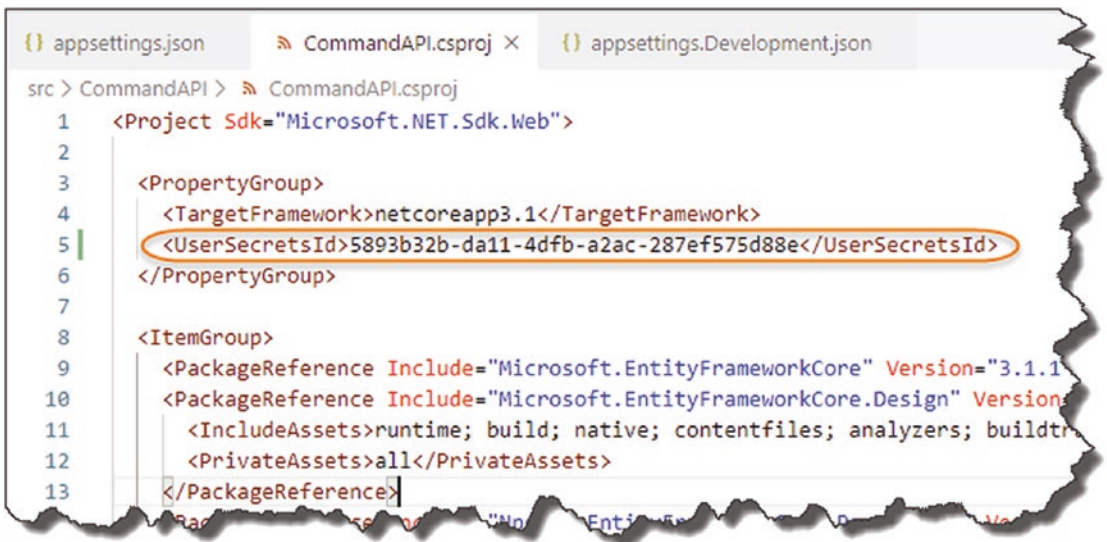


**Figure 8-12.** Select this GUID Format

- This should place the auto-generated GUID into the xml elements specified; see the following example.



**Figure 8-13.** GUID Inserted into the .CSPROJ File

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 to 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.Development.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 earlier, 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.



```
Content root path: D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI> dotnet user-secrets set "UserID" "cmddbuser"
Successfully saved UserID = cmddbuser to the secret store.
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI> 
```

**Figure 8-14.** Adding our first user secret

Repeat the same step and add the "Password" secret

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

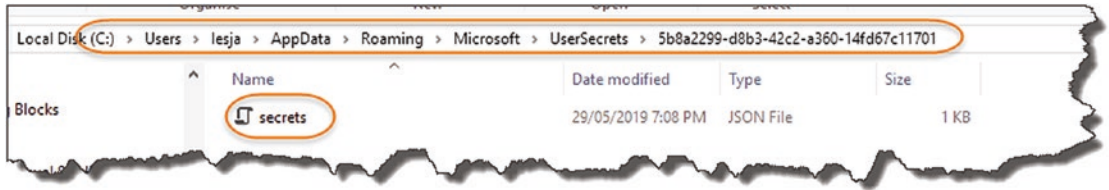
Again, you should get a similar success message.

## 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\\secrets.json
- Linux/OSX: ~/.microsoft/usersecrets/<user\_secrets\_id>/secrets.json

So, on my machine, it can be found here.<sup>5</sup>



**Figure 8-15.** Location of Secrets.Json on Windows

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

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

It's just a simple, non-encrypted JSON file.

## Code It Up

OK, so now to 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.

<sup>5</sup>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.



**Figure 8-16.** Removal of sensitive connection string attributes

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

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "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 (noting the inclusion of the new using statement at the top):

```
.
.
.
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();
            services.AddScoped<ICommandAPIRepo, SqlCommandAPIRepo>();
        }
    }
}

```

- 
- 
- 

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

```

using Microsoft.EntityFrameworkCore;
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();

            services.AddScoped<ICommandAPIRepo, SqlCommandAPIRepo>();
        }
    }
}

```

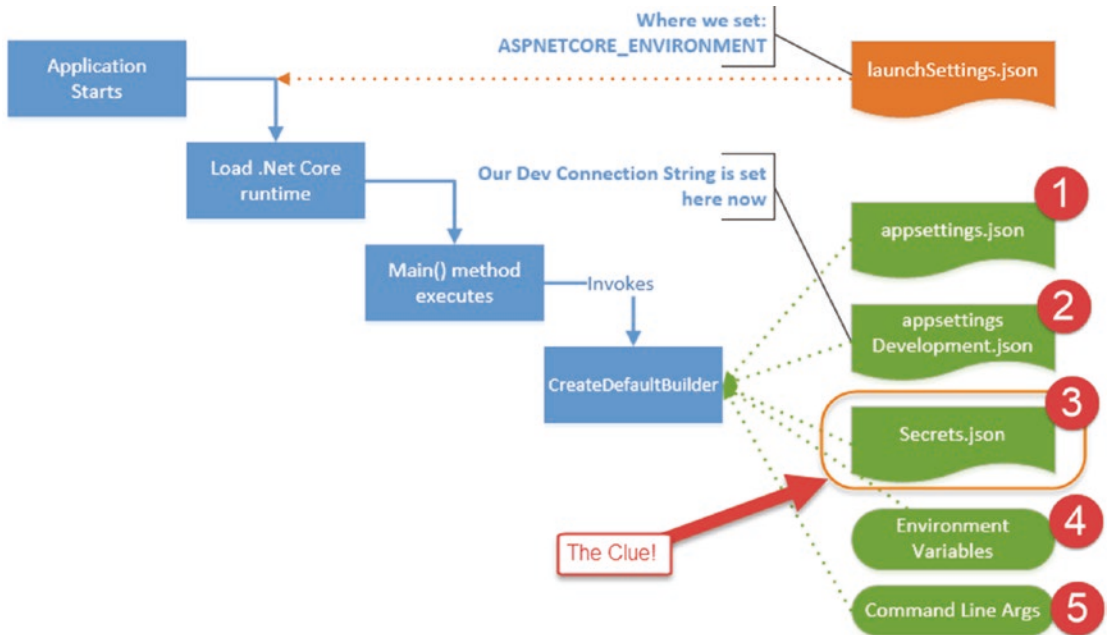
**Figure 8-17.** Updated Startup class

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 Checkpoint** 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.



**Figure 8-18.** Revisit of precedence

The .NET Configuration layer by default provides us access to the configuration sources as shown in Figure 8-18; 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 (User ID and 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)!

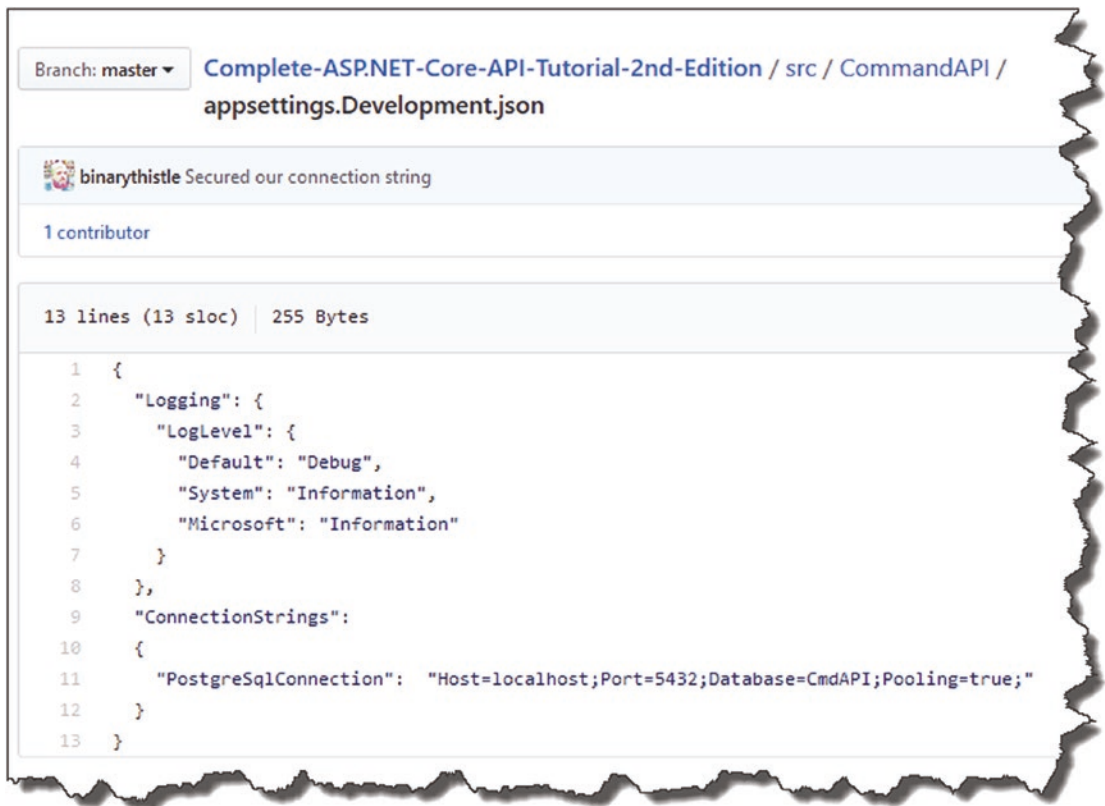


Figure 8-19. Clean Appsettings.json on GitHub



## CHAPTER 9

# Data Transfer Objects

## Chapter Summary

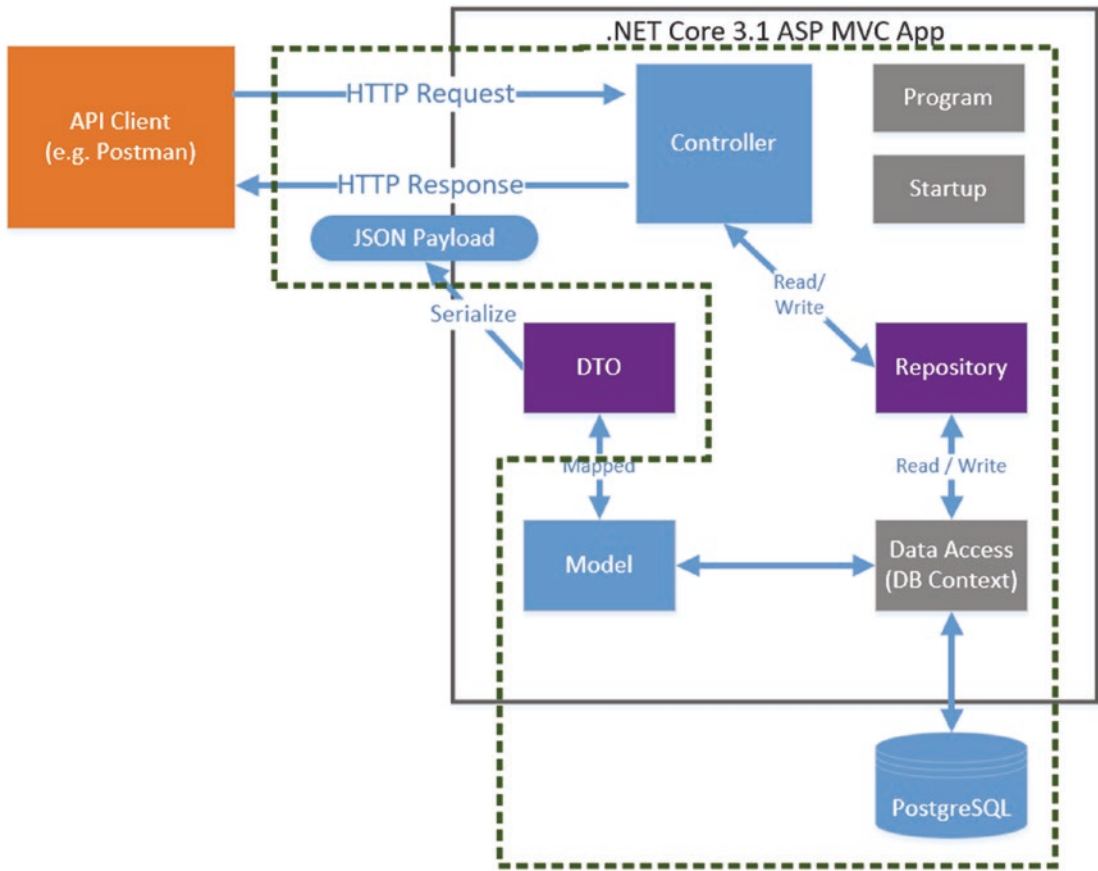
In this chapter we'll complete the final piece of our architectural puzzle and introduce Data Transfer Objects.

## When Done, You Will

- Understand what Data Transfer Objects (DTOs) are.
- Understand why you should use DTOs.
- Have started to implement DTOs in our solution.

## Architecture Review

Outlining what we've either (a) started to implement or (b) fully implemented, our architectural is evolving nicely.



**Figure 9-1.** Architecture Progress

To summarize, we've

- Fully implemented our Model
- Fully implemented our *Repository Interface*
- Partially implemented our *Concrete Repository Implementation* (using the DB Context)
- Fully implemented our DB Context
- Fully implemented our Database
- Partially completed our Controller (we still have four actions to complete)

We have not yet started on the DTOs, so that is what we'll turn our attention to in this chapter.

## The What and Why of DTOs

To answer both what DTOs are and why you'd use them, let's take a look at what we have implemented so far:

- We have implemented two Controller Actions that return serialized Command objects to the consumer.

What's wrong with that?

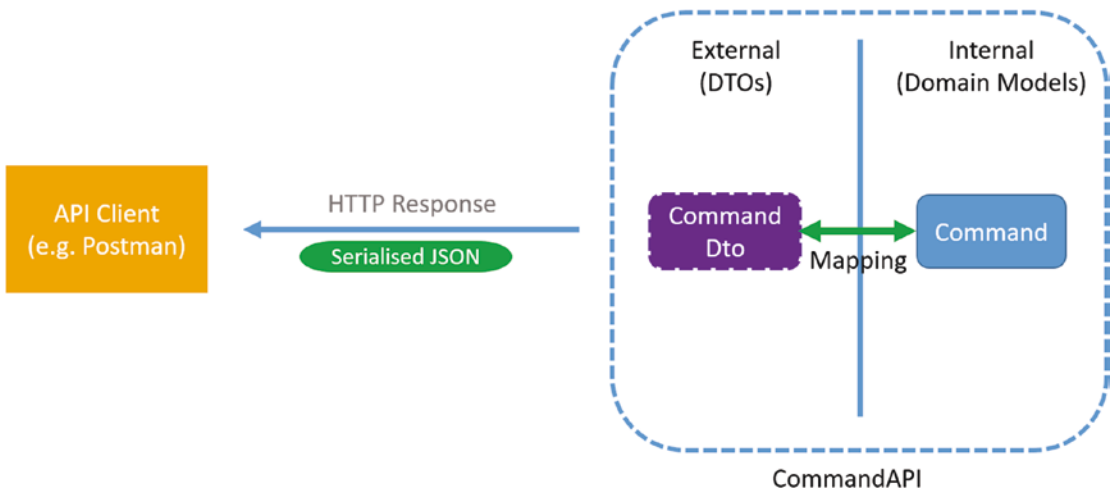
We are basically exposing "internal" domain detail out to our consumers; this has the following potential consequences:

- We may be exposing "sensitive" information.
- We may be exposing irrelevant information.
- We may be exposing information in the wrong format.
- We have "coupled" our internal implementation to our external contract, so changing our internals will be difficult if we want to maintain our contract (or we break the contract altogether – not advised).

This is not a great situation – so what is the answer?

## Decouple Interface from Implementation (Again)

Again (similar to what we did with our repository), we want to decouple our external contract (our interface) from our internal implementation (our Domain model). This is where DTOs come in; observe the following diagram:

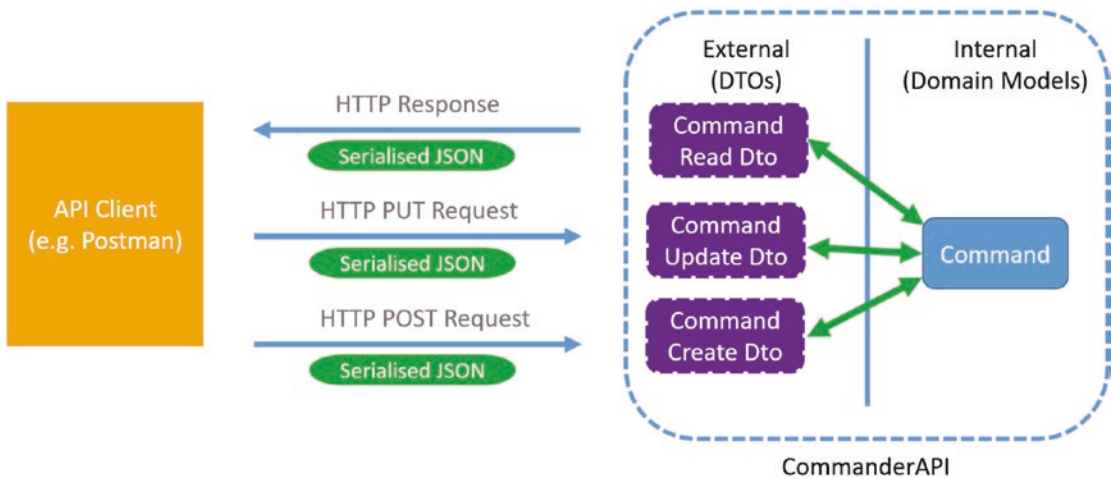


**Figure 9-2.** Example of Read DTO

DTOs are “mapped” to our internal Domain Model classes and represented externally as part of the contract, thus decoupling our implementation from our interface. We can then benefit from

- **Change Agility:** We can feel free to change our internal implementation, and as long as we perform the appropriate mapping back to our DTO, our interface remains intact.
- We can remove both sensitive and irrelevant implementation detail from our DTOs
- As part of our “mapping” operation, we can augment our internal representations and present them in an entirely new way (e.g., combining First and Last name and presenting externally as Full Name).

Taking it further, depending on what type of operation we are performing (Read, Create, Update, etc.), we may employ different variants of our DTO to cater for each, as shown below.



**Figure 9-3.** We can have DTOs for different actions

I’ll explain this concept as we start to implement; just bear it in mind for now. With that I think we should move on to coding.

## Implementing DTOs

To implement DTOs, we need to do the following:

- Create our DTO classes.
- Figure out how to perform the “mapping” mentioned previously.

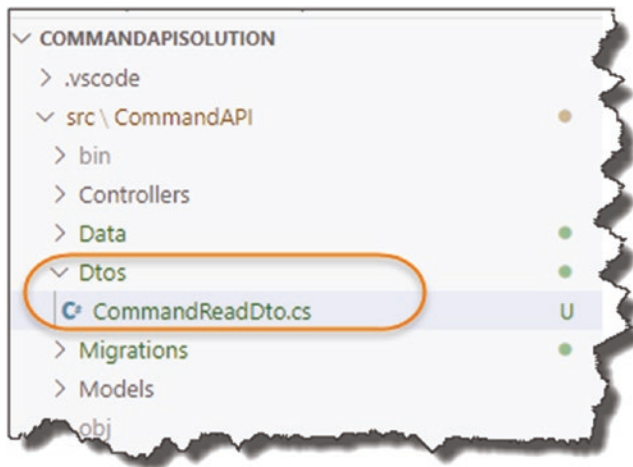
The first point is actually very straightforward, but it is the second point that introduces more options and/or complexity. We could simply perform the mapping operations manually in code we write ourselves, and while this *may* be ok for small objects, as our models grow in size and complexity, this would become

- Tiresome
- Error-prone

Therefore, we are going to employ an automation framework (called AutoMapper) to perform the mapping function for us. While this does require a little bit more upfront effort, believe me it’s worth it! Before we get involved with AutoMapper, let’s start with implementing our DTO classes.

## Create Our DTOs

Back in API Project (make sure the webserver has stopped), add a new folder to the root of our API project called *Dtos*, and add a file to it called *CommandReadDto.cs* as shown in Figure 9-4.



**Figure 9-4.** New Dtos Folder and CommandReadDto.cs file

As the name suggests, we will use this DTO when we perform any read operation, so in effect this is the object that will be serialized and sent back to the client whenever they perform a GET request.

Now at this point, you may ask yourself the question: Won't the DTO be *exactly* the same as our Command model? And to be honest, yes it will, but is nonetheless still a valid use case. With that in mind, complete the code for our DTO as follows:

```
namespace CommandAPI.Dtos
{
    public class CommandReadDto
    {
        public int Id {get; set;}

        public string HowTo {get; set;}

        public string Platform {get; set;}
    }
}
```

```

    public string CommandLine {get; set;}
  }
}

```

You can see this has more than a passing resemblance to our Command model. You will notice though that in this case, there are no Data Annotations (we will be utilizing them again, just not for this DTO).

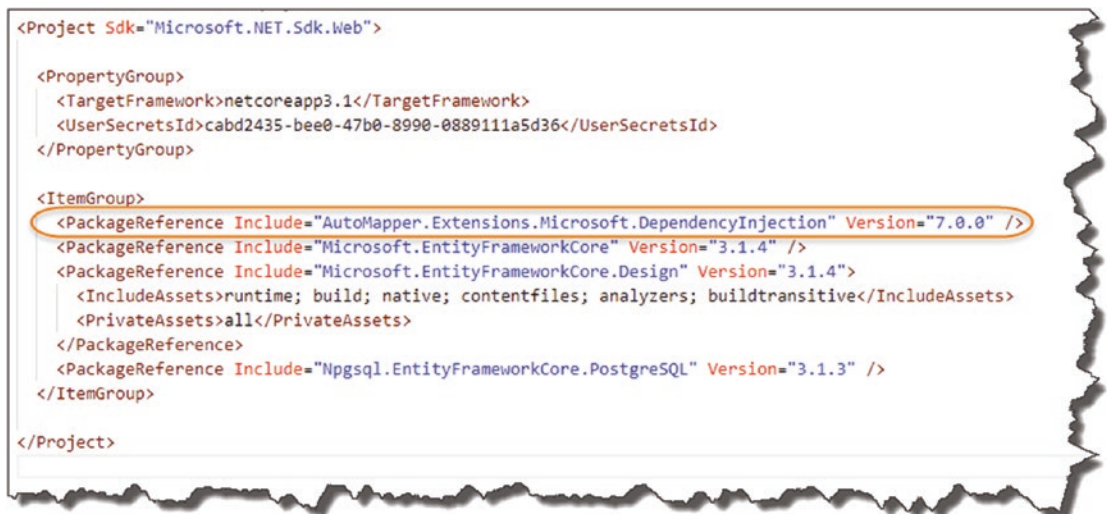
And that's essentially it for our first DTO class – I told you it was simple. We now need to move on to setting up AutoMapper.

## Setting Up AutoMapper

The first thing we need to do is install another package in our API Project, so ensure the webserver is not running (CTRL + C if it is), and at a command prompt “in” the API project folder (*CommandAPI*), enter the following:

```
dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection
```

This will install the AutoMapper package; confirm this by checking the *.csproj* file for the API project, and you should see something similar to Figure 9-5.



**Figure 9-5.** Reference to Automapper

To use AutoMapper we move over to our Startup class and register it in our Service Container by adding the following lines (making it available to us throughout our application via our old friend *Dependency Injection*):

```
.  
using AutoMapper;  
.br/>.br/>.br/>services.AddControllers();  
  
//Add the line below  
services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());  
  
services.AddScoped<ICommandAPIRepo, SqlCommandAPIRepo>();
```

To put it in context, I've highlighted those new inclusions in Figure 9-6.



**Figure 9-6.** AutoMapper service registered



---

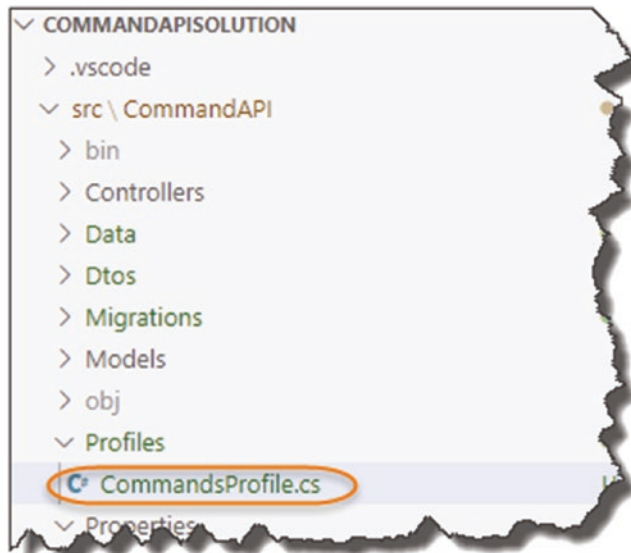
**Note** The registration of AutoMapper can really be placed anywhere in the `ConfigureServices` method; I've just chosen to place it here in case you're wondering. For more detail on how to use AutoMapper with Dependency Injection in .NET Core, refer to the AutoMapper Docs.<sup>1</sup>

---

That's our setup of AutoMapper complete – see, it wasn't that bad; we now need to move onto using it.

## Using AutoMapper

In order to use AutoMapper, we need *somewhere* to configure the mapping of our Model to our DTO, in this case mapping `Command` to `CommandReadDto`, and we do that via a “profile.” To start using AutoMapper profiles, create another folder in the root of our **CommandAPI** project called **Profiles**, and in there create a file called **CommandsProfile.cs** as so.



**Figure 9-7.** New Profiles folder and CommandsProfile.cs file

---

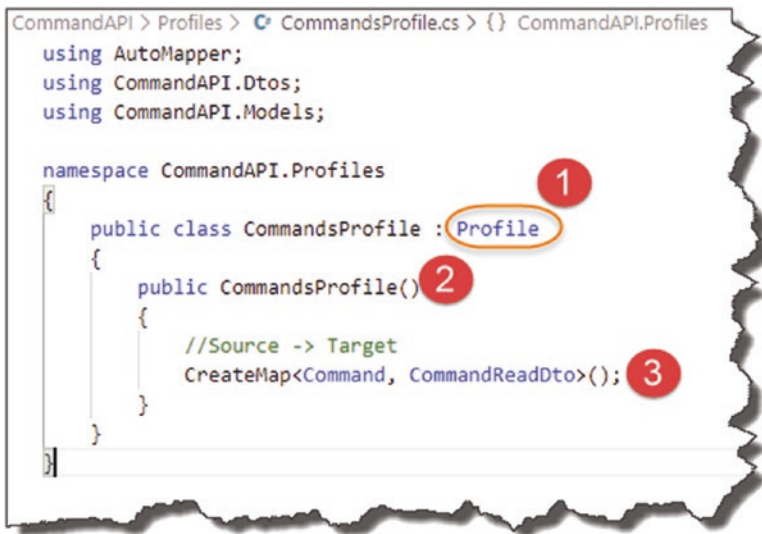
<sup>1</sup><https://docs.automapper.org/en/stable/Dependency-injection.html#asp-net-core>

Now add the following code to the file:

```
using AutoMapper;
using CommandAPI.Dtos;
using CommandAPI.Models;

namespace CommandAPI.Profiles
{
    public class CommandsProfile : Profile
    {
        public CommandsProfile()
        {
            CreateMap<Command, CommandReadDto>();
        }
    }
}
```

The class can be explained in Figure 9-8.



**Figure 9-8.** Our first AutoMapper Mapping

1. Our class inherits from `Automapper.Profile`.
2. We add a simple class constructor.

3. We use the `CreateMap` method to map our source object (`Command`) to our target object (`CommandReadDto`).

And that's our mapping complete. It's so straightforward in our case as the property names of both classes are identical; `AutoMapper` can derive the mappings easily.

Finally, we want to update our Controller to return our DTO representation (`CommandReadDto`) instead of `Command Model` for both our GET Actions. Before we do that though, we need to make `AutoMapper` "available" to our Controller. Any ideas how we do that?

For those of you that said *Constructor Dependency Injection*, well done! That's exactly what we're going to do. So over in our Controller, add the following highlighted code:

```

.
.
using AutoMapper;
using CommandAPI.Dtos;

namespace CommandAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CommandsController : ControllerBase
    {
        private readonly ICommandAPIRepo _repository;
        private readonly IMapper _mapper;

        public CommandsController(ICommandAPIRepo repository, IMapper mapper)
        {
            _repository = repository;
            _mapper = mapper;
        }
    }
.
.
.

```

To explain what we've done, have a look at the changes in context in Figure 9-9.

```
using System.Collections.Generic;
using AutoMapper;
using CommandAPI.Dtos;
using CommandAPI.Data;
using CommandAPI.Models;
using Microsoft.AspNetCore.Mvc;

namespace CommandAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CommandsController : ControllerBase
    {
        private readonly ICommandAPIRepo _repository;
        private readonly IMapper _mapper;

        public CommandsController(ICommandAPIRepo repository, IMapper mapper)
        {
            _repository = repository;
            _mapper = mapper;
        }

        [HttpGet]
```

**Figure 9-9.** *Injecting AutoMapper into the controller*

1. Added our two new using directives.
2. Created a new read-only field to hold an instance of IMapper .
3. An instance of IMapper will be injected by the DI system into our constructor.
4. We assign our injected instance to the private member \_mapper for further use.

This pattern should be very familiar to you now as we have used it multiple times within our API; the only point of note is that you can see we can inject *multiple instances* into our Constructor.

We can now update our two existing controller actions to make use of AutoMapper and return our DTO representation to our consumers as shown by the highlighted code in the following:

```
.  
.  
[HttpGet]
```

```

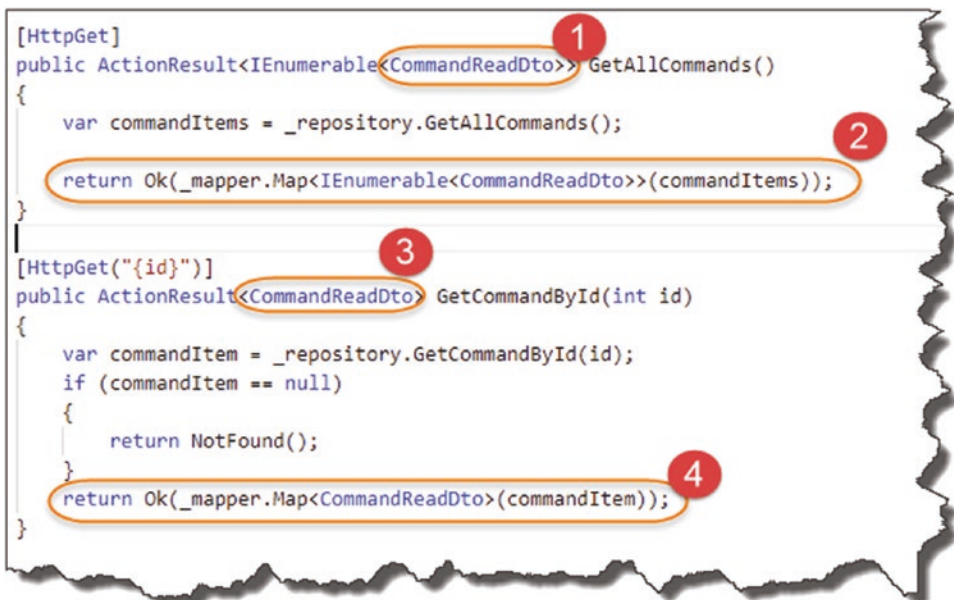
public ActionResult<IEnumerable<CommandReadDto>> GetAllCommands()
{
    var commandItems = _repository.GetAllCommands();

    return Ok(_mapper.Map<IEnumerable<CommandReadDto>>(commandItems));
}

[HttpGet("{id}")]
public ActionResult<CommandReadDto> GetCommandById(int id)
{
    var commandItem = _repository.GetCommandById(id);
    if (commandItem == null){
        return NotFound();
    }
    return Ok(_mapper.Map<CommandReadDto>(commandItem));
}
.
.

```

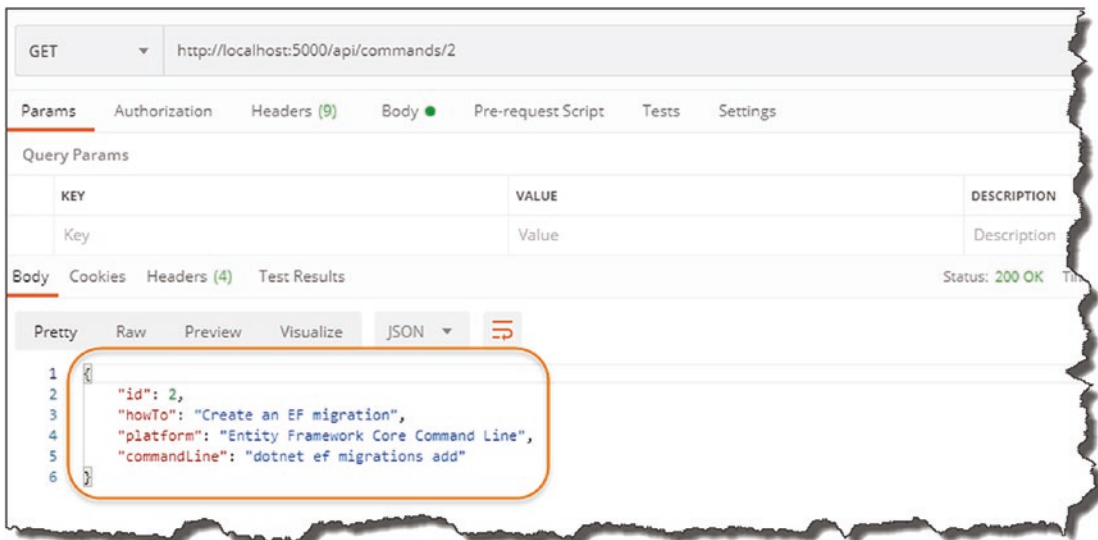
The changes are shown and explained in Figure 9-10.



**Figure 9-10.** Use of AutoMapper in our 2 GET Controller Actions

1. We ensure our ActionResult return type is changed from Command to CommandReadDto.
2. We call the Map method on our \_mapper instance. It maps our collection of Command objects to an IEnumerable of CommandReadDtos that we return in our OK method.
3. We ensure our ActionResult return type is changed from Command to CommandReadDto.
4. Does the same thing as #1, except we are working with a single Command object as the source and returning (if available) a single CommandReadDto object in our OK method.

Save all your code and run as before.



**Figure 9-11.** *CommandReadDTO Returned*

The “problem” is that it looks exactly the same as before (well it’s not a *problem*; technically it’s working). So just to demonstrate what is possible with DTOs, let’s comment out the Platform property on our CommandReadDto, as shown here:

```

namespace CommandAPI.Dtos
{
    public class CommandReadDto
    {
        public int Id {get; set;}

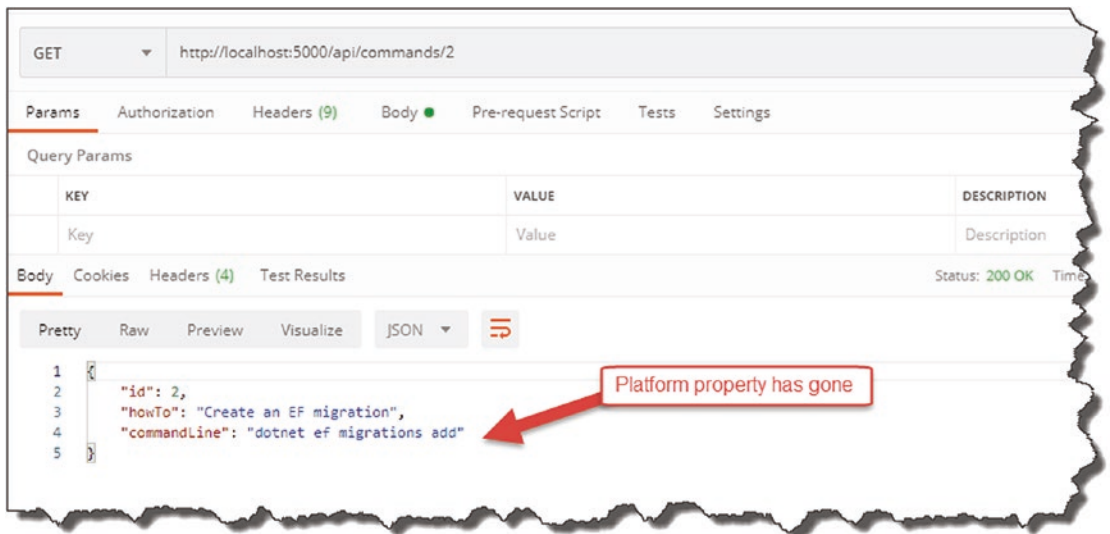
        public string HowTo {get; set;}

        //Comment out the line below
        //public string Platform {get; set;}

        public string CommandLine {get; set;}
    }
}

```

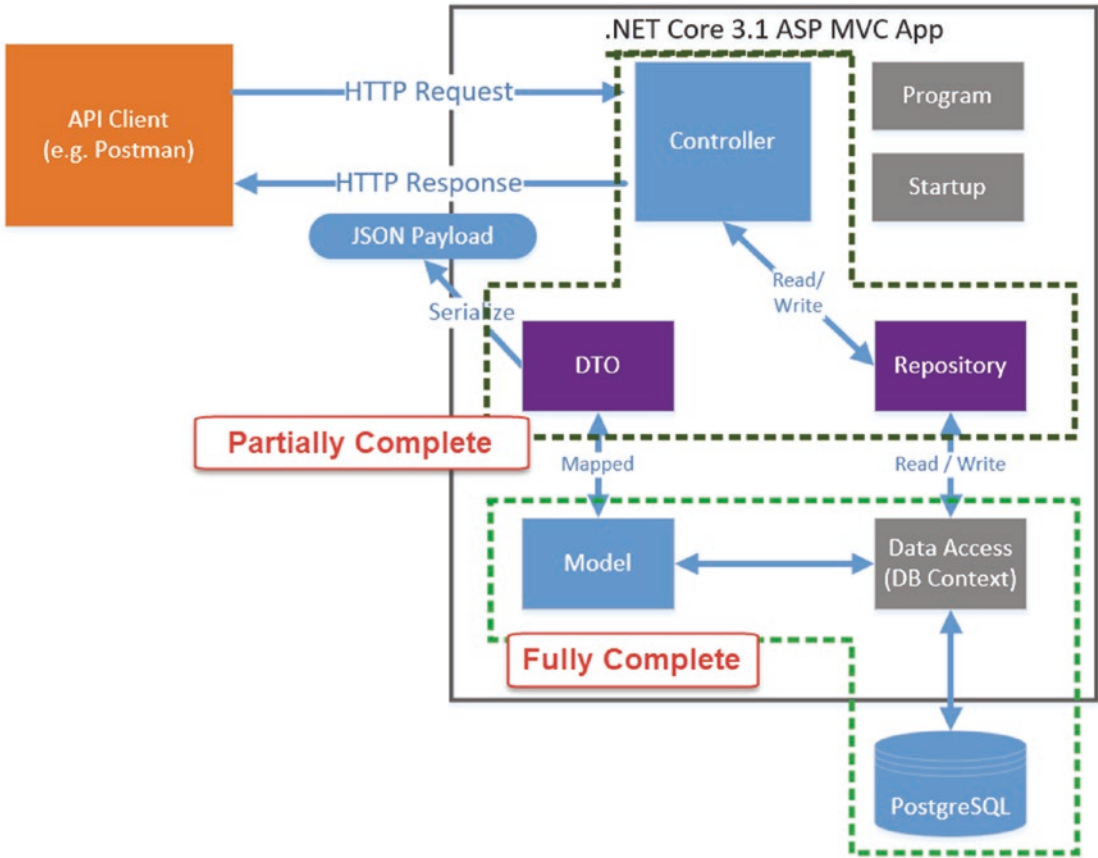
Once you've saved your changes, restart the webserver and rerun your Postman query.



**Figure 9-12.** *CommandReadDto* returned with platform removed

You'll see that our DTO representation has in fact been returned! Once you're happy, *revert those changes* so we're returning the full object.

A quick look at our application architecture and you can see that we have now completed the groundwork for all our architectural components (although some components are only partially complete as depicted in Figure 9-13):



**Figure 9-13.** Architecture Check

We can leave DTOs there for now, but we will return to them as we build out our remaining controller actions next.



## CHAPTER 10

# Completing Our API Endpoints

## Chapter Summary

In the last few chapters, we have put a lot of work into the underlying architectural fabric of our API, but we've only implemented two of our endpoints (controller actions). In this chapter we address this and move up a gear to finalize our remaining four endpoints.

## When Done, You Will

- Understand how data changes are persisted by Entity Framework Core.
- Have fully implemented our Create (POST) resource endpoint.
- Have fully implemented our two Update (PUT and PATCH) resource endpoints.
- Have fully implemented our Delete (DELETE) resource endpoint.
- Understand more about REST best practice.

## Persisting Changes in EF Core

So far, we have used an EF Core DB Context (via our Repository) to read data from our PostgreSQL database and return it to our consumer (using DTOs). These endpoints are considered “safe” as they cannot change the data in our database; they can only read it.

Our four remaining endpoints, (shown below) are slightly more *dangerous* in that they are able to *change* the data in our database, or to use a slightly more dramatic term – they are considered “unsafe.”

Verb	URI	Operation	Description
POST	/api/commands	Create	Create a new resource
PUT	/api/commands/{Id}	Update (full)	Update all of a single resource (by Id)
PATCH	/api/commands/{Id}	Update (partial)	Update part of a single resource (by Id)
DELETE	/api/commands/{Id}	Delete	Delete a single resource (by Id)

The reason I’m calling out this fairly obvious point is because I want to shine a little light on how changes to data occur in EF Core and in particular when using a DB Context, as it becomes relevant in the sections that follow.

## DB Context Tracks Changes

Let’s take a simple example of adding a new Command resource to the PostgreSQL DB; using our DB Context, we will

1. Obtain the Command object to be added (don’t worry where we get this for now).
2. Add that Command object to the CommandItems DBSet in our DB Context.
3. Save the changes *pending* on the DB Context.
4. Changes will then be reflected in the PostgreSQL database.

The point I’m making here is that just by adding (or removing/updating) objects on our DB Context does not mean those changes will be automatically reflected down on the PostgreSQL database. We need to further *Save* the pending changes for that to happen.

What you can take from this is that the DB Context tracks (multiple) changes to the data “internally,” be they create, update, or delete operations, but will only persist those changes to the DB when we explicitly tell it to – by *Saving Changes*.

Again, I wanted to call that out here, as it becomes relevant in a couple of areas as we move into implementing our remaining endpoints.

## The Create Endpoint (POST)

The next endpoint we want to implement is the “Create” endpoint, which gives us the ability to add resources to our DB. A quick reminder of our high-level definition is shown here.

Verb	URI	Operation	Description
POST	/api/commands	Create	Create a new resource

We’ll also introduce some other attributes that will help us understand, build, and ultimately test our endpoint; they are shown in the following table.

Attribute	Description
Inputs (x1)	<p>The “command” object to be created.</p> <p>This will be added to the request body of our POST request; an example is shown here:</p> <pre>{   "howTo": "Example how to",   "platform": "Example platform",   "commandLine": "Example command line" }</pre>
Process	Will attempt to add a new command object to our DB
Success	<ul style="list-style-type: none"> <li>• HTTP 201 Created Status</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>• Newly Created Resource (response body)</li> <li>• URI to newly created resource (response header)</li> </ul>
Failure Outputs	<ul style="list-style-type: none"> <li>• HTTP 400 Bad Request</li> <li>• HTTP 405 Not Allowed</li> </ul>
Safe	No – Endpoint can alter our resources
Idempotent	No – Repeating the same operation will incur a different result

Most of this should make sense, but there are probably three callouts for me before we move onto coding.

## Input Object

You'll notice that the object we can expect to attach to the request body in order to create a resource *does not* contain an "Id" attribute – why is that? Simply because the responsibility for creating a unique id has been devolved down to our PostgreSQL Database. When a new row is inserted to our `CommandItems` table, it is at that point that a new (unique) id will be created for us. (You should remember this when we manually added data to our DB via SQL commands in Chapter 7.)

---

 **Learning Opportunity** As our input command object is different to our *internal domain command model*, what technique could we use to deal with this?

---

## Success Outputs

The issuing of a 201 Created Http Status code is self-explanatory, but what you may not have expected is that we should pass back both:

- The newly created resource (with Id)
- A URI (or "route") to where we can obtain that resource again if needed

The second point in particular is to allow us to align with the REST architectural principles, so we'll follow it here in our API. Further discussion on this can be found in this article on REST.<sup>1</sup>

## Idempotency

I've already mentioned "safety" in the opening to this chapter, but I've also included whether this endpoint is "idempotent." What is idempotency?

*An operation is idempotent when performing the same operation again gives the same result.*

So, in the case of our create endpoint, the first time we fire off a request (assuming it's successful), we'll get the newly created resource returned. If we perform the *exact same*

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Representational\\_state\\_transfer](https://en.wikipedia.org/wiki/Representational_state_transfer)

*request again*, we'll get a *different result*. Why? Because we'll have created a whole new resource (with a new Id) in addition to the first one. Our create endpoint is therefore *not* idempotent.

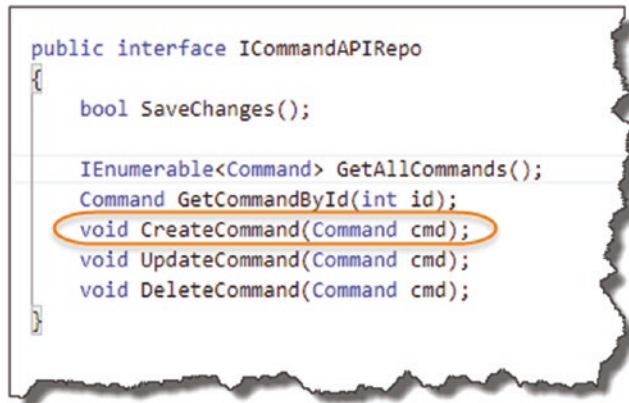
Compare that with one of our existing GET requests; we can perform the same request time and time again and get the *same result* – these *are* idempotent.

Why have I included this? Simply because I've seen the use of the term increase dramatically over the short term (although the concept is not new), so I would be doing you a disservice if I didn't introduce it to you here.

Enough theory – lets code.

## Updating the Repository

Let's work from “the ground up” and return to our repository. Refer to Figure 10-1 that details the repository interface definition ICommandAPIRepo.



```
public interface ICommandAPIRepo
{
    bool SaveChanges();

    IEnumerable<Command> GetAllCommands();
    Command GetCommandById(int id);
    void CreateCommand(Command cmd);
    void UpdateCommand(Command cmd);
    void DeleteCommand(Command cmd);
}
```

**Figure 10-1.** *CreateCommand Repository Method*

We can see that for the highlighted repository method, we simply require a `Command` object to be passed in (and, as inferred, added to our DB Context – and ultimately our PostgreSQL database). We don't expect anything returned back. Moving over to our concrete implementation, `SqlCommandAPIRepo`, add the following code to the `CreateCommand` method (making sure to include the `using System` namespace):

```
using System
```

- 
- 
-

```

public void CreateCommand(Command cmd)
{
    if(cmd == null)
    {
        throw new ArgumentNullException(nameof(cmd));
    }
    _context.CommandItems.Add(cmd);
}

```

To put these in context, see Figure 10-2.



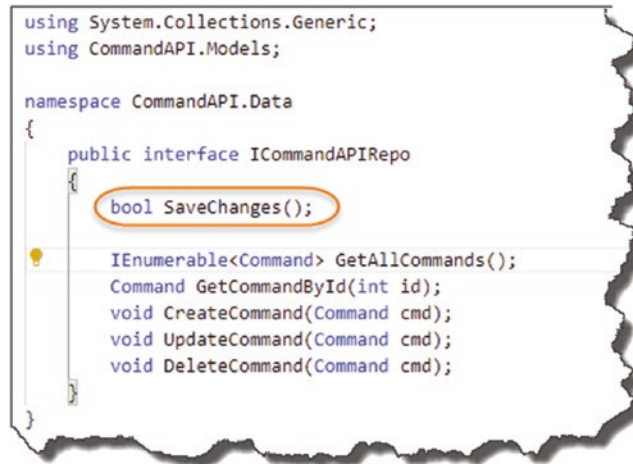
**Figure 10-2.** *Implementation of CreateCommand*

1. We check to see if the object passed in is null, and if so throw an exception (this case will be caught in our controller when it comes to validate the command model we have; however, we don't know where else our repository implementation may be used, so it's good practice to put code like this in any way).
2. Using our DB Context instance (`_context`), we reference our `CommandItems` DB Set and call the `Add` method, passing in our `Command` object.

Going back to our discussion on how data is persisted in EF Core, you'll be aware that just calling this method *will not persist our changes down to the DB*; at this point we only have the `Command` object added to the DB Context/DB Set.

## Implement SaveChanges

Returning once again to our repository interface definition, `ICommandAPIRepo`, you'll remember a mysterious method definition (well probably not *that* mysterious anymore).



```
using System.Collections.Generic;
using CommandAPI.Models;

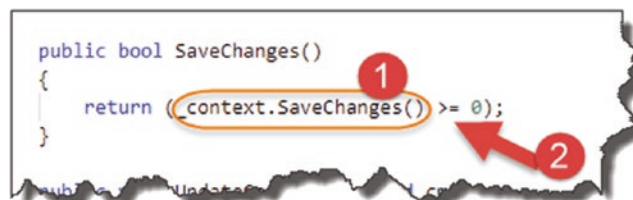
namespace CommandAPI.Data
{
    public interface ICommandAPIRepo
    {
        bool SaveChanges();
        IEnumerable<Command> GetAllCommands();
        Command GetCommandById(int id);
        void CreateCommand(Command cmd);
        void UpdateCommand(Command cmd);
        void DeleteCommand(Command cmd);
    }
}
```

**Figure 10-3.** The `SaveChanges` Interface method

Well we need to implement that now in our concrete implementation, so back over in `SqlCommandAPIRepo`, add the following code to the `SaveChanges` method:

```
public bool SaveChanges()
{
    return (_context.SaveChanges() >= 0);
}
```

In context, these changes look like this.



```
public bool SaveChanges()
{
    return (context.SaveChanges() >= 0);
}
```

**Figure 10-4.** Implementation of `SaveChanges`

1. Call the `SaveChanges` method on our DB Context; this replicates all pending changes on the DB Context down to the PostgreSQL DB and persists them.
2. We use this comparison operator to return `true` if the result of save changes is greater than or equal to 0 (this will be a positive integer reflecting the number of entities affected or of course 0 if none are<sup>2</sup>).

We'll use the `SaveChanges` repository operator from our Controller, and we'll use it for all four of our remaining "unsafe" endpoints in order to persist data (not just our Create endpoint).

That's our repository sorted for our Create method, what's next?

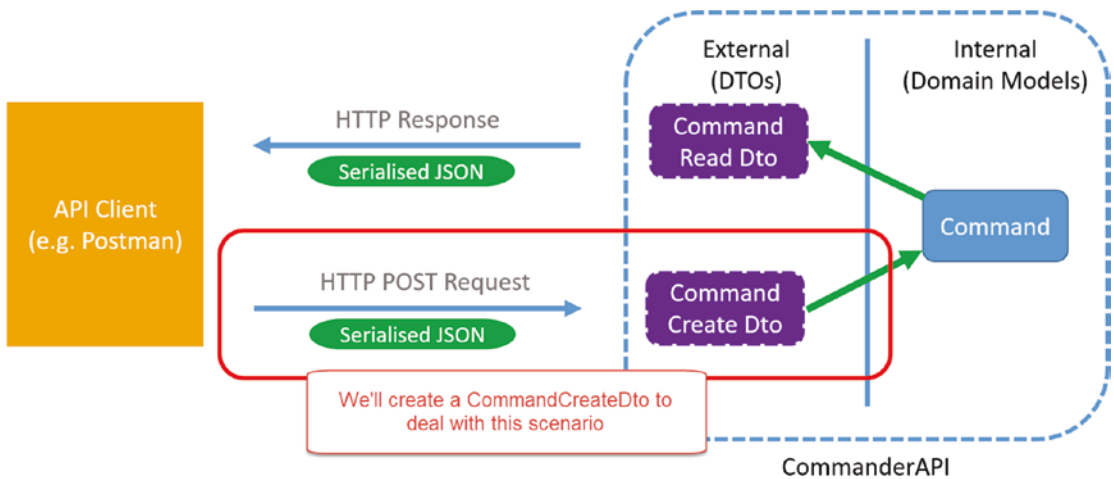
## CommandCreateDto

Earlier in this chapter I asked what *technique* could we use to deal with the fact that the representation of the command resource we expect from our POST request will be different to our internal command model? For those of you that answered with "DTOs," give yourself a pat on the back – yes we're going to use a DTO to represent the input for our command resource and, using AutoMapper, map it back to an internal command model we can pass over to our repository, I've shown a slightly simplified version of this scenario in Figure 10-5.

---

<sup>2</sup><https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbcontext.savechanges>

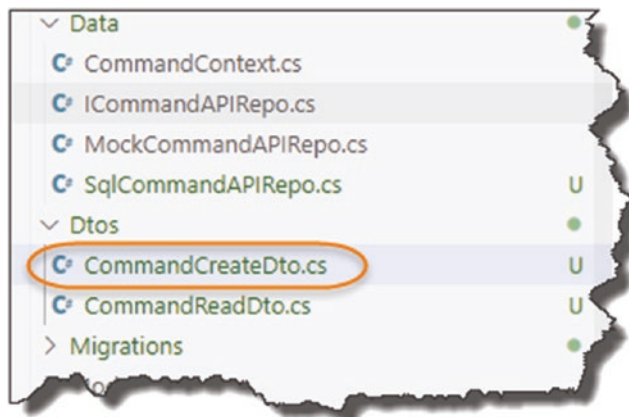




**Figure 10-5.** *CommandCreate DTO Example*

## Create the New DTO

Back over in our project, create a file called *CommandCreateDto.cs* in the *Dtos* folder as so.



**Figure 10-6.** *CommandCreateDTO.cs created*

Into that file add the following code:

```
using System.ComponentModel.DataAnnotations;

namespace CommandAPI.Dtos
{
    public class CommandCreateDto
    {
        [Required]
        [MaxLength(250)]
        public string HowTo { get; set; }

        [Required]
        public string Platform { get; set; }

        [Required]
        public string CommandLine { get; set; }
    }
}
```

This is exactly the same as our internal command model (noting we have included the use of annotations), except that we have *not included* the `Id` property. Make sure you remember to save the file.

## Update the AutoMapper Profile

You'll remember we had to create a profile mapping for our first DTO, which mapped our "source" (a command model) to a target (our `CommandReadDto`). Well we have to do the exact same thing here; we just have to be careful with what our "source" is vs. what our "target" is. So over in the ***CommandsProfile.cs*** file in the ***Profiles*** folder, add the following mapping:

```
public class CommandsProfile : Profile
{
    public CommandsProfile()
    {
        //Source ➤ Target
        CreateMap<Command, CommandReadDto>();
    }
}
```

```

    CreateMap<CommandCreatedDto, Command>();
}
}

```

I won't display the usual "code in context" image for explanation purposes as I feel this is straightforward, but in essence our "source" is the `CommandCreatedDto` (as will be supplied in our POST request body), and the target is our internal `Command` model.

So with

- The new `CommandCreatedDto` created
- An updated `AutoMapper` mapping profile

We can move on to implementing our controller action (our Create endpoint).

## Updating the Controller

So fair warning, although the code for our next action is not particularly large in volume, there are a lot of concepts in this section. Thinking about the best way to present it to you, I'd decided to include all the code in one go (rather than layering it up which I feel would not translate well to the written page and be more confusing than helpful). Don't worry, we go through it all line by line by way of explanation afterward.

So over in our `CommandsController` class, add the following code to create our new controller action:

```

[HttpPost]
public ActionResult <CommandReadDto> CreateCommand
    (CommandCreatedDto commandCreatedDto)
{
    var commandModel = _mapper.Map<Command>(commandCreatedDto);
    _repository.CreateCommand(commandModel);
    _repository.SaveChanges();

    var commandReadDto = _mapper.Map<CommandReadDto>(commandModel);

    return CreatedAtRoute(nameof(GetCommandById),
        new { Id = commandReadDto.Id }, commandReadDto);
}

```

To put those changes in context, see Figure 10-7.

```
[Route("api/{controller}")]
[ApiController]
public class CommandsController : ControllerBase
{
    private readonly ICommandAPIRepo _repository;
    private readonly IMapper _mapper;

    public CommandsController(ICommandAPIRepo repository, IMapper mapper)
    {
        _repository = repository;
        _mapper = mapper;
    }

    [HttpPost]
    public ActionResult<CommandReadDto> CreateCommand(CommandCreateDto commandCreateDto)
    {
        var commandModel = _mapper.Map<Command>(commandCreateDto);
        _repository.CreateCommand(commandModel);
        _repository.SaveChanges();

        var commandReadDto = _mapper.Map<CommandReadDto>(commandModel);

        return CreatedAtRoute(nameof(GetCommandById), new { Id = commandReadDto.Id }, commandReadDto);
    }
}
```

Figure 10-7. CreateCommand Implementation

Let’s go through this:

## 1. HttpPost

We decorate the action with [HttpPost], which I feel is straightforward enough. As mentioned before, this action will respond to the Class-wide route of `api/commands`

with the POST verb, which in combination makes it unique to this Controller.

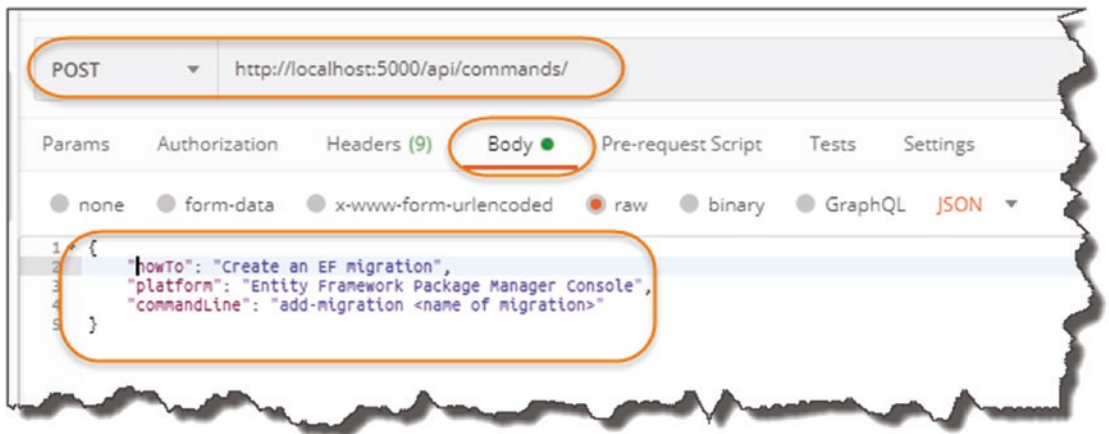
## 2. Return DTO Type

As described in the endpoint attributes, we expect to return the newly created resource as part of our response back to the consumer. In this instance (as with our existing two GET actions), we return a `CommandReadDto`.

### 3. Input DTO Type

Our action expects `CommandCreatedDto` as input, fair enough, but where does that come from? As mentioned, *when* we come to using Postman to test this, we'll place a “`CommandCreatedDto`” in the body of the request, as shown next.

**Important** *Don't* test this Action yet as we still have some more code changes to make before it'll work; I've just shown the Body payload in Figure 10-8 to illustrate this point.



**Figure 10-8.** POST Request in Postman

But that still doesn't answer the question of how does our action “know” to get this data from the Body of the request and pass it in as the `commandCreatedDto` parameter.

The answer to that is *Binding Sources*.

A controller action can derive its inputs from a number of Binding Sources:

- From the Query String
- From the Route (we obtain the `Id` attribute in our URI form here)
- From the Request Body
- From Form fields
- From the Request Header

We can explicitly tell our action where to locate this data or we can fall back on the default behaviors provided to us. For controllers that are decorated with the [ApiController] attribute (as ours is), the default location of *model objects* is the request *Body*.

Therefore, the `commandCreateDto` parameter of our action will be populated with the object we provide in our POST request body.

For a deeper discussion in this, I'd refer you to the [Microsoft docs](#).<sup>3</sup>

## 4. Map Our `CommandCreateDto` to a Command Object

In this step we make use of our AutoMapper profile mapping and, taking our input `commandCreateDto`, map it to a newly created Command object.

## 5. Persist Our Data

In these two steps, we take the newly created Command model from step 4 and pass it to the `CreateCommand` method of our repository.

We then call the `SaveChanges` method on our repository to persist the changes down to the PostgreSQL DB.

## 6. Map Our Created Command Back to a `CommandReadDto`

We have already said that we need to pass back a `CommandReadDto` as part of our endpoint specification, so we do this once again using AutoMapper, to map the newly created Command object back to a `CommandReadDto`. What is of note here is that as we have persisted the Command to the PostgreSQL DB; we now have access to the `Id` attribute (by reference), which is needed going forward – see step 7.

## 7. Created at Route

Then finally we return `CreatedAtRoute` (see definition on Microsoft Docs<sup>4</sup>) where we:

- Specify the “route” where our Created resource resides (more on this below).

---

<sup>3</sup><https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-3.1>

<sup>4</sup><https://docs.microsoft.com/en-us/dotnet/api/system.web.http.apicontroller.createdatroute>

- The Id of the resource (used to generate the route).
- Content value of the body returned.

To summarize, this method will

- Return a 201 – Created Http status code.
- Pass back the created resource in the body response.
- Pass back the URI (or route if you prefer) in the response header.

It basically fulfills the desired behavior of our Create endpoint. If we take a look at this method again, we need to explore one item a little further.



```
[HttpPost]
public ActionResult<CommandReadDto> CreateCommand(CommandCreateDto commandCreateDto)
{
    var commandModel = _mapper.Map<Command>(commandCreateDto);
    _repository.CreateCommand(commandModel);
    _repository.SaveChanges();

    var commandReadDto = _mapper.Map<CommandReadDto>(commandModel);

    return CreatedAtRoute(nameof(GetCommandById), new { Id = commandReadDto.Id }, commandReadDto);
}
```

**Figure 10-9.** *CreatedAtRoute Route Name Parameter*

The first parameter of `CreatedAtRoute` is the `routeName` which in our case is just the existing GET action that returns a single resource based on a supplied Id: `GetCommandById`. In order for the call to `CreatedAtRoute` to work, we need to return to the `GetCommandById` action and “name” it.

So, staying in our controller code, make the necessary highlighted changes to the `GetCommandById` action:

```
[HttpGet("{id}", Name="GetCommandById")]

public ActionResult<CommandReadDto> GetCommandById(int id)
{
    var commandItem = _repository.GetCommandById(id);
    if (commandItem == null)
```

```

{
    return NotFound();
}
return Ok(_mapper.Map<CommandReadDto>(commandItem));
}

```

I've highlighted what's changed in Figure 10-10.



**Figure 10-10.** Naming our *GetCommandById* method

We have explicitly named our action so the call from *CreatedAtRoute* resolves correctly.

Phew! I told you there was a lot to this action – don't worry the remaining actions are not that complex.

All that remains to do is perform some manual tests.

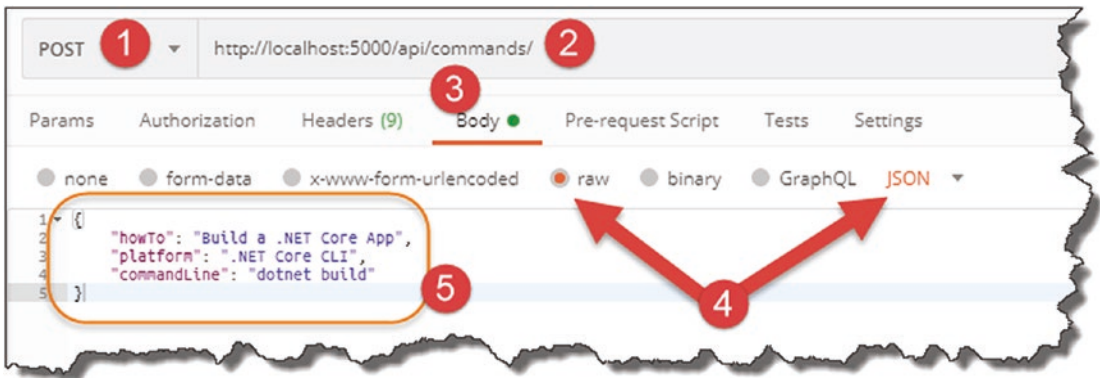
## Manually Testing the Create Endpoint

Before you do anything else, make sure you save all your code (we've made quite a few changes), and perform a `dotnet build` just to check for errors. Assuming all is well, run up your server and move over to Postman.

### Successful Test Case

Here we'll supply the necessary inputs to generate a successful outcome; take a look at my Postman setup in Figure 10-11.

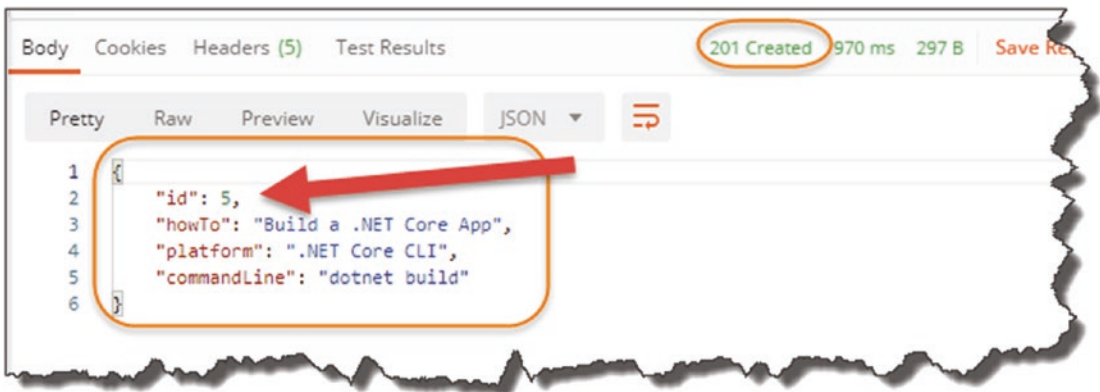




**Figure 10-11.** Test our *CreatCommand Endpoint*

1. Ensure POST is the selected verb.
2. Make sure the route is correct (note there is no Id passed).
3. Select “Body” for the request.
4. Set “Raw” and “JSON” for the request body data type.
5. Supply a valid JSON object that adheres to our *CommandCreatedDto*.

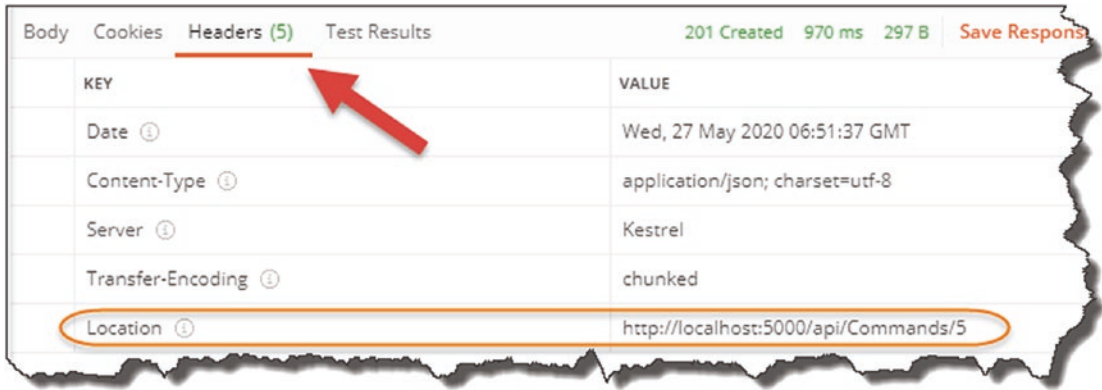
With all that set up, click Send and you should get the following response.



**Figure 10-12.** Successful 201 Result

- 201 Http Created Status Code.
- The newly created resource with Id.

Selecting the Headers Tab, you should get the following.



**Figure 10-13.** URI of our newly created resource is returned in the header

Looks good!

---

**🎓 Learning Opportunity** What else can you do to check that the resource has been created?

---

## Unsuccessful test Case – Badly Formed JSON

Let's issue that exact same request, but this time make some change to the JSON body (e.g., remove all the commas) so that we have badly formed JSON. Click Send, and you should see the following.



**Figure 10-14.** Bad request

We get a Http 400 – Bad request along with some helpful guidance on what’s wrong. We didn’t specifically code this behavior in our controller action – we get this behavior by default as we have decorated our controller with the [ApiController] attribute – see how useful it is!

## Unsuccessful Test Case – Contradict Our Annotations

The last unsuccessful test case I want to run is making sure we violate the data annotations we’ve placed on our CommandCreateDto, specifically the [Required] attribute on one of our properties. To test, reformat the JSON so it’s valid, and remove the Platform property. Click Send again and you should get the following.

We get another 400 Bad Request Http response, with some detail about the validation error.



Figure 10-15. Bad Request with validation detail

**🎓 Learning Opportunity** Test what will happen if you remove the [Required] attribute from the Platform property on our CommandCreatedto, and rerun the same request.

We'll return to testing all our endpoints further in Chapter 11, but for now let's move on to implementing our Update endpoints.

## The Update Endpoint #1 (PUT)

The next endpoint we want to implement is the first "Update" action which gives us the ability to *fully update* a single resource in our DB using a PUT request. A quick reminder of our high-level definition is shown in the table.

Verb	URI	Operation	Description
PUT	/api/commands/{Id}	Update (Full)	Update all of a single resource (by Id)

As before, I've introduced some other attributes that will help us understand, build, and test our endpoint more effectively.

Attribute	Description
Inputs (x2)	<p>The Id of the resource to be updated. This will be present in the URI of our PUT request.</p> <p>The full “command” object to be updated.</p> <p>This will be added to the request body of our PUT request; an example is shown here:</p> <pre>{   "howTo": "Example how to",   "platform": "Example platform",   "commandLine": "Example command line" }</pre>
Process	Will attempt to fully update an existing command object in our DB
Success Outputs	<ul style="list-style-type: none"> <li>• HTTP 204 No Content response code</li> </ul>
Failure Outputs	<ul style="list-style-type: none"> <li>• HTTP 400 Bad Request</li> <li>• HTTP 404 Not Found</li> <li>• HTTP 405 Not Allowed</li> </ul>
Safe	No – Endpoint can alter our resources
Idempotent	Yes – Repeating the same operation will not incur a different result

Again, quite straight forward, but I'd call out the following points of note.

## Input Object

This is identical to our Create endpoint – does this mean we can reuse our `CommandCreatedDto`? Theoretically we could, but in the interests of true decoupling, we're going to create a separate `CommandUpdatedDto`, just to future-proof our solution should these objects diverge in the future.

The other point of note is that this object does not contain the `Id` attribute. We do require it for this operation (otherwise, how would we know which object to update), but in this case, we get this value from the URI (which is another stipulation of REST), so we don't need to double up on it here.

## Success Outputs

Very simple in this case, we just supply a 204 No Content http result.

## Idempotent

This method is idempotent as you can repeat it multiple times and the result will be the same.



**Les' Personal Anecdote** The PUT request has fallen out of favor when compared to the PATCH request these days, mainly due to the fact you have to supply *all* the object attributes to be updated, even the ones that are not changing!

This is really inefficient for large objects. Say you have an object with 20 properties and you only need to change 1, you still have to supply all 20 to the PUT request, ensuring that you provide the correct (same) value for each of the 19 that are *not changing*.

If you inadvertently provide the wrong value or omit it altogether for 1 of the 19, you could end up in real strife! *Cough, cough; I have never done that.*

Not only is it problematic/inefficient in this respect; from a network perspective it's not optimal; you're essentially sending potentially large amounts of redundant data over the wire (or through the air).

The only reason I've included it here is for completeness and because I'm a nice guy.

---

## Updating the Repository

Again, starting at the repository level, let's take a look at the update method signature in our ICommandRepoAPI interface.

```

public interface ICommandAPIRepo
{
    bool SaveChanges();

    IEnumerable<Command> GetAllCommands();
    Command GetCommandById(int id);
    void CreateCommand(Command cmd);
    void UpdateCommand(Command cmd);
    void DeleteCommand(Command cmd);
}

```

**Figure 10-16.** *UpdateCommand Interface Method*

We accept a `Command` object (update the database if required), and don't expect to pass anything back. You'll notice my choice of words: "update the database *if* required"; the reason I've chosen these will become clearer below.

What I'd like to remind you about our repository interface is that it is *technology-agnostic* – meaning that it is an interface specification we could use against different persistence providers, for example, Entity Framework Core, nHibernate, Dapper, etc. We just so happen to be using it with Entity Framework Core, and we therefore have to provide a specific, concrete implementation for that ORM. And this is where it gets weird.

Moving over to our `SqlCommandAPIRepo` implementation class, update the `UpdateCommand` method as follows:

```

public void UpdateCommand(Command cmd)
{
    //We don't need to do anything here
}

```

Yes, that's right – it contains "no implementation" – just a smart-arsed comment from me. I've not gone mad, let me explain.

## Remember How Our DB Context Works

Cast your mind back to the lengthy explanation of how EF Core persists data at the start of this chapter; not only was that just generally useful information to know, but it was done in expectation of this explanation. This is the payoff.

We will actually perform the *update* of our existing Command object *in our Controller action*, so we don't need to put any code in our repository implementation. It will probably become clearer when we come to code it up, but let me explain further how this will work:

1. The Update action will be called (with the CommandUpdateDto object in the request Body).
2. **In our controller:** Based on the Id in the request URI, we'll search the DB Context to see if we have an existing Command object with that Id.
3. **If it doesn't exist:** We return a 404 Not Found Result and return.
4. **If it does exist:** We'll "Map" the CommandUpdateDto received in the request body to the Command object we just received from our DB Context in Step 2. **It is at this point the Command object is updated in the DB Context.** We therefore don't need any implementation code in our SqlCommandAPIRepo repository.
5. We call the SaveChanges method on our repository, and the changes will be persisted to the database.

You may then ask the very valid question: If we *don't need* implementation code here, why not remove it altogether from our repository interface? The answer to that is to once again remind you that the repository interface is technology agnostic, so while we don't require an implementation *in this instance*, if we choose to switch our persistence provider, they *may require* a coded implementation.

So logically speaking it makes sense to specify an Update method signature in our interface, even if in this instance we *don't* need to implement it.

Anyway, with that we're done with the repository "implementation" and can move on to the DTO.

## CommandUpdateDto

As recently described, we're going to expect a CommandUpdateDto in our request body and map it over to the Command retrieved from our DB Context. To enable this, create a file in the *Dtos* folder called **CommandUpdateDto.cs**, and add the following code:



```

using System.ComponentModel.DataAnnotations;

namespace CommandAPI.Dtos
{
    public class CommandUpdatedDto
    {
        [Required]
        [MaxLength(250)]
        public string HowTo {get; set;}

        [Required]
        public string Platform {get; set;}

        [Required]
        public string CommandLine {get; set;}
    }
}

```

This is exactly the same as our `CommandCreatedDto`, but we'll maintain a separate instance for future-proofing purposes. Save the file, and move on to updating out AutoMapper profile mappings.

## Update the AutoMapper Profile

We need to add a mapping with the `CommandUpdatedDto` as the mapping source and the `Command` model as the target, so update the `CommandsProfile` class with the following mapping entry:

```

using AutoMapper;
using CommandAPI.Dtos;
using CommandAPI.Models;

namespace CommandAPI.Profiles
{
    public class CommandsProfile : Profile
    {
        public CommandsProfile()
        {

```

```

    //Source ► Target
    CreateMap<Command, CommandReadDto>();
    CreateMap<CommandCreateDto, Command>();
    CreateMap<CommandUpdateDto, Command>();
}
}
}

```

I don't believe at this stage we require any further explanation on this!

## Updating the Controller

Moving back to our controller, we need to add a new controller action to host our new endpoint, so in the `CommandsController` class, add the following code to achieve this:

```

[HttpPut("{id}")]
public ActionResult UpdateCommand(int id, CommandUpdateDto
commandUpdateDto)
{
    var commandModelFromRepo = _repository.GetCommandById(id);
    if (commandModelFromRepo == null)
    {
        return NotFound();
    }
    _mapper.Map(commandUpdateDto, commandModelFromRepo);
    _repository.UpdateCommand(commandModelFromRepo);

    _repository.SaveChanges();

    return NoContent();
}

```

Let's walk through the code.

```

[HttpPut("{id}")] 1
public ActionResult UpdateCommand(int id, CommandUpdateDto commandUpdateDto) 2
{
    var commandModelFromRepo = _repository.GetCommandById(id); 3
    if (commandModelFromRepo == null) 4
    {
        return NotFound();
    }
    _mapper.Map(commandUpdateDto, commandModelFromRepo); 5
    _repository.UpdateCommand(commandModelFromRepo); 6
    _repository.SaveChanges(); 7
    return NoContent(); 8
}

```

**Figure 10-17.** *UpdateCommand Controller Action Implementation*

## 1. HttpPut

We decorate the `UpdateCommand` method with the `[HttpPut]` attribute (no real controversy there), but we also expect an `Id` as part of the route; this means this endpoint will respond to the class-wide route plus the `Id`, so

```
api/commands/{id}
```

## 2 Inputs

The `UpdateCommand` method expects two parameters:

1. **id**: this is the `id` passed in from the route, which equates to the unique `id` of the resource we want to attempt update.
2. **commandUpdateDto**: this is the object passed in in the request body.

## 3. Attempt Command Resource Retrieval

We make use of the `id` passed in from the route and, using our existing repository method, `GetCommandById`, attempt to retrieve it. Irrespective of the result, we place the result of this operation in `commandModelFromRepo`.

## 4. Return 404 Not Found

Not much more to say here; if the “object” we attempted to retrieve from the repository is null, then we just return with a 404 Not Found Http response.

## 5. Update our Command

This is where the actual update occurs! We use a slightly different form of the Map method on our `_mapper` instance to map the DTO to our Command. By reference the Command object is updated in the DB Context. Again, this is not yet reflected down to our PostgreSQL DB.

## 6. Update Nothing

This is a controversial one! You probably think I’ve definitely gone mad now.

This line does nothing,(at the moment). But remember, we may at any point in time swap out our Entity Framework Core implementation for another provider that *may need* a call to the `UpdateCommand` method in our repository. This is essentially the same reason for keeping the definition in the repository interface in the first place.

By keeping this call here (even though it’s currently redundant), if we do swap out our repository implementation (that requires a call to `UpdateCommand`), we won’t need to change our Controller code, which is kind of the point of doing all this!



**Personal Perspective** This is probably the single most contentious part of the build and the one that I do get questioned on quite a lot. If you feel more comfortable *not including* step 6, that is your choice; the code will still work in *this* instance.

I personally find that the embedding of knowledge and understanding comes from your own practical approaches and experiments - so go with what you feel works best for you. However, just remember this section if you do ever have to rework your code in the future.

---

## 7. Save Changes

An obvious one, don't think I need to go on here.

## 8. Return 204 No Content

Nice and simple, we return a 204 No Content. You'll also notice that the `UpdateCommand` method does not have a return type.

OK, save your work, start your engines, and let's move on to performing a few manual tests.

## Manually Testing the Update (PUT) EndPoint

### Successful Test Case

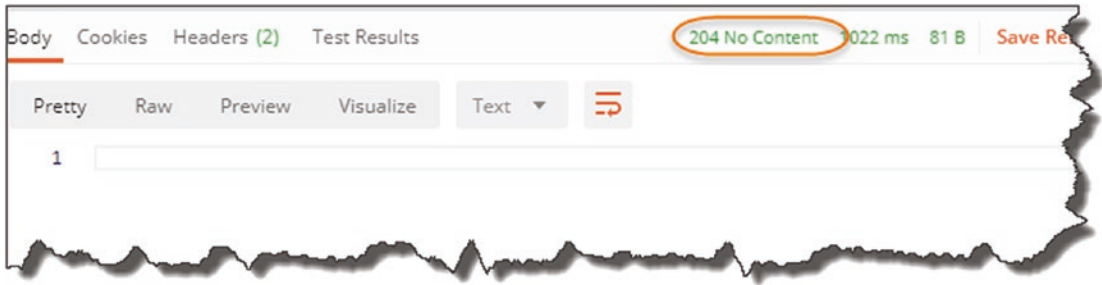
Here we supply the necessary inputs to generate a successful outcome as reflected in the Postman setup in Figure 10-18.



**Figure 10-18.** Testing the Put Action Result

1. Ensure PUT is selected.
2. The route to the resource you want to update needs to be valid.
3. I've just updated `howTo` and `commandLine`, but I've also had to supply the `Platform` even though this is not changing.

Click Send and you should get a similar result to the one in Figure 10-19.



**Figure 10-19.** Success – 204 No Content Returned

Fairly basic, just a 204 No Content Http Response. I’ll leave it to you to check if this actually did update the resource in the DB.

## Unsuccessful Test Case - Contradict Our Annotations

In this test, case we’ll attempt to update an existing resource and not supply a [Required] attribute; see my Postman setup in Figure 10-20.



**Figure 10-20.** Force a validation error

Click send and you’ll see something like the following.



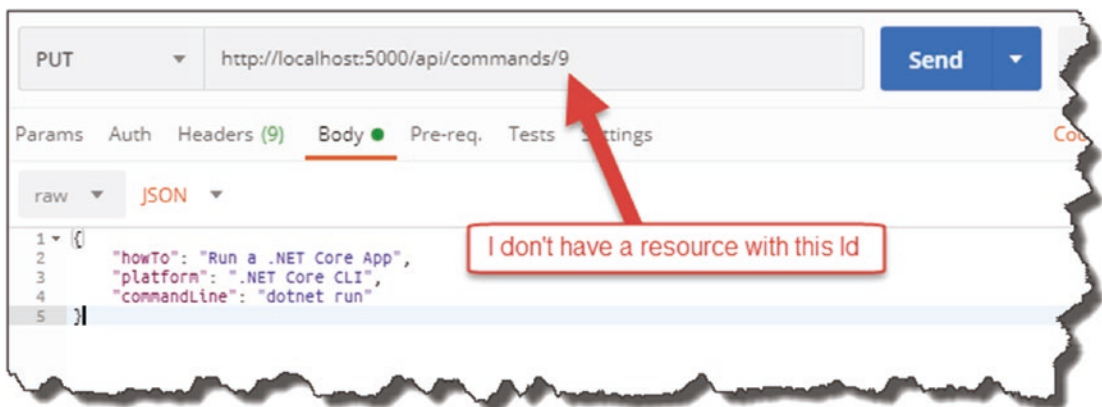
**Figure 10-21.** Validation Error Returned

This demonstrates the usefulness of data annotations. But I hear you cry: I thought you said we needed to supply all attributes in a PUT request anyway?

Great question, the answer to that is yes you do if you want them to be updated (or remain the same). Taking the [Required] annotation out the equation, if we *could* supply a null value for Platform and don't supply it in our PUT request – that will work. What will happen though is that the existing value for Platform will not be persisted as is; it will revert to the default value, and if it doesn't have one, then null!

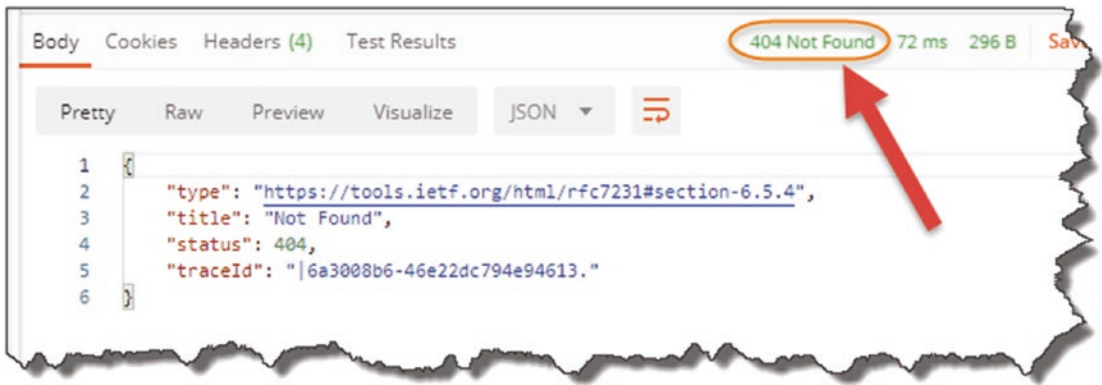
## Unsuccessful Test Case – Invalid Resource ID

Here, we just supply an Id for a resource that does not exist.



**Figure 10-22.** Test with nonexistent resource

Click Send, and you'll get the following.



**Figure 10-23.** 404 Not Found Returned

So, looks like everything is working as per our requirements; let's move on to the arguably more interesting PATCH update endpoint.

## The Update Endpoint #2 (PATCH)

The next endpoint we want to implement is the second “Update” endpoint (using the PATCH verb), which gives us the ability to perform partial updates on a resource. This addresses many of the inefficiencies of the PUT endpoint that we noted in the last section. A quick reminder of our high-level definition is shown in the table.

Verb	URI	Operation	Description
PATCH	/api/commands/{Id}	Update (partial)	Update part of a single resource (by Id)

To further flesh out our definition, I've included some additional attributes here.



Attribute	Description
Inputs (x2)	<p>The Id of the resource to be updated. This will be present in the URI of our PATCH request.</p> <p>The change-set or “patch document” to be applied to the resource This will be added to the request body of our PATCH request; an example is shown here:</p> <pre>[   {     "op": "replace",     "path": "/howto",     "value": "Some new value"   },   {     "op": "test",     "path" : "commandline",     "value" : "dotnet new"   } ]</pre>
Process	<p>Will attempt to perform the updates as specified in the patch document Note: If there is more than one update, all those updates need to be successful. If one fails, then they all fail.</p>
Success Outputs	<ul style="list-style-type: none"> <li>• HTTP 204 Not Content HTTP Status</li> </ul>
Failure Outputs	<ul style="list-style-type: none"> <li>• HTTP 400 Bad Request</li> <li>• HTTP 404 Not Found</li> <li>• HTTP 405 Not Allowed</li> </ul>
Safe	No – Endpoint can alter our resources
Idempotent	No – Repeating the same operation <i>may</i> incur a different result

There are a few new concepts here so let’s go through them.

## Input Object

Instead of supplying a representation of the resource we want to update, we supply a series of changes that we want to perform against that resource. We call this a Patch Document or Change Set.

Our Patch Document can perform the following *operations*:

- **Add:** Adds a new property to our object (this requires “dynamic” objects which we won’t be using).
- **Remove:** Again, requires dynamic objects and allows us to remove a property from our resource.
- **Replace:** Allows us to change an existing property (this is the one we’ll be using).
- **Copy:** As the name suggests, this allows us to copy a resource property value to another.
- **Move:** The same as combining Copy and Remove operations.
- **Test:** Allows us to test the value of a given resource property.

In addition to specifying what operation we want to perform against a property, we need to supply

- A path to that resource property
- The new value we want to assign

Refer to the example in [Figure 10-24](#) for clarity.



```
[
  {
    "op": "replace",
    "path": "/howto",
    "value": "Some new value"
  },
  {
    "op": "test",
    "path": "/commandline",
    "value": "dotnet new"
  }
]
```

**Figure 10-24.** Example of a simple patch document

The Patch Document is attempting to perform two operations:

1. Replace the value of the `howto` property with the value “Some new value.”
2. Test to see if the `commandline` property contains the value “dotnet new”

For this Patch Document to be successful, both of these operations will need to succeed.

For more information on the PATCH specification, refer to the RFC 6902 standard.<sup>5</sup>

## Idempotent

The PATCH operation is not idempotent as running the same request multiple times may yield different results.

## Updating the Repository

There is no requirement to perform further update on our repository.

<sup>5</sup><https://tools.ietf.org/html/rfc6902>

## CommandUpdateDto

While there isn't a requirement to make any changes to our `CommandUpdateDto`, we do need to add one further mapping to our AutoMapper Profiles. I'm not going to explain why here; we'll just make the necessary change and circle back to it when we come to implementing the controller action as it will be easier to explain at that point.

So, open the `CommandsProfile` class in the *Profiles* folder, and add the following (final) mapping:

```
public class CommandsProfile : Profile
{
    public CommandsProfile()
    {
        //Source ► Target
        CreateMap<Command, CommandReadDto>();
        CreateMap<CommandCreateDto, Command>();
        CreateMap<CommandUpdateDto, Command>();
        CreateMap<Command, CommandUpdateDto>();
    }
}
```

You should be comfortable of *what* is happening here; we'll cover off the *why* below.

## Install Dependencies for PATCH

Unlike the other endpoints we've covered, PATCH requests require some further package dependencies to be installed in order for PATCH requests to work correctly. So at a command prompt (and making sure you are "in" the *CommandAPI* project folder), issue the following commands:

```
dotnet add package Microsoft.AspNetCore.JsonPatch
dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson
```

The first adds support for the PATCH request; the second is required to correctly work with Patch Documents in our controller.

To make sure the dependencies were installed, check the *.csproj* file for our project.

```

<ItemGroup>
  <PackageReference Include="AutoMapper.Extensions.Microsoft.DependencyInjection" Version="7.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.JsonPatch" Version="3.1.4" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson" Version="3.1.4" />
  <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.4" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.4">
    <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    <PrivateAssets>all</PrivateAssets>
  </PackageReference>
  <PackageReference Include="Npgsql.EntityFrameworkCore.PostgreSQL" Version="3.1.3" />
</ItemGroup>

</Project>

```

**Figure 10-25.** Packages required to support Patch

## Updating the Startup Class

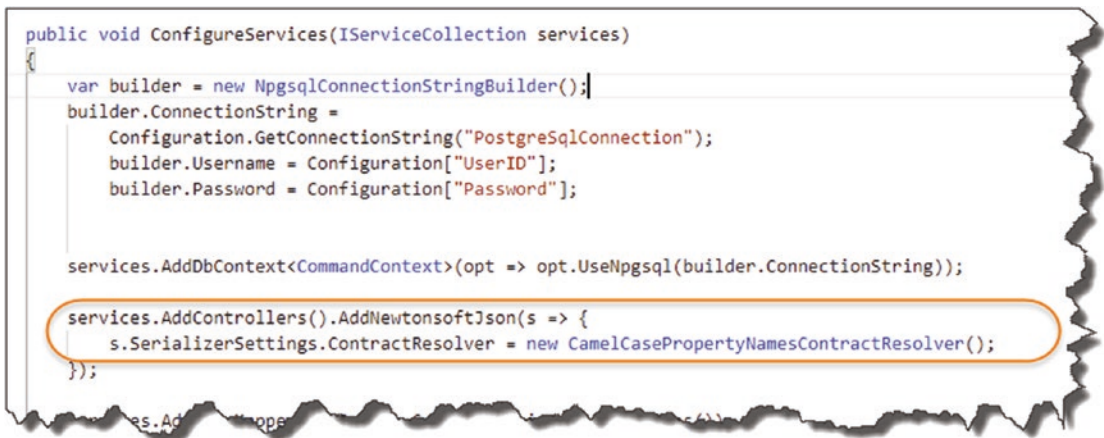
To make use of the second package we added earlier, we need to make a minor addition to our Startup class as shown below (make sure to include the using statement too):

```

.
.
.
using Newtonsoft.Json.Serialization;
.
.
services.AddControllers().AddNewtonsoftJson(s =>
{
    s.SerializerSettings.ContractResolver = new
    CamelCasePropertyNamesContractResolver();
});
.
.

```

To put those changes in context (for brevity I've *not* shown the new using statement below), see [Figure 10-26](#).



**Figure 10-26.** *Serializer settings on our controllers*

As you can see, we require the use of NewtonsoftJson package within our controller; this allows for the correct parsing of our Patch document.

With that set up, we’re now ready to move over to our controller.

## Updating the Controller

As we did with our create action, I’m just going to get you to enter the entire code for this action, and we’ll then step through the code line by line by way of explanation.

using Microsoft.AspNetCore.JsonPatch;

```

[HttpPatch("{id}")]
public ActionResult PartialCommandUpdate(int id,
    JsonPatchDocument<CommandUpdateDto> patchDoc)
{
    var commandModelFromRepo = _repository.GetCommandById(id);
    if(commandModelFromRepo == null)
    {
        return NotFound();
    }

    var commandToPatch = _mapper.Map<CommandUpdateDto>(commandModelFromRepo);
    patchDoc.ApplyTo(commandToPatch, ModelState);
}

```

```

if(!TryValidateModel(commandToPatch))
{
    return ValidationProblem(ModelState);
}

_mapper.Map(commandToPatch, commandModelFromRepo);

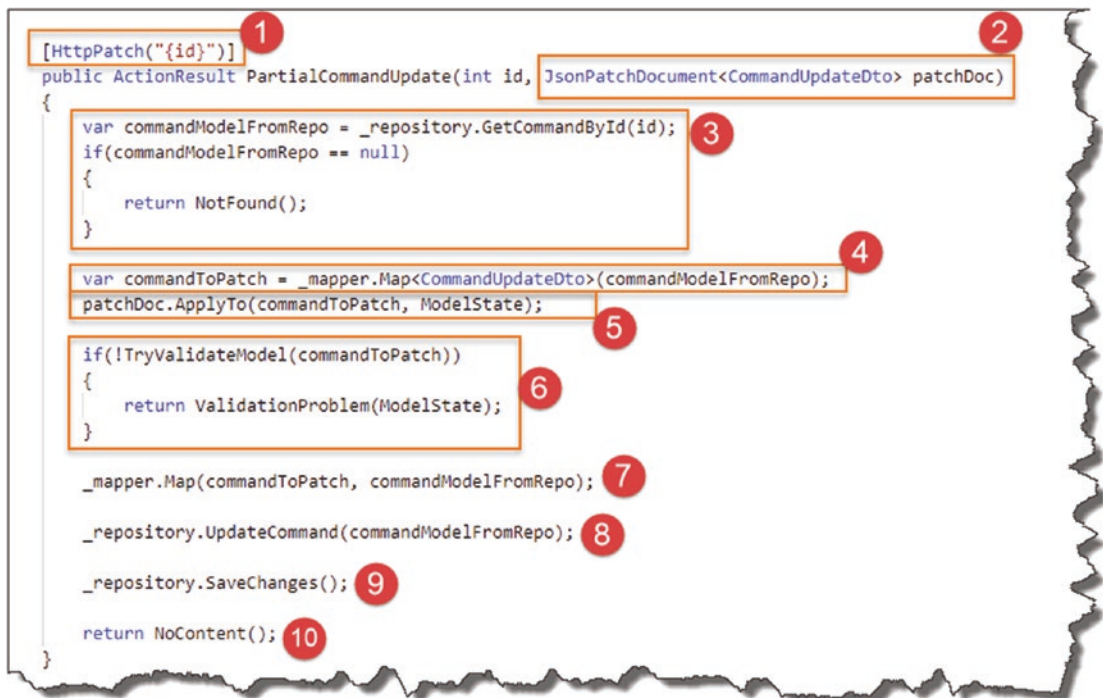
_repository.UpdateCommand(commandModelFromRepo);

_repository.SaveChanges();

return NoContent();
}

```

Quite a lot to take in there, so let's go through the code before we come on to testing it.



**Figure 10-27.** Patch Controller Action

## 1. HttpPatch

Shouldn't be any surprises here; we need to decorate with `[HttpPatch]` and specify that we expect a resource `Id` in the route.

## 2. JsonPatchDocument

We expect a `JsonPatchDocument` in the request body that “applies” to a `CommandUpdateDto`. This feeds into the validations that are performed below. (We need to specify the object type the `JsonPatchDocument` “applies to” in order to deduce if it's valid.)

## 3. Attempt Command Resource Retrieval

This is exactly the same code that we had in our `PUT` action; it doesn't require further qualification.

## 4. Create Placeholder CommandUpdateDto

We need to create a `CommandUpdateDto` object based on the `Command` object we've just successfully retrieved. Why? Well as mentioned in Step 1, the `JsonPatchDocument` has to “apply to” a specific object type; in this case we've specified a `CommandUpdateDto`, so we need to create one for use.

---

**i Circle Back** This is why we needed to add our fourth and final `AutoMapper` Profile mapping.

---

## 5. Apply the Patch Document

Here we apply the Patch Document received in our request body to the newly created `CommandUpdateDto: commandToPatch`.

---

**i Circle Back** Had we not included and used the `Microsoft.AspNetCore.Mvc.NewtonsoftJson` package, we would not be able to correctly perform this operation.

---



## 6. Validate Model Changes

Following the application of the desired changes in our Patch Document, we then attempt to see if the model validation (via our Data Annotations) is valid. For example, if the Patch Document requested a Replace operation on the `HowTo` property that was greater than 250 characters, it would be picked up here.

## 7. Map Updated Dto to Command and Return

Our `CommandUpdateDto` (`commandToPatch`) has been successfully updated at this point. We now use `AutoMapper` to map it back to our `Command` object in our `DB Context`.

---

**Note** From this point onward, the code is identical to our previous `PUT` action, so to save on duplicating that explanation, please just refer to the recent explanation.

---

And with that – we’re done with coding this controller action; make sure you save everything, and we’ll move on to some manual tests.

## Manually Testing the Update (PATCH) EndPoint

### Successful Test Case

The most important thing to get right here is the Patch Document; to begin with I’m keeping it simple and updating the `HowTo` property of an existing resource as shown in my Postman setup.

1. `PATCH` Verb is selected.



**Figure 10-28.** Test our Patch ActionResult

2. URI to an existing resource.
3. Our Patch Document with a single operation.

---

**⚠ Warning!** You’ll note that even though our Patch Document has only one operation, we still need to enclose it in square parenthesis [].

Remembering from our discussion on JSON, square brackets [] denote an array.

---

**🎓 Learning Opportunity** I’ll refrain from detailing our failing or unsuccessful test cases here and leave it to you to explore these – have fun! (They’re not much different from the ones we ran for PUT.)

---

## The Delete Endpoint (DELETE)

The final endpoint we want to implement is the “Delete” endpoint, which gives us the ability to remove resources from our DB. A quick reminder of our high-level definition is shown here.

Verb	URI	Operation	Description
DELETE	/api/commands/{Id}	Delete	Delete a single resource (by Id)

Further details of how this Endpoint should operate are listed here.

Attribute	Description
Inputs (x1)	The Id of the resource to be deleted. This will be present in the URI of our DELETE request
Process	Will attempt to delete an existing command object to our DB
Success Outputs	<ul style="list-style-type: none"> <li>• HTTP 204 No Content HTTP result</li> </ul>
Failure Outputs	<ul style="list-style-type: none"> <li>• HTTP 404 Not Found HTTP result</li> </ul>
Safe	No – Endpoint can alter our resources
Idempotent	Yes – Repeating the same operation will incur the same result

There's not much to call out here, so let's move on to what we need to code.

## Updating the Repository

Refer back to our repository interface as detailed in Figure 10-29.

```
public interface ICommandAPIRepo
{
    bool SaveChanges();

    IEnumerable<Command> GetAllCommands();
    Command GetCommandById(int id);
    void CreateCommand(Command cmd);
    void UpdateCommand(Command cmd);
    void DeleteCommand(Command cmd);
}
```

**Figure 10-29.** Delete Interface Method

We expect the Command object to be deleted and do not expect anything to be passed back.

Moving over to our implementation class `SqlCommandAPIRepo`, update the `DeleteCommand` method as follows:

```
public void DeleteCommand(Command cmd)
{
    if(cmd == null)
    {
        throw new ArgumentNullException(nameof(cmd));
    }
    _context.CommandItems.Remove(cmd);
}
```

The code is pretty straightforward, so I don't feel further explanation is needed. Just remember that calling this method only marks the Command for deletion in the DB Context; we still need to call `SaveChanges` to effect a change in the database.

## CommandDeleteDto

There is no requirement for a `CommandDeleteDto`.


## Updating the Controller

Thankfully the code for our delete action is very simple (compared to the last three Endpoints we've done); it's shown here:

```
[HttpDelete("{id}")]
public ActionResult DeleteCommand(int id)
{
    var commandModelFromRepo = _repository.GetCommandById(id);
    if(commandModelFromRepo == null)
    {
        return NotFound();
    }
}
```

```
_repository.DeleteCommand(commandModelFromRepo);  
_repository.SaveChanges();  
  
return NoContent();  
}
```

---


 **Learning Opportunity** I feel at this stage there is little benefit in me adding extra narrative on both what this code is doing and how to manually test it!

We've covered significantly more complex use-cases, so I think you can round off the manual testing of this Endpoint yourself.

---

## Wrap Up

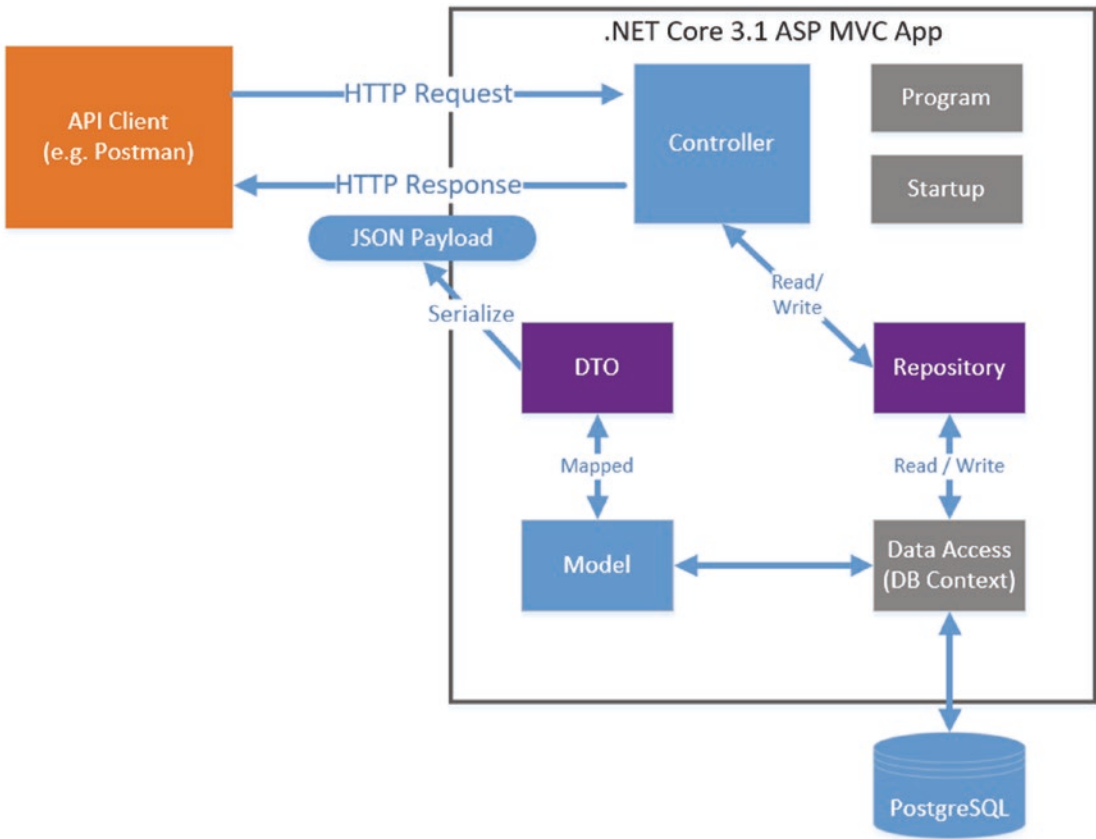
---

 **Celebration Checkpoint** We have fully implemented our API now!  
Congratulations!

---

We covered a lot of code in this chapter; I did consider splitting it across multiple chapters but thought that may have interrupted the flow of what we were tackling.

Once again, let us return to our application architecture as shown in Figure [10-30](#).



**Figure 10-30.** *Architecture Checkpoint*

You can see that we have now implemented everything – great job! Don’t celebrate too quickly though as we are not done just yet. In the chapters that follow, I take you through

- Automating Testing for our API
- Deploying the API to Production using a CI/CD Pipeline
- Securing our API from unwanted guests

Before you move on, remember to save everything and commit locally to GitHub.

## CHAPTER 11

# Unit Testing Our API

## Chapter Summary

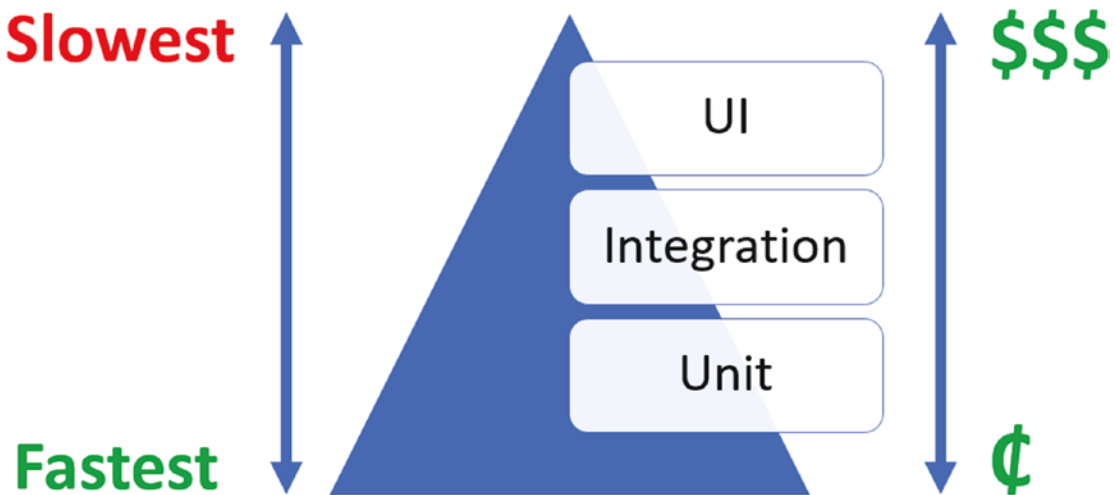
In this chapter we'll introduce you to Unit Testing, what it is, and why you'd use it. We'll then create unit tests to test the core functionality of our API Controller, providing us with an automated regression suite (don't worry if you don't know what that means!).

## When Done, You Will

- Understand what Unit Testing is and why you should use it.
- Understand the power of the Repository Interface once again!
- Understand how to use a mocking or isolation framework in unit testing.
- Write Unit Tests using *xUnit* to test our API functionality.

## What Is Unit Testing

Probably the best way to describe what Unit Testing is to put it in context of the other general types of “testing” you will encounter, so I refer you to the “Testing Pyramid” in Figure [11-1](#).



*Figure 11-1. Testing Pyramid*

Unit tests are

- **Abundant:** There should be more of them than other types of test.
- **Small:** They should test one thing only, that is, 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 life cycle.
- 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 (which 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.

---

**i** When naming your unit test methods, they should follow a construct similar to `<method name>_<expected result>_<condition>`

For example:

`GetCommandItem_Returns2000K_WhenSuppliedIDIsValid`

Note: There are variants on the 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 the Unit Testing Best Practices<sup>1</sup> guide by Microsoft; it's well worth a read, but again we cover more than enough here to get you going. So, the characteristics of a good unit test are

- **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, for example, 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.

---

<sup>1</sup><https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices>

- **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 earlier) should test only one 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 \$1,00,000 question is what should we actually test? The characteristics detailed earlier 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*: for example, you would not write unit tests to check basic arithmetic operations— that would be overkill and not terribly useful. Taking this further, unit testing code you cannot change (i.e., code you did not write) may be somewhat pointless: discuss.

---

## Unit Testing Frameworks

I asked a question at the start of the book about what [xUnit](#) is. Well xUnit is simply a unit testing framework; it's open source and was used heavily 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.

Unit Test Project	mstest	[C#], F#, VB	Test/MSTest
NUnit 3 Test Project	nunit	[C#], F#, VB	Test/NUnit
NUnit 3 Test Item	nunit-test	[C#], F#, VB	Test/NUnit
xUnit Test Project	xunit	[C#], F#, VB	Test/xUnit
Razor Component	razorcomponent	[C#]	Web/ASP.NET

**Figure 11-2.** Unit Testing .NET Core Project templates

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, and 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 “setup” of your test. For example, 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

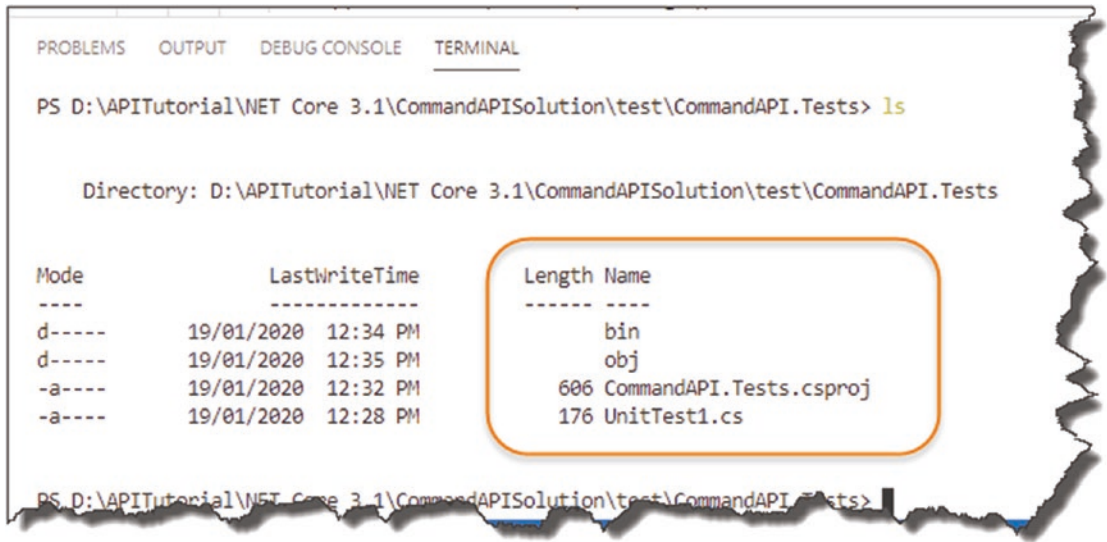
This is where you “check” the *actual result* against the *expected result*. How that assertion goes will depend on whether your test passes or fails.

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 *one assertion per test*. If you assert multiple conditions, the unit tests become 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 **CommandAPI.Tests** folder, listing the contents of that folder you should see.



**Figure 11-3.** Anatomy of a xUnit Project

We have

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

You should be familiar with the first three of these, as they are the same artifacts we had in our API project. With regard to 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 fourth and final artifact here is a default class set up for us when we created the project; open it, and take a look.

```
using System;
using Xunit; 1

namespace CommandAPI.Tests
{
    public class UnitTest1
    {
        [Fact] 2
        public void Test1()
        {
        }
    }
}
```

**Figure 11-4.** Simple xUnit Test Case

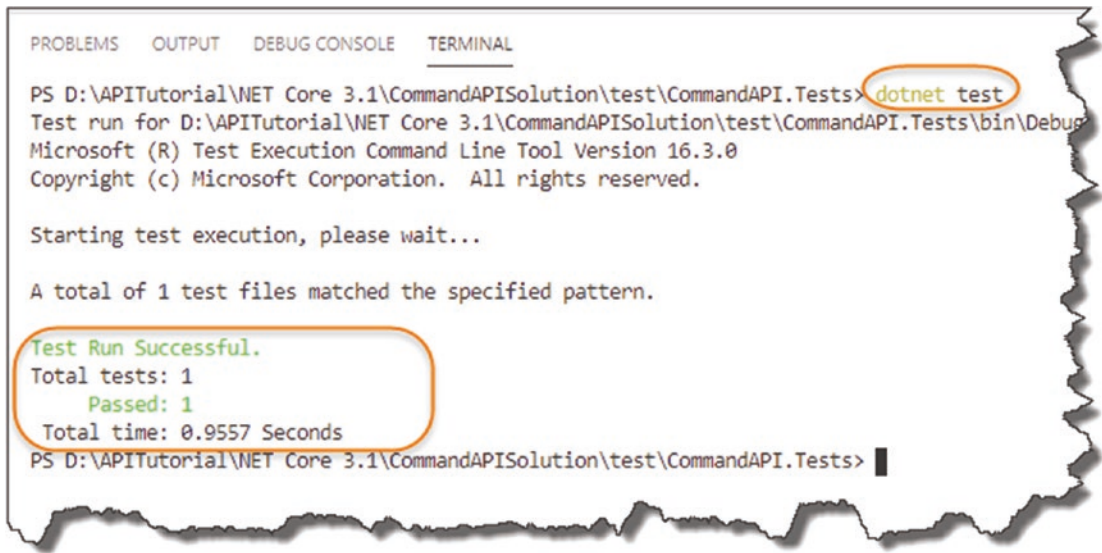
This is just a standard class definition, with only two points of note:

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.



**Figure 11-5.** Running Tests in xUnit

OK, we know our testing setup is good to go, so let's start writing some tests.



**Les' Personal Anecdote** When running through the code again myself (yes I actually followed the book all the way through to make sure it made sense!), I got a warning at this stage complaining that Microsoft.EntityFrameworkCore.Relational was at a different version in the xUnit project compared to the main API project.

Note that this package *was not* explicitly listed in the package references in our xUnit projects .csproj file.

To rectify this I installed the Microsoft.EntityFrameworkCore.Relational package in my xUnit project:

```
dotnet add package Microsoft.EntityFrameworkCore.  
Relational --version 3.1.4
```

noting that I did specify a version this time to ensure both packages across both projects were in alignment. If you encounter this same behavior, take note of the version that gets complained about, and act accordingly.

Even though I believe this warning was benign, I don't like warnings lingering in the background.

---

## 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.

```
using System.ComponentModel.DataAnnotations;

namespace CommandAPI.Models
{
    public class Command
    {
        [Key]
        [Required]
        public int Id {get; set;}

        [Required]
        [MaxLength(250)]
        public string HowTo {get; set;}

        [Required]
        public string Platform {get; set;}

        [Required]
        public string CommandLine {get; set;}
    }
}
```

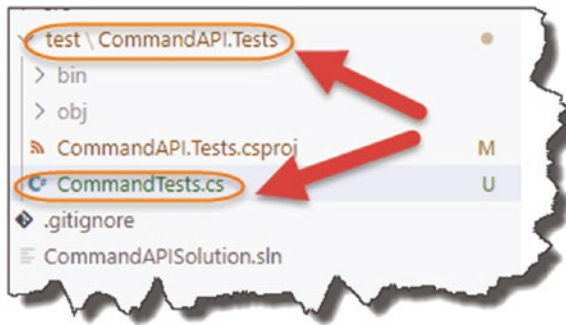
**Figure 11-6.** Revisiting the model

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 the root of our **CommandAPI.Tests** Project.



**Figure 11-7.** Tests for our Model

Add the following code to this class:

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

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

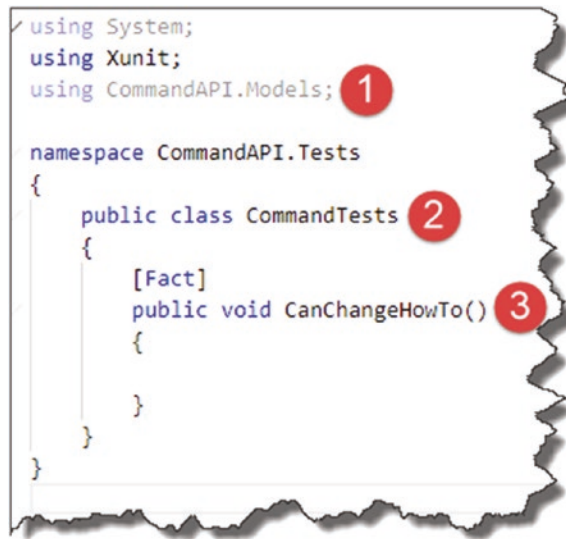


**i** This is such a trivial test (we're not even testing a method); we can't really use the unit test naming convention mentioned earlier:

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

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

The following sections are of note.



```
using System;
using Xunit;
using CommandAPI.Models; 1

namespace CommandAPI.Tests
{
    public class CommandTests 2
    {
        [Fact]
        public void CanChangeHowTo() 3
        {
        }
    }
}
```

**Figure 11-8.** *Our First Model test*

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 us 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 here.

```

using System;
using Xunit;
using CommandAPI.Models;

namespace CommandAPI.Tests
{
    public class CommandTests
    {
        [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);
        }
    }
}

```

**Figure 11-9.** *Arrange, Act, and Assert*

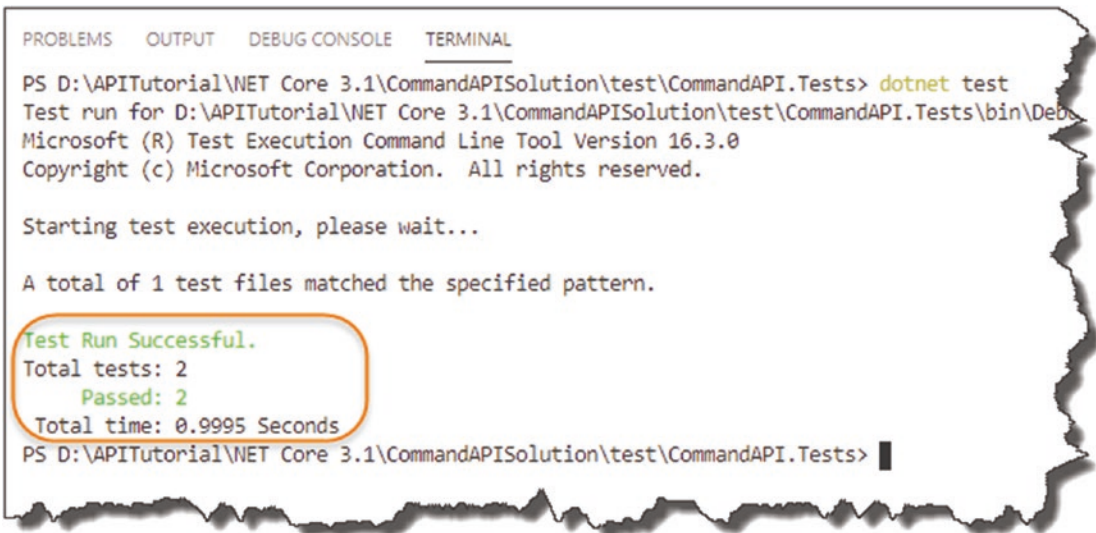
1. **Arrange:** Create a `testCommand` and populate with initial values.
2. **Act:** Perform the action we want to test, that is, change the value of `HowTo`.
3. **Assert:** Check that the value of `HowTo` matches what we expect.

Steps 1 and 2 are straightforward, so it's really step 3 and the use of the `xUnit Assert` class to perform the "Equal" operation that are 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 the following.



**Figure 11-10.** We have two passing tests?

It says two 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 second test is being picked up. Before we continue, let's **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, for example.

```

[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);
}

```

**Figure 11-11.** Forcing test Failure

Save the file, and rerun your tests; you'll get a failure response with some verbose messaging.

```

Starting test execution, please wait...

A total of 1 test files matched the specified pattern.
[xUnit.net 00:00:00.58] CommandAPI.Tests.CommandTests.CanChangeHowTo [FAIL]
X CommandAPI.Tests.CommandTests.CanChangeHowTo [4ms]
  Error Message:
  Assert.Equal() Failure
    (pos 0)
Expected: Test will fail
Actual:   Execute Unit Tests
    (pos 0)
  Stack Trace:
    at CommandAPI.Tests.CommandTests.CanChangeHowTo() in D:\APITutorial\NET Core 3...


Test Run Failed.
Total tests: 2
  Passed: 1
  Failed: 1
Total time: 1.0458 Seconds
PS D:\APITutorial\NET Core 3-4\CommandAPI> dotnet test CommandAPI.Tests>

```

**Figure 11-12.** As expected, failed test

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 two 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 two 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 three test methods in your `CommandTests` class, with three 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.<sup>2</sup>

One thing you'll notice is that the `Arrange` component for each of the three tests is identical and therefore a bit wasteful. When you have a scenario like this, that is, you need to perform some standard setup that multiple tests use; xUnit allows for that.

The xUnit documentation describes this concept as *Shared Context* between tests and specifies three 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)

---

<sup>2</sup><https://github.com/binarythistle/Complete-ASP-NET-3-API-Tutorial-Book>

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 `CommandTests` 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 of the sections, we have added [Figure 11-13](#).



```

using System;
using Xunit;
using CommandAPI.Models;

namespace CommandAPI.Tests
{
    public class CommandTests : IDisposable 1
    {
        Command testCommand; 2

        public CommandTests() 3
        {
            testCommand = new Command
            {
                HowTo = "Do something",
                Platform = "Some platform",
                CommandLine = "Some commandline"
            };
        }

        public void Dispose() 4
        {
            testCommand = null;
        }

        [Fact]
        public void CanChangeHowTo() 5
        {
            //Arrange
            //Act
            testCommand.HowTo = "Execute Unit Tests";

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

```

**Figure 11-13.** *Don't Repeat Yourself - refactored Model tests*

1. We inherit the IDisposable interface (used for code cleanup).
2. Create a “global” instance of our Command class.
3. Create a Class Constructor where we perform the setup 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 one test here for brevity).

For more information, refer to the xUnit documentation.<sup>3</sup>

Run your tests again and you should see three passing tests.



```
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:
Microsoft (R) Test Execution Command Line Tool Version 16.5.0
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: 3
    Passed: 3
Total time: 0.9666 Seconds
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\test\CommandAPI.Tests>
```

Figure 11-14. 3 Passing Tests

## Test Our Controller

OK, so testing our model was just an *amuse-bouche*<sup>4</sup> 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.

## Revisit Unit Testing Characteristics

I think before we move on, it's worth revisiting our Unit Testing Characteristics:

- **Fast:** Individual tests should execute quickly (required as we can have 1000s of them), and when we say quick, we're talking in the region of milliseconds.

<sup>3</sup><https://xunit.net/docs/shared-context>

<sup>4</sup>Bite-sized hors d'oeuvre, literally means "mouth amuser" in French. They differ from appetizers in that they are not ordered from a menu by patrons but are served free and according to the chef's selection alone.

- **Isolated:** Unit tests should not be dependent on external factors, for example, 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 earlier) should test only one thing.

I often struggle with the **Focused** characteristic and frequently have to pull myself back to testing *just one thing*, rather than wandering into integration test territory (and attempting to test an end-to-end flow). But that's not the characteristic I'm most worried about in this instance.

When we come to Unit testing our controller, the **Isolation** characteristic will present as problematic. Why? Let's remind ourselves of our Controller constructor.

```
[Route("api/{controller}")]
[ApiController]
public class CommandsController : ControllerBase
{
    private readonly ICommandAPIRepo _repository;
    private readonly IMapper _mapper;

    public CommandsController(ICommandAPIRepo repository, IMapper mapper)
    {
        _repository = repository;
        _mapper = mapper;
    }

    [HttpDelete("{id}")]
```

**Figure 11-15.** Reminder of the dependencies injected into the constructor

Even though we are using Dependency Injection (which is awesome), they are still dependencies as far as our controller is concerned, so when we come to unit testing the controller – how do we deal with this? Dependency Injection again? Stick a pin in that for now – I just want to plant the seed.

As before, I think the best way to learn about this is to get coding, so let's turn our attention back to our very first controller action: `GetAllCommands`.

## GetAllCommands Unit Tests and Groundwork

### GetAllCommands Overview

Let's remind ourselves of how `GetAllCommands` is supposed to be called.

Verb	URI	Operation	Description
GET	/api/commands	Read	Read all command resources

Additionally, I've provided some of the more detailed attributes of `GetAllCommands` that should help drive our testing.

Attribute	Description
Inputs	None; we simply make a GET request to the URI in the preceding table
Process	Attempt to retrieve a collection of command resources
Success Outputs	<ul style="list-style-type: none"> <li>HTTP 200 OK Status</li> </ul>
Failure Outputs	N/A: If this endpoint exists, it can't really be called "incorrectly"
Safe	Yes – Endpoint cannot alter our resources
Idempotent	Yes – Repeating the same operation will provide the same result

### GetAllCommands Unit Tests

What to test can be somewhat subjective, and from a test perspective, this is probably our simplest controller action, so I've settled on the following test cases.

Test ID	Arrange and action	Assert
Test 1.1	Request Resources when 0 exist	Return 200 OK HTTP Response
Test 1.2	Request Resources when 1 exists	Return a Single Command Object
Test 1.3	Request Resources when 1 exists	Return 200 OK HTTP Response
Test 1.4	Request Resources when 1 exists	Return the correct “type”

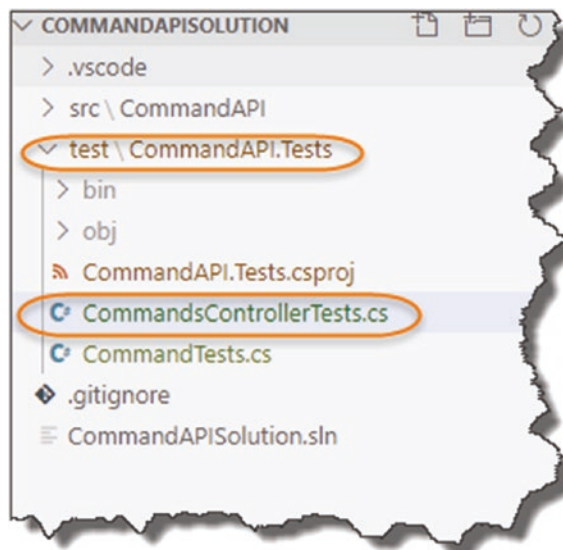
You’ll see that tests 1.2, 1.3, and 1.4 have the same Arrange and Action:

- Request a Resource when one exists.

So why not roll these into one test and perform the three assertions there? Well again that would break our **Focused** characteristic – we should be testing for one thing only per test.

## Groundwork for Controller Tests

As with our Command model, we want to create a separate test class in our unit test project to hold the controller tests, so create a class called *CommandsControllerTests.cs*, as shown in Figure 11-16.



**Figure 11-16.** Tests for the Controller

Place the following code into the *CommandsControllerTests.cs* file to get started:

```
using System;
using Xunit;
using CommandAPI.Controllers;
using Microsoft.AspNetCore.Mvc;

namespace CommandAPI.Tests
{
    public class CommandsControllerTests
    {
        [Fact]
        public void GetCommandItems_ReturnsZeroItems_WhenDBIsEmpty()
        {
            //Arrange
            //We need to create an instance of our CommandsController class
            var controller = new CommandsController( /* repository, AutoMapper */);
        }
    }
}
```

So straight away we want to start *arranging* our tests so that we have access to a *CommandsController* class to work with, but how do we create one when it has two dependencies (the repository and AutoMapper)? Dependency Injection – I hear you say! But if you look back at the anatomy of our Unit Test project, there’s no equivalent of the Startup class in which to Register our services for injection. We could start to add one I guess, but that would then lead to the problem of testing against our repository.

Even if we were to use Dependency Injection here, we’d still need to provide a concrete implementation instance; which one would we use? *SqlCommanAPIRepo*? That requires a DB Context, which in turn requires our Database. Argh! Not only is that horrifically complicated, we’re breaking the **Isolation** characteristic in a big way by dragging all that stuff into our unit testing.

We could move back to *MockCommandAPIRepo* and implement test code in there that wasn’t dependent on external factors, a possibility, but still a hassle – don’t worry there is a better way!

## Mocking Frameworks

Thankfully we can turn to something called “mocking,” which means we can quickly create “fake” (or mock) copies of any required objects to use within our unit tests. It allows us to self-contain everything we need in our unit test project and adhere to the **Isolation** principle. We can certainly use mocking for our repository and *possibly* AutoMapper.

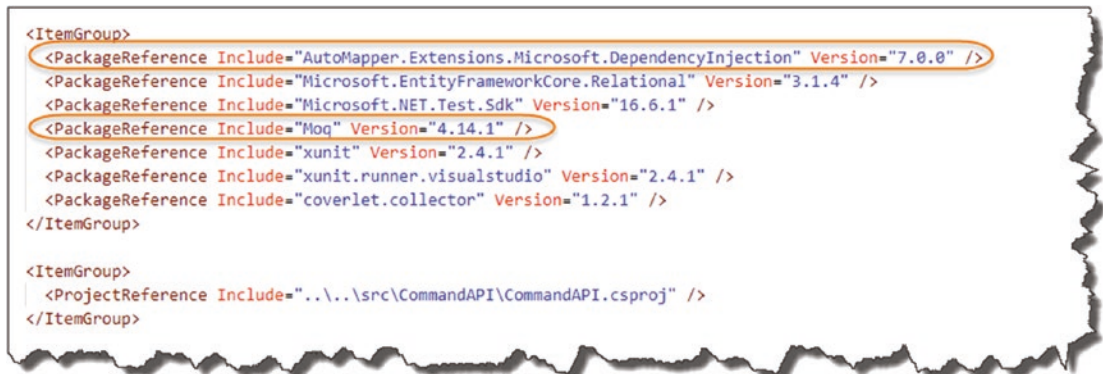
In order to use mocking, we need to turn to an external framework for this; the one I’ve chosen for us is called **Moq**. It’s fairly well understood and used within the C# .NET Community, so I thought it was a good choice for us.

## Install Moq and AutoMapper

Open a command prompt, and make sure you’re “in” the *CommandAPITests* folder, and issue the following commands:

```
dotnet add package Moq
dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection
```

Confirm that these dependencies have been added to the *CommandAPITests.csproj* file.



```
<ItemGroup>
  <PackageReference Include="AutoMapper.Extensions.Microsoft.DependencyInjection" Version="7.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Relational" Version="3.1.4" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.6.1" />
  <PackageReference Include="Moq" Version="4.14.1" />
  <PackageReference Include="xunit" Version="2.4.1" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.4.1" />
  <PackageReference Include="coverlet.collector" Version="1.2.1" />
</ItemGroup>

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

**Figure 11-17.** Package References for Moq

You’ll notice we’ve added AutoMapper in addition to Moq; we’ll require this later.

## Using Moq (Mock the Repository)

Returning to our `CommandsControllerTests` class, add the following code (taking note of our new using directives):

```
using System;
using System.Collections.Generic;
using Moq;
using AutoMapper;
using CommandAPI.Models;
using CommandAPI.Data;
using Xunit;
using CommandAPI.Controllers;
using Microsoft.AspNetCore.Mvc;

namespace CommandAPI.Tests
{
    public class CommandsControllerTests
    {
        [Fact]
        public void GetCommandItems_Returns200OK_WhenDBIsEmpty()
        {
            //Arrange
            var mockRepo = new Mock<ICommandAPIRepo>();

            mockRepo.Setup(repo =>
                repo.GetAllCommands()).Returns(GetCommands(0));

            var controller = new CommandsController(mockRepo.Object,
                /* AutoMapper*/ );
        }

        private List<Command> GetCommands(int num)
        {
            var commands = new List<Command>();
            if (num > 0){
                commands.Add(new Command

```



```
{
    Id = 0,
    HowTo = "How to generate a migration",
    CommandLine = "dotnet ef migrations add <Name of Migration>",
    Platform = ".Net Core EF"
});
}
return commands;
}
}
```

Or code is **still not runnable**, but I wanted to pause here and go through what we have added as there is quite a lot going on!

---

**i** Quick reminder, all this code is on GitHub<sup>5</sup> if you don't want to type this stuff in.

---

<sup>5</sup><https://github.com/binarythistle/Complete-ASP-NET-3-API-Tutorial-Book>

```

[Fact]
public void GetCommandItems_Returns200OK_WhenDBIsEmpty()
{
    //Arrange
    var mockRepo = new Mock<ICommandAPIRepo>();
    mockRepo.Setup(repo =>
        repo.GetAllCommands()).Returns(GetCommands(0));
    var controller = new CommandsController(mockRepo.Object, /* to do */);
}

private List<Command> GetCommands(int num)
{
    var commands = new List<Command>();
    if (num > 0)
    {
        commands.Add(new Command
        {
            Id = 0,
            HowTo = "How to generate a migration",
            CommandLine = "dotnet ef migrations add <Name of Migration>",
            Platform = ".Net Core EF"
        });
    }
    return commands;
}
    
```

**Figure 11-18.** Mocking our repository

1. We set up a new “mock” instance of our repository; note that we only need to pass the interface definition.
2. Using our new mock repository, we use the Setup method to establish how it will “behave.” Here, we specify the interface method we want to mock followed by what we want it to return (as described next).
3. Still in our Setup, we specify that the repository GetAllCommands method returns GetCommands(0) – see step 5.
4. We use the Object extension on our mock to pass in a mock *object instance* of ICommandAPIRepo.
5. We’ve mocked a private method: GetCommands that will return either an empty List or a List with one Command object depending on the value of the input parameter.

You can see how easy it is to set up mock objects using this type of framework, saving us a lot of the hassle of writing up our own mock classes. It also highlights the usefulness of our repository interface definition once again.

OK, so we've created a mock of our repository that we can use to create a `CommandsController` instance, but what about `AutoMapper`?

## Mock AutoMapper?

While you *can* use Moq to mock-up `AutoMapper`, we're not going to do that here. Why? Well because in this particular instance, general consensus is that it is more effective (and useful) to use an actual instance of `AutoMapper`. Additionally, using this approach we get to test the `AutoMapper` Profiles we've set up in our API Project too.

Now I want to sense check here.

This may seem completely contrary to the **Isolation** and **Focused** principles, and to some extent it is. My response to that is one of pragmatism (you may call it a cop out!), but the Unit Test Characteristics are just that: *Characteristics*. They are not unbreakable rules.

As developers we're often faced with choices and challenges. I may take one path, and you may choose another – personally I think that's fine. Coding can be as much art as science.

What we must strive to do is solve a problem the best way we can, and sometimes that involves compromise or, as I prefer to call it, *pragmatism*. In this case (in my view), using an instance of `AutoMapper` (as opposed to a mocked instance of it) provides more benefits than downsides, so that is the approach I'm going to take.

But please feel free to disagree!

So back in our `CommandsControllerTest` class, add the following code to provide us with an instance of `AutoMapper` (taking care to note the new `using` directive to bring in our Profiles):

```
using System;
using System.Collections.Generic;
using Moq;
using AutoMapper;
using CommandAPI.Models;
using CommandAPI.Data;
using CommandAPI.Profiles;
```

```

using Xunit;
using CommandAPI.Controllers;
using Microsoft.AspNetCore.Mvc;

namespace CommandAPI.Tests
{
    public class CommandsControllerTests
    {
        [Fact]
        public void GetCommandItems_Returns200OK_WhenDBIsEmpty()
        {
            //Arrange
            var mockRepo = new Mock<ICommandAPIRepo>();

            mockRepo.Setup(repo =>
                repo.GetAllCommands()).Returns(GetCommands(0));

            var realProfile = new CommandsProfile();
            var configuration = new MapperConfiguration(cfg =>
                cfg.AddProfile(realProfile));
            IMapper mapper = new Mapper(configuration);

            var controller = new CommandsController(mockRepo.Object, mapper);
        }

        .
        .
        .
    }
}

```

To step through the changes (for brevity I've not shown the using directives below), see Figure 11-19.

```

[Fact]
public void GetCommandItems_Returns200OK_WhenDBIsEmpty()
{
    //Arrange
    var mockRepo = new Mock<ICommandAPIRepo>();

    mockRepo.Setup(repo =>
        repo.GetAllCommands()).Returns(GetCommands(0));

    var realProfile = new CommandsProfile();
    var configuration = new MapperConfiguration(cfg =>
        cfg.AddProfile(realProfile));
    IMapper mapper = new Mapper(configuration);

    var controller = new CommandsController(mockRepo.Object, mapper);
}

```

**Figure 11-19.** Using AutoMapper in our tests

1. We set up a `CommandsProfile` instance and assign it to a `MapperConfiguration`.
2. We create a concrete instance of `IMapper` and give it our `MapperConfiguration`.
3. We pass our `IMapper` instance to our `CommandController` constructor.

There is a lot of new content and groundwork there, but now we're set up; the rest of this chapter should be quite quick! Make sure you save your work, build to check for errors, commit to GitHub, and we'll move onto completing our first test!

## Finish Test 1.1 – Check 200 OK HTTP Response (Empty DB)

Just to remind ourselves what we were wanting to test, see the following table.

Test ID	Arrange and action	Assert
Test 1.1	Request Resources when 0 exist	Return 200 OK HTTP Response

Back in the `CommandsControllerTests`, complete the code for our first test (make sure you include the `using` directive):

**using CommandAPI.Dtos;**

`//Arrange`

`.`  
`.`

`var controller = new CommandsController(mockRepo.Object, mapper);`

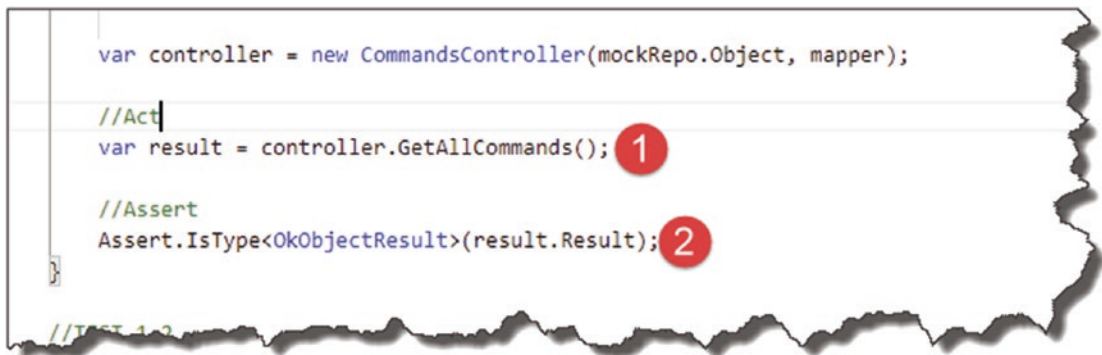
**`//Act`**

**`var result = controller.GetAllCommands();`**

**`//Assert`**

**`Assert.IsType<OkObjectResult>(result.Result);`**

To put these in context, see [Figure 11-20](#).



**Figure 11-20.** *Finalizing our Test*

1. We make a call to the `GetAllCommands` action on our Controller.
2. We Assert that the `Result` is an `OkObjectResult` (essentially equating to 200 OK).

As before, we can refactor our code to be a bit more reusable and place some of the common setup into a class constructor, as shown here:

```
public class CommandsControllerTests : IDisposable
{
    Mock<ICommandAPIRepo> mockRepo;
    CommandsProfile realProfile;
    MapperConfiguration configuration;
    IMapper mapper;

    public CommandsControllerTests()
    {
        mockRepo = new Mock<ICommandAPIRepo>();
        realProfile = new CommandsProfile();
        configuration = new MapperConfiguration(cfg => cfg.
AddProfile(realProfile));
        mapper = new Mapper(configuration);
    }

    public void Dispose()
    {
        mockRepo = null;
        mapper = null;
        configuration = null;
        realProfile = null;
    }

    [Fact]
    public void GetCommandItems_Returns200OK_WhenDBIsEmpty()
    {
        //Arrange
        mockRepo.Setup(repo =>
        repo.GetAllCommands()).Returns(GetCommands(0));

        var controller = new CommandsController(mockRepo.Object, mapper);
    }
}
```

```
//Act
var result = controller.GetAllCommands();
.
.
.
```

The only specific arrangement for this test case is the fact that we want the mock repository to return “0” resources.

If you want to “test your test,” save your work, and build the project, and then perform a `dotnet test` (of course inside the xUnit Project) to make sure it passes.

## Test 1.2 – Check Single Resource Returned

The second test checks to see that we get one resource returned.

Test ID	Arrange and action	Assert
Test 1.2	Request Resource when 1 exists	Return Single Resource



**Les’ Personal Anecdote** I debated on whether to include this test at all.

Depending on how you look at it, you may claim that this is not really testing our Controller but testing our Repository.

Nonetheless, I thought I’d include it to show you how to obtain this type of information.

In the code here, we configure our private `GetCommands` method to return one object. The “assertion” code looks a bit convoluted, but that is a consequence of how we have written our original controller action; here’s the code, and we’ll step through it here:

```
[Fact]
public void GetAllCommands_ReturnsOneItem_WhenDBHasOneResource()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetAllCommands()).Returns(GetCommands(1));
```



```

var controller = new CommandsController(mockRepo.Object, mapper);

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

//Assert
var okResult = result.Result as OkObjectResult;

var commands = okResult.Value as List<CommandReadDto>;

Assert.Single(commands);
}

```

To put in context, see Figure 11-21.

```

[Fact]
public void GetAllCommands_ReturnsOneResource_WhenDBHasOneResource()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetAllCommands()).Returns(GetCommands(1));

    var controller = new CommandsController(mockRepo.Object, mapper);

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

    //Assert
    var okResult = result.Result as OkObjectResult;
    var commands = okResult.Value as List<CommandReadDto>;
    Assert.Single(commands);
}

```

**Figure 11-21.** Getting to the Value

1. We arrange our mockRepo to return a single command resource.
2. In order to obtain the Value (see step 4), we need to convert our original result to an OkObjectResult object so we can then navigate the object hierarchy.

3. We obtain a list of `CommandReadDtos` (again we use the “as” keyword to assist here).
4. We assert that we have a `Single` result set on our `commands List`.



**Les' Personal Anecdote** Personally, I hate this code and think it's way too complex. The reason for this complexity stems from the fact that in our `GetAllCommands` controller action, we return our result set as follows:

```
return Ok(_mapper.Map<IEnumerable<CommandReadDto>>  
(commandItems));
```

Had we just used this

```
return _mapper.Map<IEnumerable<CommandReadDto>>(commandItems);
```

that is, returning our result set *not* enclosed in the `Ok()` method, navigation to our result set would be much simpler. So why did I write the controller action in the way I did? Simply because I wanted to! I wanted to be explicit in the way our successful results were returned.

There is an interesting discussion thread (isn't there always!) on this exact topic on Stack Overflow.<sup>6</sup> For now, my rant is over and we move on.

---

Save the code and perform a `dotnet test` to make sure it passes.

## Test 1.3 – Check 200 OK HTTP Response

The next test we want to check that the HTTP Response code is correct.

---

Test ID	Arrange and action	Assert
Test 1.3	Request Resource when 1 exists	Return HTTP 200 OK

---

---

<sup>6</sup><https://stackoverflow.com/questions/51489111/how-to-unit-test-with-actionresultt>

The code is quite straightforward, so don't think it requires much more explanation:

```
[Fact]
public void GetAllCommands_Returns200OK_WhenDBHasOneResource()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetAllCommands()).Returns(GetCommands(1));

    var controller = new CommandsController(mockRepo.Object, mapper);

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

    //Assert
    Assert.IsType<OkObjectResult>(result.Result);
}
```

## Test 1.4 – Check the Correct Object Type Returned

The final test is arguably the most useful one: it tests for the correct return *type*, in this case an `ActionResult` with an enumeration of `CommandReadDtos`.

Test ID	Arrange and action	Assert
Test 1.4	Request Resource when 1 exists	Return the correct “type”

```
[Fact]
public void GetAllCommands_ReturnsCorrectType_WhenDBHasOneResource()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetAllCommands()).Returns(GetCommands(1));

    var controller = new CommandsController(mockRepo.Object, mapper);

    //Act
    var result = controller.GetAllCommands();
```

```
//Assert
Assert.IsType<ActionResult<IEnumerable<CommandReadDto>>>(result);
}
```

Out of the four tests we've constructed for our first controller action, this is my favorite. Why? It's essentially testing our external contract. If we subsequently change how our controller behaves (and what it passes back to our consumers), this test will fail in regression. This is the mark of a valuable test for me!



**Les' Personal Anecdote** Now, I had an internal debate with myself whether to include the rest of the unit test code in the book or whether just to reference you off to GitHub, and we'd close this chapter off here.

The reason I had that debate was

1. I said no fluff/filler content - and you could argue that given the repeated nature of unit tests that we are going into that territory.
2. Most of the code that follows doesn't require much more explanation as we have covered the concepts already. So, it's just ends up as code on a page.

However, I *did* decide to keep the code here in the book, which means that chapter continues on. Why? In one word: Completeness. I wanted to produce the best product that I could, and I felt if I didn't keep all the code here in the book, it wouldn't be a complete product.

I hope you agree.

## GetCommandById Unit Tests

### GetCommandById Overview

Again, we'll remind ourselves how this endpoint is supposed to be called.

Verb	URI	Operation	Description
GET	/api/commands/{id}	Read	Read a single resource (by Id)

And some further detail to help us with defining our tests.

Attribute	Description
Inputs	The Id of the resource to be retrieved. This will be present in the URI of our GET request
Process	Attempt to retrieve the resource with the specified identifier
Success Outputs	<ul style="list-style-type: none"> <li>• 200 OK HTTP Response</li> <li>• Returned resource &lt;CommandReadDto&gt;</li> </ul>
Failure Outputs	<ul style="list-style-type: none"> <li>• 404 Not Found Response</li> </ul>
Safe	Yes – Endpoint <i>cannot</i> alter our resources
Idempotent	Yes – Repeating the same operation will provide the same result

## GetCommandById Unit Tests

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)	404 Not Found HTTP Response
Test 2.2	Resource ID is valid (exists in the DB)	200 Ok HTTP Response
Test 2.3	Resource ID is valid (exists in the DB)	Correct Resource Type Returned

### Test 2.1 – Check 404 Not Found HTTP Response

The code for this test is outlined here:

```
[Fact]
public void GetCommandById_Returns404NotFound_WhenNonExistentIDProvided()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetCommandById(0)).Returns(() => null);
}
```

```

var controller = new CommandsController(mockRepo.Object, mapper);

//Act
var result = controller.GetCommandById(1);

//Assert
Assert.IsType<NotFoundResult>(result.Result);
}

```

Here we setup the `GetCommandById` method on our mock repository to return null when an Id of “0” is passed in. This is a great demonstration of the real power of Moq. How simple was that to set up the behavior of our repository? The answer is very simple!

We then just check for the `NotFoundResult` type (equating to a 404 Not Found HTTP Response).

## Test 2.2 – Check 200 OK HTTP Response

The code for this test is outlined here:

```

[Fact]
public void GetCommandById_Returns200OK__WhenValidIDProvided()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetCommandById(1)).Returns(new Command { Id = 1,
        HowTo = "mock",
        Platform = "Mock",
        CommandLine = "Mock" });

    var controller = new CommandsController(mockRepo.Object, mapper);

    //Act
    var result = controller.GetCommandById(1);

    //Assert
    Assert.IsType<OkObjectResult>(result.Result);
}

```

The only novel code here is the way we set up the `GetCommandById` method on our repository to return a valid object, again very simple and quick. The rest of the code doesn't require further discussion.

## Test 2.3 – Check the Correct Object Type Returned

The code for this test is outlined here:

```
[Fact]
public void GetCommandById_Returns200OK__WhenValidIDProvided()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetCommandById(1)).Returns(new Command { Id = 1,
        HowTo = "mock",
        Platform = "Mock",
        CommandLine = "Mock" });

    var controller = new CommandsController(mockRepo.Object, mapper);

    //Act
    var result = controller.GetCommandById(1);

    //Assert
    Assert.IsType<ActionResult<CommandReadDto>>(result);
}
```

This test checks to see if we returned a `CommandReadDto`. In terms of checking for the validity of our externally facing contract, I like this test very much. If we changed our Controller code to return a different type, this test would fail, highlighting a potential problem with our contract – very useful.

# CreateCommand Unit Tests

## CreateCommand Overview

Here are the characteristics of the CreateCommand endpoint.

Verb	URI	Operation	Description
POST	/api/commands	Create	Create a new resource

A reminder on the detailed behavior outlined here.

Attribute	Description
Inputs	<p>The “command” object to be created</p> <p>This will be added to the request body of our POST request; an example is shown here:</p> <pre>{   "howTo": "Example how to",   "platform": "Example platform",   "commandLine": "Example command line" }</pre>
Process	Will attempt to add a new command object to our DB
Success	<ul style="list-style-type: none"> <li>• HTTP 201 Created Status</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>• Newly Created Resource (response body)</li> <li>• URI to newly created resource (response header)</li> </ul>
Failure Outputs	<ul style="list-style-type: none"> <li>• HTTP 400 Bad Request</li> <li>• HTTP 405 Not Allowed</li> </ul>
Safe	No – Endpoint can alter our resources
Idempotent	No – Repeating the same operation will incur a different result



## CreateCommand Unit Tests

Test ID	Condition	Expected Result
Test 3.1	Valid Object Submitted for Creation	Correct Object Type Returned
Test 3.2	Valid Object Submitted for Creation	201 Created HTTP Response

Now these tests may look a little spartan for this controller; could we not be testing more? I had originally conceived of the following additional tests

1. Test before and after object count of our repository (increment by 1).
2. Test if the content of the object passed back was correct.
3. Test for the 400 Bad Request.
4. Test for the 405 Not Allowed.

So why didn't I? Well tests 1 and 2 are not really testing our controller; they're really testing our repository. So as

- That's not the focus of our testing here.
- Our Repository is mocked.

I chose not to write unit tests for those. These cases could be considered valid *integration* tests that *included* our controller, but again that's not what we are doing here. (This is the trap I said I could fall into around the **Focused** unit test principle.)

For tests 3 and 4, the behavior demonstrated here derived from the default behaviors we get from decorating our controller with the `[ApiController]` attribute. This is not code I (or you) wrote – so I'm not going to write a unit test for code that I have no control over.

If I subsequently decided to add my own code to handle these conditions, then I'd probably introduce testing for them.

## Test 3.1 Check If the Correct Object Type Is Returned

The code for this test is outlined here:

```
[Fact]
public void CreateCommand_ReturnsCorrectResourceType_
WhenValidObjectSubmitted()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetCommandById(1)).Returns(new Command { Id = 1,
        HowTo = "mock",
        Platform = "Mock",
        CommandLine = "Mock" });

    var controller = new CommandsController(mockRepo.Object, mapper);

    //Act
    var result = controller.CreateCommand(new CommandCreatedDto { });

    //Assert
    Assert.IsType<ActionResult<CommandReadDto>>(result);
}
```

## Test 3.2 Check 201 HTTP Response

The code for this test is outlined here:

```
[Fact]
public void CreateCommand_Returns201Created_WhenValidObjectSubmitted()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetCommandById(1)).Returns(new Command { Id = 1,
        HowTo = "mock",
        Platform = "Mock",
        CommandLine = "Mock" });

    var controller = new CommandsController(mockRepo.Object, mapper);
```

```

//Act
var result = controller.CreateCommand(new CommandCreateDto { });

//Assert
Assert.IsType<CreatedAtRouteResult>(result.Result);
}

```

## UpdateCommand Unit Tests

### UpdateCommand Overview

Here are the characteristics of the UpdateCommand.

Verb	URI	Operation	Description
PUT	/api/commands/{Id}	Update (full)	Update all of a single resource (by Id)

Detailed behaviors are shown here.

Attribute	Description
Inputs (2)	<p>The Id of the resource to be updated. This will be present in the URI of our PUT request</p> <p>The full “command” object to be updated</p> <p>This will be added to the request body of our PUT request; an example is shown here:</p> <pre> {   "howTo": "Example how to",   "platform": "Example platform",   "commandLine": "Example command line" } </pre>
Process	Will attempt to fully update an existing command object in our DB
Success Outputs	<ul style="list-style-type: none"> <li>• HTTP 204 No Content response code</li> </ul>
Failure Outputs	<ul style="list-style-type: none"> <li>• HTTP 400 Bad Request</li> <li>• HTTP 404 Not Found</li> <li>• HTTP 405 Not Allowed</li> </ul>

Attribute	Description
Safe	No – Endpoint can alter our resources
Idempotent	Yes – Repeating the same operation will not incur a different result

## UpdateCommand Unit Tests

Test ID	Condition	Expected result
Test 4.1	Valid object submitted for update	204 No Content HTTP Response
Test 4.2	Nonexistent resource ID submitted for update	404 Not Found HTTP Response

Not too many tests here; points of note

- As we are not returning any resources back as part of our update, there are no tests checking for resource type this time.
- I have opted to test for the 404 Not Found result as this is behavior we actually wrote – so I want to test it.

### Test 4.1 Check 204 HTTP Response

The code for this test is outlined here:

[Fact]

```
public void UpdateCommand_Returns204NoContent_WhenValidObjectSubmitted()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetCommandById(1)).Returns(new Command { Id = 1,
        HowTo = "mock",
        Platform = "Mock",
        CommandLine = "Mock" });

    var controller = new CommandsController(mockRepo.Object, mapper);
```

```

//Act
var result = controller.UpdateCommand(1, new CommandUpdateDto { });

//Assert
Assert.IsType<NoContentResult>(result);
}

```

Here we ensure that the `GetCommandById` method will return a valid resource when we attempt to “update.” We then check to see that we get the success 204 No Content Response.

## Test 4.2 Check 404 HTTP Response

The code for this test is outlined here:

```

[Fact]
public void UpdateCommand_Returns404NotFound_
WhenNonExistentResourceIDSubmitted()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetCommandById(0)).Returns(() => null);

    var controller = new CommandsController(mockRepo.Object, mapper);
    //Act
    var result = controller.UpdateCommand(0, new CommandUpdateDto { });

    //Assert
    Assert.IsType<NotFoundResult>(result);
}

```

We setup our mock repository to return back null, which should trigger the 404 Not Found behavior.

# PartialCommandUpdate Unit Tests

## PartialCommandUpdate Overview

The behavior of the `PartialCommandUpdate` method is shown here.

Verb	URI	Operation	Description
PATCH	/api/commands/{Id}	Update (partial)	Update part of a single resource (by Id)

Detailed behavior here.

Attribute	Description
Inputs (2)	<p>The Id of the resource to be updated. This will be present in the URI of our PATCH request</p> <p>The change-set or “patch document” to be applied to the resource This will be added to the <code>request body</code> of our PATCH request; an example is shown here:</p> <pre>[   {     "op": "replace",     "path": "/howto",     "value": "Some new value"   },   {     "op": "test",     "path" : "commandline",     "value" : "dotnet new"   } ]</pre>
Process	<p>Will attempt to perform the updates as specified in the patch document</p> <p>Note: If there is more than one update, all those updates need to be successful. If one fails, then they all fail</p>
Success Outputs	<ul style="list-style-type: none"> <li>• HTTP 204 Not Content HTTP Status</li> </ul>

Attribute	Description
Failure Outputs	<ul style="list-style-type: none"> <li>• HTTP 400 Bad Request</li> <li>• HTTP 404 Not Found</li> <li>• HTTP 405 Not Allowed</li> </ul>
Safe	No – Endpoint can alter our resources
Idempotent	No – Repeating the same operation may incur a different result

## PartialCommandUpdate Unit Tests

Test ID	Condition	Expected Result
Test 5.1	Nonexistent resource ID submitted for update	404 Not Found HTTP Response

Even fewer tests here! As mentioned, when we implemented this endpoint, there are addition external dependencies required to get PATCH endpoints up and running. This cascades into unit testing too. The cost vs. benefit proposition of including the necessary inclusions to perform one unit test (testing for a 204 No Content) did not stack up for me and I assumed for you too as the reader! I have therefore included only one test below – the 404 Not Found Response.

### Test 5.1 Check 404 HTTP Response

The code for this test is outlined here:

```
[Fact]
public void PartialCommandUpdate_Returns404NotFound_
WhenNonExistentResourceIDSubmitted()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetCommandById(0)).Returns(() => null);

    var controller = new CommandsController(mockRepo.Object, mapper);
```

```

//Act
var result = controller.PartialCommandUpdate(0,
    new Microsoft.AspNetCore.JsonPatch.JsonPatchDocument<CommandUpdateDto>
{ });

//Assert
Assert.IsType<NotFoundResult>(result);
}

```

## DeleteCommand Unit Tests

### DeleteCommand Overview

An overview of our DeleteCommand is shown here.

Verb	URI	Operation	Description
DELETE	/api/commands/{Id}	Delete	Delete a single resource (by Id)

Further details of how this endpoint should operate are listed here.

Attribute	Description
Inputs	The Id of the resource to be deleted. This will be present in the URI of our DELETE request
Process	Will attempt to delete an existing command object to our DB
Success Outputs	<ul style="list-style-type: none"> <li>• HTTP 204 No Content HTTP result</li> </ul>
Failure Outputs	<ul style="list-style-type: none"> <li>• HTTP 404 Not Found HTTP result</li> </ul>
Safe	No – End point can alter our resources
Idempotent	Yes – Repeating the same operation will incur the same result



## DeleteCommand Unit Tests

Test ID	Condition	Expected result
Test 6.1	Valid resource Id submitted for deletion	204 No Content HTTP Response
Test 6.2	Nonexistent resource Id submitted for deletion	404 Not Found HTTP Response

### Test 6.1 Check for 204 No Content HTTP Response

The code for this test is outlined here:

```
[Fact]
public void DeleteCommand_Returns204NoContent_
WhenValidResourceIDSubmitted()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetCommandById(1)).Returns(new Command { Id = 1,
        HowTo = "mock", Platform = "Mock", CommandLine = "Mock" });

    var controller = new CommandsController(mockRepo.Object, mapper);

    //Act
    var result = controller.DeleteCommand(1);

    //Assert
    Assert.IsType<NoContentResult>(result);
}
```

### Test 6.2 Check for 404 Not Found HTTP Response

The code for this test is outlined here:

```
[Fact]
public void DeleteCommand_Returns_404NotFound_
WhenNonExistentResourceIDSubmitted()
{
```

```
//Arrange
mockRepo.Setup(repo =>
    repo.GetCommandById(0)).Returns(() => null);

var controller = new CommandsController(mockRepo.Object, mapper);

//Act
var result = controller.DeleteCommand(0);

//Assert
Assert.IsType<NotFoundResult>(result);
}
```

## Wrap It Up

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.

The main takeaways are

- The power of Moq to help Isolate ourselves when unit testing
- The somewhat arbitrary nature of what to test (use the characteristics as pragmatic guidelines)

With that we move into looking at how we'll deploy to production using a CI/CD pipeline on Azure DevOps!

## CHAPTER 12

# 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.
- Setup 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<sup>1</sup> or, if you prefer another analogy: a production line, where a product (in this instance working software) is taken from its raw form (code<sup>1</sup>) and gradually transformed into working software that's usable by the end users.

---

<sup>1</sup>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.

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. (Fret not, I'm not going to bang *that* drum too much.)

## CI/CD or CI/CD?

Don't worry, the heading is not a 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 one 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 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. 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).



**Figure 12-1.** *Continuous: integration, delivery, and deployment*

## So Which Is It?

Typically, when we talk about CI/CD, we talk about Continuous Integration and 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!

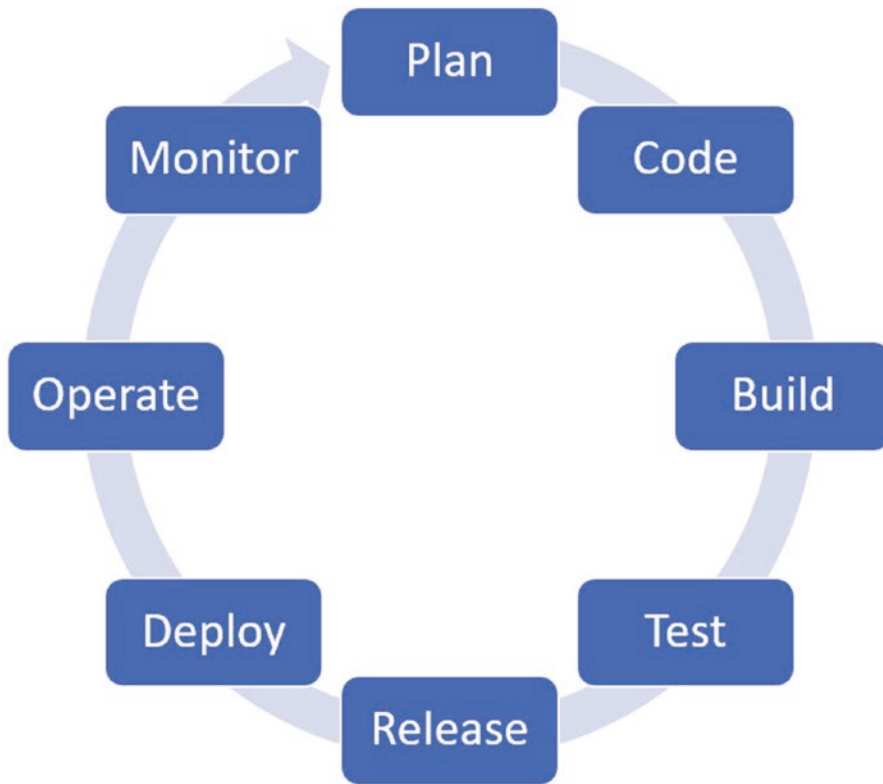
## The Pipeline

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



**Figure 12-2.** *The DevOps Pipeline*

You may also see it depicted as a “loop,” which kind of breaks the pipeline concept but is nonetheless useful when it comes to understanding the continuous cycle of DevOps activity.



**Figure 12-3.** *The DevOps "loop"*

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.



**Figure 12-4.** *Our Focus*

## What Is Azure DevOps?

Azure DevOps is a collection of tools that allow development teams to build and release software. It provides the following main features:

- **Dashboards:** For example, Red-Amber-Green (RAG) status of your pipeline, team members, etc.

- **Boards:** Allows you to capture and plan your work using methodologies like Scrum and Kanban.
- **Repos:** You can commit code (like we have done with GitHub) direct to Azure DevOps own repository.
- **Pipelines:** The automated CI/CD pipeline and our focus for Azure DevOps.
- **Test Plans:** End-to-end testing traceability for entire solutions.
- **Artifacts:** Package management, Artefact repo, etc.

In this chapter we are going to be focusing exclusively on the “Pipeline” feature and leave the other aspects untouched. As interesting as they are, to cover these would require a separate book and is outside our scope.

## Alternatives

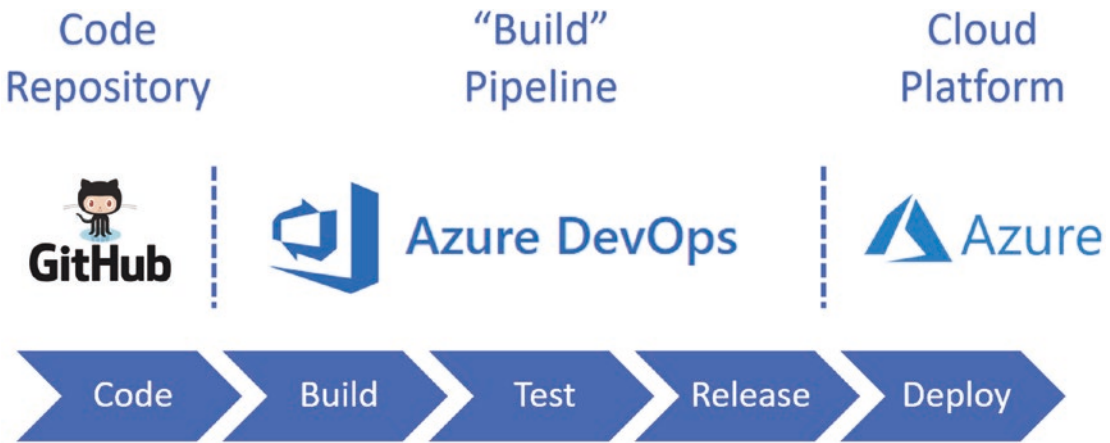
There are various on-premise and cloud-based alternatives to Azure DevOps: Jenkins is possibly the most “famous” of the on-premise solutions available, but you also have things like

- 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.

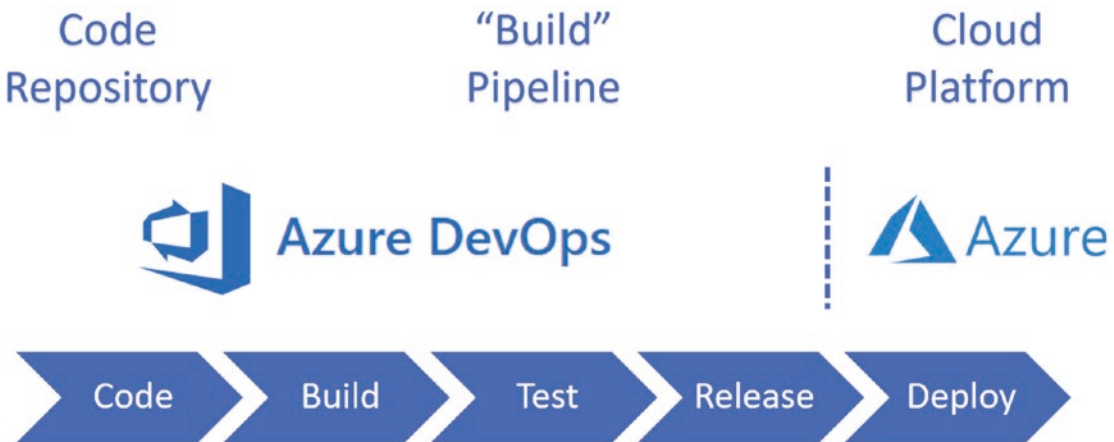
## Technology in Context

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



**Figure 12-5.** *The Technology mix we'll be using*

Indeed, Azure DevOps comes with its own "code repository" feature (as mentioned in Figure 12-5), which means we could do away with GitHub.



**Figure 12-6.** *Alternate technology mix*

So, our mix could look like the following.

Or if you wanted to take Microsoft technologies out of the picture, see Figure 12-7.





**Figure 12-7.** *Non-Microsoft mix*

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.

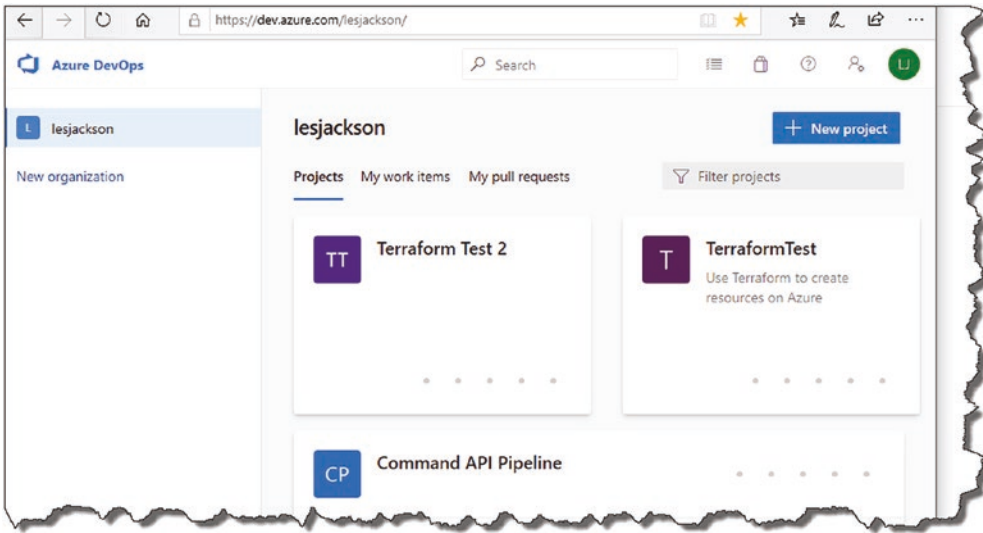
The takeaway points I wanted to make were

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 technology.

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 (be careful that you actually login to *Azure DevOps* and not Azure). The landing screen should look something like this (minus the projects I have).

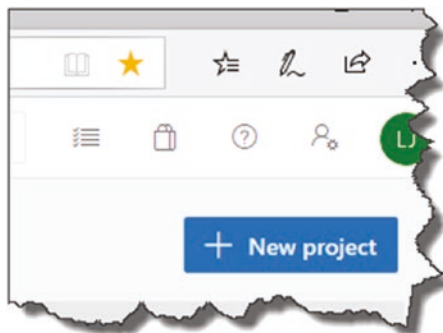


**Figure 12-8.** Azure DevOps landing page

**⚠ Warning!** When working with both Azure and Azure DevOps, one thing I’ve noticed is that the user interfaces can change rapidly. At the time of writing this (May 2020), the screenshots are correct and current, but just be aware that given the nature of these products, they can change from time to time.

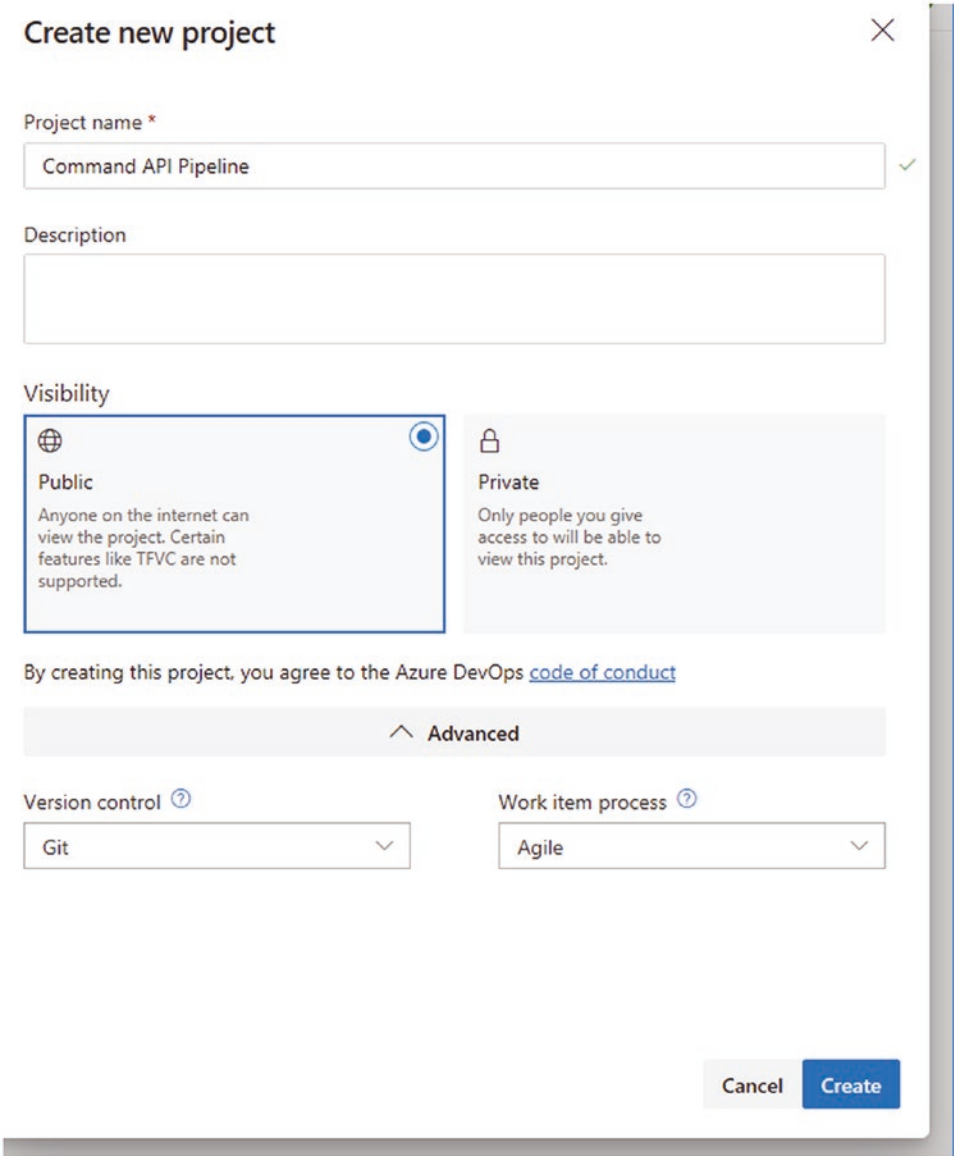
For the most part, these changes will be so small as to be inconsequential, for example, instead of “Create Project,” it becomes “New Project.” Other changes, while more significant, should still be easy enough to navigate through.

Once you have signed in/signed up, click “New Project.”



**Figure 12-9.** Create New Project

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

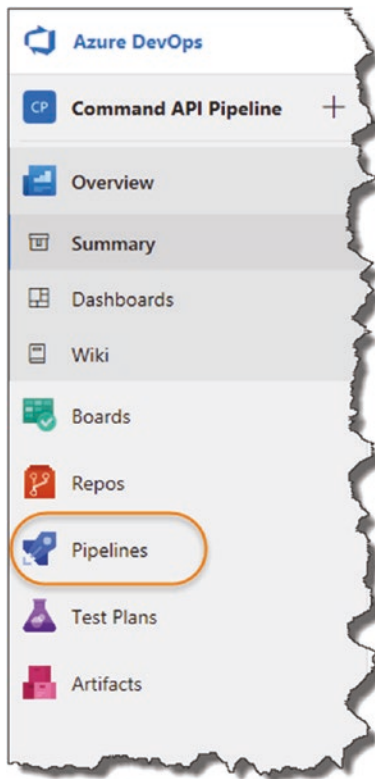


**Figure 12-10.** Name the project and select Public Visibility

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.

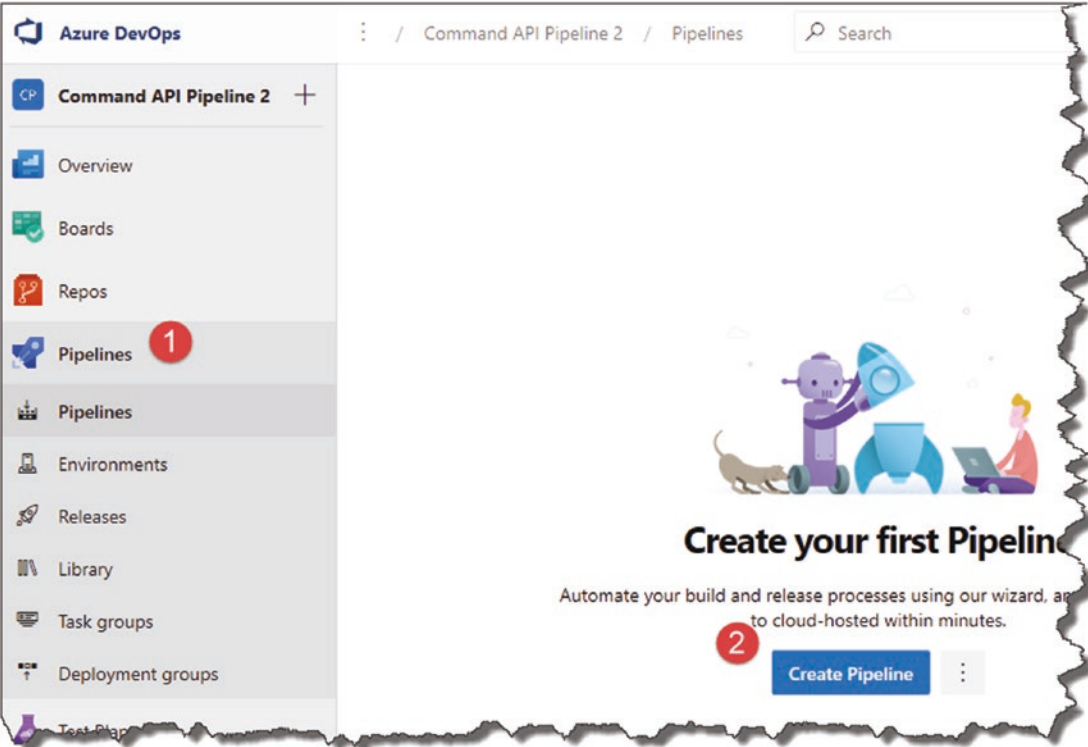
Once you’re happy, click “Create”; this will create your project and take you into the landing page.



**Figure 12-11.** *Select Pipelines*

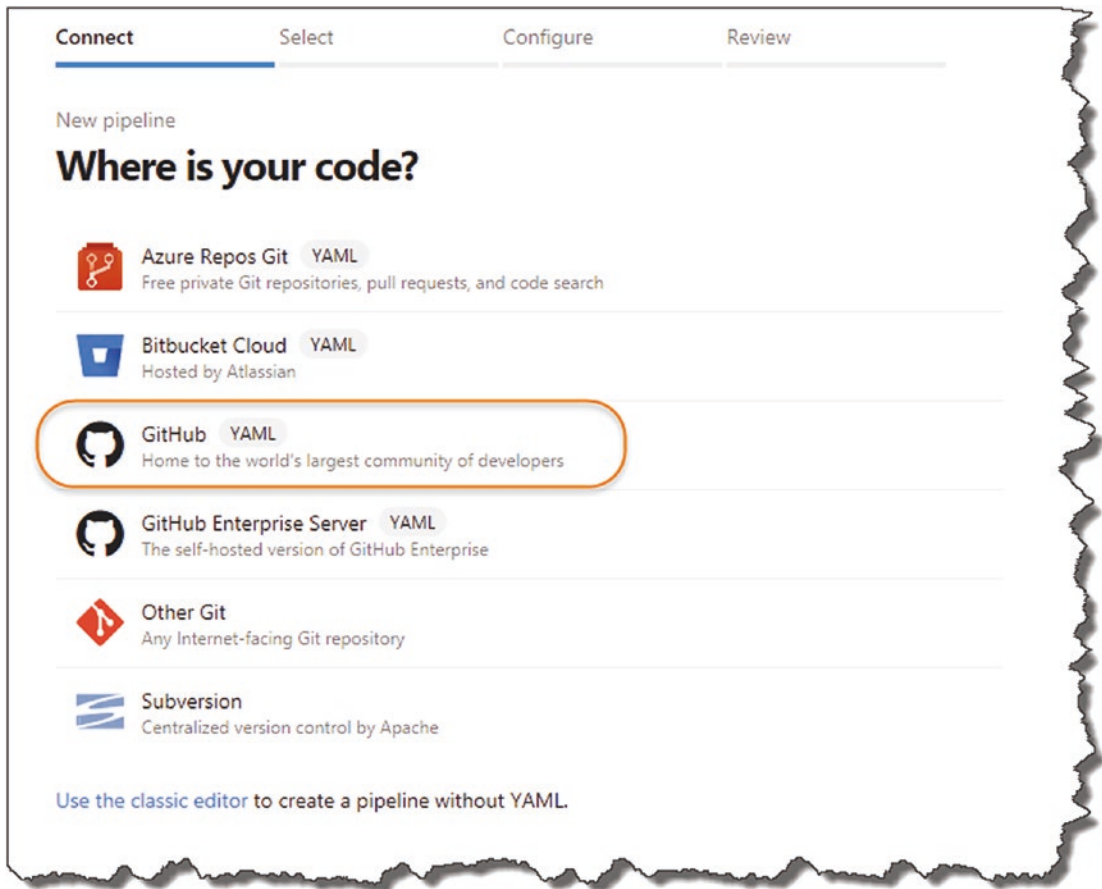
As discussed briefly, Azure DevOps has many features, but we’ll just be using the “Pipelines” for now. Select Pipelines, then

1. Create pipeline.



**Figure 12-12.** *Create a 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!



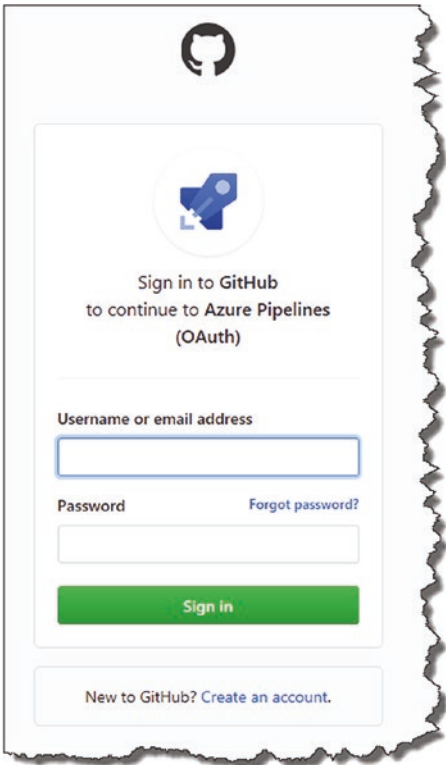
**Figure 12-13.** *GitHub is our code source*

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

---

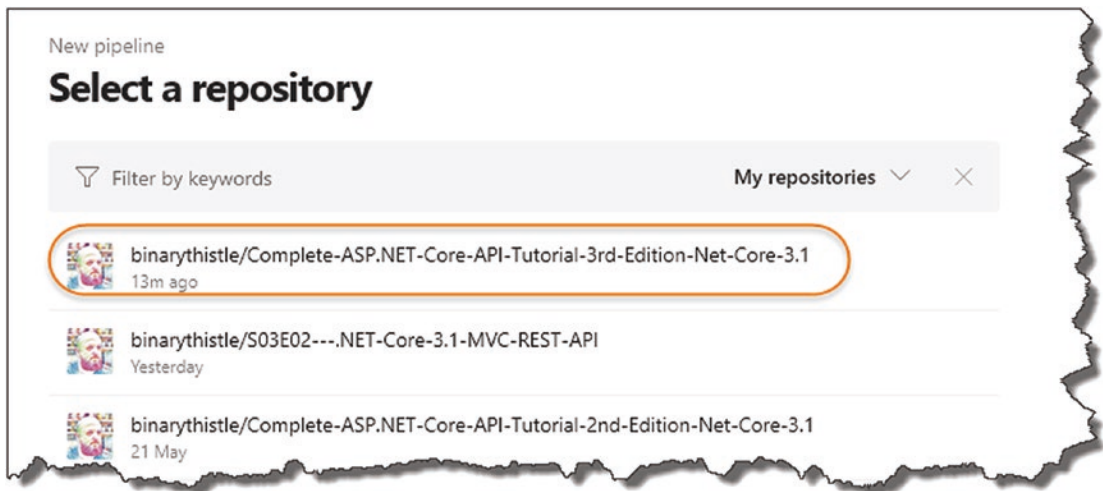
**Important** If this is the first time you're doing this, you'll need to give Azure DevOps permission to view your GitHub account.

---



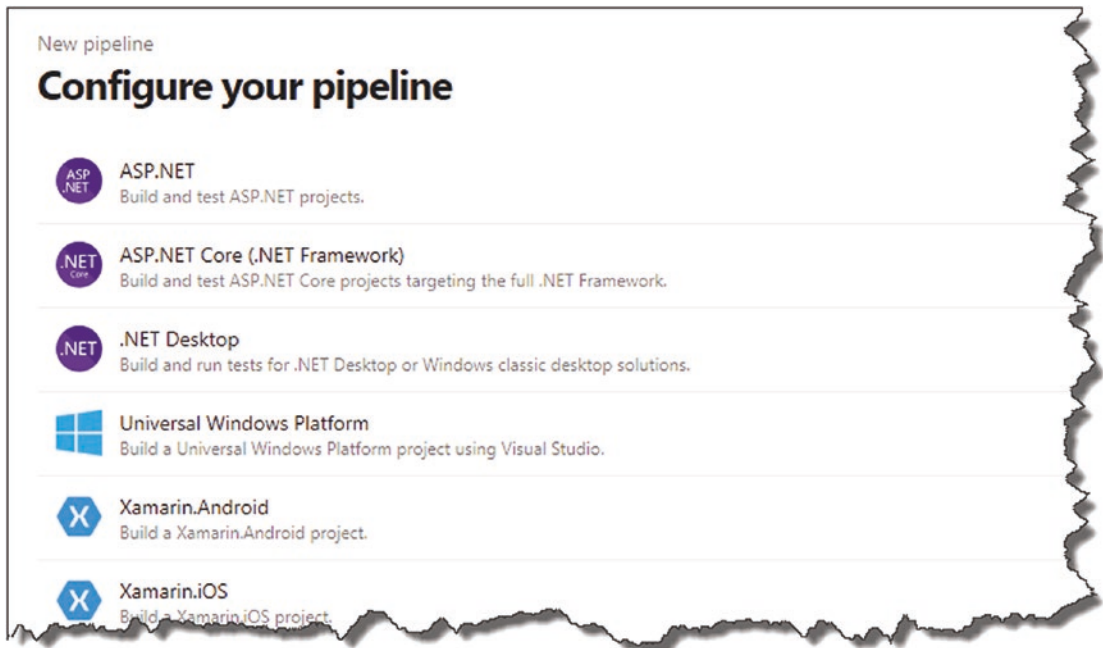
**Figure 12-14.** *You'll be asked to authenticate to GitHub*

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.



**Figure 12-15.** *Select the relevant API Repository*

Pick your repository (my example repository is shown in Figure 12-15); once you click it, Azure DevOps will go off and analyze it to suggest some common pipeline templates; you’ll see something like that in Figure 12-16.



**Figure 12-16.** *Pipeline Templates – we’ll create our own*



**i Note** Some readers have reported an additional step appearing here (that I cannot replicate) requesting that you approve and install Azure Pipelines. If you see this, I'd suggest you approve and proceed.

All this step will do is preconfigure the *azure-pipelines.yml* file for you (more on this next, but it's basically the instructions for our CI/CD pipeline). We are going to create our *azure-pipelines.yml* file from the ground up so it doesn't really matter which one you choose as we'll be overwriting it. Anyway, select an option and continue.



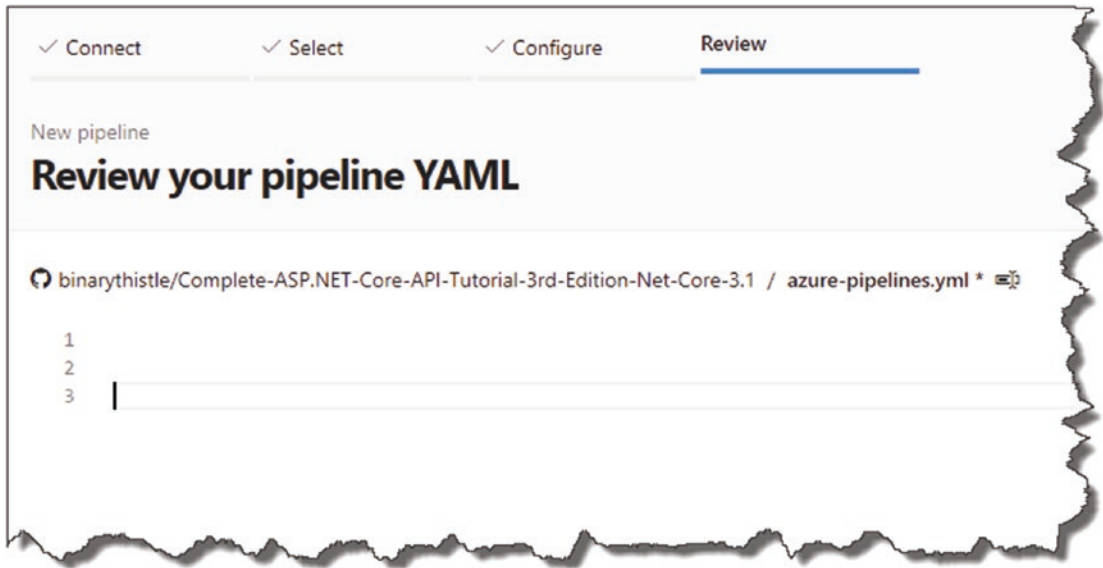
**Les' Personal Anecdote** This is one of the areas of Azure DevOps that appears to change a lot! I have at times in my career suggested using one of the off-the-shelf configurations as shown in Figure 12-16, but they seem to change so much that I felt a safer, more stable bet would be to create our own from the ground up.

Irrespective of which template you pick you'll get a default *azure-pipelines.yml* file, take a quick look (chances are yours will look different).

```
azure-pipelines.yml
1  # ASP.NET Core
2  # Build and test ASP.NET Core projects targeting .NET Core.
3  # Add steps that run tests, create a NuGet package, deploy, and more:
4  # https://docs.microsoft.com/azure/devops/pipelines/languages/dotnet-core
5
6  trigger:
7  - master
8
9  pool:
10  - vmImage: 'ubuntu-latest'
11
12  variables:
13  - buildConfiguration: 'Release'
14
15  steps:
16  - script: dotnet build --configuration $(buildConfiguration)
17  - displayName: 'dotnet build $(buildConfiguration)'
18
```

**Figure 12-17.** Example *Azure-pipelines.yml*

Select the entire contents, and press delete; your file should now be completely empty.



**Figure 12-18.** Empty Azure-Pipelines .yml

We are now going to add the first step to our file, which is simply to build our API Project. Before we do that, please read the warning below on formatting YAML files!

---

**⚠ 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.

---

Add the following code you your **azure-pipeline.yml** file:

```
trigger:
- master

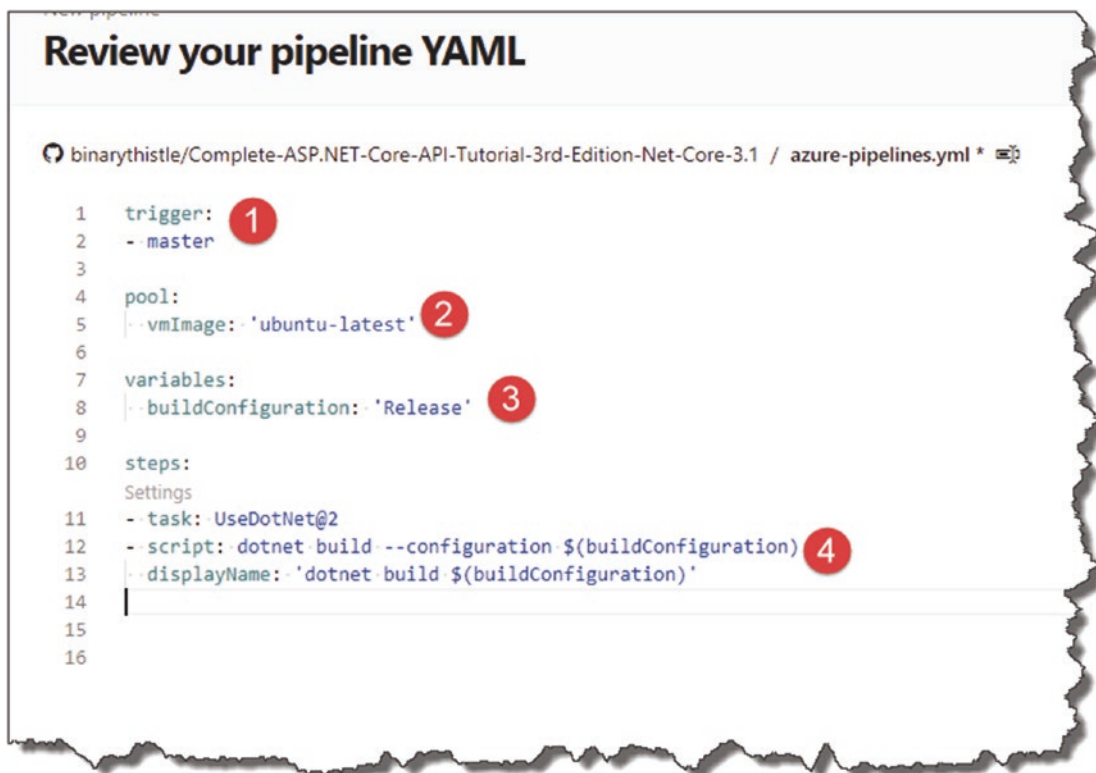
pool:
  vmImage: 'ubuntu-latest'

variables:
  buildConfiguration: 'Release'
```

steps:

- task: UseDotNet@2
- script: dotnet build --configuration \$(buildConfiguration)  
displayName: 'dotnet build \$(buildConfiguration)'

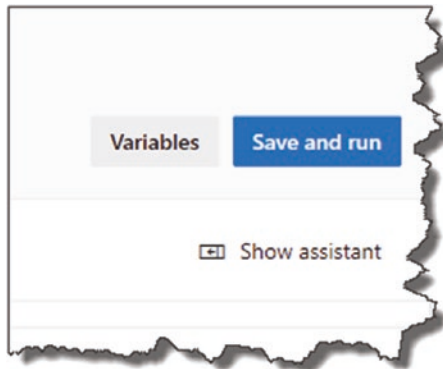
Your YAML file should look like this.



**Figure 12-19.** *Our Build Step*

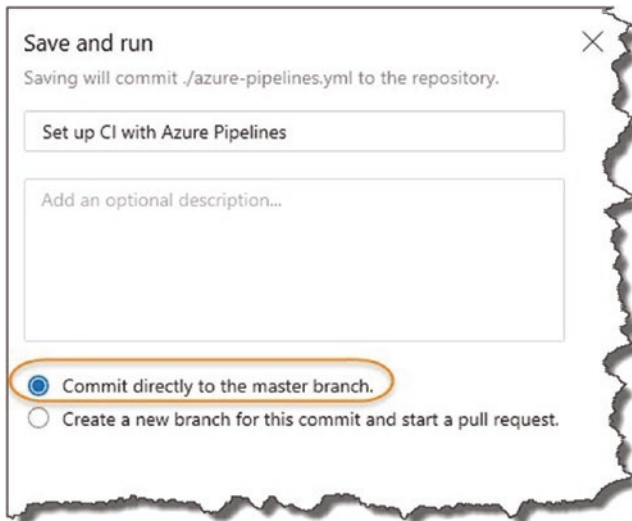
1. The trigger point for the pipeline (GitHub).
2. The image we will be performing the pipeline activities with.
3. Setup a variable to specify the build configuration.
4. A script task that performs a `dotnet build` for “Release.”

We're now ready to Click Save and run.



**Figure 12-20.** Manual Save and run

You'll then be presented with the following.



**Figure 12-21.** Commit the Azure-pipelines.yml to our GitHub repo

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 and run**.

An “agent” is then assigned to execute the pipeline; you’ll see various screens, such as in the next figures.

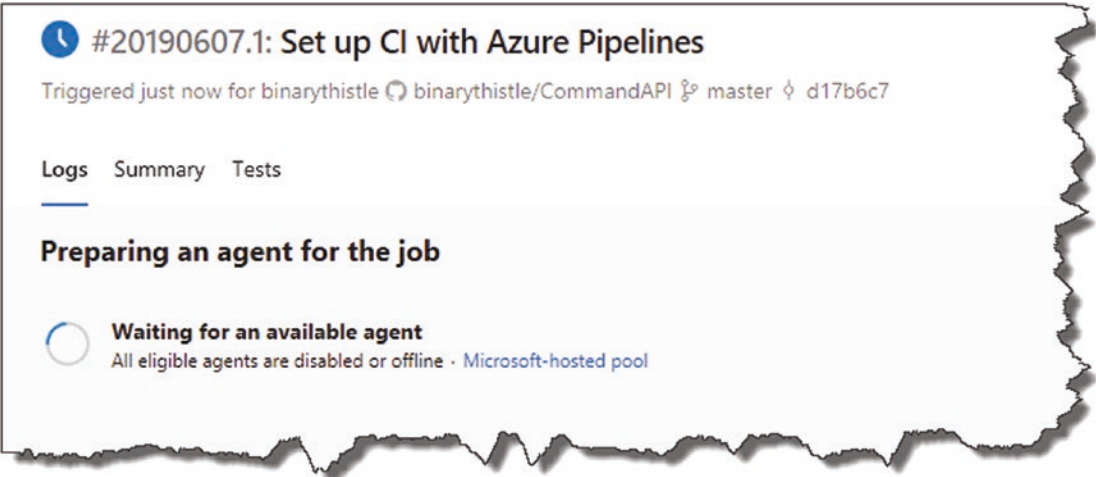


Figure 12-22. Job Preparation on Azure DevOps

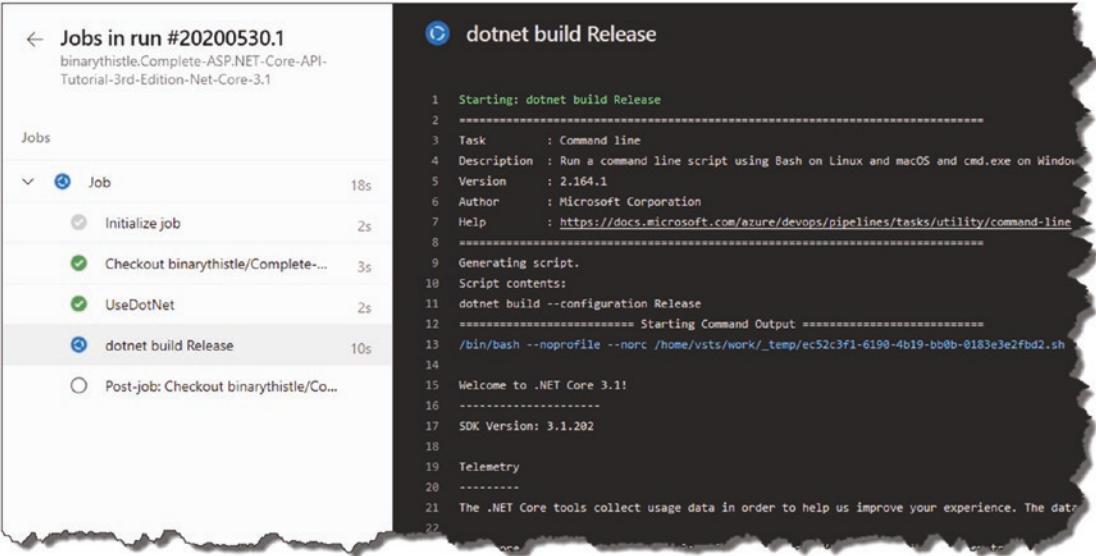


Figure 12-23. In-progress Job

And finally, you should see the completion screen.



Figure 12-24. Successful completion

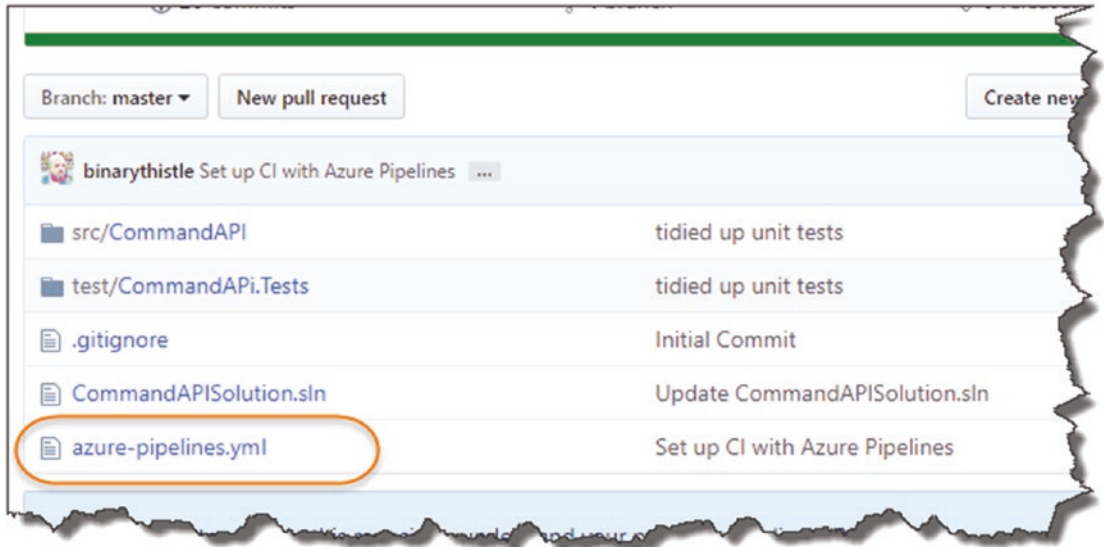
## What Just Happened?

OK, to recap

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

## Azure-Pipelines.yml File

Let's pop back over to our GitHub repository and refresh – you should see the following.



**Figure 12-25.** *azure-pipelines.yml* is in our repo

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

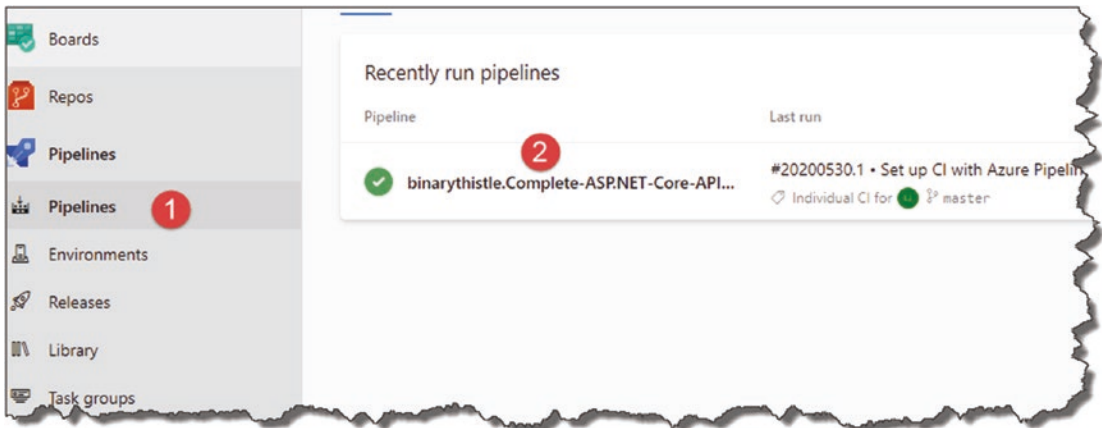
## I Thought We Wanted to Automate?

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!

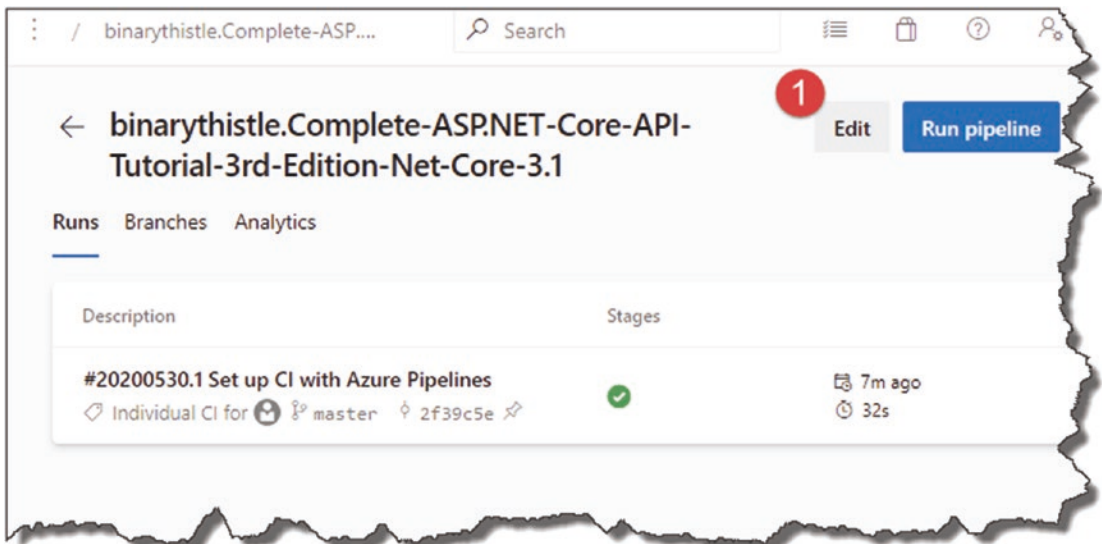
We are asked to execute when we created the pipeline that is true, but we can also set up “triggers,” meaning we can configure the pipeline to execute when it receives a particular event.

In your Azure DevOps project, click “Pipelines” under the Pipelines section, then select the pipeline.



**Figure 12-26.** Navigating back to *Azure-pipelines.yml*

Then click “Edit” on the next screen (top right).



**Figure 12-27.** Edit the pipeline

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.



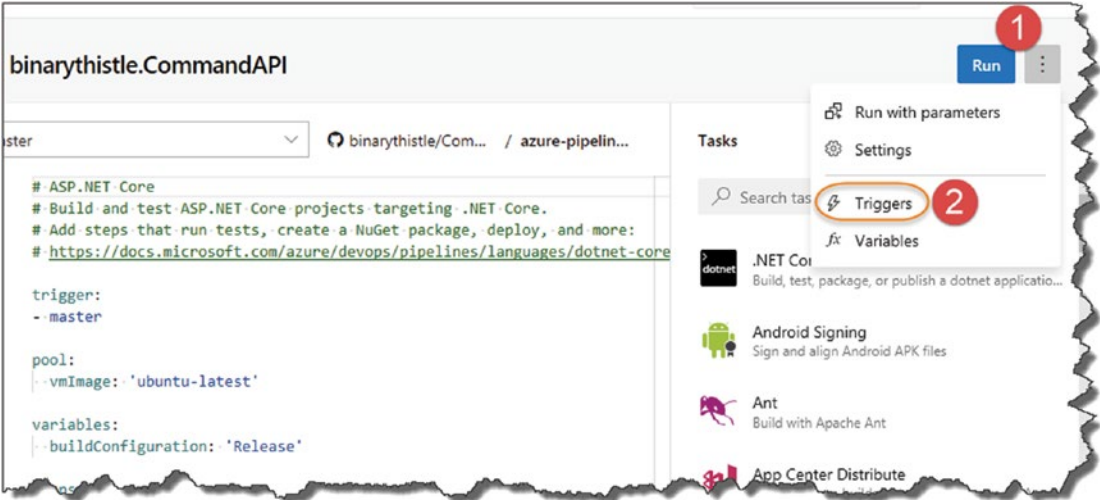


Figure 12-28. Select our Triggers

Here, you can see the Continuous Integration (CI) settings for our pipeline.

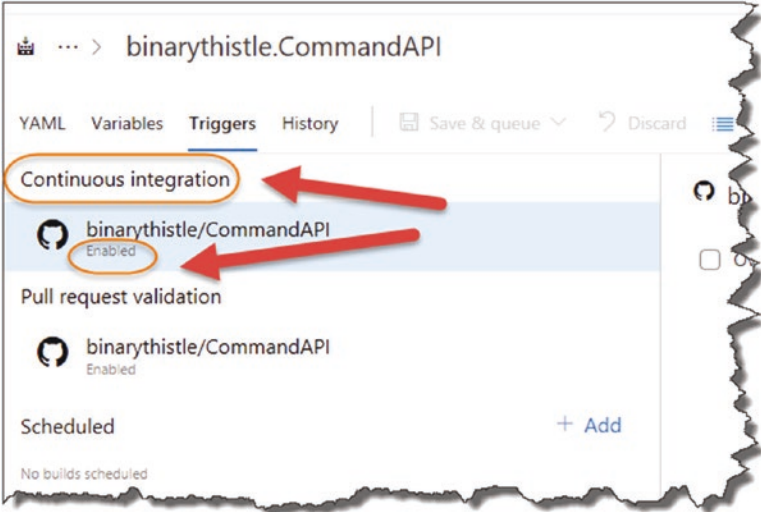
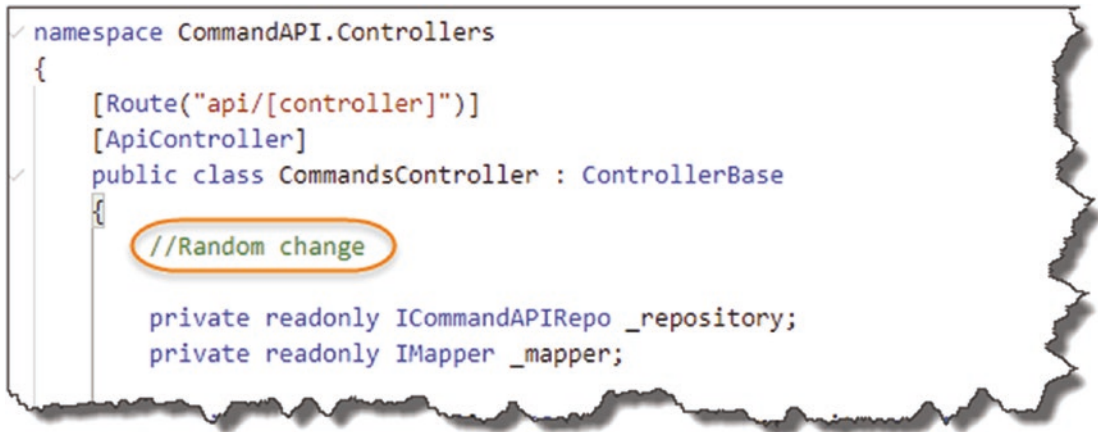


Figure 12-29. Check if Pipeline triggers are enabled for GitHub commit

You can see that the automation trigger is enabled by default (we have also configured this in the *azure-pipeline.yml* file), 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.



```
namespace CommandAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CommandsController : ControllerBase
    {
        //Random change

        private readonly ICommandAPIRepo _repository;
        private readonly IMapper _mapper;
    }
}
```

**Figure 12-30.** *Some random change*

Save the file, and perform the usual sequence of actions (make sure you are in the main Solution Folder – `CommandAPISolution`):

- `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.

```
1 file changed, 2 insertions(+), 2 deletions(-)
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> git push origin master
To https://github.com/binarythistle/Complete-ASP.NET-Core-API-Tutorial-2nd-Edition-Net-Core-3.1.git
! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/binarythistle/Complete-ASP.NET-Core-API-Tutorial-2nd-Edition-Net-Core-3.1.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
PS D:\APITutorial\NET Core 3.1\CommandAPISolution>
```

Figure 12-31. Our Local and Remote Repos are out of sync

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
```

Or if that doesn't work, use

```
git pull origin master
```

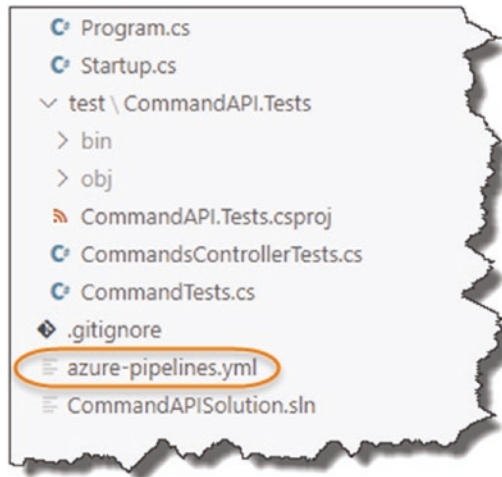
This pulls down the changes from the remote GitHub repository and merges them with our local one.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS D:\APITutorial\NET Core 3.1\CommandAPISolution> git pull
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 11 (delta 7), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (11/11), done.
From https://github.com/binarythistle/Complete-ASP.NET-Core-API-Tutorial-2nd-
259d1f5..c5d405f master -> origin/master
Merge made by the 'recursive' strategy.
azure-pipelines.yml | 22 ++++++
1 file changed, 22 insertions(+)
 create mode 100644 azure-pipelines.yml
PS D:\APITutorial\NET Core 3.1\CommandAPISolution>
```

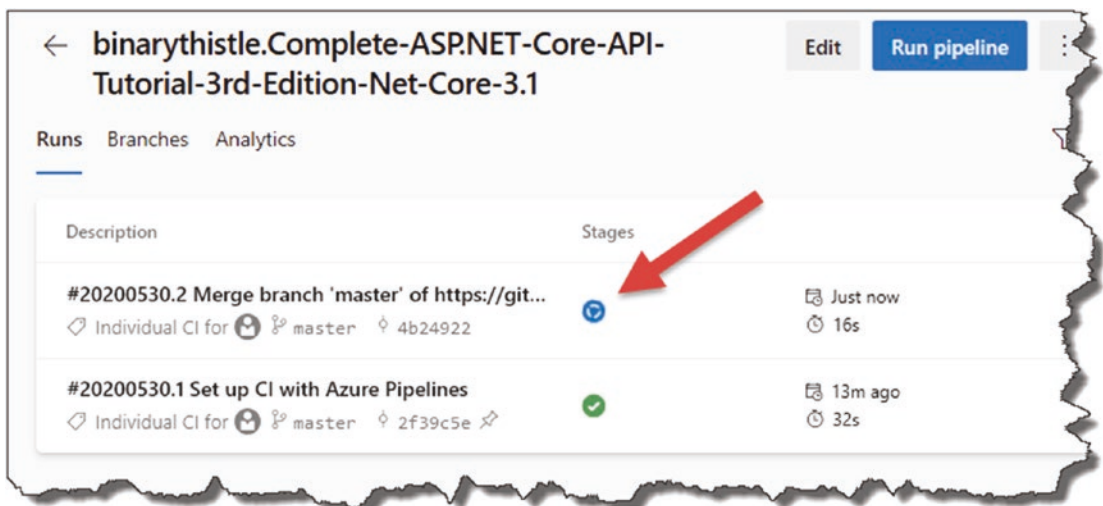
Figure 12-32. Pull down the *azure-pipelines.yml*

Indeed, if you look the VS Code file tree, you'll see our *azure-pipelines.yml* file has appeared!



**Figure 12-33.** We have *azure-pipelines.yml* locally now

Now that 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 Pipelines > Builds; you should see something like this.



**Figure 12-34.** Auto-triggered build

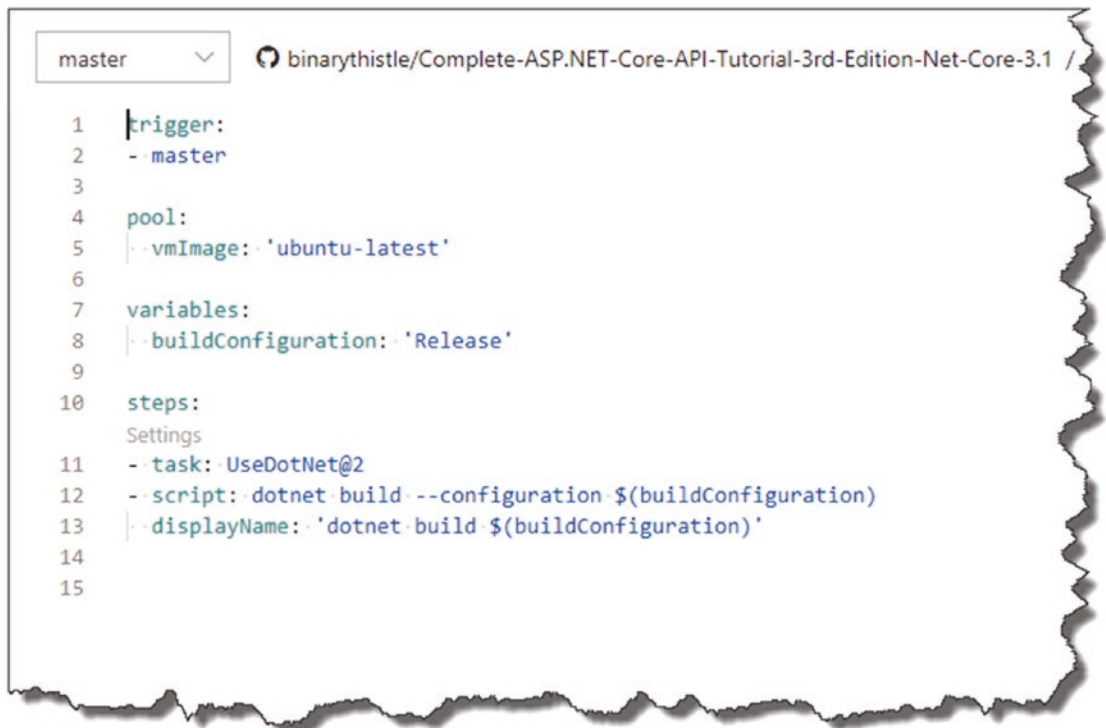
A new build has been queued to start – this time triggered by a remote commit to GitHub!

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 (follow the steps earlier if you forgot how to get here), you should see the following.

The image shows a screenshot of the Azure DevOps web interface. At the top, there is a dropdown menu set to 'master' and a breadcrumb path: 'binarythistle/Complete-ASP.NET-Core-API-Tutorial-3rd-Edition-Net-Core-3.1 /'. Below this, the content of the 'azure-pipelines.yml' file is displayed in a code editor with line numbers 1 through 15. The code defines a pipeline trigger for the 'master' branch, uses the 'ubuntu-latest' VM image, sets the build configuration to 'Release', and includes a 'UseDotNet@2' task to build the project.

```
1 trigger:
2   - master
3
4 pool:
5   - vmImage: 'ubuntu-latest'
6
7 variables:
8   - buildConfiguration: 'Release'
9
10 steps:
11   Settings
12   - task: UseDotNet@2
13     - script: dotnet build --configuration $(buildConfiguration)
14     - displayName: 'dotnet build $(buildConfiguration)'
15
```

**Figure 12-35.** Our `azure-pipelines.yml`

This is of course the code we added before; you’ll notice it doesn’t perform any testing or packaging steps, yet.

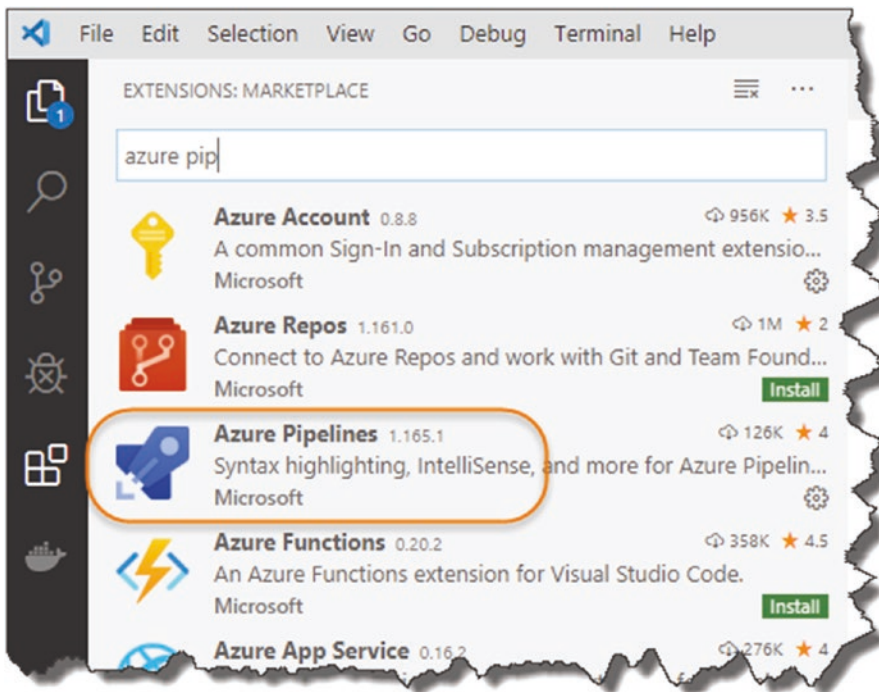
## Another VS Code Extension

As we are going to be doing a bit of editing of the *azure-pipelines.yml* file, there are two 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 has 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 the Extensions button, and search for "Azure Pipelines"; you should see the following.



**Figure 12-36.** Azure Pipelines Extension for VS Code

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, see Figure 12-37.



**Figure 12-37.** *The pipeline we'll be building*

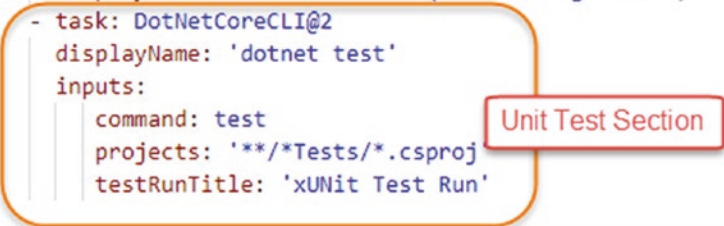
You'll see the suggested sequencing is Build ► **Test** ► Release, so let's add that task to our *azure-pipelines.yml* file now.

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
1  trigger:
2  - master
3
4  pool:
5  | vmImage: 'ubuntu-latest'
6
7  variables:
8  | buildConfiguration: 'Release'
9
10 steps:
11 - task: UseDotNet@2
12 - script: dotnet build --configuration $(buildConfiguration)
13   displayName: 'dotnet build $(buildConfiguration)'
14 - task: DotNetCoreCLI@2
15   displayName: 'dotnet test'
16   inputs:
17   | command: test
18   | projects: '**/*Tests/*.csproj'
19   | testRunTitle: 'xUnit Test Run'
20
21
```



**Figure 12-38.** Testing step added

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.



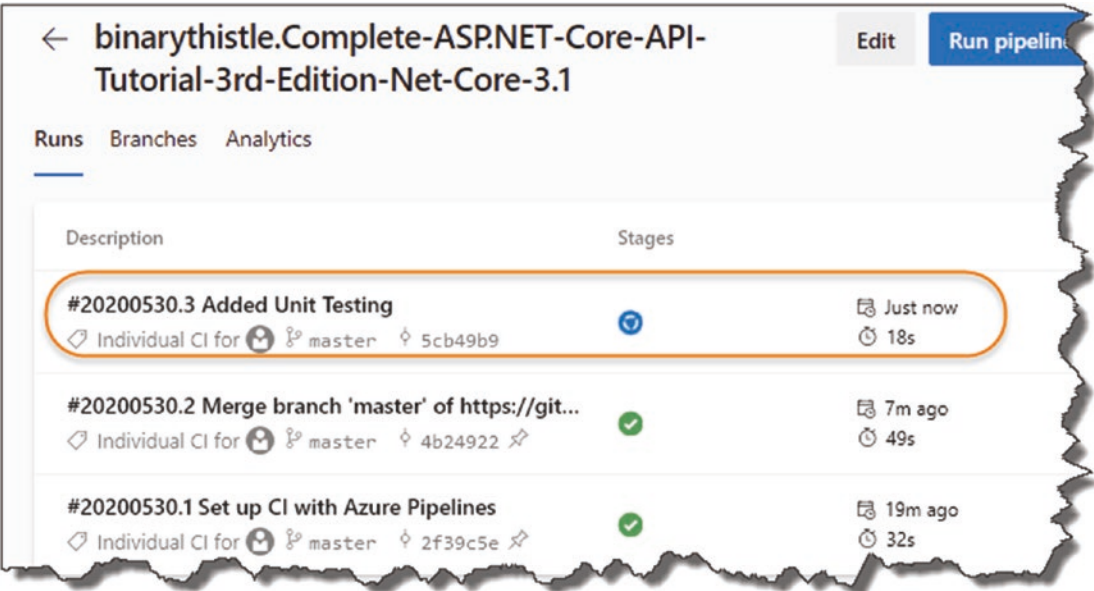
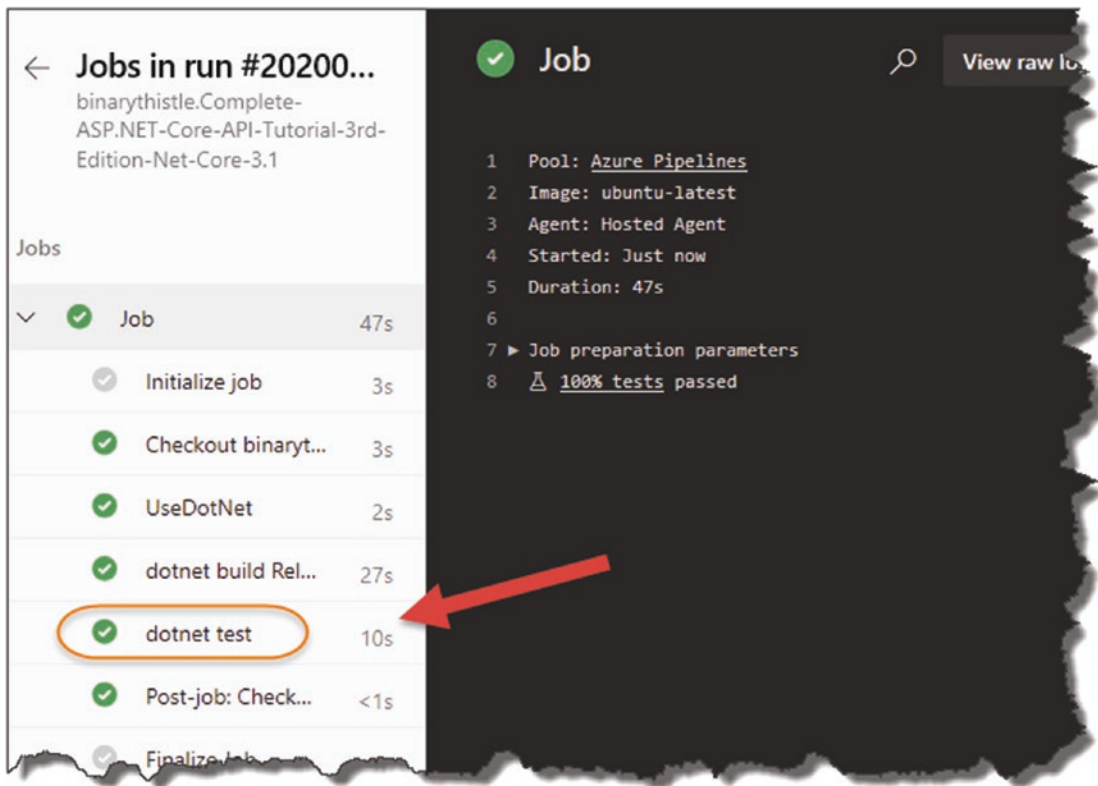


Figure 12-39. Pipeline triggered again

And this time the unit tests should execute too.



**Figure 12-40.** *Testing step has succeeded*

Click the dotnet test step as shown to drill down to see what’s going on; you should see something like the following.

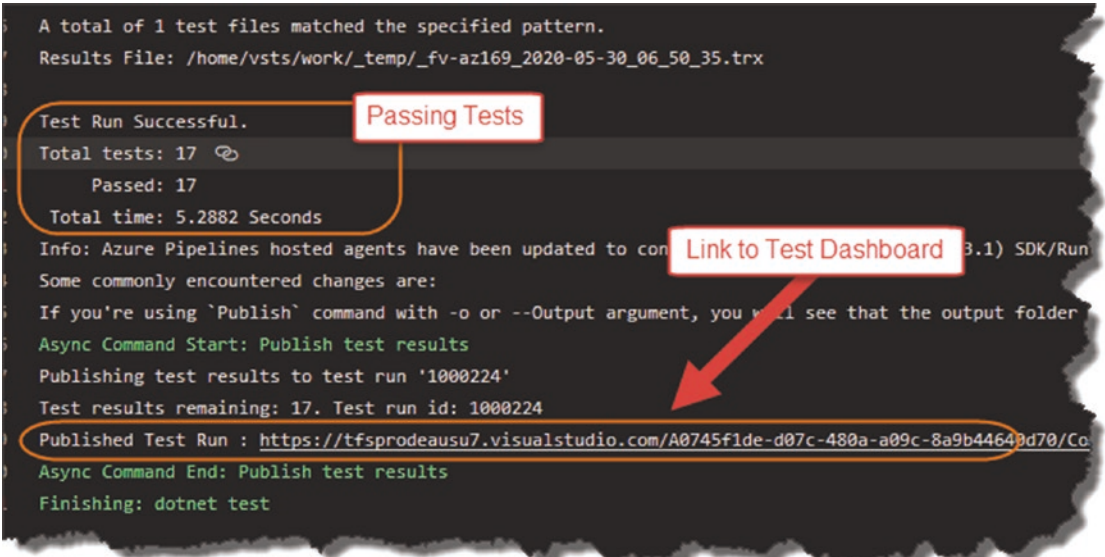


Figure 12-41. More detail on testing

Clicking the link highlighted in Figure 12-41 takes you to the test result dashboard.

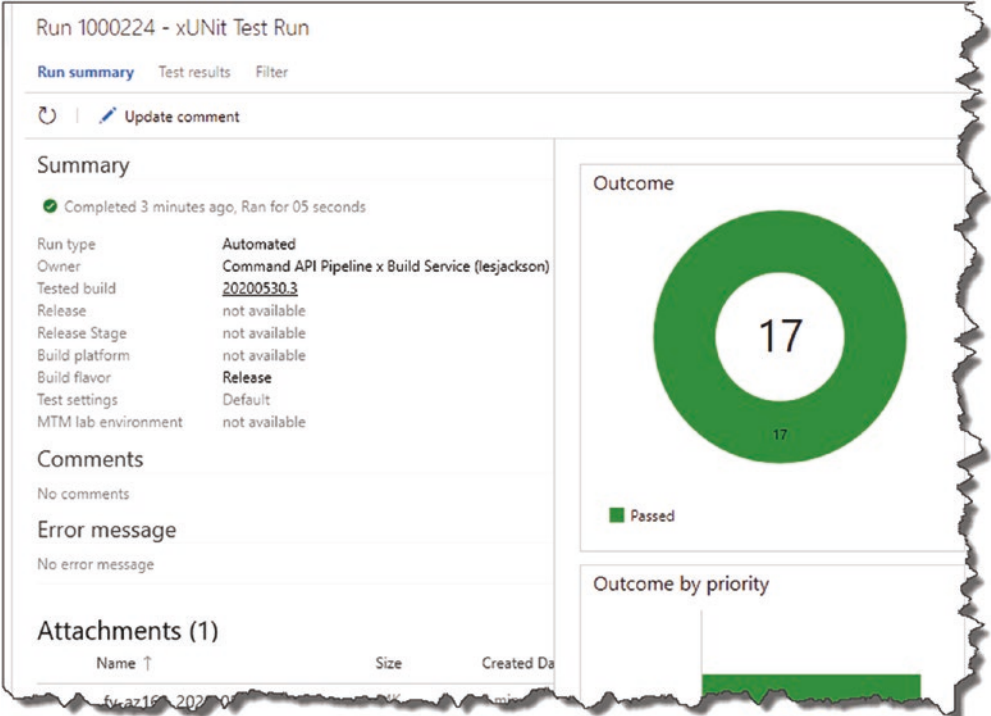


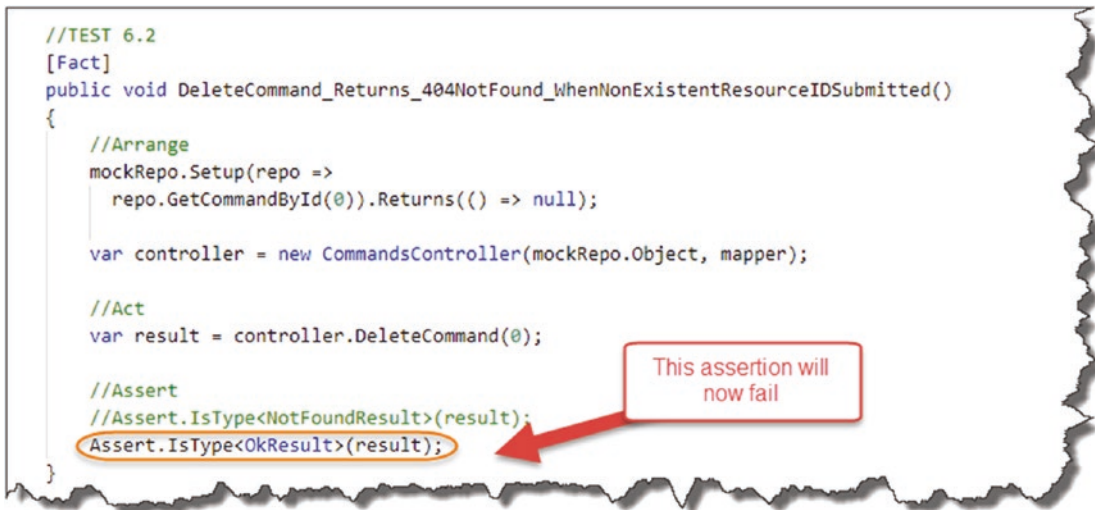
Figure 12-42. Testing Dashboard

Very nice! Indeed, this is the type of *Information Radiator* that you should make highly visible when working in a team environment, as it helps everyone understand the health of the build and, if necessary, take action to remediate any issues.

## Breaking Our Unit Tests

Now just to labor 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 here and swapped *NotFoundResult* with *OkResult*:

A screenshot of a code editor showing a unit test. The test is named 'DeleteCommand\_Returns\_404NotFound\_WhenNonExistentResourceIDSubmitted()'. It includes 'Arrange', 'Act', and 'Assert' sections. In the 'Assert' section, the line 'Assert.IsType<OkResult>(result);' is circled in orange. A red arrow points from a red-bordered callout box containing the text 'This assertion will now fail' to the circled line. The code is as follows:

```
//TEST 6.2
[Fact]
public void DeleteCommand_Returns_404NotFound_WhenNonExistentResourceIDSubmitted()
{
    //Arrange
    mockRepo.Setup(repo =>
        repo.GetCommandById(0)).Returns(() => null);

    var controller = new CommandsController(mockRepo.Object, mapper);

    //Act
    var result = controller.DeleteCommand(0);

    //Assert
    //Assert.IsType<NotFoundResult>(result);
    Assert.IsType<OkResult>(result);
}
```

**Figure 12-43.** Break our unit test

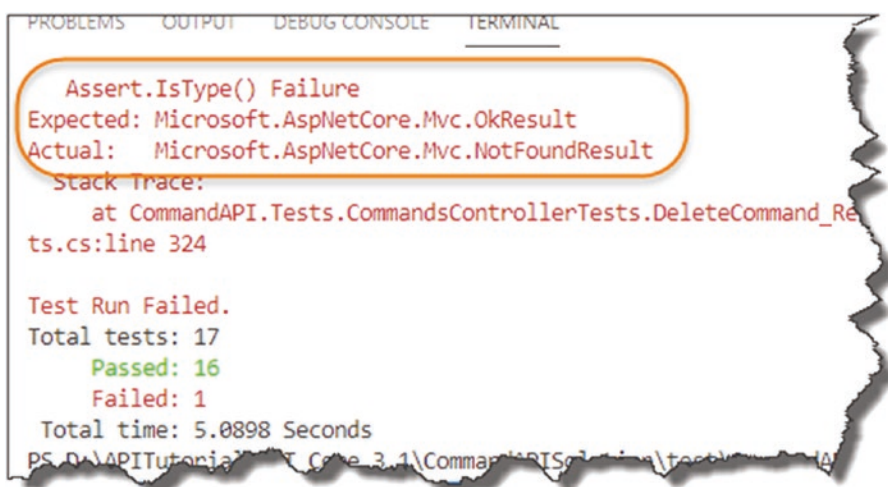
Save the file, and (ensuing you're "in" the *CommandAPI.Tests* project) run a build:

```
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.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Assert.IsType() Failure
Expected: Microsoft.AspNetCore.Mvc.OkResult
Actual: Microsoft.AspNetCore.Mvc.NotFoundResult
Stack Trace:
at CommandAPI.Tests.CommandsControllerTests.DeleteCommand_Re
ts.cs:line 324

Test Run Failed.
Total tests: 17
Passed: 16
Failed: 1
Total time: 5.0898 Seconds
PS D:\APITutorial\Code\3.1\CommandAPI\src\tests>
```

**Figure 12-44.** Test has failed locally

Now under normal circumstances, 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 we have done so and commit their code; again this just highlights the value in a CI/CD build pipeline.

---

So, perform the three “Git” steps you should be familiar with now (ensure you do this at the solution level), and once you’ve pushed to GitHub, move back across to Azure DevOps, and observe what happens.

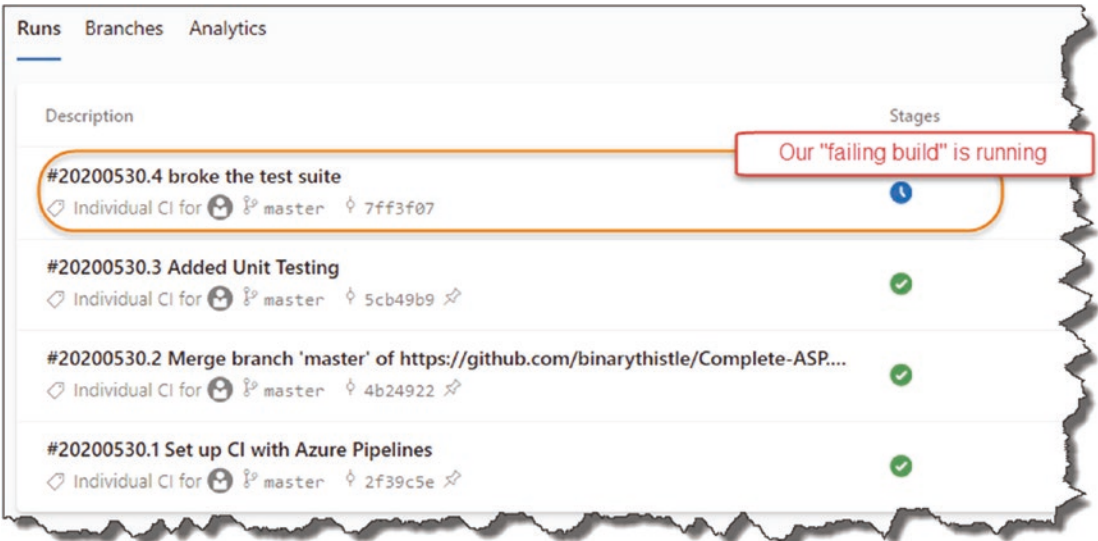


Figure 12-45. In-progress Pipeline (it will error out)

Then as expected, our test fails.

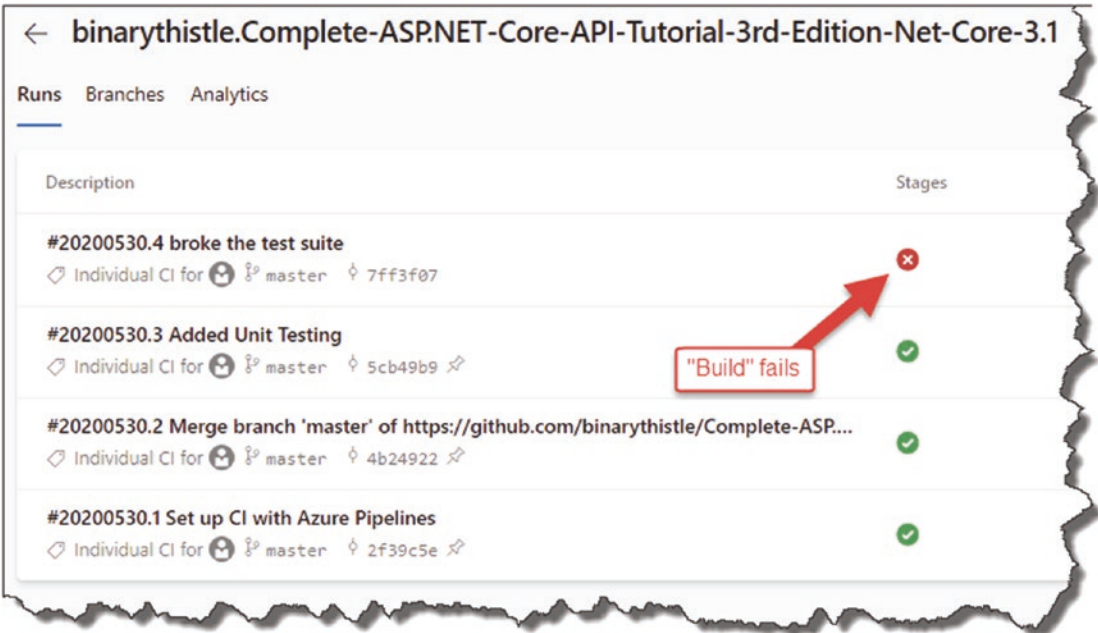


Figure 12-46. Failed!

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, see Figure 12-47.

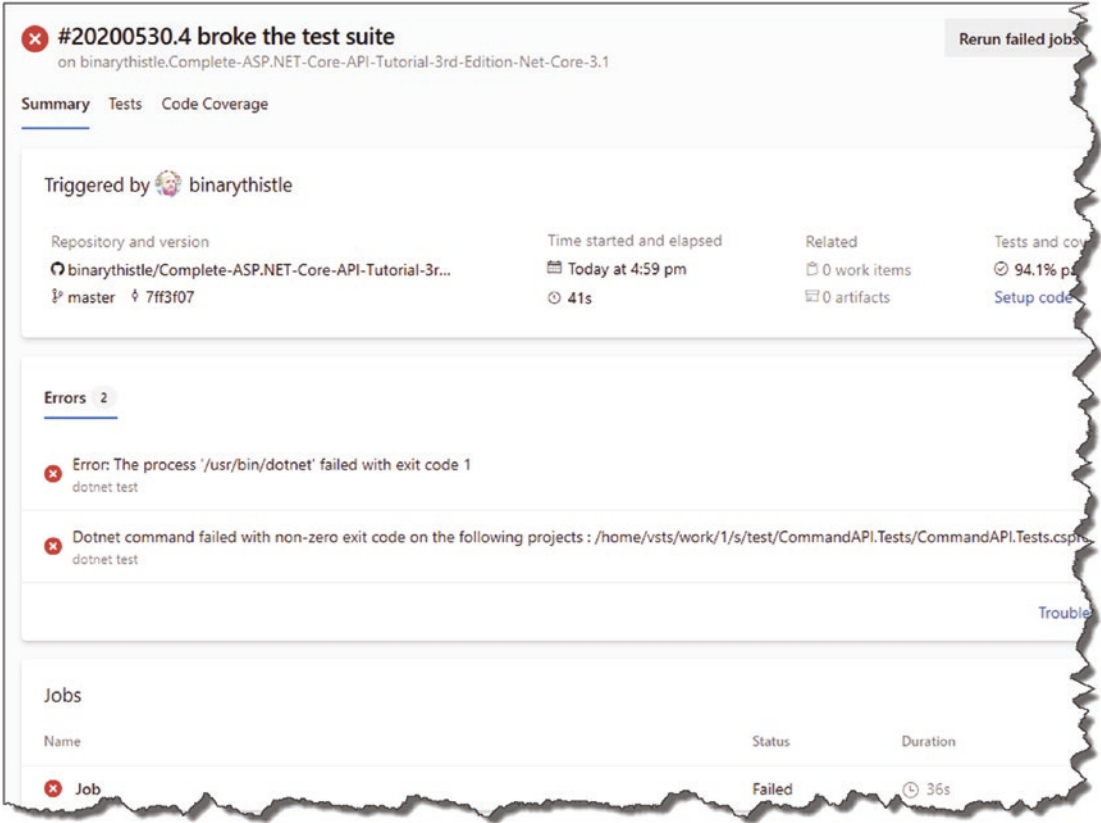


Figure 12-47. Detail of failures

And then drilling further in to the dotnet test step and going to the test results dashboard (see Figure 12-48).

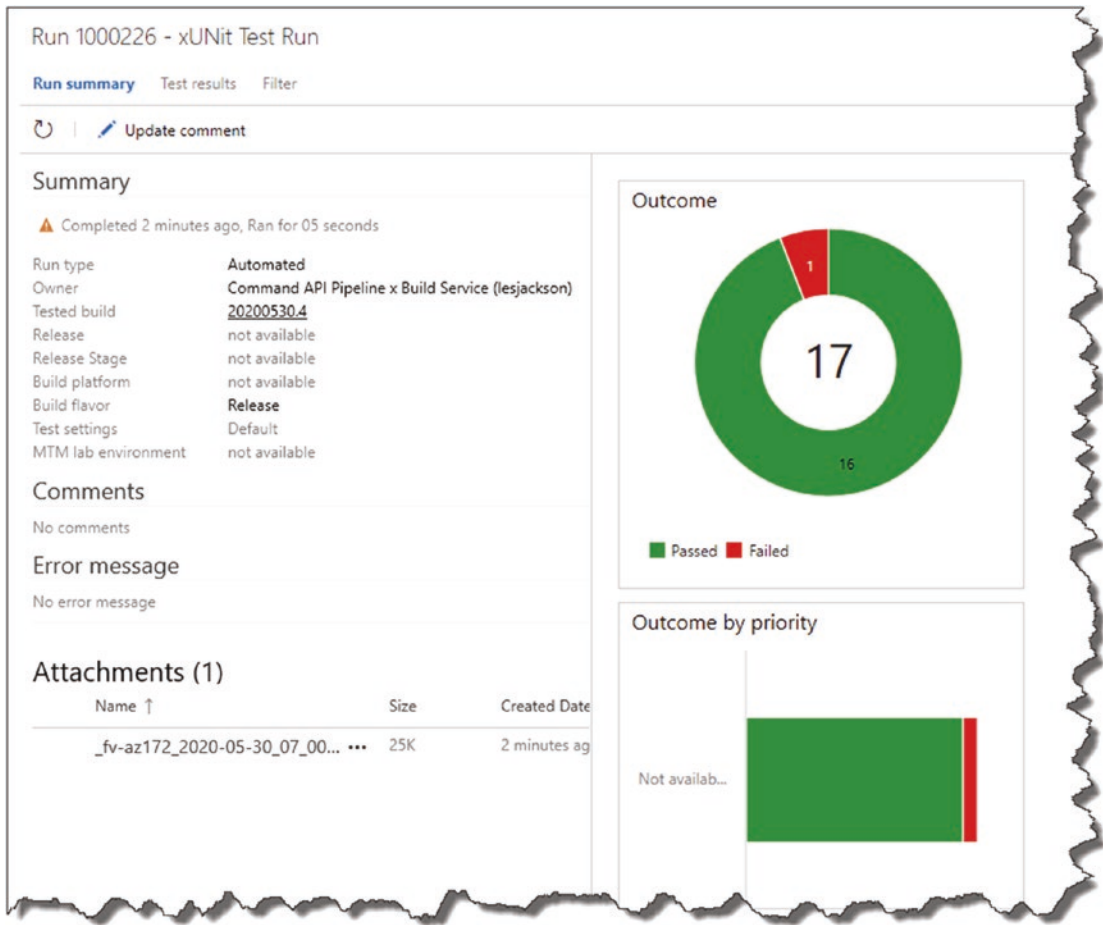


Figure 12-48. Dashboard represents the failure

## 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.

So conversely, does this mean that if all tests pass, 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!



**Figure 12-49.** Make sure you fix your pipeline before continuing

## 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.



**Figure 12-50.** Revisit our pipeline

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, your file should look like this, with the new code highlighted (again watch those spaces – the VS Code plugin we just installed should help you with this).

```

azure-pipelines.yml > [ ] steps > task
1  trigger:
2  - master
3
4  pool:
5  | vmImage: 'ubuntu-latest'
6
7  variables:
8  | buildConfiguration: 'Release'
9
10 steps:
11 - task: UseDotNet@2
12 - script: dotnet build --configuration $(buildConfiguration)
13   displayName: 'dotnet build $(buildConfiguration)'
14 - task: DotNetCoreCLI@2
15   displayName: 'dotnet test'
16   inputs:
17     command: test
18     projects: '**/*Tests/*.csproj'
19     testRunTitle: 'xUnit Test Run'
20
21 - task: DotNetCoreCLI@2
22   displayName: 'dotnet publish'
23   inputs:
24     command: publish
25     publishWebProjects: false
26     projects: 'src/CommandAPI/*.csproj'
27     arguments: '--configuration $(buildConfiguration) --output $(Build.ArtifactStagingDirectory)'
28
29 - task: PublishBuildArtifacts@1
30   displayName: 'publish artifacts'
31
32

```

**Figure 12-51.** Package and publish steps

The steps are explained in more detail in the Microsoft Documents,<sup>2</sup> but in short

- A dotnet publish command is issued for our **CommandAPI** project only.<sup>3</sup>
- The output of that is zipped.
- The zipped artifact is published.

<sup>2</sup><https://docs.microsoft.com/en-us/azure/devops/pipelines/ecosystems/dotnet-core?view=azure-devops&tabs=yaml>

<sup>3</sup>We don't want to publish our tests anywhere!

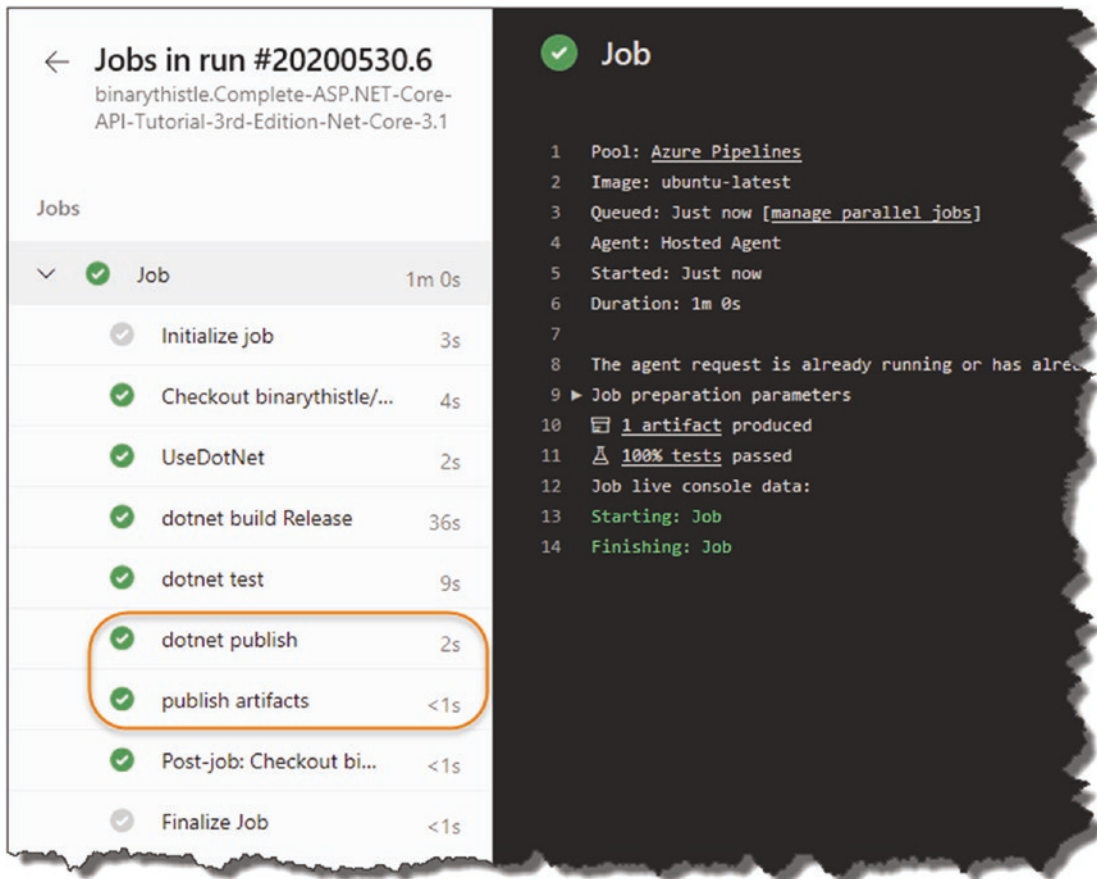


**Les' Personal Anecdote** Ensure that you put in the following line:


```
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 two additional task steps.



**Figure 12-52.** Steps shown in the running Job

 **Celebration Checkpoint** 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

- Setup a CI/CD pipeline on Azure DevOps
- Connected Azure DevOps to GitHub (and ensured CI triggers were enable)
- **Added:** Build, Test, and Packaging steps to our *azure-pipeline.yml* file

We are now almost ready to deploy to Azure!

## CHAPTER 13

# Deploying to Azure

## Chapter Summary

In this chapter we deploy our API onto Azure for use in the real world. On the way, we create the Azure resources we need and revisit 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 and database up and running in a “production-like” environment – which should be more than enough.

In simple terms, everything in Azure is a “resource,” for example, a database server, virtual machine, web app, etc. So, we need to create a few resources to house our app. There are different ways to create resources in Azure:

1. Create resources manually via the Azure Portal.
2. Create resources automatically via Azure Resource Manager Templates.
3. Create resources automatically using third-party tools, for example, Terraform.

In this chapter, we’ll be manually creating the resources we need as

- It’s simpler (in our case anyway, see next point).
- We only have a small number of resources.
- I think it’s the right approach to learning (our focus is still our API).

## Create Our API App

The first resource we are going to create is an API App; this unsurprisingly is where our API code will run. To do so, log-in to Azure (or if you don’t have an account, you’ll need to create one), and click “Create a resource”:

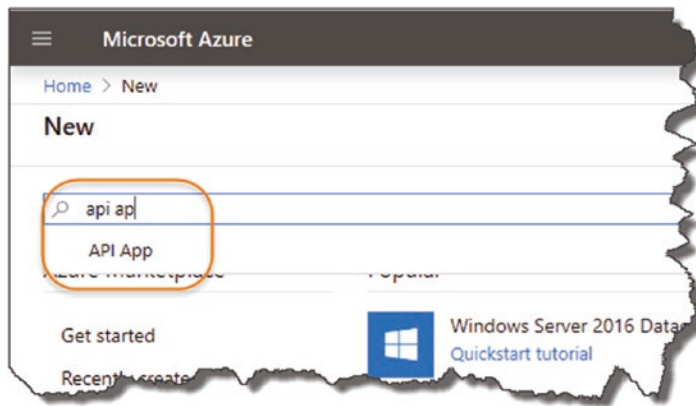


**Figure 13-1.** Create an Azure Resource

⚠ Again, I'll mention the point that the following screenshots 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 in Azure is not that difficult, so small UI changes should not stump someone as smart as yourself!

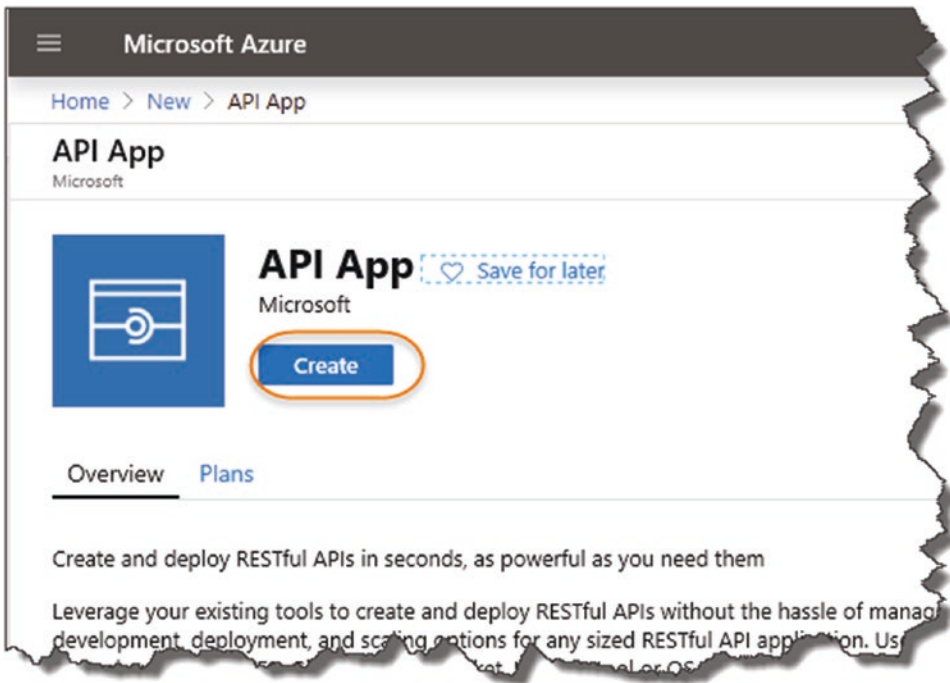
In the “search box” that appears in the new resource page, start to type “API App”; you will be presented with the API App resource type.



**Figure 13-2.** Search for API App

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



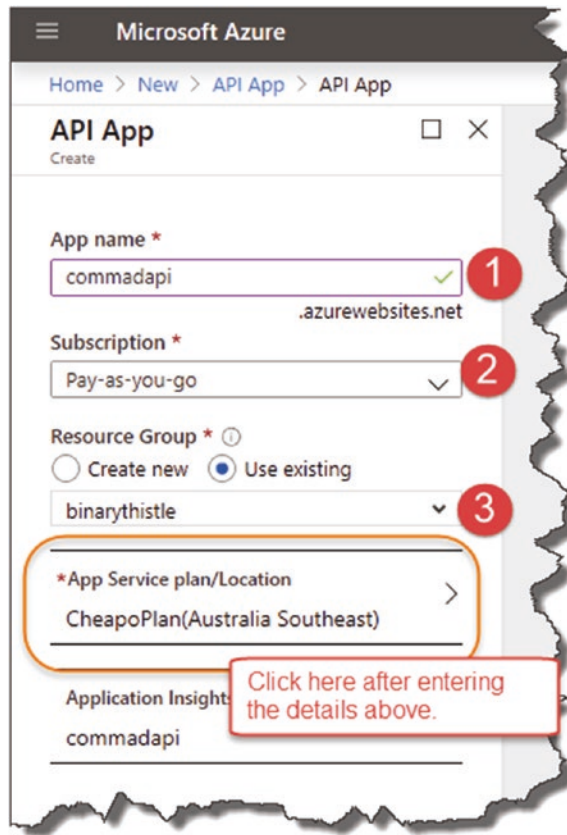


**Figure 13-3.** *Create the API App*

On the Next “page,” enter

1. A name for your API App.<sup>1</sup>
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”; if you don’t have an existing resource group, you’ll need to create one.

<sup>1</sup>This needs to be unique in Azure, so your name will be different to mine.



**Figure 13-4.** Configure your API App – make sure you configure a free plan!

**WAIT!** Before you click Create, click the App Service plan/location.



**Les' Personal Anecdote** The *API App* resource describes what you are getting; the *App Service Plan and Location* tells you how that API App will be delivered to you.

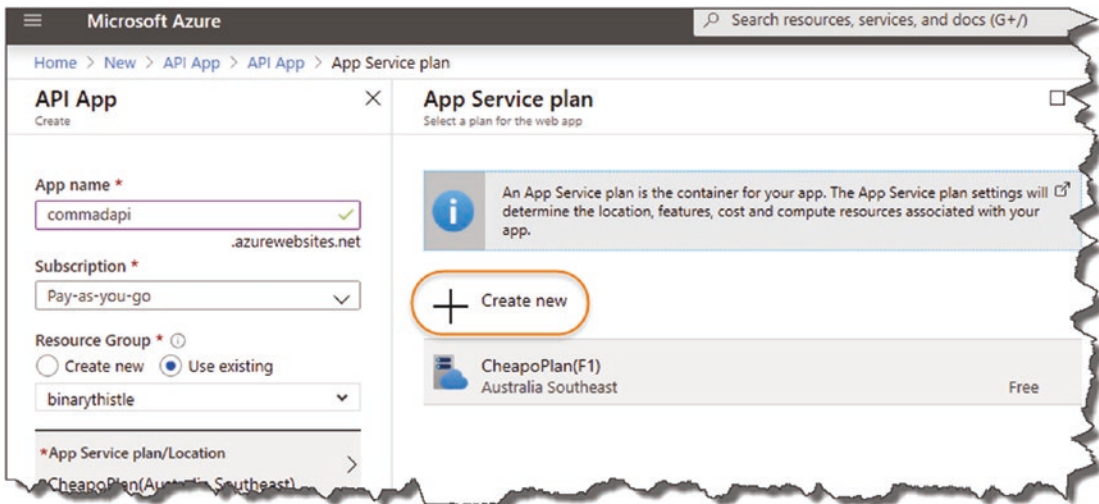
For example, do you want your API App

- Hosted in the United States, 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!)

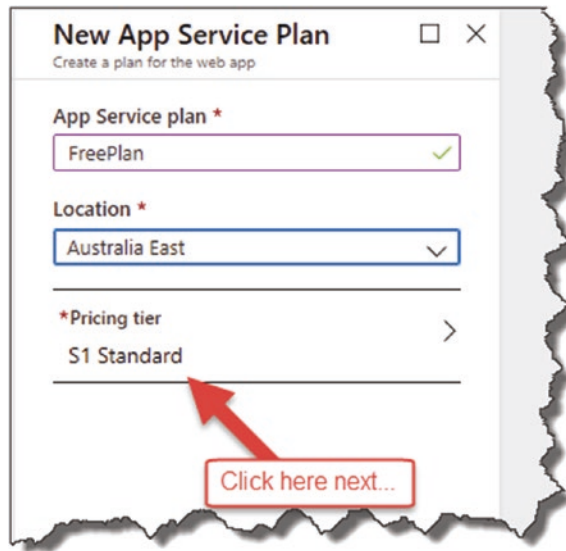
So be careful of the Service Plan you set up; I detail the free plan next.

After clicking the Service Plan, click "Create new."



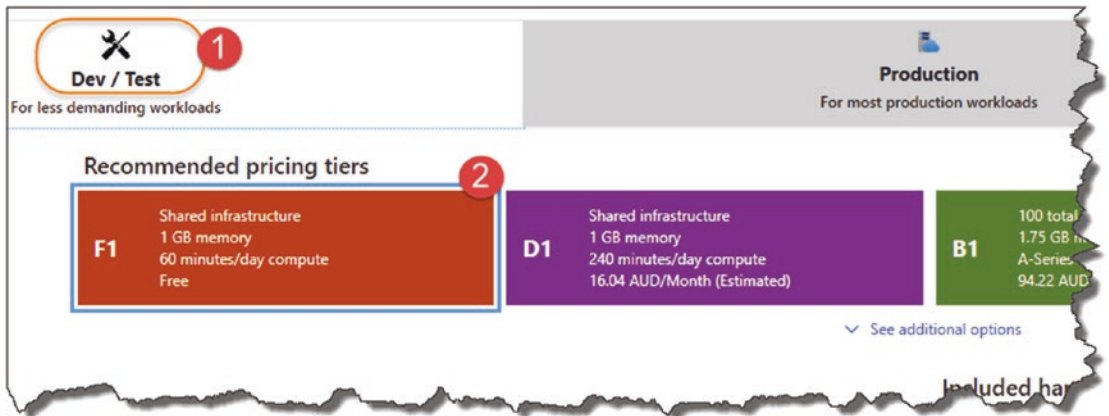
**Figure 13-5.** *Creating an App Service Plan*

On the "New App Service Plan" widget, enter an App Service Plan name, and pick your location, then click the Pricing Tier.



**Figure 13-6.** *The Pricing Tier*

After, click the *Pricing Tier*.

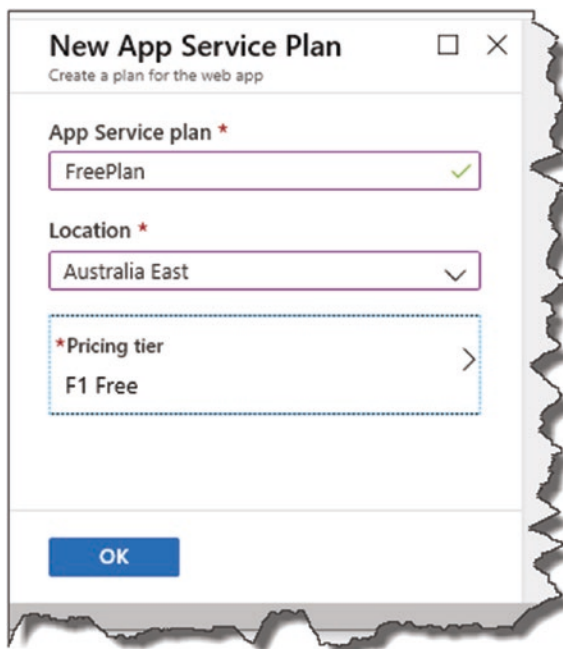


**Figure 13-7.** *Select the Free option*

1. Select the Dev/Test tab.
2. Select the “F1” Option (Shared infrastructure/60 minutes compute).
3. Click Apply.

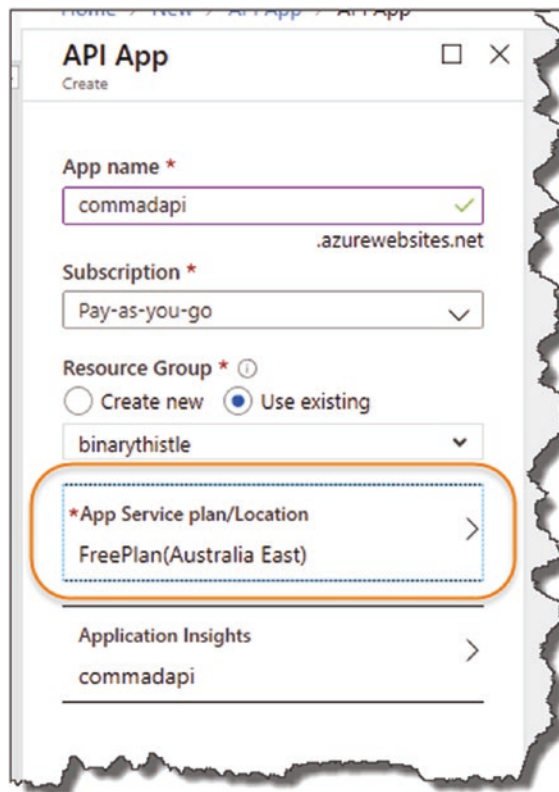
We have selected the cheapest tier with “Free Compute Minutes,” although please be aware that I cannot be held responsible for any charges on your Azure Account! (After I create and test a resource if I don’t need it – I “stop it” or delete it).

Then click OK.



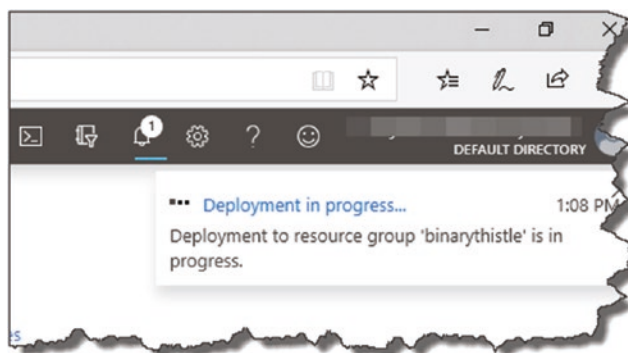
**Figure 13-8.** You're ready to go

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



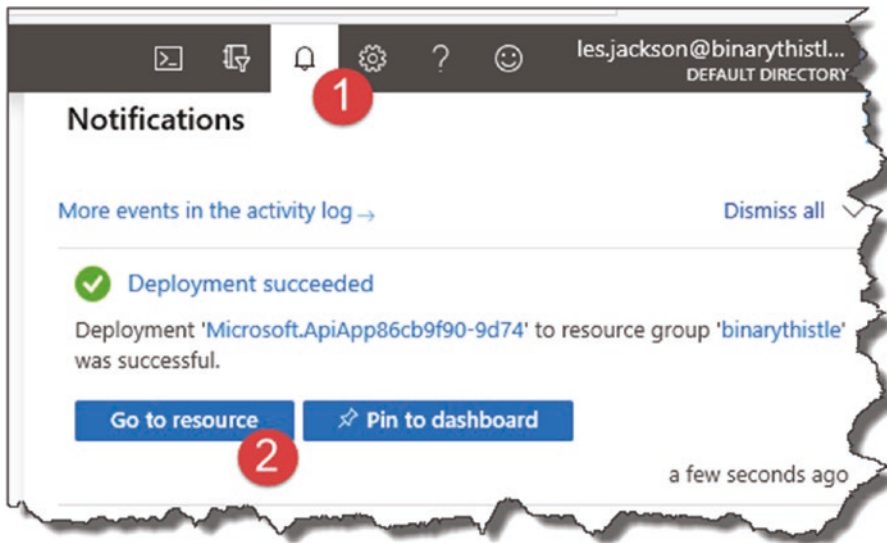
**Figure 13-9.** Free plan has been applied to the API APP

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



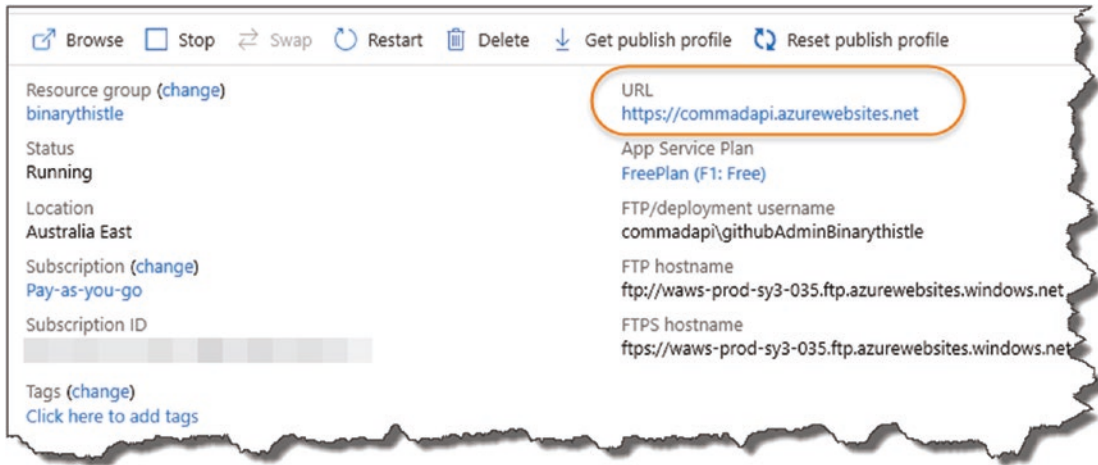
**Figure 13-10.** Deployment will take a few minutes...

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.



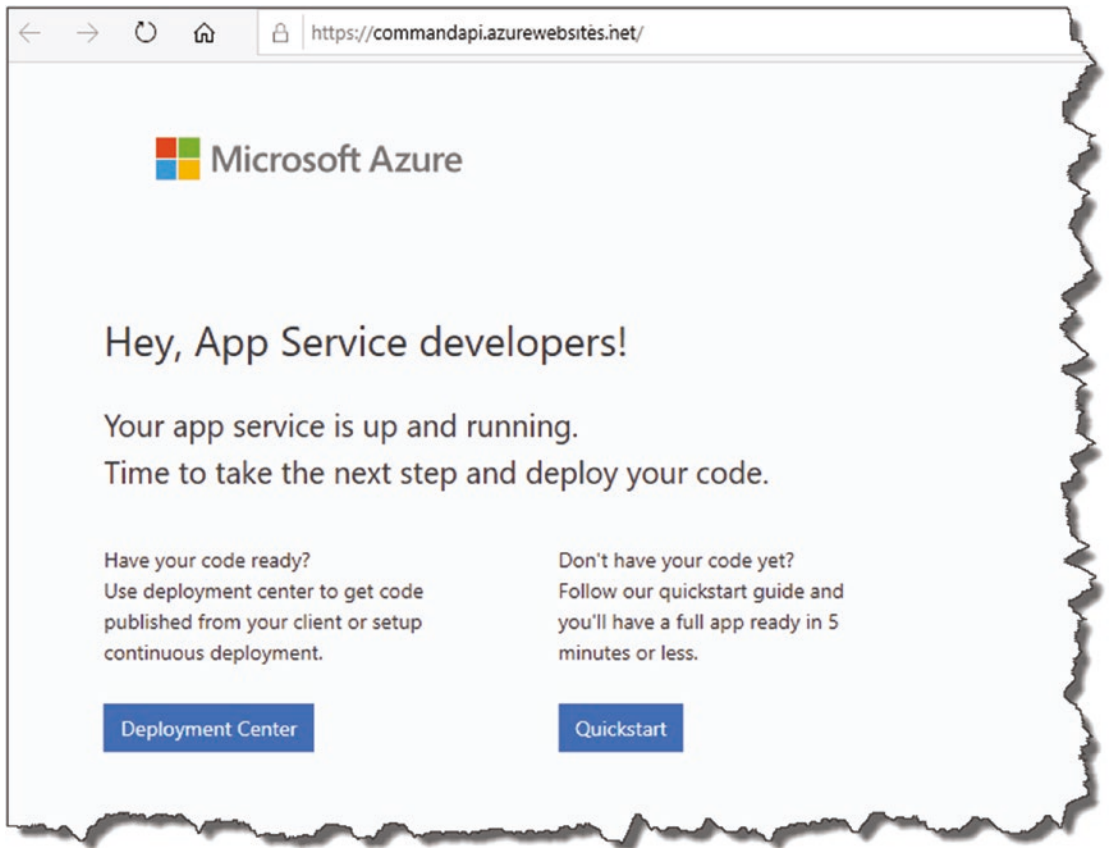
**Figure 13-11.** Notification of Resource Creation

Here you can see the resource was successfully created; now click “Go to resource.”



**Figure 13-12.** API App Overview including URI

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 the location URL, and it will take you to where the API App resides.



**Figure 13-13.** *Default public landing page*

As we have not deployed anything, you'll get a similar landing page as shown in Figure 13-13 (of course for reasons already mentioned, it may look a bit different, but that is of no consequence to us at this point).

---

**🎉 Celebration Checkpoint** You've just created your first Azure resource, one of the primary components of our production solution architecture!

---

## Create Our PostgreSQL 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).

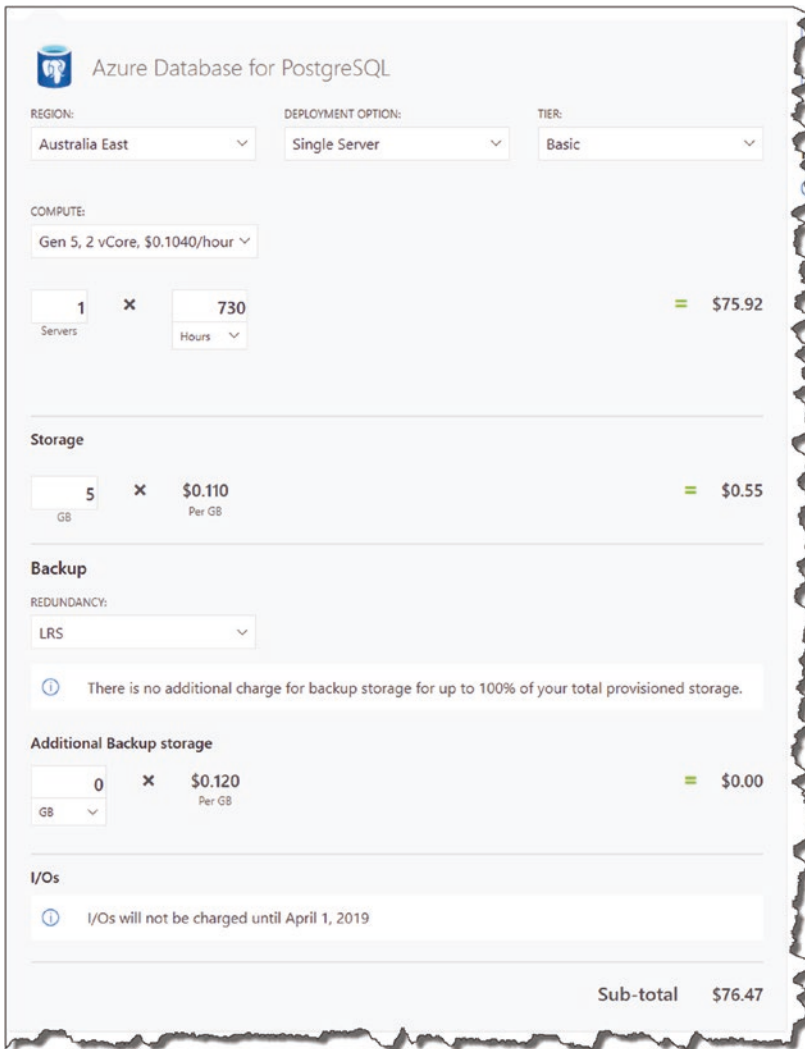


I've taken this approach primarily because the setup is so simple and the cost implications are low. To illustrate my point, compare the estimated costs for

- 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.



**Figure 13-14.** Cost estimate for Postgres Server

## Container Instance Pricing

Container Instances

REGION: Australia East OPERATING SYSTEM: Linux

Container groups: 1 Duration: 10 Seconds

Memory: 1 GB × 1 Container groups × 10 Seconds × \$0.0000015 Per GB-s = \$0.01

vCPU: 1 × 1 Container groups × 10 Seconds × \$0.0000135 Per vCPU-s = \$0.01

Sub-total \$0.01

**Figure 13-15.** 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 is a purpose-built resource that's designed to work as a database, whereas the container option I'm taking is in no way optimized for Production performance!

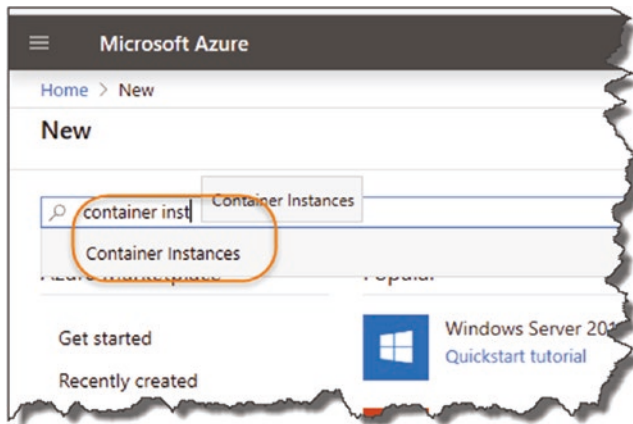
---

**⚠ Warning!** If you restart the PostgreSQL container instance that we create in the next section, it essentially resets, and **you will lose your configuration and data** relating to it – just something to bear in mind.

From a learning (and cost!) perspective, I still think this option is acceptable. If, however, you are moving to a "real" Production environment, then you'll really need to look at something a little more fit for purpose.

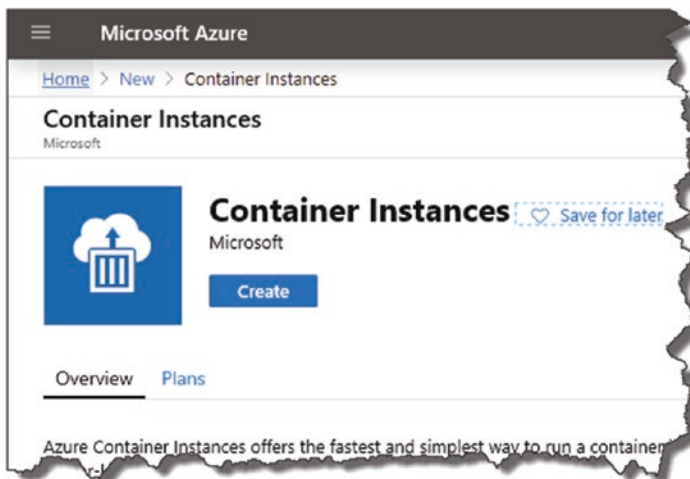
---

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



**Figure 13-16.** Search for Container Instances

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



**Figure 13-17.** Overview of Container Instances

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*.

## Create container instance

[basics](#) [networking](#) [advanced](#) [tags](#) [review + create](#)

Azure Container Instances (ACI) allows you to quickly and easily run containers on Azure without managing servers or having to learn new tools. ACI offers per-second billing to minimize the cost of running containers on the cloud.  
[Learn more about Azure Container Instances](#)

### Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* ⓘ  1

Resource group \* ⓘ  2  
[Create new](#)

### Container details

Container name \* ⓘ  3 ✓

Region \* ⓘ  4

Image source \* ⓘ  Quickstart images  
 Azure Container Registry 5  
 Docker Hub or other registry

Image type \* ⓘ  Public  Private 6

Image \* ⓘ  7 ✓  
 ⓘ If not specified, Docker Hub will be used for the container registry and the latest version of the image will be pulled.

OS type \*  Linux  Windows 8  
 ⓘ This selection must match the OS of the image chosen above.

Size \* ⓘ  9  
[Change size](#)

[Previous](#) [Next: Networking](#)

**Figure 13-18.** Configure your Container Instance

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 Source:** Select Docker Hub (this is where we'll get our postgres image).
6. **Image Type:** Select Public (the postgres image we use in the next step is publicly available on Docker Hub).
7. **Image Name:** As mentioned earlier, this needs to be the exact name of the image on Docker Hub, so in this case *postgres*.
8. **OS Type:** Select Linux.
9. **Size:** Leave these as the defaults.

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



**Figure 13-19.** Networking

And supply the following details in the Networking Tab.

Basics **Networking** Advanced Tags Review + create

Choose between three networking options for your container instance:

- **'Public'** will create a public IP address for your container instance.
- **'Private'** will allow you to choose a new or existing virtual network for your container instance. This is not yet available for Windows containers.
- **'None'** will not create either a public IP or virtual network. You will still be able to access your container logs using the command line.

Networking type  Public  Private  None **1**

DNS name label ⓘ  **2**  
.australiaeast.azurecontainer.io

Ports ⓘ

Ports	Ports protocol
80	TCP
<input type="text" value="5432"/>	<input type="text" value="TCP"/>
<input type="text"/>	<input type="text"/>

**3**

**Figure 13-20.** Networking configuration

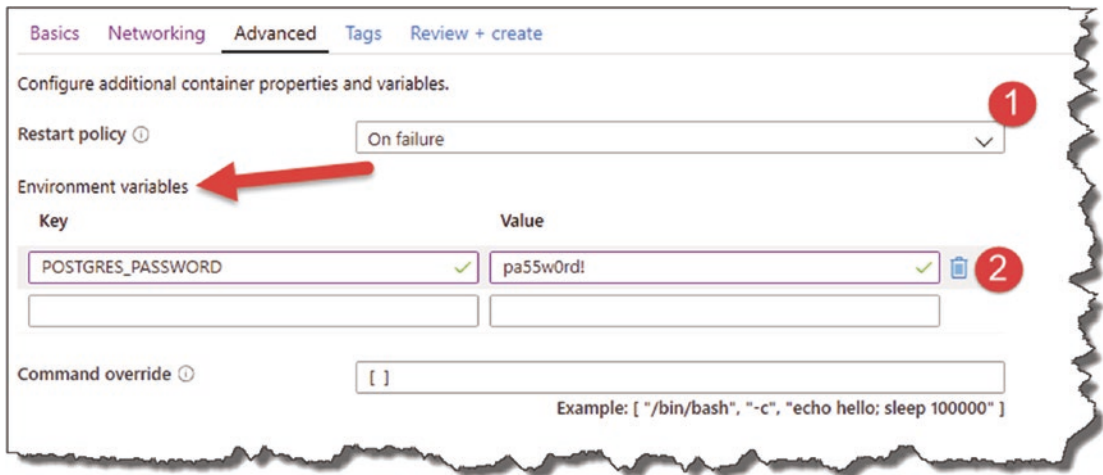
1. Select “Public” for a public IP Address (note this can change if the container restarts).
2. Add a DNS name label as the IP Address can change if the container restarts.
3. Add the standard 5432 TCP port for PostgreSQL.

When you’re happy, click “Next: Advanced >.”



**Figure 13-21.** Onto Advanced Settings

And enter the following details on the Advanced tab.



**Figure 13-22.** Setup Environment variables

1. Set the Restart Policy to “On Failure.”
2. Create an “environment variable” for the Postgres password for the default database; the Key you should use for this is

POSTGRES\_PASSWORD

The choice of password (the value) is up to you. In my case, I used  
pa55w0rd!

---

**⚠ Warning!** As you can see our PostgreSQL password is in plain text; again this is not a production-suitable solution. We are using it for (cheap!) testing purposes only.

---

If you pop back to Chapter 7 where we set up an instance of PostgreSQL locally using Docker Desktop, there is a bit more of a discussion on these settings – so we don’t need to go over old ground here. Just a point of note, however, the environment variable for the PostgreSQL DB password (POSTGRES\_PASSWORD) is exactly the same as the one we used when setting up our local Docker instance.

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 > Download a template

**Figure 13-23.** Validation Passed



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: by clicking All resources, you can see everything we have created.

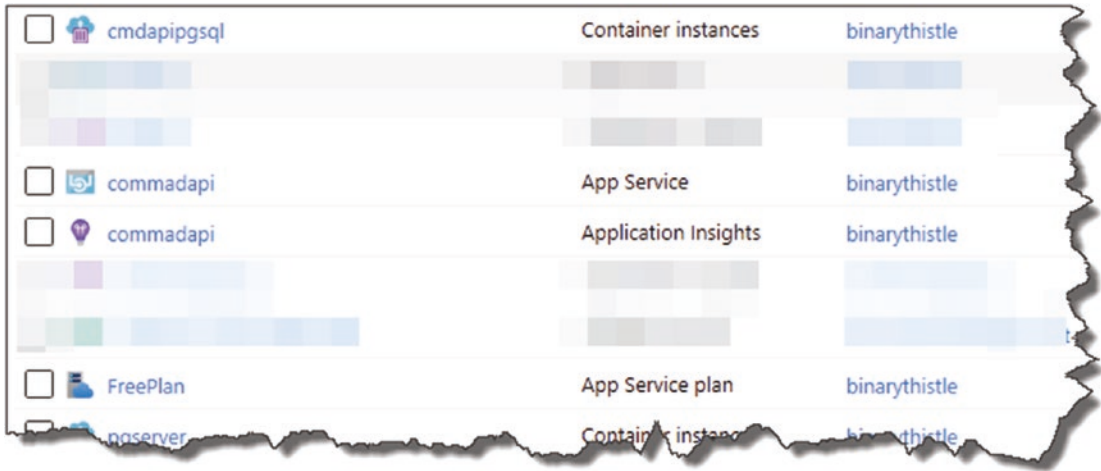


Figure 13-24. Resources up and running

## Connect and 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.

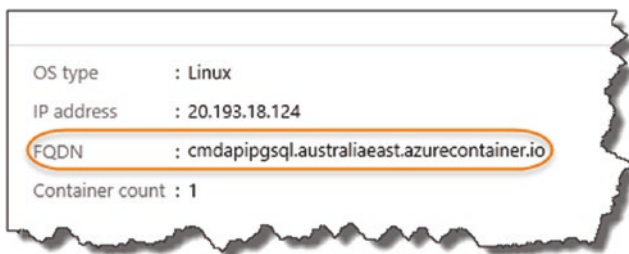
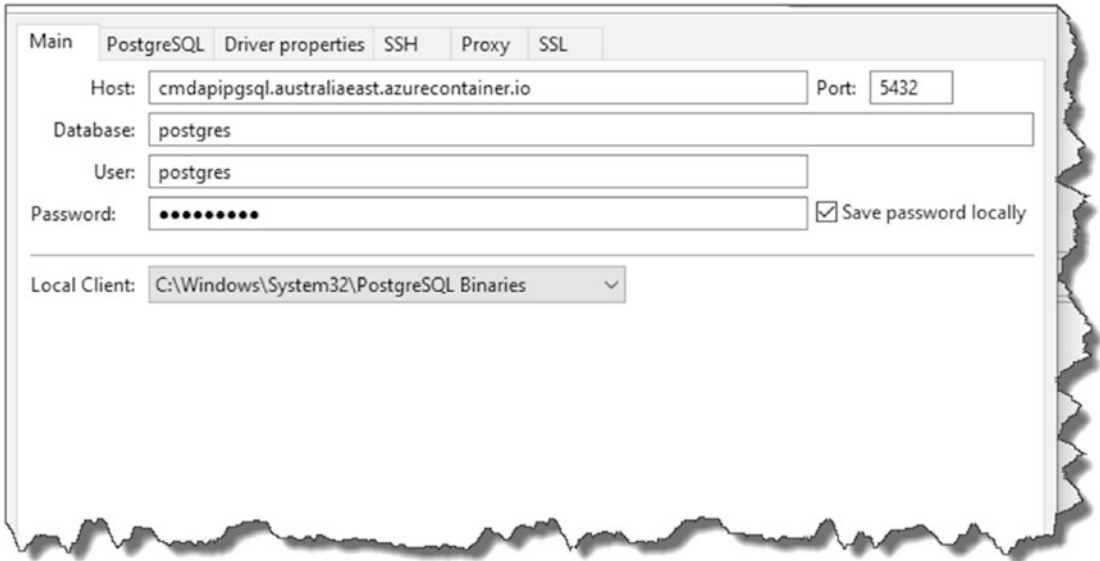


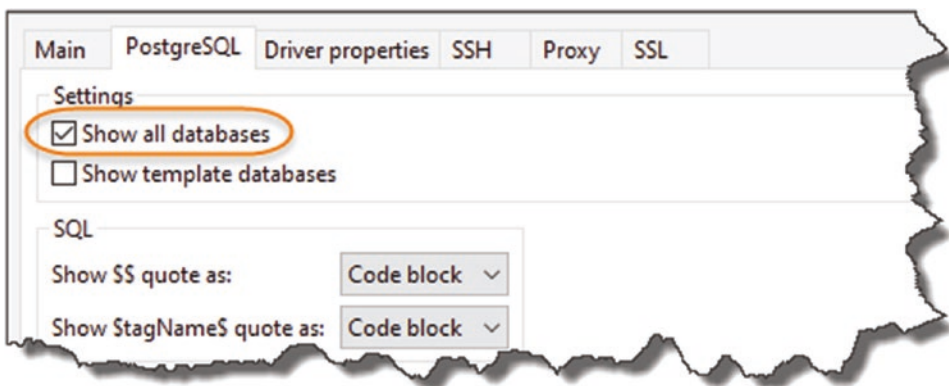
Figure 13-25. Location of the API

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 postgres user (depending on what you set in the container instance environment variables).



**Figure 13-26.** *Connect to the Azure instance*

Remember to tick “Show all databases” on the PostgreSQL tab.



**Figure 13-27.** *Ensure Show all Databases is ticked*

You can test the connection or press Finish to setup our connection to our Azure-based instance.

Again, we'll just repeat the user creation steps in Chapter 7:

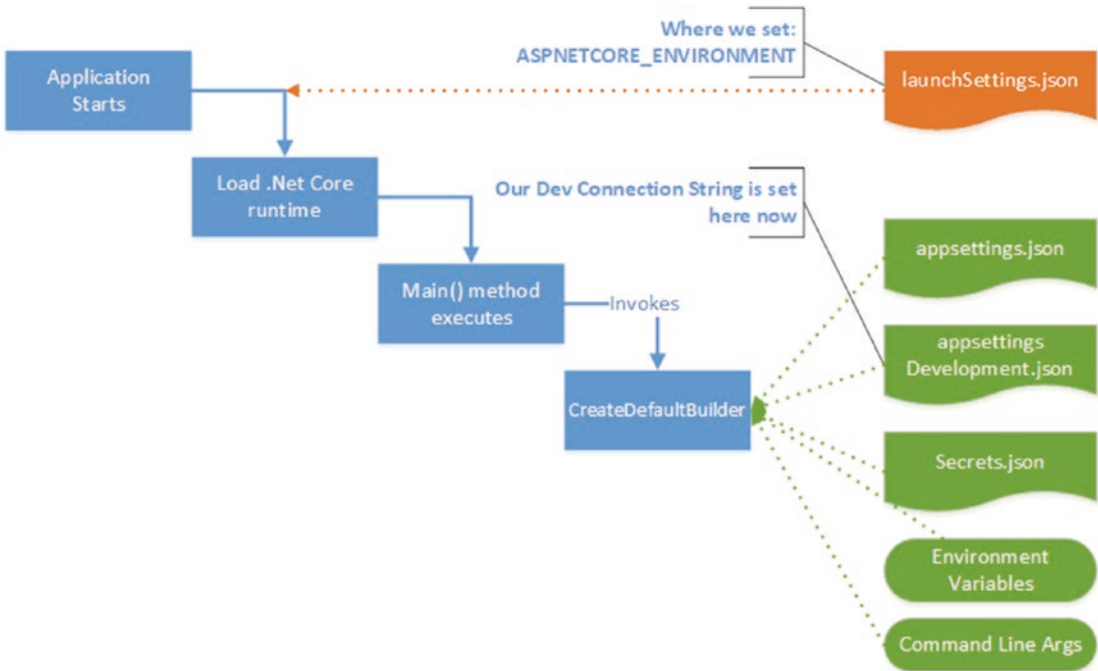
- Open a New SQL Editor Window.
- Enter and run the following SQL (you can change the password obviously!):

```
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 8, but it's worth doing a bit of a review.



**Figure 13-28.** Revisit configuration

- We set our environment in *launchSettings.json* (in the ASPNETCORE\_ENVIRONMENT variable).
- Our Connection Strings can sit in *appsettings.json* or the environment specific variants of that file, for example, *appsettings.Development.json*. This is where our Development connection string sits.
- “Secret” information, such as Database log-in 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 *appSettings.Development.json*).
- Our User ID, stored in a User Secret called UserID.
- Our Password, stored in a User Secret called Password.

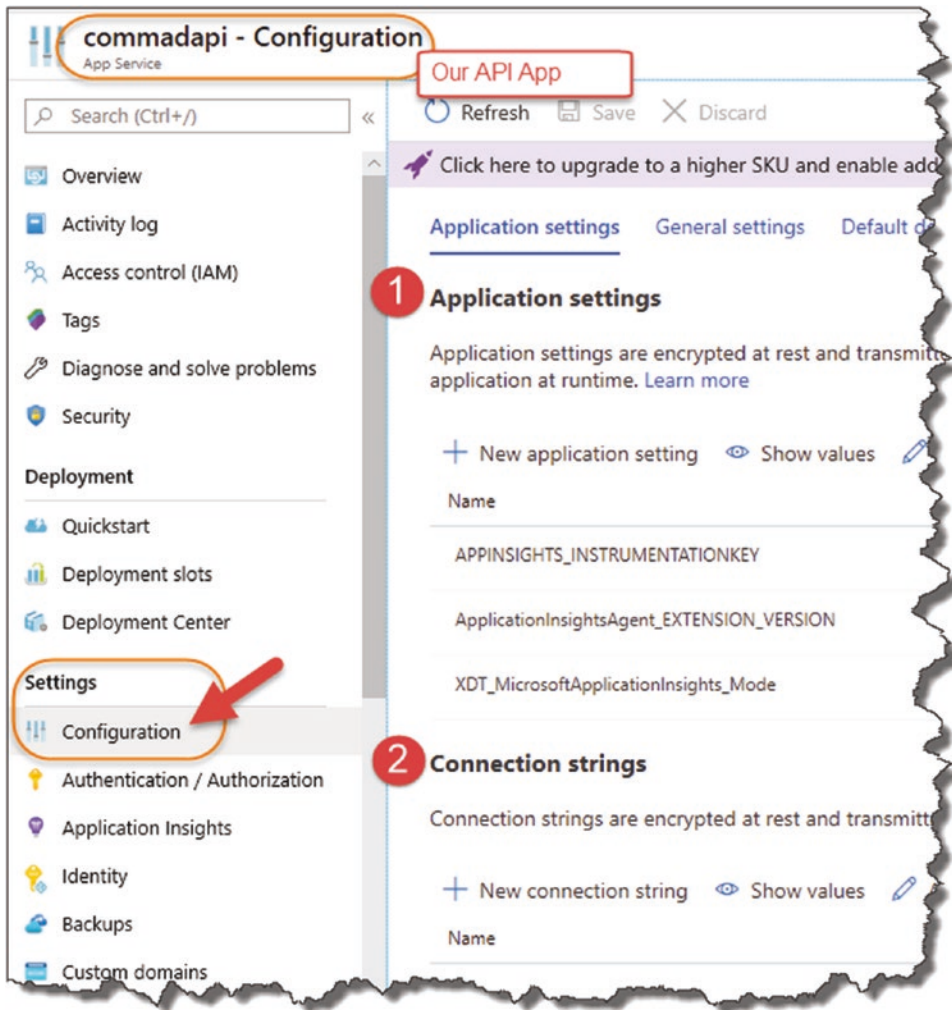
## Setting Up Config in Azure

When you deploy a .NET Core app as an Azure API App, it sits on top of a configuration layer that we access via the .NET Core Configuration API in *exactly the same way* as we have done to date with our local Development environment. 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 as shown here.



**Figure 13-29.** Application settings and connection strings in the API app

You'll see there are two 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. We simply substitute that for the PostgreSQL Container Instance FQDN that you should have set aside from the section earlier. So, I now have the following (yours will look different depending on your container instance name and location of course):

```
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 – I cannot stress that enough!**).
2. The connection string we generated earlier (note we’ll be configuring our User ID and Password separately below).
3. Set the type to *Custom*.

**Add/Edit connection string**

Name 1

Value 2

Type 3

Deployment slot setting

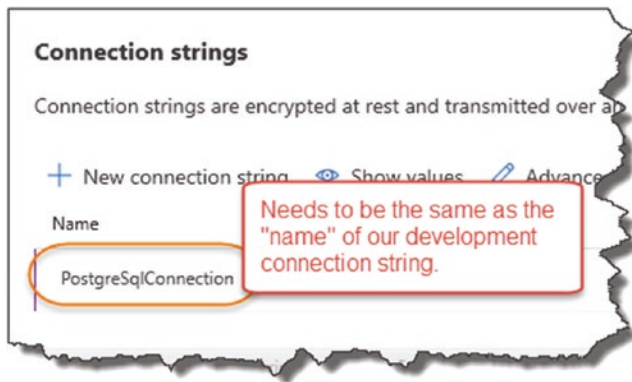
**Figure 13-30.** Add the connection string; be careful to name it correctly

---

**⚠ Warning!** You do have the option of “PostgreSQL” for the connection string type – however, I’ve had significant issues trying to use this – so use it at your peril!

---

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



*Figure 13-31. Again, ensure it's named correctly*

## Configure Our DB User Credentials

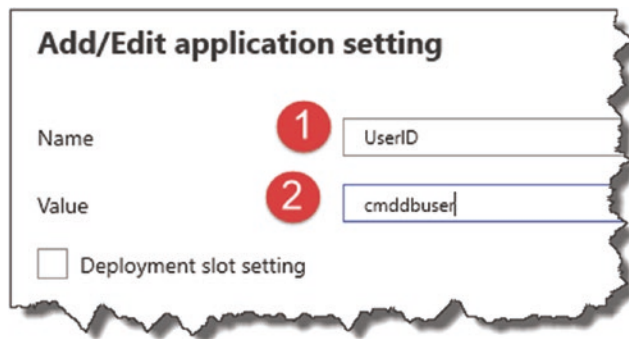
We're going to add our Production User ID and Password configuration items in a very similar way, except this time, we'll add these items to the *Application Settings* section of our API App Configuration. To add our User ID, click + *New application setting*. In the resulting form, enter

1. **Name:** This should be the same as our *User Secret* name for User ID.
2. **Value:** This is the user account you set up on the PostgreSQL Container Instance earlier.

For example, if you've been following the tutorial step by step these should be

- **Name:** UserID
- **Value:** cmddbuser

So, add them as an Application Setting as follows.



**Add/Edit application setting**

Name 1 UserID

Value 2 cmddbuser

Deployment slot setting

**Figure 13-32.** Create User ID Application Setting

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

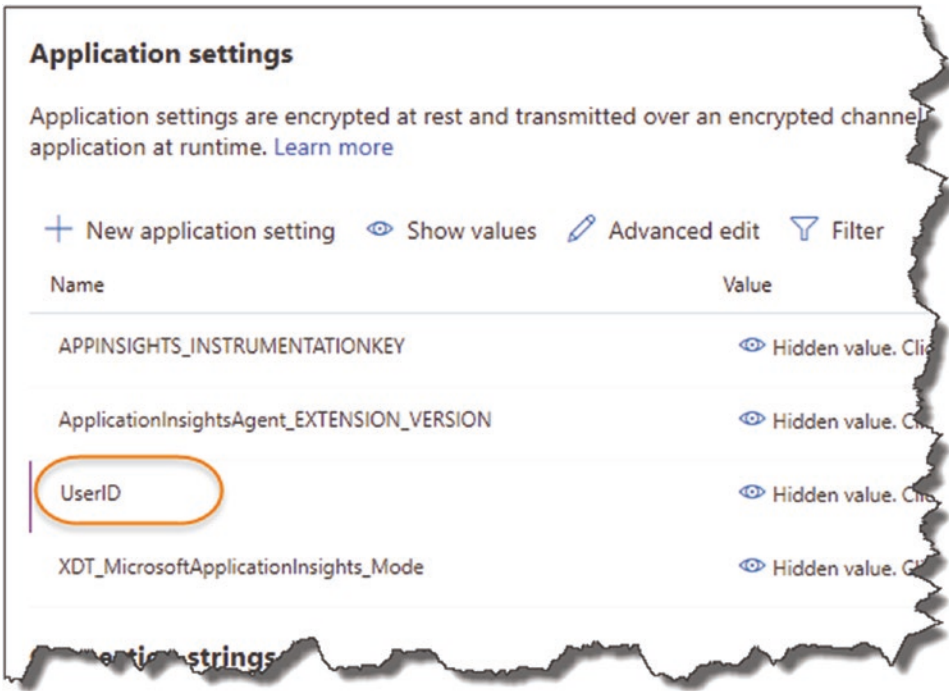
```
public void ConfigureServices(IServiceCollection services)
{
    var builder = new NpgsqlConnectionStringBuilder();
    builder.ConnectionString = Configuration.GetConnectionString("PostgreSqlConnection");
    builder.Username = Configuration["UserID"];
    builder.Password = Configuration["Password"];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

    services.AddControllers();
}
```

**Figure 13-33.** Make sure you name it correctly

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





**Figure 13-34.** UserID added to Application Settings

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

**⚠ Warning!** Storing passwords in Application Settings possibly isn't the best location for them, one reason being that you can see what they are in plain text. Even though Azure is "secured," that is, only authorized users will have access to it – plain text passwords are just generally not a great idea.

In a real production environment, you'd want to opt for something like Azure Key Vault or a third-party product such a [Vault](https://www.vaultproject.io/).<sup>2</sup> I feel that detailing that here would just be taking us too far out the way of what we want to achieve today.

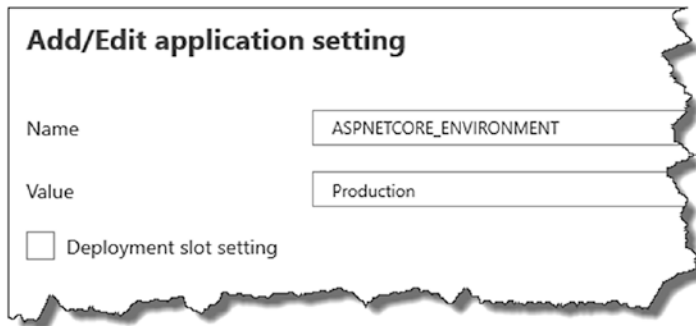
<sup>2</sup>[www.vaultproject.io/](https://www.vaultproject.io/)

## Configure Our Environment

Finally, we want to set our runtime environment to “Production”; we do this simply by adding another Application setting as follows:

- **Name:** ASPNETCORE\_ENVIRONMENT
- **Value:** Production

See Figure 13-35.



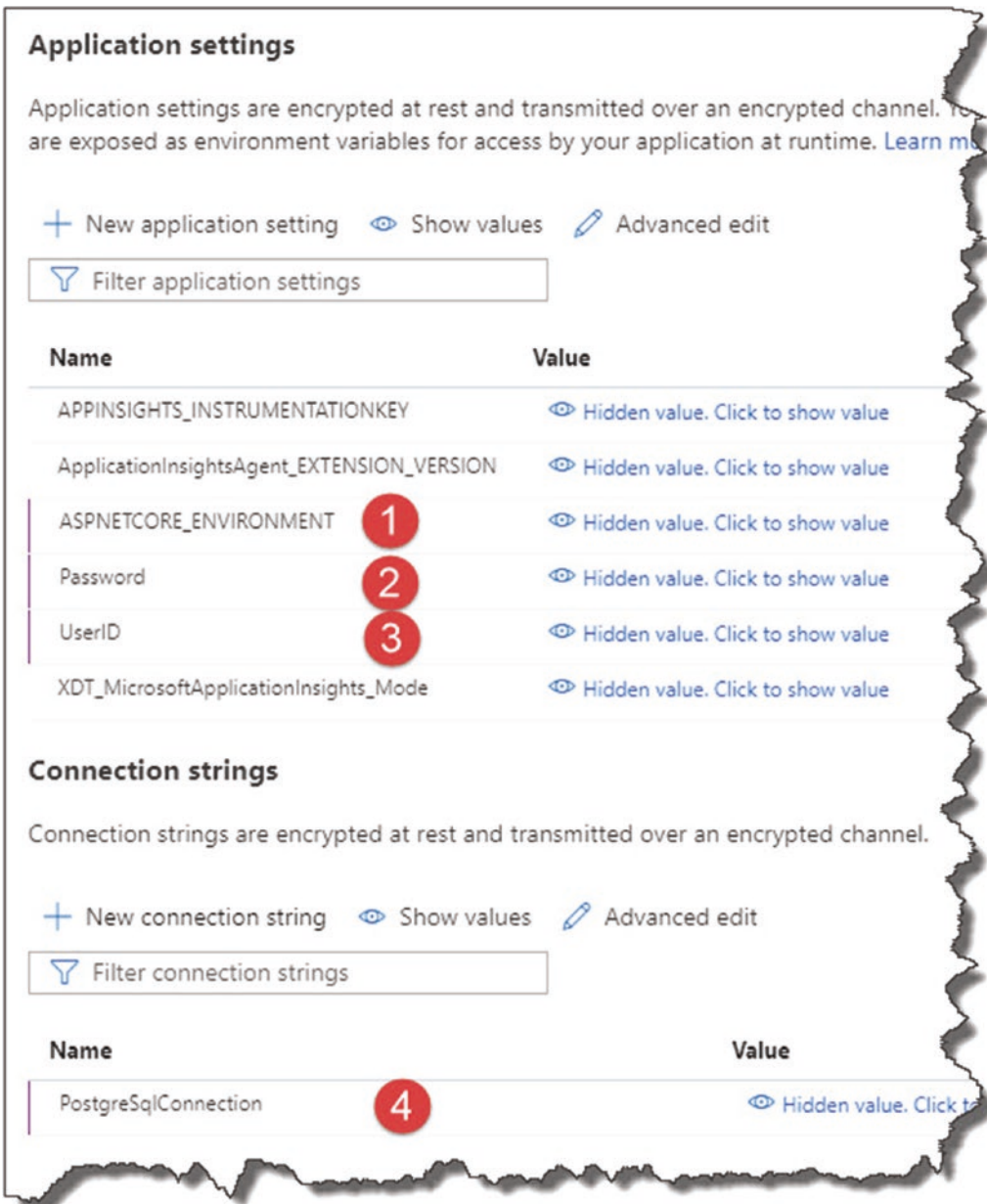
The image shows a screenshot of the 'Add/Edit application setting' dialog box. The dialog has a title bar with the text 'Add/Edit application setting'. Below the title bar, there are two text input fields. The first field is labeled 'Name' and contains the text 'ASPNETCORE\_ENVIRONMENT'. The second field is labeled 'Value' and contains the text 'Production'. Below these fields, there is a checkbox labeled 'Deployment slot setting' which is currently unchecked.

**Figure 13-35.** *Specifying our environment*

Click OK and you should now have added four production configuration settings:

1. **Application settings:** ASPNETCORE\_ENVIRONMENT
2. **Application settings:** Password
3. **Application settings:** UserID
4. **Connection string:** PostgreSqlConnection

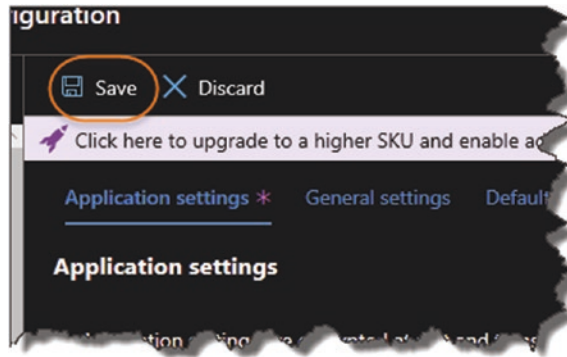
I've shown my setup as it appears in Azure in Figure 13-36.



**Figure 13-36.** Newly created Application Settings

Again, I know I keep repeating myself, but you need to make sure the *Names* of these configuration items are the same as their Development counterparts, as that is what our application is expecting – please double-check these! The values of these items I have to leave up to you to get correct!

**⚠ Warning!** Every time you make a configuration change, you need to save it - see Figure 13-37.



**Figure 13-37.** *Make sure you save!*

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

**🎉 Celebration Checkpoint** 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.



**Figure 13-38.** *The pipeline*

A quick recap on our CI/CD Pipeline so far

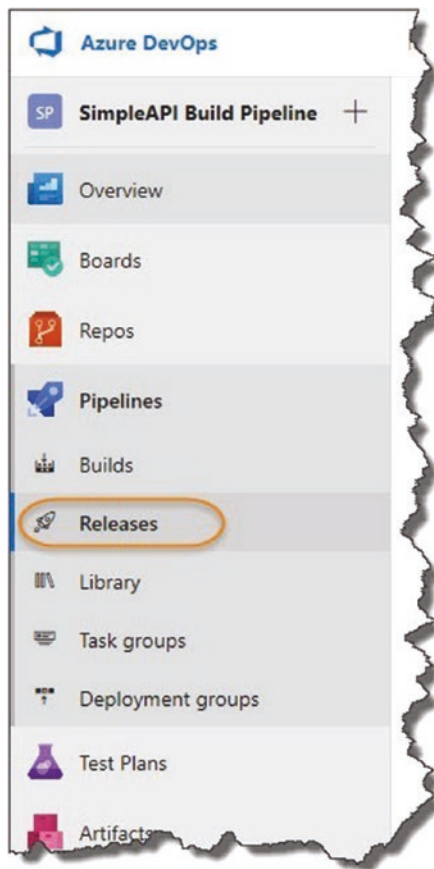
- We created what Azure DevOps calls a *Build Pipeline* that does the following:
  - Builds our projects

- 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 and releases and deploys it to Azure. So basically, our full CI/CD Pipeline = Azure DevOps **Build** Pipeline + Azure DevOps **Release** Pipeline.

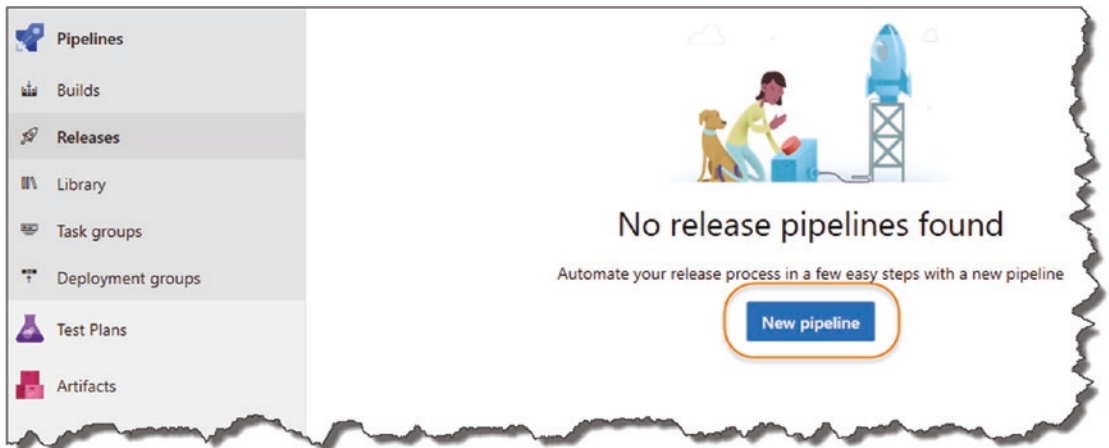
## Creating Our Azure DevOps Release Pipeline

Back in Azure DevOps, click Pipelines ► Releases.



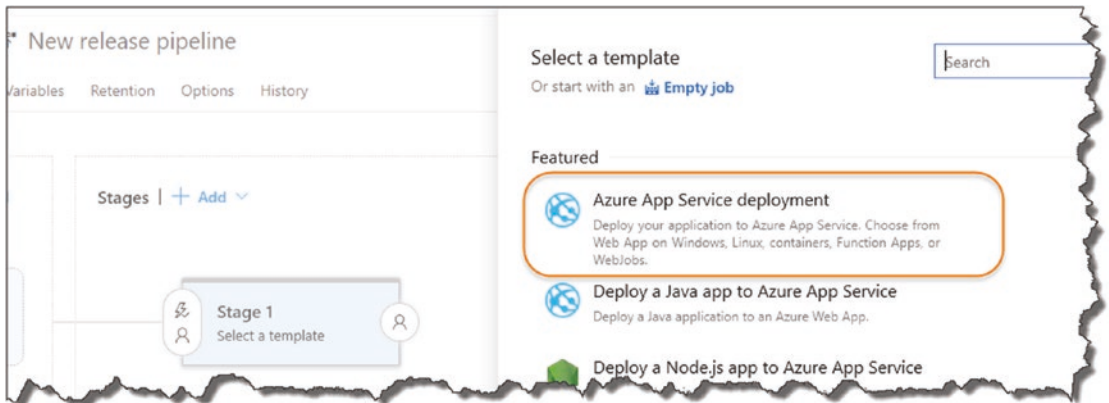
**Figure 13-39.** Release Pipeline

The click New Pipeline.



**Figure 13-40.** Create a new Release Pipeline

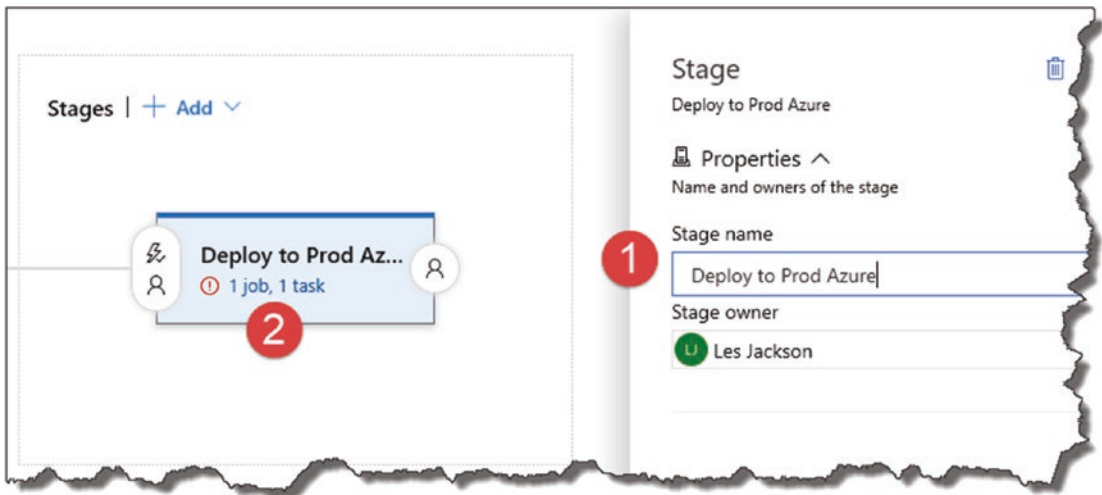
On the next screen, select and *Apply* the *Azure App Service deployment* template.



**Figure 13-41.** Select Azure App Service deployment

In the “Stage” widget

1. Change the stage name to “Deploy API to Prod Azure” (or whatever you like so long as it’s meaningful).
2. Click the Job/Task link in the designer.



**Figure 13-42.** Name the stage and fix up the task errors

Here, we need to

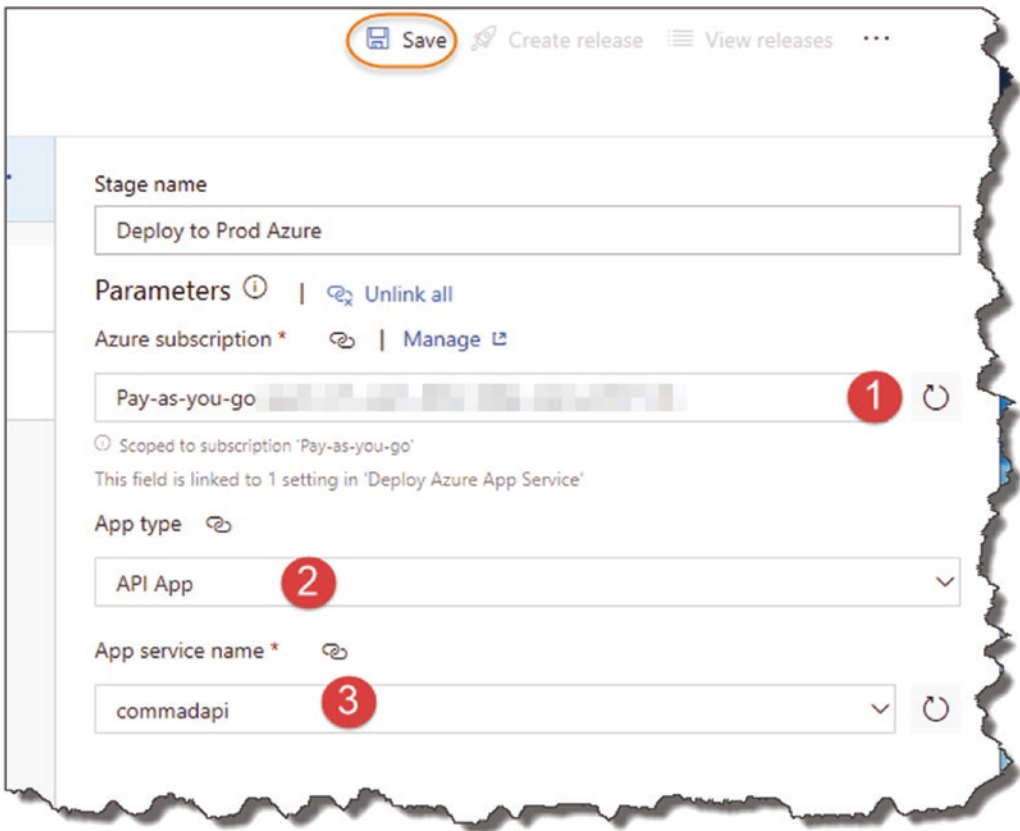
1. Select our Azure subscription (you will need to **“authorize”** Azure DevOps to use Azure).

---

**⚠ Warning!** If you’ve got an active pop-up blocker, this can cause you some issues here as the authentication window needs to “pop up.” Depending on your setup, you’ll need to allow pop-ups for this site in order to cleanly authenticate Azure DevOps to use Azure.

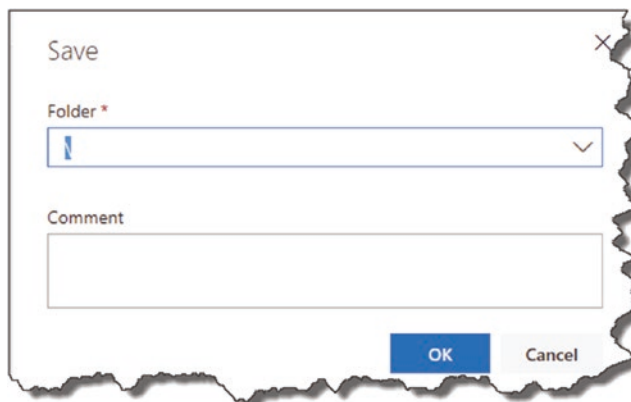
---

2. App Type (remember this is an API App)
3. App Service Name (all of your API Apps will be retrieved from Azure – select the one you created earlier)



**Figure 13-43.** Fix up the errors and remember to save!

Don't forget to *Save*. When you do, you'll be presented with the following.

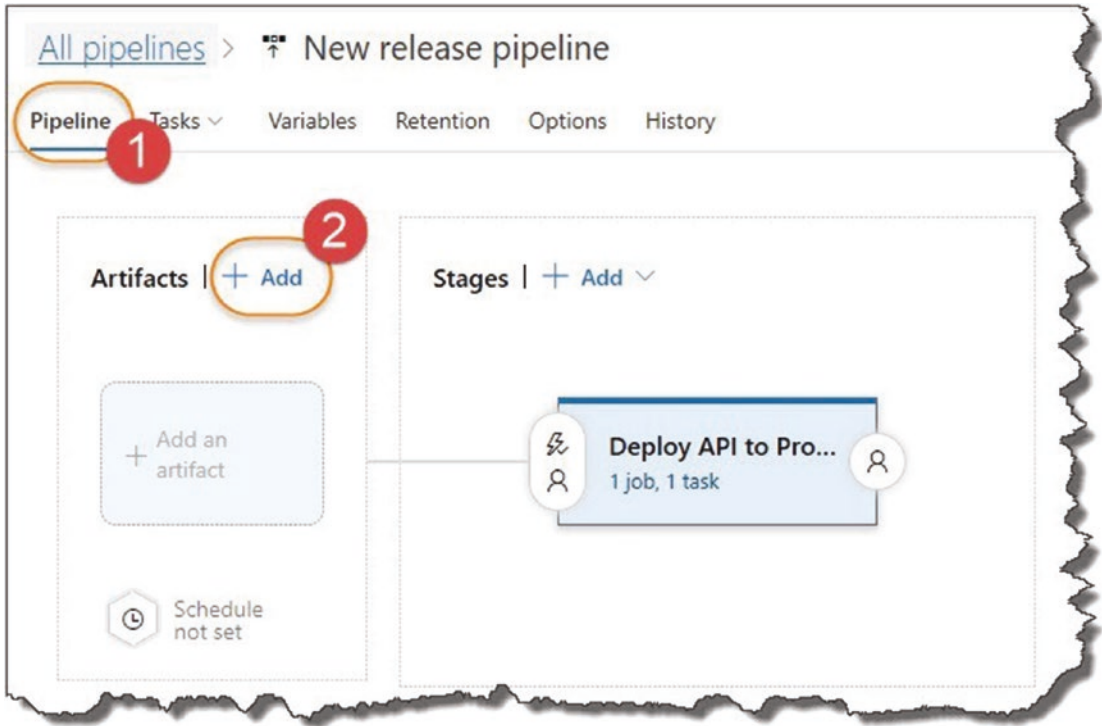


**Figure 13-44.** Add a comment if you need to



Just click OK.

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



**Figure 13-45.** Adding an artifact for deployment

Here, you will need to provide

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

**Add an artifact**

Source type

Build
  Azure Repos ...
  GitHub
  TFVC

5 more artifact types ▾

Project \* ⓘ

Command API Pipeline 1

Source (build pipeline) \* ⓘ

binarythistle.Complete-ASP.NET-Core-API-Tutorial-2nd-Edition-Net-Core-3.1 2

Default version \* ⓘ

Latest 3

Source alias \* ⓘ

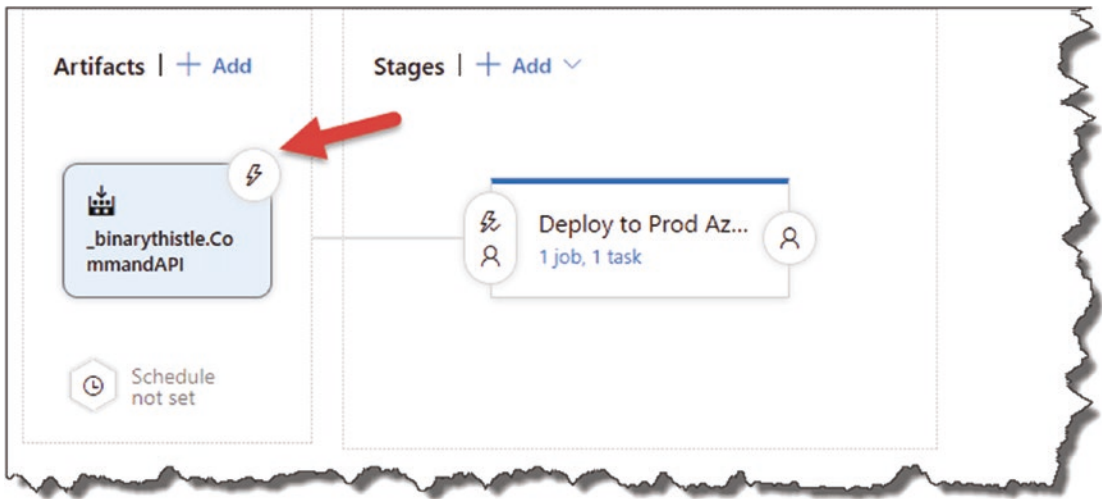
\_binarythistle.Complete-ASP.NET-Core-API-Tutorial-2nd-Edition-Net-Core-3.1

ⓘ The artifacts published by each version will be available for deployment in release pipelines. The last successful build of **binarythistle.Complete-ASP.NET-Core-API-Tutorial-2nd-Edition-Net-Core-3.1** published the following artifacts: *drop*.

**Add**

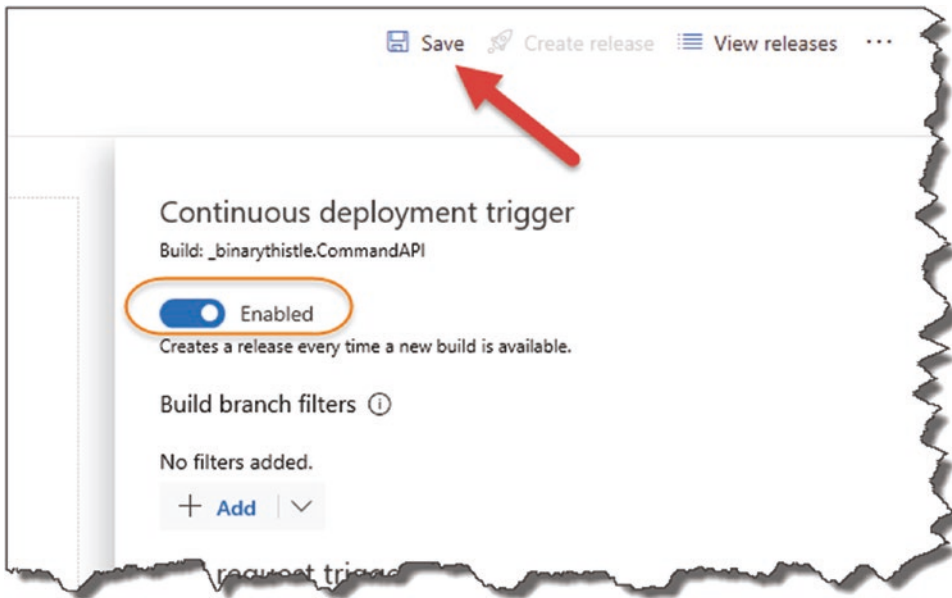
**Figure 13-46.** Configure the artifact

Click Add. Then click the lightning bolt on the newly created Artifact node.



**Figure 13-47.** Select triggers

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



**Figure 13-48.** Enable the Continuous deployment trigger

---

**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.



**Figure 13-49.** Again, add a comment if you want to

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

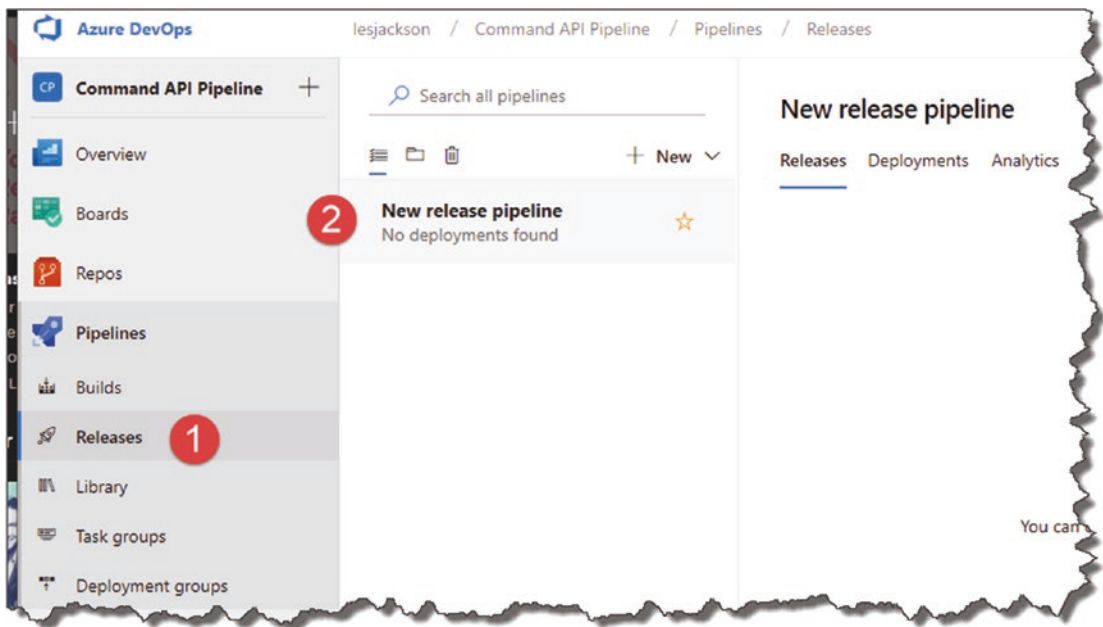


Figure 13-50. Release Pipeline created

## 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* will be triggered by the Build Pipeline, which will deploy our API App to Azure.

## Wait! What About EF Migrations?

Just before you do that – cast your mind back to Chapter 7 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 the *Startup* class.

This approach means that migrations will be applied when the app is started for the first time.

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

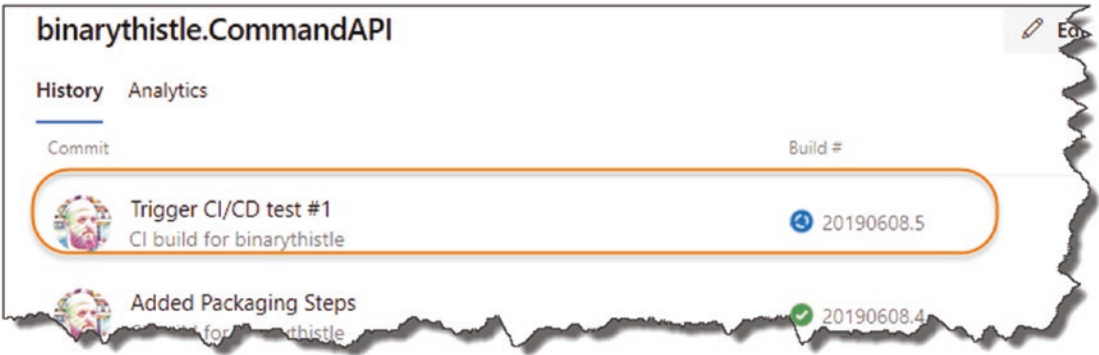
```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
CommandContext context)
{
    context.Database.Migrate();
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseMvc();
}
```

For clarity, the Configure method changes are highlighted in Figure 13-51.



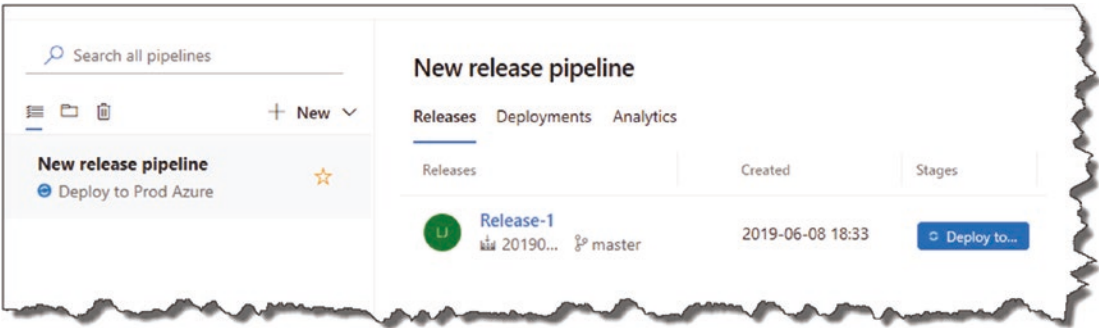
**Figure 13-51.** Migrate Database

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



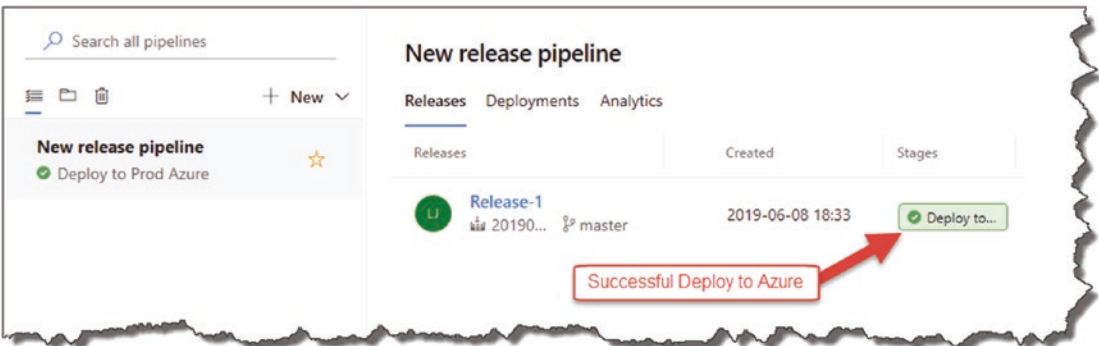
**Figure 13-52.** Pipeline triggered again

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



**Figure 13-53.** Release pipeline attempting to deploy

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).



**Figure 13-54.** Deployed!

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

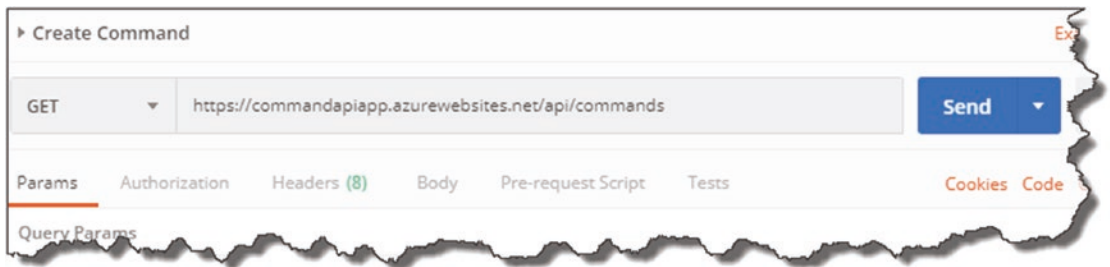
- Click All resources.
- Select you API App (App Service type).



**Figure 13-55.** Get the URI for your API App

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).



**Figure 13-56.** Call the API on Azure from Postman

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.



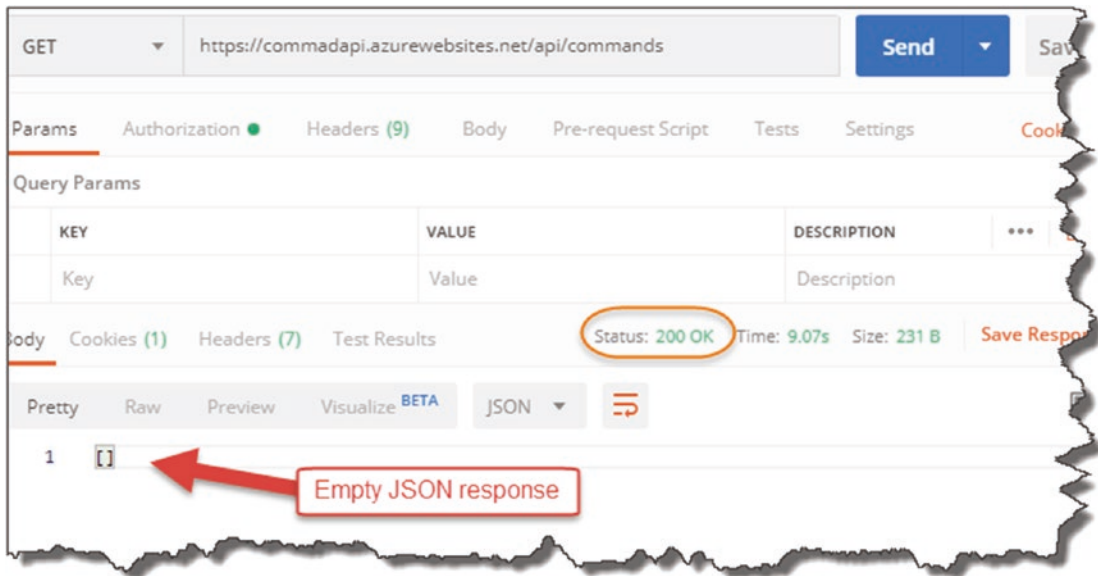


Figure 13-57. Success – but we have no data

**🎉 Celebration Checkpoint** Rad!<sup>3</sup> 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.

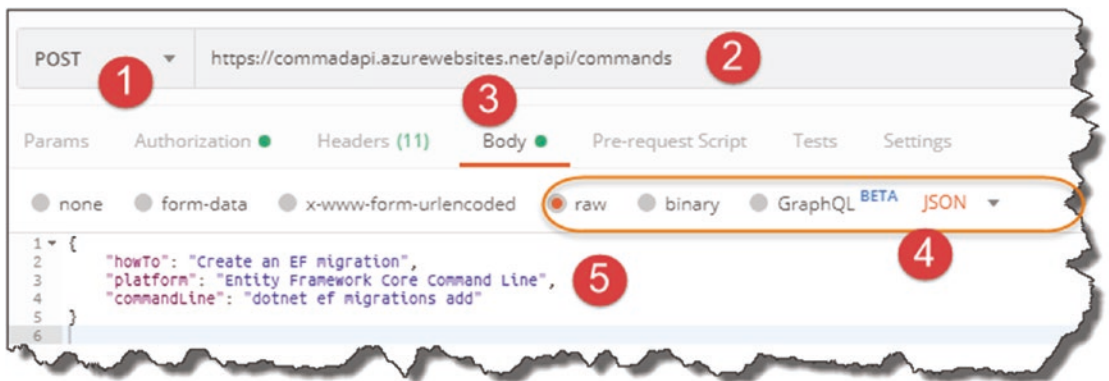
Using the following JSON string:

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

<sup>3</sup>Children of the 1990s will get this superlative.

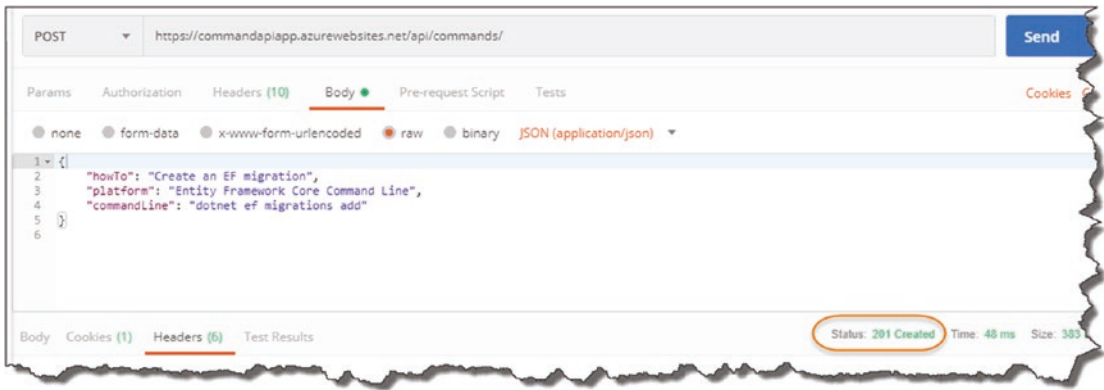
Create a new Postman request and set

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.



**Figure 13-58.** POSTing Data to out Azure hosted API

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



**Figure 13-59.** 201 Success!

And again, we have success!

---

**🎉 Celebration Checkpoint** Revel in the enormity of what you have just done! Not many people can say that have deployed an API on to the cloud via a CI/CD pipeline.

---

## CHAPTER 14

# 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 authorized clients to access our API resource through the 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 authorized 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 in Figure 14-1.

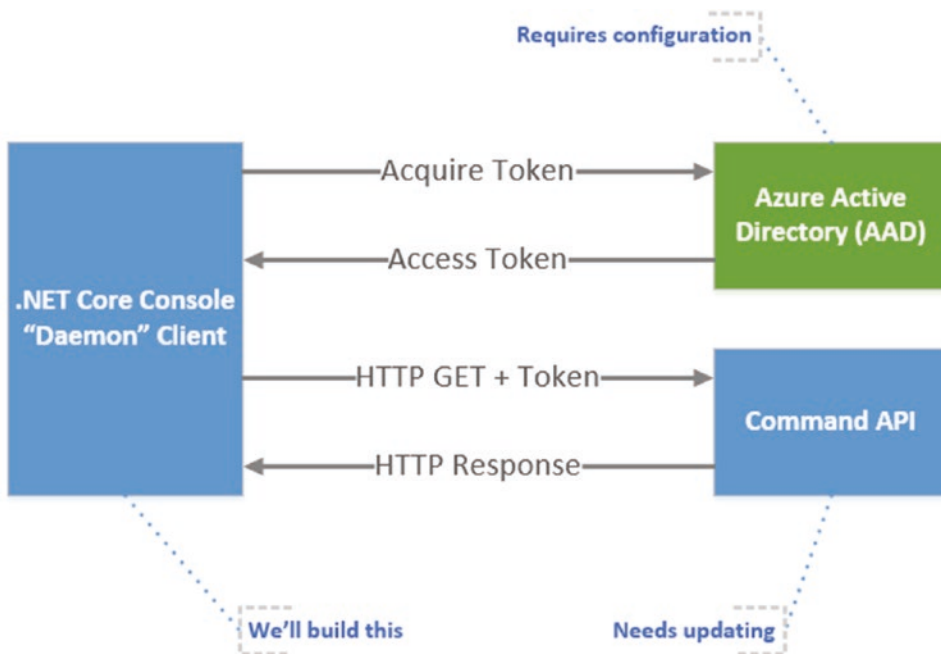


Figure 14-1. Authentication use case

## Overview of Bearer Authentication

There are a number of authentication schemes that we could have used, a non-exhaustive list is provided in the table.

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 authorization header of a http request. Natively, this is not encrypted, so it’s 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 <a href="#">RFC 6750</a> ); we’ll use to secure our API

(continued)

---

Scheme	Description
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. JWTs 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

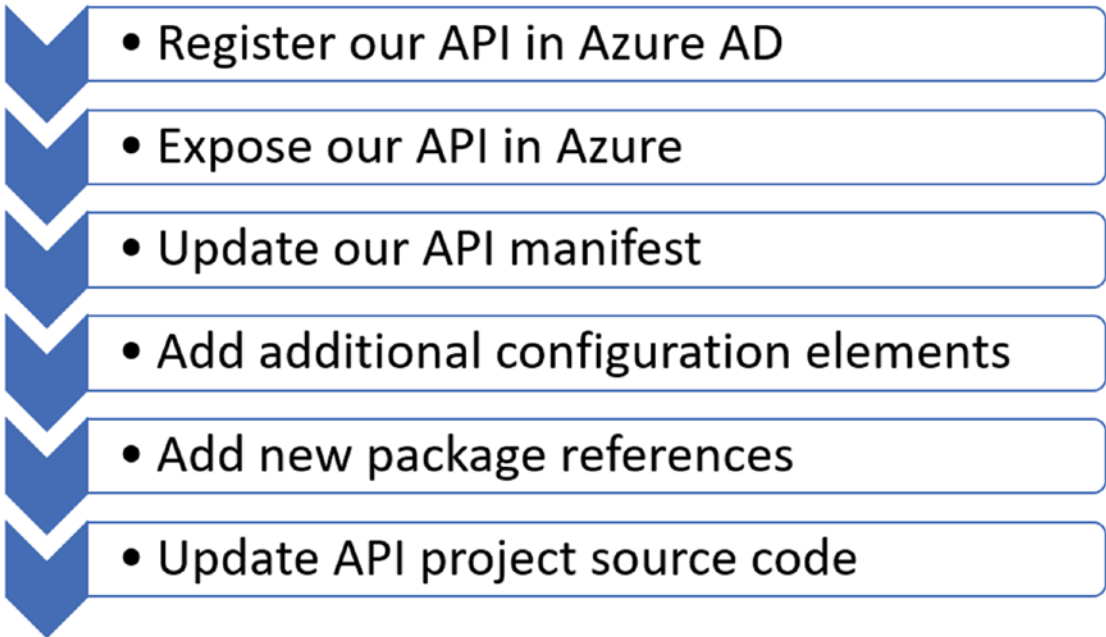
- Bearer authentication is the authentication scheme that makes use of (bearer) "tokens."
- JWT is a specific implementation of bearer tokens, in particular those with a JSON payload.

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

## Build Steps

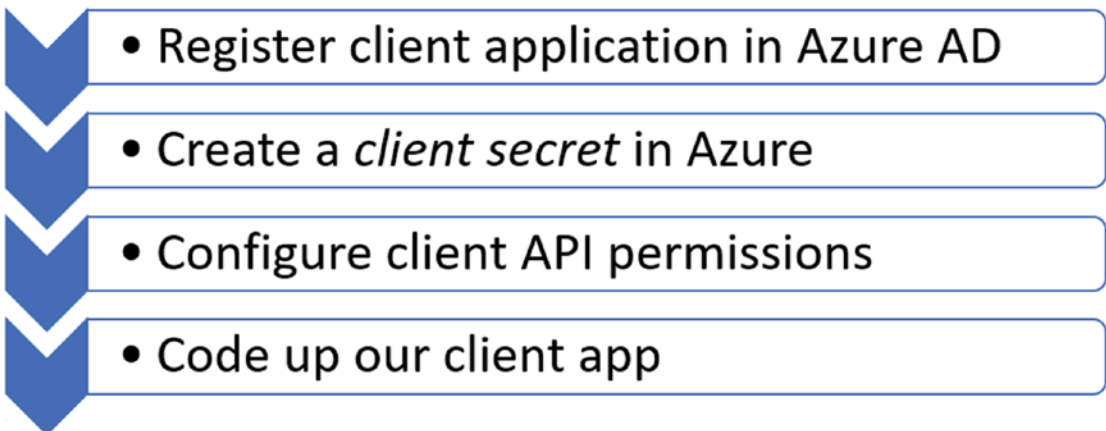
As I've mentioned before, I like a bit of 50,000ft view of what we're going to build before we start building it as it helps contextualize what we need to do, and it also allows us to understand the progress we're making. Therefore, in terms of the configuration and coding we need to perform, I've detailed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

## Steps for Our API Project

- 
- Register our API in Azure AD
  - Expose our API in Azure
  - Update our API manifest
  - Add additional configuration elements
  - Add new package references
  - Update API project source code

*Figure 14-2. API build steps*

## Steps for Our Daemon Client

- 
- Register client application in Azure AD
  - Create a *client secret* in Azure
  - Configure client API permissions
  - Code up our client app

*Figure 14-3. Client build steps*

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' 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 second 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 “organizational 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 10,000s (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 was obviously, cough, “inspired,” cough again, by NDS (and Banyan Vines – see next section) 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 been “inspired” by an earlier product called Banyan VINES.<sup>1</sup> 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.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Banyan\\_VINES](https://en.wikipedia.org/wiki/Banyan_VINES)



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

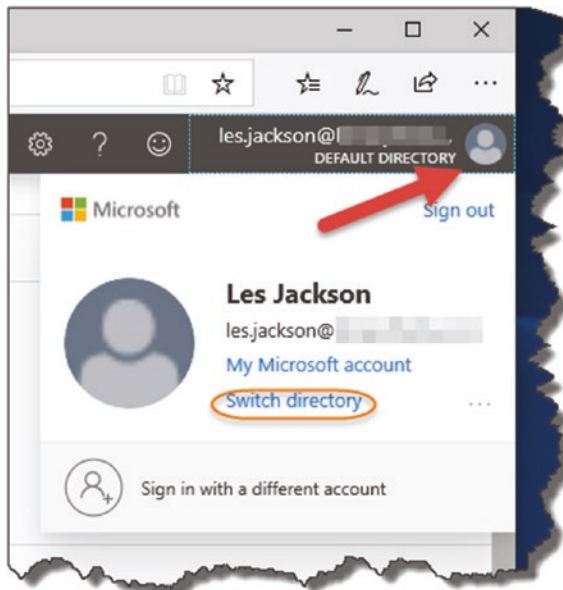
## Create a New AD?

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 (remember 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 in the Azure Portal.

## Switching Between AADs

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



**Figure 14-4.** *Switching Active Directory*

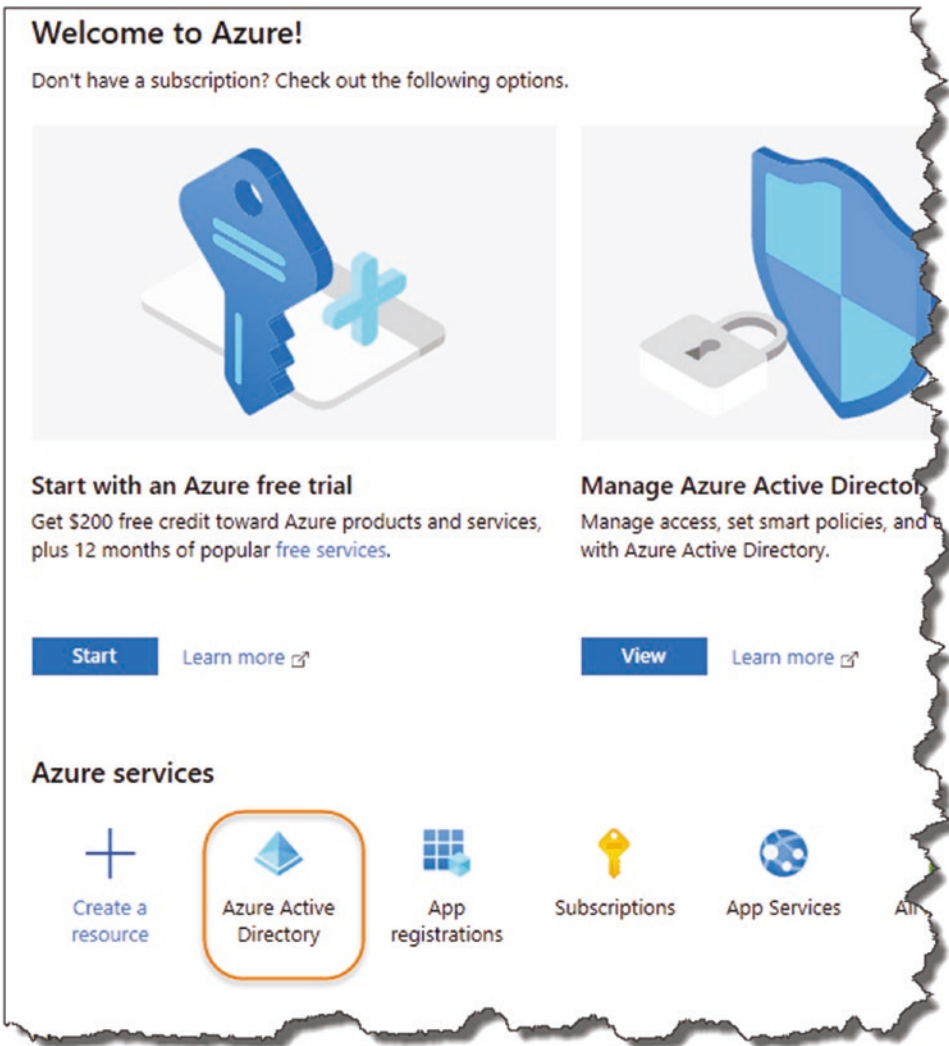
On the resulting pop-up, you can then click Switch Directory (see circled section on Figure 14-4); you should then get the option to select and switch between the AADs you have (I have two as you can see in Figure 14-5).



*Figure 14-5. I created a second AD for test purposes*

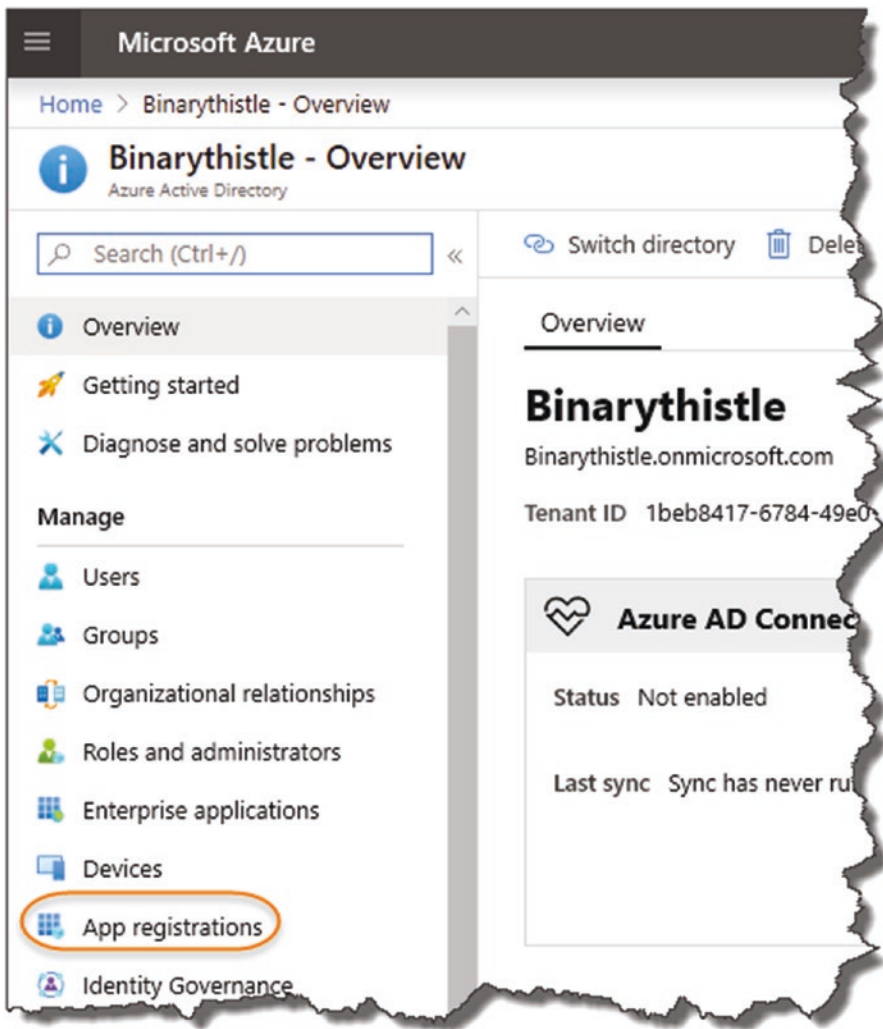
## Register Our API

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



**Figure 14-6.** *Select the AD you want to work with*

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

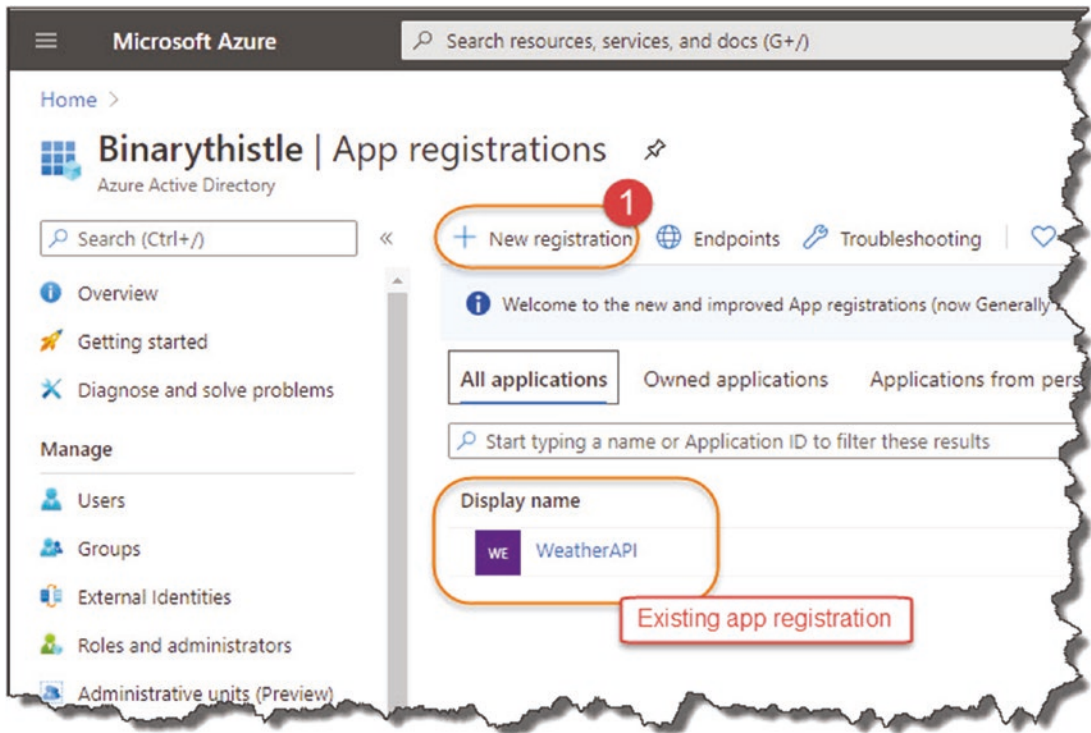


**Figure 14-7.** *Select App registrations*

Select “App registrations” as shown in Figure 14-7. You can see from the next example that I already have an existing app registered on my AAD, but we’re going to create a new one for our CommandAPI running on our “development” environment (i.e., the one running locally on our PC). We’ll come on to our Azure-deployed API later.

**i** 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 APIs (and client for that matter) are located.



**Figure 14-8.** Create a new registration

Select “New registration,” and you’ll see the following.

**Register an application**

**\* Name**  
The user-facing display name for this application (this can be changed later).

CommandAPI\_DEV

**Supported account types**  
Who can use this application or access this API?

- Accounts in this organizational directory only (Binarythistle only - Single tenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts

[Help me choose...](#)

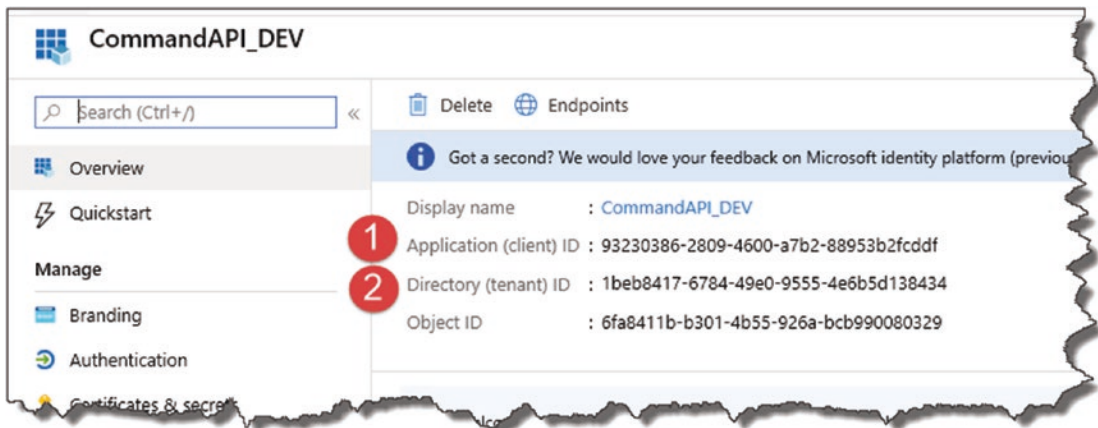
**Redirect URI (optional)**  
We'll return the authentication response to this URI after successfully authenticating the user. Pro changed later, but a value is required for most authentication scenarios.

Web

**Figure 14-9.** *Configure the registration*

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 Registrations 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.



**Figure 14-10.** We'll use client Id and tenant id

Here, we are introduced to the first two 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.

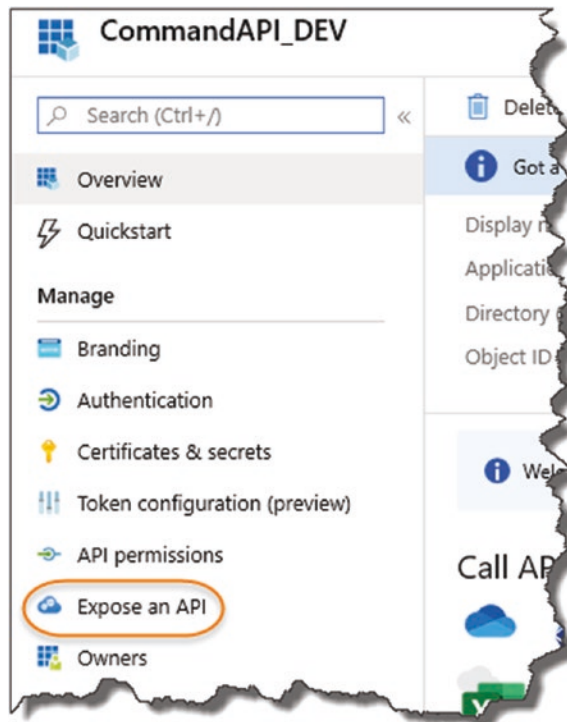
## Tenant ID

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

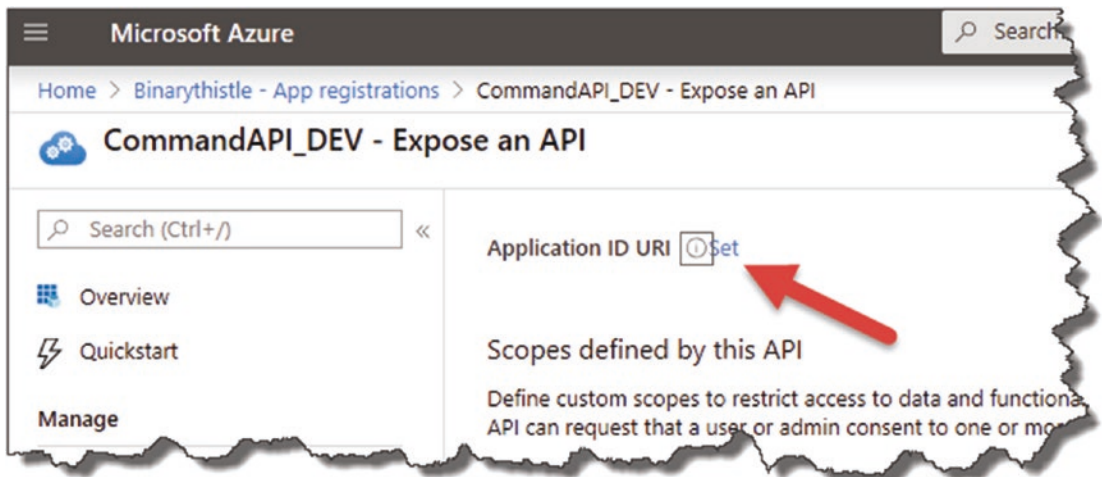
## Expose Our API

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



**Figure 14-11.** Exposing our API

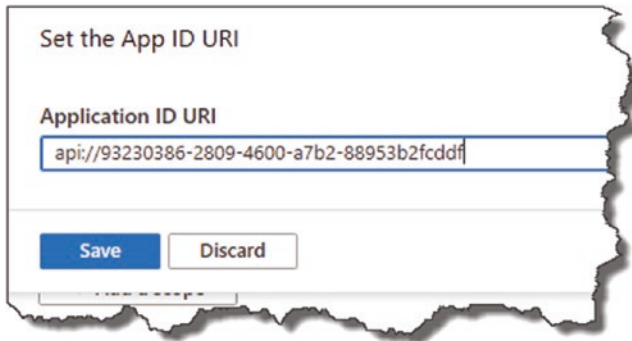
What we need to do here is create an “Application ID URI” (sometimes referred to as a “Resource ID”), so click “Set” as shown in Figure 14-12.



**Figure 14-12.** Set the Resource ID

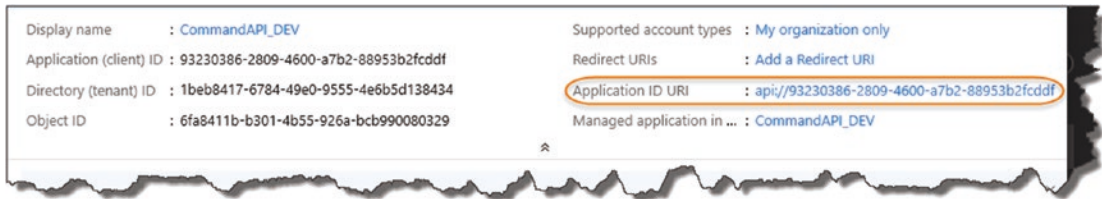


Azure will provide a default suggestion for this; go with it (it’s the Client ID with “api://” prepended).



**Figure 14-13.** Auto-generated Resource ID (Application ID URI)

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



**Figure 14-14.** Resource ID is created

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 noninteractive “daemon” app that will act as our API client. More information on the Application Manifest can be found in Microsoft Docs.<sup>2</sup>

<sup>2</sup><https://docs.microsoft.com/en-au/azure/active-directory/develop/reference-app-manifest>

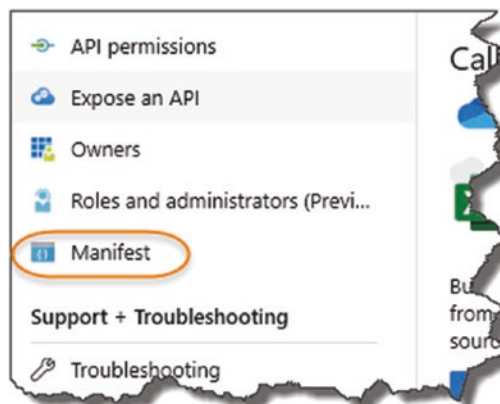
Anyway, back to the task at hand, we need to insert the following JSON snippet at the `appRoles` section of our manifest:

```

.
.
.
"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"
  }
],
.
.
.

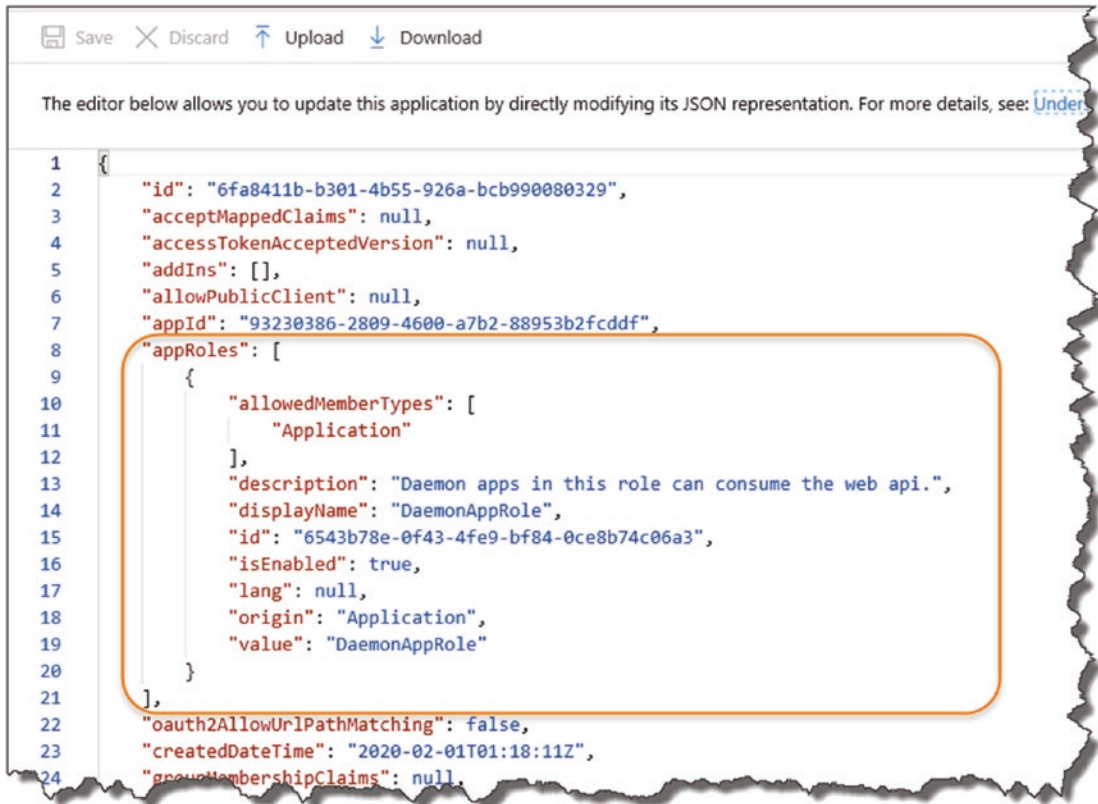
```

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



*Figure 14-15. Update the manifest*

And insert the json given earlier into the correct spot (essentially updating the existing empty appRoles section).



**Figure 14-16.** Ensure you update the manifest correctly

Make sure you keep the integrity of the json, and don't omit or introduce any additional commas. 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. You can use the example GUID I've supplied with the JSON here, or you can create your own (you can use the same GUID's across different AADs – you just can't duplicate them in the same AAD).

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 authorization.

## 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 log-in “Instance”
- Our AAD Domain
- The Tenant ID
- The Client ID
- The Application ID URL (or Resource ID)

---

**i** Remember we’re currently working with our API in our *Development* Environment, before we move on to configuring our API on Azure.

As we’ve already discussed, you can store your application config in a 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 8 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 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 log-in “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 (**CommandAPI**), 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.



```
{
  "UserID": "cmddbuser",
  "TenantId": "1beb8417-6784-49e0-9555-4e6b5d138434",
  "Password": "pa55w0rd!",
  "Instance": "https://login.microsoftonline.com/",
  "Domain": "Binarythistle.onmicrosoft.com",
  "ClientId": "93230386-2809-4600-a7b2-88953b2fcddf",
  "ResourceId": "api://93230386-2809-4600-a7b2-88953b2fcddf"
}
```

**Figure 14-17.** Example contents of my *secrets.json* file

Some points to note

- The value you have for Instance should be exactly the same as I’ve used earlier.
- The values you have for UserID and Password *may* be the same as what I’ve just shown if you’ve been following the tutorial *exactly* as I’ve described (they may of course be different if you’ve chosen your own values!).
- The values you have for TenantId, Domain, ClientId, and ResourceId will be different to mine.<sup>3</sup>

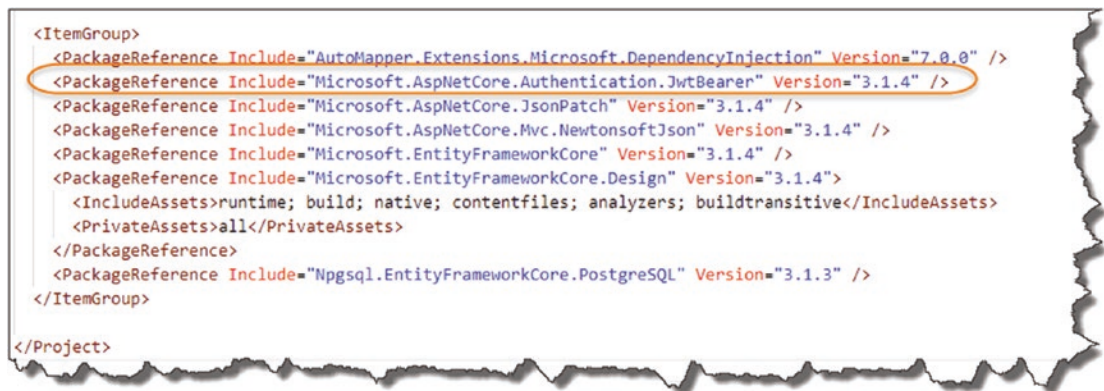
<sup>3</sup>The chances of the same GUID being generated for us both is quite slim.

## 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.



```
<ItemGroup>
  <PackageReference Include="AutoMapper.Extensions.Microsoft.DependencyInjection" Version="7.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="3.1.4" />
  <PackageReference Include="Microsoft.AspNetCore.JsonPatch" Version="3.1.4" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson" Version="3.1.4" />
  <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.4" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.4">
    <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    <PrivateAssets>all</PrivateAssets>
  </PackageReference>
  <PackageReference Include="Npgsql.EntityFrameworkCore.PostgreSQL" Version="3.1.3" />
</ItemGroup>
</Project>
```

**Figure 14-18.** Add Reference to allow JWT Bearer Authentication

## 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 is highlighted):

```
.  
.br/>.br/>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 the changes in context, it should 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().AddNewtonsoftJson(s => {  
        s.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();  
    });  
}
```

**Figure 14-19.** register Authentication service in Startup

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

- **Audience:** We set this to the ResourceID of our App Registration in Azure.
- **Authority:** Our AAD Instance that is the token issuing authority (a combination of Instance and TenantId).

## Update Configure

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

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

as shown in Figure 14-20.

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

    app.UseRouting();

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

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

**Figure 14-20.** Update the configure method in Startup



## Authentication vs. Authorization

As we've added both Authentication and Authorization 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.
- **Authorization (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* endpoints it can call on our API (authorization).

---

**⚠ Warning!** As authentication happens first (we need to identify you before we can authorize you to do anything), the order in which you add these components to the Request Pipeline (via the `Configure` method) is critically important. So please make sure you add them in the order specified earlier.

Refer back to Chapter 4 on our brief discussion on the Request Pipeline if you've forgotten (it was a while ago!); for a more in-depth conversation, refer to the Microsoft Docs.<sup>4</sup>

---

## 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 one 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 controller action, just make sure you add the following using directive at the top of our `CommandsController` class:

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

---

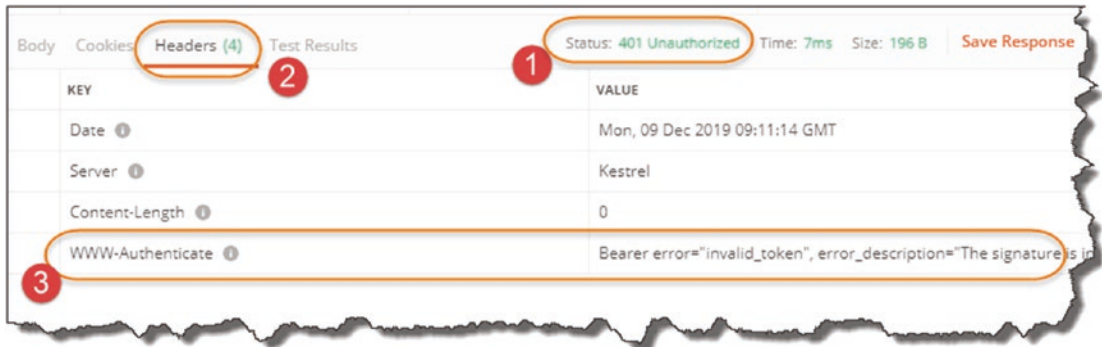
<sup>4</sup><https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/>

The new code for our controller action is simple; we just decorate it with the `[Authorize]` attribute as shown here:

### [Authorize]

```
[HttpGet("{id}", Name = "GetCommandById")]
public ActionResult<CommandReadDto> GetCommandById(int id)
{
    var commandItem = _repository.GetCommandById(id);
    if (commandItem == null)
    {
        return NotFound();
    }
    return Ok(_mapper.Map<CommandReadDto>(commandItem));
}
```

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

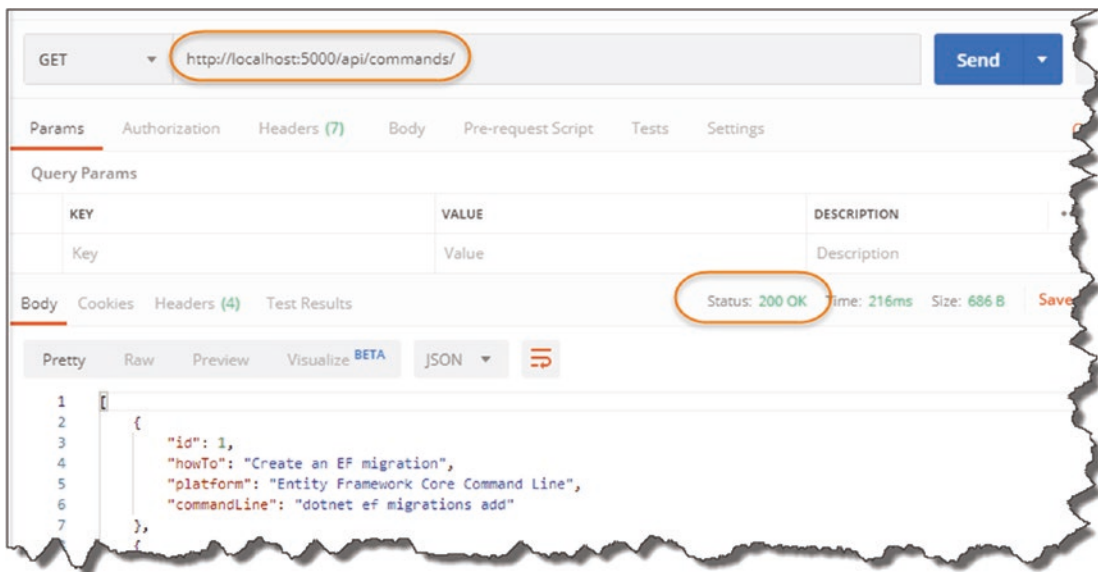


**Figure 14-21.** Our endpoint is secured

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 action, and you’ll see we still get a list of commands back.



**Figure 14-22.** This endpoint is not secured and can still be accessed

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

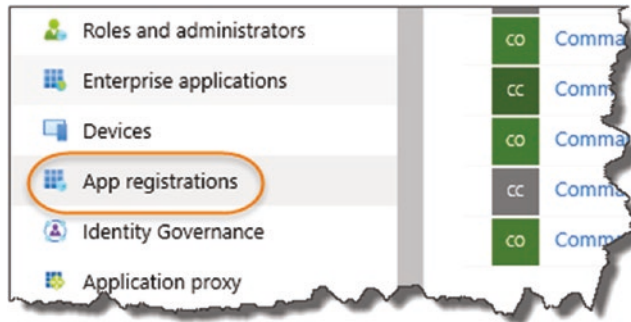
## Register Our Client App

In the next section, we’re going to write a simple .NET Core Console application that will act as an authorized “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.

**i** 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.

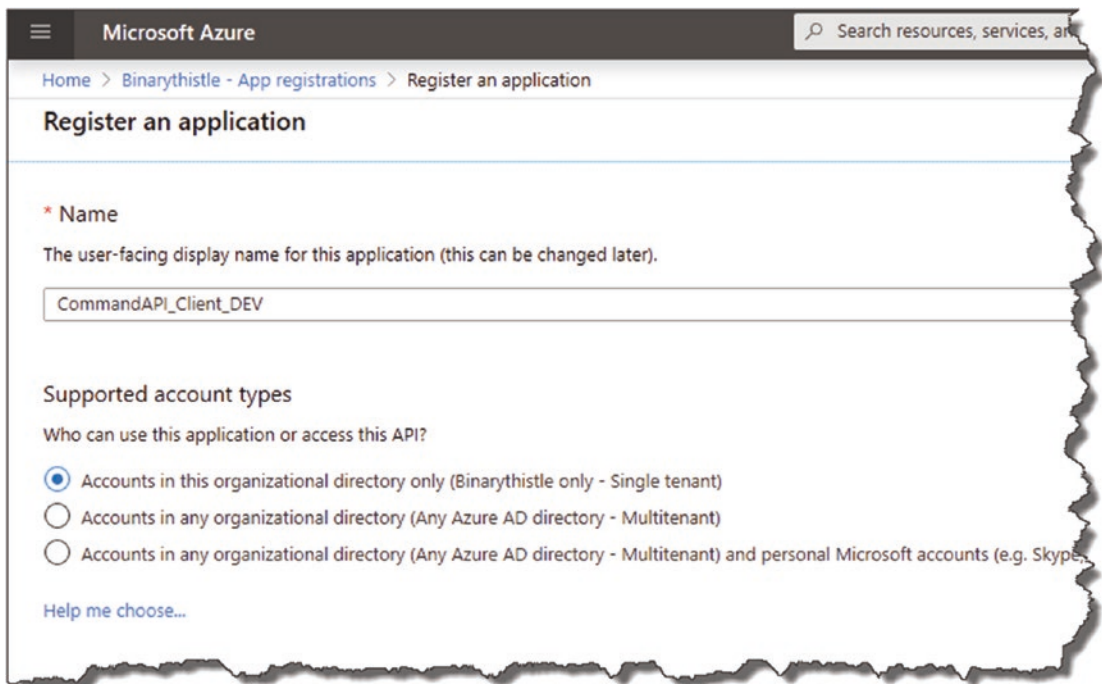
The use case I’ve decided to go with in this example (a “daemon app”) resonated with me more in terms of a real-world use case.

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



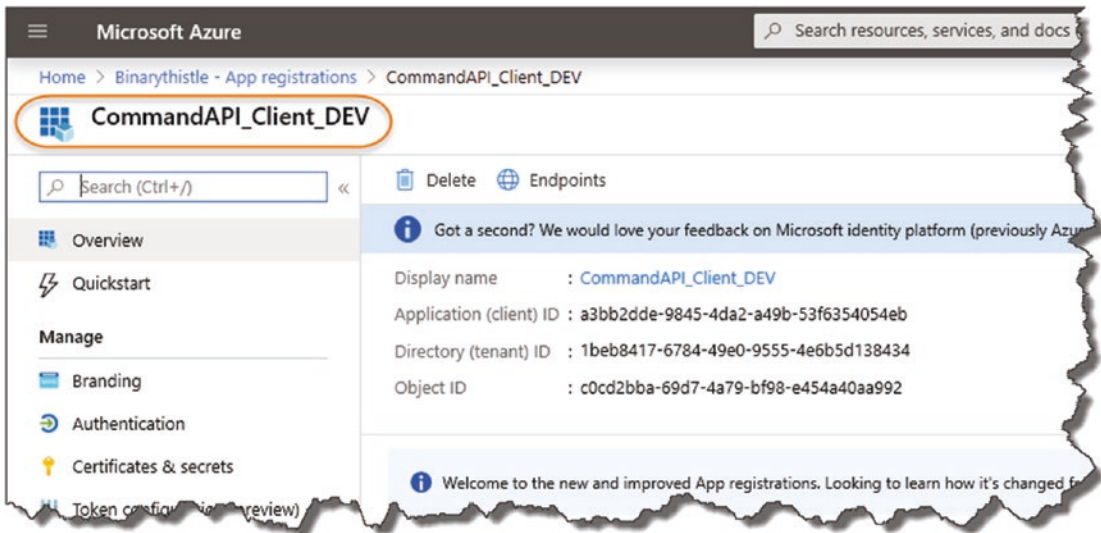
**Figure 14-23.** Create an App Registration for our client app

Then select “+ New registration,” and on the resulting screen enter a suitable name for our client app as shown next.



**Figure 14-24.** Name the registration

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.



**Figure 14-25.** Client registration overview

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

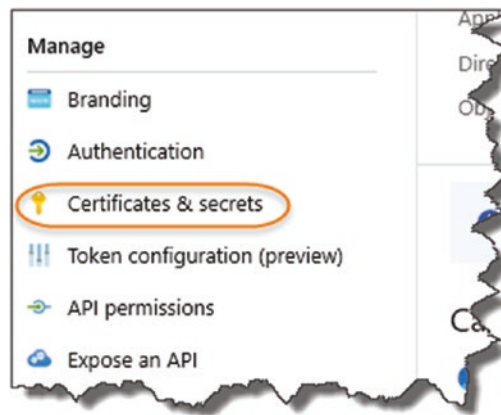
---

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

---

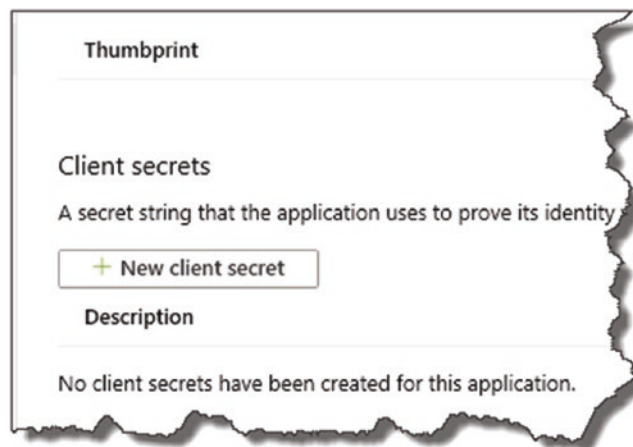
## Create a Client Secret

Next click “Certificates & secrets” in the left-hand menu.



**Figure 14-26.** Create a client secret

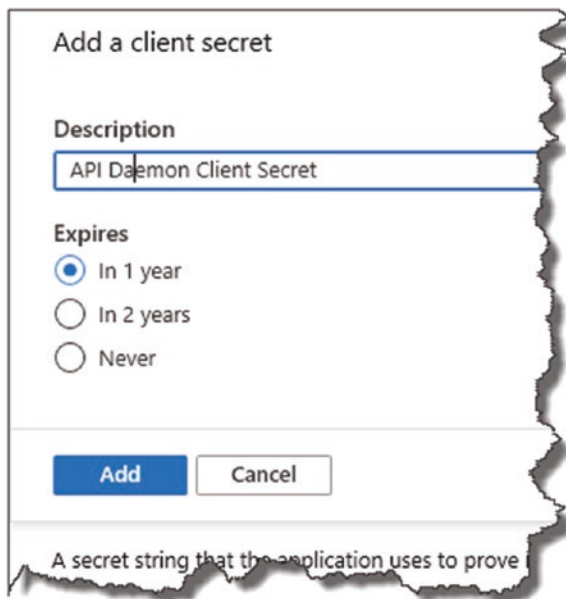
Here we are going to configure a “Client Secret.” This is a unique ID that we will use in combination with our other app registration attributes to identify and authenticate our client to our API. Click “+ New client secret.”



**Figure 14-27.** Select 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 3 options)



**Figure 14-28.** Name the secret and set expiry

When you're happy, click "Add."

---

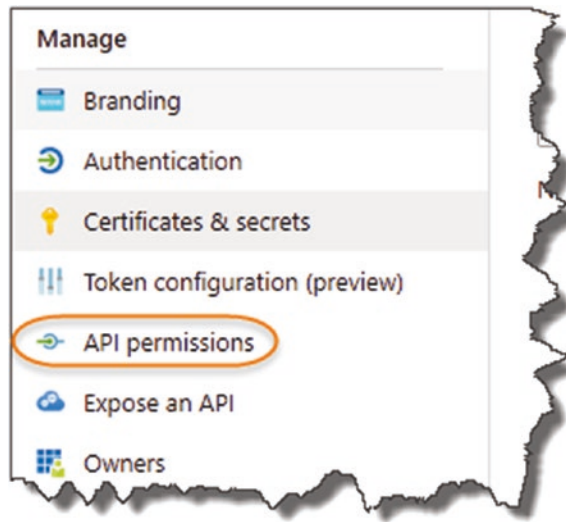
**⚠ Warning!** Make sure you **take a copy of the client secret now**; shortly after creation it will not be displayed in full again – you'll only see a redacted version, and you won't be able to retrieve it unlike our other registration attributes.

This is a by design security feature to help stop the unauthorized propagation of the client secret (which is effectively a password).

---

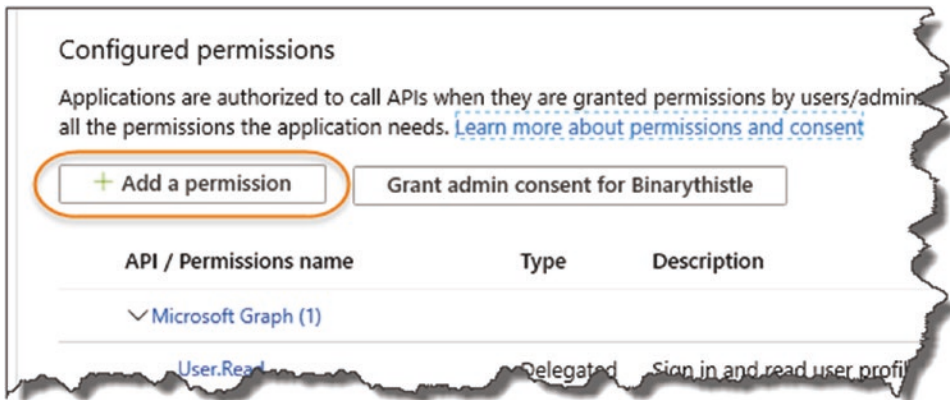
## Configure API Permissions

Now click "API Permissions"; here we are going to (drum roll please) configure access to our Command API.



**Figure 14-29.** Setup Permissions to our API

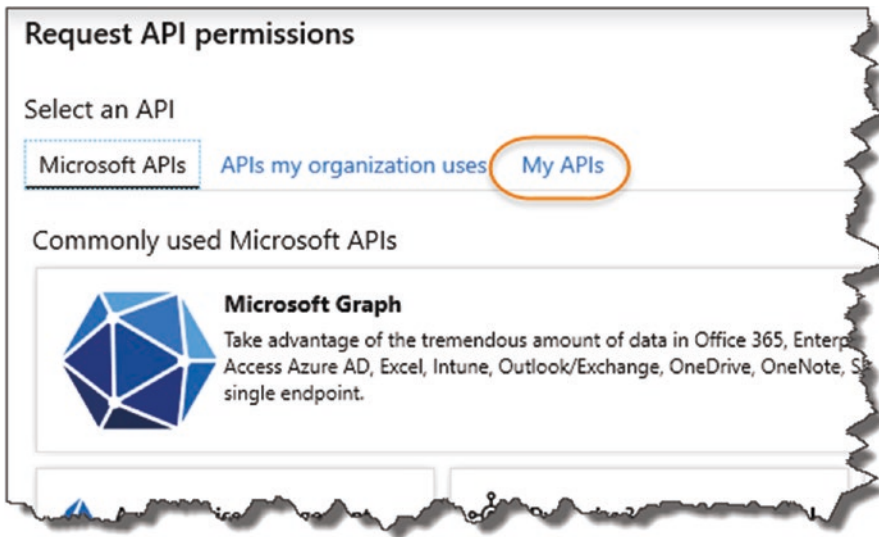
Click “+ Add a permission.”



**Figure 14-30.** Add a permission

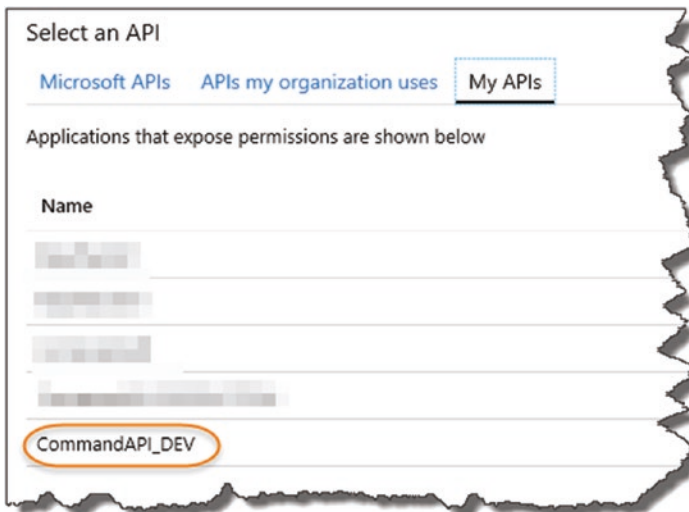
In the “Request API permissions” window that appears, select the “My APIs” tab.





**Figure 14-31.** select "My APIs"

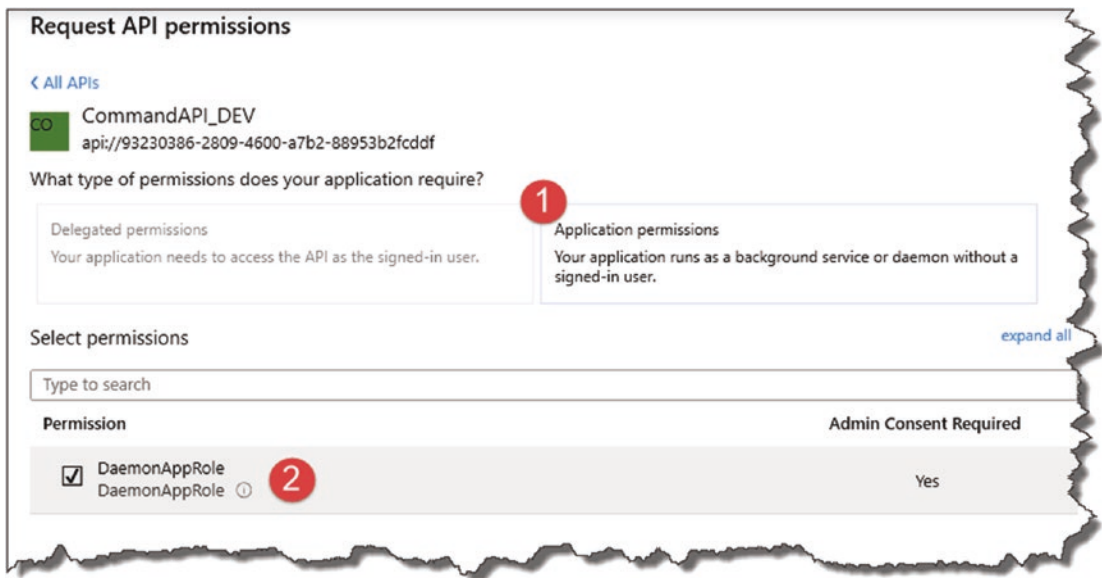
And find the Command API, and select it.



**Figure 14-32.** Select the CommandAPI\_DEV instance

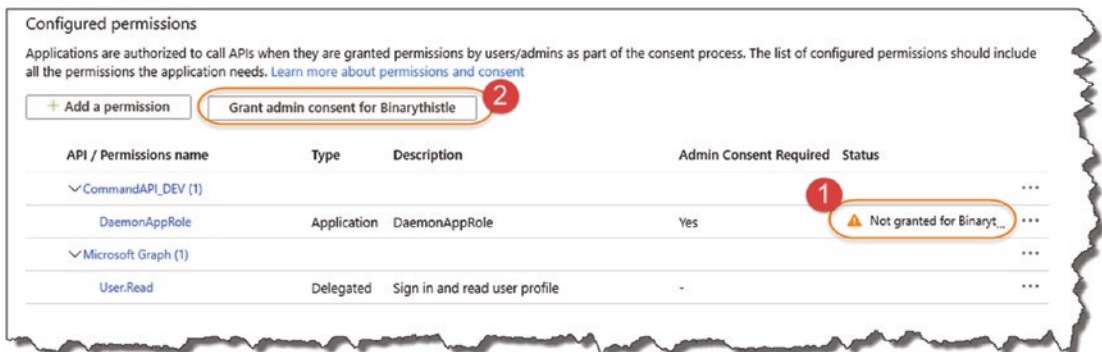
On the resulting screen, ensure that

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



**Figure 14-33.** Configure permissions accordingly

When you're happy, click "Add permission," and your permission will be added to the list.



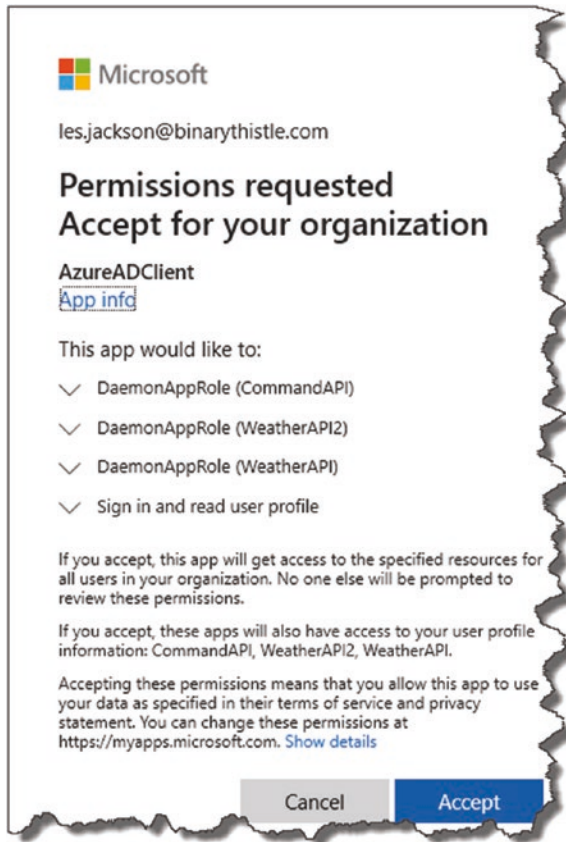
**Figure 14-34.** Grant consent

You'll notice

1. The permission has been "created" but not yet "granted."
2. You'll need to click the "Grant admin consent for <Name of Your AAD Here<sup>5</sup>>" button – do so now.

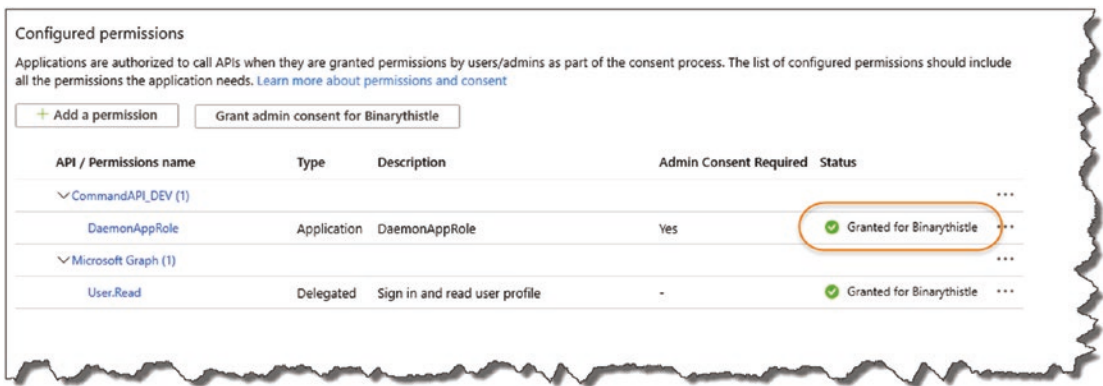
<sup>5</sup>The button will be labeled differently to mine depending on the name of your AAD.

You *may* get a Microsoft authentication pop-up; authenticate and accept any permissions requests you get (don't worry if this does not appear – it looks like this may be one of those ever-changing UI updates).



**Figure 14-35.** You may be asked to accept permission request

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



**Figure 14-36.** Permissions fully granted

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** I don't consider this app part of our "solution" (containing our API and 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:

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

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

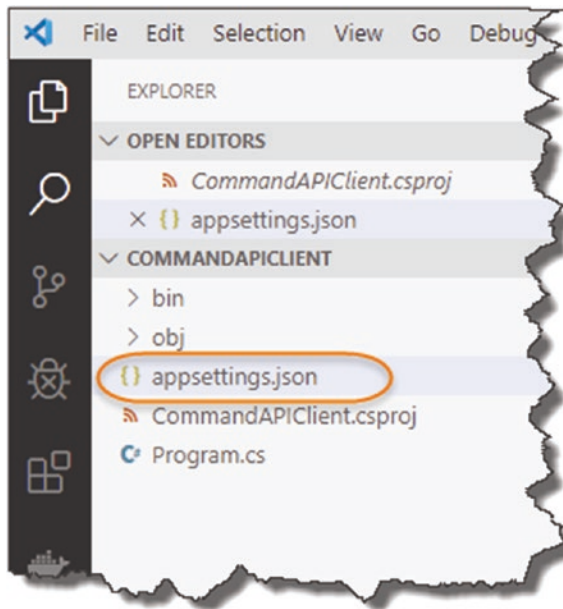
As I'm making this code [available on GitHub](#) for you to pull down and use, I'm deliberately going to store the config in an *appsettings.json* file as opposed to using User Secrets, as it will be easier for you to get going with it quickly if you choose to work with the code from the repo.<sup>6</sup> 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.



**Figure 14-37.** Create an Appsettings.json file

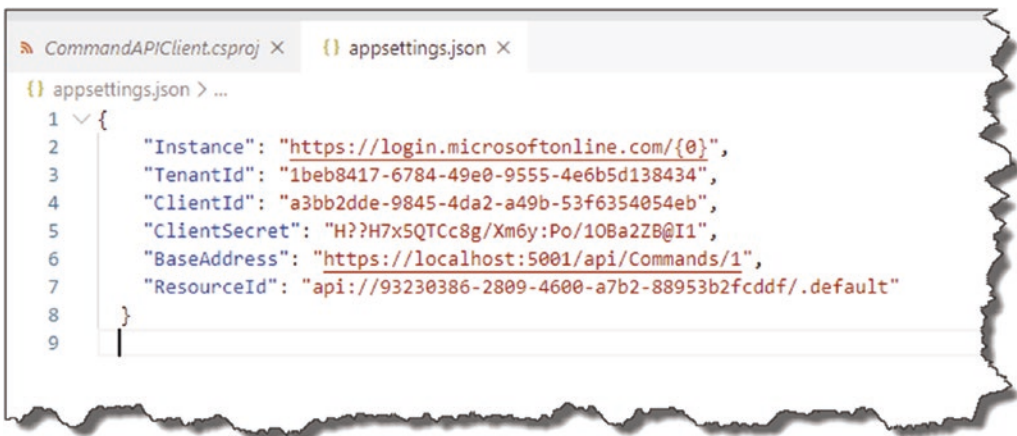
---

<sup>6</sup>I appreciate it's totally counter to the point I made before in regard to our API but feel this is a slightly different use case.

Into that file, add the following JSON; making sure to populate the correct values for **your client** application registration (TenantId, ClientId, and ClientSecret), and in the case of the ResourceId, make sure it's the ResourceId for the **API**:

```
{
  "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.



**Figure 14-38.** Client configuration

A couple of points to just double-check on:

- **BaseAddress:** This is just the local address of the command API (we'll update to our production URL later). Note that I'm deliberately specifying the API Controller Action 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 given 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, so as we've done previously, ensure your "in" the correct project folder (if you're following the tutorial exactly you should be "in" the *CommandAPIClient* folder), and issue the following command to add the first of our packages:

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

Repeat so you add all four packages; your project .csproj file should look like this when done.

```

CommandAPIClient.csproj
1  <Project Sdk="Microsoft.NET.Sdk">
2
3  <PropertyGroup>
4    <OutputType>Exe</OutputType>
5    <TargetFramework>netcoreapp3.1</TargetFramework>
6  </PropertyGroup>
7
8  <ItemGroup>
9    <PackageReference Include="Microsoft.Extensions.Configuration" Version="3.1.1" />
10   <PackageReference Include="Microsoft.Extensions.Configuration.Binder" Version="3.1.1" />
11   <PackageReference Include="Microsoft.Extensions.Configuration.Json" Version="3.1.1" />
12   <PackageReference Include="Microsoft.Identity.Client" Version="4.8.1" />
13 </ItemGroup>
14
15 </Project>

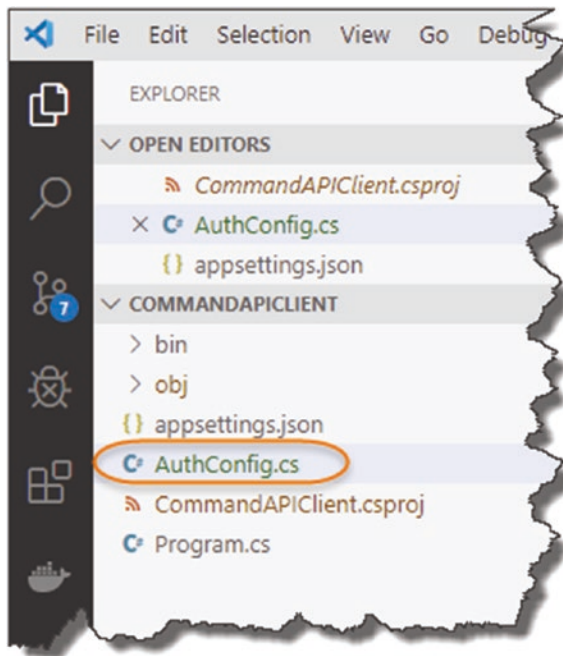
```

*Figure 14-39. Package References for our client*

## 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 in Figure 14-40.





**Figure 14-40.** *AuthConfig class to read in and manage client configuration*

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>();
        }
    }
}
```

**Figure 14-41.** Walk-through of Authconfig class

Notable code listed here


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 one 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.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS D:\APITutorial\NET Core 3.1\CommandAPIClient> dotnet run
Authority: https://login.microsoftonline.com/1beb8417-6784-49e0-9555-4e6b5d138434
PS D:\APITutorial\NET Core 3.1\CommandAPIClient> █
```

*Figure 14-42. Run the client*

## Finalize 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 next (noting new or changed code is bold and 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 here.

```

static void Main(string[] args)
{
    Console.WriteLine("Making the call...");
    RunAsync().GetAwaiter().GetResult(); 1
}

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

    IConfidentialClientApplication app; 2

    app = ConfidentialClientApplicationBuilder.Create(config.ClientId) 3
        .WithClientSecret(config.ClientSecret)
        .WithAuthority(new Uri(config.Authority))
        .Build();

    string[] ResourceIds = new string[] { config.ResourceID }; 4

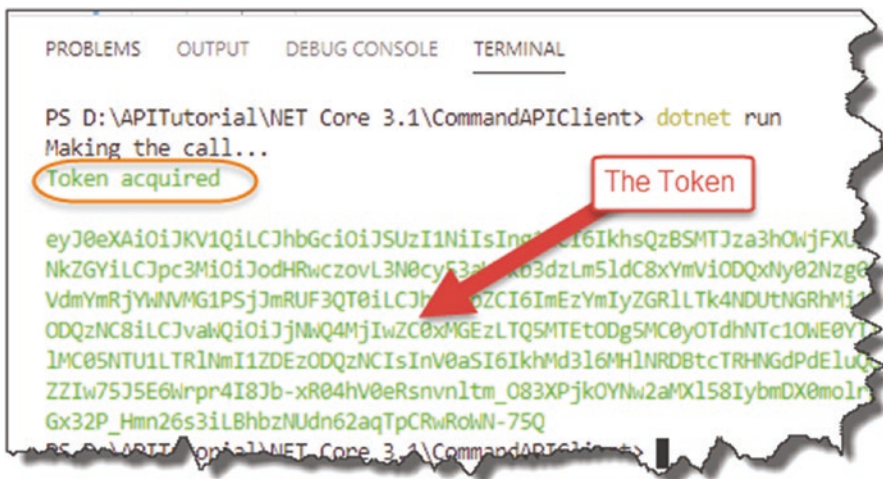
    AuthenticationResult result = null; 5
    try
    {
        result = await app.AcquireTokenForClient(ResourceIds).ExecuteAsync();
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine("Token acquired \n"); 6
        Console.WriteLine(result.AccessToken);
        Console.ResetColor();
    }
    catch (MsalClientException ex)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(ex.Message);
        Console.ResetColor();
    }
}
}

```

**Figure 14-43.** *Progressing the client*


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 AcquireTokenForClient 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; you should see the following.



**Figure 14-44.** Successful token acquisition

---

 **Celebration Checkpoint** 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 second 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 three 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 here.

```

catch (MsalClientException ex)
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(ex.Message);
    Console.ResetColor();
}
if (!string.IsNullOrEmpty(result.AccessToken))
{
    var httpClient = new HttpClient(); 1
    var defaultRequestHeaders = httpClient.DefaultRequestHeaders;
    if (defaultRequestHeaders.Accept == null ||
        !defaultRequestHeaders.Accept.Any(m => m.MediaType == "application/json")) 2
    {
        httpClient.DefaultRequestHeaders.Accept.Add(new
            MediaTypeWithQualityHeaderValue("application/json"));
    }
    defaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue("bearer", result.AccessToken); 3
    HttpResponseMessage response = await httpClient.GetAsync(config.BaseAddress); 4
    if (response.IsSuccessStatusCode) 5
    {
        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();
}

```

**Figure 14-45.** *Calling the API*

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 authorization 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.

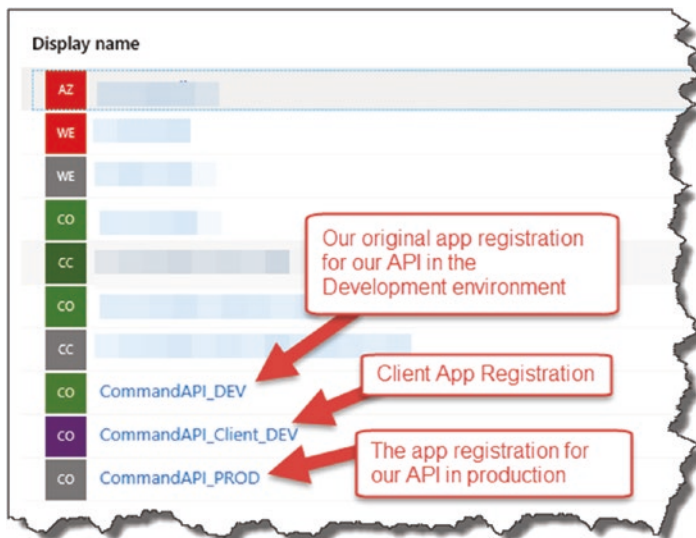


Config element	Application setting name
The log-in “instance”	Instance
Our AAD Domain	Domain
The Tenant ID	TenantId
The Client ID	ClientId
The Application UD URL (Or Resource ID)	ResourceId

Before we do that though, while we could reuse 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!

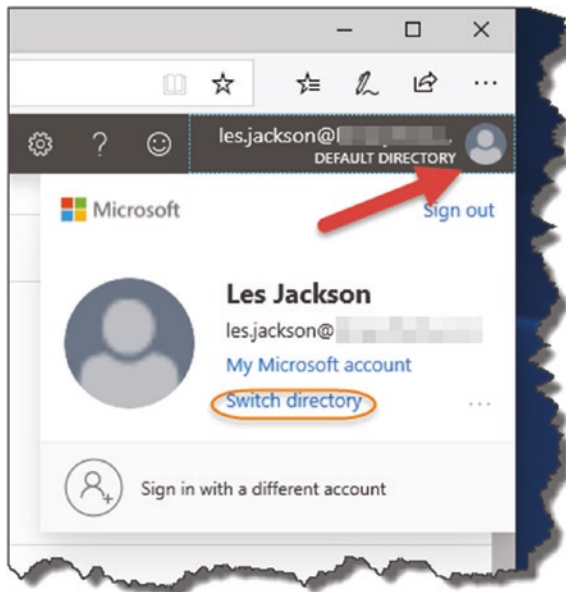
How did you go? Easy right? You should now have something similar to the following in your app registrations list.



**Figure 14-47.** Production App Registrations

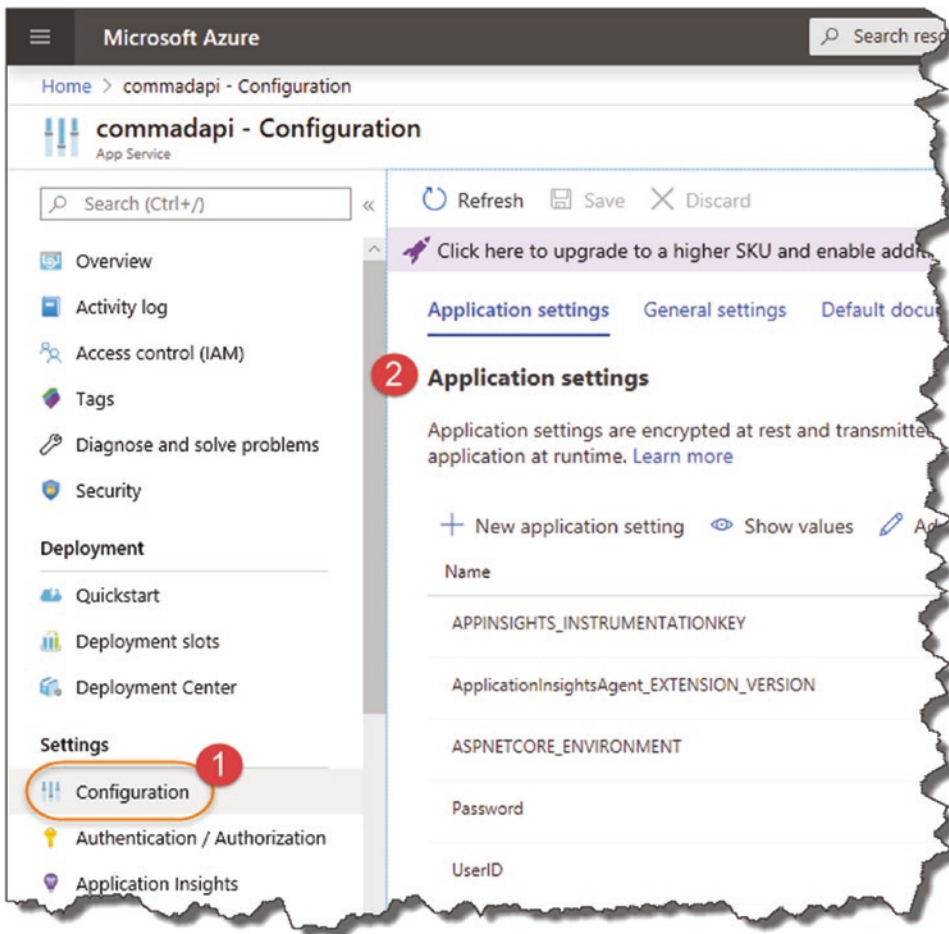
If like me you created your App Registrations in a different Azure Directory to your main one (i.e., where all your resources are), I'd take a note of all of the values for things like TenantId, ClientId, and ResourceId in the Production App Registration you just created before you switch back to your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

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



**Figure 14-48.** *Switching active directories*

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



**Figure 14-49.** API app settings

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.



**Figure 14-50.** Additional app settings to support authentication

---

**Note** Remember to Save the new Application Settings you’ve just 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 two steps mentioned.

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

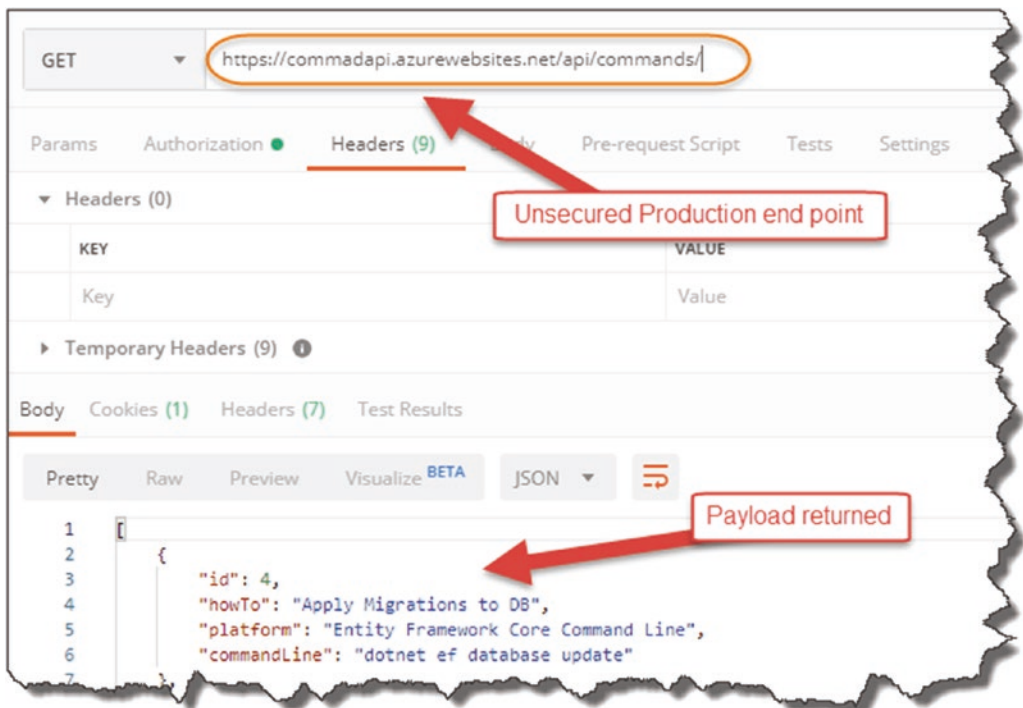
When done, come back here.

---

## 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 and push.

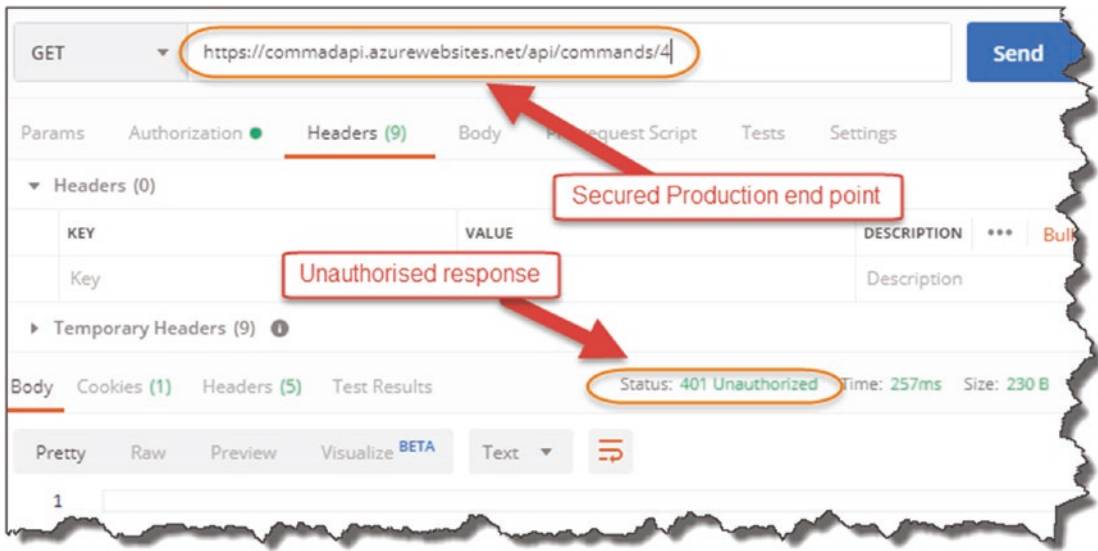
As before, our Build Pipeline should succeed as should our deployment. Using something like Postman to call an unsecured endpoint should still work as before.



**Figure 14-51.** Unsecured endpoint continues to work

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





**Figure 14-52.** Secured endpoint declines the request

Turning to our client app (with updated configuration to access Production), making a call-through to our secured endpoint will yield a successful result.



**Figure 14-53.** Successful call of our secure endpoint on Azure

# 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 a 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 two weeks. In reality it took well over four 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 (although I guess that's totally subjective depending on who you'd ask) 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.

# Index

## A

### Active Directory (AD)

- switching option, [400](#)
- test purposes, [401](#)

### Application programming interface (API)

- command-line repository, [24](#), [25](#)
- CRUD operations, [25](#)
- JSON, [26–30](#)
- meaning, [23](#)
- payloads, [26](#)

### ASP.NET Core project

- files and folders, [41](#), [42](#)
- Nuget, [42](#)
- program class, [43](#), [44](#)
- startup class
  - ConfigureServices, [45](#)
  - dependency injection, [45](#)
  - execution sequence, [44](#)
  - method, [44](#)
  - middleware components, [45](#)
  - configure, [45](#)
  - request pipeline, [46](#)

### Asynchronous operations, [69](#)

### Authentication scheme

- API project/Daemon client, [398](#)
- bearer authentication, [397](#), [398](#)
- non-exhaustive list, [396](#)
- registering API, [399](#), [400](#)
- secure, [395](#)
- user ids and passwords, [395](#)

### AutoMapper package

- API project folder, [197–199](#)
- architecture check, [206](#)
- CommandReadDTO, [204](#)
- constructor dependency injection, [201](#)
- context, [201](#)
- controller, [202](#)
- DTO representation, [202](#)
- GET controller actions, [203](#)
- mapping, [200](#)
- multiple instances, [202](#)
- postman query, [205](#)
- profiles folder/CommandsProfile.cs file, [199](#), [200](#)
- reference, [197](#)
- service registration, [198](#)
- startup class and register, [198](#)

### Azure Active Directory (AAD)

#### AD (*see* Active Directory (AD))

#### API exposes

- app registration, [408](#)
- auto-generated
  - resource ID, [408](#)
- registrations page, [406](#)
- resource ID, [407](#)

#### app registrations, [403](#)

#### authentication (*see* Authentication scheme)

#### authentication *vs.*

- authorization, [416](#)

## INDEX

### Azure Active Directory (AAD) (*cont.*)

#### Azure updates

- application settings screen, 444, 445
- client configuration, 446
- deployment, 447, 448
- production app registrations, 443
- secured endpoint, 448
- switching option, 444
- unsecured endpoint, 447
- user secrets, 442

#### client app, 418–422

#### client app creation

- appsettings.json file, 428
- AuthConfig class, 432
- configuration, 428–430
- configuration class, 431–435
- console project, 427
- package references, 430, 431
- progressing, 437
- program class, 435–442
- token acquisition, 438

#### client Id and tenant id, 406

#### CommandsController class, 416–418

#### configuration, 405

#### configuration elements, 411, 412

#### ConfigureServices method, 413–415

#### endpoint, 418

#### landing page, 401, 402

#### manifest, 408–410

#### configure method, 415, 416

#### overview screen, 402, 403

#### permissions

- accept request, 426
- access configuration, 422
- CommandAPI\_DEV instance, 424
- configuration window, 425, 426
- grant consent, 425
- resulting screen, 424

#### window, 423

#### project packages, 413

#### registration, 404

#### secrets.json file, 412

#### startup class file, 413

### Azure resources

#### API app code

##### account details, 356

##### app service plan, 356, 357

##### configuration, 353

##### creation, 350, 352

##### deployment, 357

##### free option, 355

##### notification, 358

##### pricing tier, 355

##### public landing page, 359

##### search box, 351

##### service plan, 354

##### URI overview, 358

#### connect/DB user creation

##### databases, 369

##### environment variables, 369

##### FDQN, 368

##### location, 368

#### dev environment, 370, 371

#### learning goals, 349

#### PostgreSQL, 359–368

#### resource creation, 350

#### setup configuration

##### application settings, 372

##### connection string, 371–373

##### connection strings, 372

##### DB user credentials, 374–376

##### production environment, 371

##### runtime environment, 377–379

#### trigger

##### continuously deployment, 388

##### double-check, 392–394

- EF migrations, [388–392](#)
  - pipeline, [390](#)
- B**
- Bearer authentication, [397, 398](#)
- C**
- C# extension, [11](#)
  - CI/CD Pipeline, Azure resources
    - artifact configuration, [385](#)
    - comment, [383, 387](#)
    - continuous deployment trigger, [386](#)
    - deployment, [379, 384](#)
    - DevOps releases, [380](#)
    - errors and remember, [383](#)
    - pipeline, [388](#)
    - pipeline tab, [384](#)
    - release pipeline, [381](#)
    - service deployment template, [381](#)
    - task errors, [382](#)
    - triggers selection, [386](#)
  - CommandCreateDto
    - AutoMapper profile, [216](#)
    - internal command model, [214](#)
    - CommandCreateDto.cs, [215](#)
    - scenario, [214, 215](#)
    - source code, [216](#)
  - Container instance pricing
    - advanced settings, [365](#)
    - configuration, [362, 363](#)
    - detail screen, [362](#)
    - networking tab, [364, 365](#)
    - PostgreSQL option, [361](#)
    - resources up and running, [368](#)
    - search option, [362](#)
    - setup environment variables, [366](#)
  - tags tab, [366](#)
  - validation, [367](#)
- Continuous Integration/Continuous Delivery (CI/CD)
- azure-pipelines.yml file
    - source code, [331](#)
    - unit tests, [333–338](#)
    - VS code extension, [332](#)
  - build pipeline, [342, 343](#)
  - integration, delivery/
    - deployment, [307](#)
  - learning goals, [305](#)
  - meaning, [306](#)
  - Microsoft documents, [345](#)
  - pipeline (*see* Pipeline)
  - release/packing stage
    - package and publish steps, [345](#)
    - pipeline, [343](#)
    - running job, [346](#)
    - VS Code, [344](#)
  - triggering
    - auto-triggered build, [330](#)
    - CommandAPISolution, [328](#)
    - GetCommandItems
      - method, [328](#)
    - local/remote repos, [329](#)
    - pull down, [329](#)
    - push command, [328](#)
    - random change, [328](#)
    - VS Code file tree, [330](#)
- Controller action implementation
- PUT request
    - command resource retrieval, [233](#)
    - HttpPut attribute, [233](#)
    - input parameters, [233](#)
    - return option, [234](#)
    - save/return, [235](#)
    - update command, [232–234](#)

## INDEX

### Controllers, MVC pattern

- ApiController attribute, 67
  - API run, 63
  - CommandsController.cs, 60
  - directives, 65
  - directory structure, 61
  - endpoint response, 64
  - file and folder creation, 61
  - folder, 60
  - GitHub, 62
  - HttpGet attribute, 68
  - inherit, 65
  - routing setup, 66, 67
  - synchronous vs. asynchronous, 68, 69
- Create, read, update, and delete (CRUD) operations, 25

## D

### Data Access (aka DB Context), 58

### Data Transfer Objects (DTOs), 1, 58, 191

- architecture progress, 191, 192
  - AutoMapper (*see* AutoMapper package)
  - automation framework, 195
  - benefits, 194
  - decouple interface, 193
  - different actions, 194, 195
  - Dtos tolder/CommandReadDto.cs
    - file, 196, 197
  - implementation, 195
  - potential consequences, 193
- DBeaver *vs.* pgAdmin, 19, 20
- DeleteCommand unit tests, 302–304
- Delete endpoint (DELETE)
  - attributes, 249
  - CommandDeleteDto, 250
  - controller, 250

### DeleteCommand method, 250

### high-level definition, 248

### interface method, 249

### repository interface, 250

### Dependency Injection (DI)

### ConfigureServices method, 99, 102

### constructor

#### action, 108

#### CommandsController class, 104

#### controller actions, 112

#### endpoint result, 109

#### endpoints, 106

#### GetCommandByID endpoint, 111

#### implementation, 110

#### pattern, 105

#### resources, 106

#### single command resource, 112

#### source code, 102, 103

#### statement, 107

#### straightforward, 107

### methods, 102

### service container, 101

### startup class sequence, 100

### Development environment

#### components, 170

#### connection string, 180

#### DBeaver *vs.* pgAdmin, 19, 20

#### Docker, 15–18

#### Git installation, 13–15

#### ingredients, 6–8

#### launchSettings.json file, 170–173

#### learning goals, 5

#### .NET Core SDK, 12, 13

#### PostgreSQL, 18

#### Postman installation, 20

#### software/sites, 8

#### trust local development

#### certificate, 21

- Visual Studio Code, [8–12](#)
- Windows/Mac/Linux, [5](#)
- Dev environment, [370, 371](#)
- Docker
  - containerization platform, [15](#)
  - desktop vs. CE, [16](#)
  - Hello World image, [17](#)
  - plugin VS code, [18](#)
  - post-installation check, [16, 17](#)

## E

- Endpoints (controller actions)
  - architecture checkpoint, [252](#)
  - bad request, [225](#)
  - binding sources, [219](#)
  - CommandCreateDto, [214–217, 220](#)
  - CommandsController class, [217](#)
  - CreateCommand implementation, [218](#)
  - CreatedAtRoute, [220](#)
  - data annotations, [225, 226](#)
  - data persist, [220](#)
  - delete endpoint, [248–252](#)
  - EF Core DB context, [207, 208](#)
  - GetCommandById method, [221, 222](#)
  - HttpPost, [218](#)
  - input DTO type, [219](#)
  - learning goals, [207](#)
  - map creation, [220](#)
  - parameter, [221](#)
  - POST creation, [209, 210](#)
  - POST request, [219](#)
  - repository, [211–214](#)
  - resource, [224](#)
  - return DTO type, [218](#)
  - set up, [223](#)
  - testing, [223](#)
  - validation detail, [226](#)

- Entity Framework Core (EF Core)
  - appsettings.json file
    - connection string, [137](#)
    - database, [138](#)
    - DBeaver, [133](#)
    - migrations, [133](#)
    - role permissions, [136](#)
    - source code, [136](#)
    - SQL editor, [133–138](#)
    - view role details, [135](#)
  - code first/database first
    - CommandItems, [148](#)
    - contexts, [145](#)
    - database and table, [148](#)
    - entity framework, [143](#)
    - migration files, [144, 145](#)
  - command-line tools, [127, 128](#)
  - database
    - CommandContext.cs, [131, 132](#)
    - compilation errors, [132](#)
    - context class, [128](#)
    - .csproj file, [130](#)
    - package references, [130](#)
    - reference packages, [129](#)
  - migrations, [143](#)
  - mock data
    - command-line snippets, [151](#)
    - commands, [148](#)
    - default DB, [150](#)
    - entity framework, [151](#)
    - returns, [151](#)
    - set default database, [150](#)
    - SQL INSERT commands, [149, 152](#)
    - table, [149](#)
  - object wrapper, [127](#)
  - ORM, [126](#)
  - primary benefits, [127](#)
  - startup class

## INDEX

Entity Framework Core (EF Core) (*cont.*)  
dependency injection, [140, 141](#)  
directives, [141](#)  
features, [139](#)  
.NET Core project templates, [139](#)  
services container, [142](#)  
source code, [140](#)

### Environments

appsettings.Development.json, [177](#)  
Appsettings.json, [179](#)  
ASPNETCORE\_ENVIRONMENT, [172](#)  
broad approaches, [173](#)  
code-based determination, [174](#)  
commandName, [172](#)  
configuration sources/  
    preference, [174, 175](#)  
database connection error, [178](#)  
distinction, [173](#)  
LaunchSettings.json File, [171-174](#)  
learning goals, [167](#)  
production/development, [168](#)  
refactor existing code, [167](#)  
setup, [173](#)  
wrong credentials, [177, 178](#)  
Environments/user secrets, [168, 169](#)

## F

Fully Qualified Domain Name  
(FDQN), [368](#)

## G, H

### GetAllCommands

arrange/action/assert, [274](#)  
attributes, [274](#)  
CommandReadDtos, [289](#)  
controller (groundwork)

AutoMapper/Moq, [277](#)  
command model, [275](#)  
CommandsControllerTests.cs  
    file, [276](#)  
mocking frameworks, [277](#)  
mock-up AutoMapper, [281-283](#)  
Moq, [278-281](#)

HTTP Response code, [288](#)

HTTP response (Empty DB), [283-286](#)  
return type, [289, 290](#)  
single resource returned, [286-288](#)

### GetCommandByID

attributes, [290](#)  
correct object type, [293](#)  
HTTP response, [291-293](#)  
single resource, [291](#)

### GIT version

installation, [13-15](#)  
name and email, [14](#)

Globally unique identifier (GUID), [11](#)

## I

Integrated development environment  
(IDE), [9](#)

## J, K, L

### JavaScript Object

#### Notation (JSON)

arrays, [30](#)  
attributes, [27, 30](#)  
editor online, [28](#)  
meaning, [26](#)  
nested objects, [28](#)  
object, [27](#)  
object navigation, [29](#)  
respective values, [27](#)



**M**

Model and repository classes, 85

- data annotations, 87–89
- folder and command class, 86–88

Model–View–Controller (MVC) pattern

- coding information
  - command-line type, 49
  - ConfigureServices/Configure methods, 52
  - configure method, 50
  - framework, 51
  - web browser, 53
  - web browser and navigation, 50
  - webserver, 49
- concepts, 59
- decoupling option, 59
- explanation, 56
- Git/GitHub, 70
- GitHub repository
  - file and folder, 83
  - instruction code, 83
  - landing page, 77, 78
  - repositories, 78–82
  - setup, 77
- interface/contract, 59
- local Git repository
  - .gitignore file, 74–76
  - solution directory, 71
  - track/commit, 75–77
  - untracked files, 72
- models, 58
- Postman, 53–55
- source control, 69–71
- VS code setup, 48

PUT request

- attributes, 227
- AutoMapper profile, 231
- CommandUpdateDto, 230–232

- controller action, 232
- data annotations, 237
- DB context works, 230
- idempotent, 228
- input object, 227
- invalid resource ID, 238
- nonexistent resource, 237
- outputs, 228
- Postman setup, 235
- repository level, 228–230
- SqlCommandAPIRepo
  - implementation, 229
- testing, 235, 236
- UpdateCommand interface
  - method, 229
- update endpoint, 226
- validation error, 236, 237

**N**

- .NET Core Framework, 3
  - benefits, 5
  - .NET core version SDK, 12, 13
- Nonsense User ID/Password, 164

**O**

- Object Relational Mapper (ORM), 126

**P**

- PartialCommandUpdate method, 300–302
- PATCH verb updates
  - attributes, 238
  - command/return, 247
  - CommandUpdateDto, 242
  - concepts, 239
  - controller, 244–247
  - dependencies, 242, 243

## INDEX

### PATCH verb updates (*cont.*)

- document, 246
- high-level definition, 238
- HttpPatch, 246
- idempotent, 241
- input object, 240
- JsonPatchDocument, 246
- patch document, 241
- repository, 242
- serializer settings, 244
- startup class, 243
- testing option, 248, 249
- validation, 247

### Pipeline

- authenticate, 317
- Azure DevOps
  - cloud-based alternatives, 309
  - features, 308
  - landing page, 312
  - non-Microsoft mix, 311
  - technology overlay, 309–311
- azure-pipelines.yml, 319, 320
  - automation opportunities, 325
  - editing option, 326
  - GitHub commit, 327
  - GitHub repository/refresh, 325
  - navigation, 326
  - triggers, 327
- build step-up, 321
- completion screen, 323
- creation, 315
- DevOps, 307
- features, 314
- focus, 308
- GitHub code source, 316
- GitHub repo, 322
- in-progress, 323
- job preparation, 323

- landing page, 314
- loop, 308
- manual save and run, 322
- project, 312
- public visibility selection, 313
- recap, 324
- repositories, 317, 318
- templates, 318

### PostgreSQL database

- application architecture
  - progress, 114, 115
- Azure resources
  - container instance, 361–368
  - postgres server, 360
  - search option, 361
- current state vs. end state, 154

### DB context class, 153

### DBeaver

- configuration settings, 121, 122
- connection, 120, 121
- connection issues, 126
- databases, 122, 123
- DB Beaver, 125
- test connection, 123–125

### Docker

- command line arguments, 118–120
- container status, 117
- downloaded and running, 115, 116
- postgres image, 119
- PS command, 116

### EF Core (*see* Entity Framework Core (EF Core))

### learning goals, 113

### repository implementation

- API working, 161
- existing commands, 161, 162
- concrete implementation, 155, 159
- error handling, 163

- ICommandAPIRepo statement, 155
  - interface implementation code, 156
  - login and password, 164–166
  - mock implementation, 156, 160
  - SqlCommandAPIRepo, 157
  - startup class, 157
  - System.Linq, 158
- Production environments
  - legacy systems, 168
  - PR, 168
  - setup, 169
- Q**
- Query window, 149
- R**
- Repository
  - architecture, 90
  - delete endpoint, 249
  - DI (*see* Dependency Injection (DI))
  - endpoints, 211–214
  - implementation
    - concrete class, 95
    - concrete classes, 94
    - dependency injection, 98
    - methods, 98
    - mock data code, 94
    - placeholder code, 96, 97
    - resolution, 96
  - interface
    - CRUD actions, 91
    - data folder, 92
    - ICommandAPIRepo, 93
    - ICommanderRepo.cs file, 92
    - source code, 93
    - specification, 90
  - model (*see* Model and repository classes)
  - PUT request, 228–230
- Representational State Transfer (REST),
  - see* Application programming interface (API)
- S**
- Scaffold API solution, 31
  - API project generation, 34
  - ASP.NET, 41 (*see* ASP.NET Core project)
  - components, 33
  - contents, 35
  - folder setup, 33
  - project associations
    - build creation, 40
    - command, 37
    - references, 39
    - solution file, 37, 38
    - terminal/command line, 37
  - project templates, 34
  - solution hierarchy, 32–34
  - terminal window, 33
  - unit test project, 36
- Software development toolkit (SDK), 13, 14
- T**
- Thin and wide approach, 2
- Third edition
  - code selection, 3
  - contact information, 4
  - defects/features, 4
  - introduction, 1
  - thin and wide approach, 2

## U

## Unit testing

- act, [257](#)
- approaches, [268–272](#)
- arrange, [257](#)
- assertion, [257](#)
- azure-pipelines.yml file
  - dashboard, [337](#)
  - details, [337](#)
  - pipeline triggers, [335](#)
  - pipeline view, [333](#)
  - steps, [334](#)
  - testing step, [336](#)
- build pipeline, [342](#)
- characteristics, [255, 272](#)
- CI/CD pipelines, [338](#)
  - CommandAPI.Tests project, [338](#)
  - dashboard, [342](#)
  - failing result, [338, 339](#)
  - failures details, [341](#)
  - in-progress pipeline, [340](#)
  - output result, [340](#)
- Command.Tests folder, [258](#)
- constructor, [273](#)
- controller, [272](#)
- DeleteCommand, [302–304](#)
- dependencies, [273](#)
- developers, [256](#)
- executable documentation, [255](#)
- frameworks, [256](#)
- GetAllCommands (*see* GetAllCommands)
- GetCommandByID (*see* GetCommandByID)
- learning goals, [253](#)
- model class
  - API project, [261](#)

- CanChangeHowTo method, [263](#)
    - CommandAPI.Tests project, [262](#)
    - failure response, [267](#)
    - forcing test failure, [267](#)
    - passes/fails, [266](#)
    - sections, [263–265](#)
  - .NET Core project templates, [257](#)
  - PartialCommandUpdate
    - method, [300–302](#)
  - passing tests, [272](#)
  - protection against regression, [254](#)
  - pyramid, [253, 254](#)
  - refactored model tests, [271](#)
  - regression defects, [254](#)
  - running tests, [260](#)
  - standard class definition, [259](#)
  - testCommand object, [269](#)
  - UpdateCommand, [297–299](#)
  - xUnit project, [258, 259](#)
- UpdateCommand unit test
- attributes, [297, 298](#)
  - HTTP response, [298, 299](#)
- User Secrets
- Appsettings.json, [190](#)
  - CommandAPI.csproj file, [181](#)
  - configuration elements, [180](#)
  - ConfigureServices
    - method, [186–188](#)
  - connection string, [185](#)
  - .CSPROJ file, [183](#)
  - GUID insertion, [182, 183](#)
  - learning goals, [167](#)
  - precedence, [189](#)
  - secrets.json file, [180, 183](#)
  - sensitive connection string
    - attributes, [186](#)
  - setup, [181–183](#)
  - source code, [180](#)

- startup class, [188](#)
- updated sections, [187](#)
- user ID and password, [185](#)
- Windows, [185](#)

## V

- Visual Studio code
  - C# extension, [11](#)
  - downloading, [10](#)

- GUID extension, [11](#)
- installation, [8](#)

## W, X, Y, Z

- Windows Subsystem for Linux (WSL), [16](#)