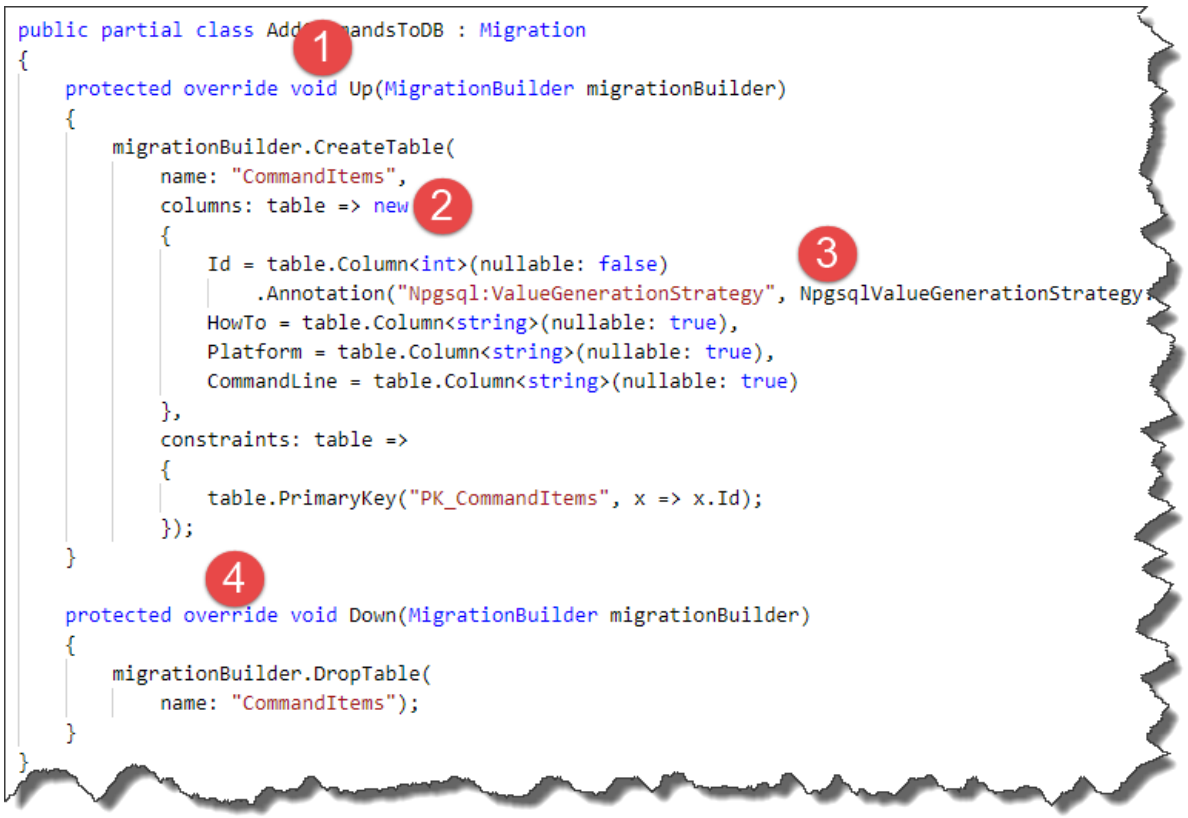
Next you should see a new folder appear in our project structure, called “Migrations”:



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

20200119040442_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 Add[illegible]mandsToDB : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "CommandItems",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("Npgsql:ValueGenerationStrategy", NpgsqlValueGenerationStrategy
                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. This method is called to create new stuff
2. The creation of a table and its columns, noticing the nature of the “Id” column
3. Database provider specific annotations / instructions
4. A “Down” method. Used to roll back the changes made in the Up method

Warning! Point 3 above is of note here. I previously thought the migrations file, (not sure why I thought this), was agnostic of the database that you’re using. That is incorrect.

The migrations file will look slightly different depending on which type of database you choose to use, (e.g. SQL Server Vs. PostgreSQL etc.). I learned this when I:

1. Used SQL Server as my database
2. Registered my DB Context in `ConfigureServices` with `opt.UseSqlServer…` (and not `opt.UseNpgsql`)
3. Ran & Generated my Migrations File
4. Then switched my provider to PostgreSQL, (`opt.UseNpgsql`), then attempted to use that migrations file to generate my DB (we’ll do this in a bit).

It failed.

You can examine a “SQL Server” migrations file and look for the differences in the Source Code for the original project for the first edition of this book here on GitHub.

Long story short, you’ll need to re-generate your migrations if you switch database providers.

Note: at this stage we *still do not* have the **CmdAPI** 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
```

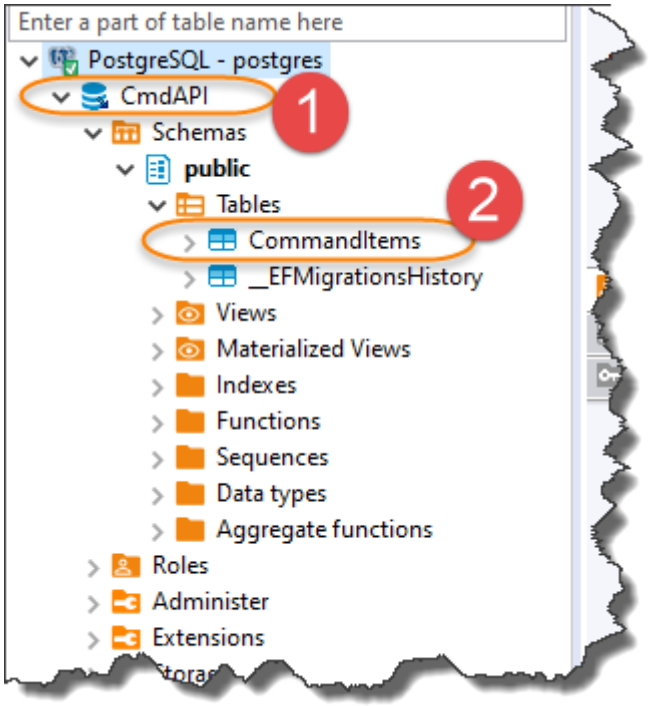
Our migration is run, as reflected in the following output:

```
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI> dotnet ef database update
Build started...
Build succeeded.
Done.
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI>
```

A number of things happen here:

1. Our database (**CmdAPI**), is created as it did not yet exist on the target server
2. 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).
3. Our **CommandItems** table is created which is the persistent equivalent of our `Command` model(s).

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



1. CmdAPI 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 DBeaver query window.

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:

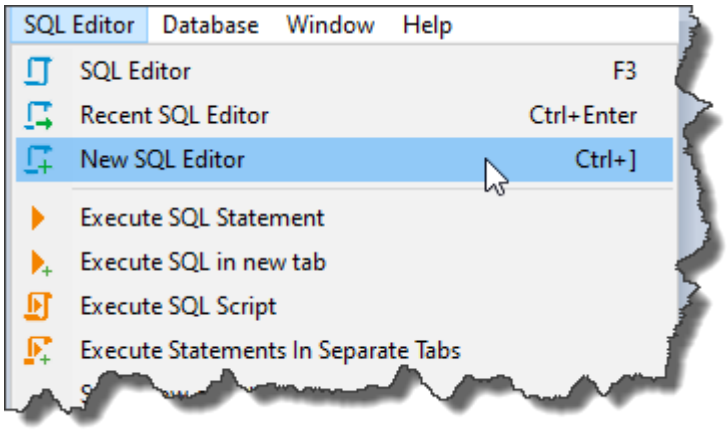
- dotnet ef migrations add
- dotnet ef database update

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

ID[16]	HowTo	Platform	CommandLine
1	Create an EF Migration	Entity Framework Core CLI	dotnet ef migrations add
2	Apply Migrations to DB	Entity Framework Core CLI	dotnet ef database update

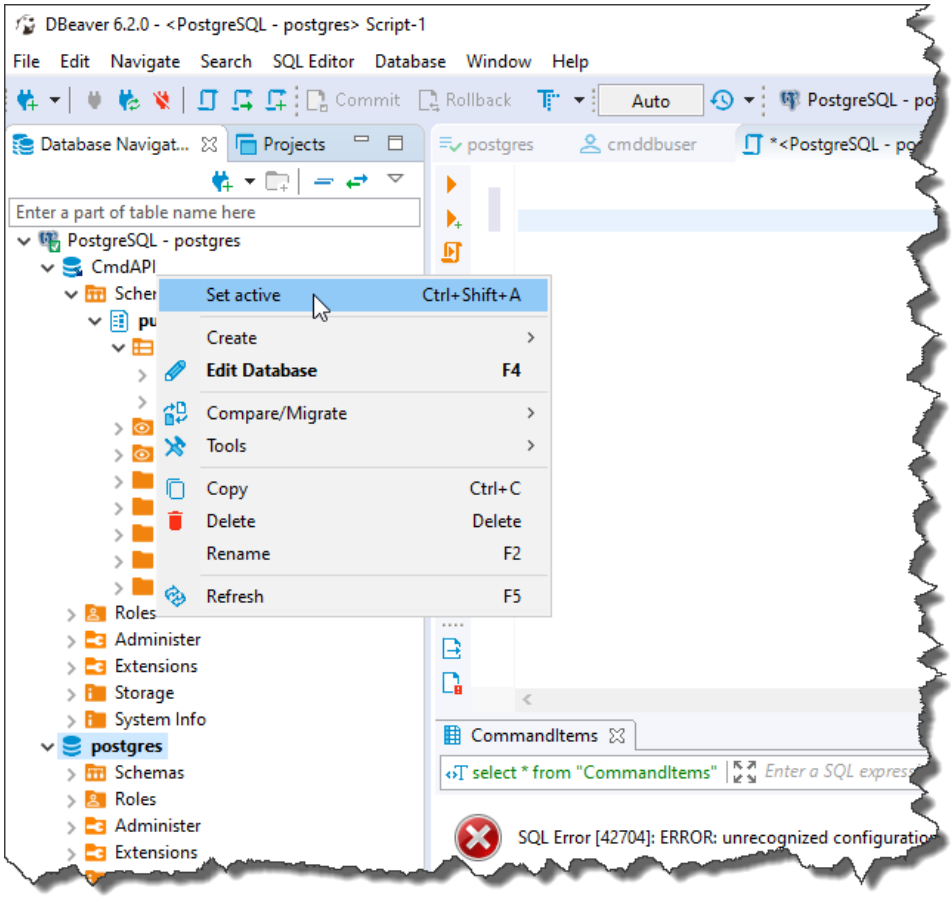
To add this data via a SQL INSERT command in DBeaver:

- Open DBeaver & connect to the server
- From the SQL Editor Menu, select "New SQL Editor"

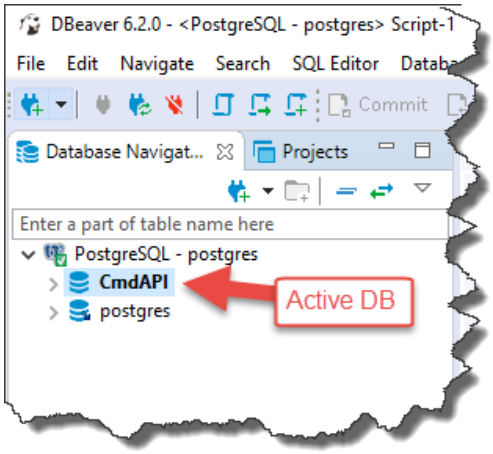


This will, (not surprisingly open a new query window. We then need to set the "active" database so that when we write a query DBeaver knows which one we want to use. Simply right click the database you want to set as the active one, (in our case CmdAPI), and select "Set active":

[16] 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.



This should change the name of the database to “bold”:



If you’re using SQL Server, you can set the active DB in the SQL editor by placing `use database_name;` before any of your queries, e.g.”

```
use CmdAPI;

select * from commanditems;
```

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

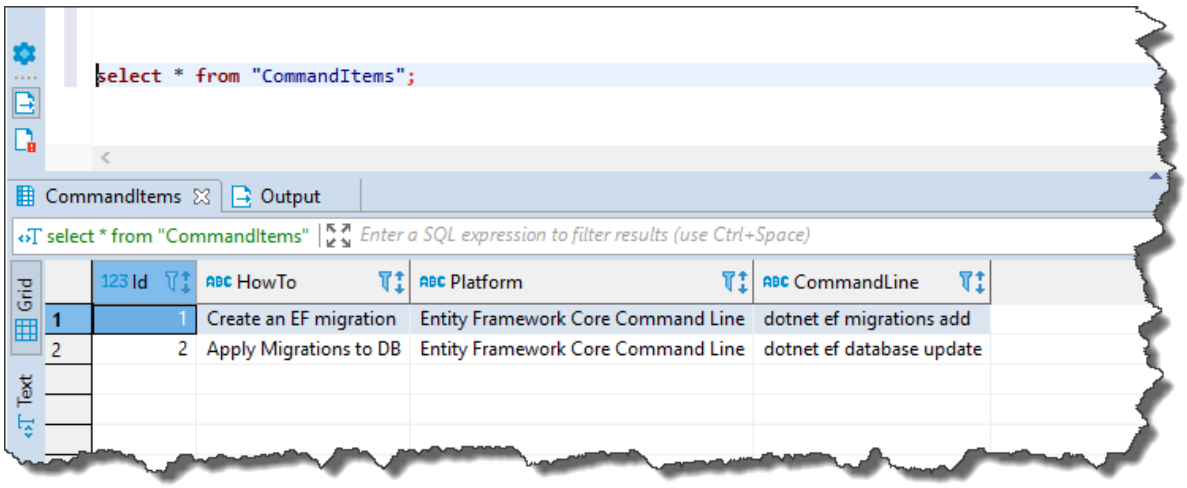
```
insert into "CommandItems" ("HowTo", "Platform", "CommandLine")
```

```
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');
```

After that hit Ctrl+Enter or select "Execute SQL Statement" from the SQL Editor menuu 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:



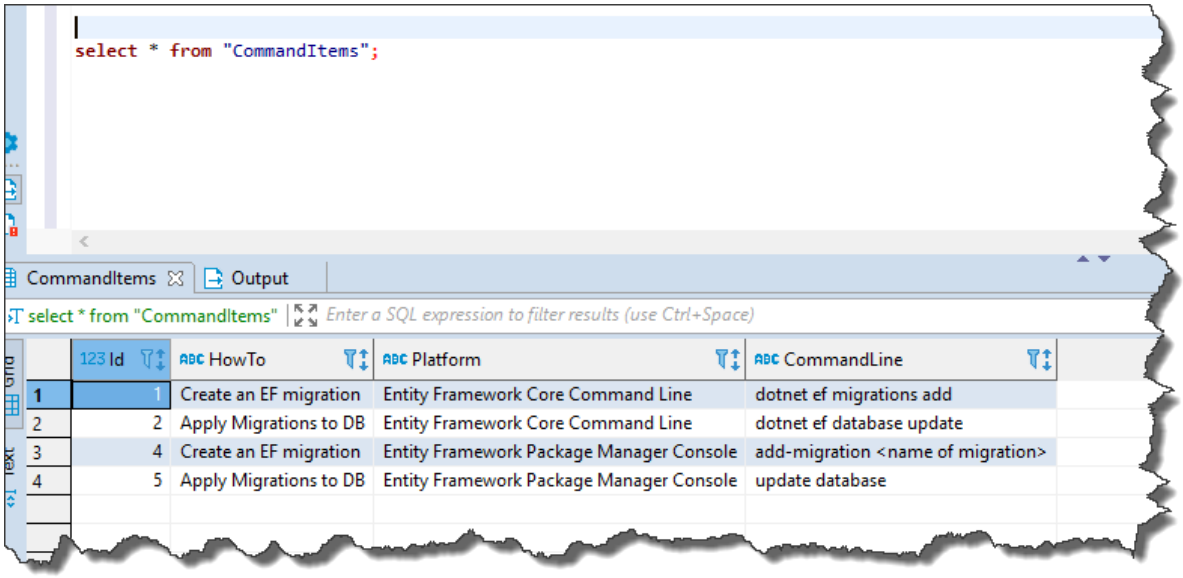
If you read my blog post on Entity Framework, you'll have noticed by now that the commands used in that tutorial are different to those used here. That's because in that tutorial, we're using the "Package Manager Console" in Visual Studio to issue commands for Entity Framework, (not Entity Framework 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:



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:

```
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"};
        }
    }
}
```

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):

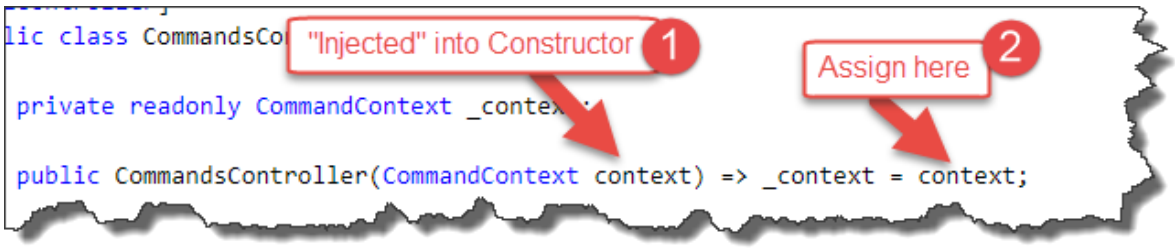
```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
```

```
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:

```
CommandAPI > Controllers > CommandsController.cs > ...
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;
        }
    }
}
```

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`).

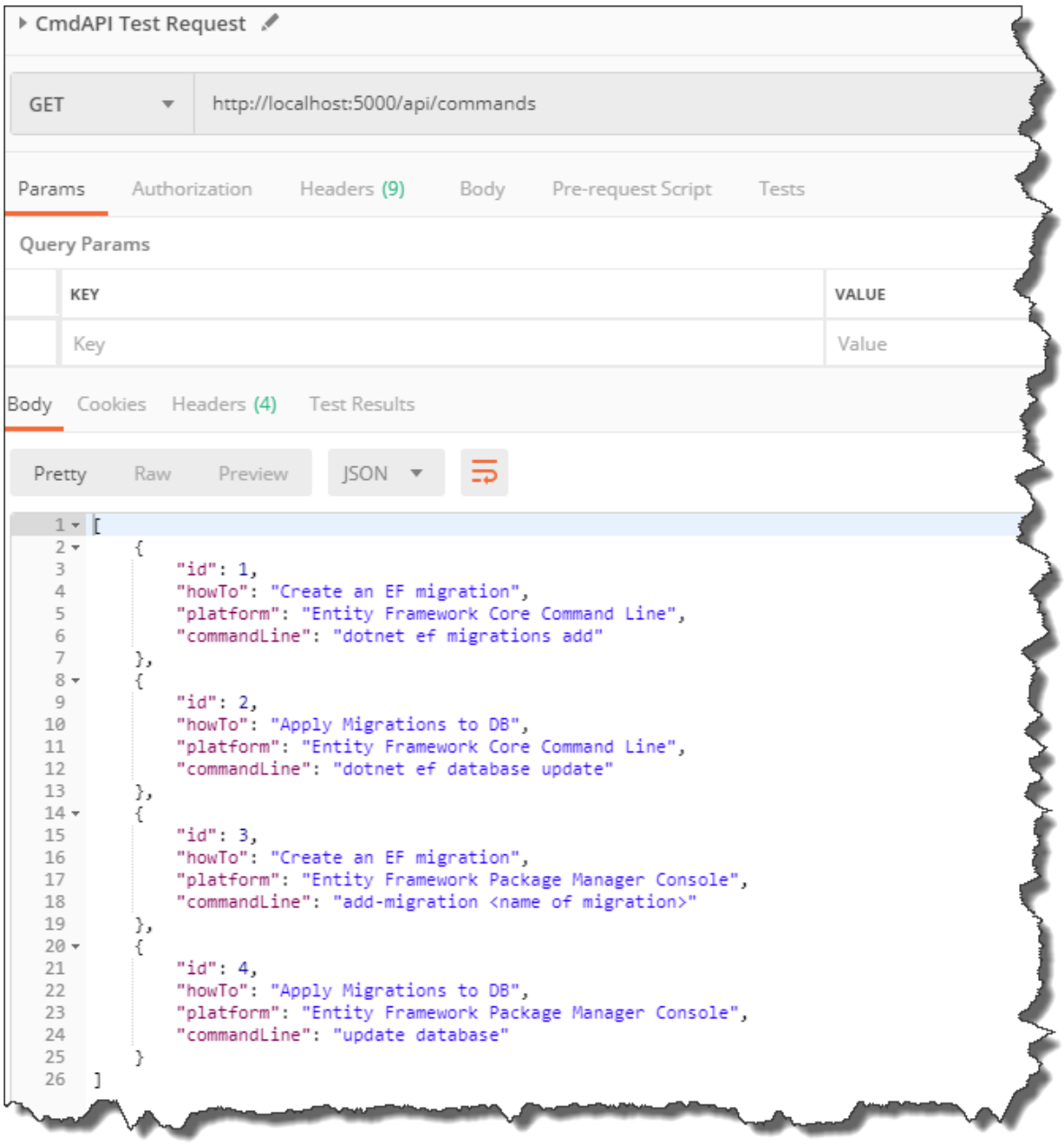
Let's save the file, and build our project to test for errors:

```
dotnet build
```

Assuming all is well let's 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 built 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 as 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…

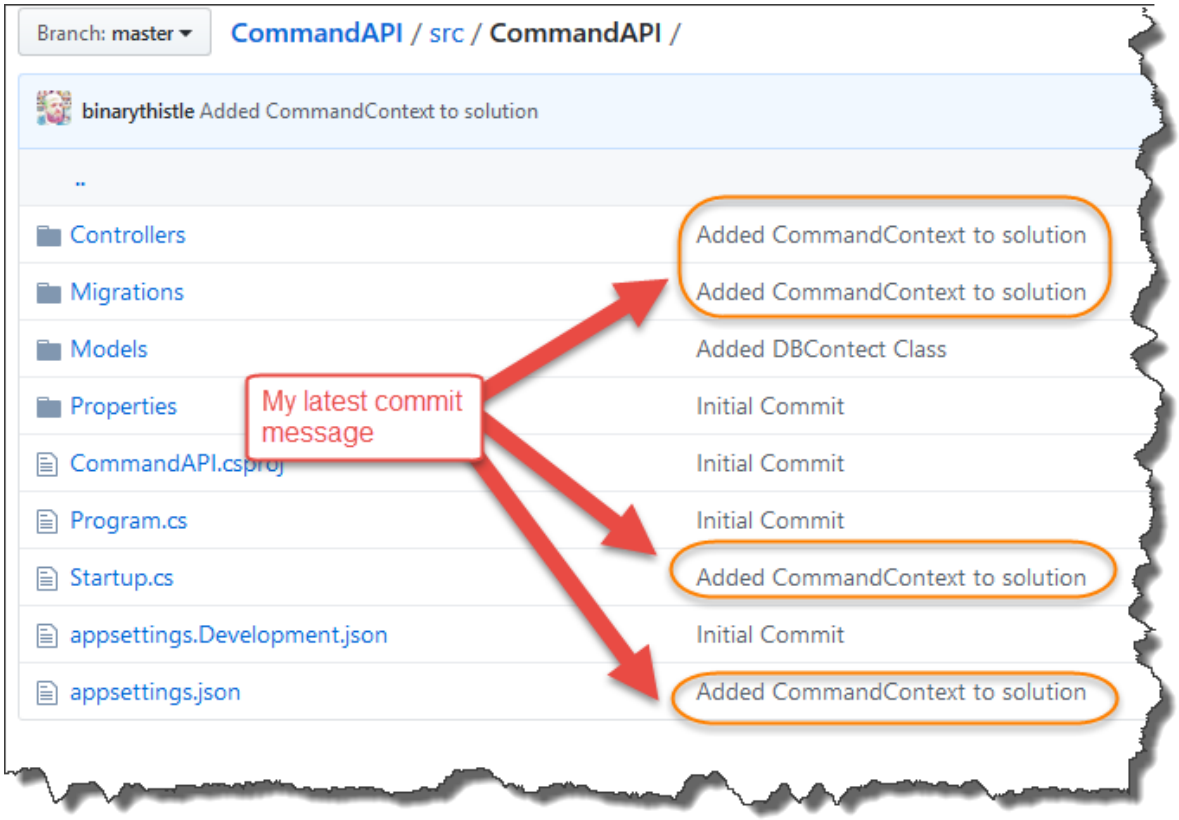
Redact Our Login and Password

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.:

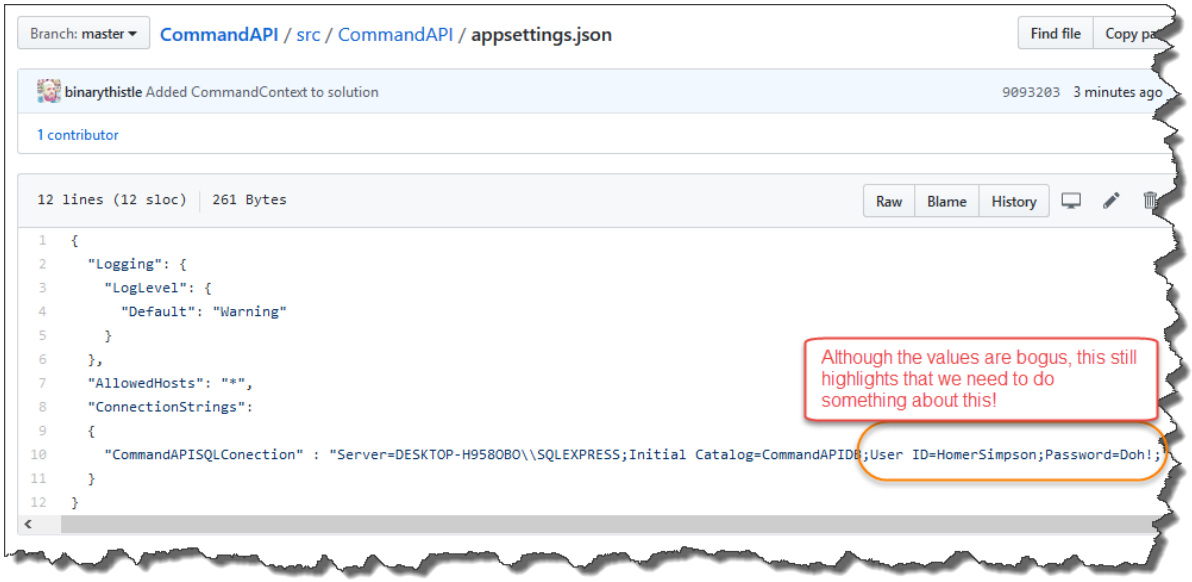
```
"ConnectionStrings":
{
  "CommandAPISQLConection" : "Server=DESKTOP-H958OBO\\SQLEXPRESS;Initial Catalog=CommandAPIDB;User ID=HomerSimpson;Password=Doh!;"
}
```

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 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 a way 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".

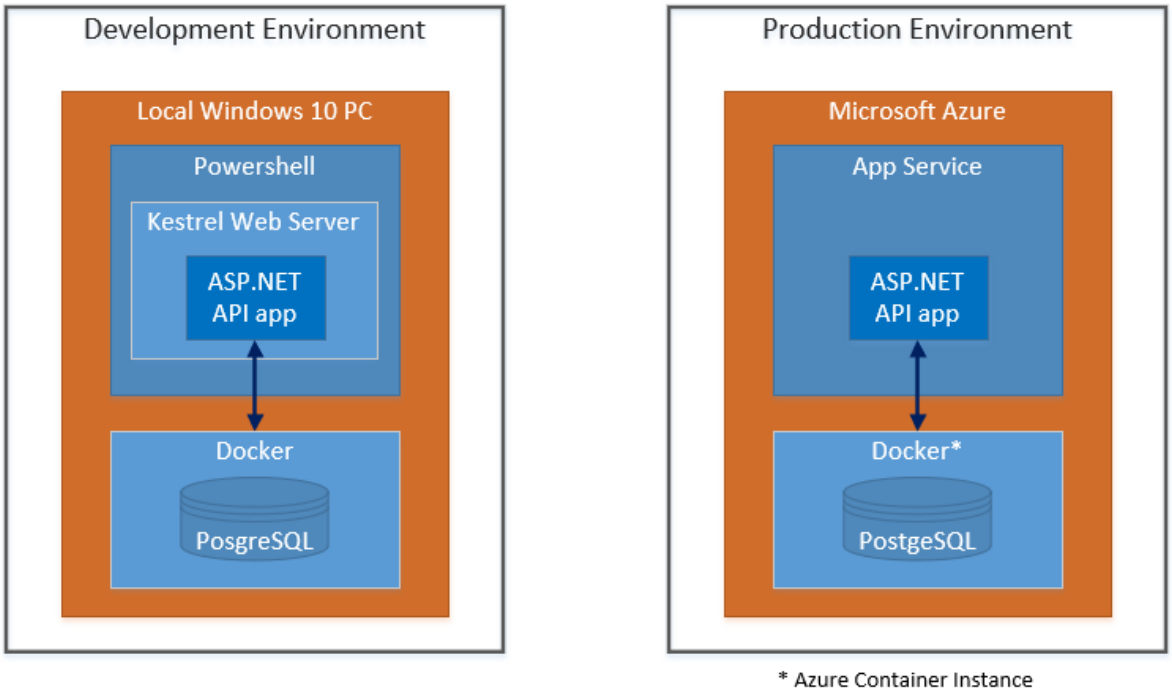
Les's Personal Anecdote: Replicating a Production environment accurately can be tricky, (and expensive), especially if you work in a large corporate environment with lots of "legacy" systems that are maintained by different 3rd party vendors – coordinating this can be a nightmare.

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:12662",
      "sslPort": 44343
    }
  },
  "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:

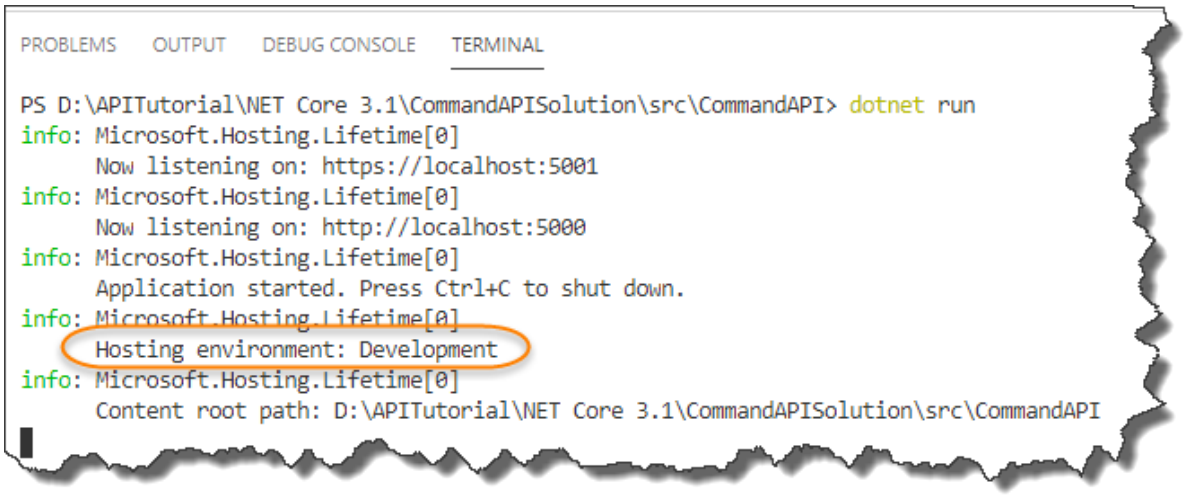
- `IISExpress`
- `IIS`
- `Project` (which launches the Kestrel web server)

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.:



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[17]!

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

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.

[17] 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.

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

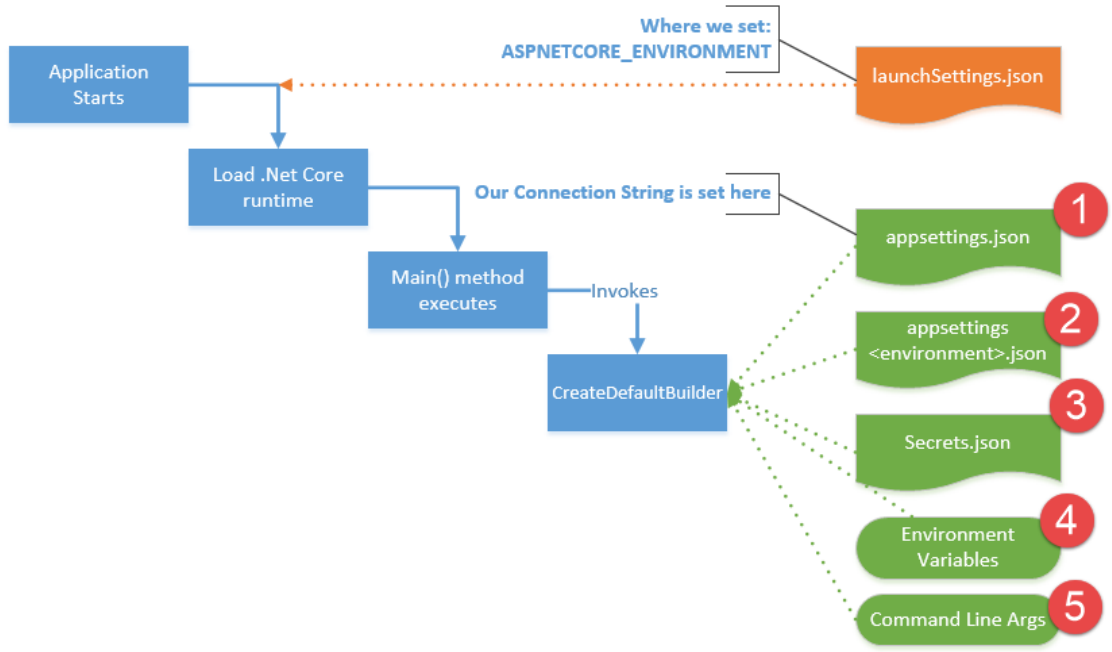
Using "code" to determine the runtime environment and make a decision.

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, `PostgreSqlConnection`, 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 `PostgreSqlConnection` 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***.

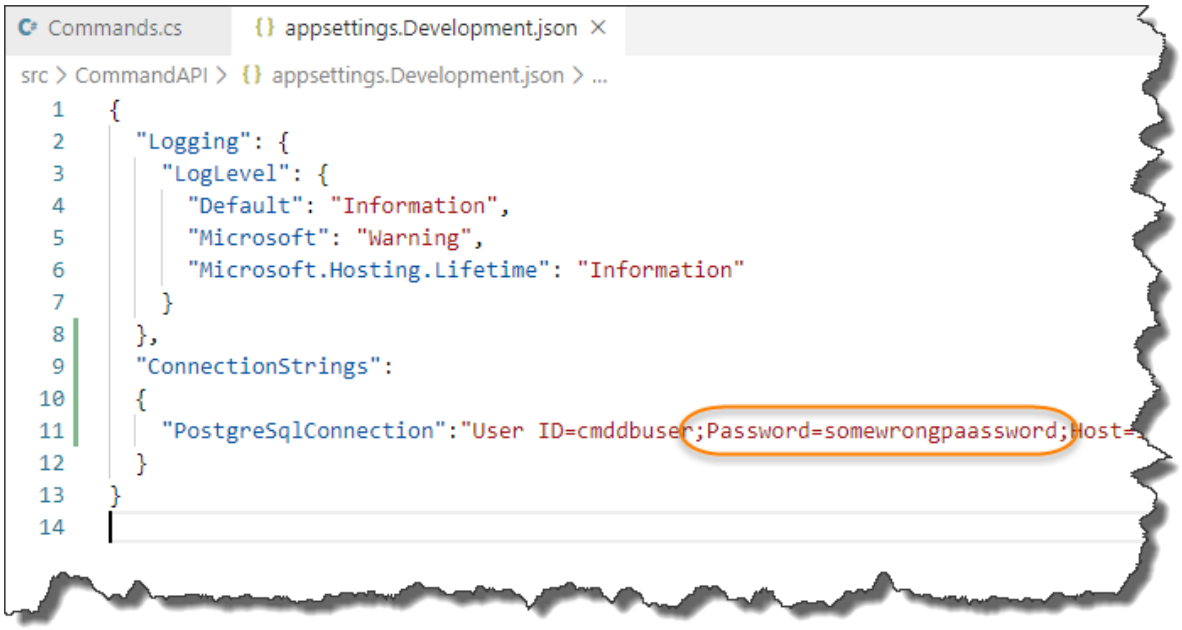
Again if you're unsure that your json is well-formed, use something like http://jsoneditoronline.org/ to check.

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

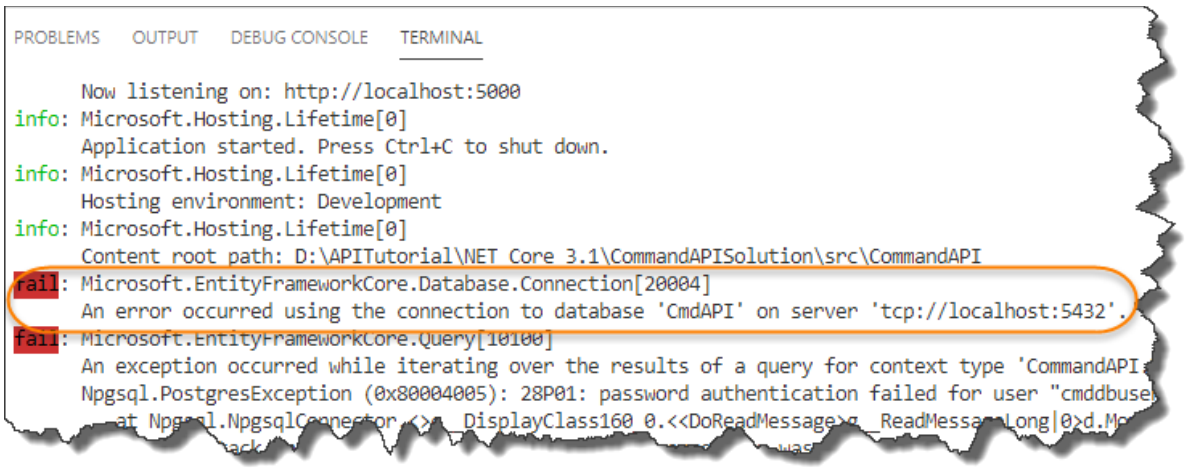
Ok so to prove the point we were making above.

- 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 PostgreSQL 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

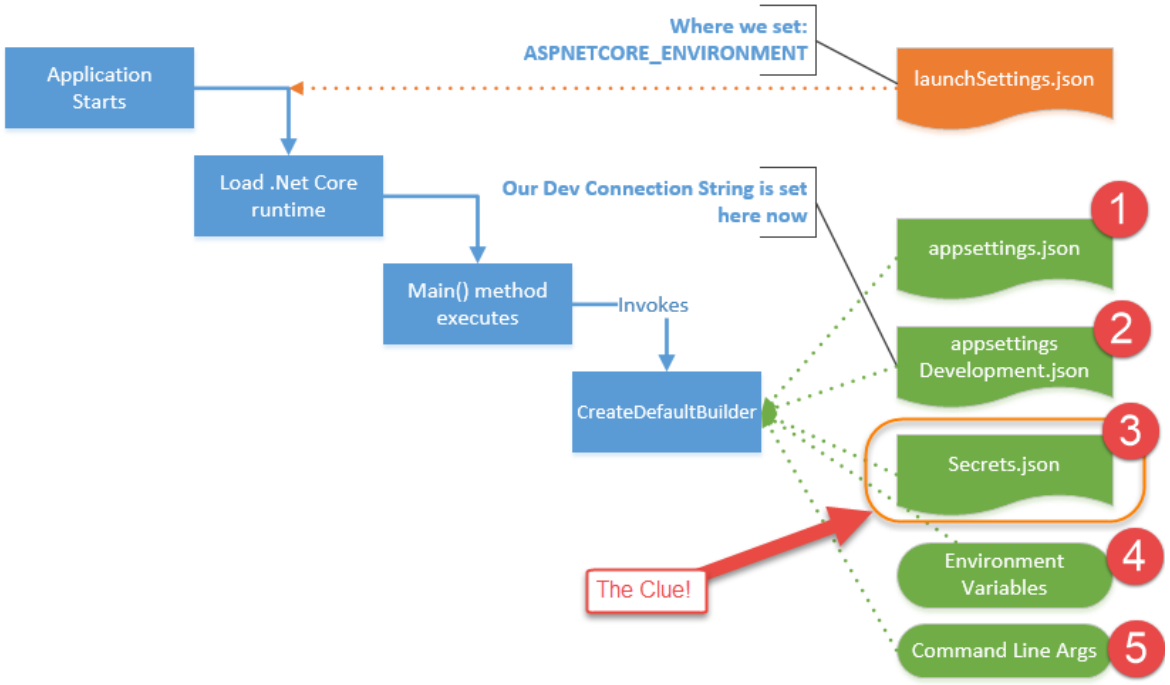
User Secrets

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.

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…

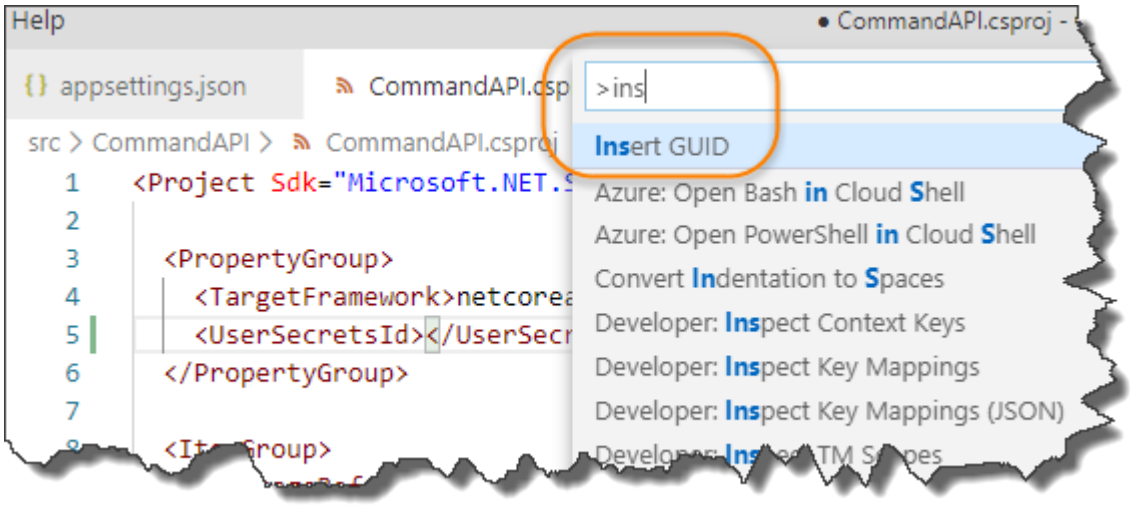
Learning Opportunity: Find out what GUID stands for and do a little bit of reading on what they are and where they can be used, (assuming you don't know this already!)

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!

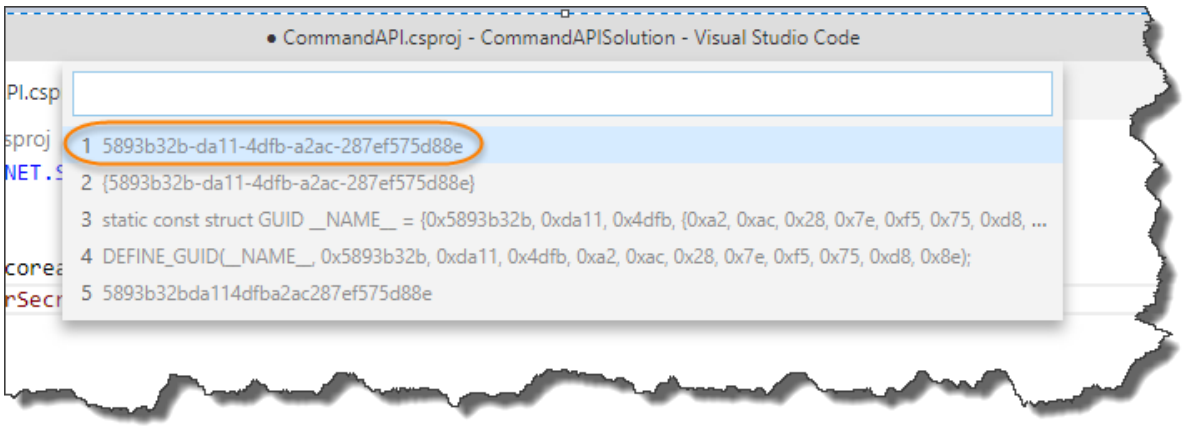
In VS Code open your ***CommandAPI.csproj*** file, and in the `<PropertyGroup>` xml element, place the xml highlighted below:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <UserSecretsId></UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.0.0" />
    .
    .
    .

</Project>
```

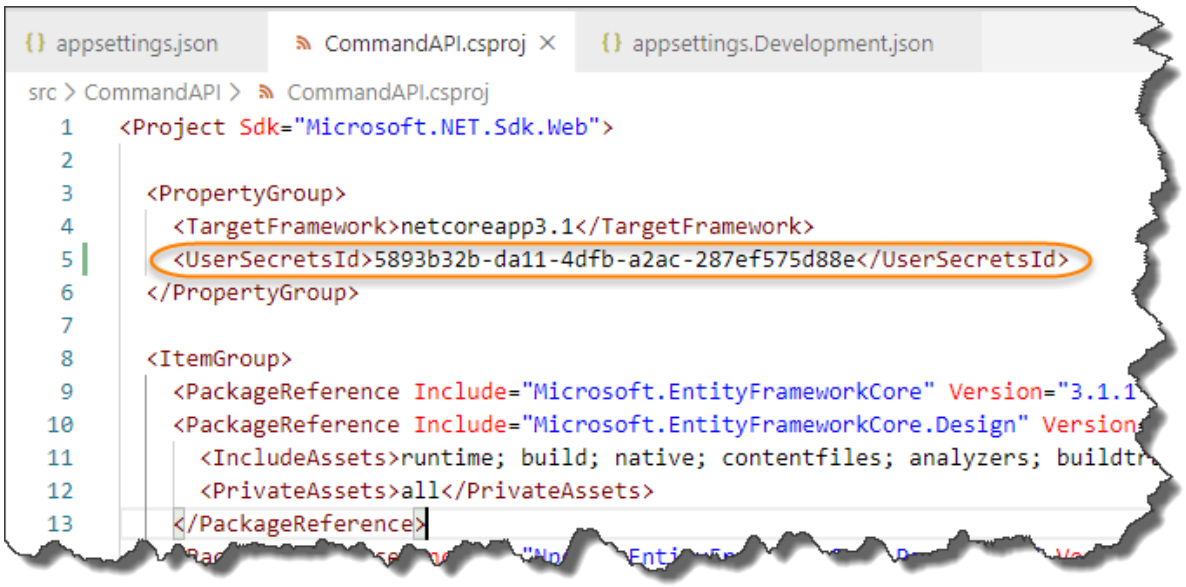
- Place your cursor in between the opening `<UserSecretsId>` and the closing `</UserSecretsId>` elements.
- Open the VS Code "Command Palette"
  - Hit: F1
  - Or: Ctrl + Shift + P
  - Or: View -> Command Palette…
- Type "Insert"



- 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

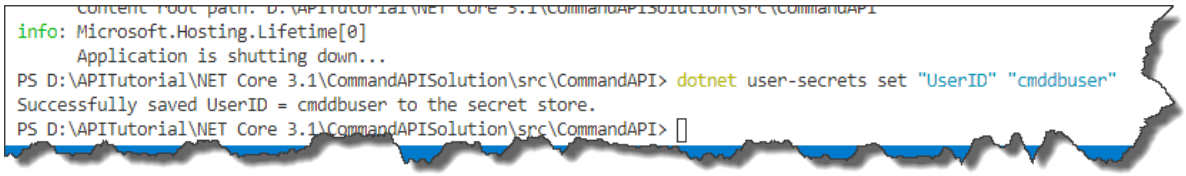
Either will work, but I'm going to go with Option 2 where we will store the individual components as "secrets".

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" "cmddbuser"
```

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



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

```
dotnet user-secrets set "Password" "pa55w0rd!"
```

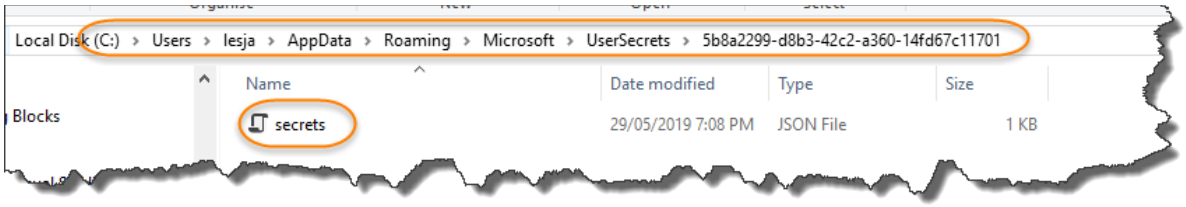
Again you should get a similar success message.

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[18]:



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

```
{
  "UserID": "cmddbuser",
  "Password": "pa55w0rd!"
}
```

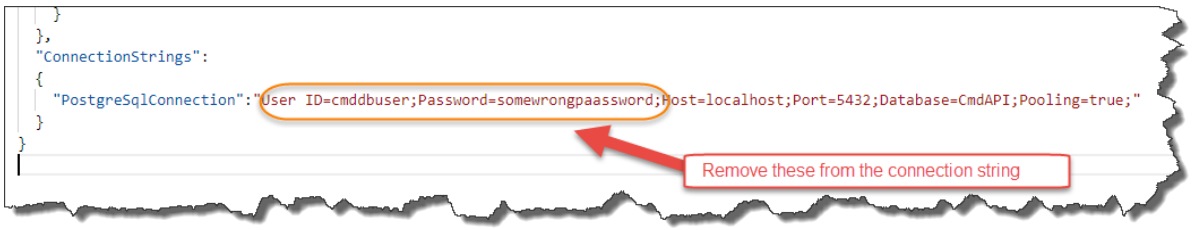
It's just simple, non-encrypted json.

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"
```

[18] On Windows you may need to ensure that you can see "Hidden items", there is a tick box on the View ribbon on Windows Explorer where you can set this.

```
    }
  },
  "ConnectionStrings":
  {
    "PostgreSqlConnection":
      "Host=localhost;Port=5432;Database=CmdAPI;Pooling=true;"
  }
}
```

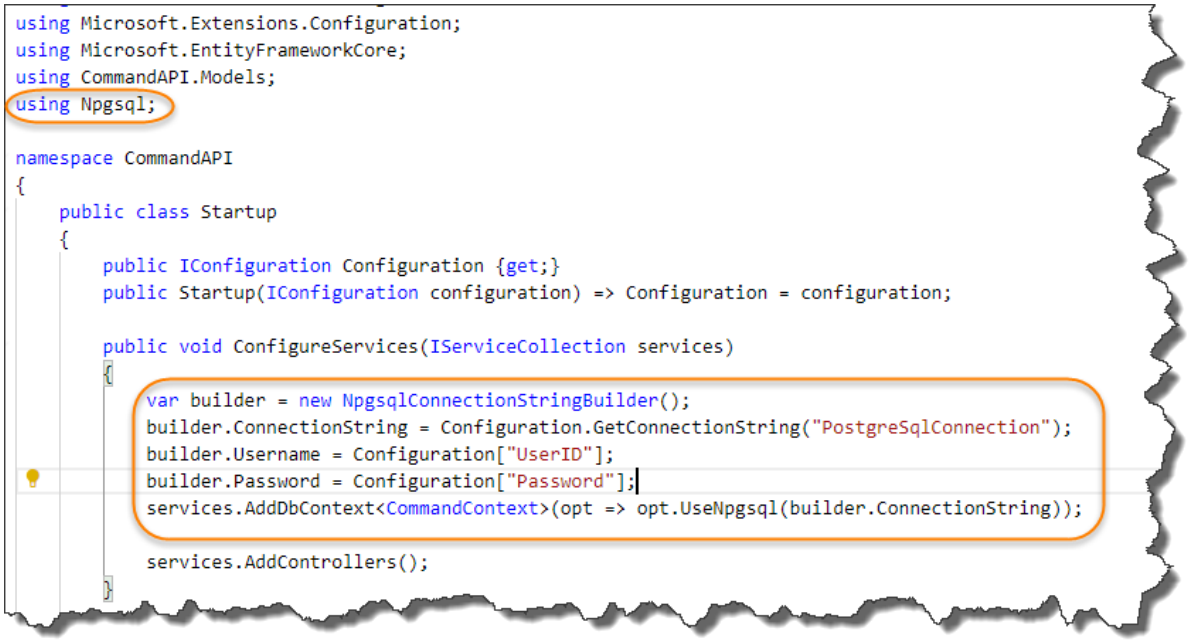
Make sure you save your file.

STEP 2: BUILD OUR CONNECTION STRING

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

```
.
.
.
using Npgsql;

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

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

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

            services.AddControllers();
        }
.
.
.
```

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



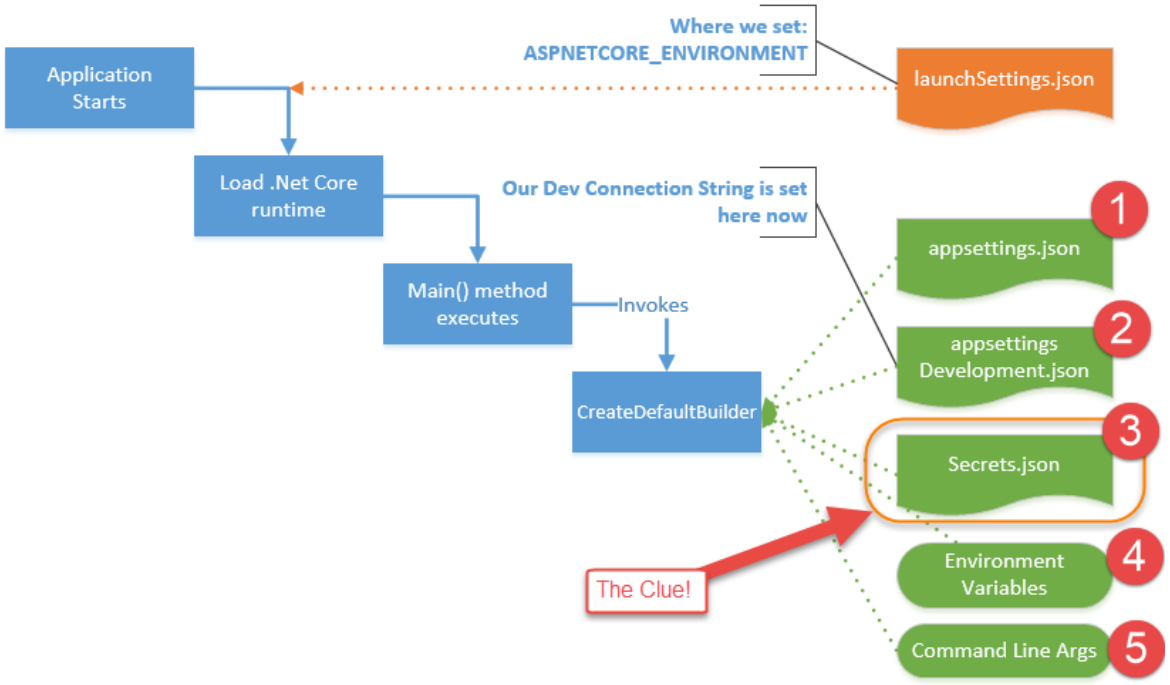
```
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using CommandAPI.Models;
using Npgsql;

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

        public void ConfigureServices(IServiceCollection services)
        {
            var builder = new NpgsqlConnectionStringBuilder();
            builder.ConnectionString = Configuration.GetConnectionString("PostgreSqlConnection");
            builder.Username = Configuration["UserID"];
            builder.Password = Configuration["Password"];
            services.AddDbContext<CommandContext>(opt => opt.UseNpgsql(builder.ConnectionString));

            services.AddControllers();
        }
```

1. We need to add a reference to `Npqsql` in order to use `NpgsqlConnectionStringBuilder`
2. This is where we:
    a. Create a `NpgsqlConnectionStringBuilder` object and pass in our "base" connection string: `PostgreSqlConnection` 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.

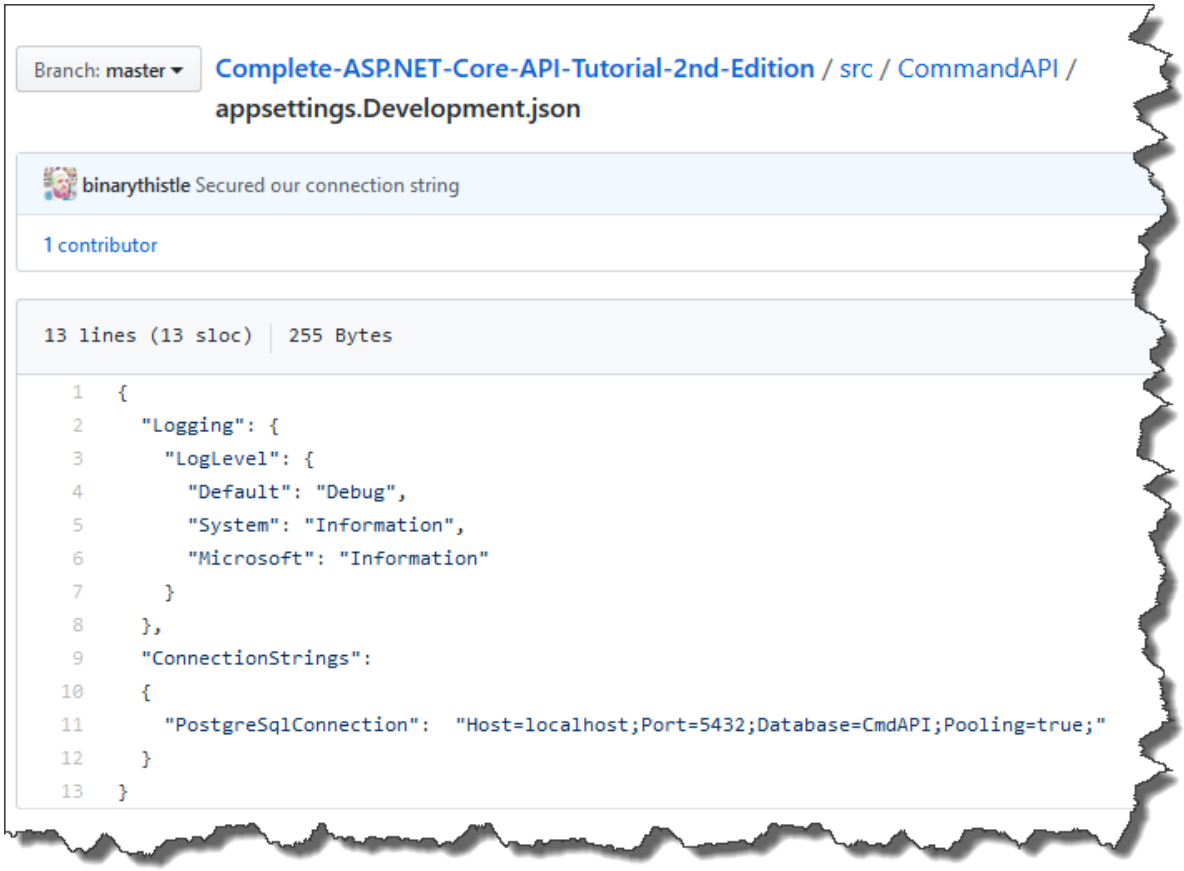
Wrap It Up

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 ▾ Complete-ASP.NET-Core-API-Tutorial-2nd-Edition / src / CommandAPI / appsettings.Development.json

binarythistle Secured our connection string

1 contributor

13 lines (13 sloc) | 255 Bytes

```
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Debug",
5        "System": "Information",
6        "Microsoft": "Information"
7      }
8    },
9    "ConnectionStrings":
10   {
11     "PostgreSqlConnection":  "Host=localhost;Port=5432;Database=CmdAPI;Pooling=true;"
12   }
13 }
```

Chapter 8 – Unit Testing & Completing Our API

Chapter Summary

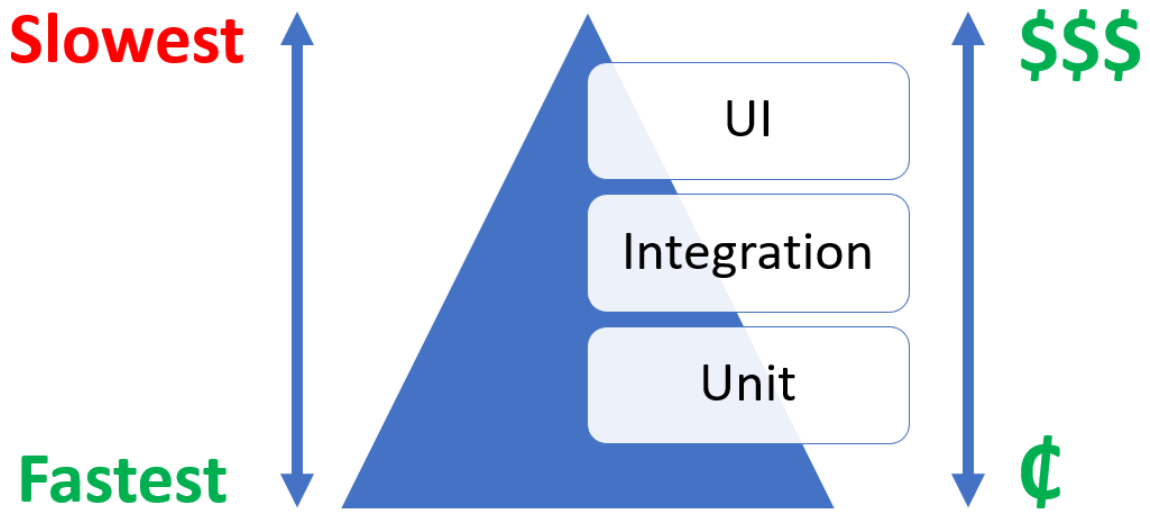
In this chapter we'll introduce you to Unit Testing, what it is, and why you'd use it. We'll then use unit testing to complete the development of our API, in an approach called *Test Driven Development*.

When Done, You Will

- Understand what Unit Testing is
- Write a Unit Test using *xUnit* to test our *existing* API functionality
- 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.

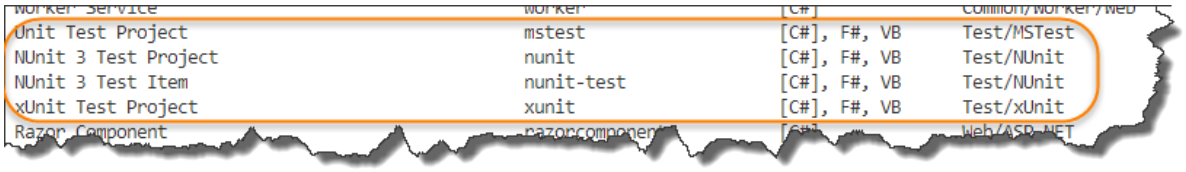
Some developers may only write a small number of unit tests that only test really novel code, others may write many more, that test more standard, trivial functionality… As our API is simple, we'll be writing tests that are pretty basic, and test quite obvious functionality. I've taken this approach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

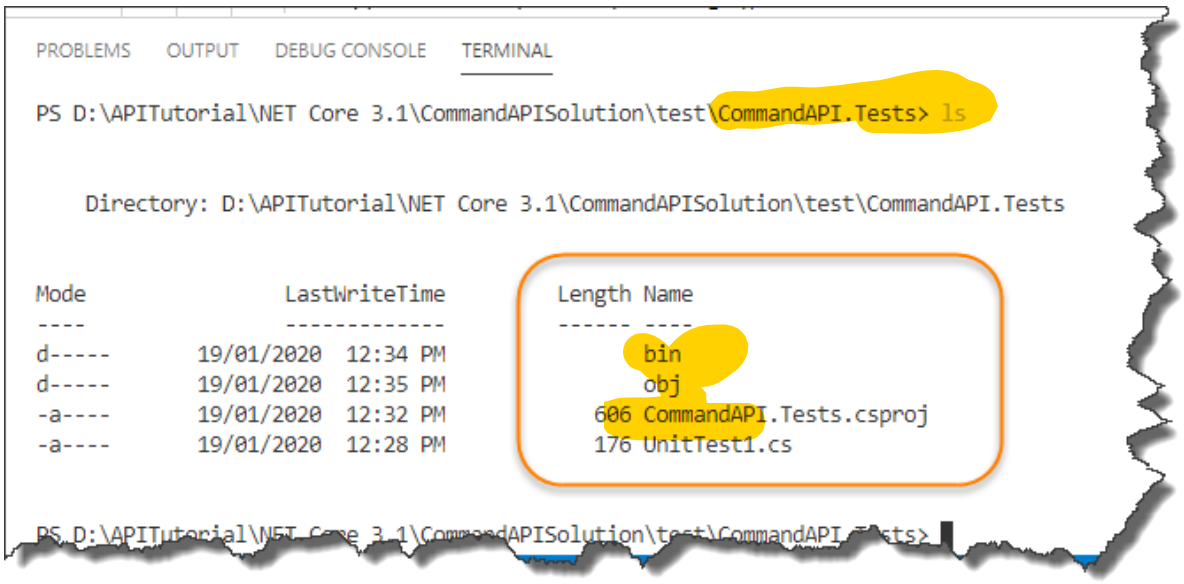
This is where you "check" the *actual result* against the *expected result*. Dependant on how that assertion goes, will depend on whether your test passes or fails.

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:

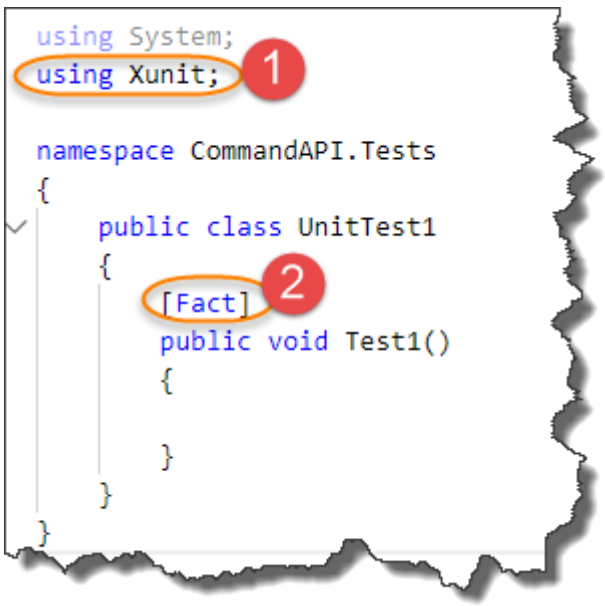


We have:

- ***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:



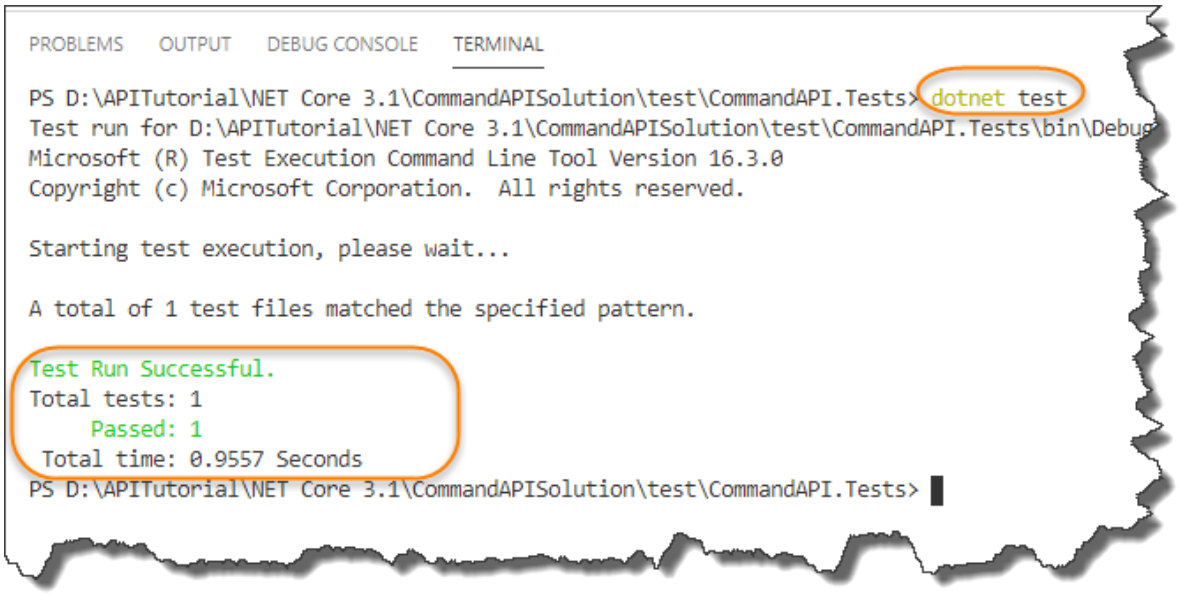
This is just a standard class definition, with only 2 points of note:

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.

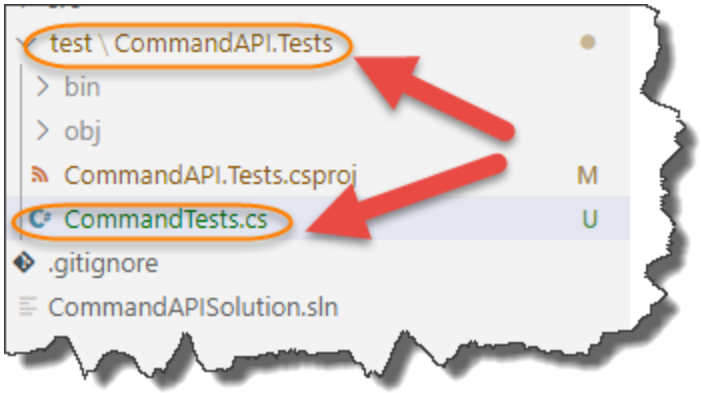
Thinking about our model what would we want to test? As a refresher here's the model class in our API project:

```
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;}
    }
}
```

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:

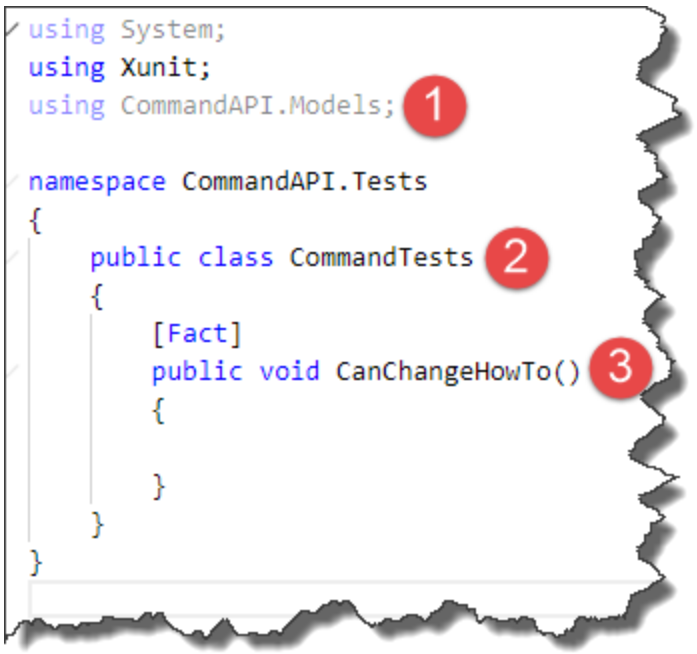
```
using System;
using Xunit;
using CommandAPI.Models;

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:

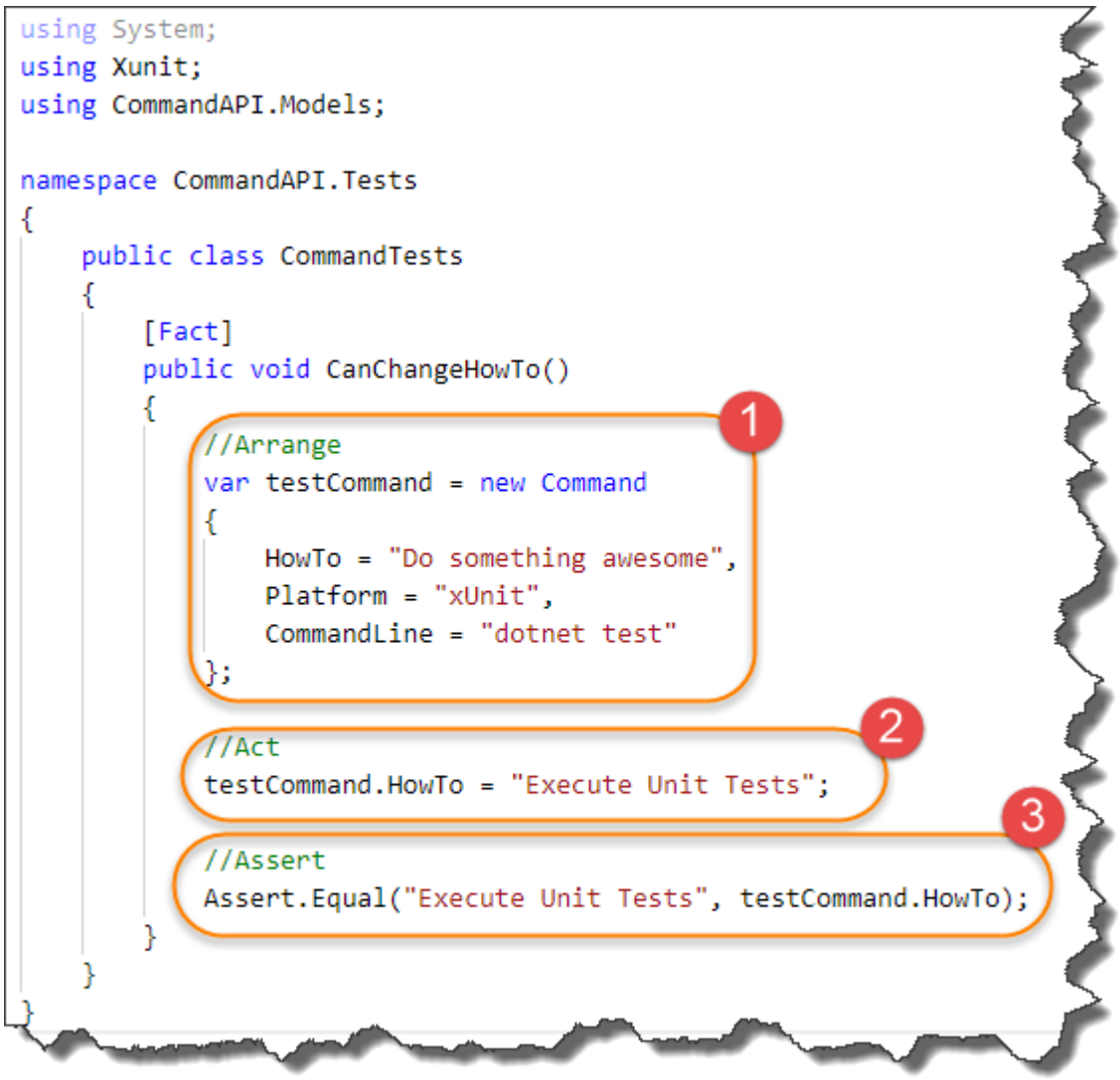


1. We have a reference to our Models in the ***CommandAPI*** project
2. Our Class is named after what we are testing (i.e. our `Command` model).
3. The naming convention of our test method is such that it tells use what the test is testing for.

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.

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:

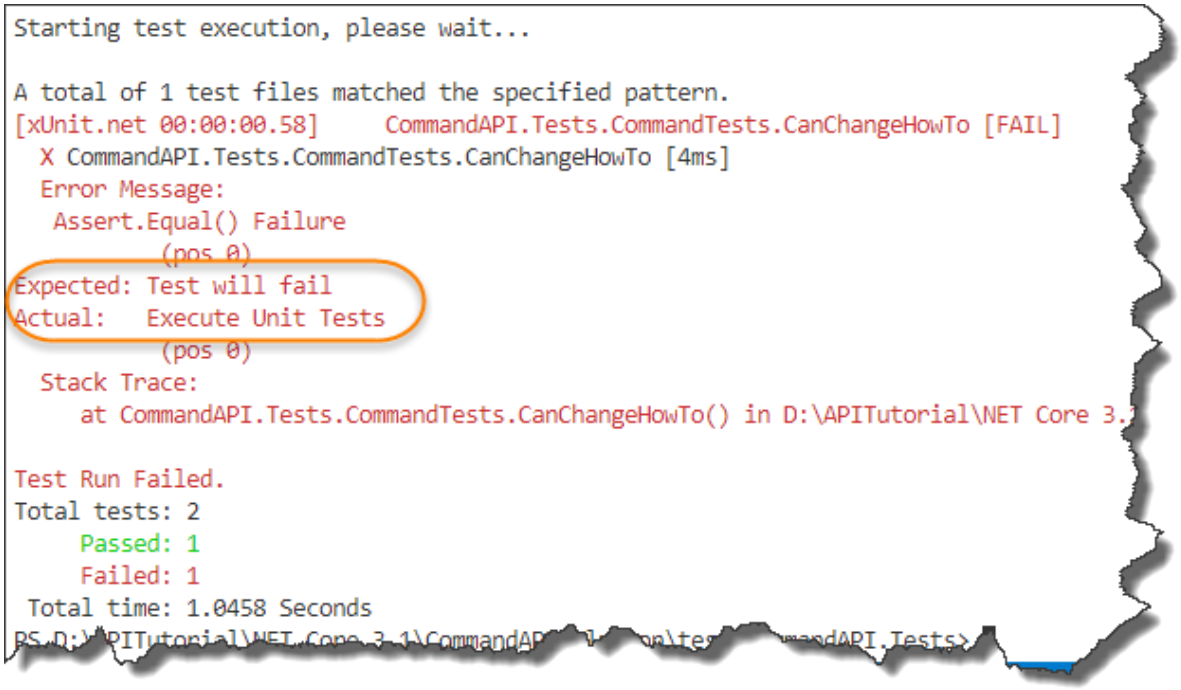
```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\test\CommandAPI.Tests> dotnet test
Test run for D:\APITutorial\NET Core 3.1\CommandAPISolution\test\CommandAPI.Tests\bin\Deb
Microsoft (R) Test Execution Command Line Tool Version 16.3.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...

A total of 1 test files matched the specified pattern.

Test Run Successful.
Total tests: 2
     Passed: 2
 Total time: 0.9995 Seconds
PS D:\APITutorial\NET Core 3.1\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:

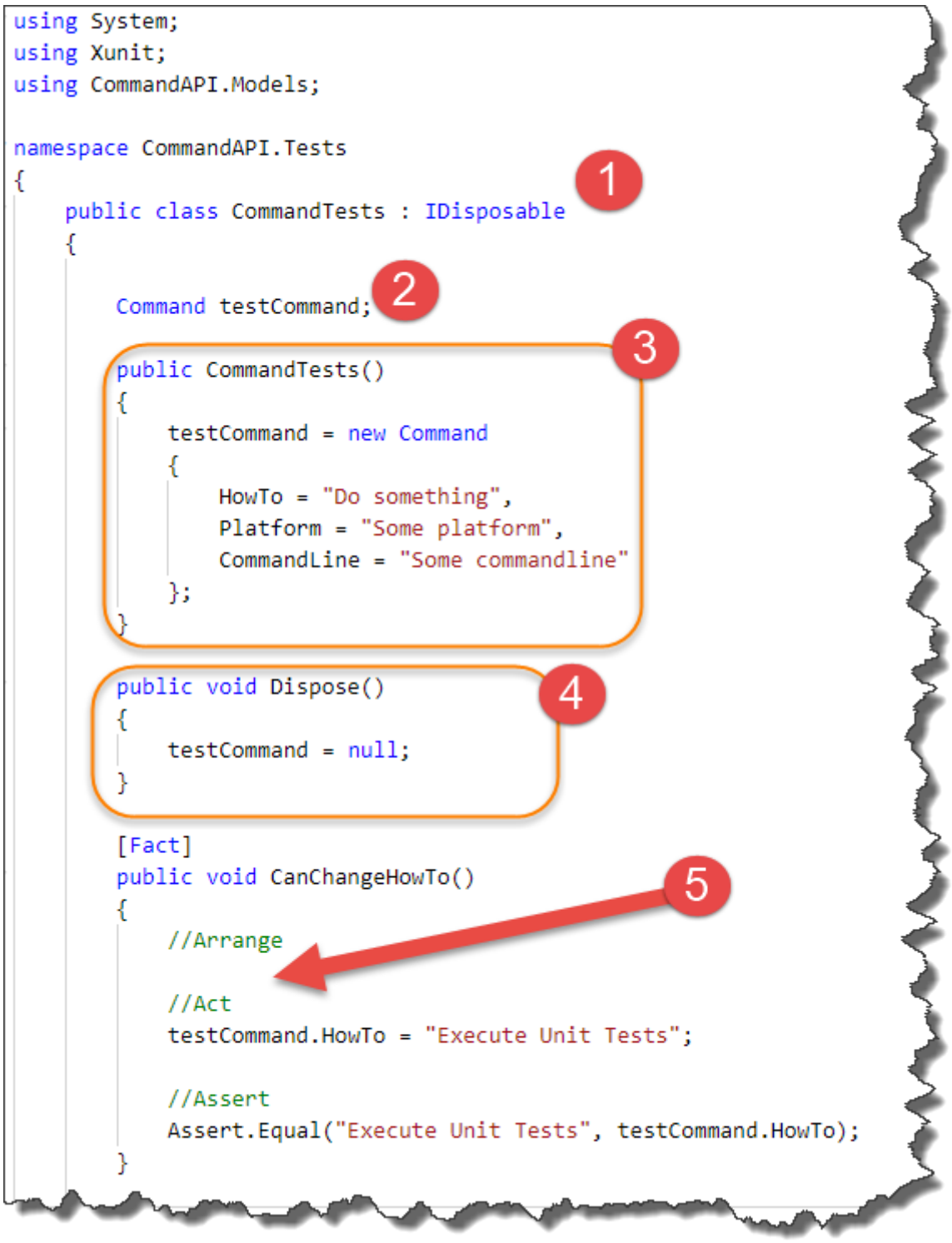
- Constructor and Dispose (shared setup/clean-up code without sharing object instances)
- Class Fixtures (shared object instance across tests in a *single class*)
- Collection Fixtures (shared object instances across multiple test classes)

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*[19] 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.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;
        }

    }
}
```

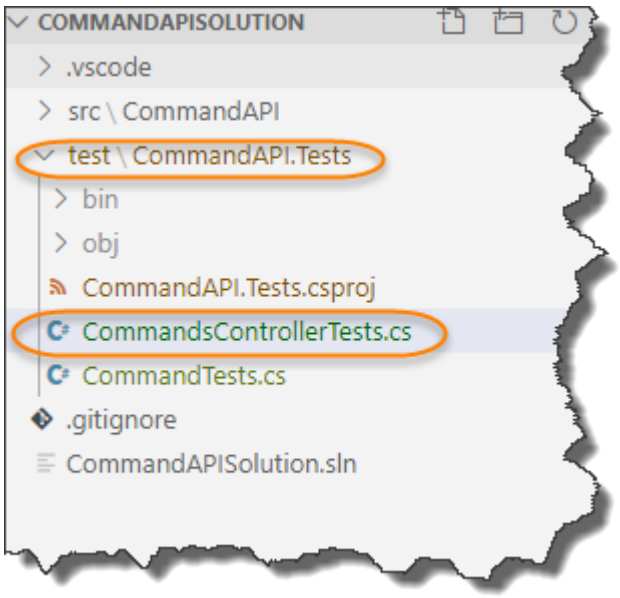
We had 1 `ActionResult, (GetCommandItems)`, that returned a list of `Command` objects as a serialized JSON string:. So, we want to test that method via unit testing. So what sort of things should we be testing for? How about:

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"

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:

[19] 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.



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.

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:

```
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;
        }

    }
}
```

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 a `DBContext` so 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`.

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 PostgreSQL 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 DBContext class implementation and use it with an "In Memory Database", thus removing the external dependency on a PostgreSQL Server.

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

To make use of the In Memory Database we need to reference the `Microsoft.EntityFrameworkCore.InMemory` Nuget package, to do type the following at a command line:

```
dotnet add package Microsoft.EntityFrameworkCore.InMemory
```

This will add a package reference to out ***CommandAPITests.csproj*** file:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>

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

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.InMemory" Version="3.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Relational" Version="3.1.1" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.2.0" />
    <PackageReference Include="xunit" Version="2.4.0" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.0" />
    <PackageReference Include="coverlet.collector" Version="1.0.1" />
  </ItemGroup>

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

</Project>
```

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:

```
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);
    }
}
```

1. We create a `DBContextOptionsBuilder` with our `CommandContext` class
2. 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)
3. 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).

```
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\test\CommandAPI.Tests> dotnet test
Test run for D:\APITutorial\NET Core 3.1\CommandAPISolution\test\CommandAPI.Tests\bin\Debug\netcor
Microsoft (R) Test Execution Command Line Tool Version 16.3.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...

A total of 1 test files matched the specified pattern.

Test Run Successful.
Total tests: 4
     Passed: 4
 Total time: 0.9986 Seconds
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\test\CommandAPI.Tests>
```

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);


  //Controller
  var controller = new CommandsController(dbContext);
}
```

```
.
.
.
```

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);


  //Controller
  var controller = new CommandsController(dbContext);

  //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 and you should still have 4 passing tests!

Finishing GetCommandItems Tests

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

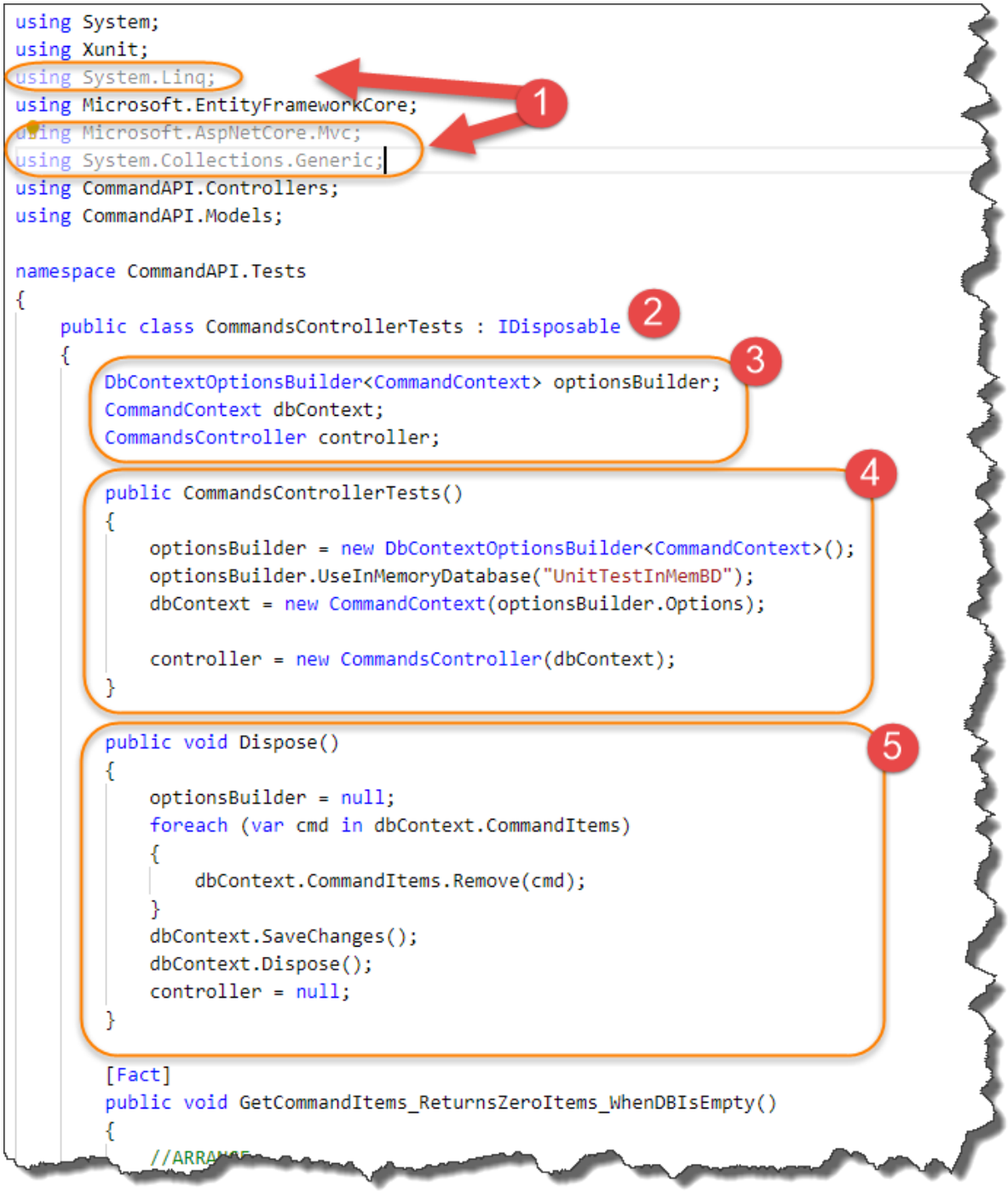
Test ID	Condition	Expected Result
~~**Test 1.1**~~	~~Request Objects when none exist~~	~~Return "nothing"~~
Test 1.2	Request Objects when 1 exists	Return Single Object
Test 1.3	Request Objects when *n* exist	Return Count of *n* Objects
Test 1.4	Request Objects	Return the correct "type"

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;
```

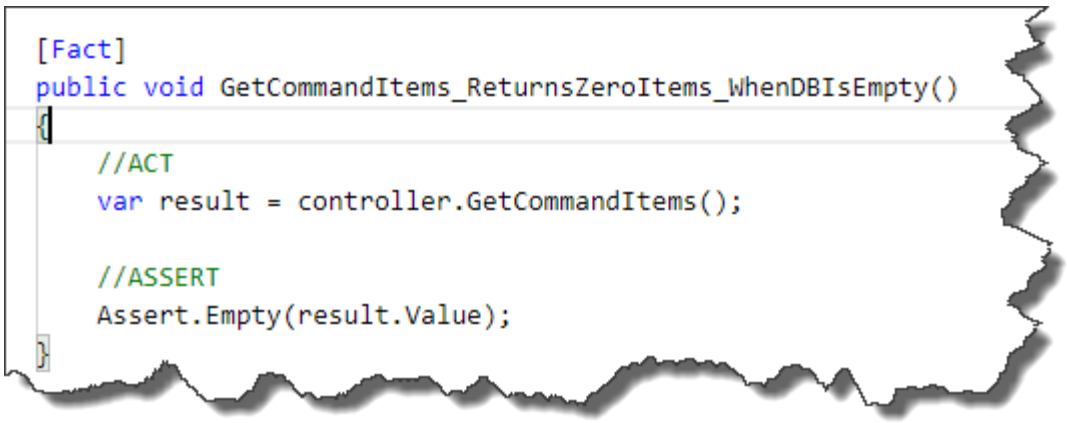
```
using Xunit;
using Microsoft.EntityFrameworkCore;
using CommandAPI.Controllers;
using CommandAPI.Models;

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

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

      controller = new CommandsController(dbContext);
    }

    public void Dispose()
    {
      optionsBuilder = null;
      foreach (var cmd in dbContext.CommandItems)
      {
        dbContext.CommandItems.Remove(cmd);
      }
      dbContext.SaveChanges();
      dbContext.Dispose();
      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. Add several new using directives that will be needed for out tests later.
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]
public void GetCommandItems_ReturnsZeroItems_WhenDBIsEmpty()
{
    //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, (and make use of "`Count`"), which is why we need to reference Linq.

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);
}
```

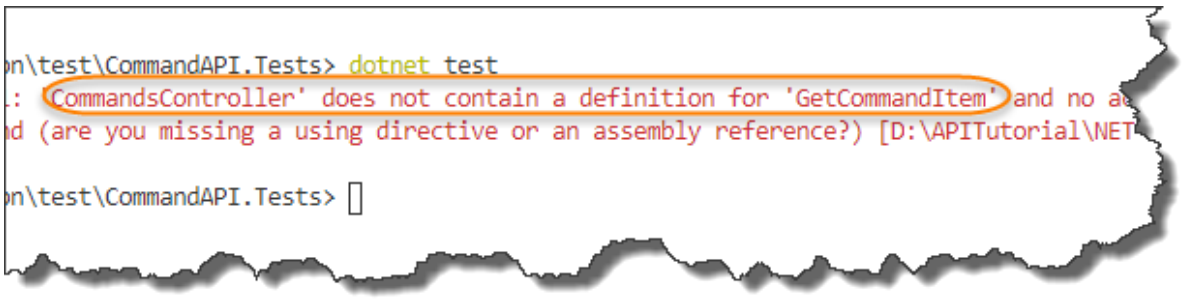
You can run all your tests again, or you can run selective tests as so:

```
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, (in our API Project), 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!

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\test\CommandAPI.Tests> dotnet test
Test run for D:\APITutorial\NET Core 3.1\CommandAPISolution\test\CommandAPI.Tests\bin\Debug\ne
Microsoft (R) Test Execution Command Line Tool Version 16.3.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...

A total of 1 test files matched the specified pattern.

Test Run Successful.
Total tests: 8
     Passed: 8
 Total time: 1.3392 Seconds
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\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...

A total of 1 test files matched the specified pattern.
[xUnit.net 00:00:00.91]     CommandAPI.Tests.CommandsControllerTests.GetComm
  X CommandAPI.Tests.CommandsControllerTests.GetCommandItemReturns404NotFoun
  Error Message:
   Assert.IsType() Failure
Expected: Microsoft.AspNetCore.Mvc.NotFoundResult
Actual:   (null)
  Stack Trace:
     at CommandAPI.Tests.CommandsControllerTests.GetCommandItemReturns404Ne
:line 144
```

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, (make sure you add the reference to `Microsoft.EntityFrameworkCore`:

```
using Microsoft.EntityFrameworkCore;
.
.
.


//PUT:      api/commands/{Id}
[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 5.1 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 5.2 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 5.3 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 5.4 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:

```
//DELETE:     api/commands/{Id}
[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[20]), 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.

[20] 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?

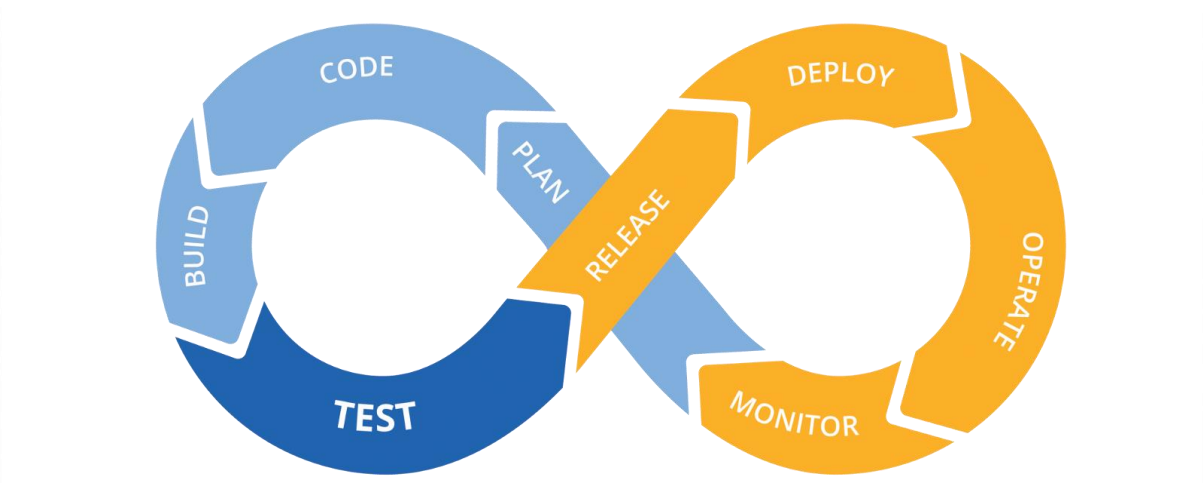
Typically, when we talk about CI/CD we talk about Continuous Integration & Continuous Delivery, although it can be dependent on the organization. Ultimately the decision to deploy software into production is a business decision, so the idea of Continuous Deployment is still overwhelming for most organizations….

In this book though we're going to go all out and practice full-on Continuous Deployment!

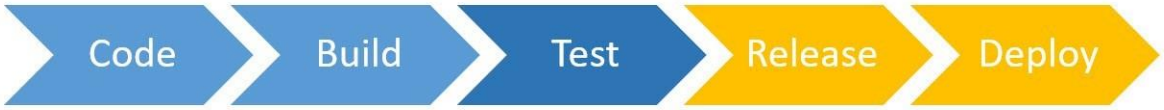
The Pipeline

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

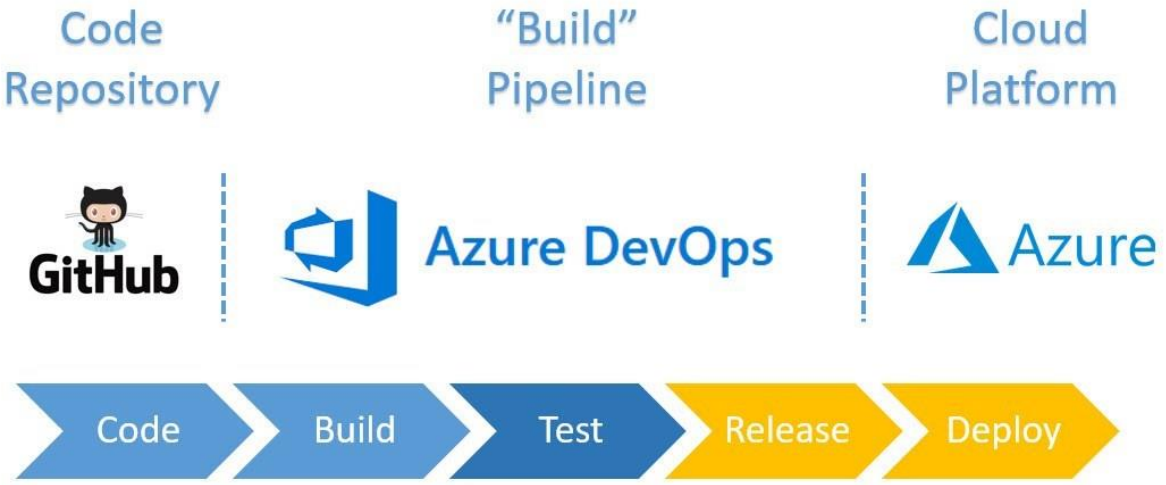
There are various on-premise and cloud-based alternatives: Jenkins is possibly the most "famous" of the on-premise solutions available, but you also have things like:

- 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.

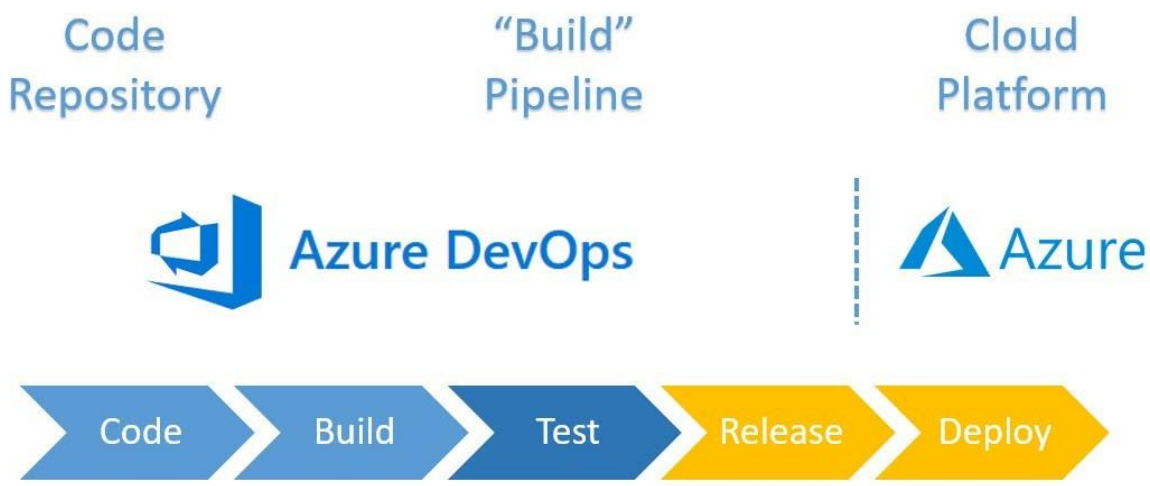
Technology In Context

Referring to our pipeline, in-terms of our technology overlay this is what we will be working with to build a CI/CD pipeline:

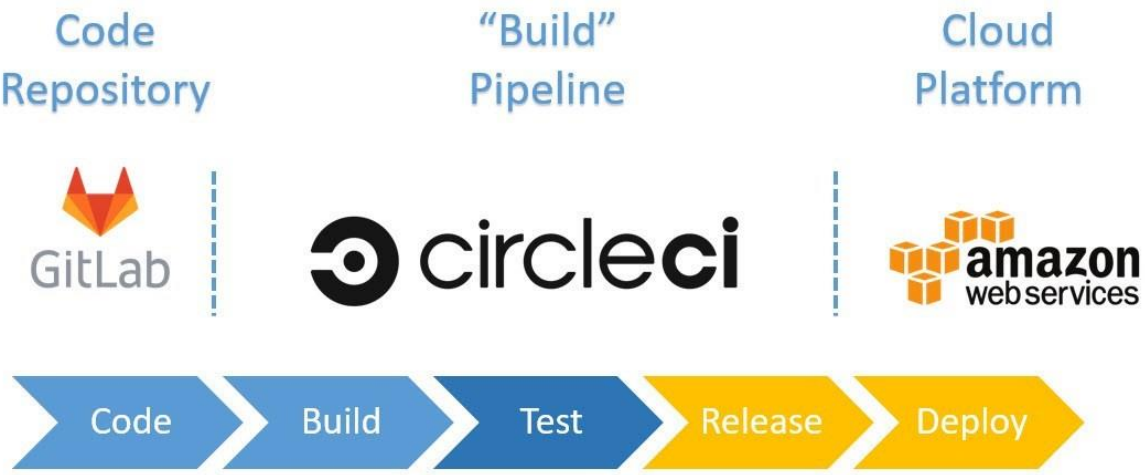


Indeed, Azure DevOps comes with its own "code repository" feature, (Azure Repos), which means we could do away with GitHub…

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.

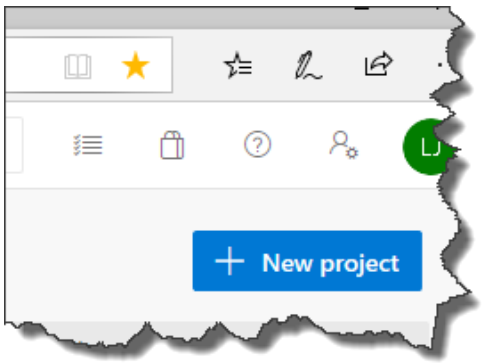
The takeaway points I wanted to make were:

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

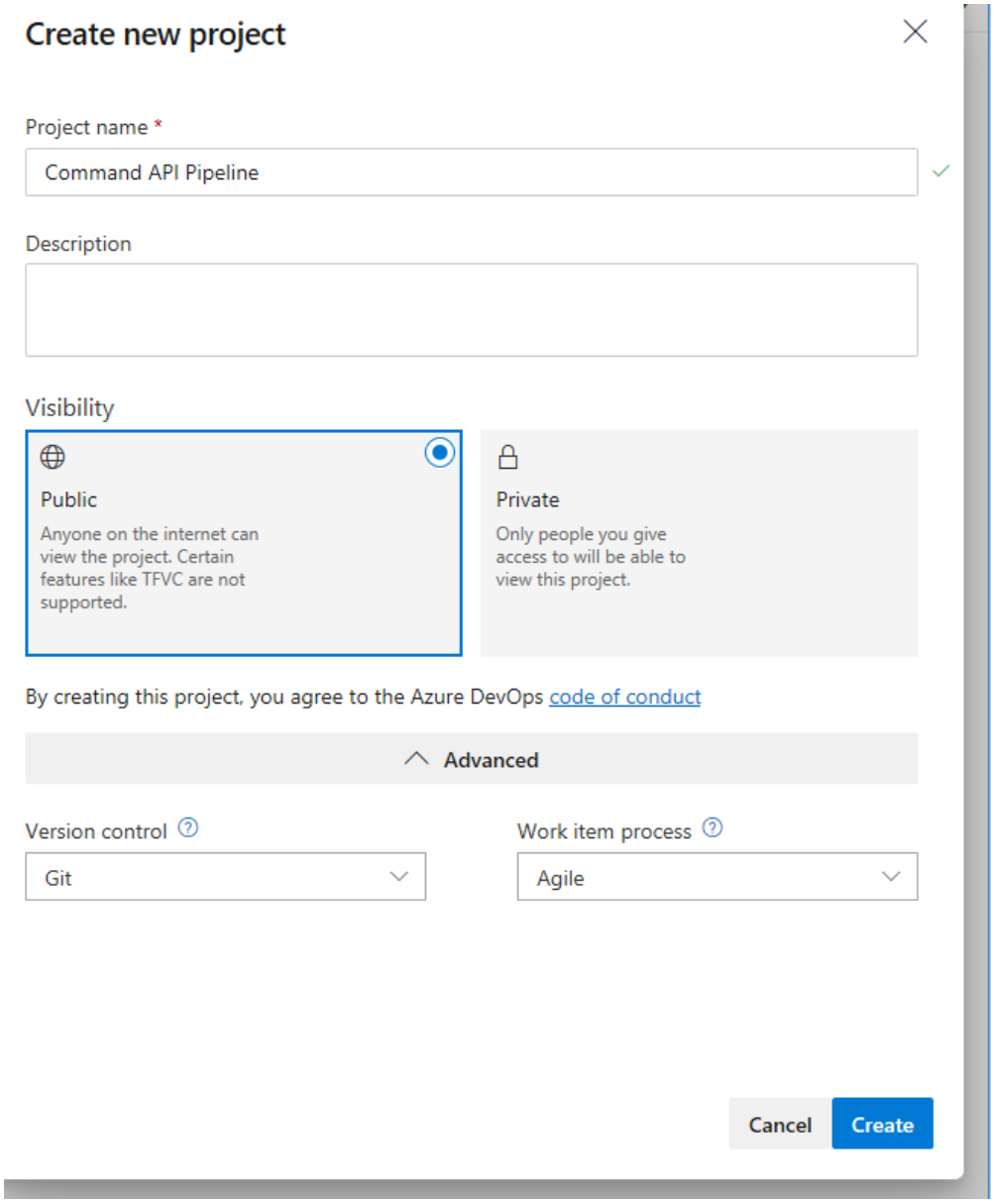
Ok enough theory, let's build our pipeline!

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 "New Project":



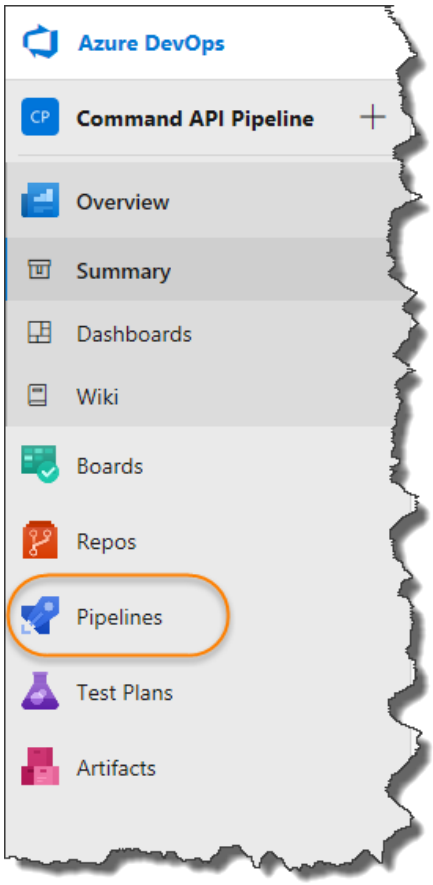
You can call it anything you like, so let's keep the theme going and call it *Command API Pipeline*:



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:

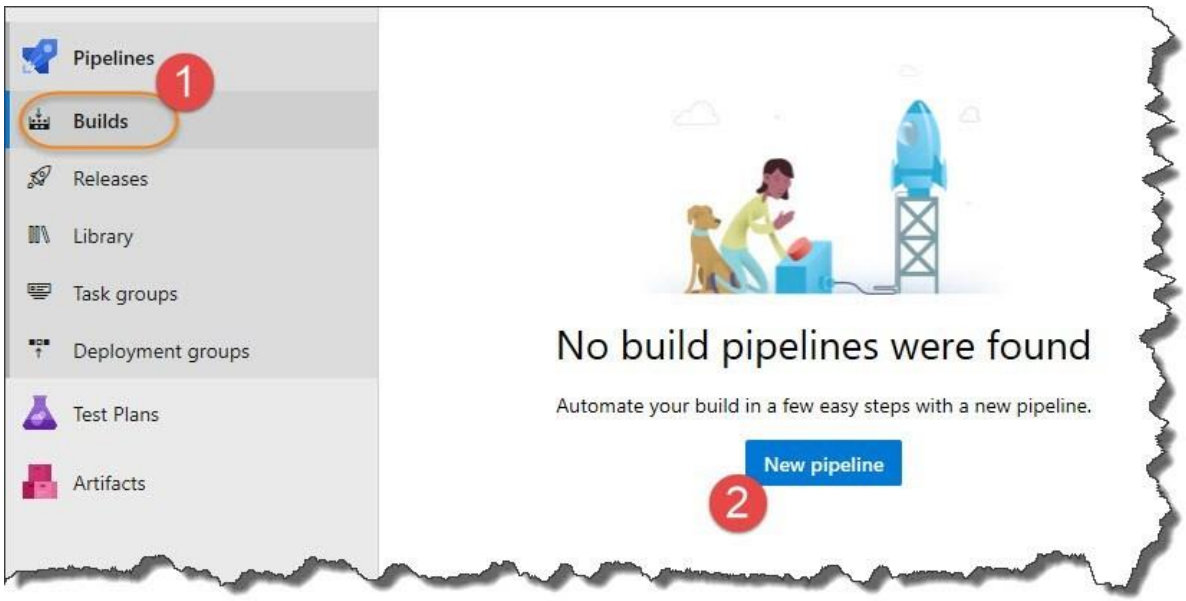


Les's Personal Anecdote: Followers of my Blog or YouTube channel will realise that the core of this chapter has been taken from a blog post & video I did on Azure DevOps. One of the things I noticed while making those was that the Azure DevOps product changes quite rapidly.

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, (~Febuary 2020), 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:

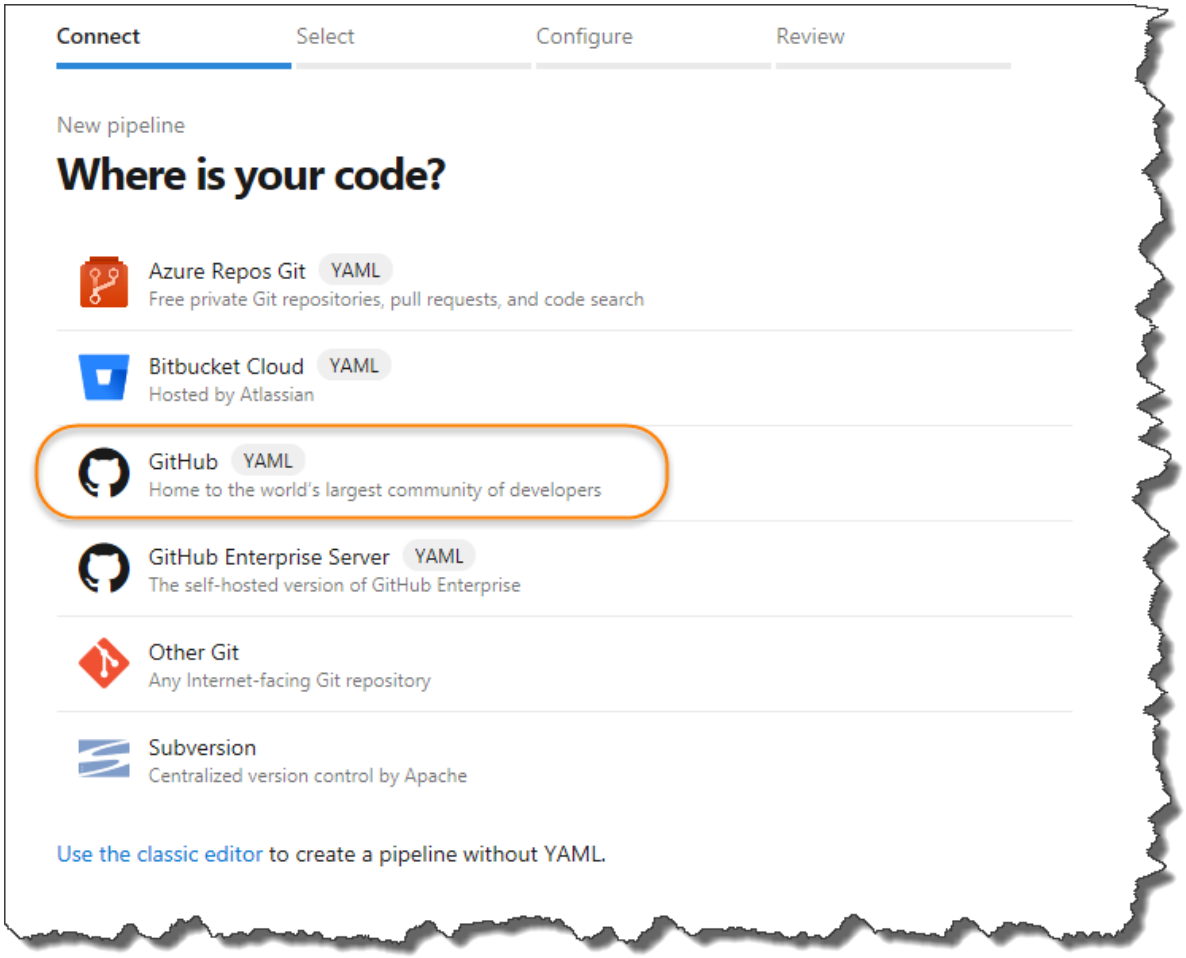
1. Builds
2. New pipeline:



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

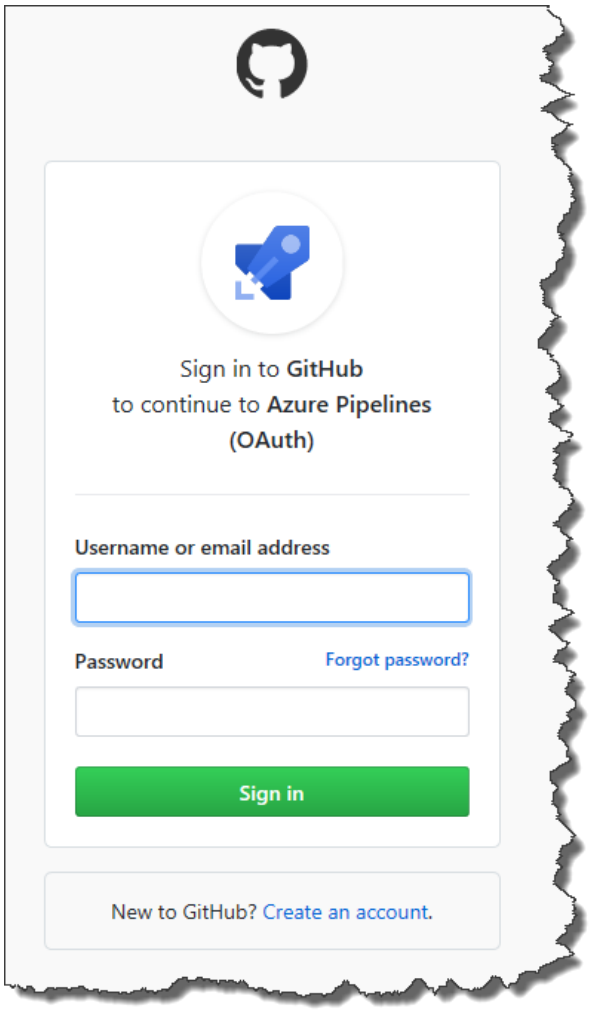
Well, where do you think?

Yeah – that’s right – in GitHub!

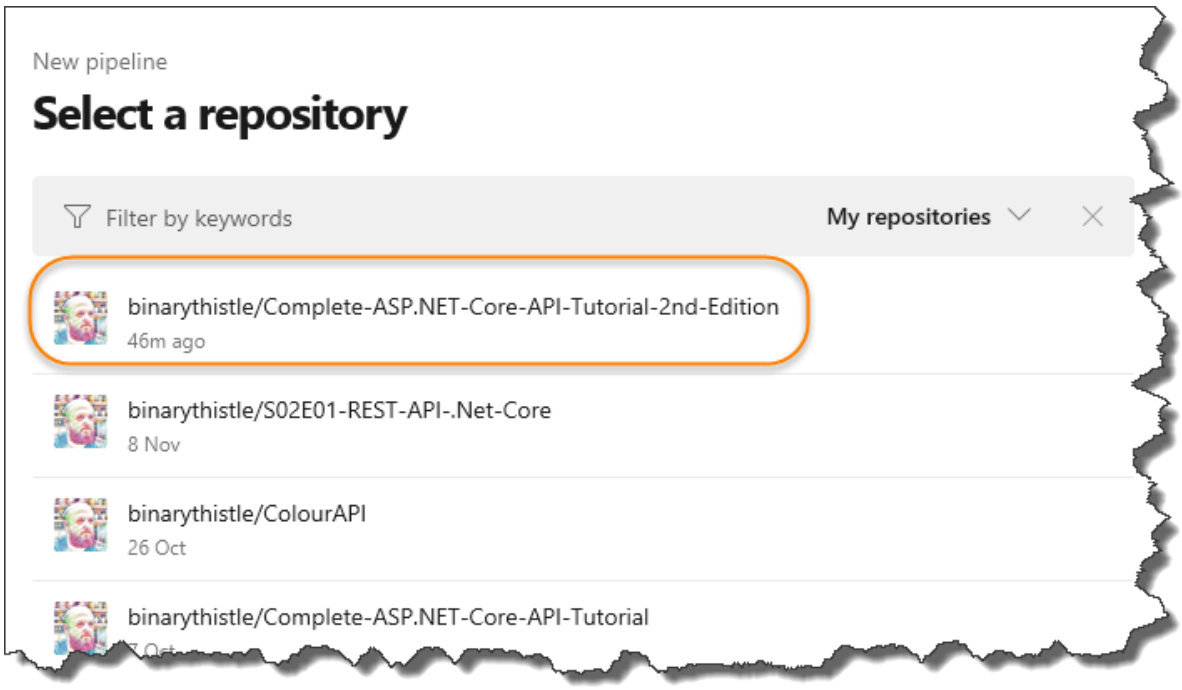


Be careful to select GitHub, as opposed to GitHub Enterprise Server, (which as the description states is the on-premise version of GitHub).

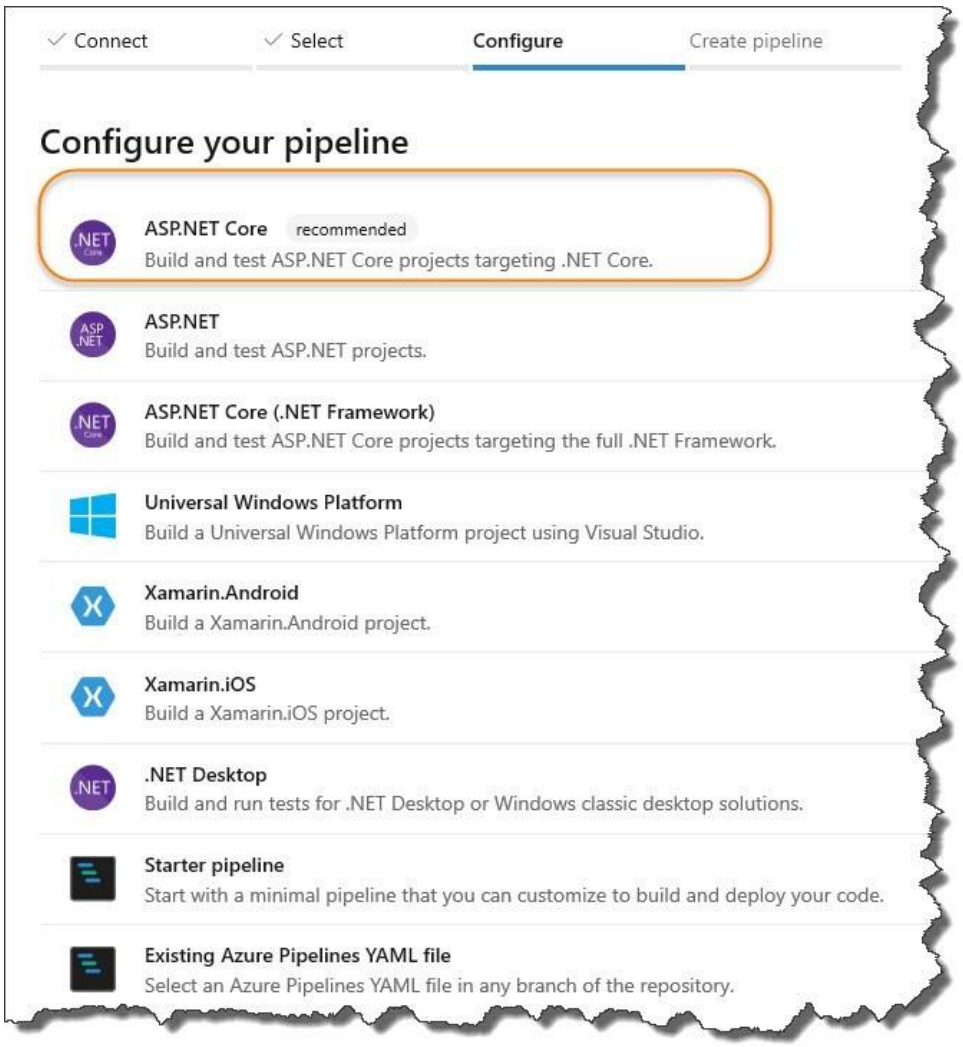
IMPORTANT: If this is the 1st time you're doing this, you'll need to give Azure DevOps permission to view your GitHub account:



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 "Complete-ASP.NET-Core-API-Tutorial-2nd-Edition"), 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)'

```

The ***azure-pipelines.yml*** file is the central artefact you need to configure with Azure Devops, and straight away we'll need to make a quick edit…

UPDATE AZURE-PIPELINES.YML TO USE .NET CORE 3.1

Previously I would just tell you to "run" the pipeline you've just created, but at the time of writing, (Feb 2020), Azure DevOps doesn't support version 3.1 of the .NET Core framework out the box, and you will receive an error when you execute the pipeline. We therefore need to make our first edit to the ***azure-pipelines.yml*** file, (above).

Les's Personal Anecdote: Not so much an anecdote, but an opinion… This is an example of some of the "edge-cases" you'll get when using the latest version of any framework. It's one of the reasons I tend to hold back on adopting the "bleeding-edge"

FYI it's entirely possible that this particular "issue" will be resolved when you come to read the book.

So we're going to insert some instructions to specify the .NET Core framework version here:

We can edit the file directly in the browser, so add the following config, (in bold), to the file:

```
.
.
steps:
- task: UseDotNet@2
  displayName: ".NET Core 3.1.x"
  inputs:
    version: '3.1.x'
    packageType: sdk
- script: dotnet build --configuration $(buildConfiguration)
  displayName: 'dotnet build $(buildConfiguration)'
```

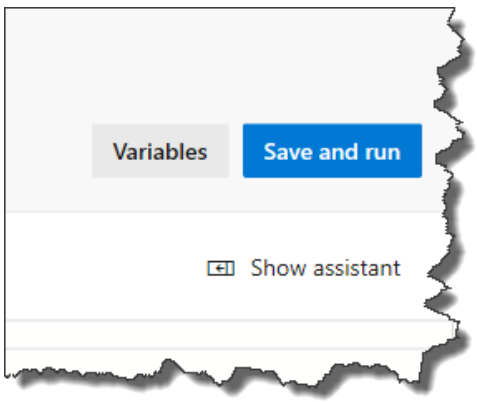
Warning! YAML files are white-case sensitive, so you need to ensure the indentation is absolutely spot on! Thankfully the in browser editor will complain if you've not indented correctly.

Your edit YAML file should look like this:

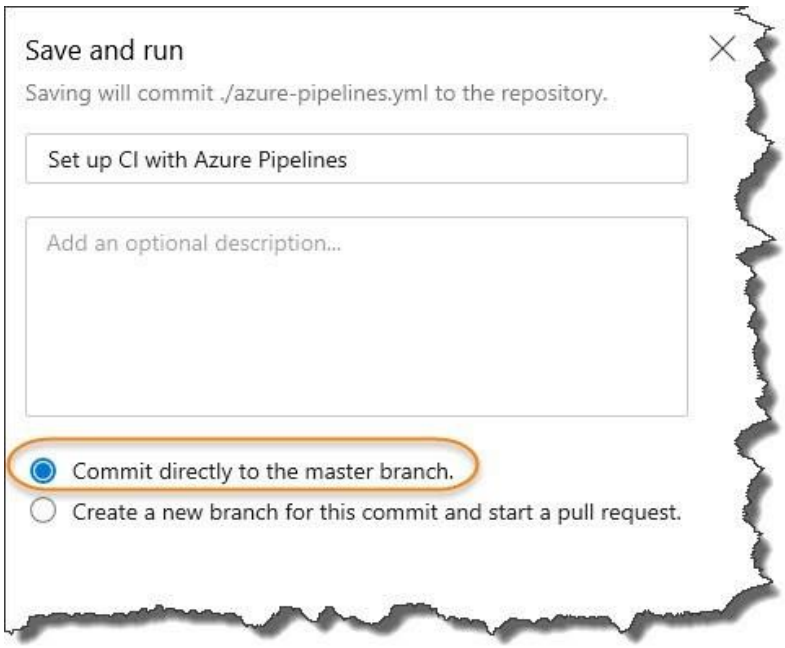
```yaml
# ASP.NET Core
# Build and test ASP.NET Core projects targeting .NET Core.
# Add steps that run tests, create a NuGet package, deploy, and more:
# https://docs.microsoft.com/azure/devops/pipelines/languages/dotnet-core

trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

variables:
  buildConfiguration: 'Release'

steps:
Settings
- task: UseDotNet@2
  displayName: ".NET Core 3.1.x"
  inputs:
    version: '3.1.x'
    packageType: sdk
- script: dotnet build --configuration $(buildConfiguration)
  displayName: 'dotnet build $(buildConfiguration)'
```

Indeed if you look closely, you'll see that the in-browser editor displays periods, ('.'), to denote the white space indentation.

We're now ready to Click Save & Run:

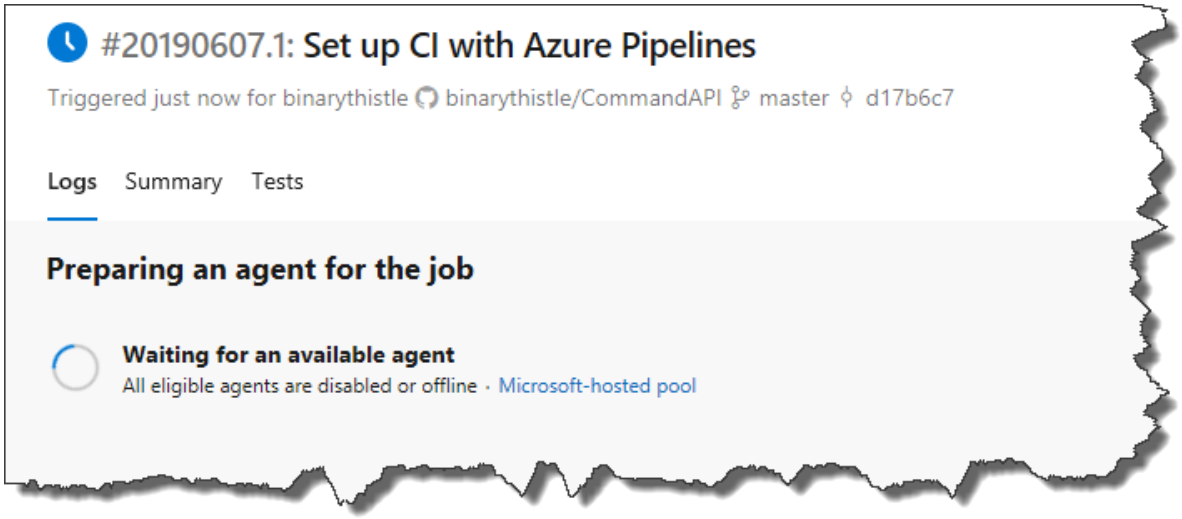


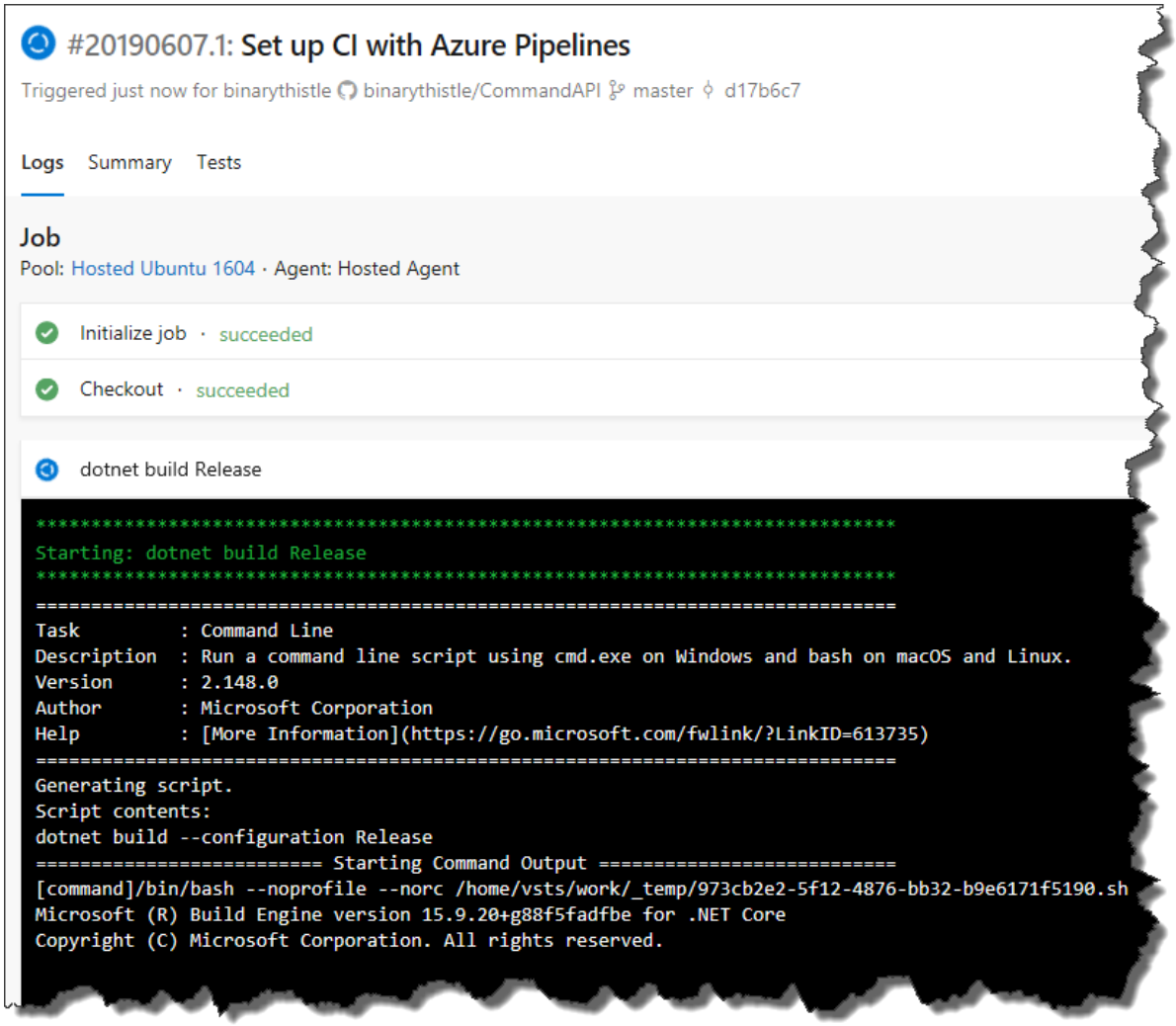
You'll then be presented with:



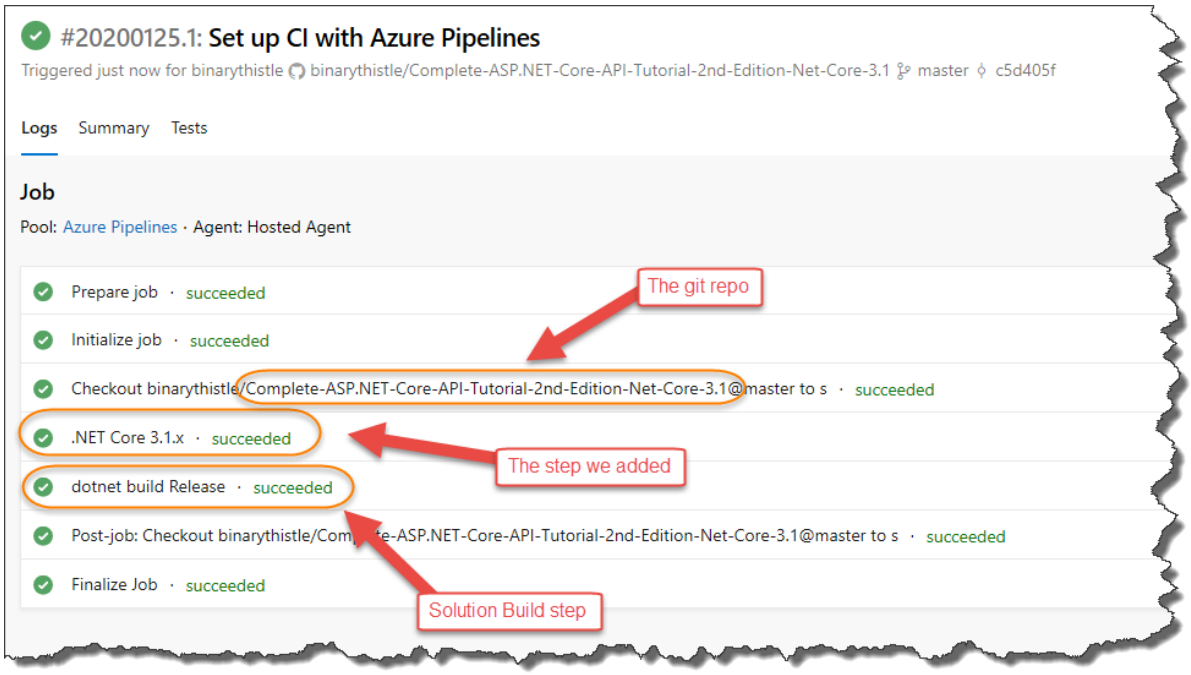
This is asking you where you want to store the ***azure-pipelines.yml*** file, in this case we want to add it directly to our GitHub repo, (remember this selection though as it comes back later!), so select this option and click **Save & Run.**

An "agent" is then assigned to execute the pipeline, you'll see various screens, such as:





And finally you should see the completion screen:



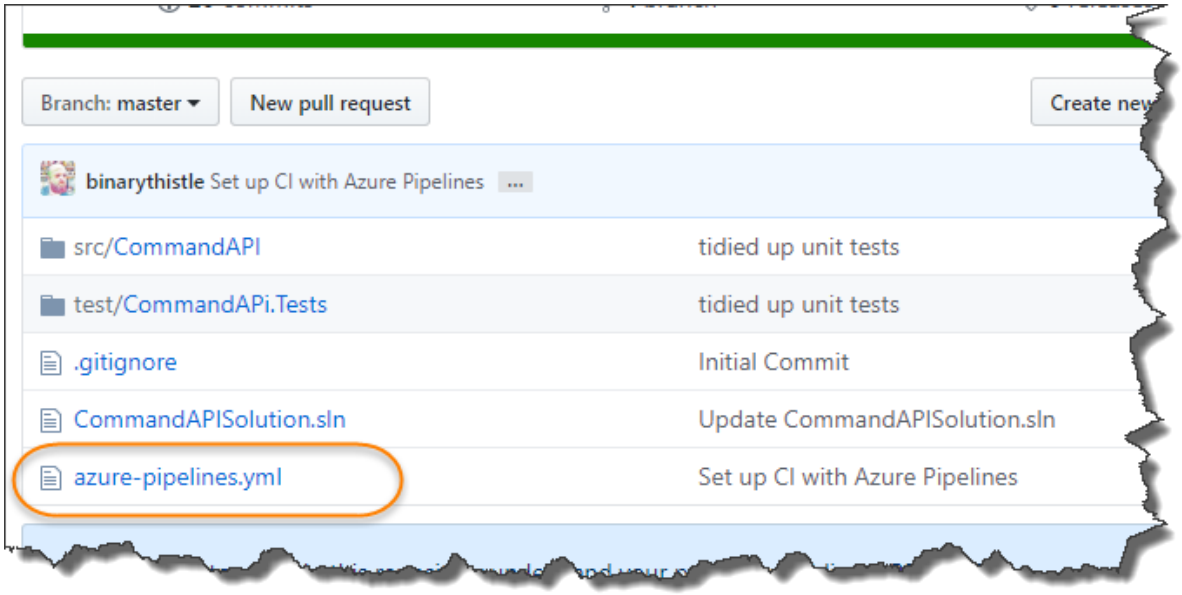
What Just Happened?

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 updated the YAML file to ensure we use .NET Core 3.1
- We manually ran the pipeline
- Pipeline ran through the ***azure-pipelines.yml*** file and executed the steps
- Our Solution was built

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…)

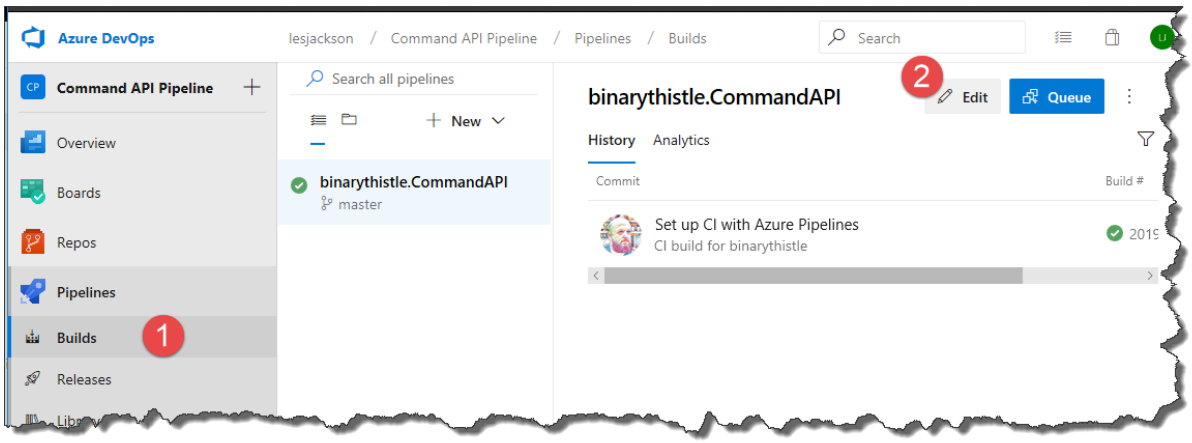
I thought we wanted to automate?

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!

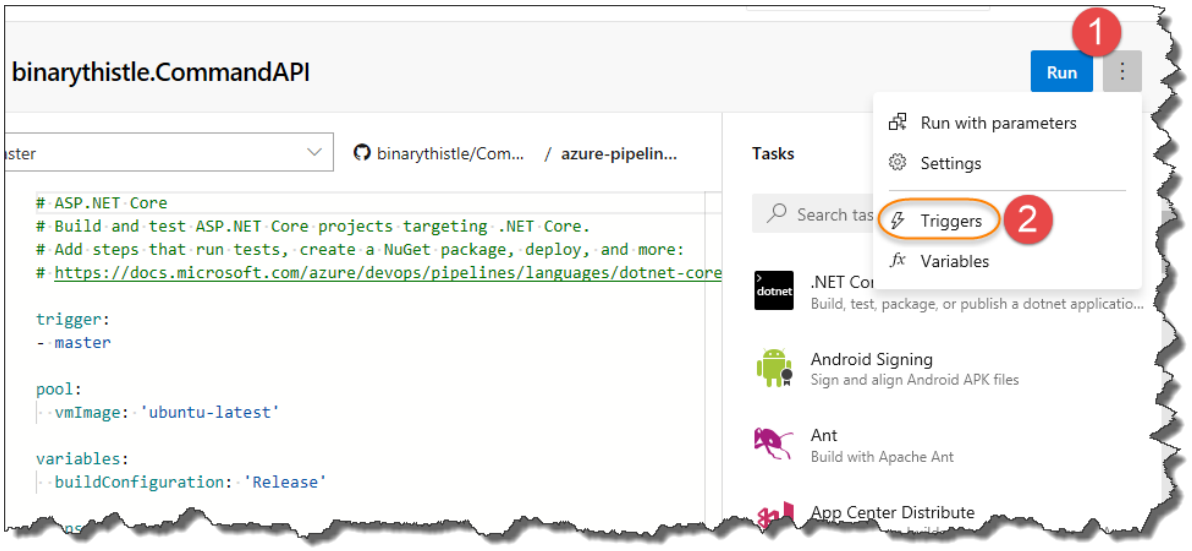
We are asked to execute when we created the pipeline that is true, but we can also set up "triggers", meaning we can configure the pipeline to execute when it receives a particular event…

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:

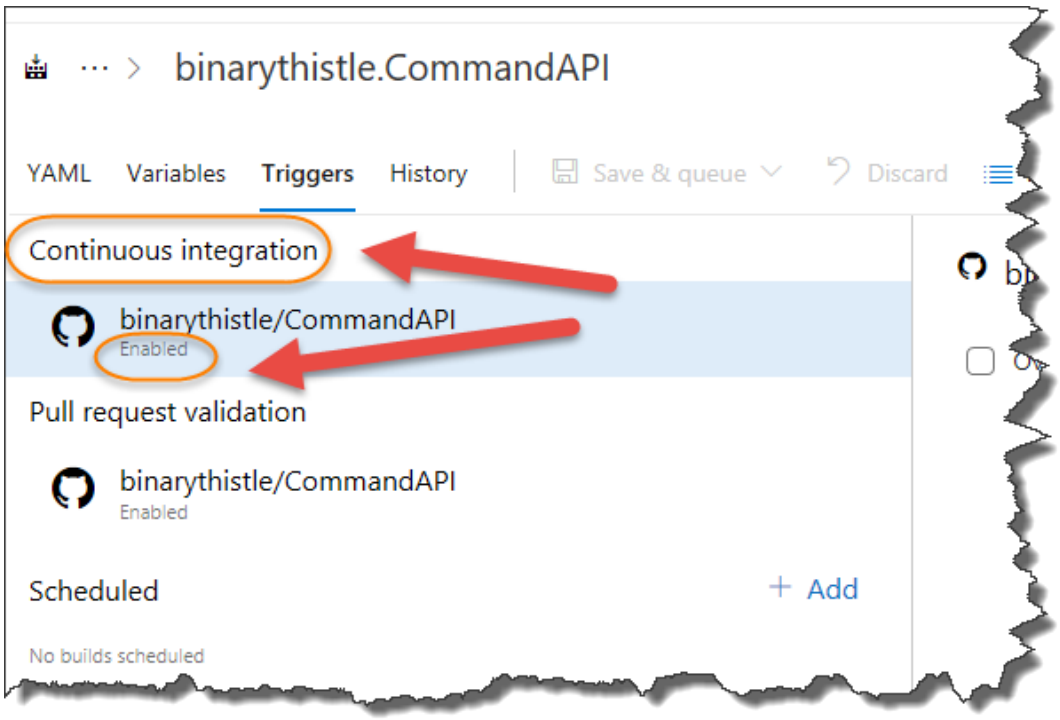


After doing that you should be returned to the ***azure-pipelines.yml*** file, (we will return here to edit it later),

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

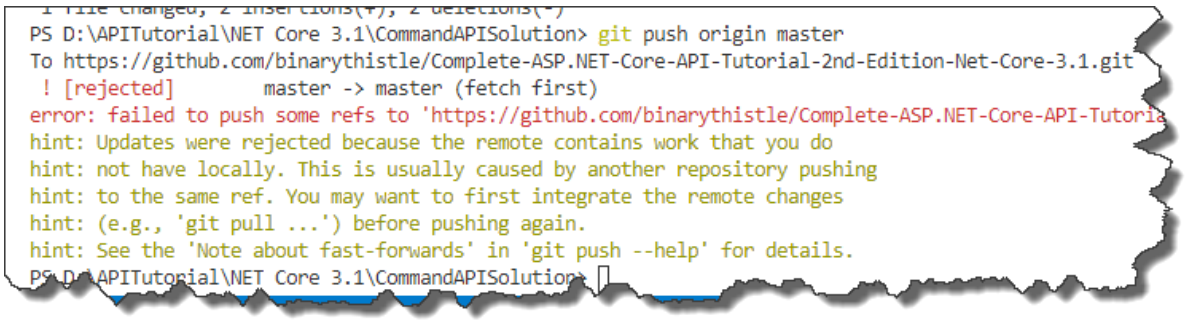
Triggering a build starts with a `git push origin master` to GitHub, so really any code change, (including something trivial like adding or editing a comment), will suffice. With that in mind, back in VS Code open `CommandsController` class in the "main" ***CommandAPI*** project and put a comment in our `GetCommandItems` method, reminding us to remove this code once we move to production:

```
//GET:      api/commands
[HttpGet]
public ActionResult<IEnumerable<Command>> GetCommandItems()
{
    //Random comment
    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:

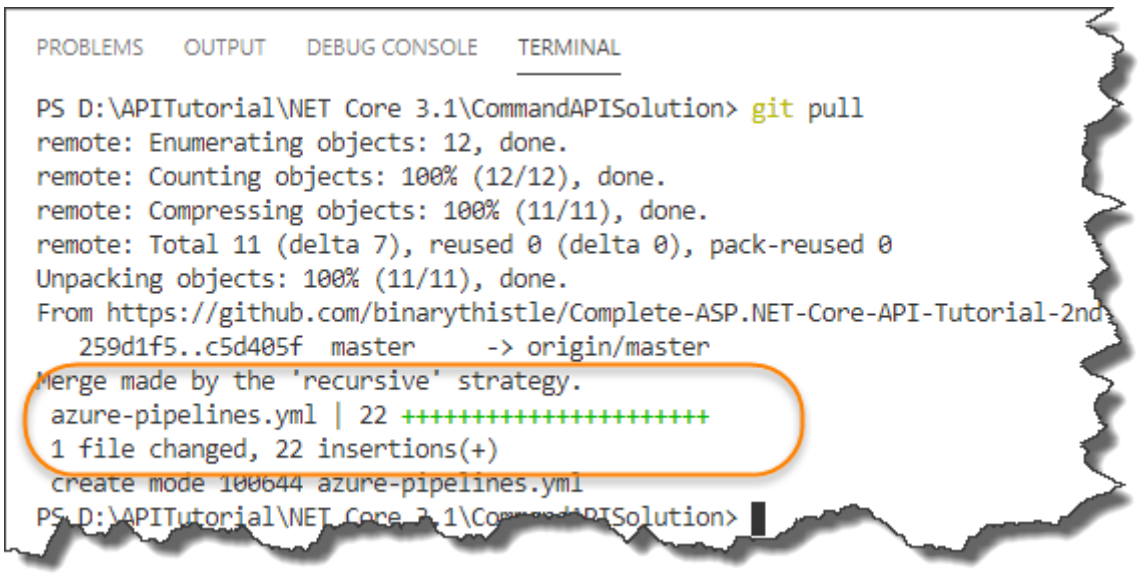


What does this mean?

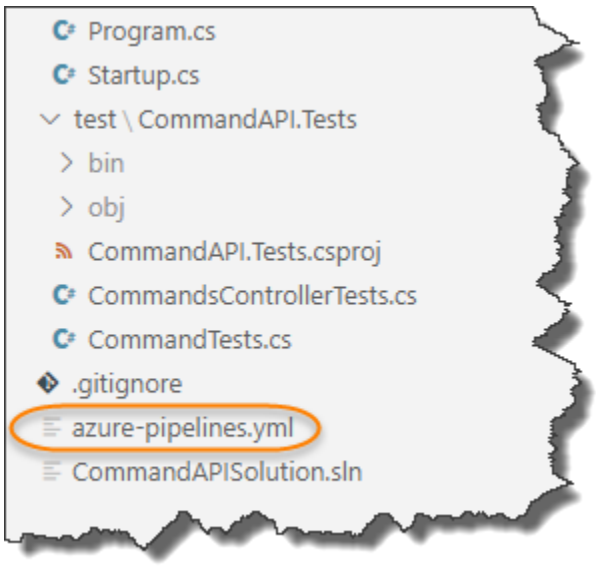
Well remember we added the ***azure-pipelines.yml*** file to the GitHub repo? Yes? Well that's the cause, essentially the local repository and the remote GitHub repository are out of sync, (the central GitHub repo has some newer changes than our local repository). To remedy this, we simply type:

```
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:

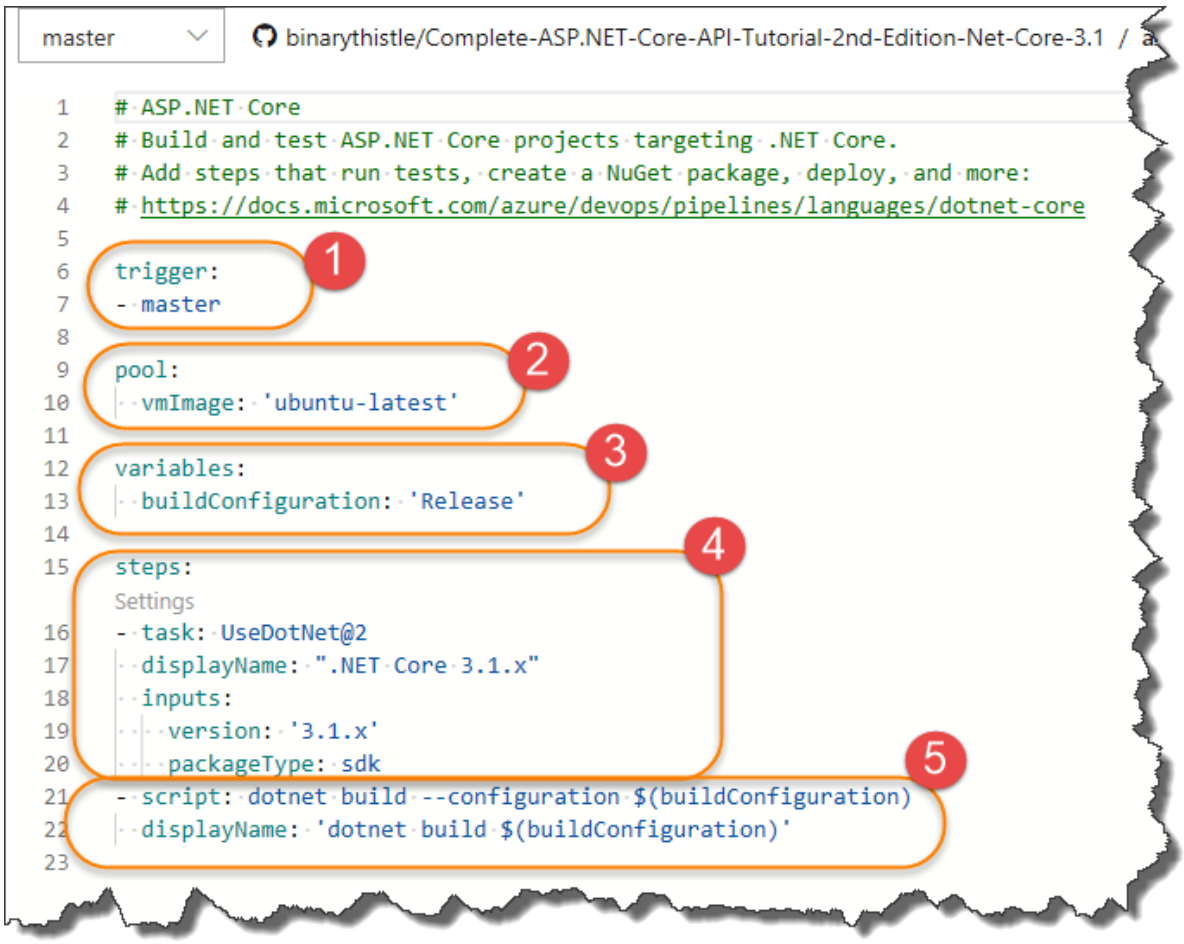
A new build has been queued to start, this time triggered by a remote commit to GitHub!

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:



This should look familiar as we edited this file in the last section, the sections highlighted are explained below:

1. Defines when the build is triggered and on what branch
2. Specified where VM Image were using to perform the build
3. Sets the `buildConfiguration` variable
4. Defines the version of the .NET Core Framework we want to use
5. Finally performs the build of the solution

You'll notice it doesn't perform any testing or packaging step, yet…

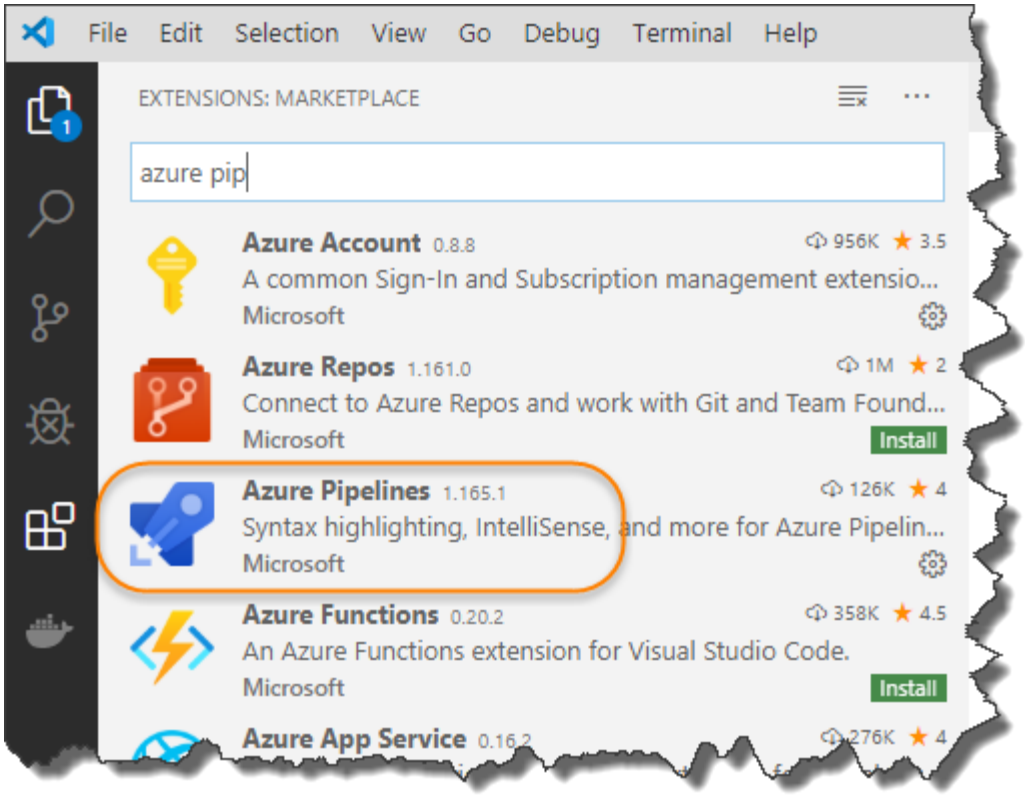
Another VS Code Extension

As we are going to be doing a bit of editing of the ***azure-pipelines.yml*** file, there are 2 places you can do this:

1. Directly in the browser (we've already done this)
2. In VS Code

The advantage that editing in the browser *had* was that it gave you some Intellisense-like functionality where it suggested some code snippets etc. However, Microsoft have now released a VS Code extension to provide similar functionality in VS Code, so we're going to install and use that, (it means we do all our coding in the one place).

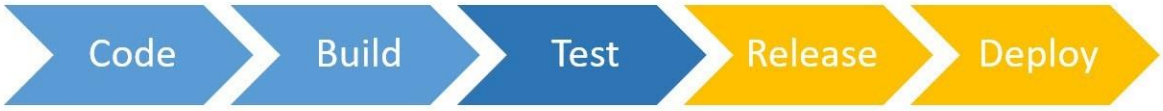
In VS Code, click on the Extensions button and search for "Azure Pipelines", you should see:



Install it and then open ***azure-pipelines.yml*** file that we just pulled down from GitHub.

Running Unit Tests

Returning to the steps in our pipeline view:



You'll see the suggested sequencing is: Build->**Test->** Release, so let's add that task to our ***azure-pipelines.yml*** file now.

Move back to VS Code, open ***azure-pipelines.yml*** and *append* the following Task *after* the build Task:

```
- task: DotNetCoreCLI@2
  displayName: 'dotnet test'
  inputs:
      command: test
      projects: '**/*Tests/*.csproj'
      testRunTitle: 'xUNit Test Run'
```

So overall the file should like this, again with our new Task step highlighted:

azure-pipelines.yml > [ ] steps > displayName

```yaml
trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

variables:
  buildConfiguration: 'Release'

steps:
- task: UseDotNet@2
  displayName: ".NET Core 3.1.x"
  inputs:
    version: '3.1.x'
    packageType: sdk
- script: dotnet build --configuration $(buildConfiguration)
  displayName: 'dotnet build $(buildConfiguration)'

- task: DotNetCoreCLI@2
  displayName: 'dotnet test'
  inputs:
    command: test
    projects: '**/*Tests/*.csproj'
    testRunTitle: 'xUnit Test Run'
```

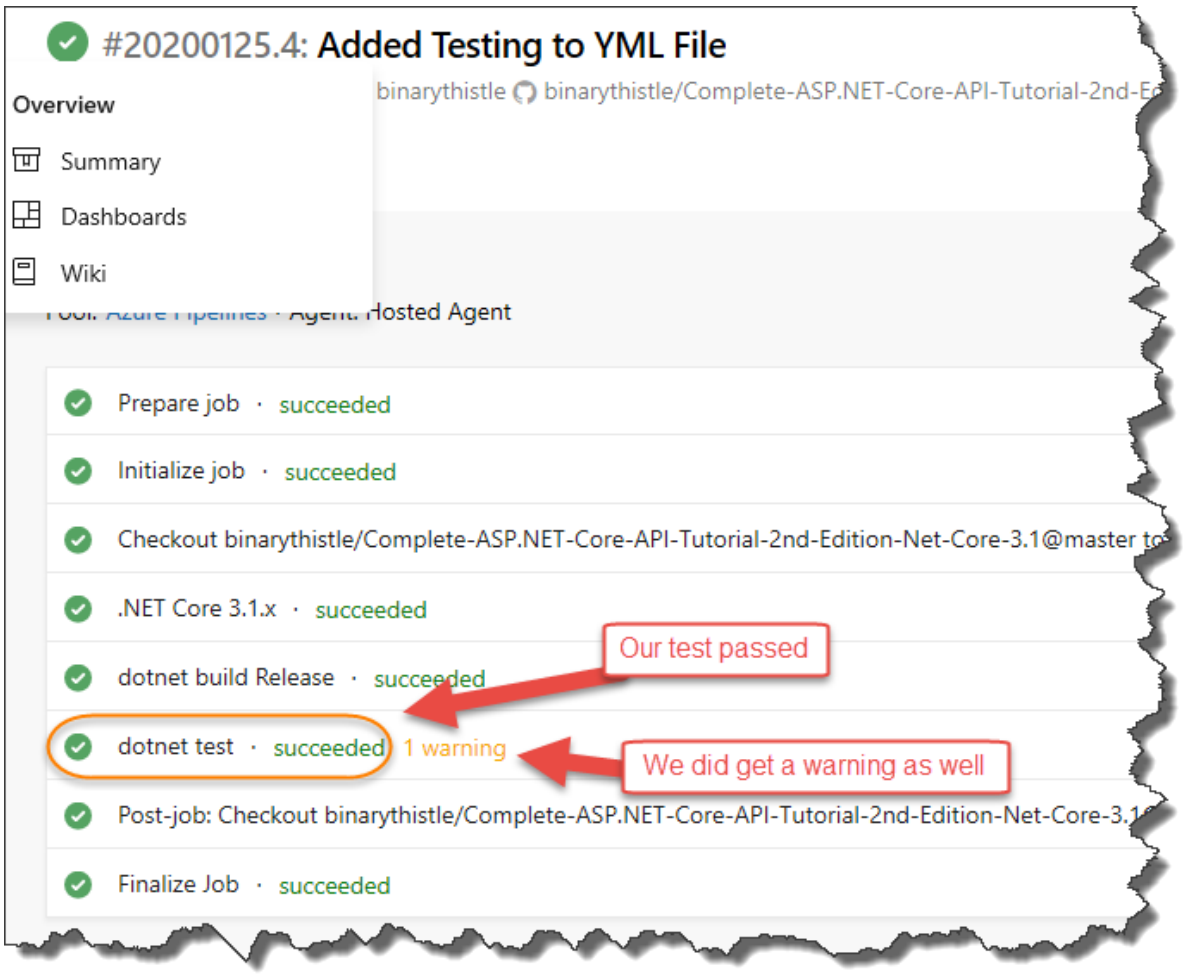
Unit test section

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.

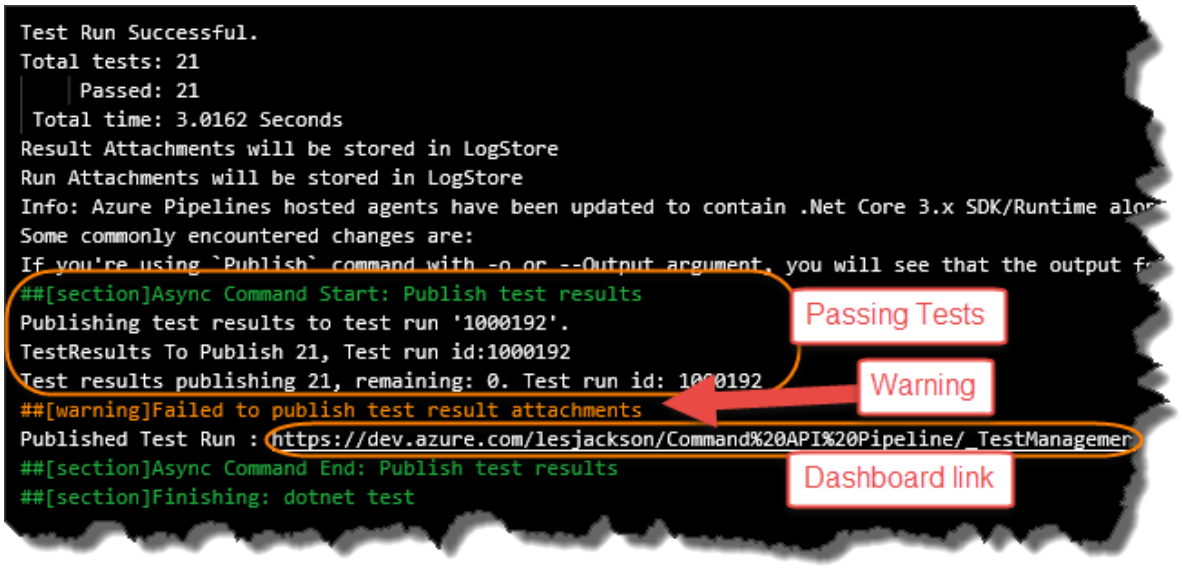
I'm in good company getting things back to front though…

- 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)

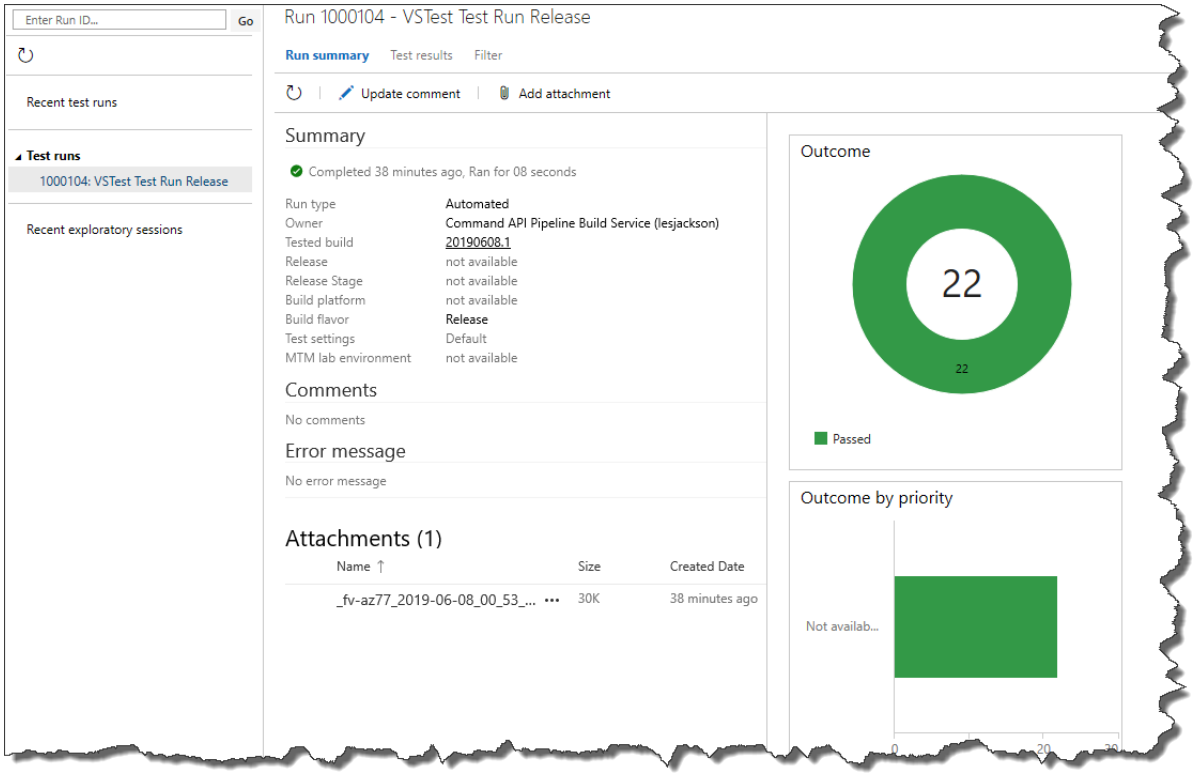
The steps are quite self-explanatory, so save the file in VS Code, and perform the necessary Git command line steps to commit your code and push to GitHub – this should trigger another build of our pipeline – this time the unit tests should execute too:



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…

As for the warning I'm getting, doing a bit of Googling it looks like a, (benign), common problem / annoyance with no way to rectify at the time of writing. Although annoying and a little disconcerting I think we can safely ignore it.

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`:

```
[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
```

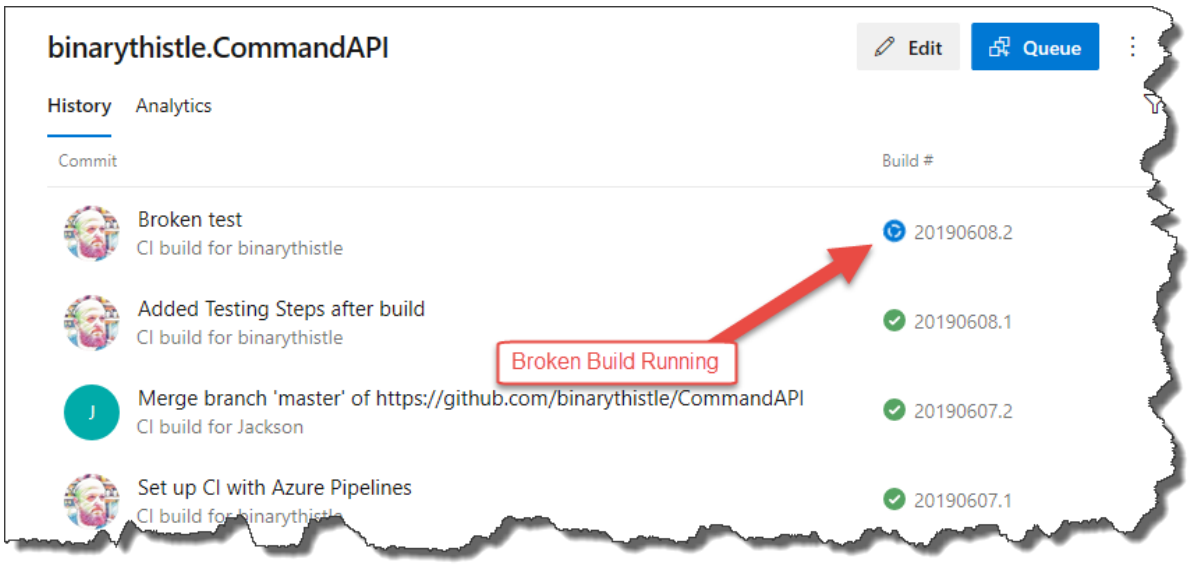
We'll of course get a failing result:

```
  X CommandAPI.Tests.CommandsControllerTests.DeleteCommandItem_Returns404No
  Error Message:
   Assert.IsType() Failure
Expected: Microsoft.AspNetCore.Mvc.OkResult
Actual:   Microsoft.AspNetCore.Mvc.NotFoundResult
  Stack Trace:
     at CommandAPI.Tests.CommandsControllerTests.DeleteCommandItem_Retur
s.cs:line 408


Test Run Failed.
Total tests: 21
     Passed: 20
     Failed: 1
 Total time: 1.3283 Seconds
PS D:\APITutorial\NET Core 3.1\CommandAPISolution>
```

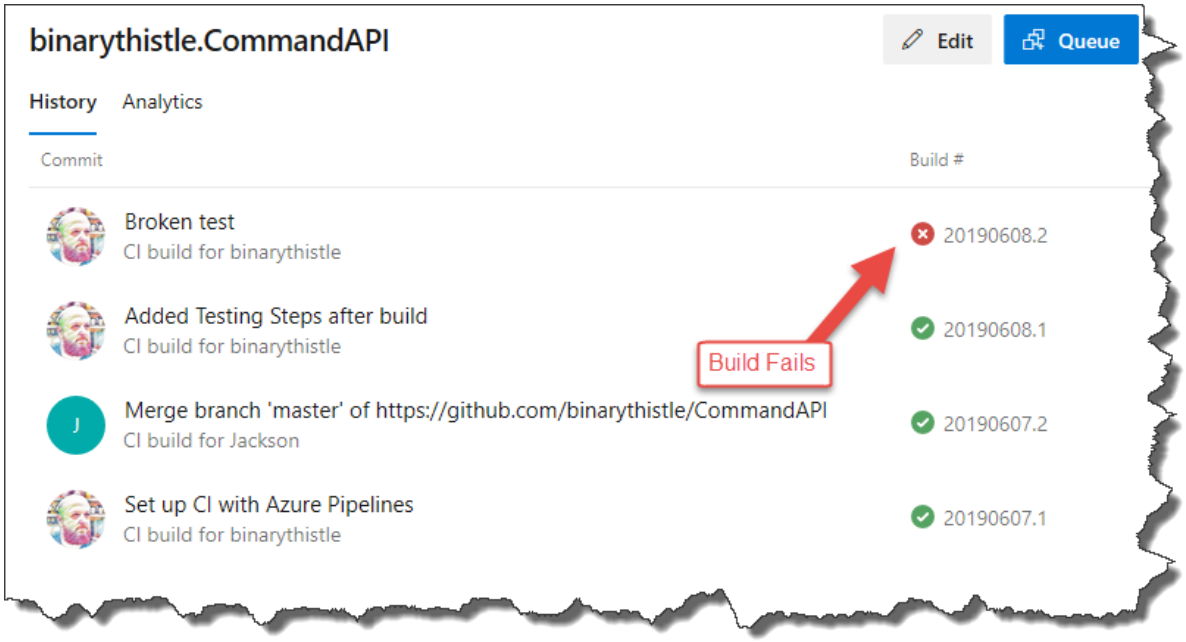
Now under normal circumstance, having just caused our unit test suite to fail locally, you **would not then commit** the changes and push them to GitHub! However, that is exactly what we are going to do just to prove the point that the tests will fail in the Azure DevOps build pipeline too.

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.

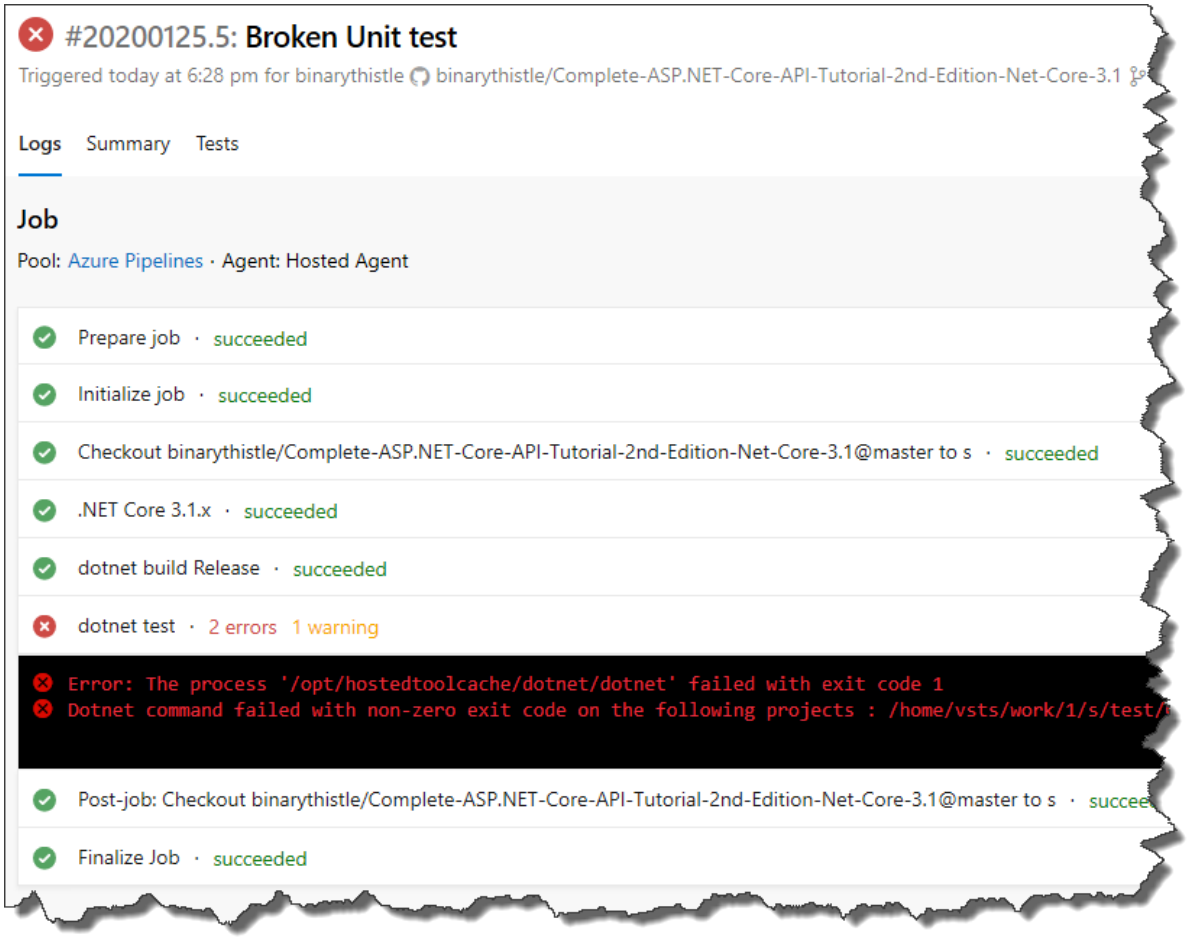
So perform the 3 "Git" steps you should be familiar with now, and once you've pushed to GitHub, move back across to Azure DevOps and observe what happens…



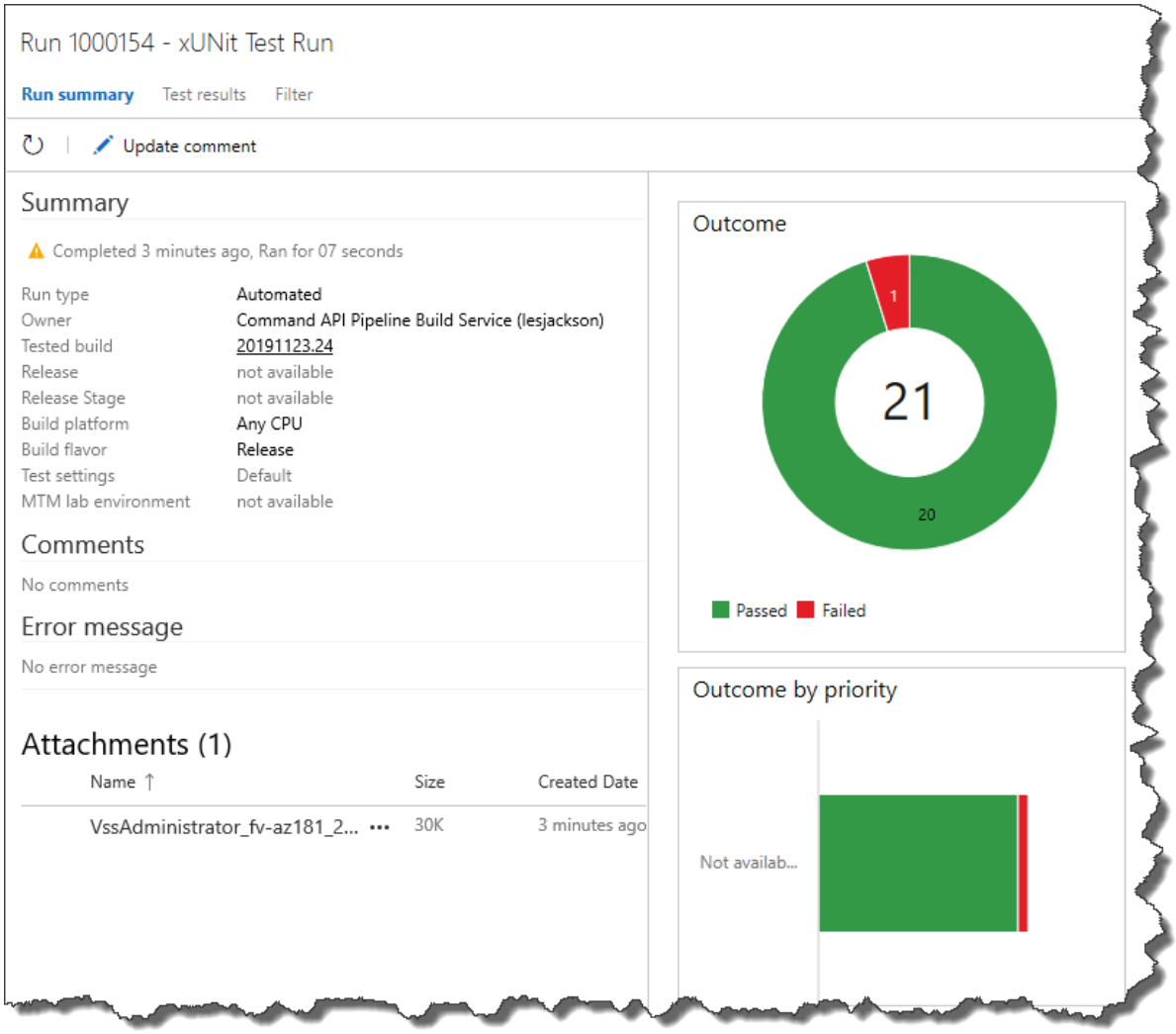
Then as expected, our test fails:



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. Looking at the individual steps:



And then drilling further in to the dotnet test step and going to the test results dashboard:



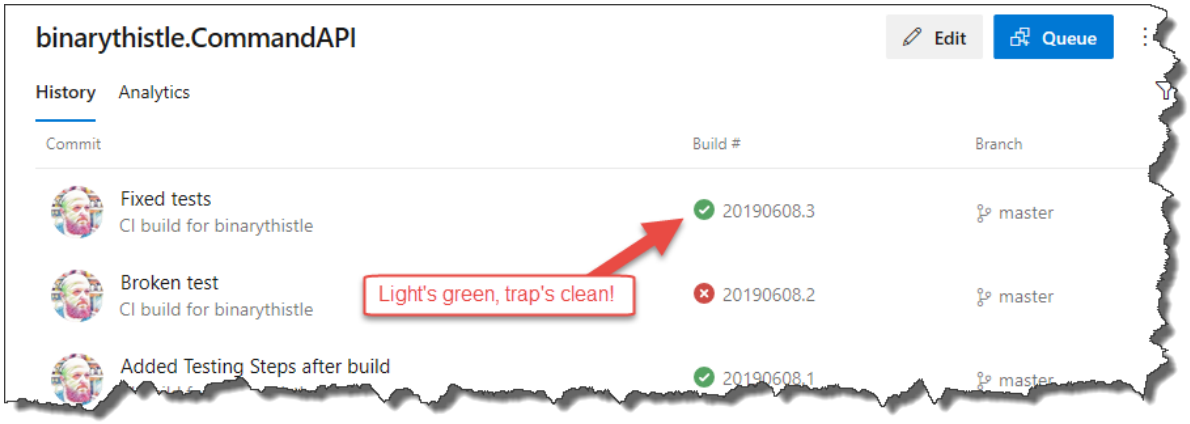
Testing – The Great Catch All?

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…

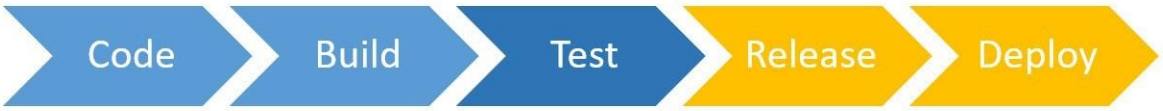
Before we do that though revert the change, we just made to ensure that all our unit tests are passing, and that our pipeline returns to a green state.

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 into VS Code and open ***azure-pipelines.yml*** file, and append the following steps:

```
- task: DotNetCoreCLI@2
  displayName: 'dotnet publish'
  inputs:
    command: publish
    publishWebProjects: false
    projects: 'src/CommandAPI/*.csproj'
    arguments: '--configuration $(buildConfiguration) --output
$(Build.ArtifactStagingDirectory)'

- 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:
- task: UseDotNet@2
  displayName: ".NET Core 3.1.x"
  inputs:
    version: '3.1.x'
    packageType: sdk
- script: dotnet build --configuration $(buildConfiguration)
  displayName: 'dotnet build $(buildConfiguration)'

- task: DotNetCoreCLI@2
  displayName: 'dotnet test'
  inputs:
    command: test
    projects: '**/*Tests/*.csproj'
    testRunTitle: 'xUnit Test Run'

- task: DotNetCoreCLI@2
  displayName: 'dotnet publish'
  inputs:
    command: publish
    publishWebProjects: false
    projects: 'src/CommandAPI/*.csproj'
    arguments: '--configuration $(buildConfiguration) --output $(Build.ArtifactStagingDirectory)'

- 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 [21]
- The output of that is zipped
- Finally zipped output is published

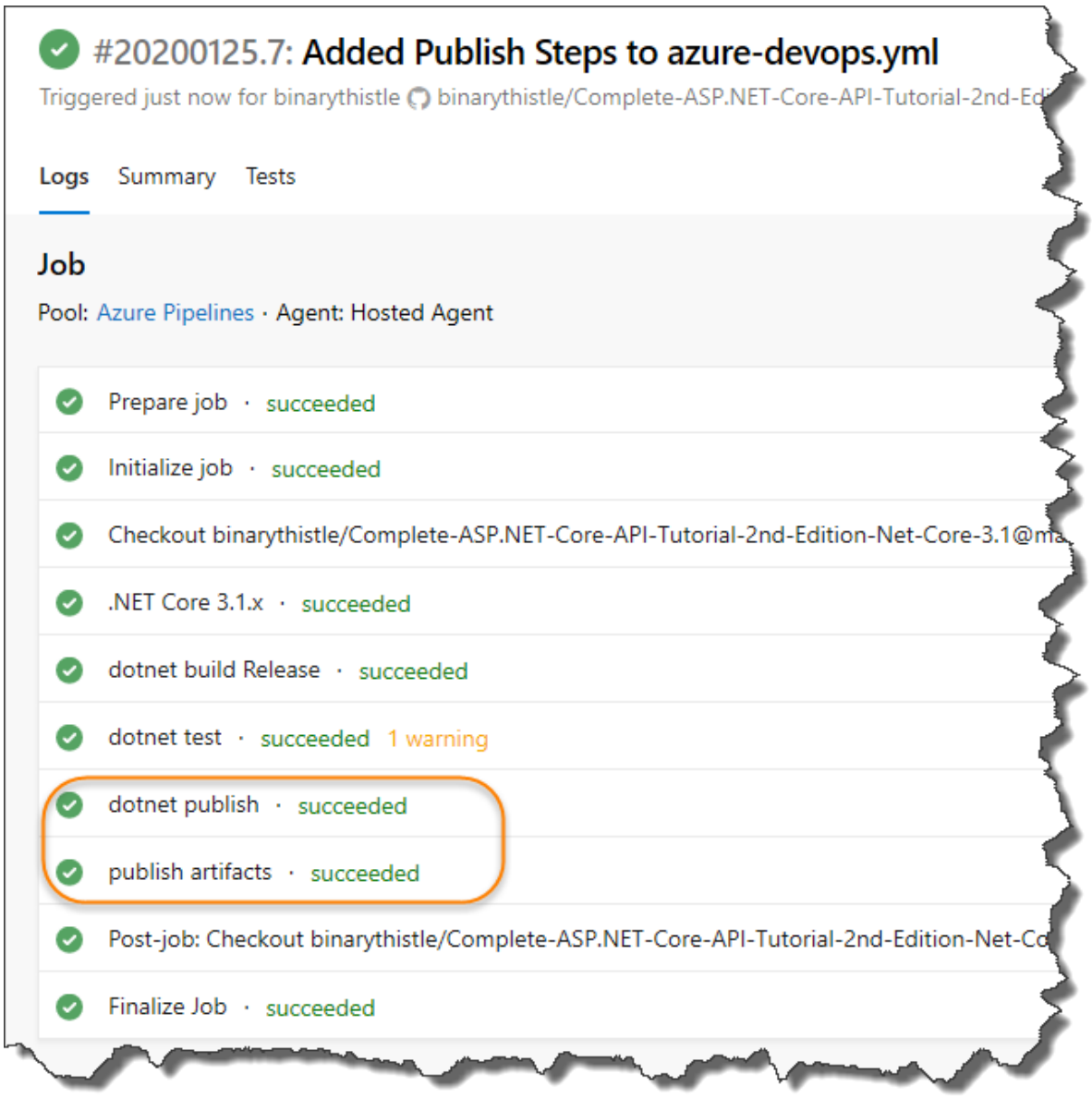
Les's Personal Anecdote: Ensure that you put in the following line:

```
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:

[21] 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:

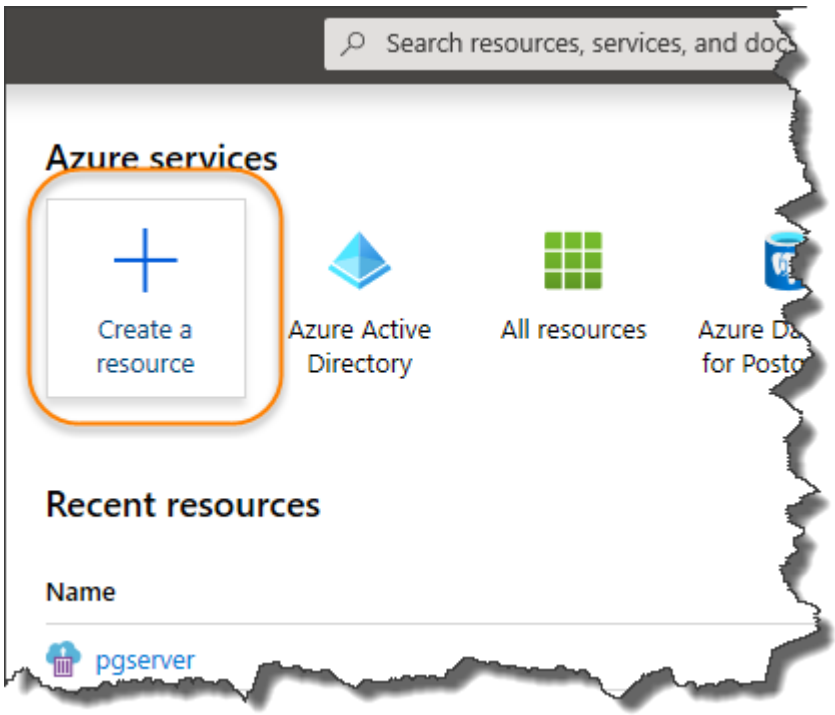
1. Create resources manually via the Azure Portal
2. Create resources automatically via Azure Resource Manager Templates

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

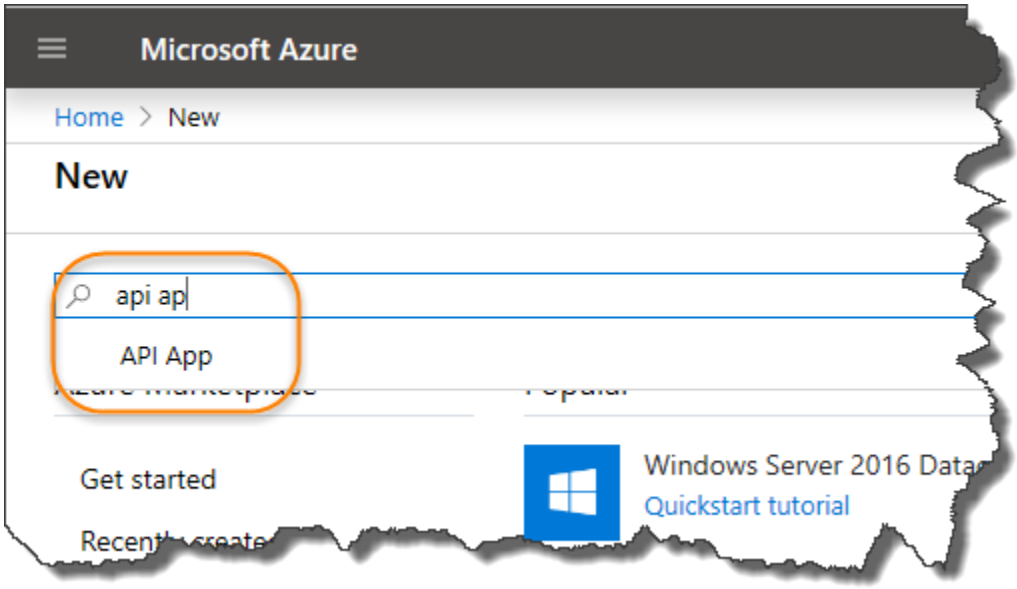
The 1st resource we are going to create is an API App, this unsurprisingly is where our API code will run! To do so, Login to Azure, (or if you don't have an account you'll need to create one), and click on "Create a resource":



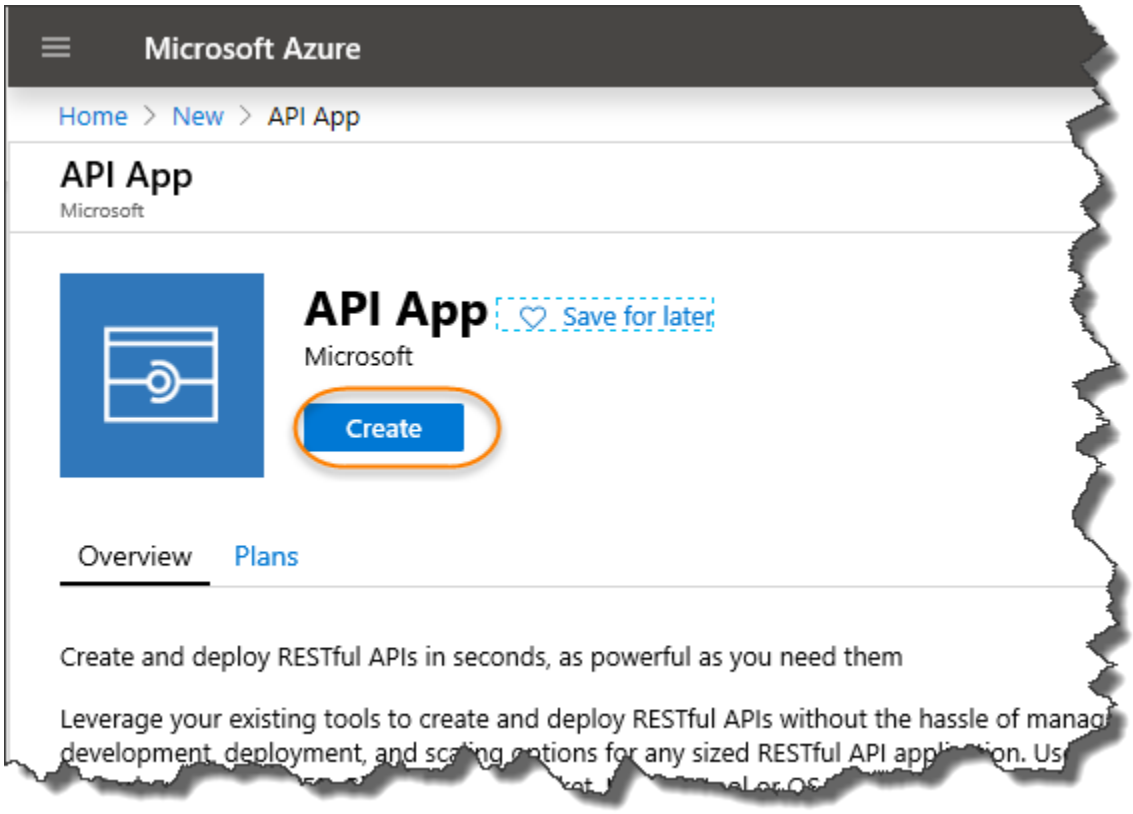
Again, I'll mention the point that the following screenshot were correct at the time of writing but given the fast pace of change in Azure they may be subject to change.

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:



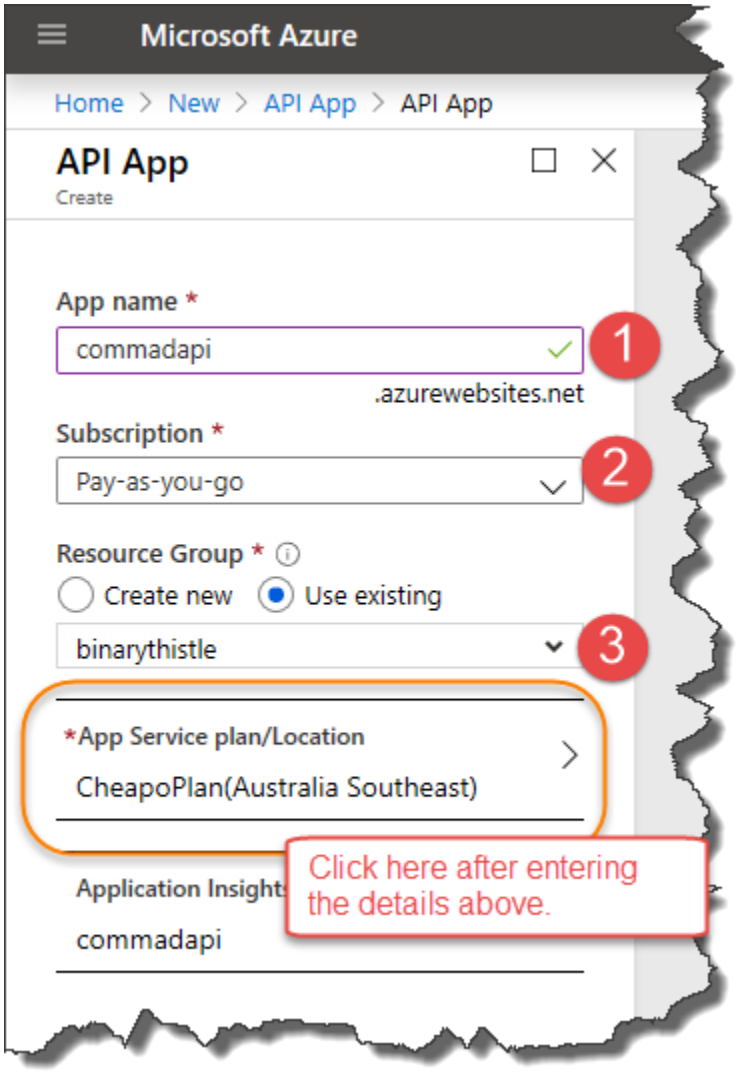
Select “API App”, then click “Create”:



On the Next “page”, enter:

1. A name for your API App[22]
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!

[22] 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…

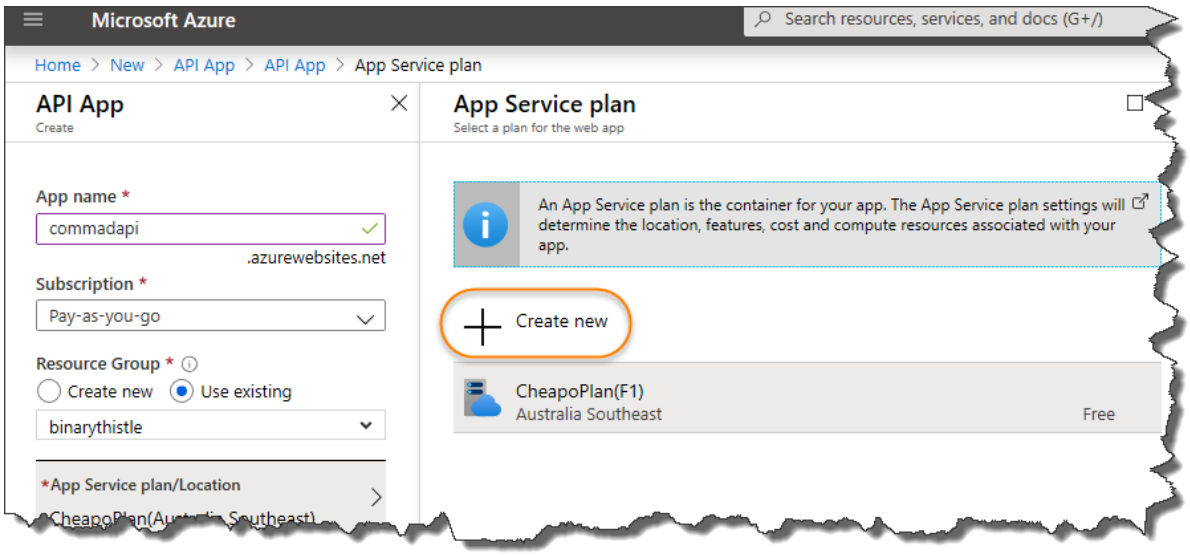
E.g. Do you want your API App:

- Hosted in the US, Western Europe, Asia etc,
- On shared or dedicated hardware
- Running on certain processor speed, etc.

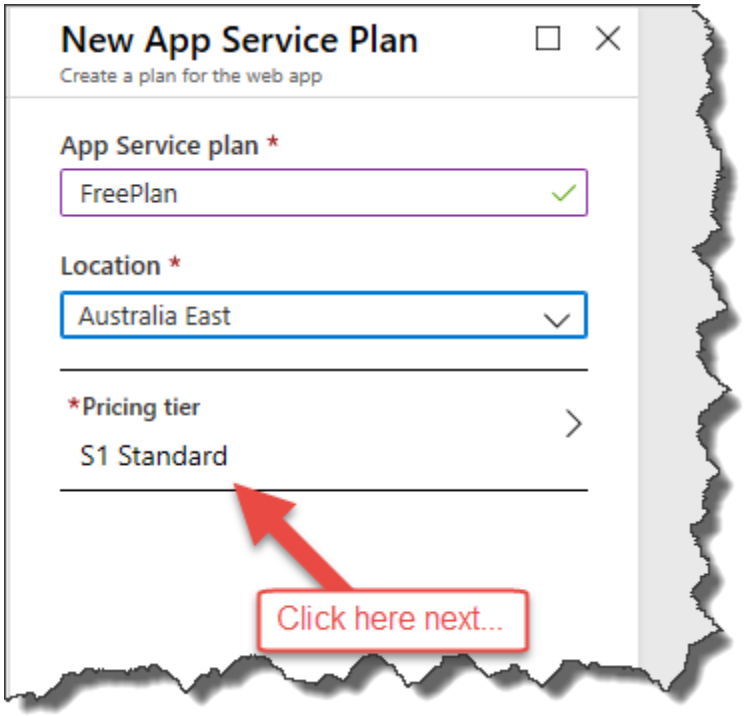
By default if you've not used Azure before you'll be placed on a Standard plan which can incur costs! (This is a personal anecdote because I did that and was shocked when my test API started costing me money!).

So be careful of the Service Plan you set up, I detail the free plan next.

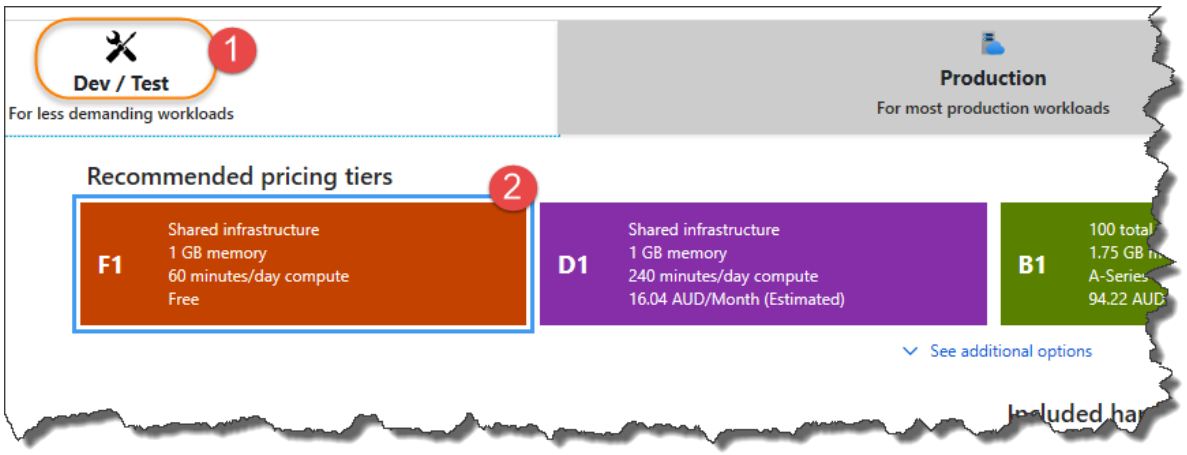
After clicking on the Service Plan, click on "Create new":



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



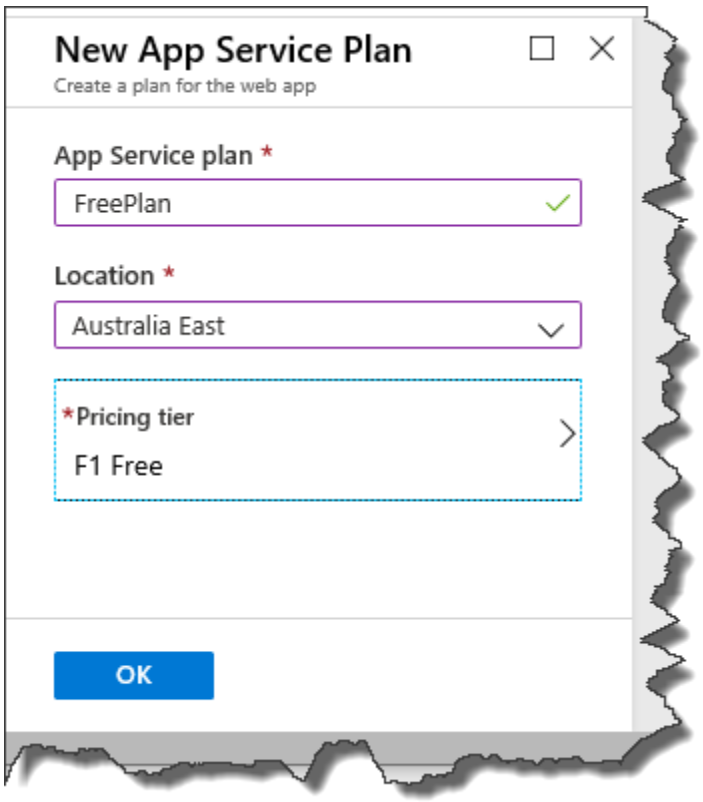
After, click on the *Pricing Tier:*



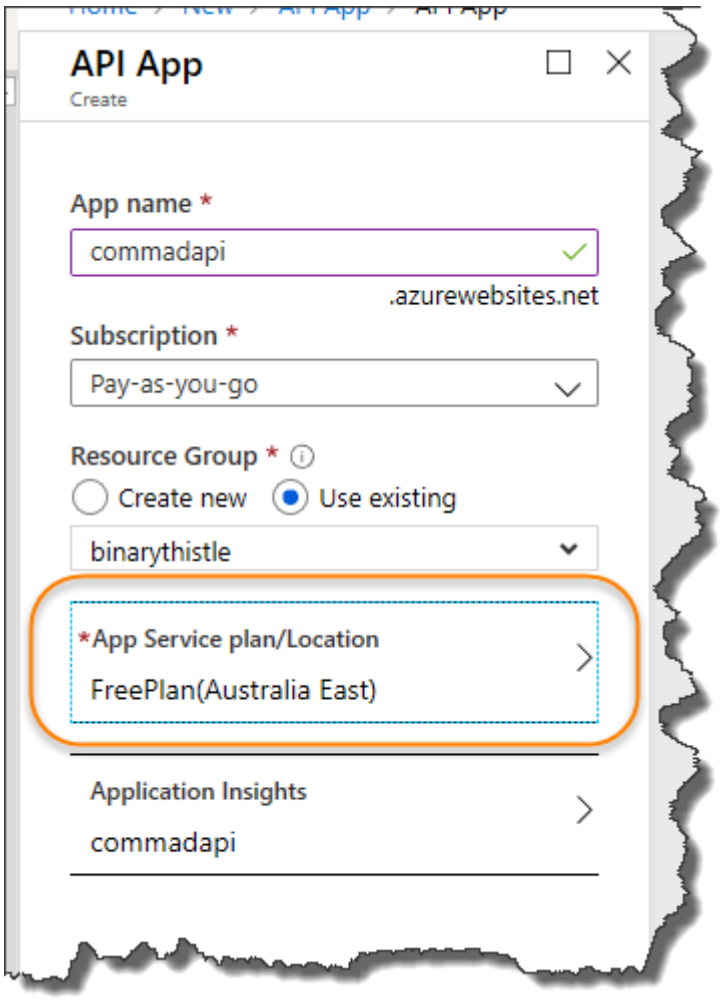
1. Select the Dev/Test Tab
2. Select the "F1" Option (Shared Infrastructure / 60 minutes compute)
3. Click Apply

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).

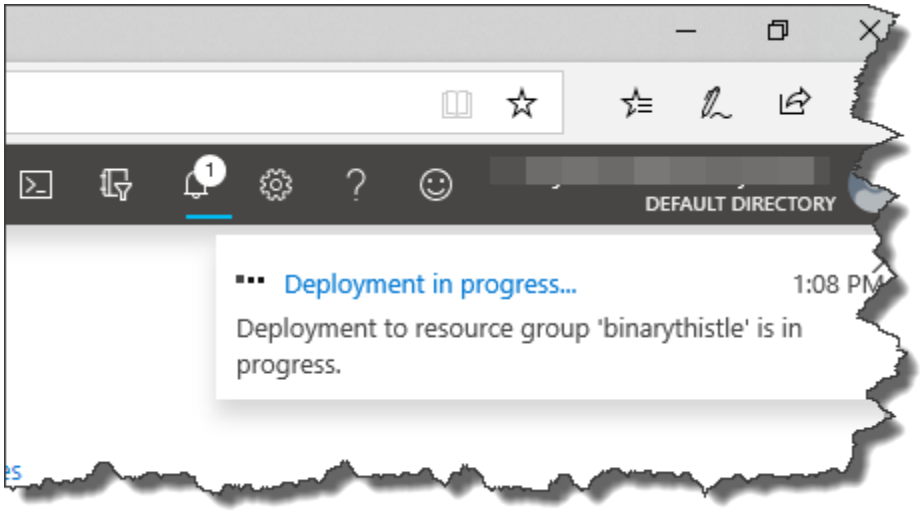
Then Click OK…



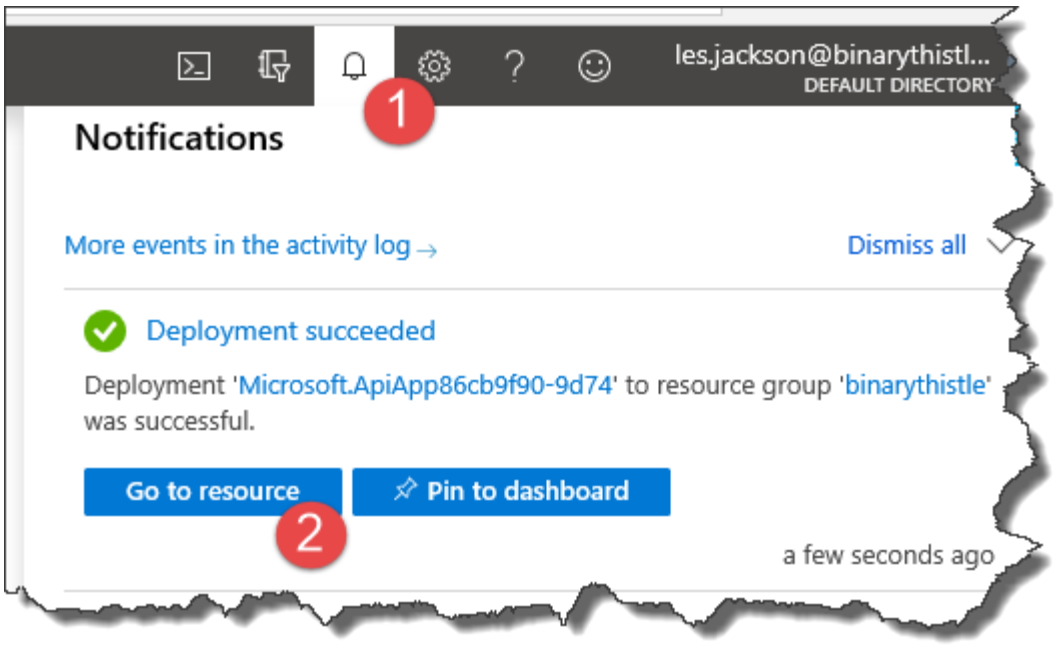
Then click "Create", (ensure your new App Service Plan is selected):



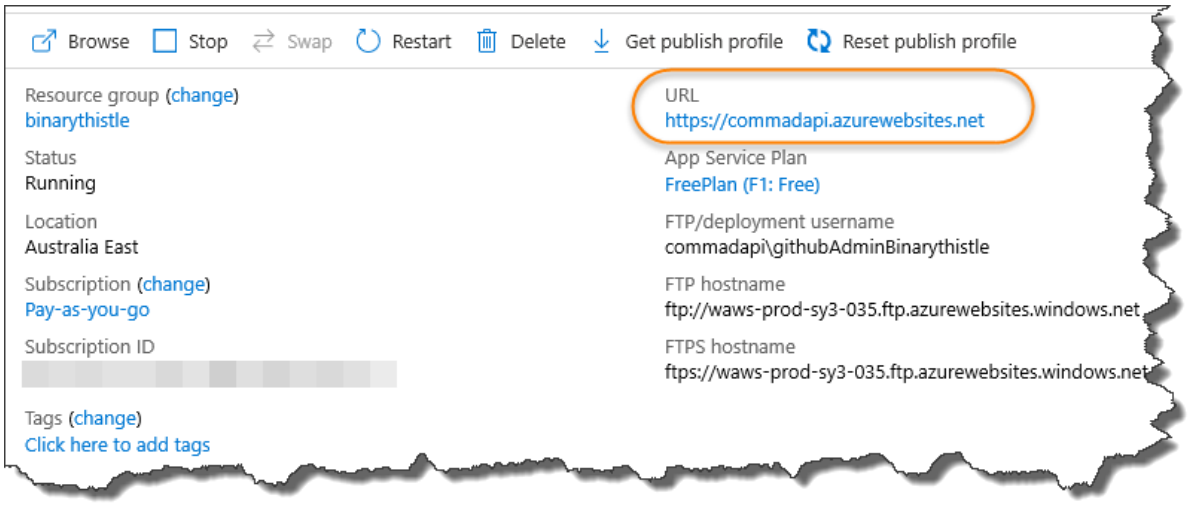
After clicking Create, Azure will go off an create the resource ready for use:



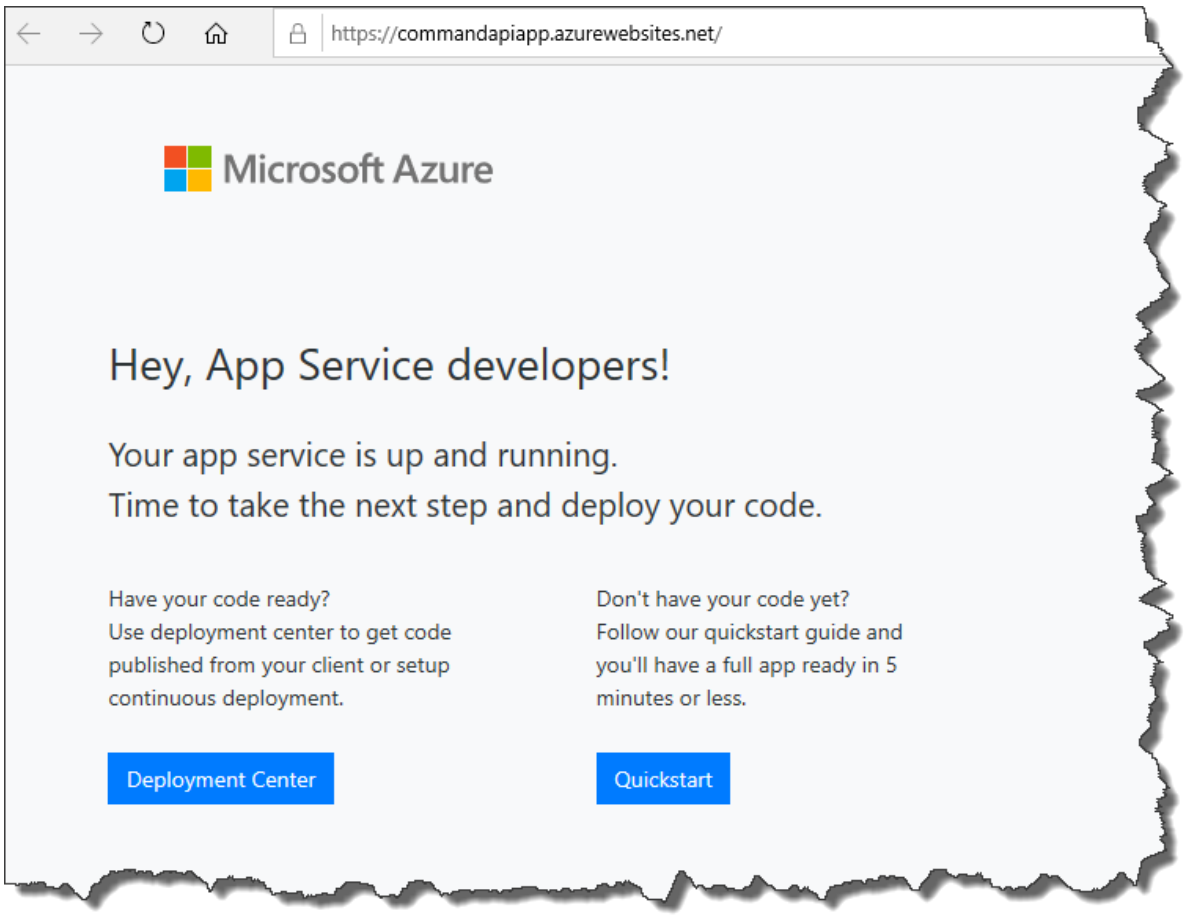
You will get notified when the resource is successfully created, if not, click the little “Alarm Bell” icon near the top right-hand side of the Azure portal:



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 PostgeSQL Server

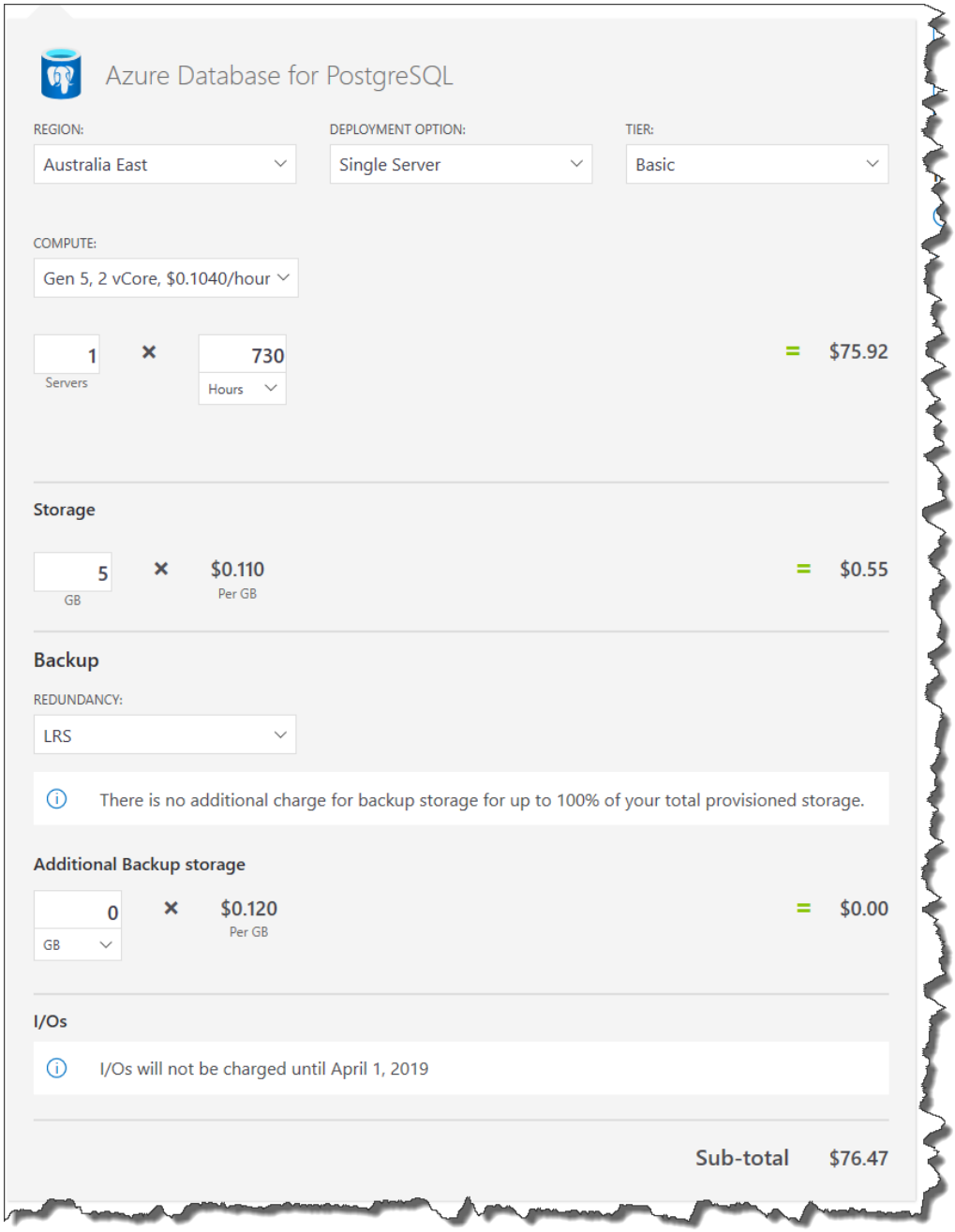
Now there are a number of different ways that you can create a PostgreSQL database on Azure, but I'm going to take a slightly unorthodox route and spin up a PostgreSQL Server in a Container Instance in Azure, (think Docker containers).

I've taken this approach primarily because the set up is so simple and the cost implications are low. To illustrate my point, compare the estimated costs for:

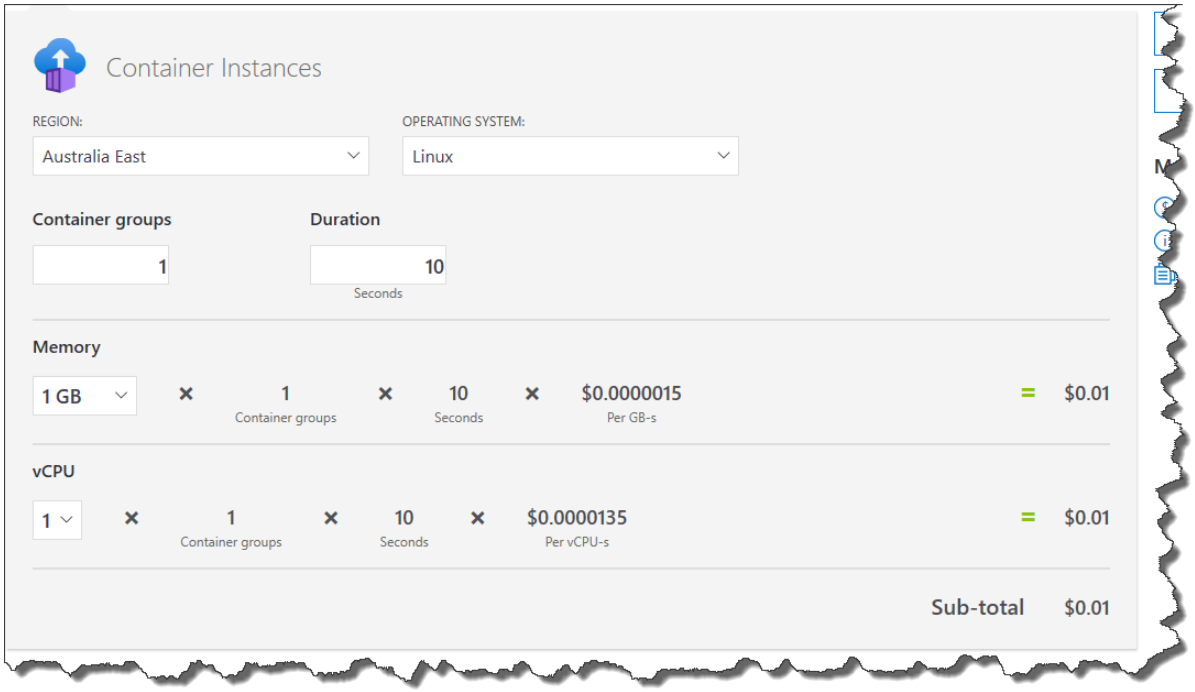
- Azure Database for PostgreSQL Servers
- Container Instance running a PostgreSQL Image

Azure Database for PostgreSQL Servers

I've configured the most basic example of this that I could.



CONTAINER INSTANCE PRICING

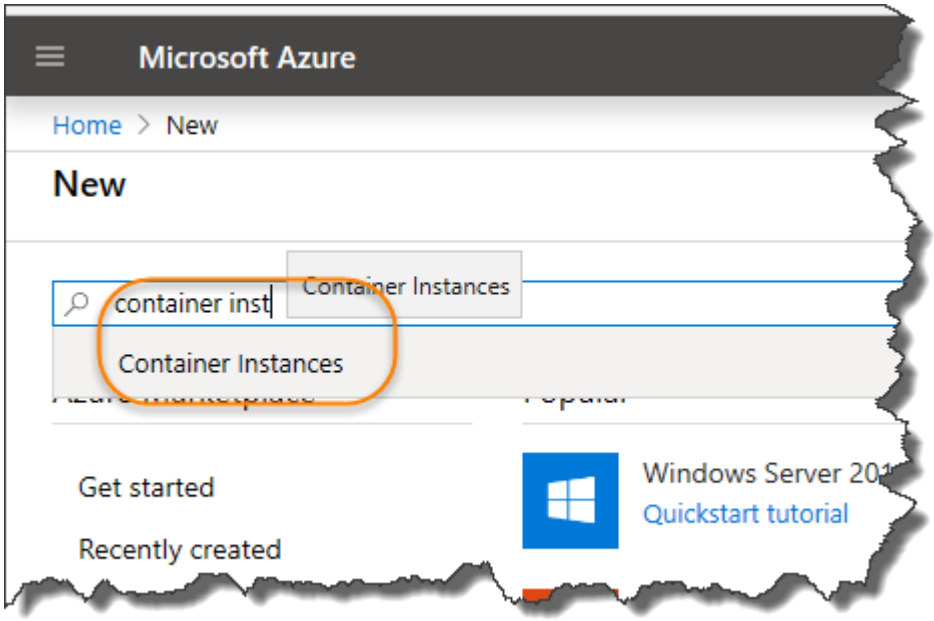


Now I don't need to tell you that "you get what you pay for" in this life, so clearly the *Azure Database for PostgreSQL* option a purpose build resource that's designed to work as a database, whereas the container option I'm taking is in no way optimised for Production performance.

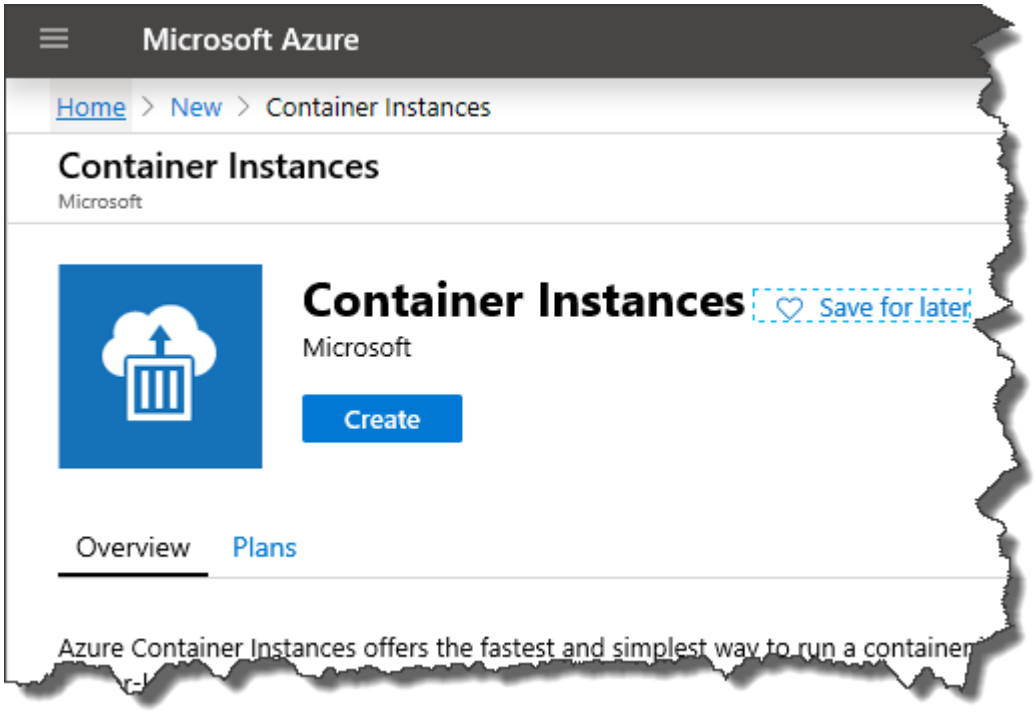
Warning! If you restart the PostgreSQL container instance that we create in the section below, it essentially re-sets and you will lose your configuration and data – just something to bear in mind…

For the purposes of setting up our "production" environment on Azure, the container option is fine.

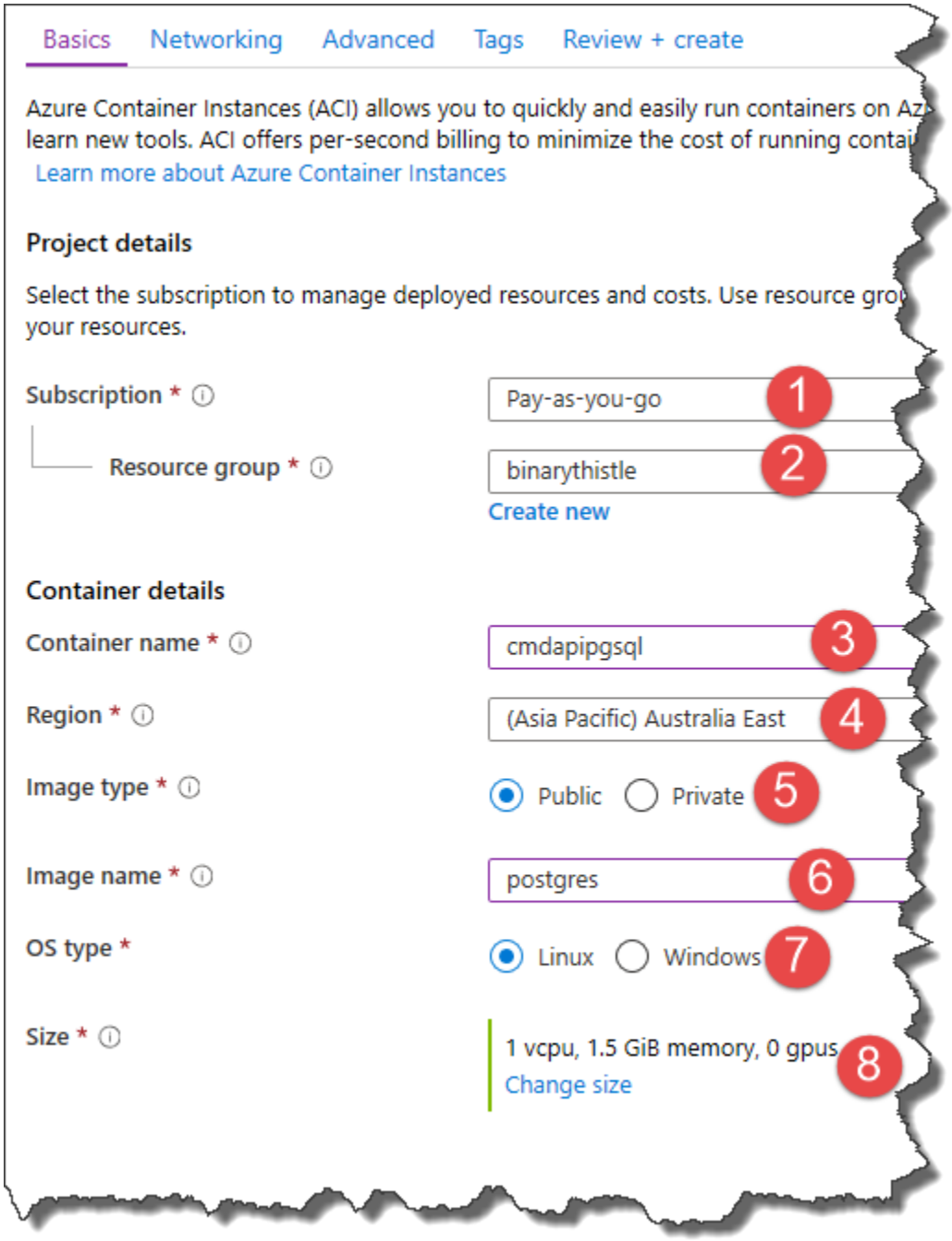
So back in Azure, once again click "Create a Resource" and this time search for "Container Instances":



Select “Container Instances” from the options drop-down, and you should be displayed the Container Instances detail screen, click “Create” continue:

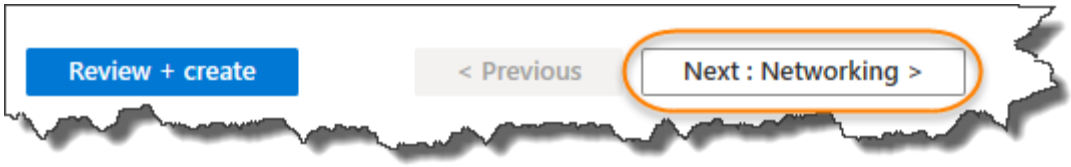


You’ll get taken to the Basics tab on the creating wizard, fill out the details as relevant to you, however the image name must be ***postgres***, see note below:

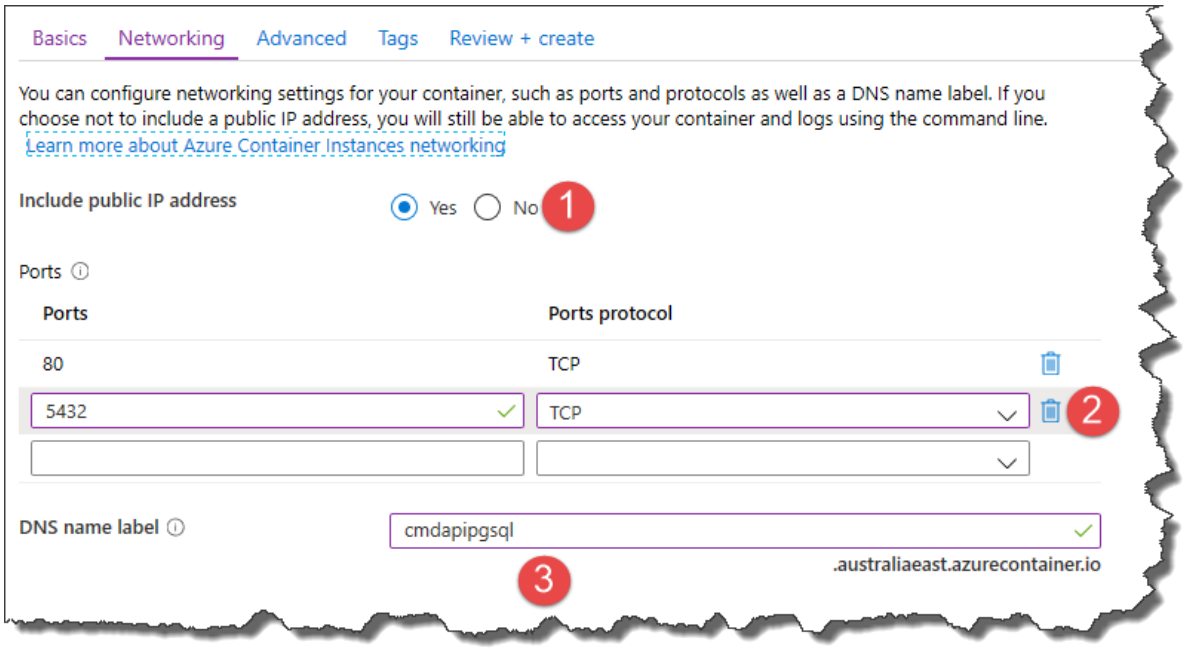


1. Your subscription
2. Resource Group, (I'd make this the same as the one you placed the API app into)
3. Container name can be anything, but I'd name it something that identified it as a PostgreSQL server
4. Region, (I'd make this the same as the one you placed the API app into)
5. Image type, select Public (the postgres image we use in the next step is publicly available on Docker Hub)
6. Image name, as mentioned above this needs to be the exact name of the image on Docker Hub, so in this case ***postgres***
7. OS Type, select Linux
8. Size, leave these as the defaults.

When you're happy click "Next: Networking >"

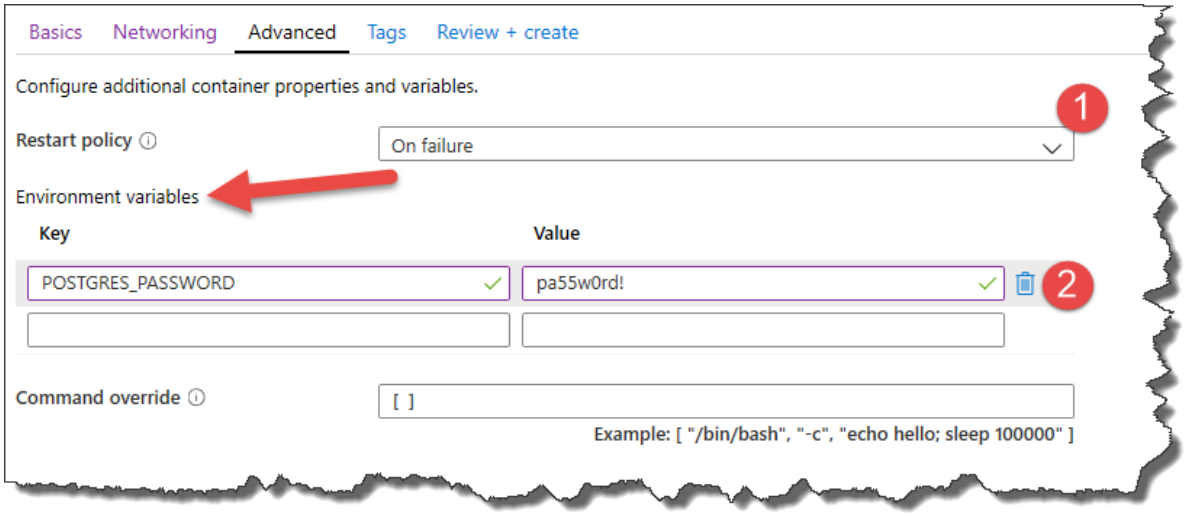


And supply the following details in the Networking Tab:



1. Select “Yes” for Include public IP Address (not that this is not a static address)
2. Add the standard 5432 TCP port
3. I’d add a DNS name label as the IP Address can change if the container re-startsWhen you’re happy click “Next: Advanced >”

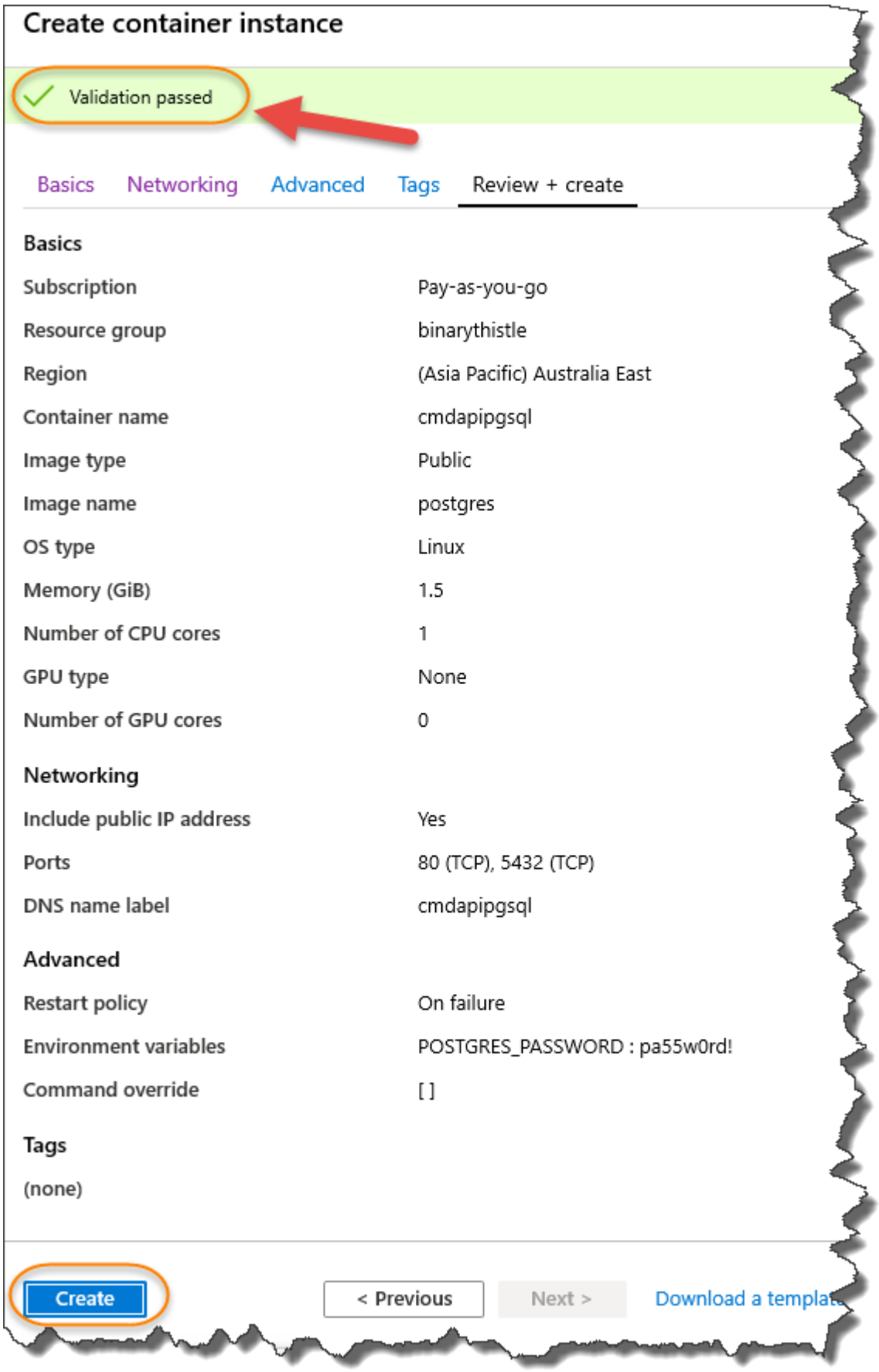
And enter the following detains on the Advance Tab:



1. Set the Restart Policy to “On Failure”
2. Create an “environment variable” for the Postgres password for the default database

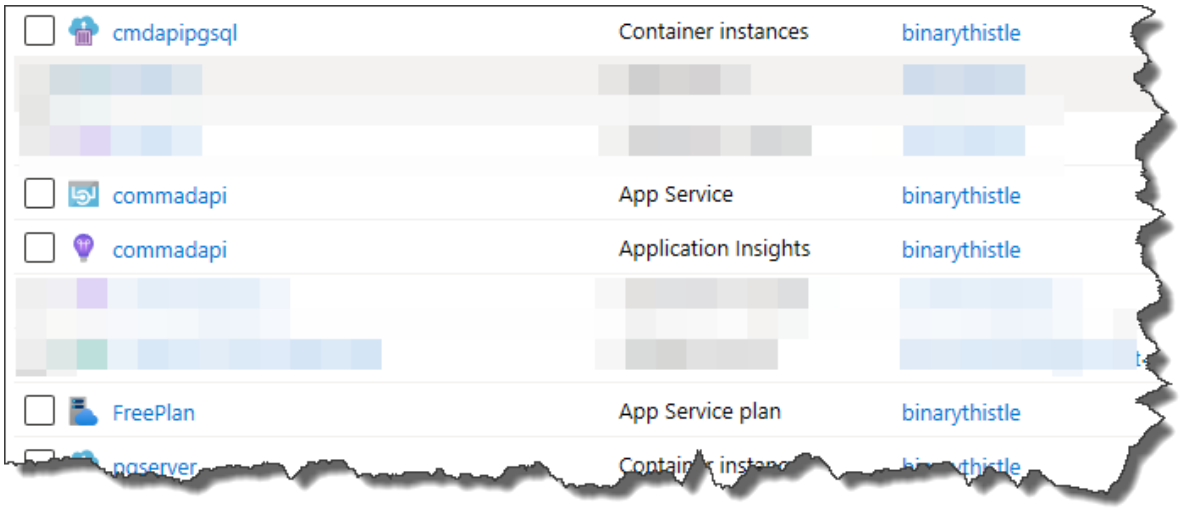
If you pop back to Chapter 6 where I set up an instance of PostgreSQL locally using Docker Desktop, there is a bit more of a discussion on this – I don’t want to repeat myself here but it’s quite self-explanatory.

Click “Review and Create”, (we can skip the “Tags” tab):


Create container instance
Validation passed
Basics
Networking
Advanced
Tags
Review + create
Basics
Subscription
Pay-as-you-go
Resource group
binarythistle
Region
(Asia Pacific) Australia East
Container name
cmdapipgsql
Image type
Public
Image name
postgres
OS type
Linux
Memory (GiB)
1.5
Number of CPU cores
1
GPU type
None
Number of GPU cores
0
Networking
Include public IP address
Yes
Ports
80 (TCP), 5432 (TCP)
DNS name label
cmdapipgsql
Advanced
Restart policy
On failure
Environment variables
POSTGRES_PASSWORD : pa55w0rd!
Command override
[ ]
Tags
(none)
Create
< Previous
Next >

You should see “Validation Passed” at the top of the screen, when you’re happy click create, and in a similar way to the API App, Azure will go off and create your resource.

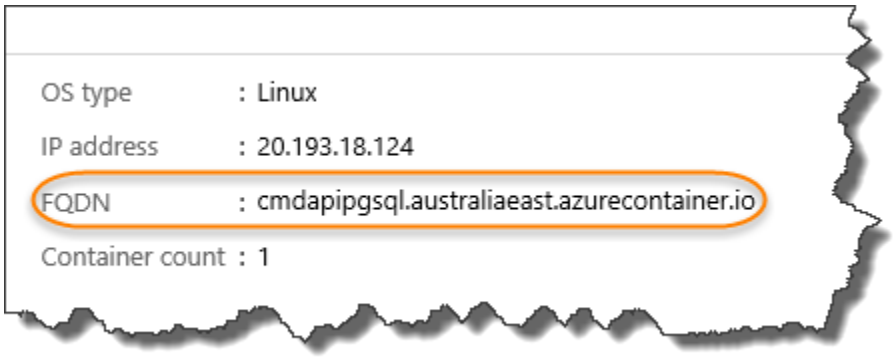
You’ll get notified when both your resources are set up: clicking on All resources, you can see everything we have created:



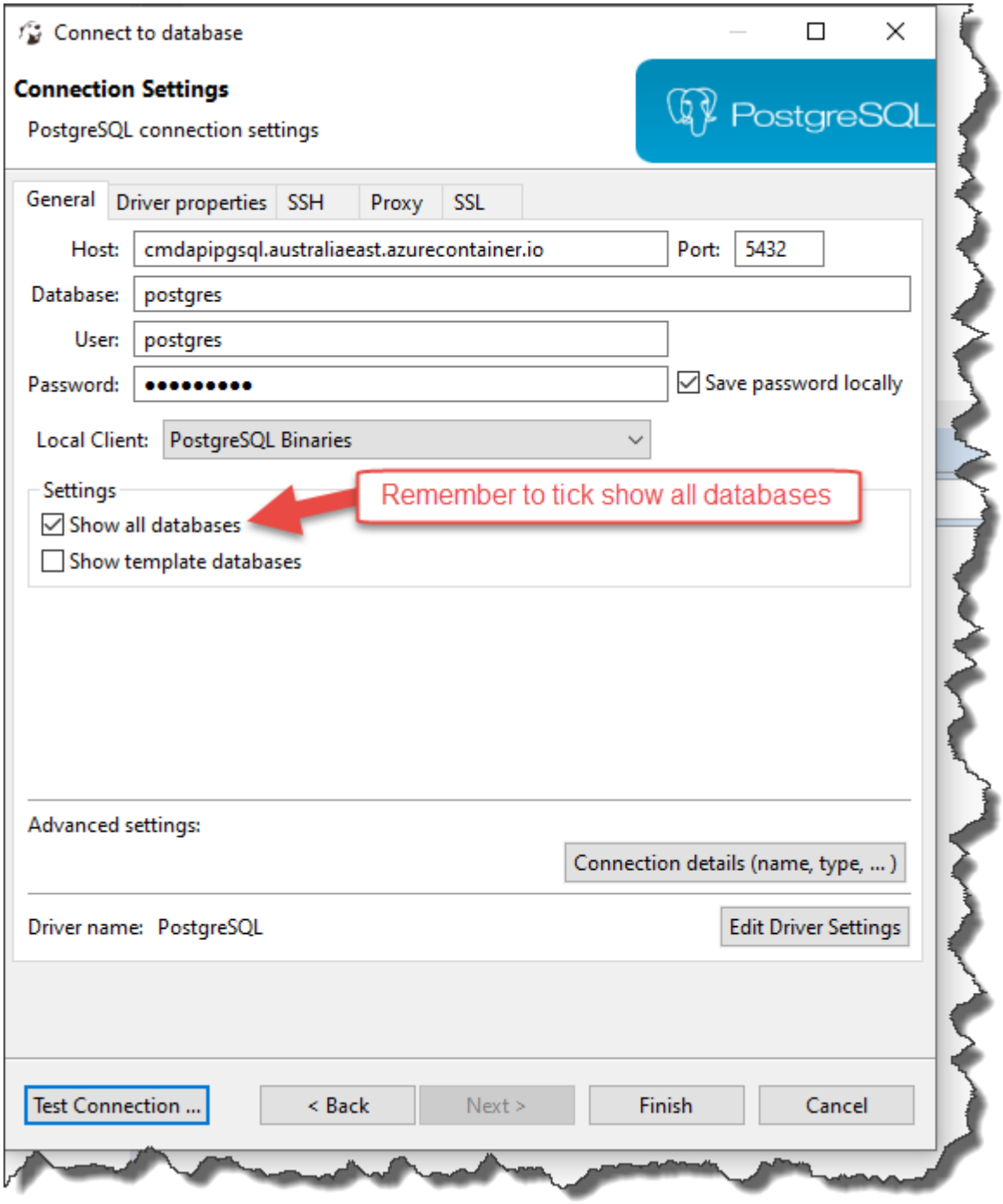
Connect & Create Our DB User

As before we want to create a dedicated user to connect in and use our database, the exercise is also a great opportunity to test that our PostgreSQL container instance is up and running.

First we need to get the Fully Qualified Domain Name, (FDQN), of the container instance, so in Azure find your container instance resource and select it, this will display a number of details, most important of which is our FDQN:



Make a note of the FDQN and move over to DBeaver, and create a new connection to a PostgreSQL instance – this is exactly the same as when we connected into our local instance, the only differences being the host and possibly the password for the postges user, (depending on what you set in the container instance environment variables):



You can test the connection or hit Finish to set up our connection to our Azure-based instance.

Again, we'll just repeat the user creation steps in chapter 6:

- Open a New SQL Editor Window
- Enter and run the following SQL, (you can change the password obviously!)

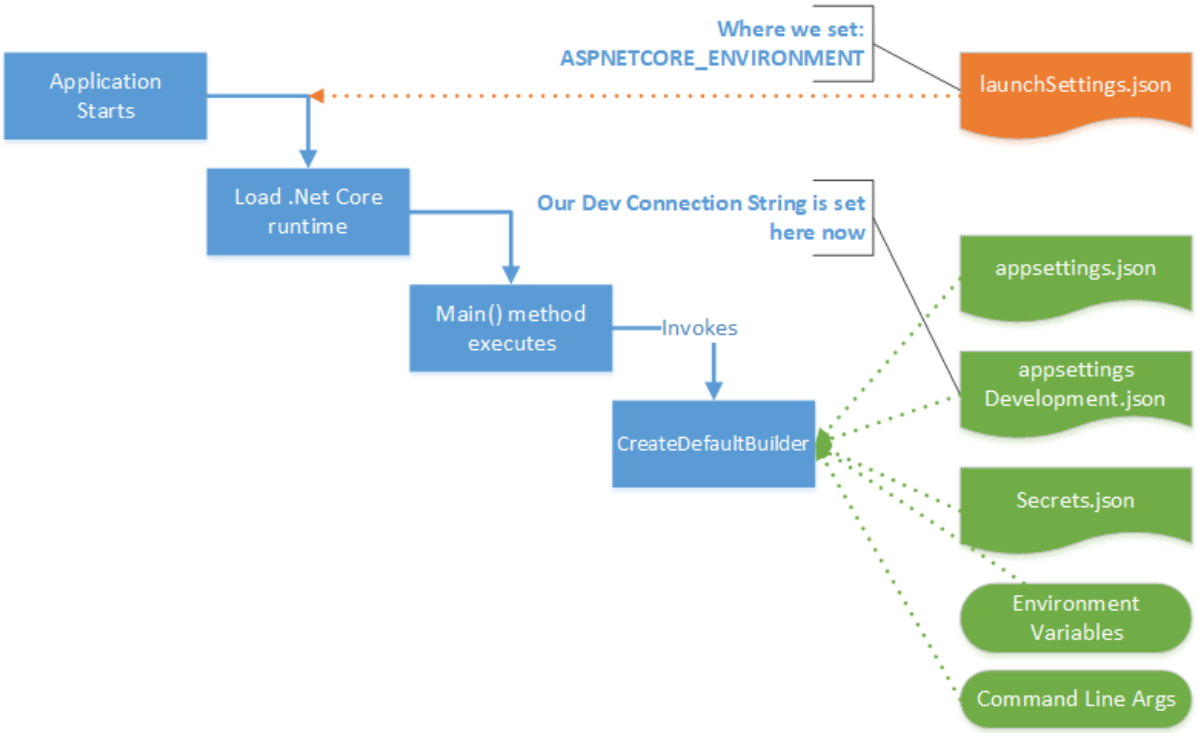
```
create user cmddbuser with encrypted password 'pa55w0rd!' createdb;
```

And again check that the role was created and that it has create database rights.

Along with the FDQN, set aside the user ID and password for later.

Revisit Our Dev Environment

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 in ***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`

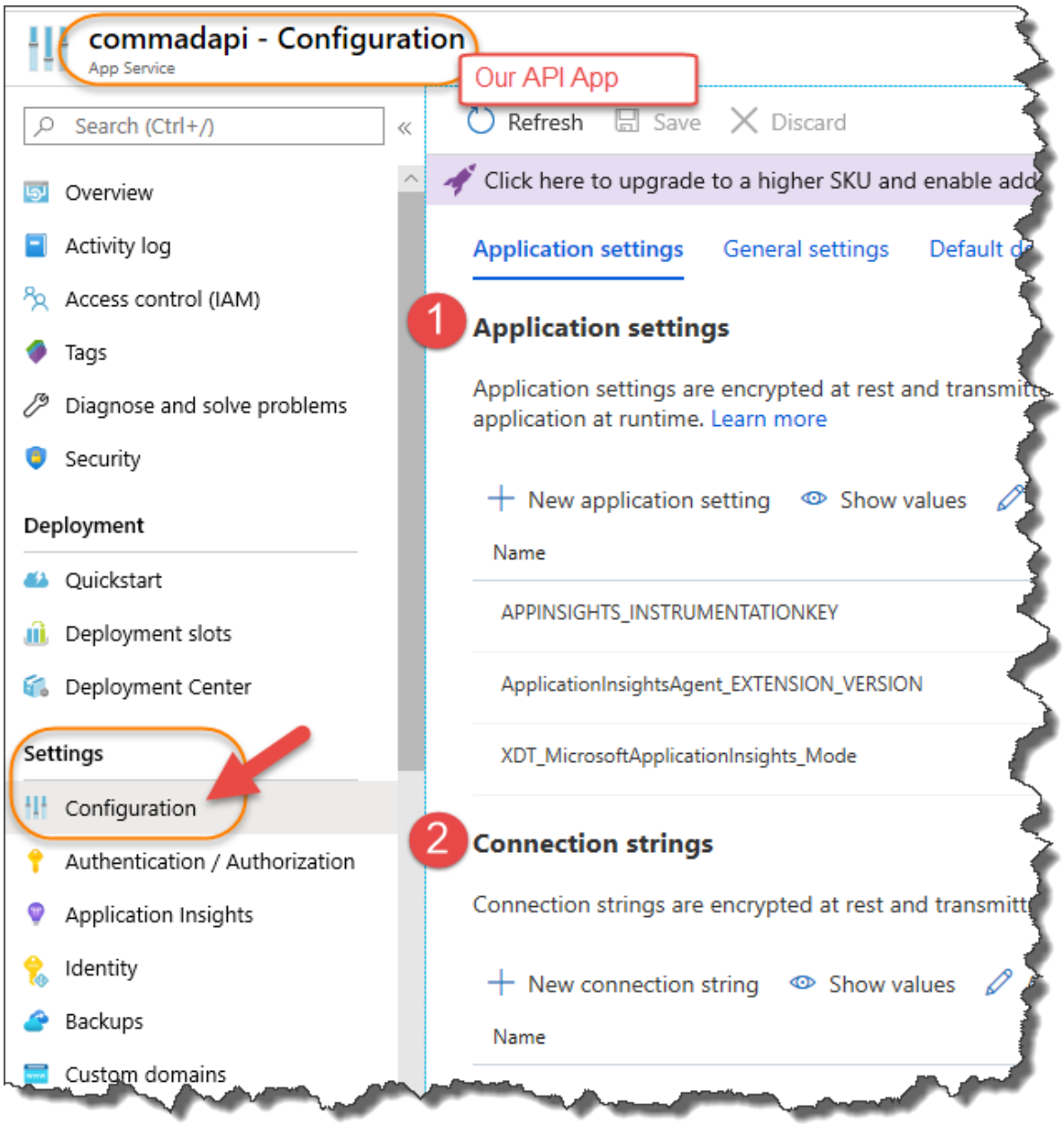
Setting Up Config In Azure

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. In setting up our production environment we 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.

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 our *Production Connection* string to the, (surprise, surprise), Connection Strings settings of our API App. Looking at the development connection string I have in ***appsettings.Development.json***:

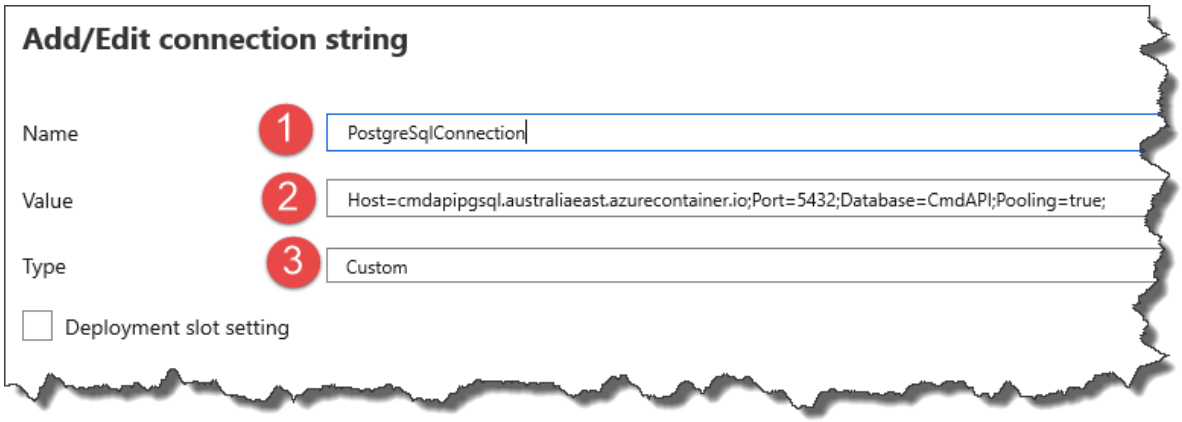
```
Host=localhost;Port=5432;Database=CmdAPI;Pooling=true;
```

Not that much needs to change, except the Host attribute, so we simply substitute that for either our PostgreSQL container instance IP address of FDQN, so we now have:

```
Host=pgserver.australiaeast.azurecontainer.io;
Port=5432;Database=CmdAPI;Pooling=true;
```

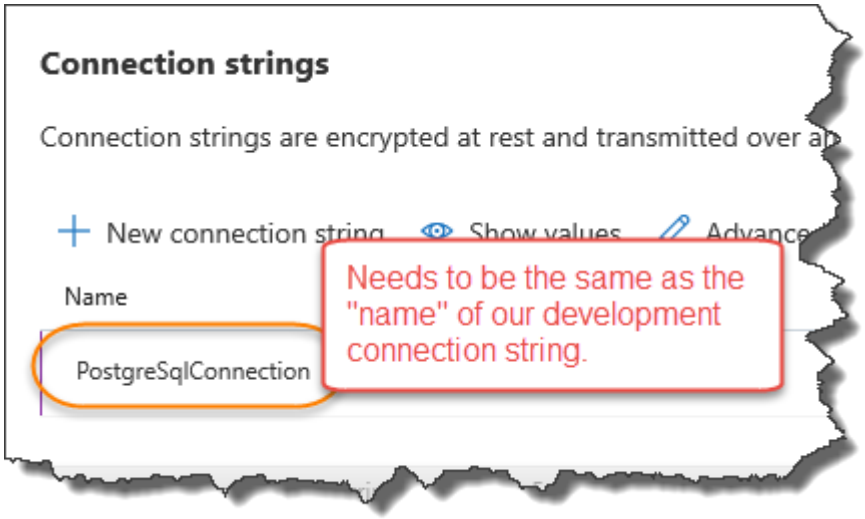
To add this string to our API App Connection String settings, click: *+ New connection string*, in the resulting form enter:

1. Connection String Name, (**this should be the same name as our development connection string**)
2. The connection string we obtained above, (note we'll be configuring our User ID and Password separately)
3. Set the type to Custom



Warning! You do have the option of "PostgreSQL" for the connection string type – however I had significant issues using this – so use it at your peril!

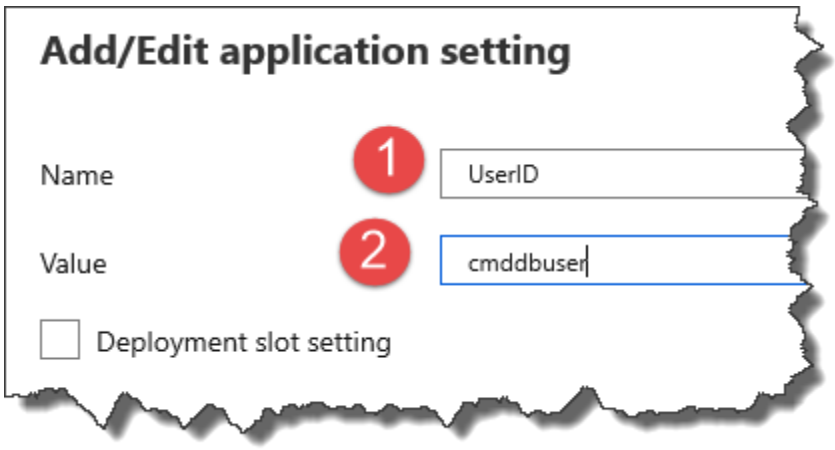
Click OK, and you'll see the connection string has been added to our collection:



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:

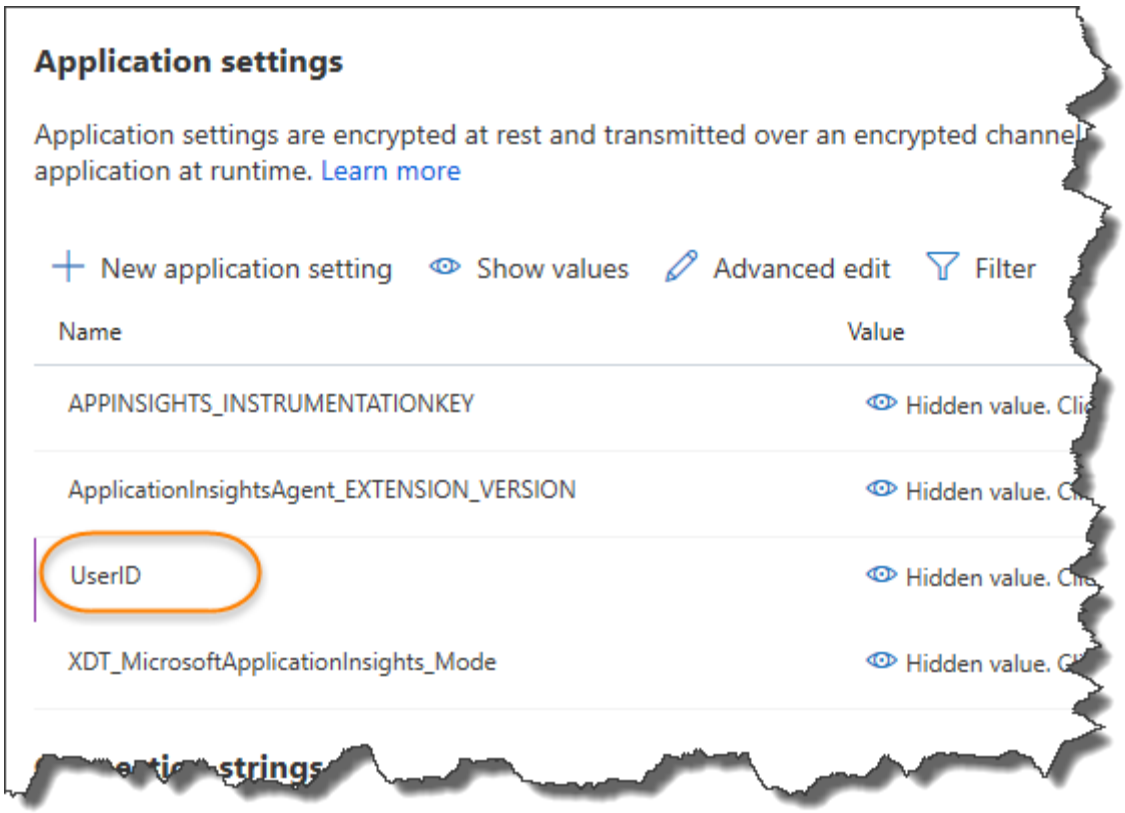
1. Name of our setting, (this should be the same as our User Secret name for User ID)
2. Value. This is the user account you created when you set up the SQL Server



Again, just be careful that the User ID attribute is exactly the same as the local user secret, and what your app is expecting to ingest when it creates the connection string:

```
public void ConfigureServices(IServiceCollection services)
{
    var builder = new NpgsqlConnectionStringBuilder();
    builder.ConnectionString = Configuration.GetConnectionString("PostgreSqlConnection");
    builder.Username = Configuration["UserID"];
    builder.Password = Configuration["Password"];
    services.AddDbContext<CommandContext>(opt => opt.UseNpgsql(builder.ConnectionString));

    services.AddControllers();
```

Click OK, and you'll see the new UserID application setting:



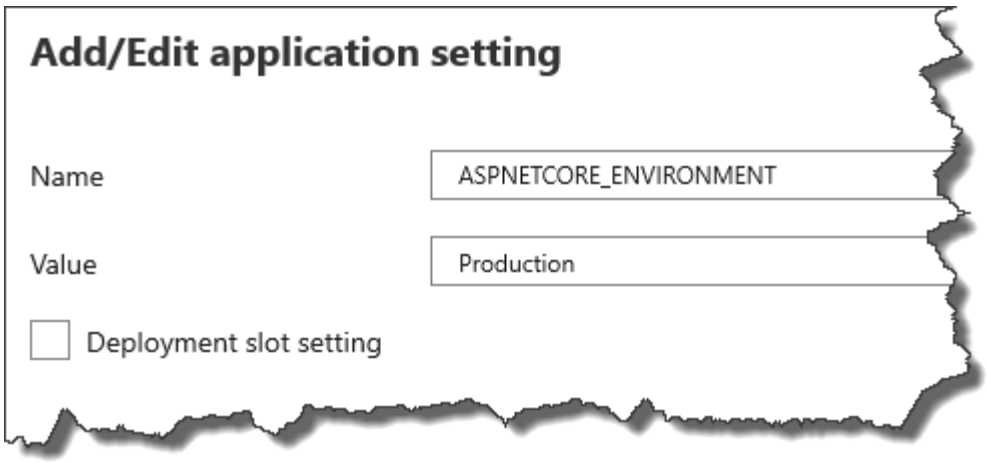
Learning Opportunity: Add a second *Application setting* for our ***Password***. This should follow the same process as UserID.

Configure Our Environment

We want to set our runtime environment to "Production", we do this simply by adding another Application setting with:

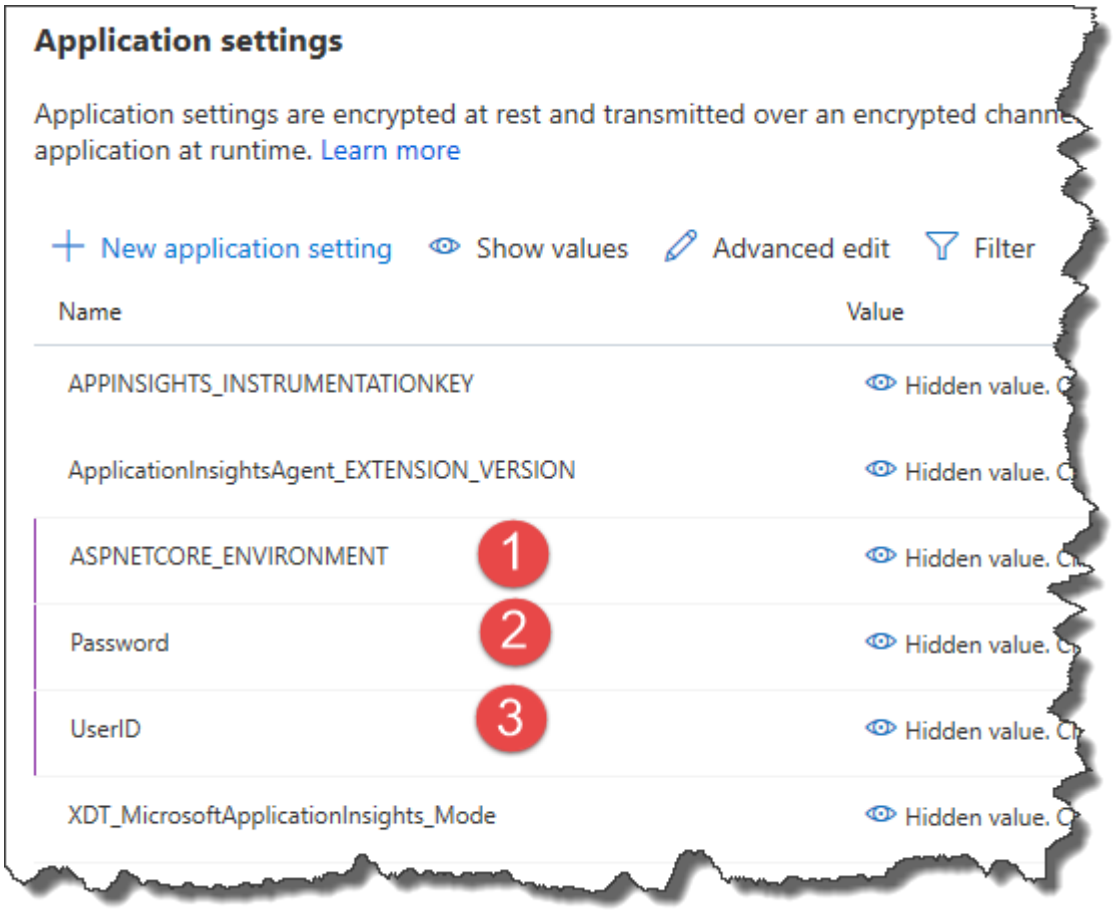
- A Name of: ASPNETCORE_ENVIRONMENT
- A Value of: Production

As shown below:

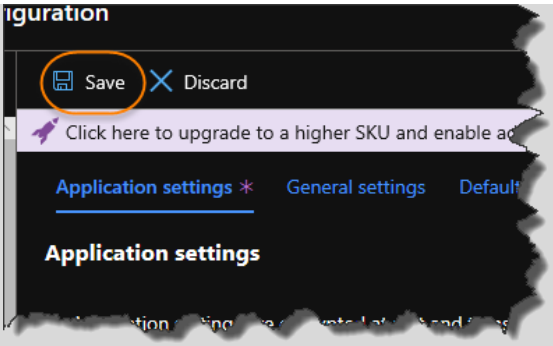


Click OK and you should now have:

1. Application settings: ASPNETCORE_ENVIRONMENT
2. Application settings: Password
3. Application settings: UserID
4. Connection string: PostgrsSQLConnection



Warning! Every time you make a configuration change you need to save it - See below:

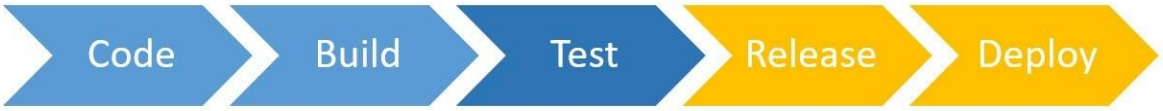


Make sure you click Save to apply your changes (when starting out with this stuff I didn't and spent a lot time trying to understand what as wrong!)

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.



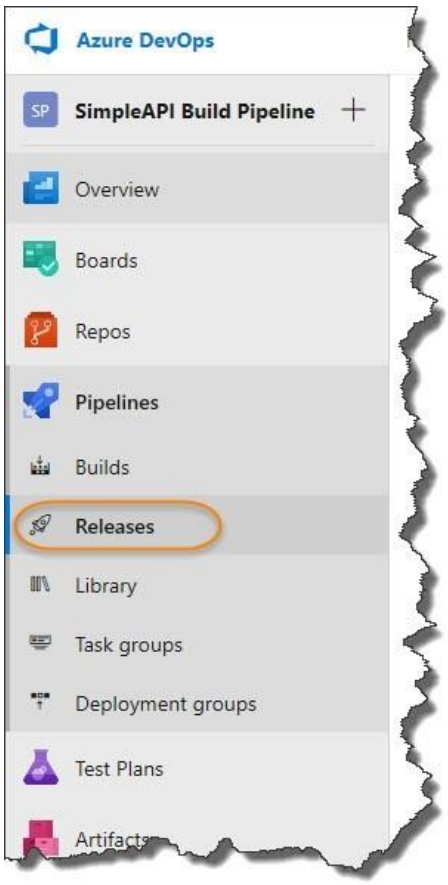
A quick recap on our CI/CD Pipeline so far:

- We created what Azure DevOps calls a *Build Pipeline* that does the following:
  - Builds Our Projects
  - Runs our unit tests
  - Packages our release

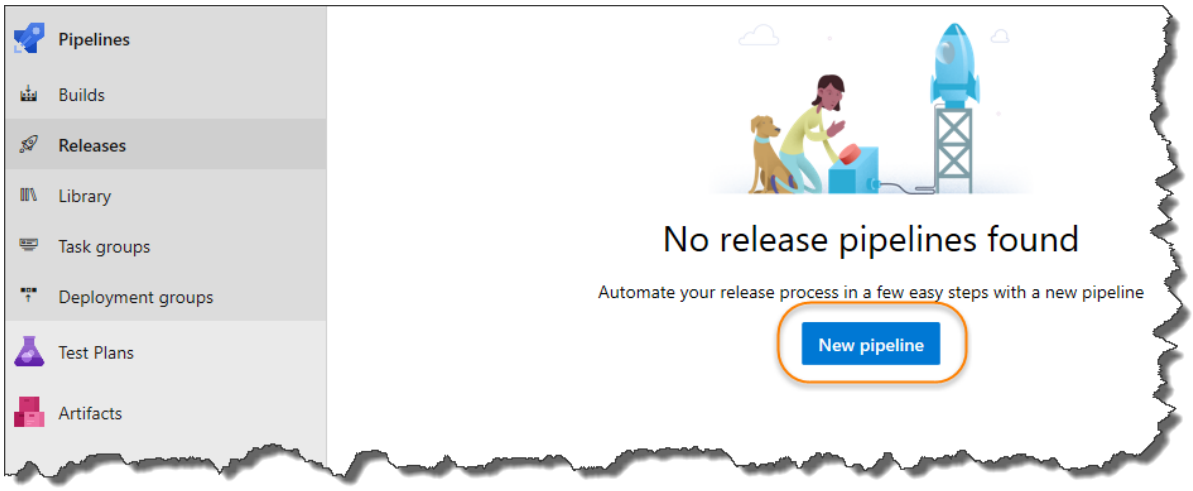
What we now need to do in Azure DevOps is create a *Release Pipeline* that takes our package release and deploys it to Azure. So basically our full CI/CD Pipeline = Azure DevOps **Build** Pipeline + Azure DevOps **Release** Pipeline.

Creating Our Azure DevOps Release Pipeline

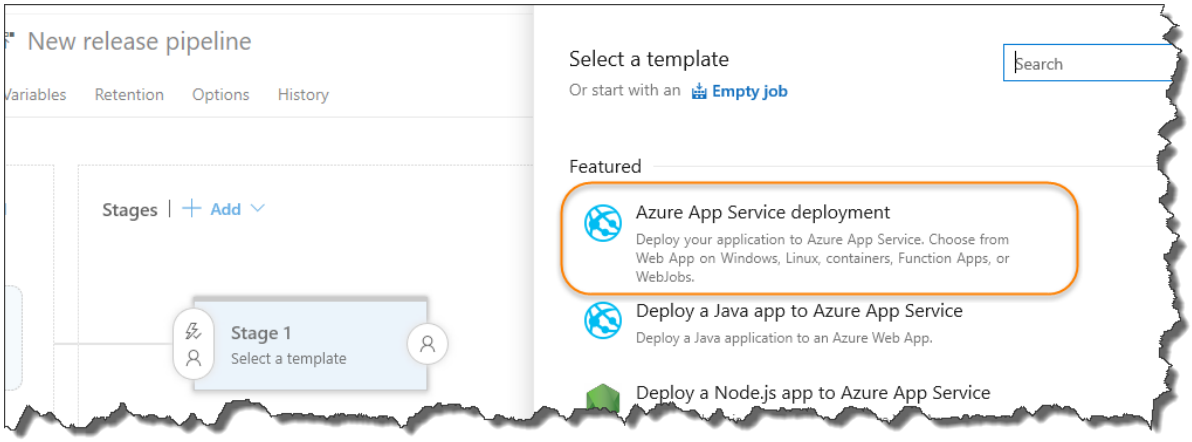
Back in Azure DevOPs, click on Pipelines -> Releases



The click on New Pipeline:

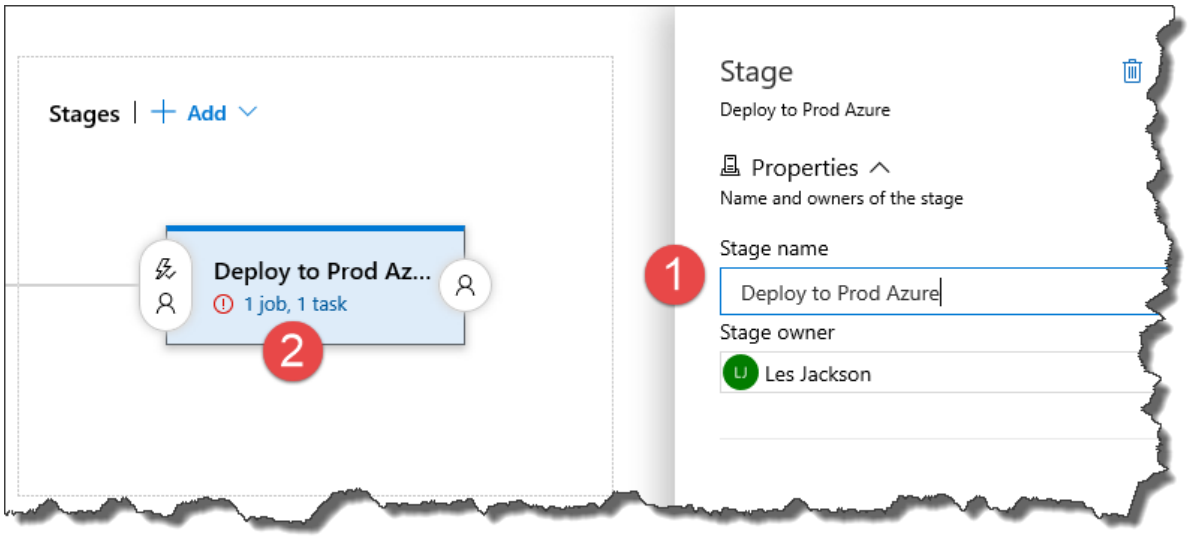


On the next screen, select and *Apply* the *Azure App Service deployment* template:



In the "Stage" widget:

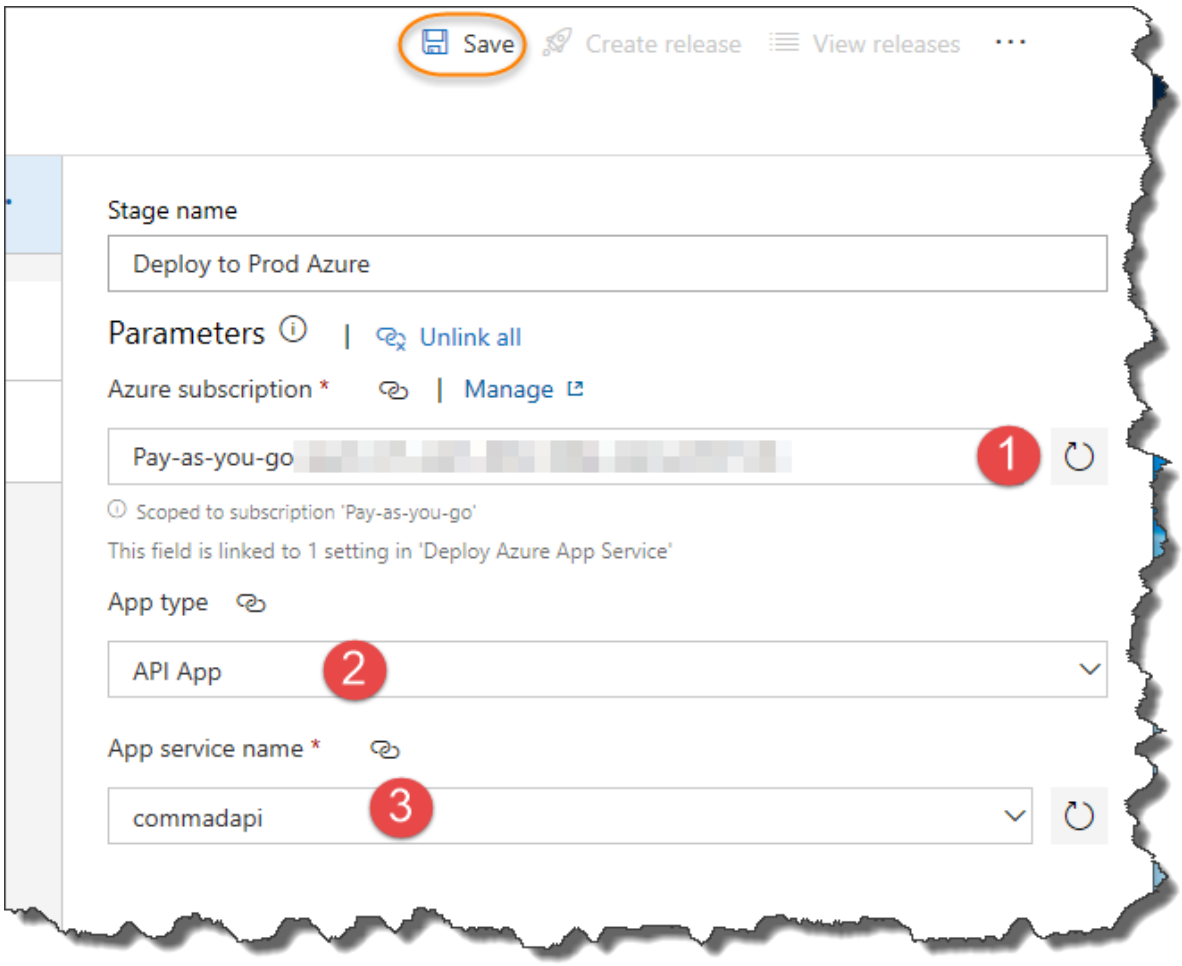
1. Change the stage name to: "Deploy API to Prod Azure"
2. Click on the Job / Task link in the designer



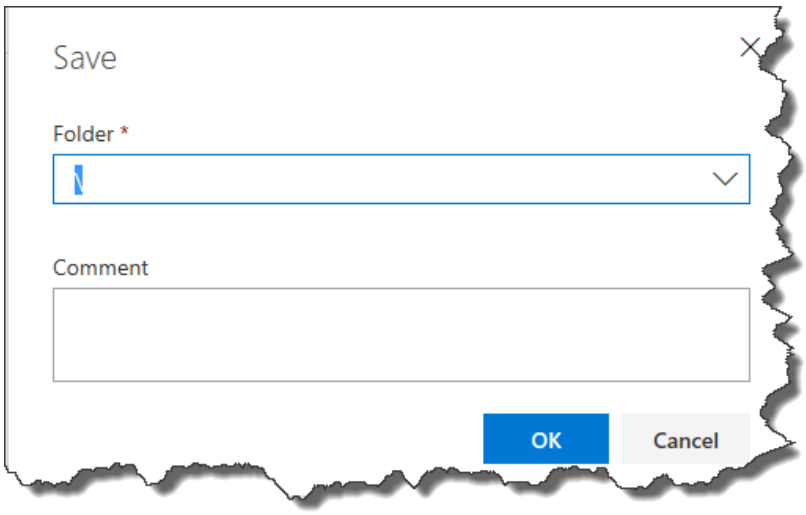
Here we need to:

1. Select Our Azure Subscription (you will need to "authorise" Azure DevOps to use Azure[23])
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)

[23] Be careful if you have a pop up blocker in place as the authorisation workflow opens a new window.

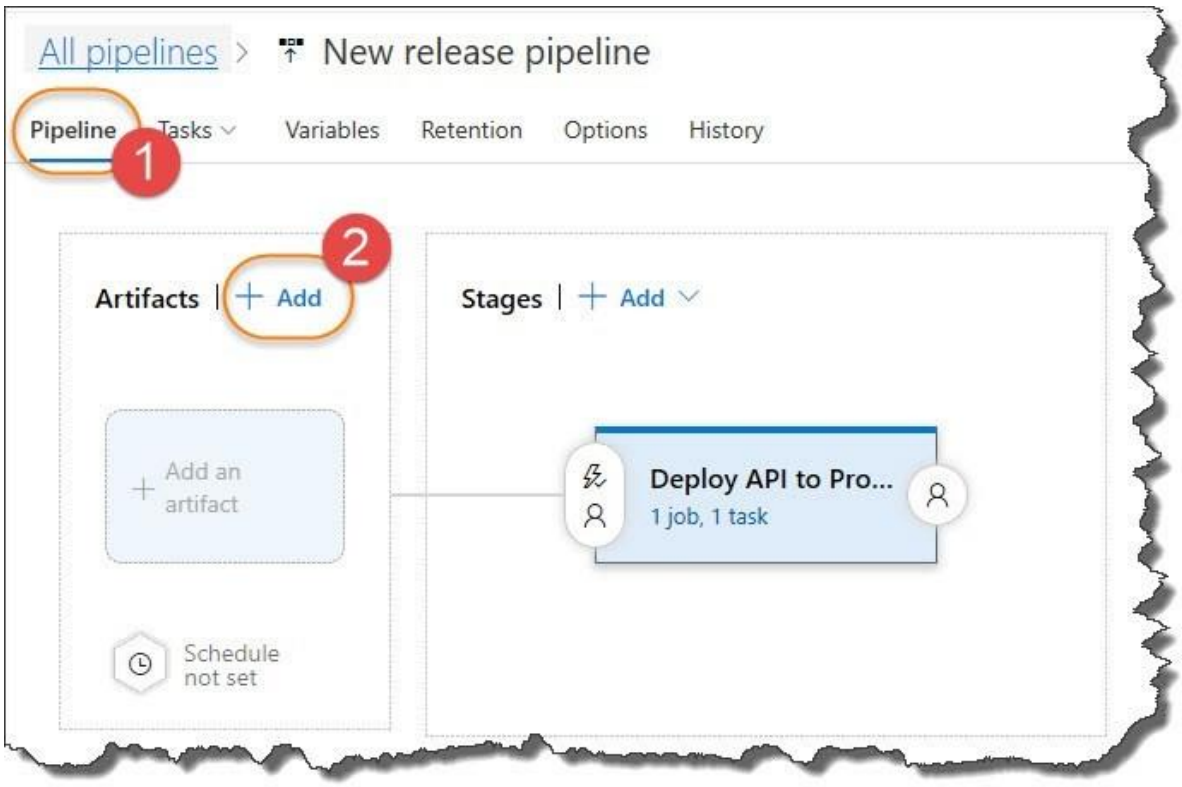


Don't forget to *Save*. When you do you'll be presented with:



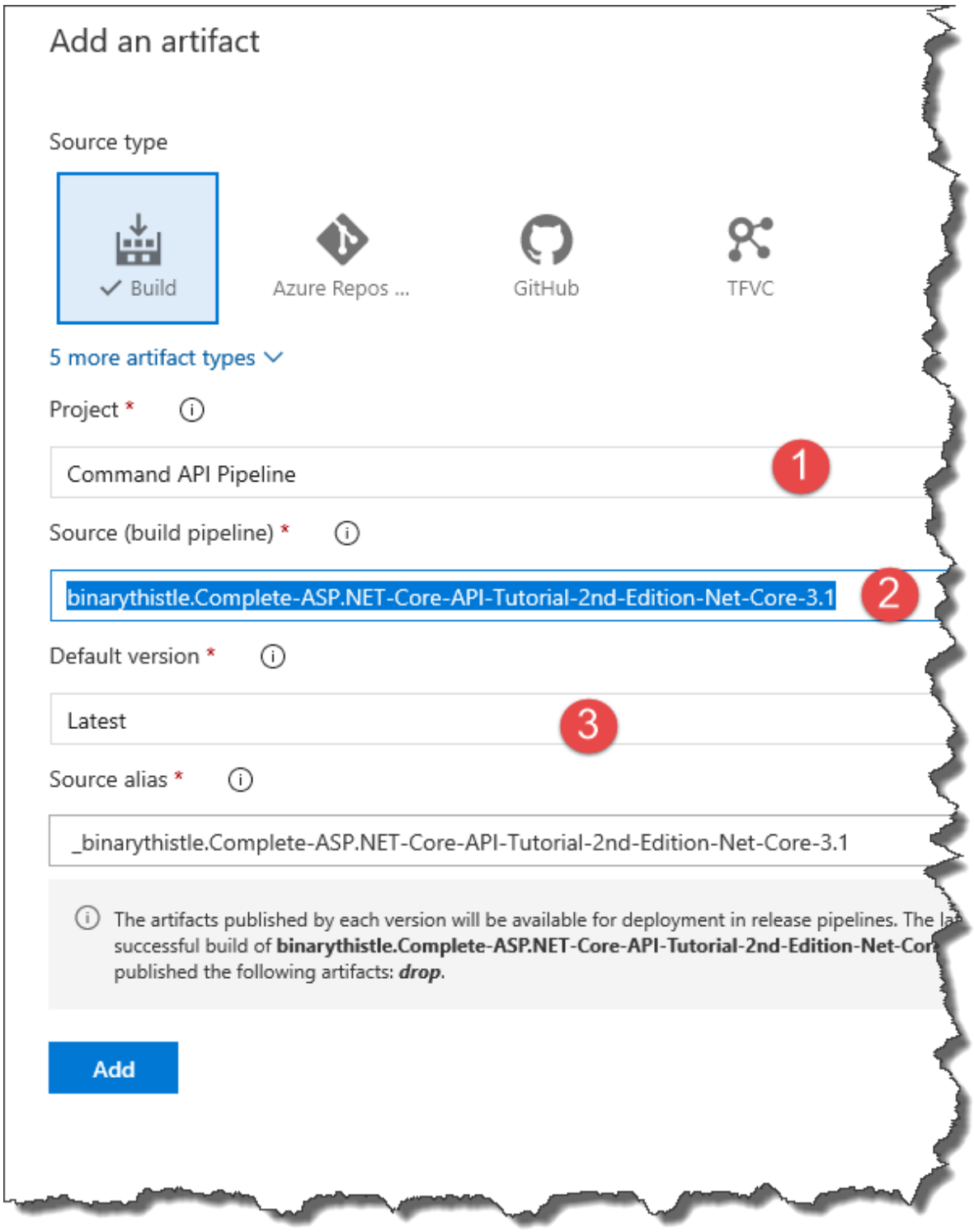
Just click OK.

Click back on the "Pipeline" tab, then on Add (to add an artefact):

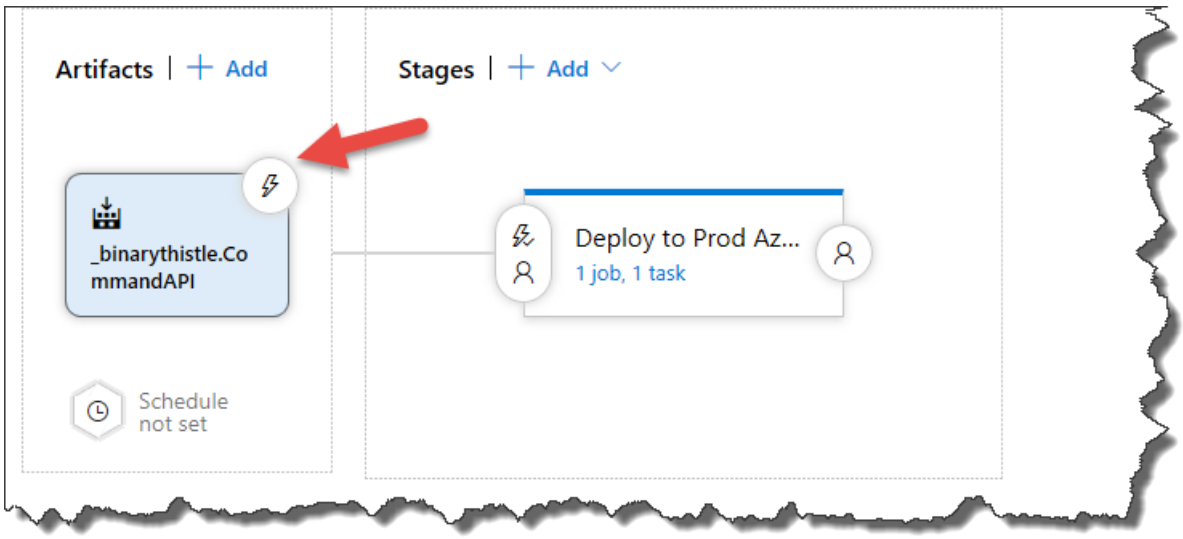


Here you will need to provide:

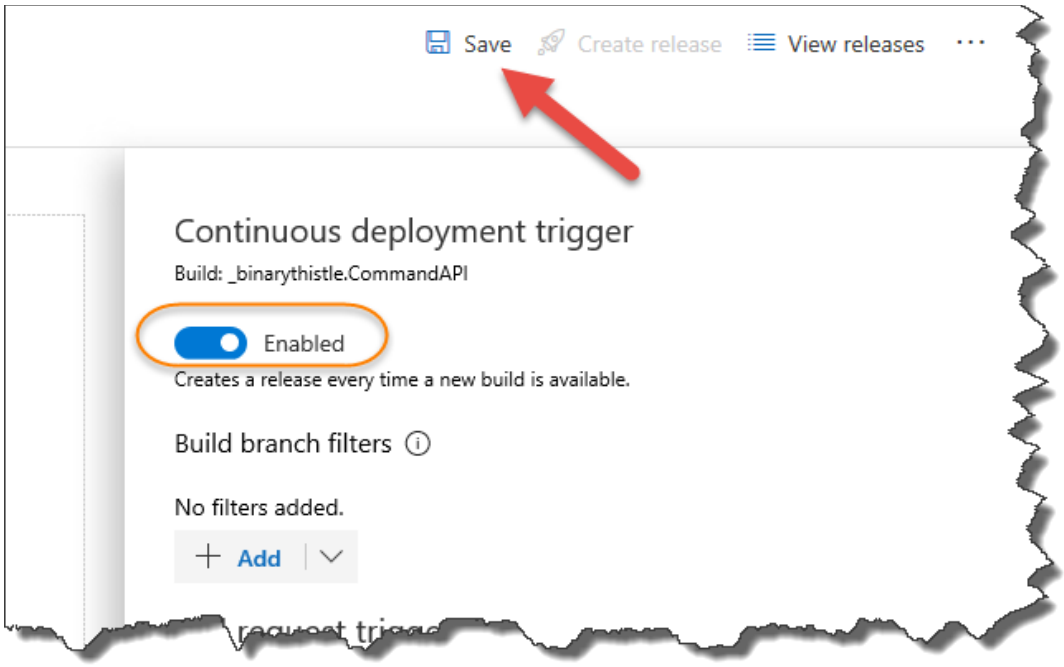
1. The Project (this should be pre-selected)
2. The Source Pipeline (this is our build pipeline we created previously)
3. Default version (select “Latest” from the drop down)



Click Add. Then click on the lightening bolt on the newly created Artefact node:

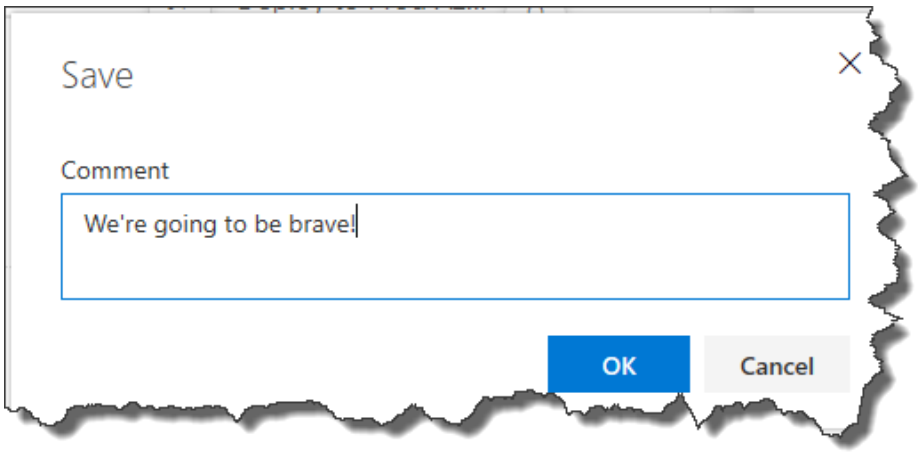


In the resulting pop up, ensure that Continuous deployment trigger is **enabled**, then click *Save*.

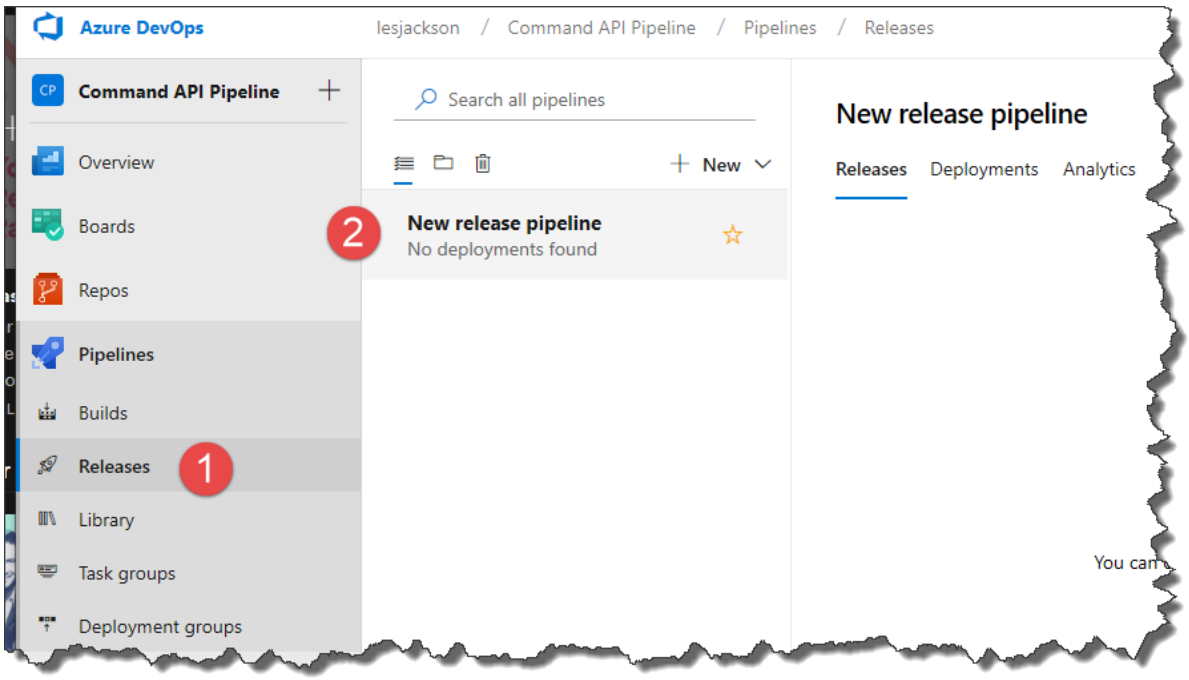


Note: it is this setting that switches us from Continuous Delivery to Continuous Deployment…

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 PostgreSQL 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.

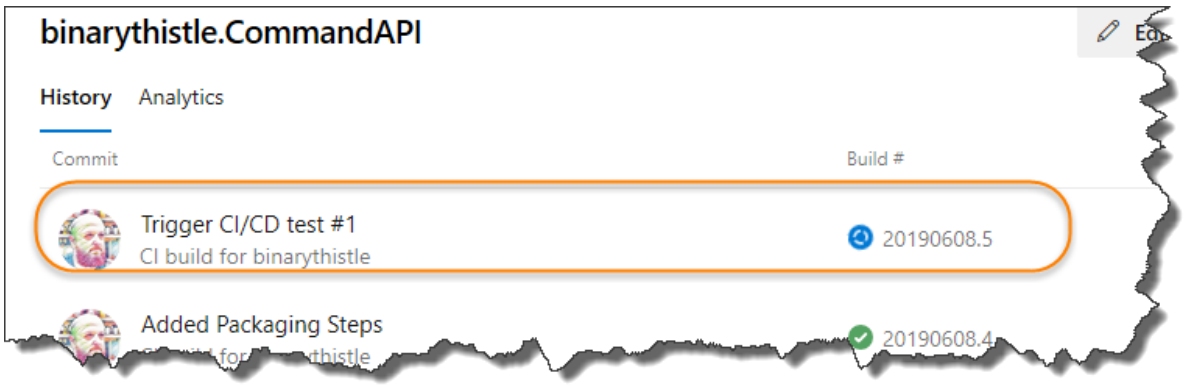
In VS Code, open the `Startup` class and make the following alterations to the `Configure` method:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
CommandContext context)
{
  context.Database.Migrate();
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }
  app.UseMvc();
}
```

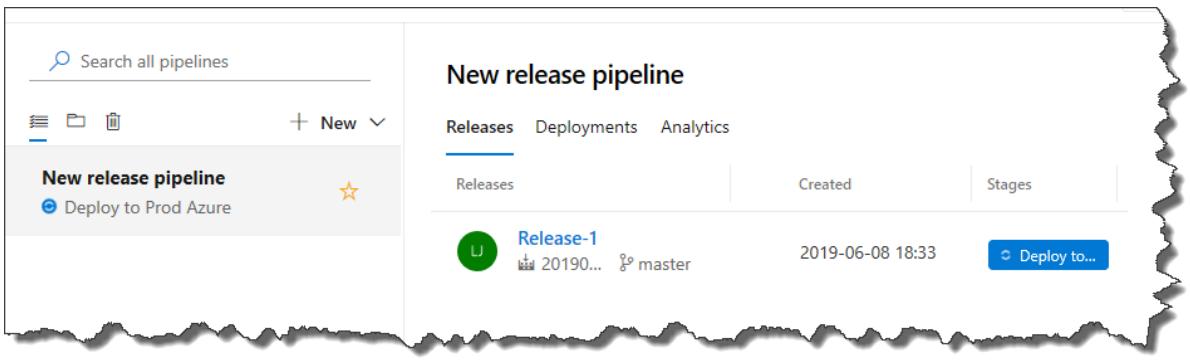
For clarity, the `Configure` method changes are highlighted below:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env, CommandContext context)
{
    context.Database.Migrate();
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
```

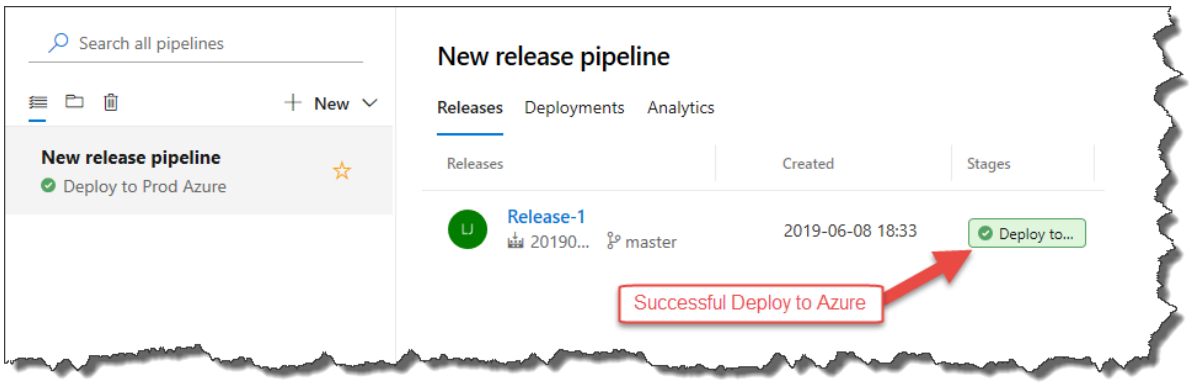
Save your changes and: Add, Commit and Push your code as usual, this should trigger the build pipeline…



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):

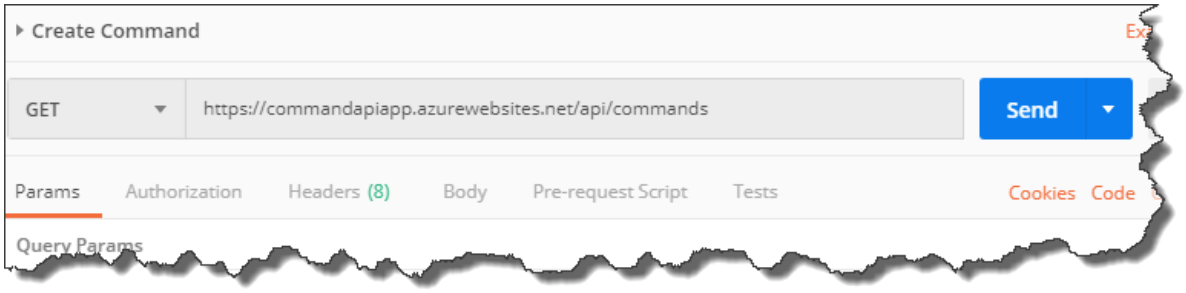


And now the moment of truth, let's see if our API is working, first obtain the base app URL from Azure:

- Click All resources
- Select you API App (App Service type)

Note: yours will be named differently…

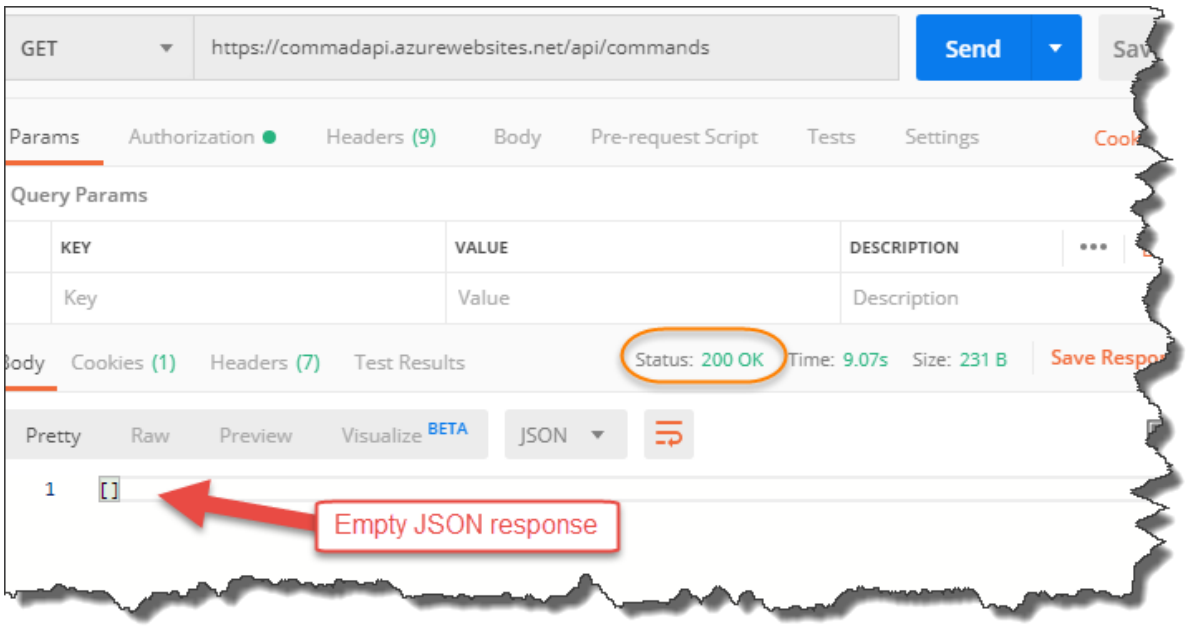
Now fire up Postman, and prepare to make a GET request to retrieve all our commands (we won't have any yet):



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 and an OK 200 Status:



Celebration Check Point: Rad[24]! 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.

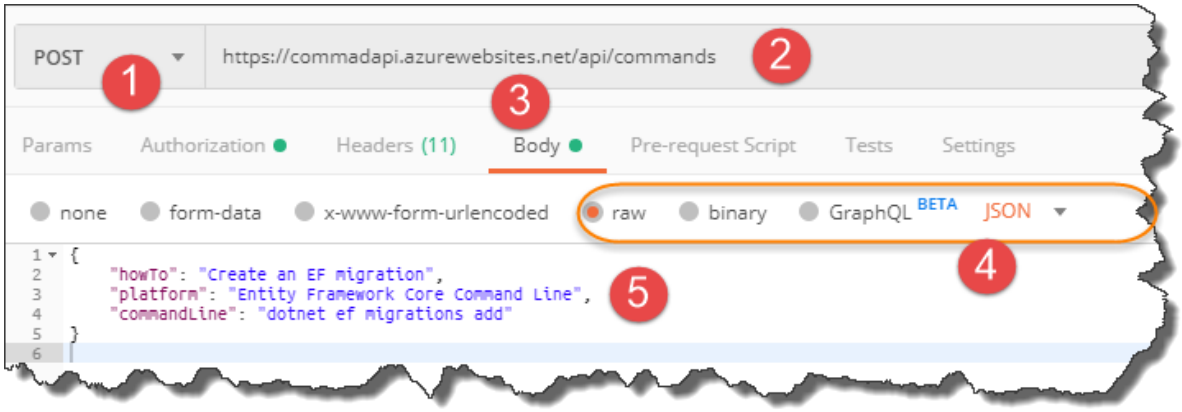
Using the JSON string below:

```
{
    "howTo": "Create an EF migration",
    "platform": "Entity Framework Core Command Line",
    "commandLine": "dotnet ef migrations add"
}
```

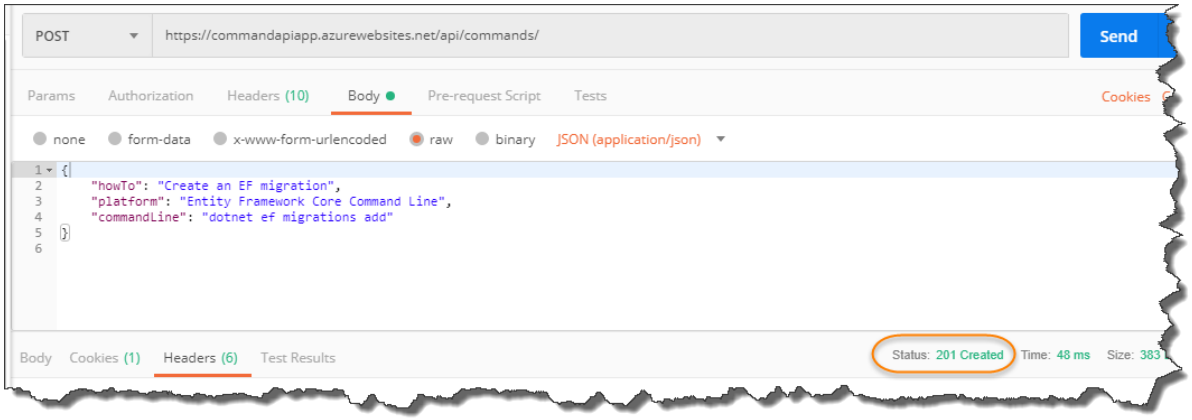
Create a new Postman request and set:

1. Request Verb to POST
2. The request URL is correct (e.g.: https://commadapi.azurewebsites.net/api/commands)
3. Click Body
4. Select Raw and JSON for the request body format
5. Paste the JSON into the body payload window

[24] Children of the 90s' will get this superlative



Finally, if you're brave enough click "Send" to make the request.



And again we have success!

Chapter 11 – Securing Our API

Chapter Summary

In this chapter we discuss how we can secure our API, specifically we'll add the "Bearer" authentication scheme into the mix that will allow only authorised clients to access our API resource through use of Tokens.

When Done, You Will

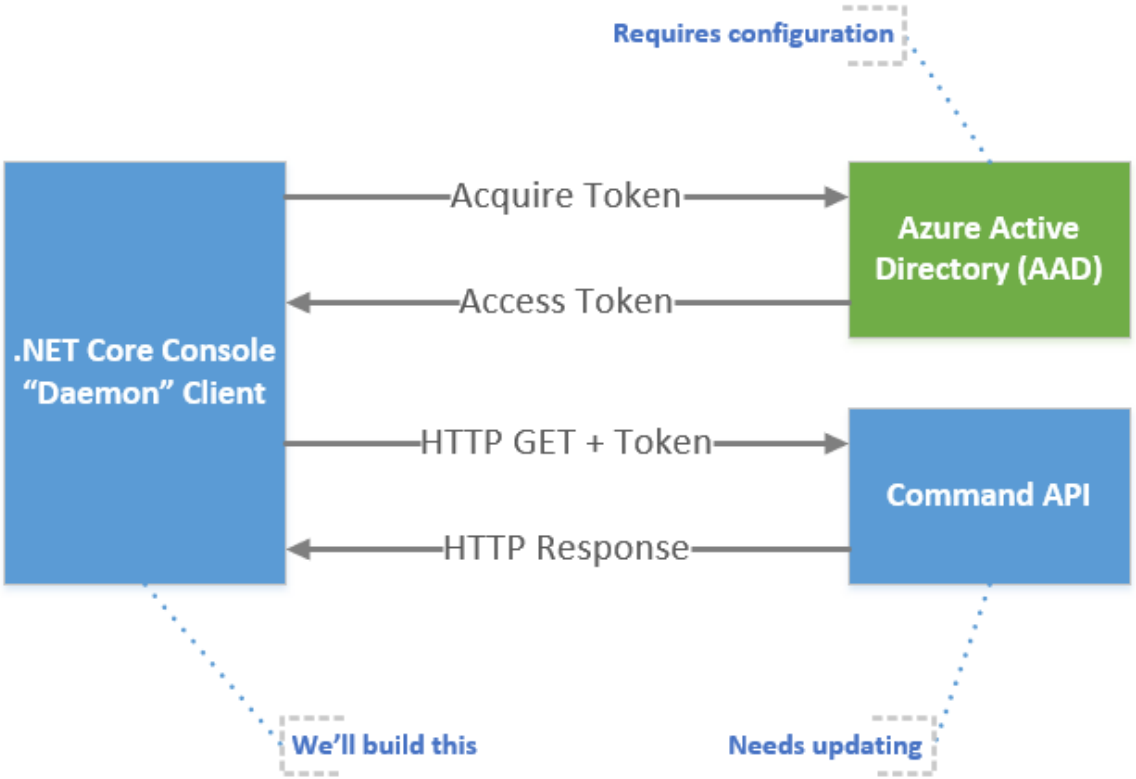
- Understand the Bearer authentication scheme
- Use Azure Active Directory to secure our API
- Create a simple client that is authorised to use the API
- Deploy to Azure

We have a lot to cover – so let's get going!

What We're Building

Our Authentication Use Case

Before delving into the technicalities of our chosen authentication scheme, I just wanted to cover our authentication *use case*. For this example we are going to "secure" our API by using Azure Active Directory, (AAD), and then create and configure a client, (or daemon), app with the necessary privileges to authenticate through and use the API. We are *not* going to leverage "interactive" user-entered User Ids and passwords. This use case is depicted below:



Overview of Bearer Authentication

There are a number of authentication schemes that we could have used, a non-exhaustive list is provided below:

Scheme	Description
Basic	A common, relatively simple authentication scheme. Requires the supply of a user name and password that's then encoded as a Base64 string, this is then added to the authorisation header of a http request. Natively this is not encrypted so is not that secure, unless you opt so make requests over https, in which case the transport is encrypted.
Digest	Follows on from Basic Authentication, but is more secure as it applies a hash function to any sensitive data, (e.g. username and password), before sending.
Bearer	Token based authentication scheme where anyone in possession of a valid "token" can gain access to the associated secured resources, in this case our API. Considered secure, it is widely adopted in industry and is the scheme, (specified in RFC 6750), we'll use to secure our API.
NTLM	Microsoft-specific authentication scheme, using Windows credentials to authenticate. Perfectly decent, secure scheme but as it's somewhat "proprietary", (and I'm trying to avoid that), we'll leave our discussion there for now.

Bearer Token Vs JWT

The use of "tokens" in Bearer authentication is a central concept. A token is issued to a requestor, (in this case a daemon client), and the client, (or "bearer of the token"), then presents it to a secure resource in order to gain access.

So what's JWT?

JWT, (or JSON Web Tokens), is an encoding standard, (specified in RFC 7519), for tokens that contain a JSON payload. JWT's can be used across a number of applications, however in this instance we're going to use JWT as our encoded token through our use of Bearer authentication.

In short:

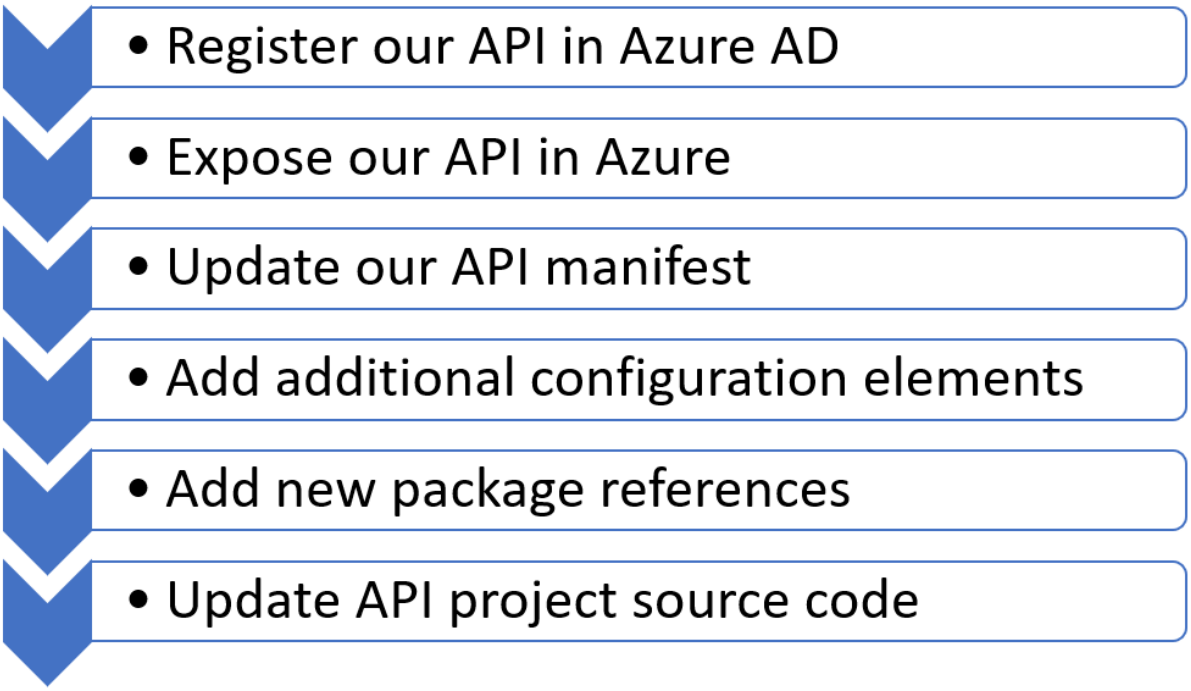
- Bearer authentication is the authentication scheme that makes use of, (bearer), "tokens".
- JWT is a specific implementation of bearer tokens, in particular those with a JSON payload.

Again, rather than dwelling on copious amounts of theory, the concepts will make more sense as we build them below.

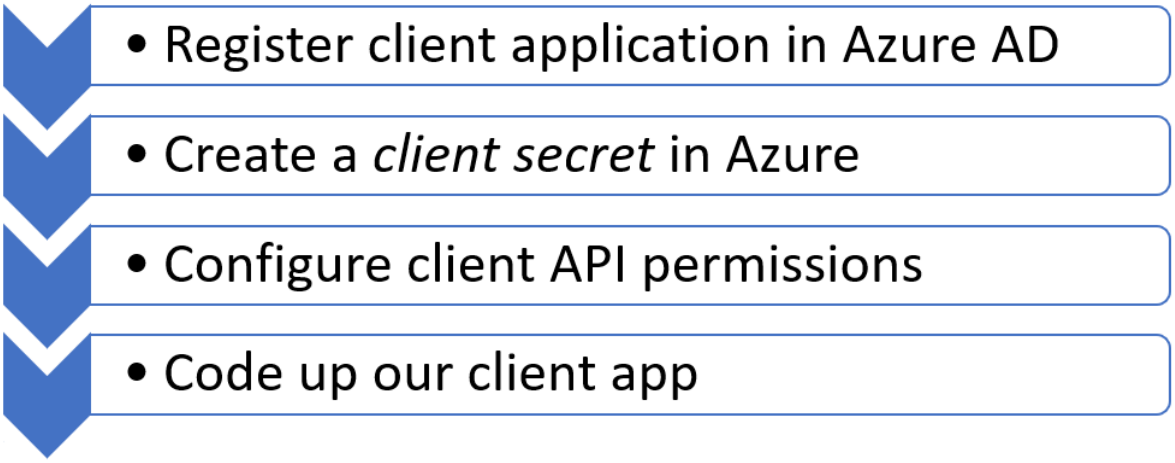
Build Steps

As I've mentioned before, I like a bit of 50,000ft view of what we're going to build before we start building it as it helps contextualise what we need to do, and it also allows us to understand the progress we're making. Therefore in terms of the configuration and coding we need to perform, I've detailed the steps we'll follow below:

Steps for our API Project



Steps for our Daemon Client



Phew! You can see there is actually a lot to do – so let's get on it!

Registering our API in Azure AD

The first thing we need to do is register our API with Azure Active Directory, (AAD), as we're using AAD as our Identity and Access Management *directory service*.

Les's Personal Anecdote: One of my first jobs out of university was as part of a team supporting a large, (I believe at the time the 2nd largest in the world), deployment of Novell NetWare Directory Services, (NDS). Which was weird as I had neither the background, nor inclination to learn NDS…

Anyhow, this product was considered relatively leading-edge at the time as it took the approach of storing user accounts, (as well as other "organisational objects"), in a hierarchical directory tree structure that was both distributed and replicated, (in this case), nationwide. In short it was hugely scalable and could cater for 10,000's, (we had well over 100,000), of user accounts.

At the time Microsoft only used Windows NT Domains which were arguably more basic, (they were "flat"), less scalable, distributable and reliable than their NetWare counterparts… Blue screen of death anyone?

Microsoft were obviously, cough, "inspired", cough again, by NDS, (and Banyan Vines – see below), to such an extent that they brought out a rival product: Active Directory, which bore a remarkable resemblance to, drum roll: NDS… You could argue this was poetic justice as Novell had ~~stolen~~ been "inspired" by an earlier product called Banyan VINES. Interestingly, Jim Allchin, engineering supremo at Banyan joined Microsoft due to creative and strategic differences with the Banyan leadership…

The rest is history.

Banyan and Novell's products withered and died due to a number of different strategic missteps, as well as the fact that Microsoft had a compelling value proposition.

So, if you use a Windows PC at work and have to "login", then you're most likely logging into an Active Directory. Now with the emergence of Azure, you don't even need to host you AD on-premise, and can opt to use Azure Active Directory, which is what we'll be using for this chapter.

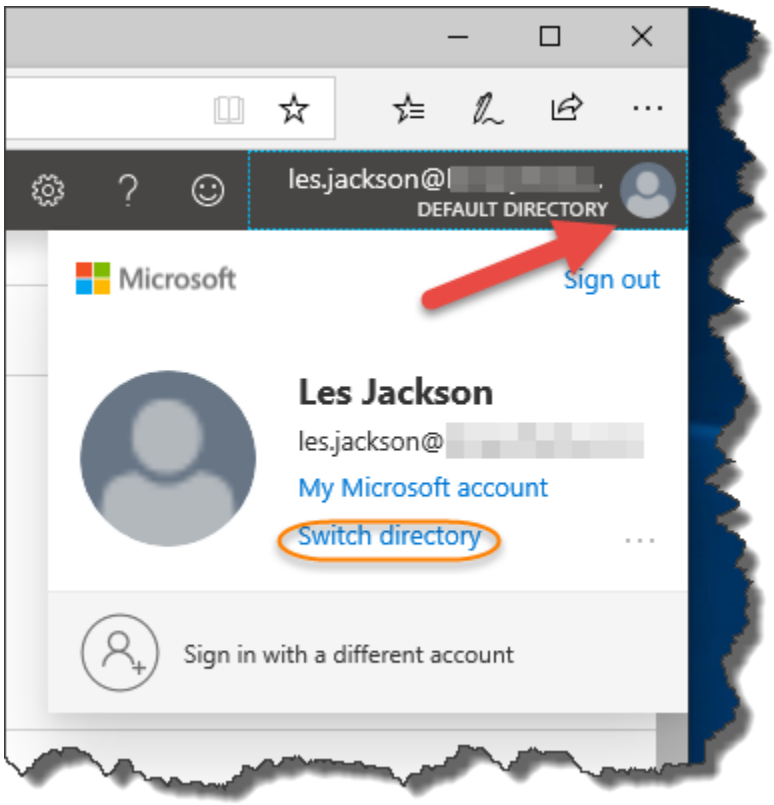
Create a New AD?

Now this step is optional, but I have created a "test" AAD in addition to the AAD that gets created when you sign up for Azure. This is really just to ring-fence what is in essence my "production AAD", (the one that holds my login for Azure), from any development activities I undertake.

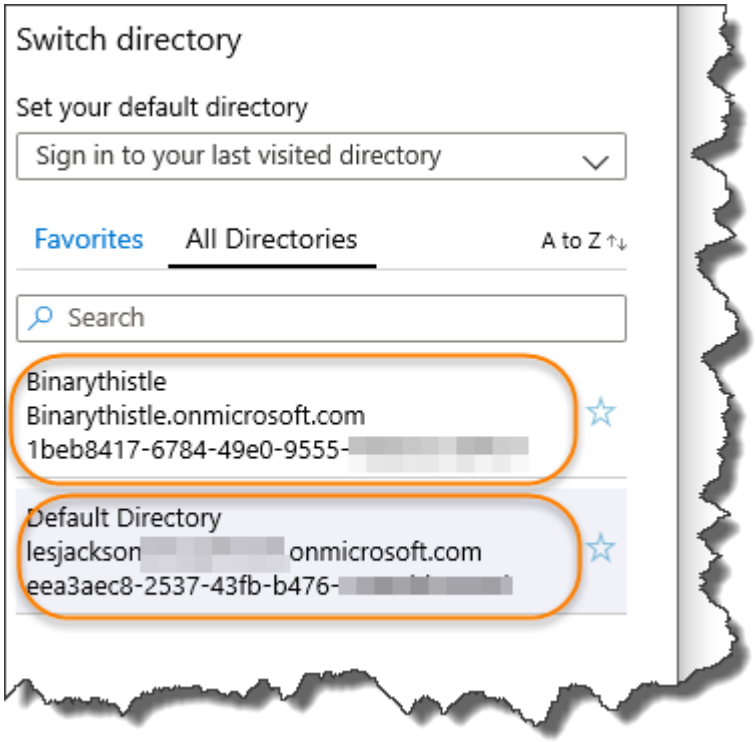
You can create a new AAD in exactly the same way as you create any other resources in Azure, so I won't detail the steps here. If you do opt for this approach though, (it is optional), the only thing you need to be aware of is that when you want to create objects in your "Development AAD", you'll need to switch to it the Azure Portal.

Switching Between AADs

To switch between your AADs, click on the person icon at the top right hand of the Azure Portal:

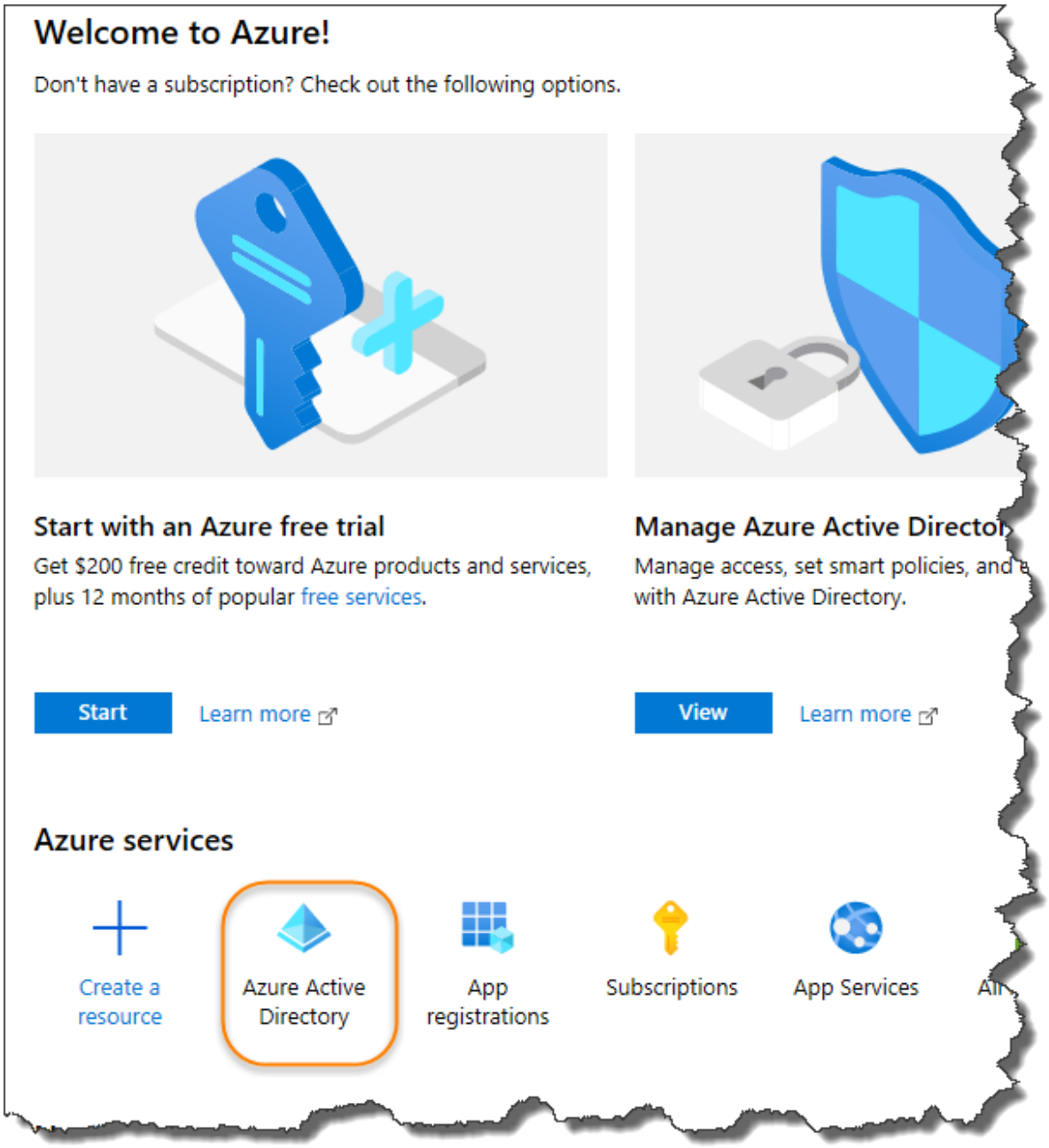


On the resulting pop up, you can then click Switch Directory, (see circled above), you should then get the option to select and switch between the AAD's you have, (I have 2 as you can see below):

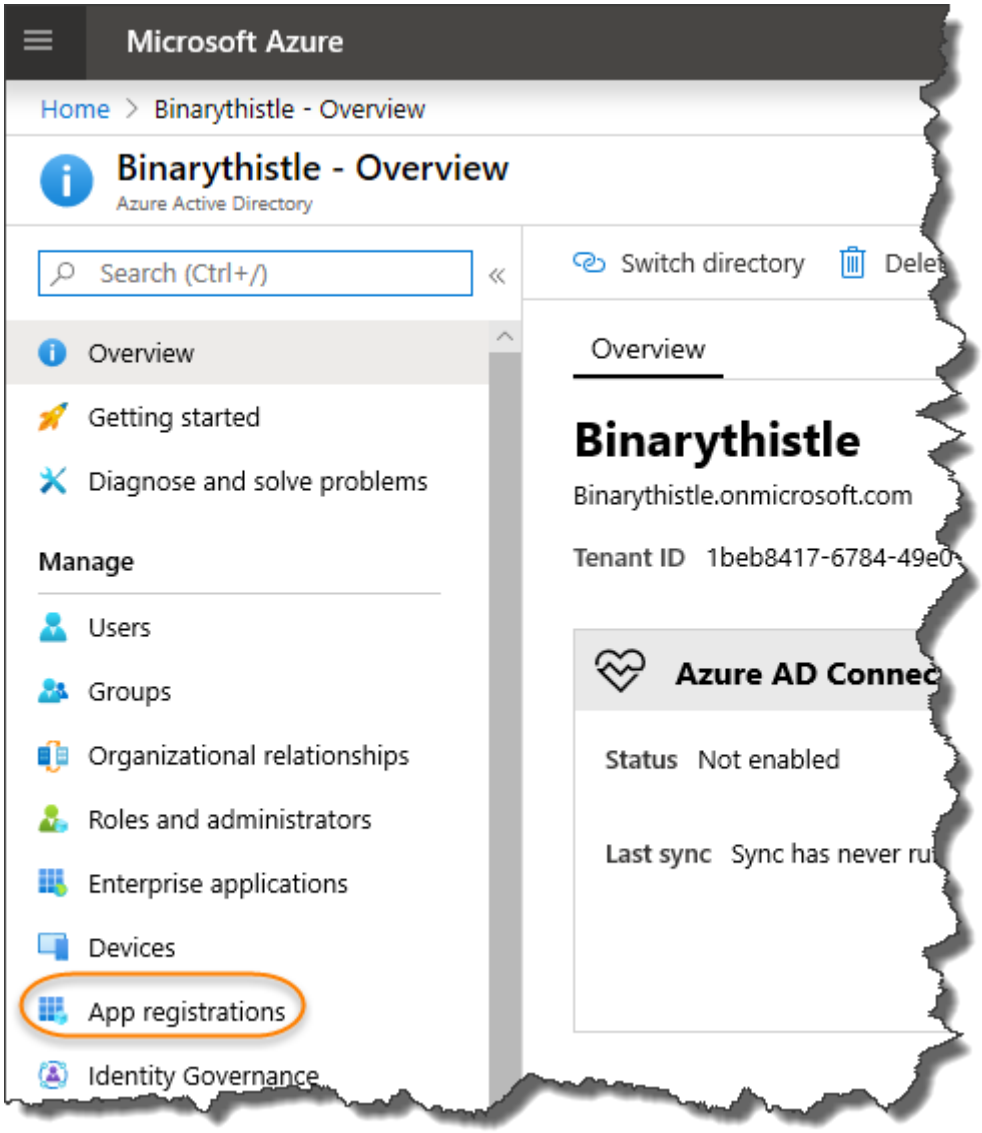


REGISTER OUR API

Select the AAD you're using for this exercise; the click on Azure Active Directory from your portal landing page:



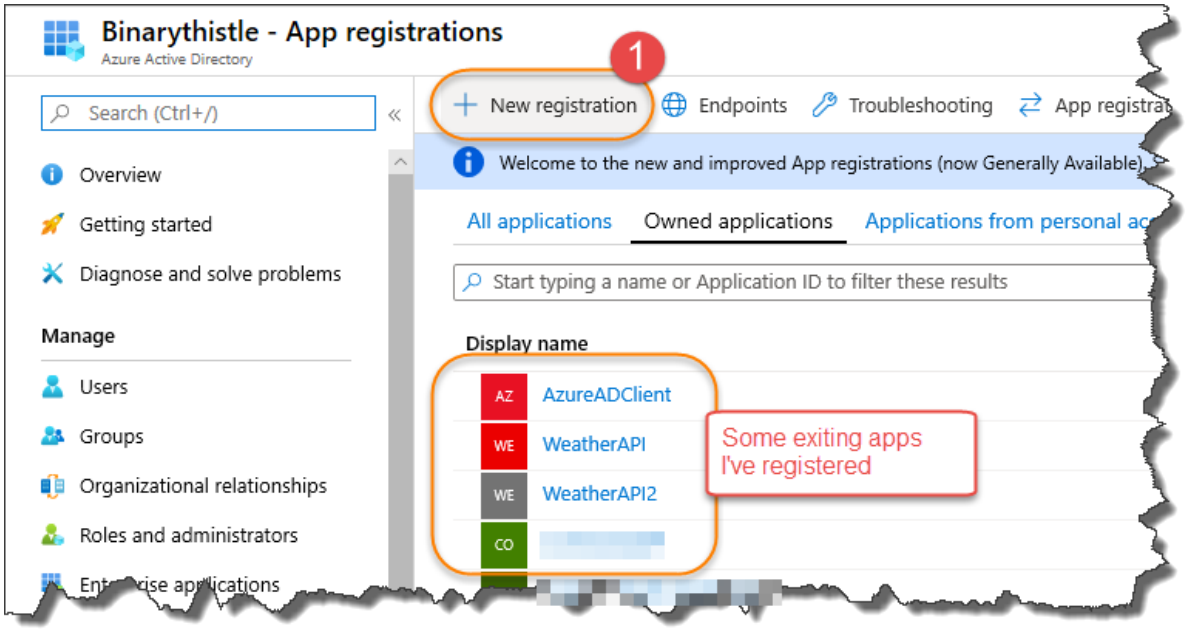
This should then take you into the Azure Active Directory overview screen:



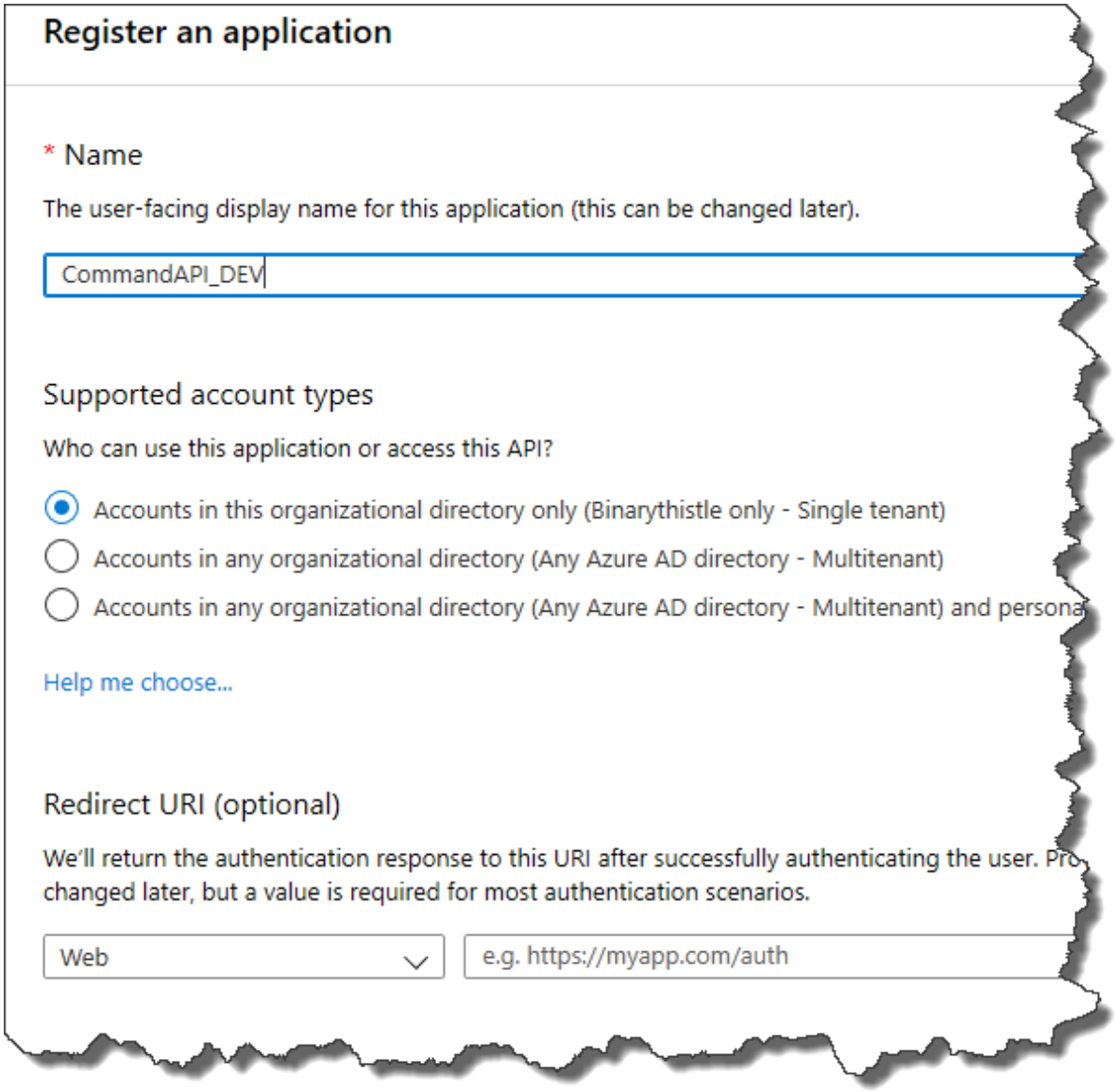
Select “App registrations” as shown above. You can see from the example below I already have a few apps registered on my AAD, but we’re going to create a new one for our CommandAPI in our “development” environment, (i.e. the one running locally on our PC).

Even though we are running our development API on our local machine, we can still make use of AAD as our Identity management service, (assuming our development PC has connectivity to the internet!)

The point I’m making here is that we can use AAD no matter where our API, (and client for that matter), are located.

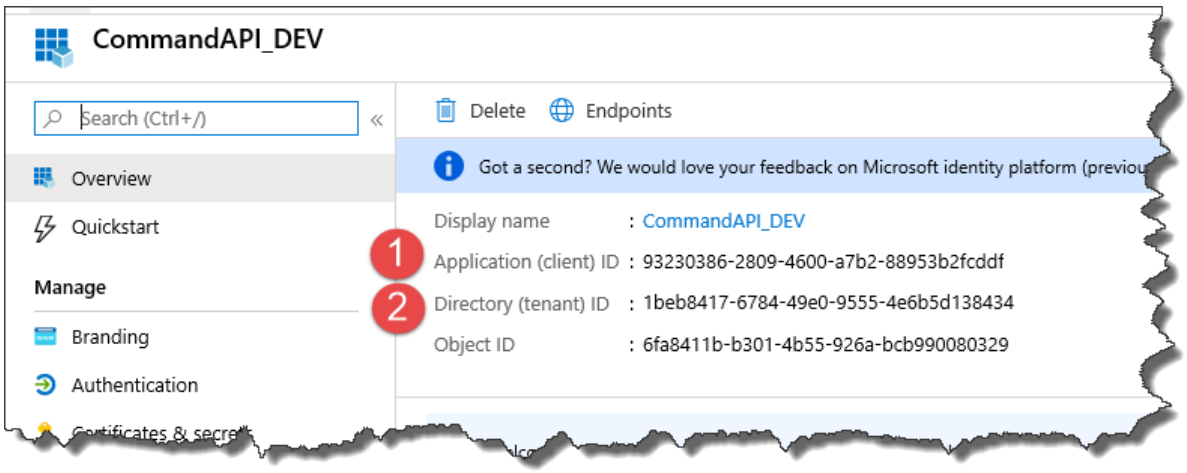


Select “New registration”, and you’ll see:



Enter a name for the app registration, it can be anything but make it meaningful, (I've appended "_DEV" to this registration to differentiate it from any Production Registration we subsequently create). Also ensure that "Accounts in this organization directory only ([*Your AAD Name*] only – Single tenant), is selected.

We don't need a Redirect URI, so click "register" to complete the initial registration, after which you'll be taken to the overview screen:



Here we are introduced to the first 2 important bits of information that we need to be aware of:

1. Application (client) ID
2. Directory (tenant) ID

Going forward I'm going to use the terms Client ID and Tenant ID, but what are they?

Client ID

The client ID is essentially just a unique identifier that we can refer to the *Command API* in reference to our AAD.

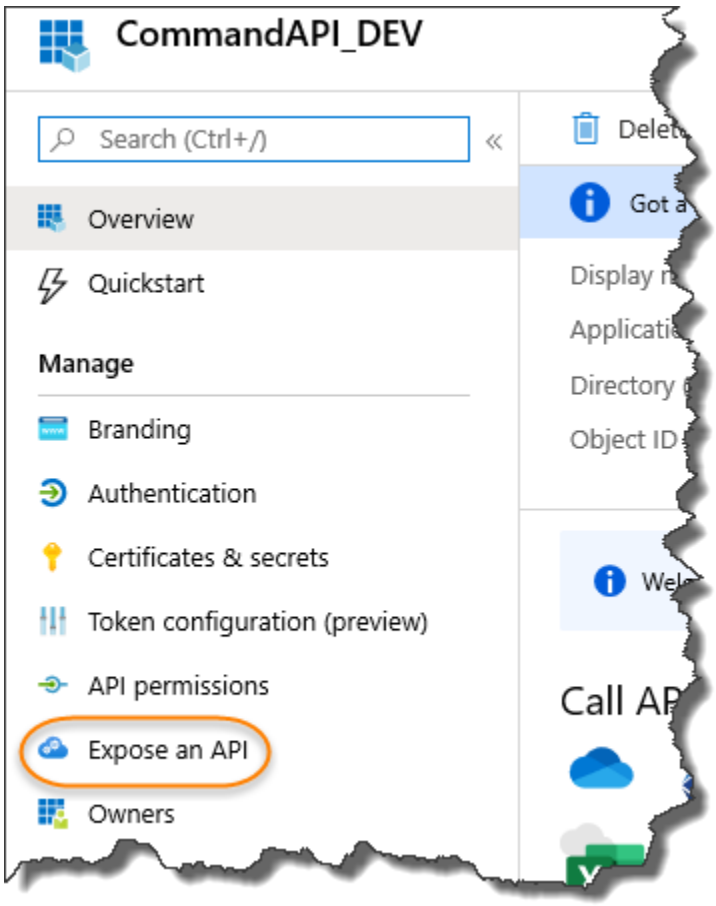
Tenant ID

A unique id relating to the AAD we're using, remembering that we can have multiple, (i.e. multi-tenant), AAD's at our disposal.

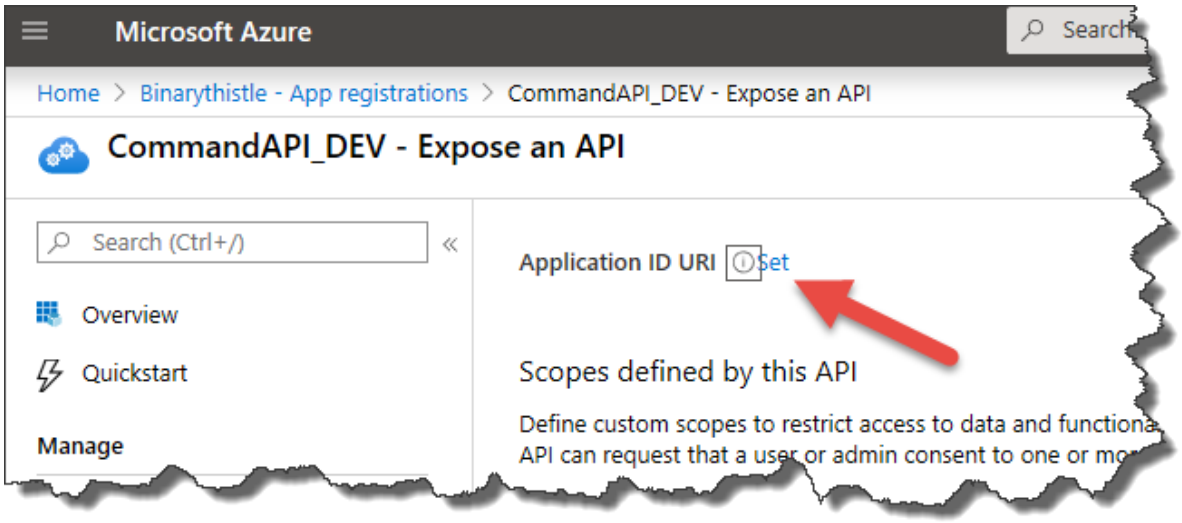
We'll come back to these items later when we come to configuring things at the application end, for now we need to move on as we're not quite finished.

Expose Our API

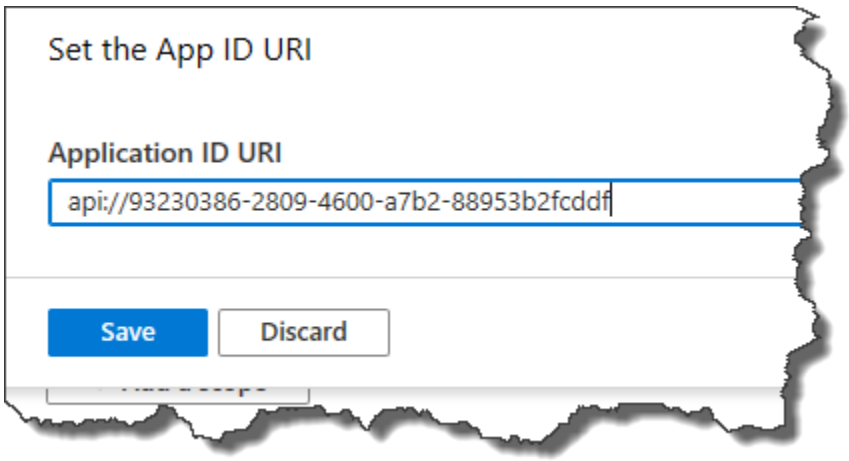
So far we've merely *registered* our API, we now need to *expose* it for use, so click on "Expose an API" from our left hand menu options on our registrations page:



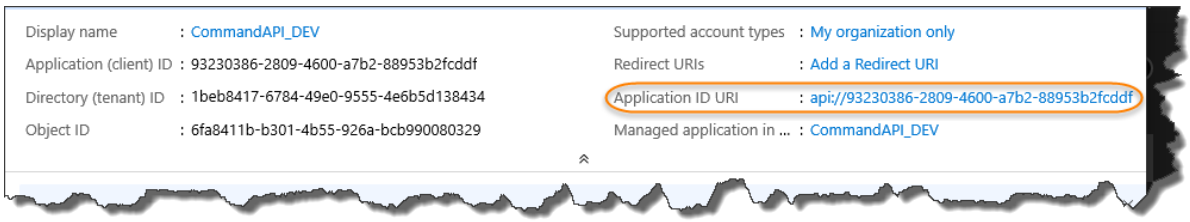
What we need to do here is create an “Application ID URI”, (sometimes referred to as a “Resource ID”), so click on “Set” as shown below:



Azure will provide a default suggestion for this, go with it, (it’s the Client ID with “`api://`” prepended):



Click Save and you're done. Clicking back into the overview of the registration and you should see this reflected here too:



We're almost finished with our API configuration in AAD, but have one more bit of configuration to complete.

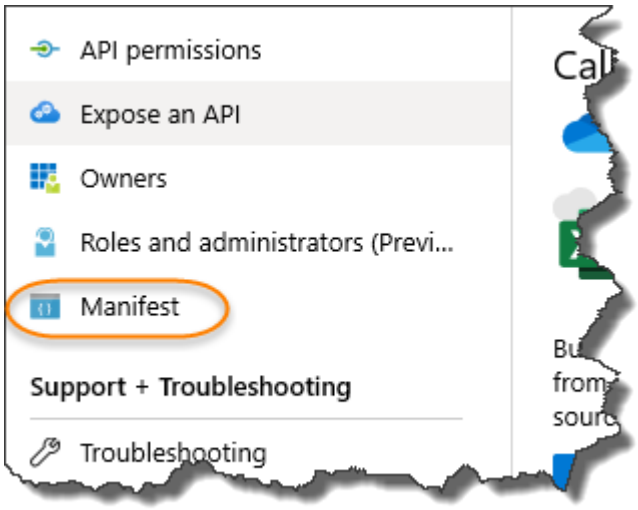
Update our Manifest

Here we update the `appRoles` section of our application manifest which specifies the type of application role(s) that can access the API. In our case we need to specify a non-interactive "daemon" app that will act as our API client. More information on the Application Manifest can be found here.

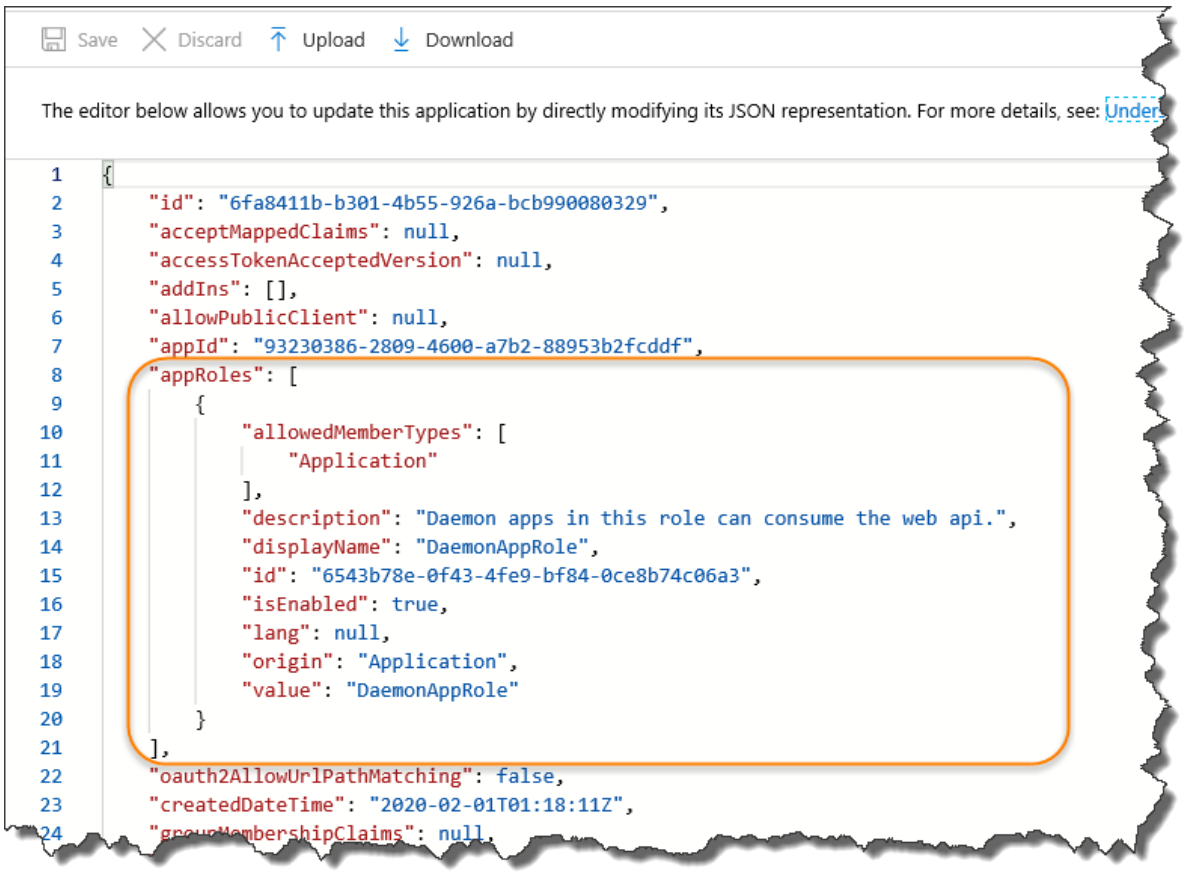
Anyway, back to the task at hand, we need to insert the following JSON snippet at the `appRoles` section of our manifest:

```
.
.
.
"appRoles": [
  {
    "allowedMemberTypes": [
      "Application"
    ],
    "description": "Daemon apps in this role can consume the web api.",
    "displayName": "DaemonAppRole",
    "id": "6543b78e-0f43-4fe9-bf84-0ce8b74c06a3",
    "isEnabled": true,
    "lang": null,
    "origin": "Application",
    "value": "DaemonAppRole"
  }
],
.
.
.
```

So, click on “Manifest” in the left-hand window of our App Registration config page:



And insert the json above into the correct spot, (essentially updating the existing empty `appRoles` section):



Make sure you keep the integrity of the json and don’t omit or introduce any additional commas, (for example). You can always use something like https://jsoneditoronline.org/ to check.

You can add multiple appRoles to this section, we need only one, although if you do decide to add some additional roles you’ll need to ensure that the “id” attribute is a unique guid.

When completed, don’t forget to save the file.

That's it for our API registration in Azure, we need to move over to our API now and make some config and code changes so it can make use of AAD for authorisation.

Add Configuration Elements

We need to make our API "aware" of the AAD settings we've just set up so that it can use AAD for authenticating clients. We need to configure:

- The login "Instance"
- Our AAD Domain
- The Tenant ID
- The Client ID
- The Application ID URL (Or Resource ID)

As we've already discussed you can store your application config in number of places, (e.g. ***appsettings.json, appsettings.Development.json*** etc.), in this section I'm going to make use of User Secrets once again, (refer to Chapter 7 for a refresher).

The primary reason I'm taking this approach is that I'll be pushing my code up to a public GitHub repository, and I don't want those items visible in something like ***appsettings.json***.

The table below details the name of the user secret variables I'm going to use for each of the config elements.

Config Element	User Secret Variable
The Login "Instance"	Instance
Our AAD Domain	Domain
The Tenant ID	TenantId
The Client ID	ClientId
The Application UD URL (Or Resource ID)	ResourceId

As a quick refresher to add the "Instance" User Secret, at a command prompt "inside" the API Project root folder type:

```
dotnet user-secrets set "Instance" "https://login.microsoftonline.com/"
```

This will add a value for our Login Instance, (you should use the same value I've used here). The other User Secrets I'll leave for you to add yourself, as the values you need to supply will be unique to your own App Registration, (refer to these values on the App Registration overview screen for your API).

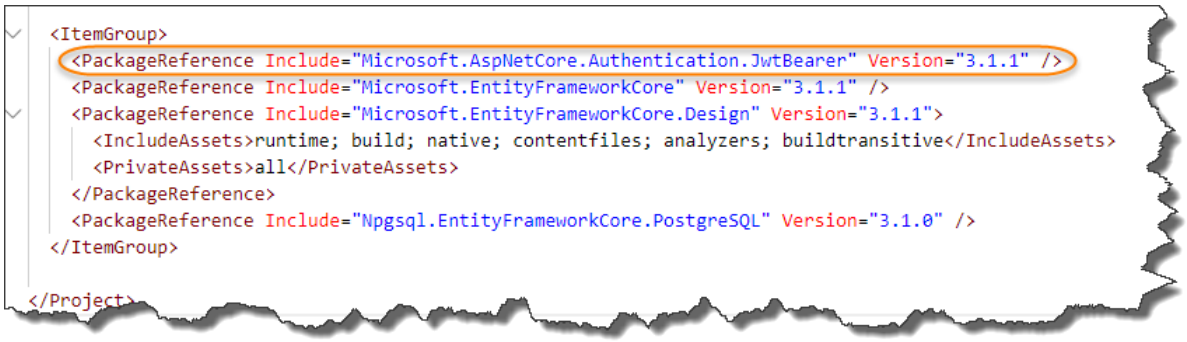
After adding all my User Secrets, the contents of my ***secrets.json*** file now looks like this:

Update our Project Packages

Before we start coding, we need to add a new package that will be required to support the code we're going to introduce, so at a command prompt "inside" the API project type:

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

This should successfully add the following package reference to the .csproj file:



Updating our Startup Class

Over in the `startup` class of our API project we need to update both our `ConfigureServices` and `Configure` methods. First though, add the following using directive to the top of the `startup` class file:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
```

Update Configure Services

We need to set up bearer authentication in the ConfigureServices method, to do so add the following code, (new code if highlighted):

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

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
  .AddJwtBearer(opt =>
  {
    opt.Audience = Configuration["ResourceId"];
    opt.Authority = $"{Configuration["Instance"]}{Configuration["TenantId"]}";
  });

services.AddControllers();
.
.
.
```

To put more in context it'll look like this:

```
public void ConfigureServices(IServiceCollection services)
{
    var builder = new NpgsqlConnectionStringBuilder();
    builder.ConnectionString = Configuration.GetConnectionString("PostgreSqlConnection");
    builder.Username = Configuration["UserID"];
    builder.Password = Configuration["Password"];
    services.AddDbContext<CommandContext>(opt => opt.UseNpgsql(builder.ConnectionString));

    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(opt => {
            opt.Audience = Configuration["ResourceId"];
            opt.Authority = $"{Configuration["Instance"]}{Configuration["TenantId"]}";
        });

    services.AddControllers();
}
```

The code above adds authentication to our API, specifically Bearer authentication using JWT Tokens. We then configure 2 options:

- Audience: We set this to the ResourceID of our App Registration in Azure
- Authority: Our AAD Instance that is the token issuing authority

Update Configure

All we need to do now is add authentication & authorization to our request pipeline via the `Configure` method:

```
app.UseAuthentication();
app.UseAuthorization();
```

As shown below:

```
public void Configure(IApplicationBuilder app, IWebHostE
{
    context.Database.Migrate();
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
       endpoints.MapControllers();
    });
}
```

Authentication Vs Authorisation

As we've added both Authentication and Authorisation to our request pipeline, I just want to quickly outline the difference between these two concepts before we move on.

- Authentication (the "who"): Verifies who you are, essentially it checks your identity is valid
- Authorisation (the "what"): Grants the permissions / level of access that you have

So in our example our client app will be authenticated via AAD, once it has, we can then determine *what* end points it can call on our API, (authorisation).

Update Our Controller

We have added the foundations of Bearer authentication using JWT tokens to our `Startup` class to enable it to be used throughout our API, but now we want to use it to protect 1 of our endpoints. We can of course protect the entire API, but let's just start small for now. We can pick any of our API endpoints, but let's just go with one of our simple GET methods, specifically our ability to retrieve a single Command.

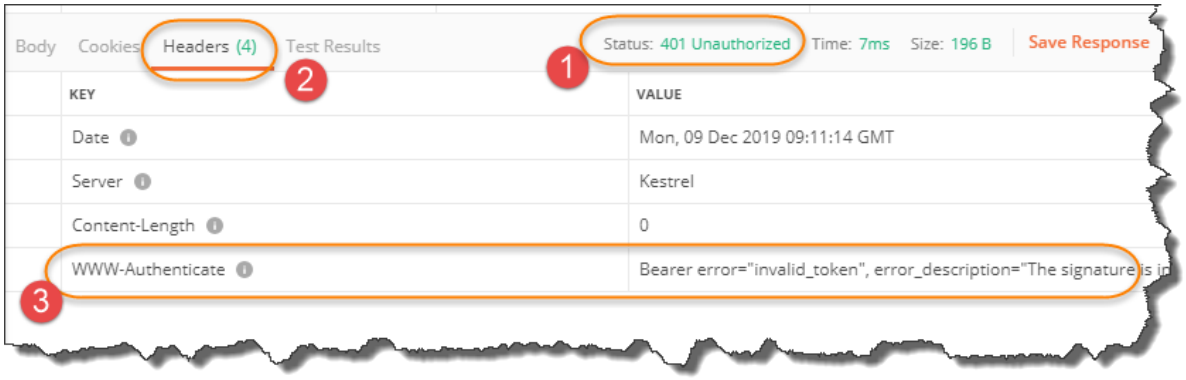
Before we update our ActionResult, just make sure you add the following using directive at the top of our `CommandsController` class:

```
using Microsoft.AspNetCore.Authorization;
```

The new code for our ActionResult is simple, we just decorate our ActionResult with `[Authorize]`:

```
//GET:      api/commands/Id
[Authorize]
[HttpGet("{id}")]
public ActionResult<Command> GetCommandItem(int id)
{
  var commandItem = _context.CommandItems.Find(id);

  if (commandItem == null)
    return NotFound();

  return commandItem;
}
```

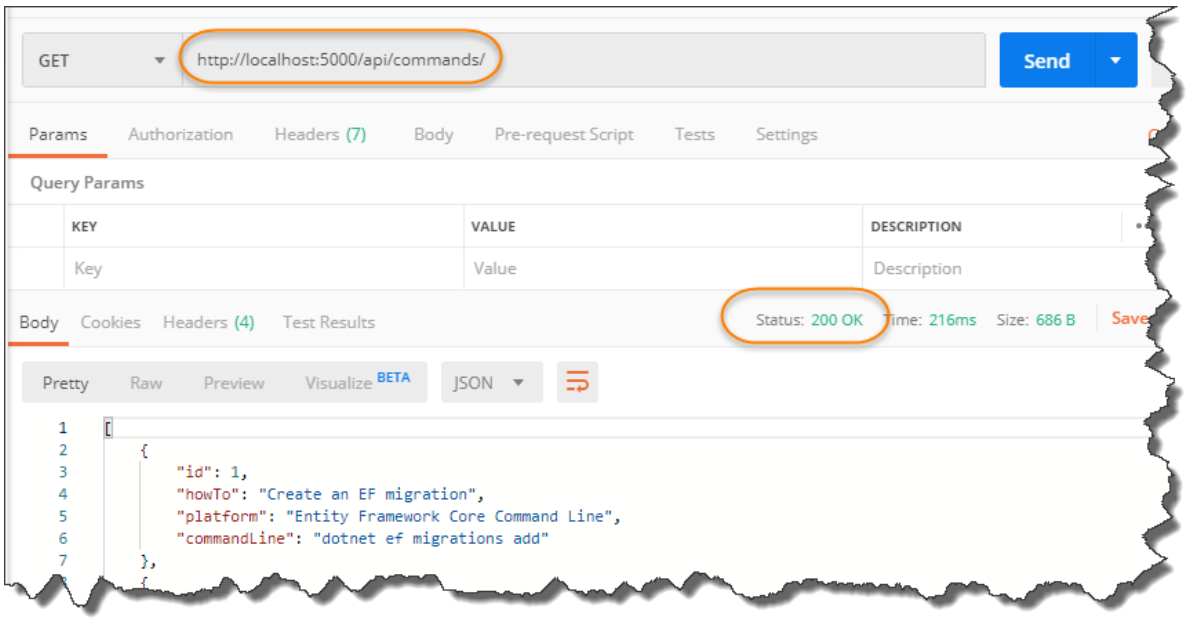
Save all the new code, build then run the API locally. Once running make a call to our newly protected endpoint in Postman:



Here you will see:

1. We get a 401 Unauthorized response
2. Selecting the return headers we see…
3. That the authentication type is “Bearer” (and we have a token error back from AAD)

To double check we have only protected this endpoint, make a call to our other GET ActionResult and you’ll see we still get a list of commands back:



Learning Opportunity: What happens if we run our Unit Test suite? Will some of our tests break because we require authorisation on one of our API endpoint methods? If not, why not?

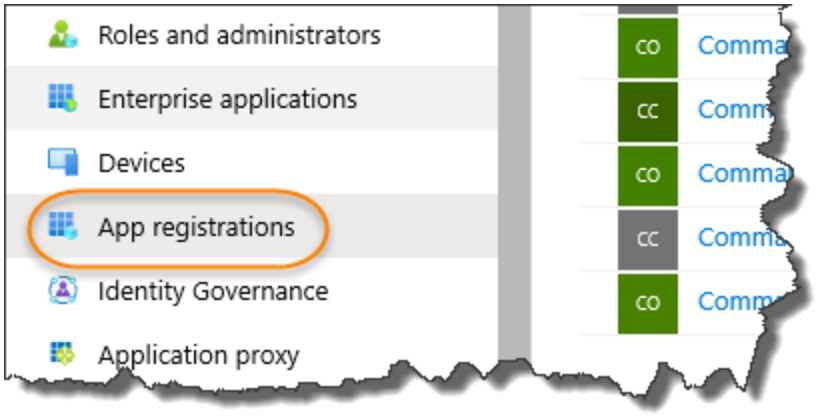
Register Our Client App

In the next section we're going to write a simple .NET Core Console application that will act as an authorised "client" of the API. As this is a "daemon app" it needs to run without user authentication interaction, so we need to configure it as such.

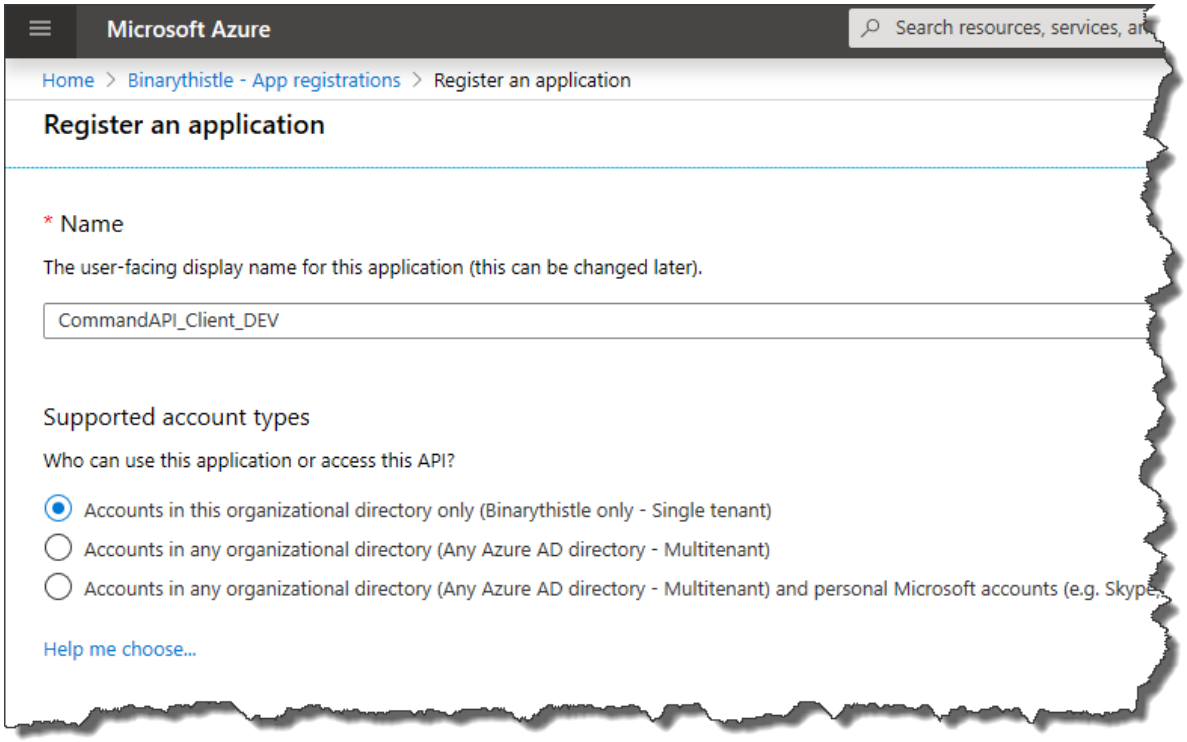
There are a number of different authentication use-cases we could explore when it comes to consuming an API, for example a user authenticating against AAD, (username / password combo), to grant access to the API.

The use-case I've decided to go with in this example, (a "daemon app"), resonated with me more in terms of a real-world use-case. You may of course disagree…

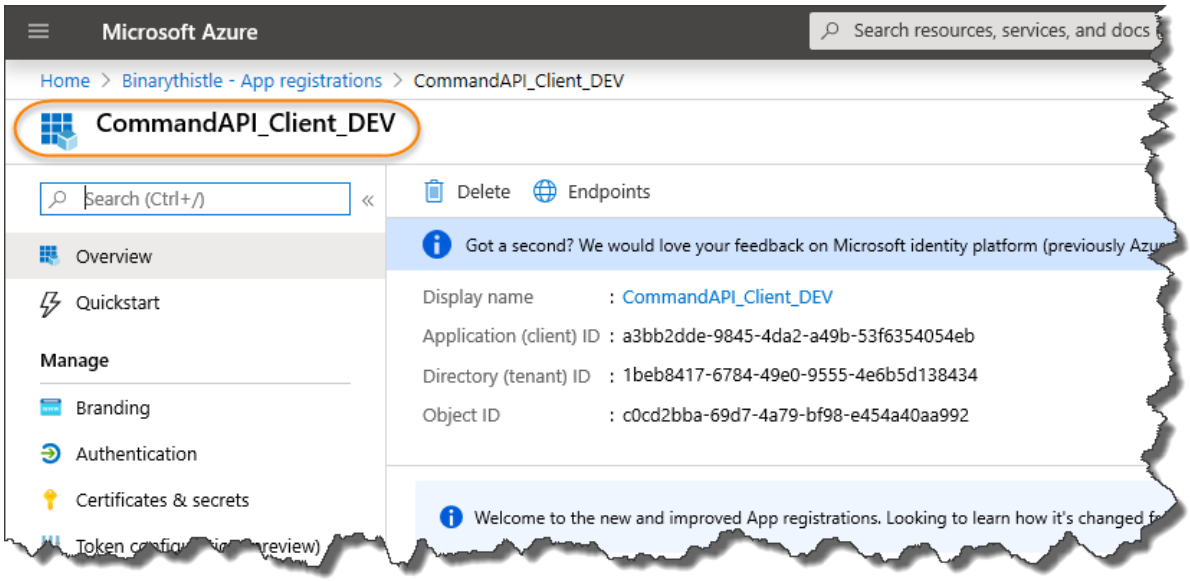
Back over in Azure, select the same AAD that you registered the API in, and select App Registrations once again:



The select "+ New registration", and on the resulting screen enter a suitable name for our client app as shown below:



Again, select the *Single tenant* Supported account type option and click "Register", this will take you to the overview screen of your new app registration:

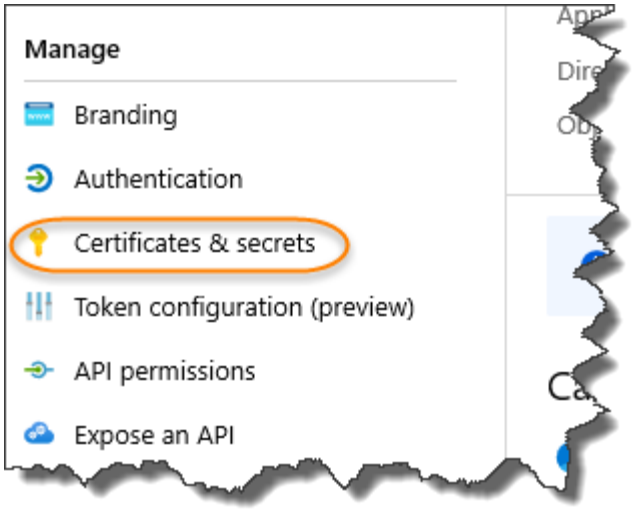


As before it will prepopulate some of the config elements for you, e.g. Client ID, Tenant ID etc.

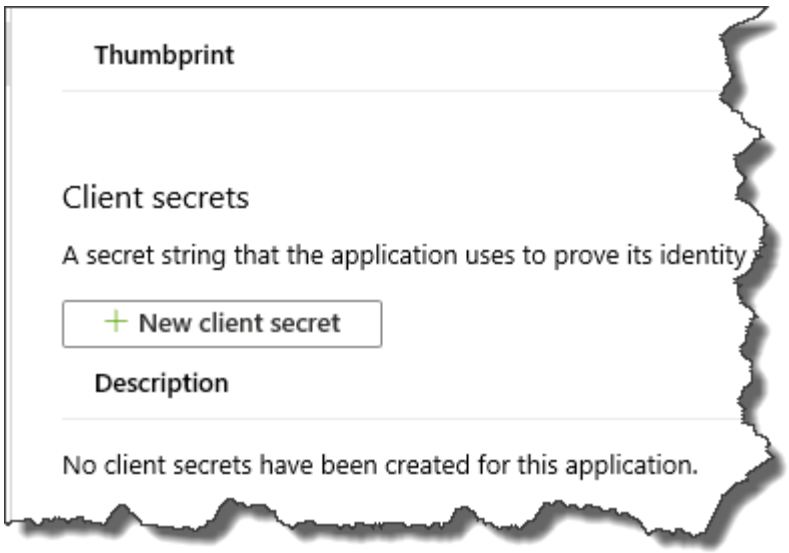
Learning Opportunity: What do you notice about the Tenant ID for our client registration when compared to the Tenant ID of API registration?

Create a Client Secret

Next click on "Certificates & secrets" in the left-hand menu:

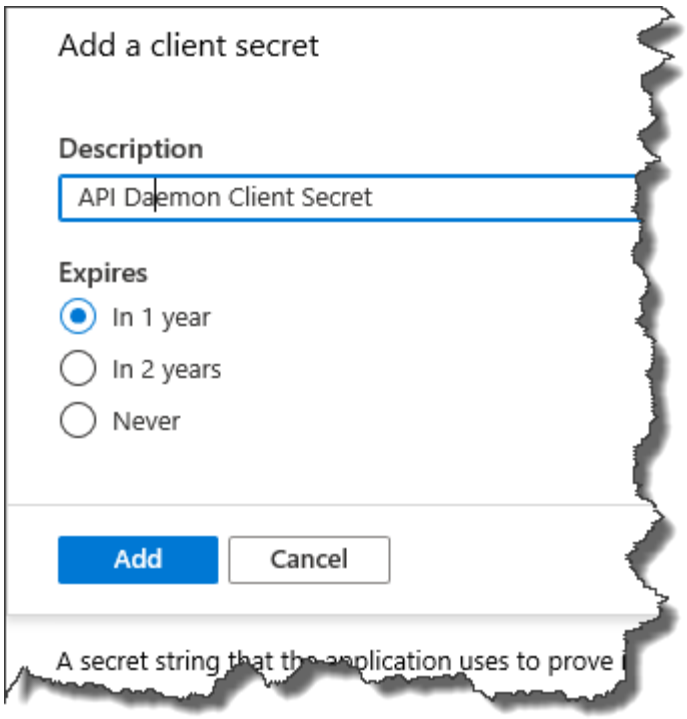


Here we are going to configure a “Client Secret”. This is a unique ID that we will use in combination with our other app registration attributes to identify and authenticate our client to our API. Click “+ New client secret”:



And on the resulting screen give it:

- A description (can be anything but make it meaningful)
- An expiry (you have a choice of options)

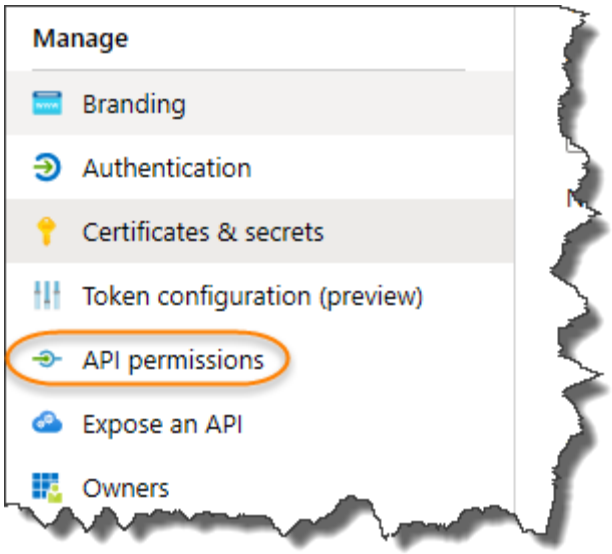


When you're happy click "add".

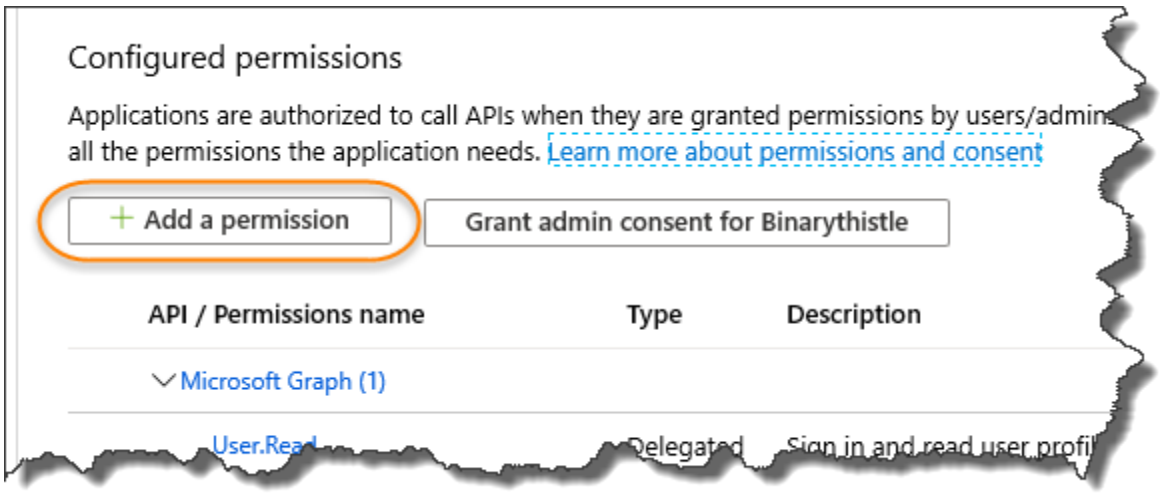
Warning! Make sure you **take a copy of the client secret now,** shortly after creation it will not be displayed in full again – you'll only see a redacted version, and you won't be able to retrieve it unlike our other registration attributes.

Configure API Permissions

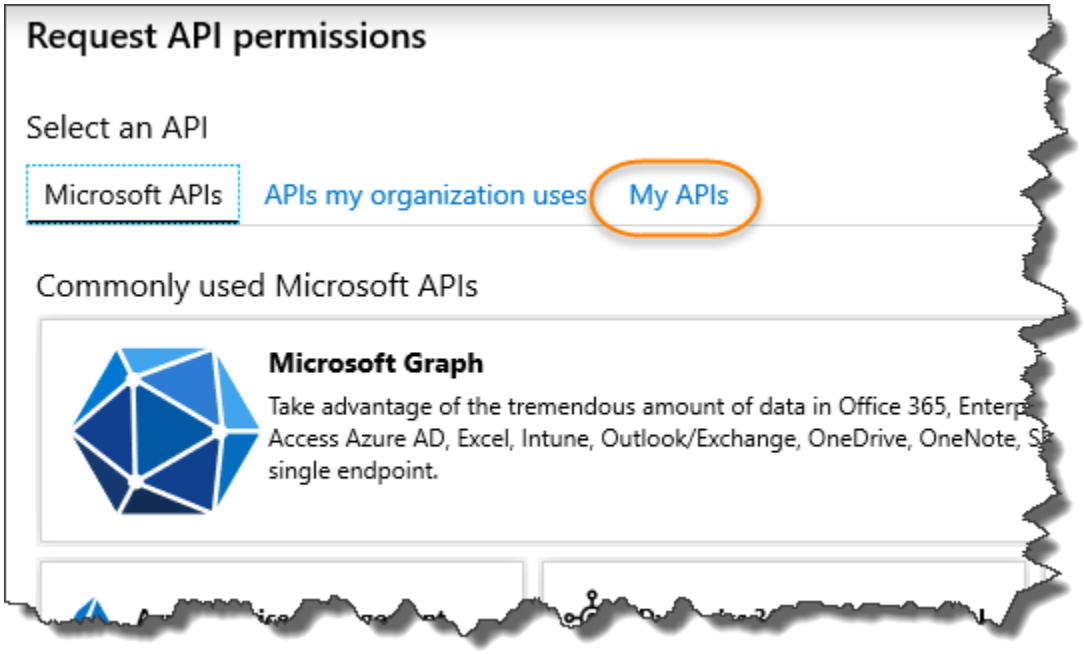
Now click on "API Permissions", here we are going to, (drum role please), configure access to our command API:



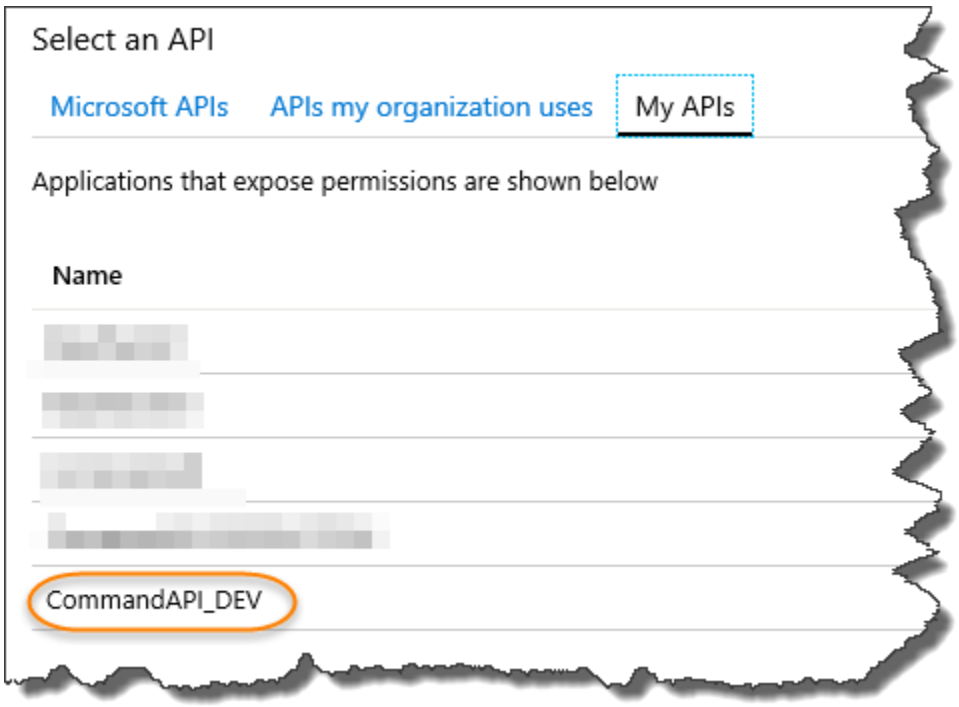
Click on "+ Add a permission"



In the “Request API permissions” window that appears, select the “My APIs” tab:

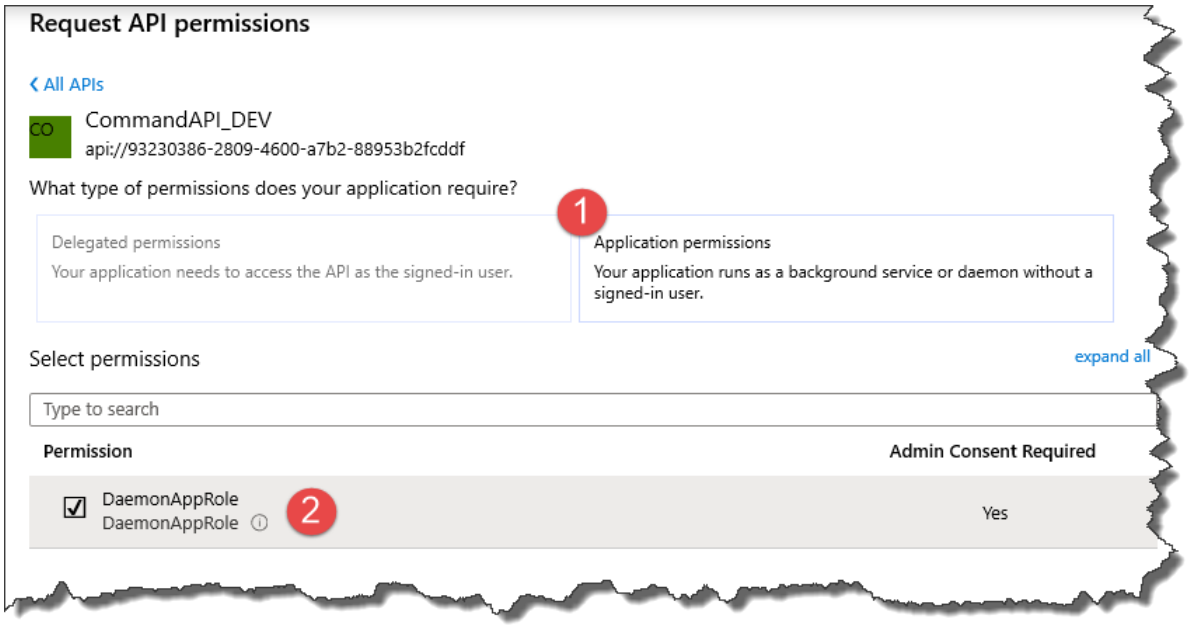


And find the command API, and select it:

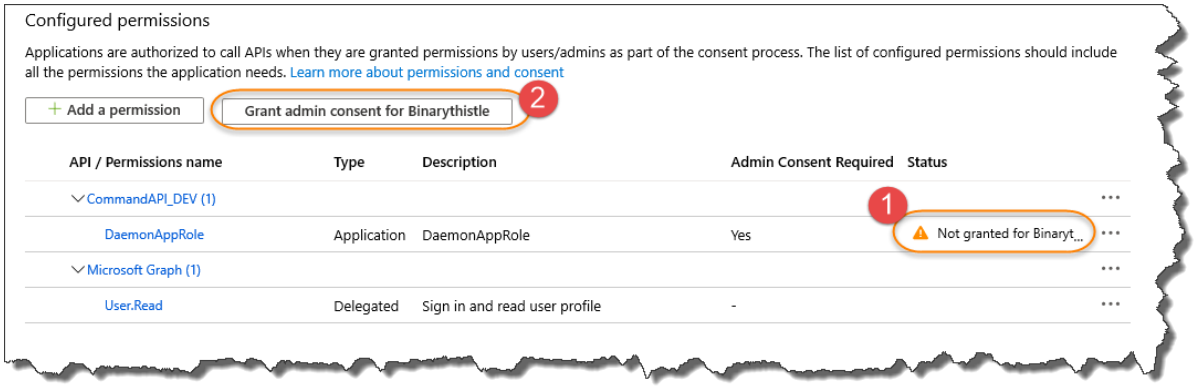


On the resulting screen ensure that:

1. Application permissions is selected
2. You “check” the DaemonAppRole Permission



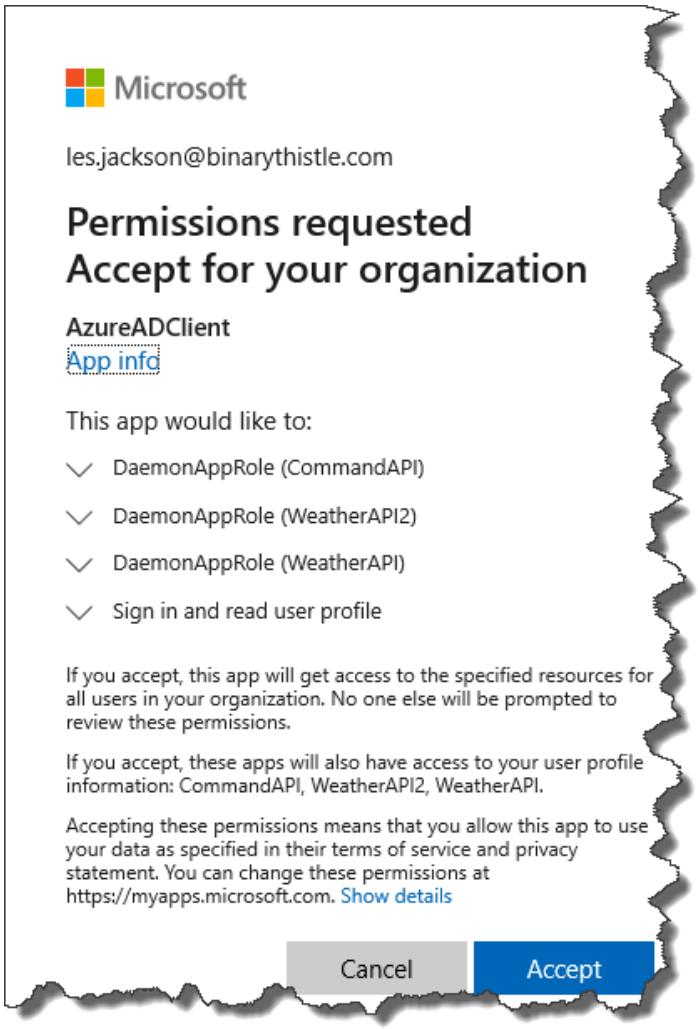
When you’re happy, click “Add permission” and your permission will be added to the list:



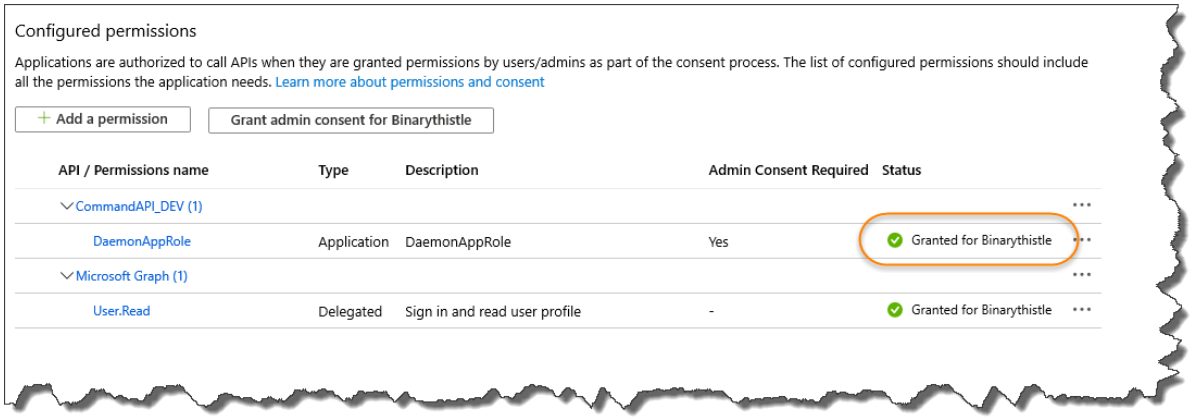
You'll notice:

1. The permission has been "created" but not yet "granted
2. You'll need to click the "Grant admin consent for Binarythistle" button – do so now:

You'll get a Microsoft authentication pop up, authenticate and Accept any permissions request you get:



You'll be returned to the Configure permissions window, where after a short time, your newly created API Permission will have been granted access:



And with that the registration of our, (yet to be created), client app is complete.

Create Our Client App

The final part of this chapter is to create a simple client that we can use to call our protected API, so we're going to create new console project to do just that.

I don't consider this app part of our "solution", (containing our API & Test Projects), so I'm going to create it in a totally separate working project directory outside of ***CommandAPISolution*** folder.

Note: as we'll only be creating 1 project, I'm *not* going to make use of a "solution" structure.

You can find the code to this project here on GitHub

At a command prompt in a new working directory "outside" of our ***CommandAPISolution*** folder, type:

```
dotnet new console -n CommandAPIClient
```

Once the project has been created open the project folder ***CommandAPIClient*** in your development environment, so if you're using VS Code you could type:

```
code -r CommandAPIClient
```

This will open the project folder ***CommandAPIClient*** in VS Code.

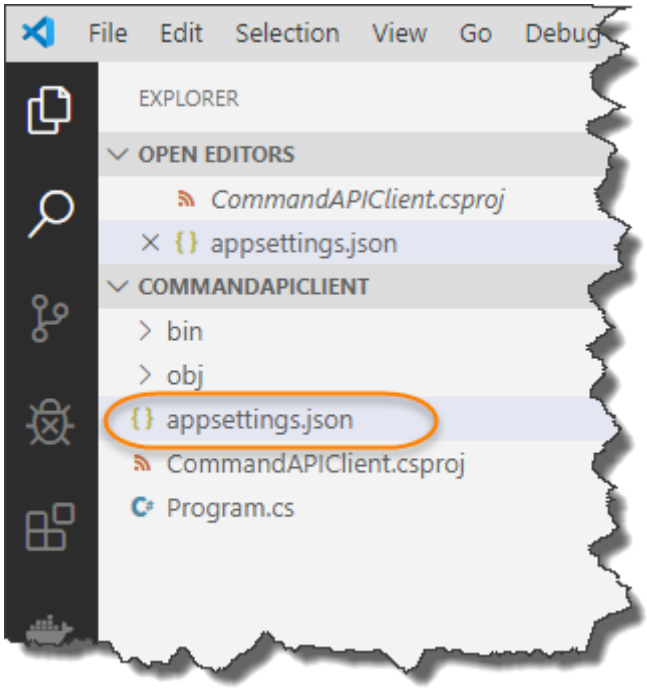
Our Client Configuration

As I'm making this code available on GitHub for you to pull down and use, I'm deliberately going to store the config in an ***appsettings.json*** file as opposed to using User Secrets, as it will be easier for you to get going with it quickly[25]. We will therefore be storing sensitive config elements in here, therefore for production systems **you would not do this!**

Learning Opportunity: Following the approach we took for our API, "convert" the Client App example here to use user secrets.

Create an ***appsettings.json*** file in the root of your project folder, once done it should look like this if you're using VS Code:

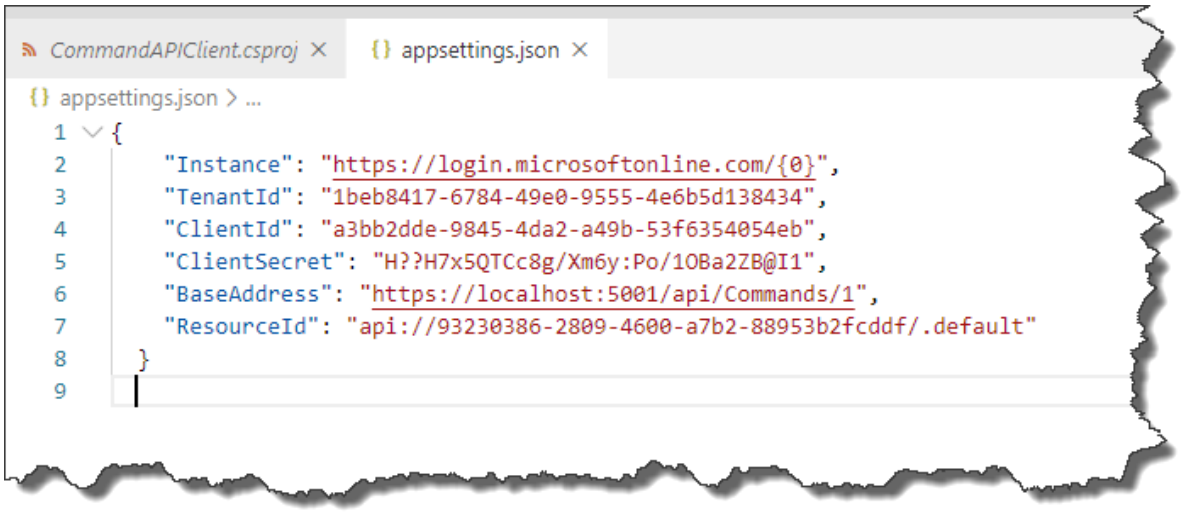
[25] I appreciate its totally counter to the point I made before in regard to our API



Into that file add the following JSON, making sure to populate the correct values for **your client/daemon** application registration, and in the case of the `ResourceId & BaseAddress`, **your API** application registration.

```
{
  "Instance": "https://login.microsoftonline.com/{0}",
  "TenantId": "[YOUR TENANT ID]",
  "ClientId": "[YOUR CLIENT ID]",
  "ClientSecret": "[YOUR CLIENT SECRET]",
  "BaseAddress": "https://localhost:5001/api/Commands/1",
  "ResourceId": "api://[YOUR API CLIENT ID]/.default"
}
```

So for example my file looks like this:



A couple of points here:

- BaseAddress: This is just the local address of the command API (we'll update to our production URL later). Note, I'm deliberately specifying the API Action Result that requires authorization.
- ResourceId: This is the `ResourceId` of our **API App** Registration

The other attributes are straightforward and can be retrieved from Azure, except the `ClientSecret` which you should have made a copy of when you created it.

Warning! All the attributes above are enough to get access to our restricted API without the need for any additional passwords etc. So you **should not** store it like this in production, you should make use of user secrets, or something similar.

Again, I've chose to provide it in an ***appsettings.json*** file to allow you to get up and running quickly with the code and have left it as a learning exercise for you to implement the *user secrets* approach.

Add Our Package References

Before we start coding, we need to add some package references to our project to support some of the features we're going to use, so we'll add:

- Microsoft.Extensions.Configuration
- Microsoft.Extensions.Configuration.Binder
- Microsoft.Extensions.Configuration.Json
- Microsoft.Identity.Client

I prefer to do this by using the dotnet CLI as we've done previously so:

```
dotnet add package <package name>
```

So, for example issue the following command inside the CommanAPIClient app fll:

```
dotnet add package Microsoft.Extensions.Configuration
```

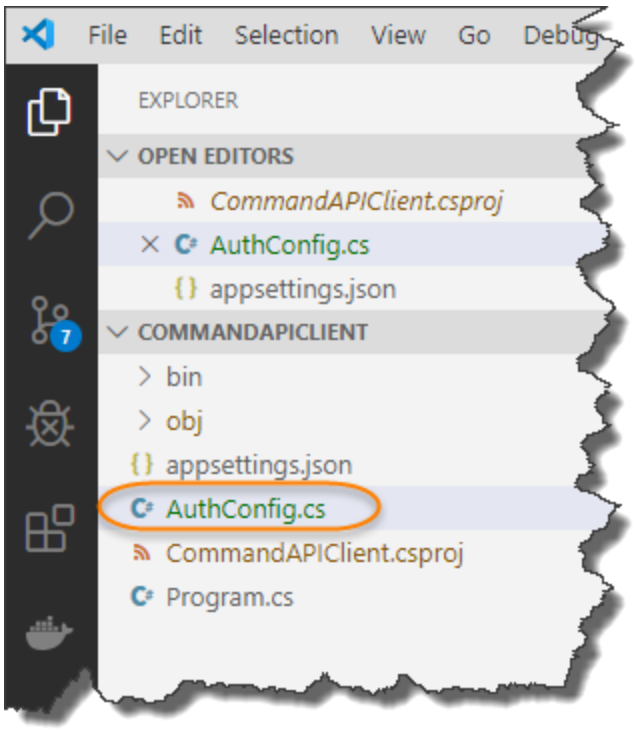
Repeat so you add all 4 packages, your project .csproj file should look like this when done:

CommandAPIClient.csproj

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Configuration" Version="3.1.1" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Binder" Version="3.1.1" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json" Version="3.1.1" />
    <PackageReference Include="Microsoft.Identity.Client" Version="4.8.1" />
  </ItemGroup>

</Project>

```

Client Configuration Class

For ease of use we're going to create a custom class that will allow us to read in our ***appsettings.json*** file and then access those config elements as class attributes. In the client project create a new class file in the root of the project and call it AuthConfig.cs as shown below:

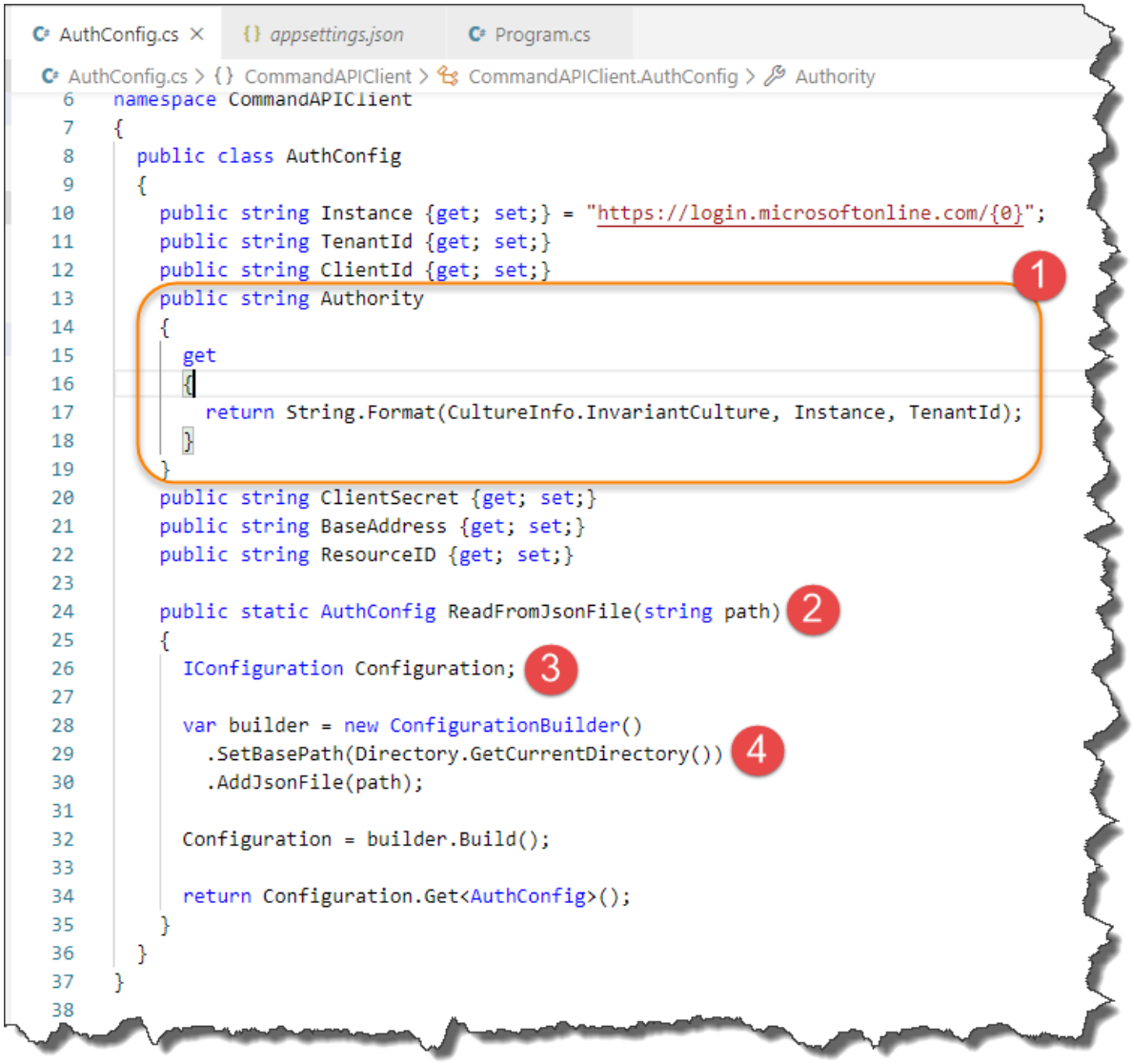


Then enter the following code:

```
using System;
using System.IO;
using System.Globalization;
using Microsoft.Extensions.Configuration;

namespace CommandAPIClient
{
  public class AuthConfig
  {
    public string Instance {get; set;} =
      "https://login.microsoftonline.com/{0}";
    public string TenantId {get; set;}
    public string ClientId {get; set;}
    public string Authority
    {
      get
      {
        return String.Format(CultureInfo.InvariantCulture,
                             Instance, TenantId);
      }
    }
    public string ClientSecret {get; set;}
    public string BaseAddress {get; set;}
    public string ResourceID {get; set;}

    public static AuthConfig ReadFromJsonFile(string path)
    {
      IConfiguration Configuration;

      var builder = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile(path);
```

```
        Configuration = builder.Build();

        return Configuration.Get<AuthConfig>();
    }
  }
}
```

When complete your AuthConfig class should look like this:




```
namespace CommandAPIClient
{
  public class AuthConfig
  {
    public string Instance {get; set;} = "https://login.microsoftonline.com/{0}";
    public string TenantId {get; set;}
    public string ClientId {get; set;}
    public string Authority
    {
      get
      {
        return String.Format(CultureInfo.InvariantCulture, Instance, TenantId);
      }
    }
    public string ClientSecret {get; set;}
    public string BaseAddress {get; set;}
    public string ResourceID {get; set;}

    public static AuthConfig ReadFromJsonFile(string path)
    {
      IConfiguration Configuration;

      var builder = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile(path);

      Configuration = builder.Build();

      return Configuration.Get<AuthConfig>();
    }
  }
}
```

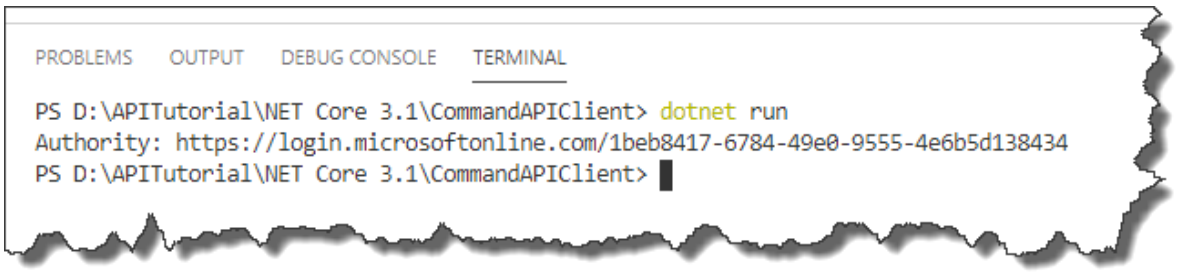
Notable code listed below:

1. We combine the Instance and our AAD Tenant to create something called the “Authority”, this is required when we come to attempting to connect our client later…
2. Our class has 1 static method that allows us to specify the name of our json config file
3. We create an instance of the .NET Core Configuration subsystem
4. Using `ConfigurationBuilder` we read the contents of our json config file
5. We pass back our read-in config bound to our AuthConfig class

To quickly test that this all works, perform a build, and assuming we have no errors, move over to our Program class and edit the Main method so it looks like this:

```
static void Main(string[] args)
{
  AuthConfig config = AuthConfig.ReadFromJsonFile("appsettings.json");

  Console.WriteLine($"Authority: {config.Authority}");
}
```

Build your code again then run it, assuming all is well you should get output similar to this:



Finalise Our Program Class

As mentioned previously the first thing our client will have to do is obtain a JWT token that it will then attach to all subsequent requests in order to get access to the resources it needs, so let's focus in on that.

Still in our Program class we're going to create a new static asynchronous method called `RunAsync`, the code for our reworked `Program` class is shown below, (noting new or changed code is bold & highlighted):

```
using System;
using System.Threading.Tasks;
using Microsoft.Identity.Client;

namespace CommandAPIClient
{
  class Program
  {
    static void Main(string[] args)
    {
        Console.WriteLine("Making the call...");
        RunAsync().GetAwaiter().GetResult();
    }

    private static async Task RunAsync()
    {
      AuthConfig config = AuthConfig.ReadFromJsonFile("appsettings.json");

      IConfidentialClientApplication app;

      app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
          .WithClientSecret(config.ClientSecret)
          .WithAuthority(new Uri(config.Authority))
          .Build();

      string[] ResourceIds = new string[] {config.ResourceID};

      AuthenticationResult result = null;
      try
      {
        result = await app.AcquireTokenForClient(ResourceIds).ExecuteAsync();
        Console.ForegroundColor = ConsoleColor.Green;
```

```
            Console.WriteLine("Token acquired \n");
            Console.WriteLine(result.AccessToken);
            Console.ResetColor();
        }
        catch (MsalClientException ex)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(ex.Message);
            Console.ResetColor();
        }
    }
  }
}
```

I've tagged the points of interest below:

```
static void Main(string[] args)
{
    Console.WriteLine("Making the call...");
    RunAsync().GetAwaiter().GetResult();
}

private static async Task RunAsync()
{
    AuthConfig config = AuthConfig.ReadFromJsonFile("appsettings.json");

    IConfidentialClientApplication app;

    app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
        .WithClientSecret(config.ClientSecret)
        .WithAuthority(new Uri(config.Authority))
        .Build();

    string[] ResourceIds = new string[] { config.ResourceID };

    AuthenticationResult result = null;
    try
    {
        result = await app.AcquireTokenForClient(ResourceIds).ExecuteAsync();
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine("Token acquired \n");
        Console.WriteLine(result.AccessToken);
        Console.ResetColor();
    }
    catch (MsalClientException ex)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(ex.Message);
        Console.ResetColor();
    }
}
```

1. Our `RunAsync` method is asynchronous and returns a result we're interested in, so we chain the `GetAwaiter` and `GetResult` methods to ensure the console app does not quit before a result is processed and returned.
2. `ConfidentialClientApplication` is a specific class type for our use case, we use this in conjunction with the `ConfidentialClientApplicationBuilder` to construct a "client" with our config attributes.
3. We set up our app with the values derived from our `AuthConfig` class
4. We can have more than one `ResourceId`, (or scope), that we want to call hence we create a string array to cater for this
5. The `AuthenticationResult` contains, (drum roll), the result of a token acquisition
6. Finally we make an asynchronous AquireTokenForClient call to, (hopefully!), return a JWT Bearer token from AAD using our authentication config

Save the file, build your code and assuming all's well, run it too, should see:

Celebration Check Point: Good job! There was a lot of config and coding to get us to this point, obtaining a JWT token, so the rest of this chapter is all too easy!

So well done!

We move onto the 2nd and final part of our `RunAsync` method, and that is to call our protected API endpoint with the token we just obtained in the previous step, so directly after the `catch` statement in our `RunAsync` method, add the following code, (take note of the 3 additional using statements too):

```
using System.Net.Http;
using System.Net.Http.Headers;
using System.Linq;
.
.
.
if (!string.IsNullOrEmpty(result.AccessToken))
{
  var httpClient = new HttpClient();
  var defaultRequestHeaders = httpClient.DefaultRequestHeaders;

  if(defaultRequestHeaders.Accept ==null ||
     !defaultRequestHeaders.Accept.Any(m => m.MediaType == "application/json"))
```

```
  {
    httpClient.DefaultRequestHeaders.Accept.Add(new
      MediaTypeWithQualityHeaderValue("application/json"));
  }
  defaultRequestHeaders.Authorization =
    new AuthenticationHeaderValue("bearer", result.AccessToken);

  HttpResponseMessage response = await httpClient.GetAsync(config.BaseAddress);
  if (response.IsSuccessStatusCode)
  {
    Console.ForegroundColor = ConsoleColor.Green;
    string json = await response.Content.ReadAsStringAsync();
    Console.WriteLine(json);
  }
  else
  {
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($"Failed to call the Web Api: {response.StatusCode}");
    string content = await response.Content.ReadAsStringAsync();
    Console.WriteLine($"Content: {content}");
  }
  Console.ResetColor();
}
```

I've highlighted some interesting code sections below:

```
catch (MsalClientException ex)
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(ex.Message);
    Console.ResetColor();
}
if (!string.IsNullOrEmpty(result.AccessToken))
{
    var httpClient = new HttpClient();
    var defaultRequestHeaders = httpClient.DefaultRequestHeaders;

    if (defaultRequestHeaders.Accept == null ||
       !defaultRequestHeaders.Accept.Any(m => m.MediaType == "application/json"))
    {
        httpClient.DefaultRequestHeaders.Accept.Add(new
          MediaTypeWithQualityHeaderValue("application/json"));
    }
    defaultRequestHeaders.Authorization =
      new AuthenticationHeaderValue("bearer", result.AccessToken);

    HttpResponseMessage response = await httpClient.GetAsync(config.BaseAddress);
    if (response.IsSuccessStatusCode)
    {
        Console.ForegroundColor = ConsoleColor.Green;
        string json = await response.Content.ReadAsStringAsync();
        Console.WriteLine(json);
    }
    else
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine($"Failed to call the Web Api: {response.StatusCode}");
        string content = await response.Content.ReadAsStringAsync();
        Console.WriteLine($"Content: {content}");
    }
    Console.ResetColor();
}
```

1. We use a `HttpClient` object as the primary vehicle to make the request
2. We ensure that we set the media type in our request headers appropriately
3. We set out authorisation header to “bearer” as well as attaching our token received in the last step
4. Make an asynchronous request to our protected API address
5. Check for success and display

Save your code, build it, and run it, (also ensure the Command API is running), you should see something like:

Where we have the JSON for our protected API endpoint returned.

Note: if you get at error similar to the following:

```
System.Security.AuthenticationException: The remote certificate is invalid
```

Just check that you took the steps in Chapter 2 to "trust" local SSL Certificates. If you're too lazy to pop back, just type the following at a command line and re-run the client:

```
dotnet dev-certs https --trust
```

Updating for Azure

In order for our API code to continue to work when we deploy to Azure, we're going to have to add the following Application Settings to our Command API on Azure, (remember we currently have these stored as *user secrets* in our local development instance…):

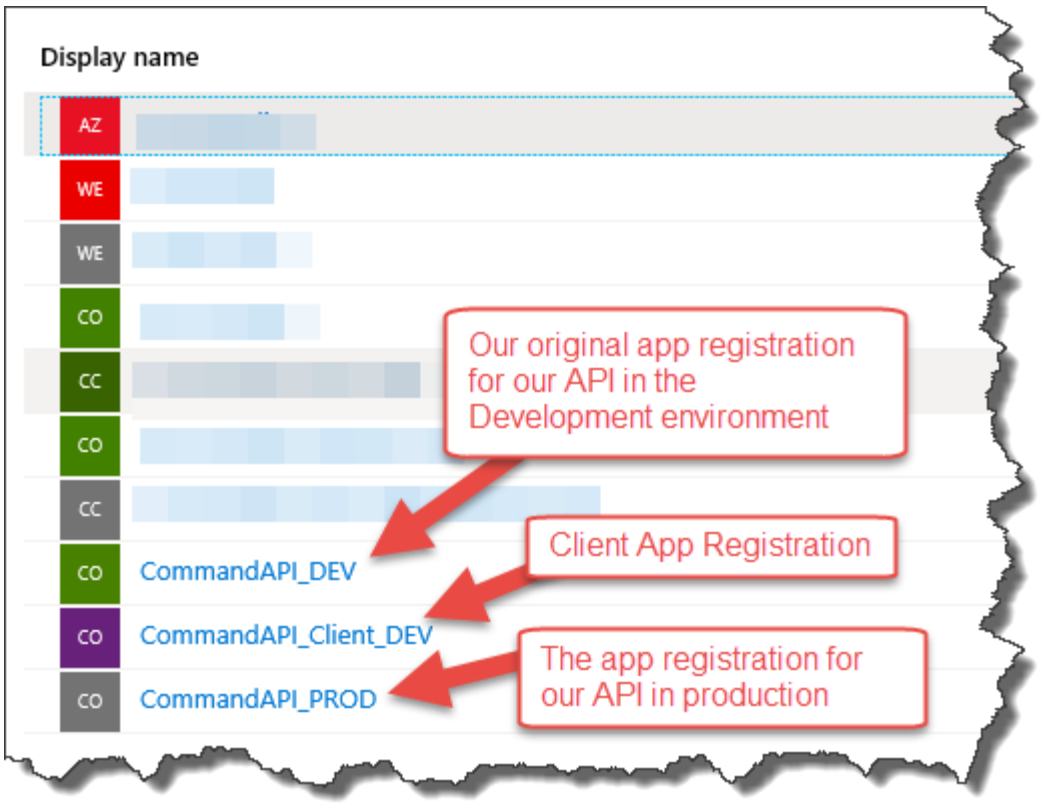
Config Element	Application Setting Name
The Login "Instance"	Instance
Our AAD Domain	Domain
The Tenant ID	TenantId
The Client ID	ClientId
The Application UD URL (Or Resource ID)	ResourceId

Before we do that though, while we could re-use the existing API App Registration, (CommandAPI_DEV), that we created for our "local" Command API, I think its good practice to set up a new "production" registration for our Command API.

Learning Opportunity: Rather than step through the exact same instructions to create a new "production" Command API registration, I'm going to leave you to do that now. As a suggestion call this new app registration: CommandAPI_PROD

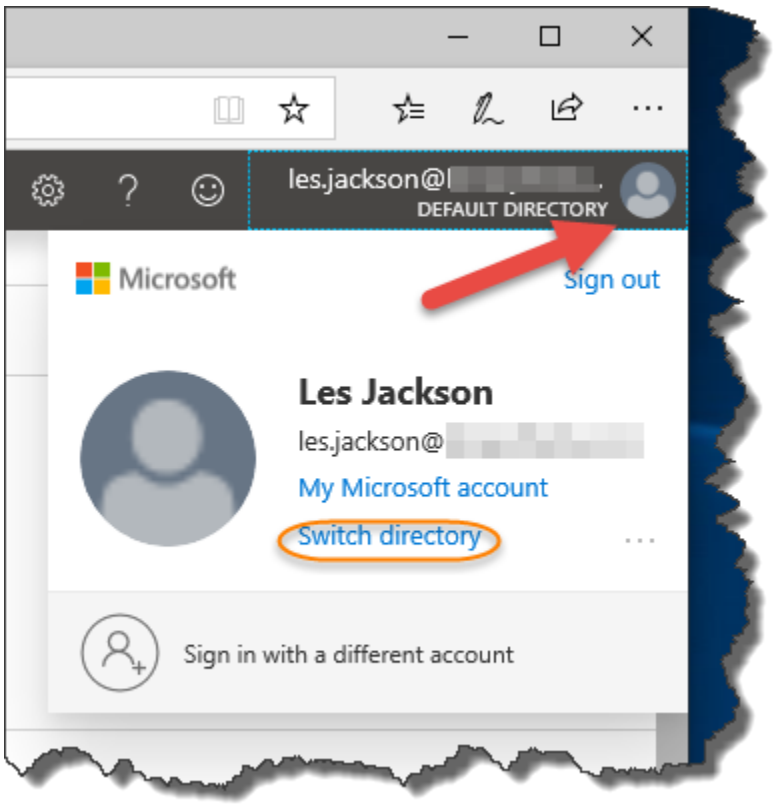
Come back here when you're done!

How did you go? Easy right? You should now have something similar to the following in your app registrations list:

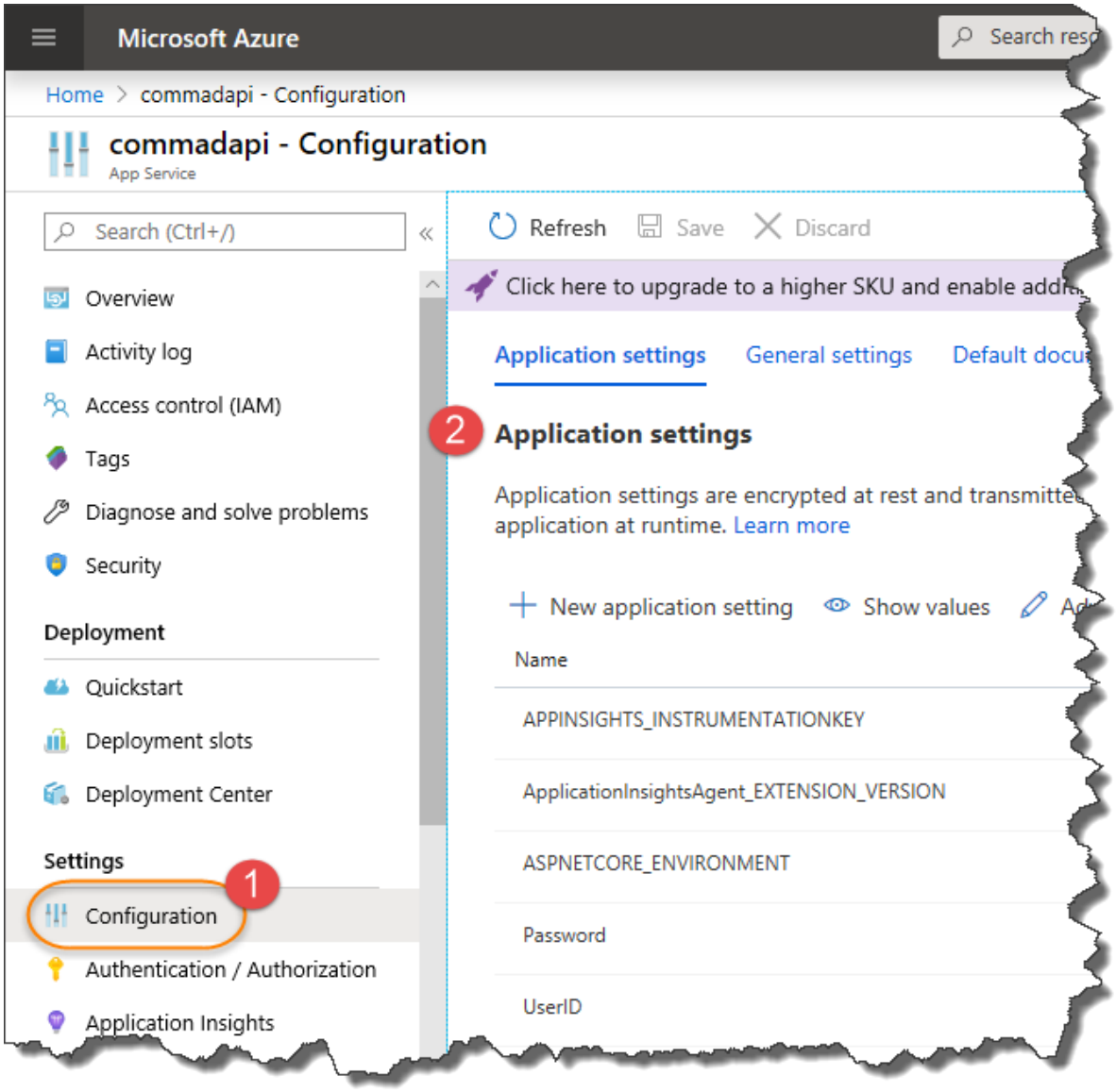


If like me you created your App Registrations in a different Azure Directory to your main one, (i.e. where all your resources are), I'd take a note of all of the values for things like: TenantId, ClientId and ResourceId in the App Registration you just created before you switch back to your main AAD to add the new Application Settings.

So, if needed, switch back to the AAD where you created the actual API App & Container instances:



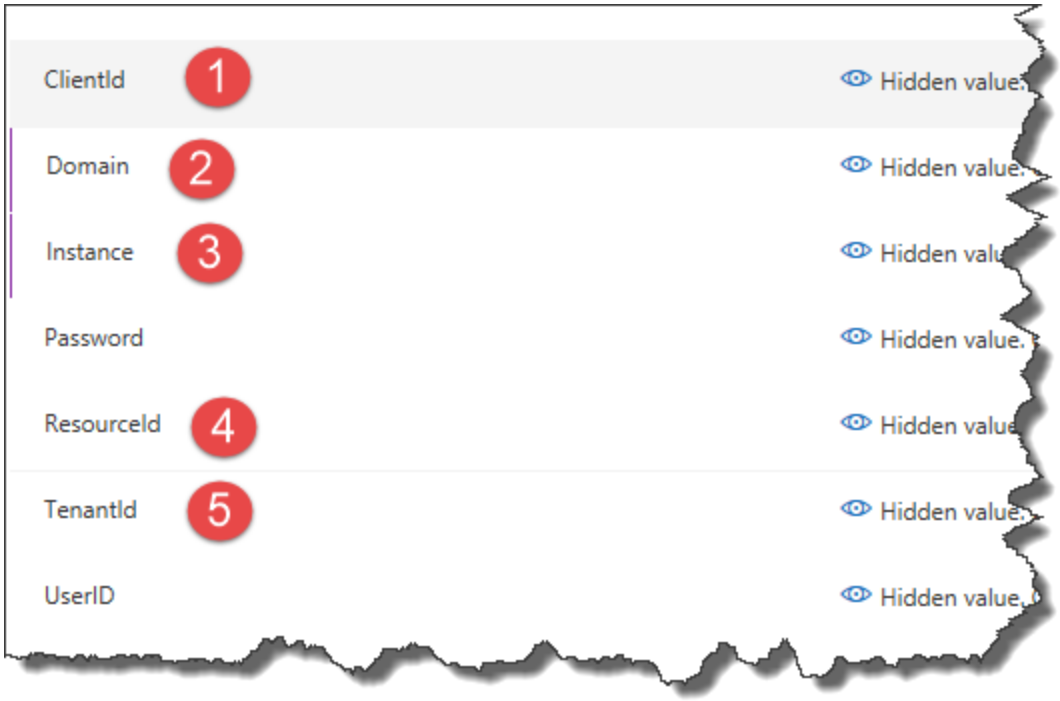
Select your Command API service, then “Configuration” to take you to the Application Settings screen:



Again, we've already added Application Settings before so I'm going to leave it to you, to add all the necessary Application Settings to allow our API to be correctly configured from an authentication perspective.

Warning! Make sure you give your Application Settings the exact same name as the User Secrets you set up before, with the relevant values from the Production API App registration (CommandAPI_PROD)…

Here you can see the new Application Settings I've added,



Client Configurations

To ensure our client can Authenticate to our Production API we should:

1. Create a Production Client App Registration on Azure
2. Update the necessary local settings in our Client App's ***applicationsettings.json*** file

Learning Opportunity: You have learned everything you need to know in order to complete this work, so again I'm going to leave it to you complete the 2 steps above.

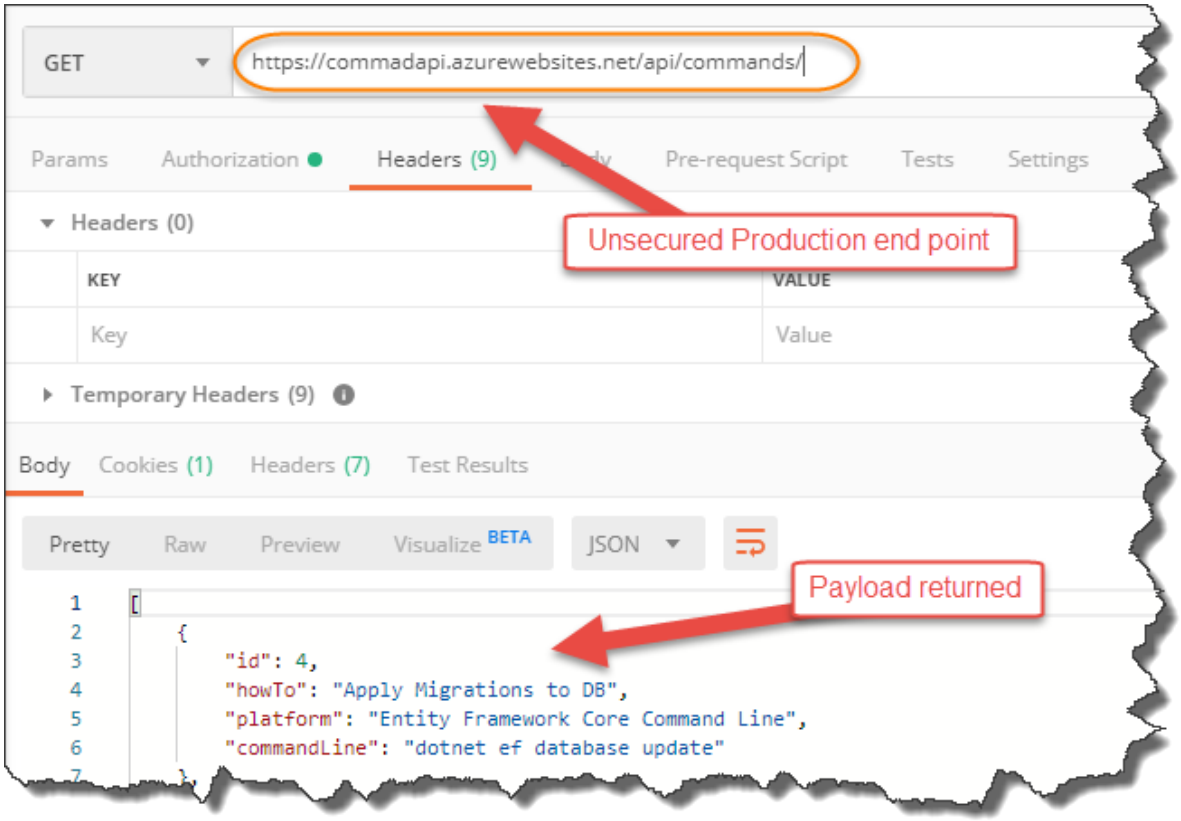
Take your time and remember to copy down the new values that are generated as part of the new production client app registration.

When done, come back here.

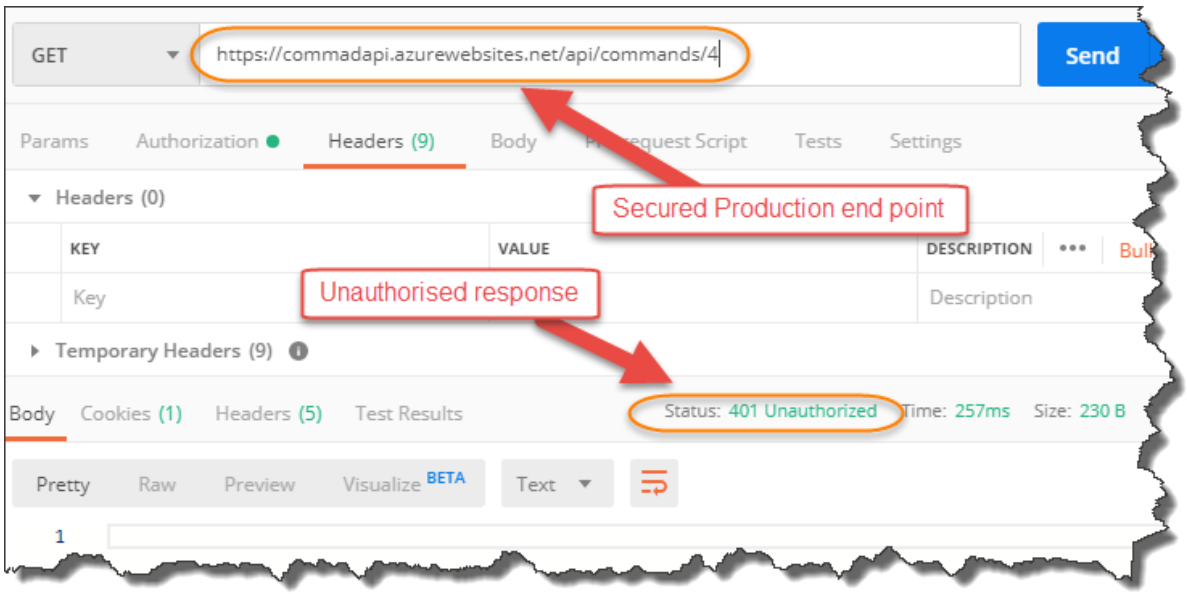
Deploy Our API to Azure

Back in our Command API Solution we just want to kick off a deploy to Azure, so if you don't have any pending commits, make an arbitrary change to your code, (insert a comment somewhere), and add / commit & push.

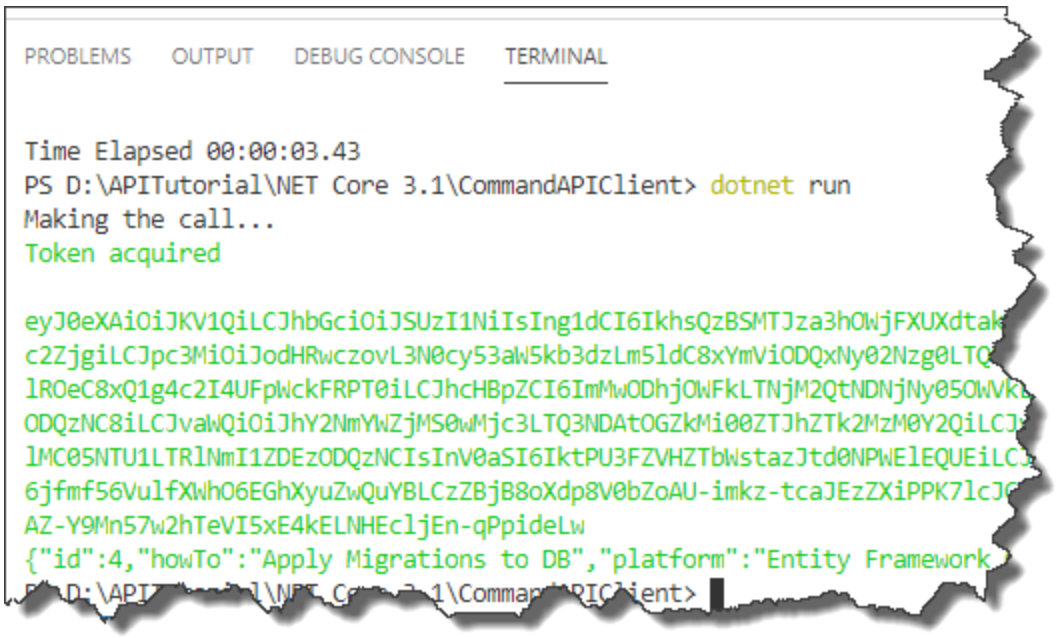
As before our Build Pipeline should succeed as should our deployment. Using something like Postman to call an unsecured endpoint should still work as before:



However as expected when we attempt to call the secured end point, (without a token), we should get a 401 Unauthorised response:



Turning top our client app, (with updated configuration), making a call through to our secured endpoint will yield a successful result:


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
Time Elapsed 00:00:03.43
PS D:\APITutorial\NET Core 3.1\CommandAPIClient> dotnet run
Making the call...
Token acquired
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6IkhsQzBSMTJza3hOWjFXUXdtak
c2ZjgiLCJpc3MiOiJodHRwczovL3N0cy53aW5kb3dzLm5ldC8xYmViODQxNy02Nzg0LTQ
lROeC8xQ1g4c2I4UFpWckFRPT0iLCJhcHBpZCI6ImMwODhjOWFkLTNjM2QtNDNjNy05OWVk
ODQzNC8iLCJvaWQiOiJhY2NmYWZjMS0wMjc3LTQ3NDAtOGZkMi00ZTJhZTk2MzM0Y2QiLCJ
lMC05NTU1LTRlNmI1ZDEzODQzNCIsInV0aSI6IktPU3FZVHZTbWstazJtd0NPWElEQUEiLCJ
6jfmf56VulfXWhO6EGhXyuZwQuYBLCzZBjB8oXdp8V0bZoAU-imkz-tcaJEzZXiPPK7lcJ
AZ-Y9Mn57w2hTeVI5xE4kELNHEcljEn-qPpideLw
{"id":4,"howTo":"Apply Migrations to DB","platform":"Entity Framework
PS D:\API
1\Command
ient>

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.
- There are so many clever, creative people out there sharing their knowledge, that without them I'd not be able to complete such a book