



The Complete ASP.NET Core 3 API Tutorial

Hands-On Building, Testing,
and Deploying

—
Les Jackson

Apress®

The Complete ASP.NET Core 3 API Tutorial

**Hands-On Building, Testing,
and Deploying**

Les Jackson

Apress®

The Complete ASP.NET Core 3 API Tutorial: Hands-On Building, Testing, and Deploying

Les Jackson
Melbourne, VIC, Australia

ISBN-13 (pbk): 978-1-4842-6254-2
<https://doi.org/10.1007/978-1-4842-6255-9>

ISBN-13 (electronic): 978-1-4842-6255-9

Copyright © 2020 by Les Jackson

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Joan Murray
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484262542. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

For Quynh

Table of Contents

About the Author	xvii
About the Technical Reviewer	xix
Acknowledgments	xxi
Chapter 1: Introduction.....	1
Why I Wrote This Book	1
Apress Edition	1
The Approach of This Book	2
Where Can You Get the Code?.....	2
Main Solution Repository (API and Unit Tests).....	3
Secure Daemon Client Repository	3
Conventions Used in This Book	3
Version of the .net Core Framework	3
Contacting the Author	4
Defects and Feature Improvements	4
Chapter 2: Setting Up Your Development Environment.....	5
Chapter Summary	5
When Done, You Will	5
The Three Amigos: Windows, Mac, and Linux	5
Your Ingredients	6
Links to the Software and Sites	8
Install VS Code	8
C# for Visual Studio Code	10
Insert GUID.....	11

TABLE OF CONTENTS

- Install .NET Core SDK 12
- Install GIT 13
 - Name and Email 14
- Install Docker [Optional]..... 15
 - What Is Docker? 15
 - Docker Desktop vs. Docker CE 15
 - Post-installation Check..... 16
 - Docker Plugin for VS Code 18
- Install PostgreSQL..... 18
- Install DBeaver CE..... 19
 - DBeaver vs. pgAdmin 19
- Install Postman 20
- Trust Local Host Development Certs 21
- Wrapping It Up 22
- Chapter 3: Overview of Our API 23**
 - Chapter Summary 23
 - When Done, You Will 23
 - What Is a REST API? 23
 - Our API 24
 - Payloads..... 26
 - Five Minutes On JSON 26
- Chapter 4: Scaffold Our API Solution 31**
 - Chapter Summary 31
 - When Done, You Will 31
 - Solution Overview 31
 - Scaffold Our Solution Components 33
 - Creating Solution and Project Associations 36
 - Anatomy of An ASP.NET Core App..... 41
 - The Program and Startup Classes 43

Chapter 5: The “C” in MVC	47
Chapter Summary	47
When Done, You Will	47
Quick Word on My Dev Setup	48
Call the Postman	53
What Is MVC?	56
Model–View–Controller	57
Models, Data Transfer Objects, Repositories, and Data Access.....	58
Our Controller.....	60
1. Using Directives.....	65
2. Inherit from Controller Base	65
3. Set Up Routing.....	66
4. ApiController Attribute	67
5. HttpGet Attribute.....	67
6. Our Controller Action	68
Source Control	69
Git and GitHub.....	70
Setting Up Your Local Git Repo.....	71
.gitignore file	72
Track and Commit Your Files	75
Set Up Your GitHub Repo.....	77
Create a GitHub Repository	78
So What Just Happened?	82
Chapter 6: Our Model and Repository	85
Chapter Summary	85
When Done, You Will	85
Our Model.....	85
Data Annotations	87

TABLE OF CONTENTS

- Our Repository 89
 - What Is an Interface?..... 90
 - What About Implementation? 94
- Dependency Injection..... 99
 - Back to the Start (Up) 99
 - Applying Dependency Injection 102
- Chapter 7: Persisting Our Data 113**
 - Chapter Summary 113
 - When Done, You Will 113
 - Architecture Progress Check 113
 - PostgreSQL Database 115
 - Using Docker 115
 - Docker Command Prompt 118
 - Connecting with DBeaver 120
 - Entity Framework Core 126
 - The What and Why of ORMs..... 127
 - Entity Framework Command-Line Tools..... 127
 - Create Our DB Context..... 128
 - Update appsettings.json..... 132
 - Revisit the Startup Class 139
 - Create and Apply Migrations 143
 - Adding Some Data 148
 - Tying It Altogether 153
 - Create a New Repository Implementation 154
 - Get All Command Items 161
 - Get A Single Command (Existing) 161
 - Get A Single Command (Not Existing)..... 162
 - Wrapping Up the Chapter 163
 - Redact Our Login and Password 164

Chapter 8: Environment Variables and User Secrets	167
Chapter Summary	167
When Done, You Will	167
Environments	167
Our Environment Setup	169
The Development Environment	170
So What?	173
Make the Distinction.....	173
Order of Precedence.....	174
It's Time to Move	176
User Secrets.....	180
What Are User Secrets?.....	180
Setting Up User Secrets.....	181
Deciding Your Secrets.....	183
Where Are They?.....	184
Code It Up	185
Wrap It Up	189
Chapter 9: Data Transfer Objects	191
Chapter Summary	191
When Done, You Will	191
Architecture Review	191
The What and Why of DTOs	193
Decouple Interface from Implementation (Again).....	193
Implementing DTOs.....	195
Create Our DTOs	196
Setting Up AutoMapper.....	197
Using AutoMapper	199

TABLE OF CONTENTS

- Chapter 10: Completing Our API Endpoints 207**
 - Chapter Summary 207
 - When Done, You Will 207
 - Persisting Changes in EF Core 207
 - DB Context Tracks Changes 208
 - The Create Endpoint (POST) 209
 - Input Object 210
 - Success Outputs 210
 - Idempotency 210
 - Updating the Repository 211
 - CommandCreatedto 214
 - Updating the Controller 217
 - Manually Testing the Create Endpoint 222
 - The Update Endpoint #1 (PUT) 226
 - Input Object 227
 - Success Outputs 228
 - Idempotent 228
 - Updating the Repository 228
 - CommandUpdateDto 230
 - Updating the Controller 232
 - Manually Testing the Update (PUT) EndPoint 235
 - The Update Endpoint #2 (PATCH) 238
 - Input Object 240
 - Idempotent 241
 - Updating the Repository 241
 - CommandUpdateDto 242
 - Install Dependencies for PATCH 242
 - Updating the Startup Class 243
 - Updating the Controller 244
 - Manually Testing the Update (PATCH) EndPoint 247

The Delete Endpoint (DELETE).....	248
Updating the Repository.....	249
CommandDeleteDto.....	250
Updating the Controller.....	250
Wrap Up	251
Chapter 11: Unit Testing Our API	253
Chapter Summary	253
When Done, You Will	253
What Is Unit Testing	253
Protection Against Regression.....	254
Executable Documentation.....	255
Characteristics of a Good Unit Test	255
What to Test?	256
Unit Testing Frameworks	256
Arrange, Act, and Assert.....	257
Arrange.....	257
Act	257
Assert	257
Write Our First Tests.....	258
Testing Our Model.....	261
Don't Repeat Yourself.....	268
Test Our Controller	272
Revisit Unit Testing Characteristics	272
GetAllCommands Unit Tests and Groundwork.....	274
GetAllCommands Overview	274
GetAllCommands Unit Tests.....	274
Groundwork for Controller Tests.....	275
Finish Test 1.1 – Check 200 OK HTTP Response (Empty DB).....	283
Test 1.2 – Check Single Resource Returned.....	286
Test 1.3 – Check 200 OK HTTP Response.....	288
Test 1.4 – Check the Correct Object Type Returned	289

TABLE OF CONTENTS

- GetCommandByID Unit Tests..... 290
 - GetCommandByID Overview 290
 - GetCommandByID Unit Tests 291
 - Test 2.1 – Check 404 Not Found HTTP Response 291
 - Test 2.2 – Check 200 OK HTTP Response..... 292
 - Test 2.3 – Check the Correct Object Type Returned 293
- CreateCommand Unit Tests 294
 - CreateCommand Overview 294
 - CreateCommand Unit Tests 295
 - Test 3.1 Check If the Correct Object Type Is Returned..... 296
 - Test 3.2 Check 201 HTTP Response 296
- UpdateCommand Unit Tests 297
 - UpdateCommand Overview 297
 - UpdateCommand Unit Tests..... 298
 - Test 4.1 Check 204 HTTP Response 298
 - Test 4.2 Check 404 HTTP Response 299
- PartialCommandUpdate Unit Tests..... 300
 - PartialCommandUpdate Overview 300
 - PartialCommandUpdate Unit Tests 301
 - Test 5.1 Check 404 HTTP Response 301
- DeleteCommand Unit Tests 302
 - DeleteCommand Overview 302
 - DeleteCommand Unit Tests..... 303
 - Test 6.1 Check for 204 No Content HTTP Response 303
 - Test 6.2 Check for 404 Not Found HTTP Response..... 303
- Wrap It Up 304
- Chapter 12: The CI/CD Pipeline..... 305**
 - Chapter Summary 305
 - When Done, You Will 305

What Is CI/CD?	305
CI/CD or CI/CD?	306
What's the Difference?	306
So Which Is It?	307
The Pipeline	307
What Is Azure DevOps?.....	308
Alternatives	309
Technology in Context	309
Create a Build Pipeline.....	311
What Just Happened?	324
Azure-Pipelines.yml File.....	325
Triggering a Build.....	328
Revisit azure-pipelines.yml.....	331
Another VS Code Extension	332
Running Unit Tests.....	333
Breaking Our Unit Tests.....	338
Testing – The Great Catch All?	342
Release/Packaging	343
Wrap It Up	347
Chapter 13: Deploying to Azure	349
Chapter Summary	349
When Done, You Will	349
Creating Azure Resources.....	349
Create Our API App	350
Create Our PostgreSQL Server	359
Connect and Create Our DB User.....	368
Revisit Our Dev Environment	370

TABLE OF CONTENTS

- Setting Up Config in Azure 371
 - Configure Our Connection String 371
 - Configure Our DB User Credentials..... 374
 - Configure Our Environment 377
- Completing Our Pipeline 379
 - Creating Our Azure DevOps Release Pipeline 380
- Pull the Trigger – Continuously Deploy 388
 - Wait! What About EF Migrations? 388
 - Double-Check..... 392
- Chapter 14: Securing Our API 395**
- Chapter Summary 395
 - When Done, You Will 395
- What We’re Building..... 395
 - Our Authentication Use Case 395
 - Overview of Bearer Authentication..... 396
 - Build Steps 397
- Registering Our API in Azure AD 399
 - Create a New AD?..... 400
 - Register Our API..... 401
 - Expose Our API 406
 - Update Our Manifest..... 408
- Add Configuration Elements..... 411
- Update Our Project Packages 413
- Updating our Startup Class 413
 - Update Configure Services 413
 - Update Configure..... 415
- Update Our Controller 416
- Register Our Client App..... 418
 - Create a Client Secret..... 420
 - Configure API Permissions 422

Create Our Client App.....	427
Our Client Configuration	428
Add Our Package References	430
Client Configuration Class	431
Finalize Our Program Class	435
Updating for Azure	442
Client Configurations	446
Deploy Our API to Azure	447
Epilogue	449
Index.....	451

About the Author



Les Jackson is originally from Glasgow, Scotland, but has lived and worked in Melbourne, Australia, since 2009. Since completing his computer science degree in 1998, he has worked in IT, primarily in the telecommunications industry and with the incumbent national telecom providers. Les holds several industry accreditations and has reacquired a Microsoft Certified Solutions Developer certification, although he still believes there is no substitute for experience and passion and says, “beware of people touting certifications!” Aside from his day job, Les enjoys producing content for his YouTube channel and blog, where he hopes to grow his wonderful audience over the coming years. In his downtime he likes cycling, trying to grow vegetables, making (and drinking) beer, and traveling with his partner.

About the Technical Reviewer



As a freelance Microsoft technologies expert, **Kris van der Mast** helps his clients to reach their goals. Actively involved in the global community, he is a [Microsoft MVP](#) since 2007. First for [ASP.NET](#) and since 2016 achieving in two disciplines: Azure and Visual Studio and Development Technologies. Kris is also a Microsoft ASP Insider, Microsoft Azure Advisor, aOS ambassador, and a Belgian Microsoft Extended Experts Team (MEET) member. In the Belgian community, Kris is active as a board member

of the Belgian Azure User Group [AZUG](#) and is chairman of the Belgian User Group Initiative (BUG). Since he started with .NET back in 2002, he's also been active on the [ASP.NET forums](#) where he is also a moderator. His personal site can be found at www.krisvandermast.com. Kris is a public (inter)national speaker and is a co-organizer of the [CloudBrew](#) conference.

Personal note:

I enjoyed reviewing this book. It's easy to follow, and I liked the fact that unit tests were added to the story. The approach of using Docker, and how to set it up, gives this book that extra which the reader will find handy in her/his professional environment.

Acknowledgments

Writing this book (my first) has been a real eye-opener for me... I greatly underestimated the extent to which I would rely on other people (either directly or indirectly) to inspire, encourage, and just generally help me to finish it. So, in true “Oscars style,” and in no particular order, I’d like to thank the following groups of people in helping to bring this book into the world. Without them, this book would not exist.

For their good humor, endless support, and indulgence of me, I’d like to thank my friends, family, and wonderful partner (to whom this book is dedicated).

For their patience, support, and belief in a first-time author, I’d like to express sincere thanks to the wonderful, professional editorial staff at Apress.

For their insights, time, and willingness to share their knowledge, I’d like to thank the fantastic community of C#/.NET professionals.

And finally, along with the countless others that have read my blog or watched my YouTube channel, I’d like to thank you – the reader of this book. You may never know just how significant supporting me in this way has been...

CHAPTER 1

Introduction

Why I Wrote This Book

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

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

Apress Edition

Prior to publishing this book now with Apress, I had released two earlier editions of the book. Having taken a Lean Startup approach (releasing versions as is when they were ready), I received feedback on each of those to make each successive version better. With the release of .NET Core 3.1 in November 2019, it seemed like the perfect time to release the second edition which was updated for that version of the framework, as well as some other updates, primarily a move to PostgreSQL as the backend Database.

This Apress edition sees the introduction of the use of Data Transfer Objects (DTOs), as well as the use of the Repository Pattern, both of which speak to the idea of decoupling interfaces from implementation, which has a range of benefits as you will see. I’ve also added an endpoint to our example API that responds to the “PATCH” verb, which allows us to perform partial updates on resources. This was a sorely missing component from the previous versions of the book and was long overdue for inclusion.

¹Fans of *Peep Show*, I took this term from one of my favorite episodes of Season 9: www.imdb.com/title/tt2128665/?ref_=ttep_ep4

The Approach of This Book

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

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

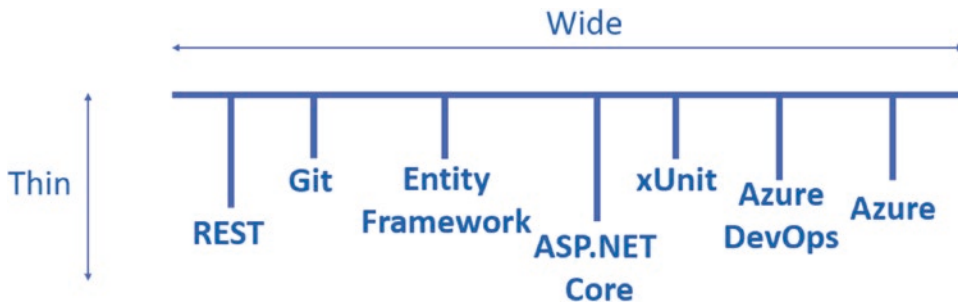


Figure 1-1. *Thin and wide approach*



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

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

Where Can You Get the Code?

While I think you'll get more value by following along throughout the book and typing in the code yourself (the book has been written so you can follow along step by step), you may of course prefer to download the code and use that as a reference. Indeed, as there

may be errata (heaven forbid!), it's prudent that I provide a repository for you, so you can just head over to GitHub and get the code there.

Main Solution Repository (API and Unit Tests)


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

Secure Daemon Client Repository

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

Conventions Used in This Book

The following style conventions are used in this book.

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

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

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

 **Celebration Checkpoint** Good job, milestone, worth calling out. Allows you to reflect and check learning.



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

Version of the .net Core Framework

At the time of writing (May 2020), I'm using version 3.1 of the .NET Core Framework.

Contacting the Author

You can contact me through the following channels:

- les@dotnetplaybook.com
- <https://dotnetplaybook.com/>
- www.youtube.com/binarythistle

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

Defects and Feature Improvements

Defects (errata) and suggestions for improvement should be sent to les@dotnetplaybook.com

Any corrections, additions, or improvements to the code will be reflected in the GitHub Repository.

CHAPTER 2

Setting Up Your Development Environment

Chapter Summary

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

When Done, You Will

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

The Three Amigos: Windows, Mac, and Linux

One of the benefits of the .NET Core Framework (when compared with the original .NET Framework) is that it's truly cross-platform,¹ meaning that you can develop and run the same apps on Windows, OSX (Mac), or Linux. For the vast majority of this book, the OS that you run on should make little difference in following along with the examples, so the choice of OS is almost irrelevant and of course entirely up to you.

¹Yes, there were things like "Mono," but overall, I'd say the original .NET Framework was Microsoft Windows-centric.

I've moved to PostgreSQL as the database backend which is available natively on Windows, Linux, and OSX. I will, however, be running it as a Docker container, but more of that later.

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

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

Note If there's any additional *nonstandard* config/setup required, I will of course cover that.

Your Ingredients

I'm going to assume you have the absolute basic things like a PC or Mac, a web browser, and an Internet connection (if not, you'll have to get all of those!), so the software I've listed below is the extra stuff you'll likely need to follow along.²

Ingredient	What is it?	Cost	Required for	Platform
VS Code	Cross-platform, fully featured text editor	Free	Writing code! Note: This is just my personal preference; you can of course choose an editor that you are more comfortable with	Cross-platform

(continued)

²Links to where you can locate the software have been provided separately in the section that follows.

Ingredient	What is it?	Cost	Required for	Platform
.NET Core SDK	.NET Core Runtime and SDK	Free	It's the framework we'll be building our API on. As mentioned in the opening, we'll use 3.1 in this book	Cross-platform
Git	Local source Code control	Free	Local source control and pushing our code to GitHub for eventual publishing to Azure	Cross-platform
PostgreSQL	Local database	Free	We'll use this as our local development/test database	Cross-platform or Docker image
DBeeer CE	Database-independent management tool	Free	Writing and executing SQL queries, setting up DB users, etc.	Cross-platform
Postman	API Testing Tool	Free	You can opt to use a web browser to test our API; Postman just gives us more options and is highly recommended	Cross-platform
Docker Desktop/ Docker CE	Containerization platform (run Docker containers)	Free	[Optional] I use Docker to quickly spin up and run a PostgreSQL database without the need to install it (PostgreSQL) locally on my desktop	Cross-platform: Docker Desktop – Windows and OSX Docker CE – Linux
GitHub.com	Cloud-based git repository used for team collaboration	Free	Used as the code repository component of our continuous integration/continuous delivery (CI/CD) pipeline	N/A – browser-based
Azure	The Microsoft cloud services offering	Free ³	We'll use Azure to host our production API as well as our “production” PostgreSQL Database	N/A – browser-based

(continued)

³At the time of writing new, sign-ups get \$280USD credit (to use within first 30 days), with an additional 12 months of “popular” services free. Other charges may be applicable though; please check the Azure website for the latest offer: <https://azure.microsoft.com/>

Ingredient	What is it?	Cost	Required for	Platform
Azure DevOps	Cloud-based build/test/deployment platform	Free	We use Azure DevOps primarily as the vehicle to publish our API to Azure. We will also leverage its centralized build/test features	N/A – browser-based

Links to the Software and Sites

- **VS Code:** <https://code.visualstudio.com/download>
- **.NET Core SDK:** <https://dotnet.microsoft.com/download>
- **Git:** <https://git-scm.com/downloads>
- **PostgreSQL (Native Install):** www.postgresql.org/download/
- **PostgreSQL (Docker Image):** https://hub.docker.com/_/postgres
- **DBeaver:** <https://dbeaver.io/download/>
- **Postman:** www.postman.com/
- **Docker Desktop (Windows and OSX):** www.docker.com/products/docker-desktop
- **Docker CE (Linux):** <https://docs.docker.com/get-docker/>
- **GitHub:** <https://github.com/>
- **Azure:** <https://portal.azure.com/>
- **Azure DevOps:** <https://dev.azure.com/>

Install VS Code

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

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

You do of course have other options, most notably Visual Studio,⁴ which is a fully integrated development environment (IDE) available on Windows and now OSX. If you don't want to use a full IDE, then there are a range of other text editors, for example, Notepad ++ on Windows, TextMate on OSX, etc., that you can use.



Les' Personal Anecdote I'm often asked why I choose to use VS Code over Visual Studio, and I always answer with the same analogy.

I compare it to learning to drive a manual transmission (aka "stick shift) vs. learning to drive a car with an automatic transmission. In my view, if you learn to drive a manual transmission, you can transfer to driving an automatic with relative ease. I don't think the reverse is as true.

Therefore, while VS Code can be a little more "involved" and may not do as much for you as Visual Studio, I think it just provides you with a better understanding of how things work. Once you get the hang of things though, Visual Studio is an incredible tool.

Anyway, to install VS Code, go to <https://code.visualstudio.com/download>, select your OS, (see Figure 2-1), and follow the provided instructions for your OS.

⁴The "free" version of Visual Studio is called the "Community Edition"; just Google it for the download site.

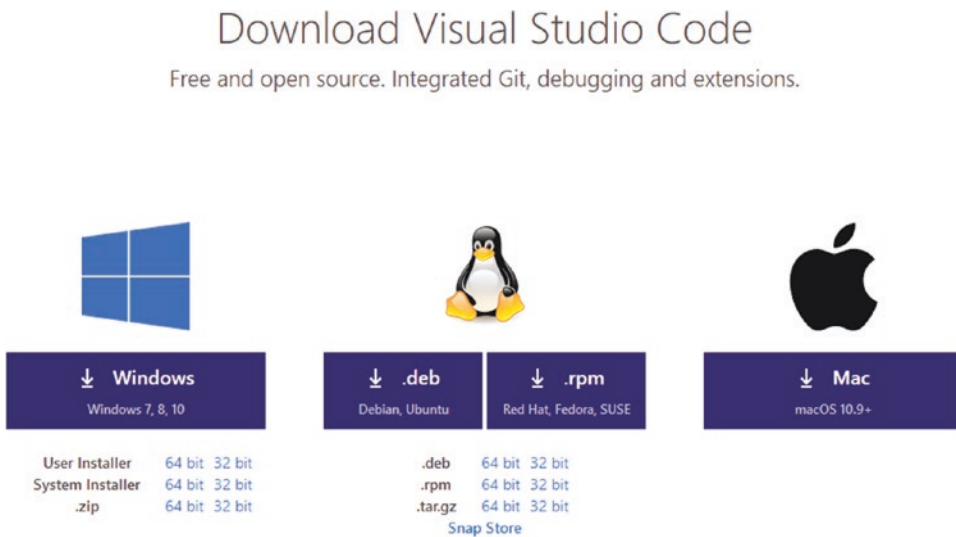


Figure 2-1. VS Code download

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

C# for Visual Studio Code

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

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

1. Click the “Extensions” icon in the left-hand toolbar of VS Code.
2. Type all or part of the name of the extension you want, for example, C#.
3. Click the name of the extension you'd like.

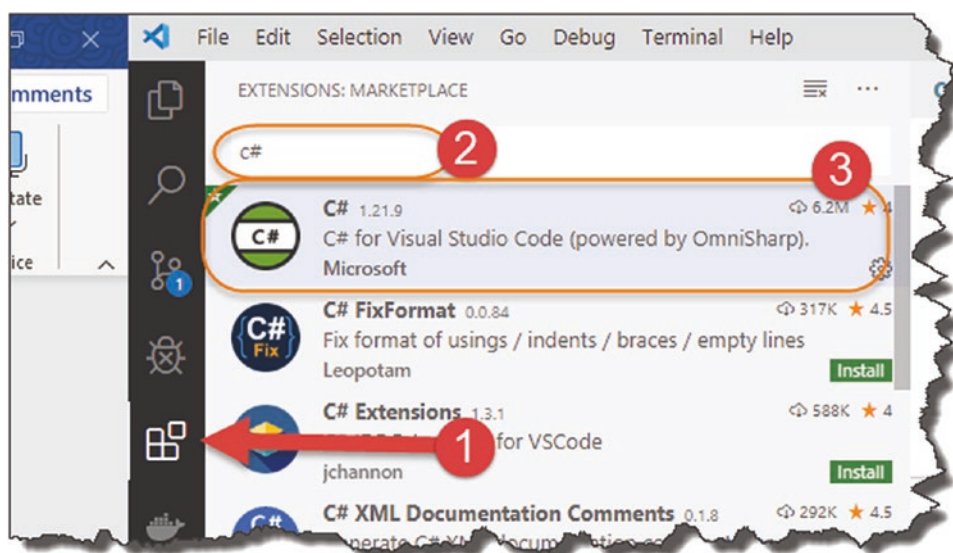


Figure 2-2. Install C# extension for VS Code

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

Insert GUID

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



Figure 2-3. Install Insert GUID extension for VS Code

🎓 Learning Opportunity Install the “Insert GUID” VS Code extension yourself – it’s not hard!

OK, we’re done with VS Code setup for now so let’s move on to the next install.

Install .NET Core SDK

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

```
dotnet --version
```

If installed, you should see something like this.

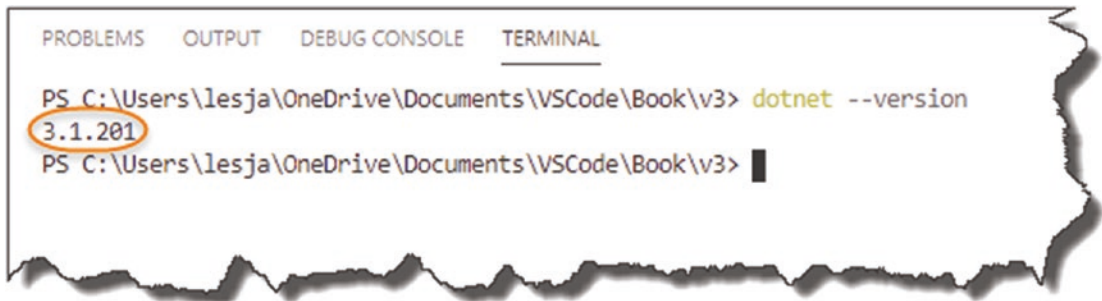


Figure 2-4. Check .NET Core Version

Even if it is installed, it’s probably worth checking to see what the latest version is to make sure that you’re not too far behind. From the screenshot in Figure 2-4, you can see I’m running 3.1 which at the time of writing is the latest version.

If it’s not installed (or you want to update your version), pop over to <https://dotnet.microsoft.com/download>, and select “Download .NET Core SDK,” as shown in the following figure.

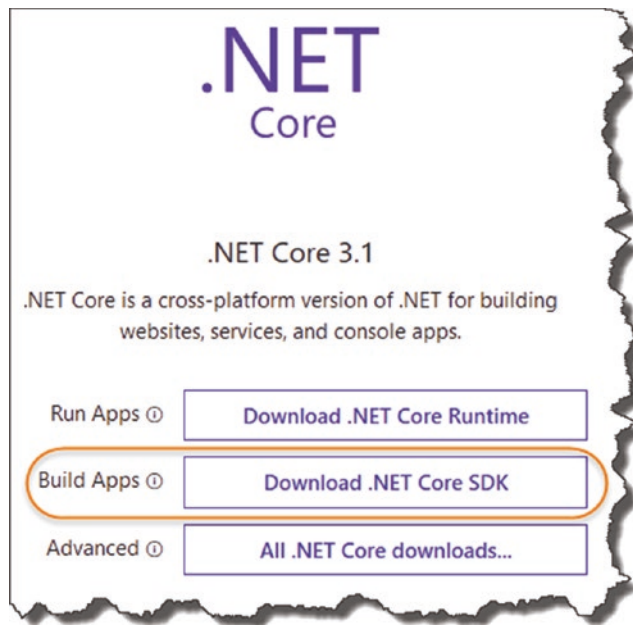


Figure 2-5. Download .NET Core SDK

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

As usual follow the respective install procedures for your OS; once completed, you should now be able to run the same `dotnet --version` command as shown in Figure 2-5, resulting in the latest version being returned.

Install GIT

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

At a command prompt/terminal, type

```
git --version
```

If already installed, you’ll see something similar to that shown in Figure 2-6.

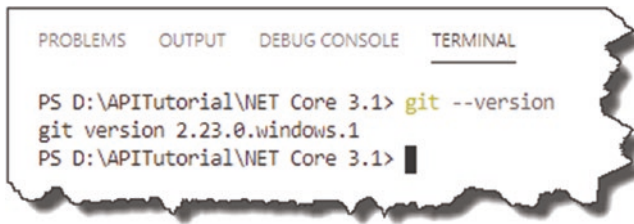


Figure 2-6. Check GIT version

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

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

Name and Email

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

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

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

For example see my configuration in Figure 2-7.



Figure 2-7. Configure GIT name and email

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

Install Docker [Optional]

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

What Is Docker?

Docker is a containerization platform that enables you to

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

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

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

Docker Desktop vs. Docker CE

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

- **Docker Desktop Here:** www.docker.com/products/docker-desktop
- **Docker CE Here:** <https://docs.docker.com/get-docker/>

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

⚠ Warning! At the time of writing, Docker Desktop can only be installed “directly” on Windows 10 Professional. However, if you’re running Windows 10 Home, you can work around this by using something called Windows Subsystem for Linux (WSL).

As I’ve said before, I’m not going to go into detail on how to do this as the Docker guys have provided great instructions for this here:

<https://docs.docker.com/docker-for-windows/install-windows-home/>.

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

Post-installation Check

Irrespective of which flavor of Docker you install, post-installation, open a command line, and type

```
docker --version
```

You should get something like the following.

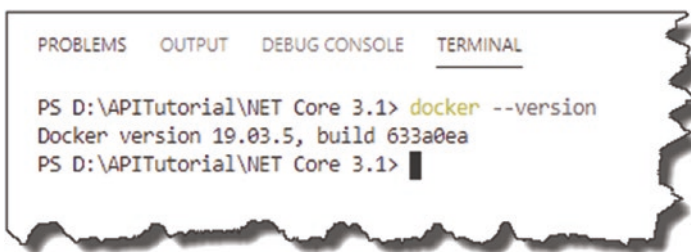


Figure 2-8. Check Docker version

To further test that it is fully working, type

```
docker run hello-world
```

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

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
-----  -
d-----  9/11/2019  11:00 AM  1 Raw Video Dump
PS D:\APITutorial\NET Core 3.1> docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:9572f7cdcee8591948c2963463447a53466950b3fc15a247fcad1917ca215a2f
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/

```

Figure 2-9. Hello World Docker image download and run

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

Docker Plugin for VS Code

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

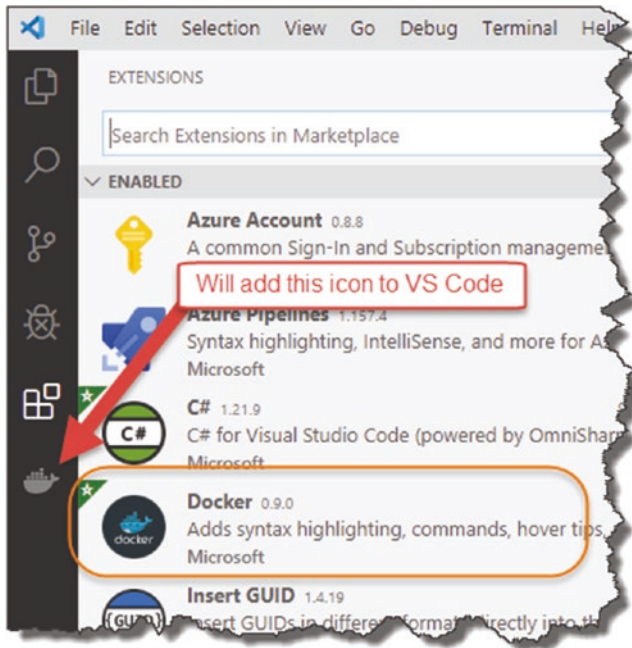


Figure 2-10. Docker extension for VS Code

Install PostgreSQL

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

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

For me though, struggling with the nuances of installing a DB detract from the act of coding, which is what I *really* want to be doing. Hence the reason why I *strongly* suggest the use of Docker.

Native Windows and OSX installations of PostgreSQL are (as usual) much easier.

Install DBeaver CE

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

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

DBeaver vs. pgAdmin

Probably the most popular admin tool for PostgreSQL is [pgAdmin](http://www.pgadmin.org/),⁵ and in fact this would have been the tool I’d have recommended previously.



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

⁵www.pgadmin.org/

Having looked at several graphical database management tools for PostgreSQL, I've landed on DBeaver Community Edition – which is free. This is a database agnostic management tool that you can use to connect to and manage most of the popular RDBMSs⁶ out there. It's also cross-platform, which is even better – you can download your copy here: <https://dbeaver.io/download/>.

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



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

The only reason I've not used it is simply because I've decided to use PostgreSQL as the RDBMS (you can only use SQL Server Management Studio to manage MS SQL Servers – it also only runs on Windows).

Install Postman

This is optional, and up to you if you want to install – but I *highly recommend* it. I'll be using it at various points throughout the book, and given that it's both free and excellent, I don't see why you wouldn't. If you're going to be doing API development going forward, then it's essentially mandatory. It's available as both a browser plugin or as a stand-alone client. For more details on how to install and download, go over to www.getpostman.com/downloads/, and take a look.

No further configuration is required at this point – I cover how to use it later.

⁶Relational Database Management Systems.

Trust Local Host Development Certs

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

To do that, at a command prompt, type

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

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



Figure 2-11. Trust local certificates

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

Wrapping It Up

All the other required components are Web-based and only require

- Web browser
- Internet connection
- User account

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

- **GitHub:** <https://github.com/>
- **Azure:** <https://portal.azure.com/>
- **Azure DevOps:** <https://dev.azure.com/>

All of which (at least initially!) are free.

CHAPTER 3

Overview of Our API

Chapter Summary

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

When Done, You Will

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

What Is a REST API?

APIs will eventually cure world hunger, bring about lasting peace, and enable mankind to explore the universe together, forever, in harmony¹ – or so some people (usually salesmen types) would have you believe. I of course don't believe that and am being somewhat facetious.


REST (or representational state transfer if you prefer) is an architectural style defined by Roy Fielding in 2000, that is used for creating web services. OK yes, but *what does that mean?* In short REST, or *RESTful* APIs, are a lightweight way to transfer textual

¹Credits to the late great Bill Hicks, whom I'm paraphrasing.

representations of “resources,” for example, books, authors, cars, etc. They are usually (although don’t need to be) built around the HTTP protocol and the standard set of HTTP verbs, for example, GET, POST, PUT, etc.

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

For me personally, actually building out the API is going to help you understand “REST” more fully than if I were to continue writing about it here, so we’ll leave the theory there for now. Be assured though that I do cover the central REST concepts as we build out our API endpoints.

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

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

Our API

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

Each “resource” will have the following attributes:

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

Here's a list of some snippets (aka "resources") as an example.

HowTo	Platform	Commandline
How to generate a migration in EF Core	.Net Core EF	dotnet ef migrations add <Name of Migration>
How to update the database (run migration)	.Net Core EF	dotnet ef database update
List Service Status - Linux	Ubuntu	service --status-all
Start a service ubuntu	Ubuntu	sudo service <service name> start
Stop a service Ubuntu	Ubuntu	sudo service <service name> stop
Restart a service Ubuntu	Ubuntu	sudo service <service name> restart
How to List all active migrations	.Net Core EF	dotnet ef migrations list
Roll back a migration	.Net Core EF	dotnet ef migrations remove
Create a Solution File	.Net Core CLI	dotnet new sln --name <Name of Solution>
Add a Project Reference to another project	.Net Core CLI	dotnet add <path to "host" project> reference <path to referenced project>
Add Projects to Solution File	.Net Core CLI	dotnet sln <Solution File> add <project1 .csproj file> <projectn .csproj file>
Override run command	Docker CLI	docker run <image name> command!
List running containers	Docker CLI	docker ps
List all containers that have ever run	Docker CLI	docker ps --all
Create a container from an image	Docker CLI	docker create <image name>

Figure 3-1. Example command-line snippets

Our API will follow the standard set of create, read, update, and delete (CRUD) operations common to most REST APIs, as described in the following table below.

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

Quick Note The Verb and URI *in combination* should be unique for a given API. We cover this in more detail later, but just make a mental note of that for now.

Payloads

As mentioned earlier, REST APIs are “a lightweight way to transfer *textual* representations of resources.” What do we mean by this?

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

- JavaScript Object Notation (JSON)
- Extensible Markup Language (XML)
- Hypertext Markup Language (HTML)
- Yet Another Markup Language (YAML)

and so on.

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

Five Minutes On JSON

What is JSON?

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

Anatomy of a Simple JSON Object

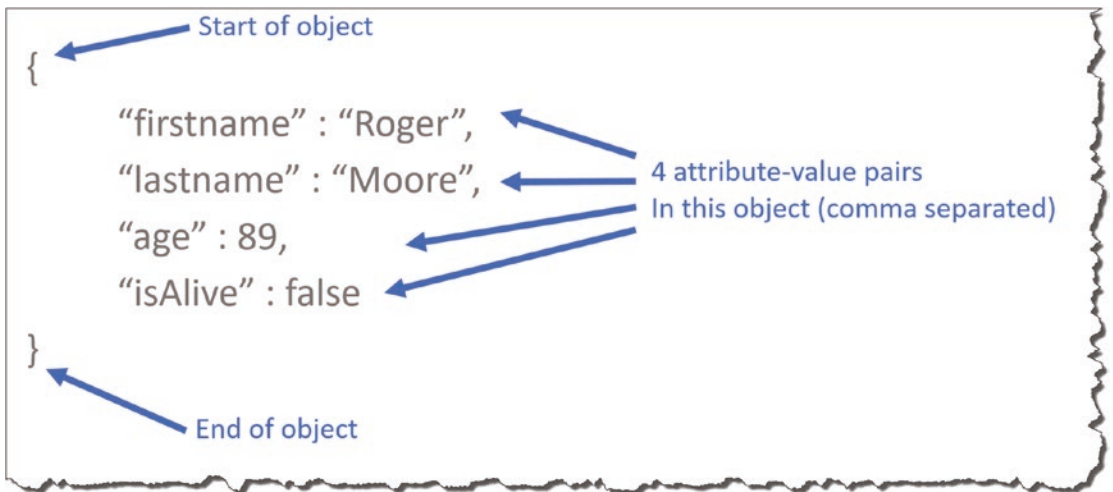


Figure 3-2. A Simple JSON object

In the example in Figure 3-2, we have a “Person” object with four attributes:

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

With the following respective values

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

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



Figure 3-3. JSON Editor Online

A (Slightly) More Complex Example

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

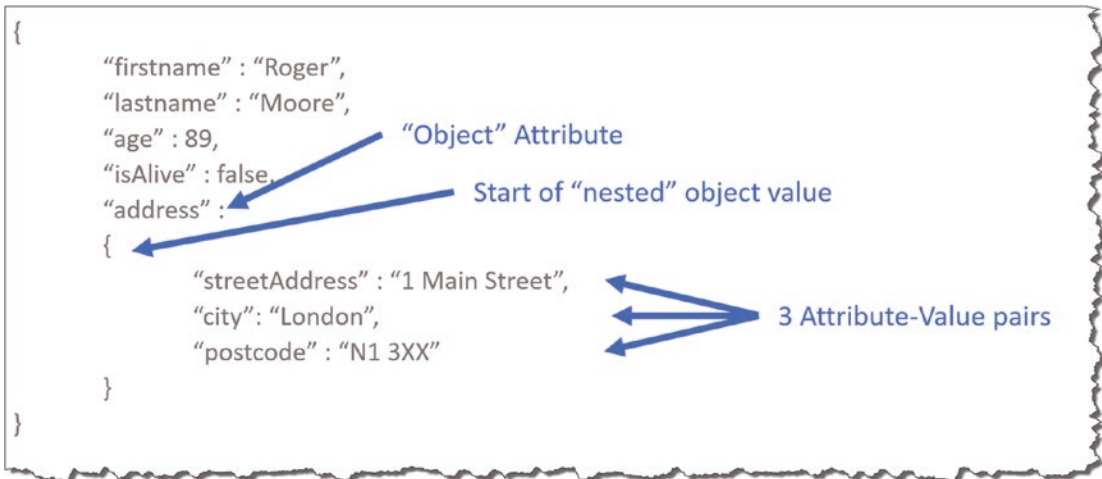


Figure 3-4. Nested JSON object

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

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

The values of all these attributes contain strings, so no need to labor that point further. This nesting can continue ad nauseum.

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

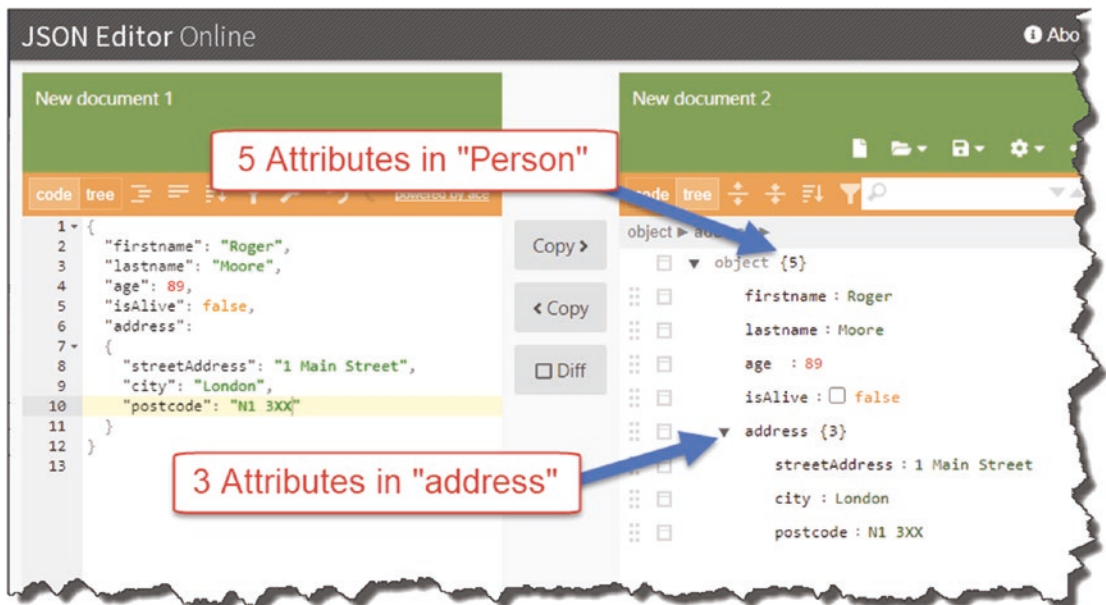


Figure 3-5. Object navigation in JSON Editor Online

A Final Example

On to our last example which this time includes an array of phone number objects.

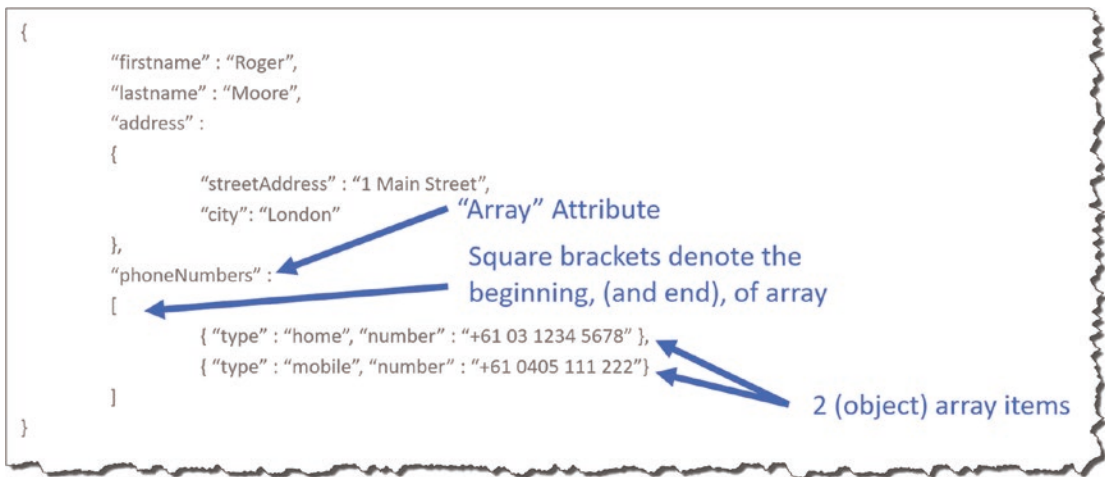


Figure 3-6. *Introducing JSON arrays*

Note I removed “age” and “isAlive” attributes from the person object as well as the “postcode” attribute from the address object purely for brevity and readability.

You’ll observe that we added an additional attribute to our Person object, “phoneNumbers”, and unlike the “address” attribute, it contains an array of other objects as opposed to just a single nested object.

The reason I chose these specific examples was to get you familiar with JSON and some of its core constructs, specifically

- The start and end of an object, “curly brackets”: { }
- Attribute–value pairs
- Nested objects (or objects as attribute values)
- Array of objects, “square brackets”: []

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

CHAPTER 4

Scaffold Our API Solution

Chapter Summary

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

When Done, You Will

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

Solution Overview

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

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

A “Solution” is really nothing more than a container for one or more related projects; projects in turn contain the code and other resources to do something useful. You would not put code directly into a Solution.

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

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

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

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

The overall layout for our solution is detailed in Figure 4-1.



Figure 4-1. Our Solution hierarchy

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

Scaffold Our Solution Components

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

- Create main “solution” folder called *CommandAPISolution*.
- Create two subdirectories called in solution folder called *src* and *test*.

You should have something like the following.

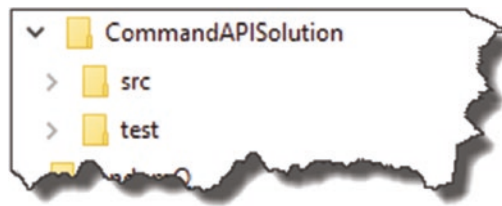


Figure 4-2. Basic folder setup

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

i .NET Core provides a number of “templates” we can use when creating a new project; selecting a particular template will impact any additional “scaffold” code automatically generated.

To see a list of the templates available, type
`dotnet new`

You should see something like the following.

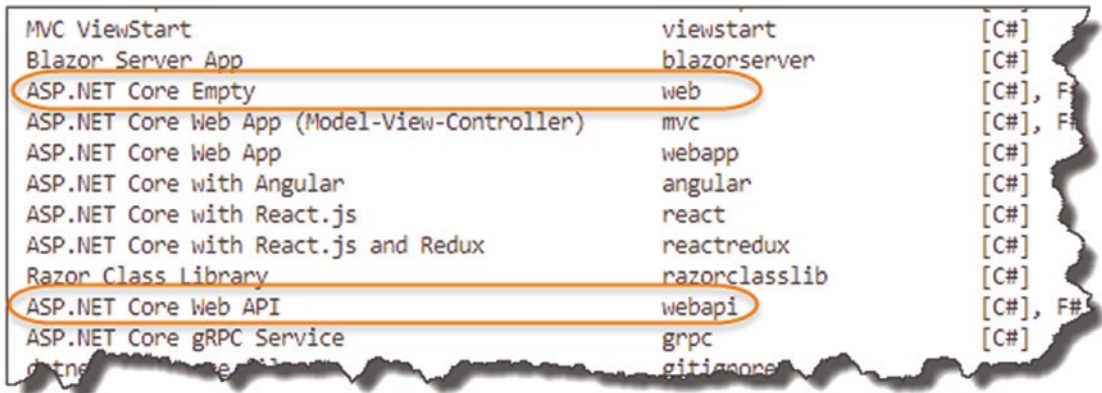


Figure 4-3. .NET Core Project templates

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

To generate our new “API” project, type (again ensure you are “inside” the *src* directory)

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

Where

- web is our template type.
- -n CommandAPI names our project and creates our project and folder.

You should see something like the following.

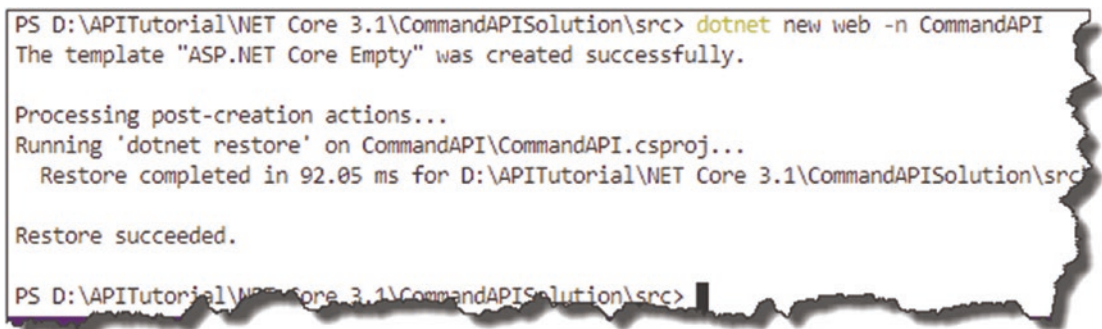


Figure 4-4. API Project generation

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

```

Change "into" the CommandAPI Folder
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\src> cd Com*
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI> ls

List the contents of the directory

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

Mode                LastWriteTime         Length Name
----                -
d-----            19/01/2020  12:22 PM             obj
d-----            19/01/2020  12:22 PM          Properties
-a-----            19/01/2020  12:22 PM    162 appsettings.Development.json
-a-----            19/01/2020  12:22 PM    192 appsettings.json
-a-----            19/01/2020  12:22 PM    148 CommandAPI.csproj
-a-----            19/01/2020  12:22 PM    718 Program.cs
-a-----            19/01/2020  12:22 PM   1290 Startup.cs

PS D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI>

```

Figure 4-5. Listing the contents of our API Project

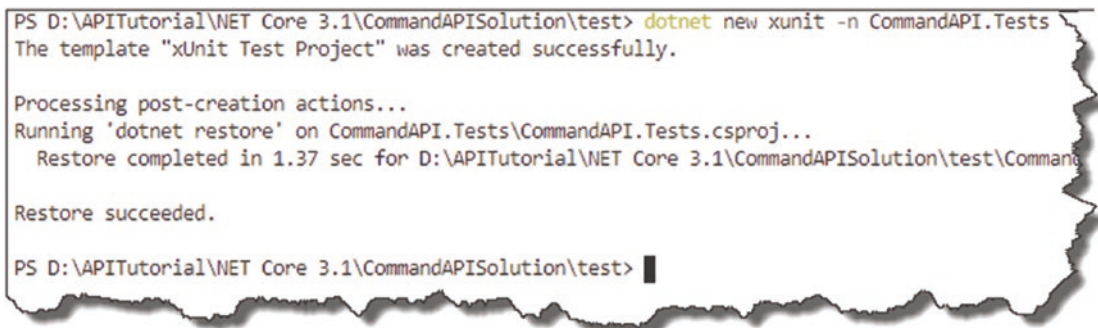
i If you're not familiar with navigating folders using a command-line interface, it may be worth Googling some basic commands. As I'm using a "PowerShell" terminal, the commands I used are similar to those you'd find on a Unix/Linux system. If you're using a Windows Command Prompt, you'd type `cd <name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

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

- Navigate into the *test* folder¹ contained in the main solution directory *CommandAPISolution*.
- At the command line, type

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

You should see the following output.



```
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\test> dotnet new xunit -n CommandAPI.Tests
The template "xUnit Test Project" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on CommandAPI.Tests\CommandAPI.Tests.csproj...
  Restore completed in 1.37 sec for D:\APITutorial\NET Core 3.1\CommandAPISolution\test\Command
Restore succeeded.

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

Figure 4-6. Unit Test Project creation

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

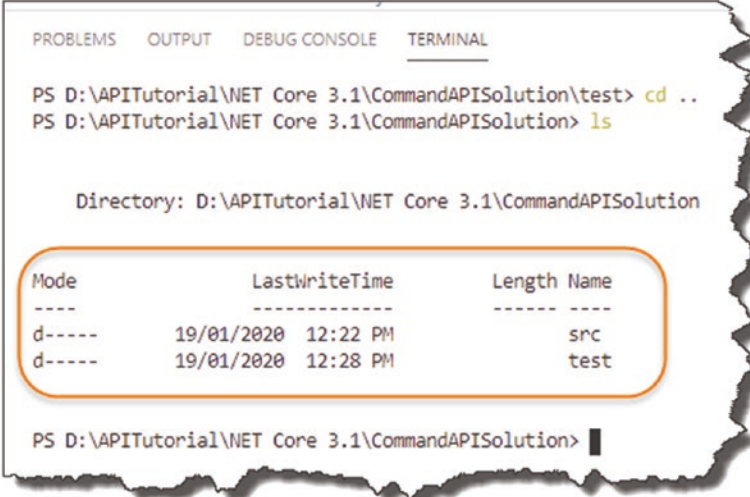
Creating Solution and Project Associations

OK, so we've created our two projects, but now we need to

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

¹Hint: `cd ..` moves you up a directory.

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



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

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

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

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

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

```

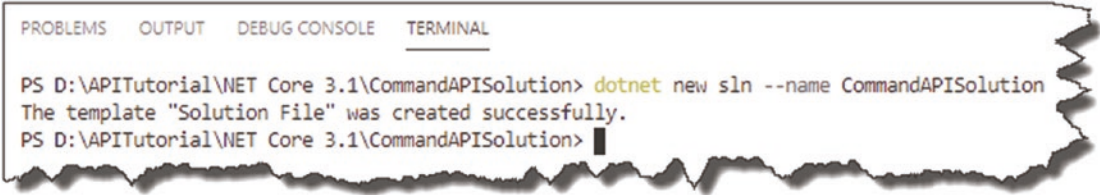
Figure 4-7. Check our directory listing

You should see the two directories: **src** and **test**.

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

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

This should create our *empty solution* file, as shown here.



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS D:\APITutorial\NET Core 3.1\CommandAPISolution> dotnet new sln --name CommandAPISolution
The template "Solution File" was created successfully.
PS D:\APITutorial\NET Core 3.1\CommandAPISolution>

```

Figure 4-8. Create the solution file

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

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

Note The preceding command is all one line.

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

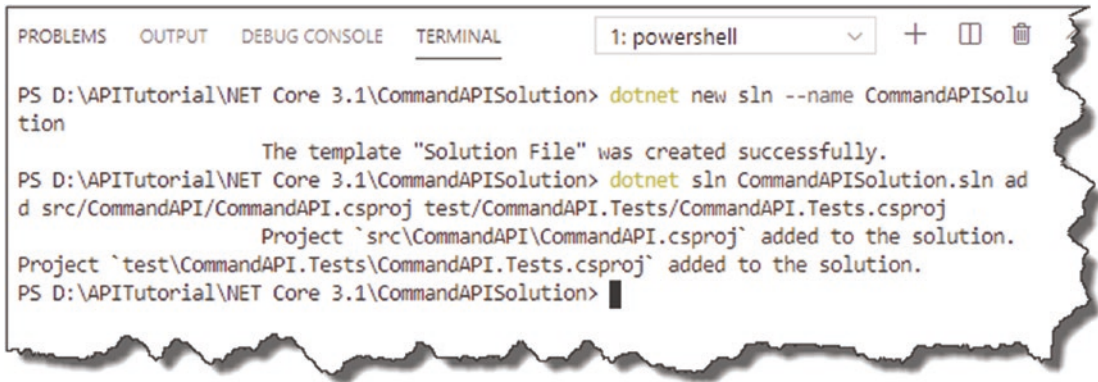


Figure 4-9. Projects added to our solution

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

All this really does is tell our solution that it has two projects. The projects themselves are unaware of each other. This is similar to a parent knowing that they have two children, but the children being *unaware* of each other – we're going to rectify that now, well for one of the siblings anyway.

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

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

You should get something like the following.



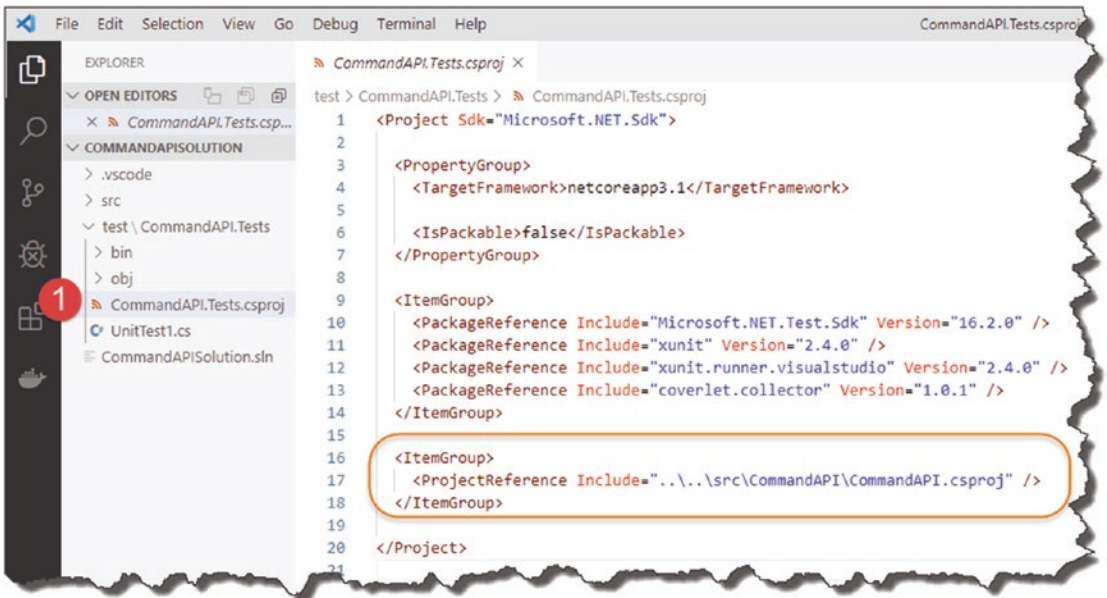
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: powershell
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> dotnet add test/CommandAPI.Tests/CommandAPI.Tests.csproj reference src/CommandAPI/CommandAPI.csproj
Reference '..\..\src\CommandAPI\CommandAPI.csproj' added to the project.
PS D:\APITutorial\NET Core 3.1\CommandAPISolution>

```

Figure 4-10. API Project added as a reference

Open VS Code (or whatever editor you chose), and open the *CommandAPISolution* folder²; find the *CommandAPI.Tests.csproj* file, and open it – you should see a reference (as well as other things) to the CommandAPI project:



```

File Edit Selection View Go Debug Terminal Help
CommandAPI.Tests.csproj
EXPLORER
OPEN EDITORS
CommandAPI.Tests.csp...
COMMANDAPISOLUTION
.vscode
src
test \ CommandAPI.Tests
bin
obj
1 CommandAPI.Tests.csproj
UnitTest1.cs
CommandAPISolution.sln
test > CommandAPI.Tests > CommandAPI.Tests.csproj
1 <Project Sdk="Microsoft.NET.Sdk">
2
3 <PropertyGroup>
4 <TargetFramework>netcoreapp3.1</TargetFramework>
5
6 <IsPackable>>false</IsPackable>
7 </PropertyGroup>
8
9 <ItemGroup>
10 <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.2.0" />
11 <PackageReference Include="xunit" Version="2.4.0" />
12 <PackageReference Include="xunit.runner.visualstudio" Version="2.4.0" />
13 <PackageReference Include="coverlet.collector" Version="1.0.1" />
14 </ItemGroup>
15
16 <ItemGroup>
17 <ProjectReference Include="..\..\src\CommandAPI\CommandAPI.csproj" />
18 </ItemGroup>
19
20 </Project>
21

```

Figure 4-11. Check reference has been added

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

²In VS Code got to File ► Open Folder and select your solution or project folder.

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

```
dotnet build
```

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

Assuming all is well, the *solution build* should succeed, which comprises our two projects.

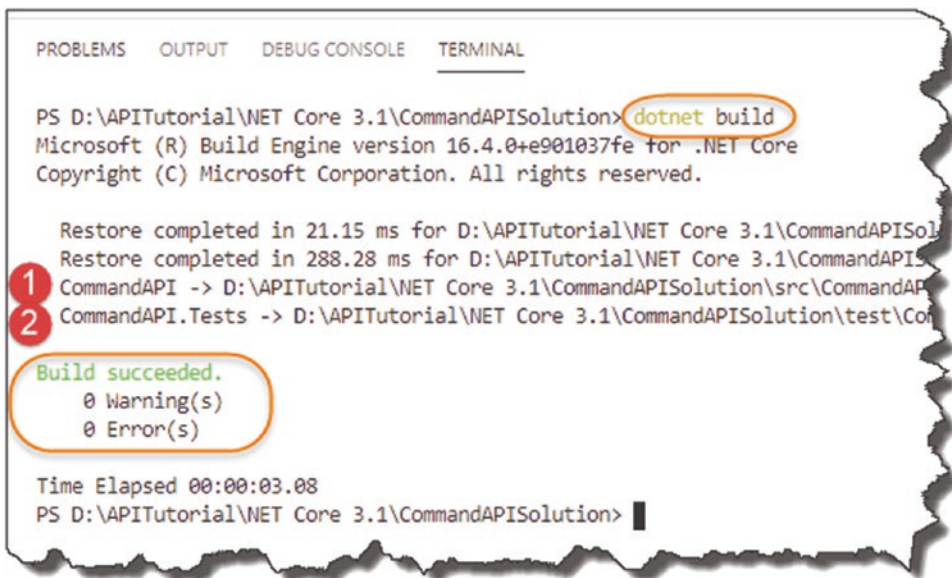


Figure 4-12. Perform our first build

🎉 Celebration Checkpoint Good Job! You’ve reached your first milestone; our app is scaffolded up and ready to rock and roll (that means coding).

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

Anatomy of An ASP.NET Core App

The following table describes the *core*³ files and folders that you will typically encounter when you create an ASP.NET Core project. Just be aware that depending on the project (or scaffold template) type you select, you may have additional files and folders – however, the ones described here are common to most project types.

File/folder	What is it?
.VS Code	This folder stores your VS Code workspace settings, so it's not really anything to do with the actual project. In fact, if you've chosen to dev this in something other than VS Code, you won't have this file (you may have something else)
<i>bin</i> (folder)	Location where final output binaries along with any dependencies and or other deployable files will be written to
<i>obj</i> (folder)	Used to house intermediate object files and other transient data files that are generated by the compiler during a build
<i>Properties</i> (folder) <i>launchSettings.json</i>	Contains the <i>launchSettings.json</i> file. This file can be used to configure application environment variables, for example, Development. It is also used to configure how the webserver running your app will operate, for example, which port it will listen on, etc.
<i>appsettings.json</i> <i>appsettings.</i> <i>Development.json</i>	File used to hold, surprise surprise, “application settings.” In the sections that follow, we'll store the connection string to our database here Also, environment-specific settings can be contained in additional settings files (e.g., Development) as shown by the <i>appsettings.Development.json</i> file
<ProjectName>. <i>csproj</i>	The configuration for the project principally tells us the .NET Core Framework version we're using along with other Nuget packages (see info box) that the application will reference and use Also, as you've previously seen, this is where we can place references to other projects that we need to be aware of
<i>Program.cs</i>	It all starts here This class configures the “hosting” platform, along with the “Main()” entry point method for the entire app

(continued)

³Core in this sense is pertaining to “part of something that is central to its existence or character,” not .NET Core.

File/folder	What is it?
<i>Startup.cs</i>	<p>This class is used to configure the application services and the request pipeline. More on those later.</p> <p>In my opinion, if you learn the workings of the <code>Startup</code> class, you'll be in a <i>really</i> good position to understand how ASP.NET Core applications work generally – so it's worth investing some effort here</p>

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

In short

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

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

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

The Program and Startup Classes

The Program Class

As previously mentioned, this is the main entry point for the entire app and is used to configure the “hosting” environment. It then goes on to use the Startup class to finalize the configuration of the app.

Let’s take a quick look at the templated code (which we’re *not* going to change) and see what it does.

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

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

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

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Figure 4-13. Standard contents of Program class

The execution sequence is as follows.

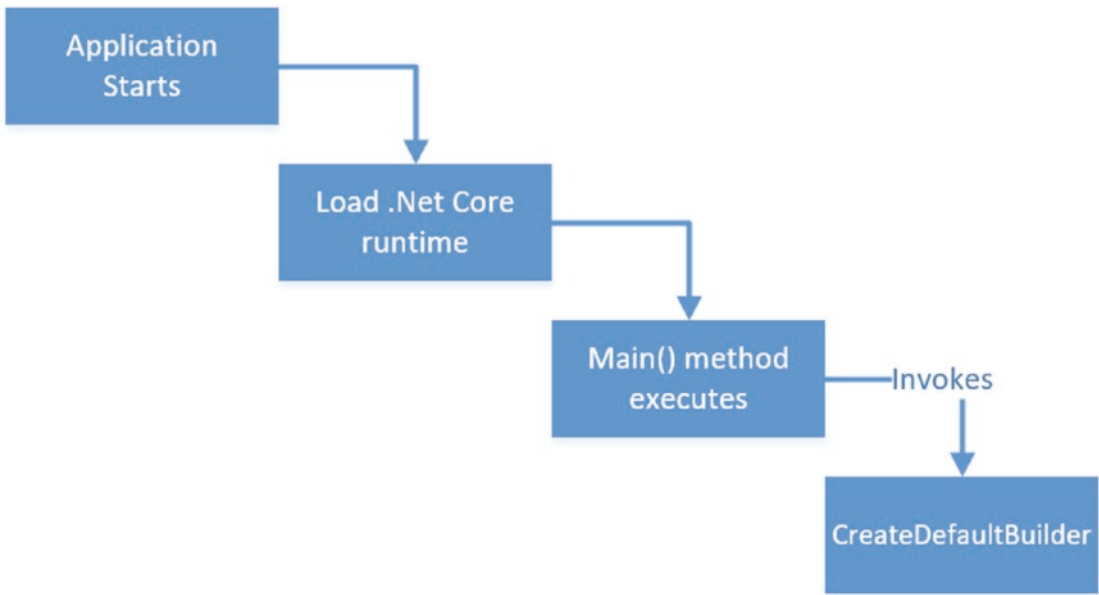


Figure 4-14. Program Class execution sequence

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

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

The Startup Class

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

- `ConfigureServices`
- `Configure`

The execution sequence is as follows.

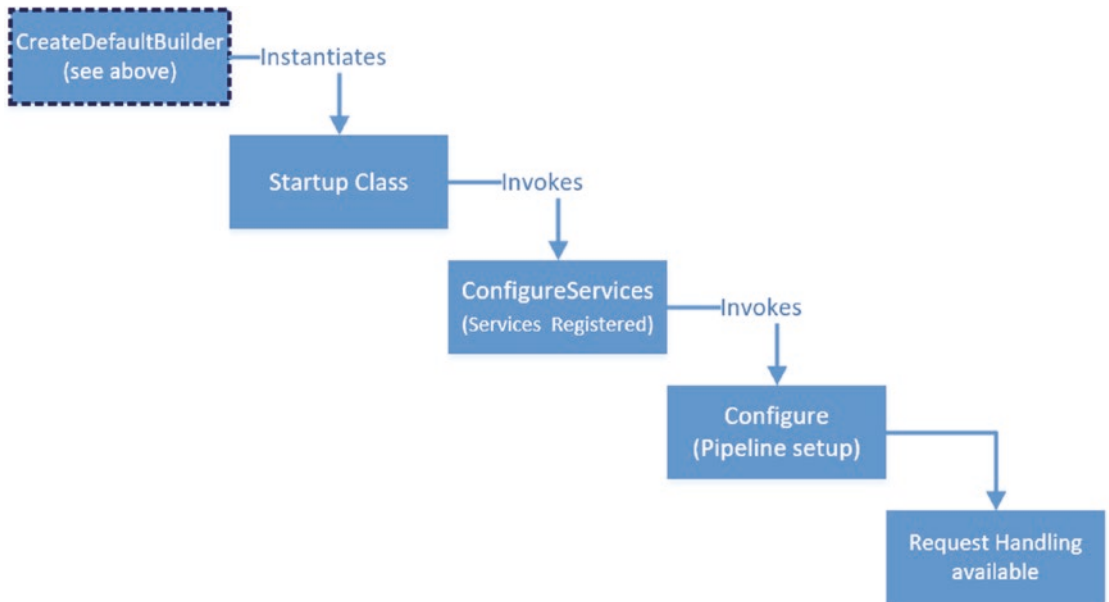


Figure 4-15. Startup class execution sequence

ConfigureServices

In ASP.NET Core we have the concept of “services,” which are just objects that provide functionality to other parts of the application. For those of you familiar with the concept of *dependency injection*,⁴ this is where dependencies are registered inside the default Inversion of Control (IoC) container provided by .NET Core. We’ll cover dependency injection in much more detail when we come to working with our “repository” in Chapter 6.

Configure

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

Depending on how the multiple middleware components are created, it will affect at what stage they get involved with the request and what (if anything) they do to impact it.

In the following diagram, you can see how a request would traverse the middleware components added in the `Configure` method. The nature of the request (e.g., is it an

⁴This can be a tricky subject; I cover this in some detail in Chapter 6 and throughout the tutorial.

attempt to open a Web Socket?) and the logic in the middleware will determine what will happen to that request, with the ability to “short-circuit” traversing further middleware if required (not shown).

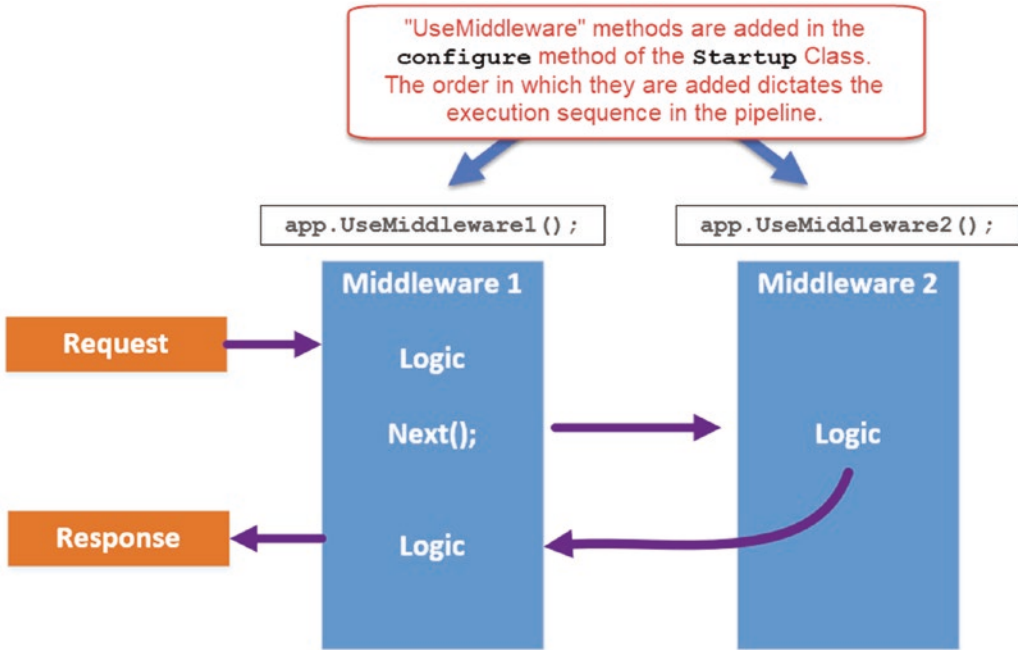


Figure 4-16. Very simple example of .NET Core Middleware

🎓 Learning Opportunity In the preceding diagram, I’ve use a generic construct to describe the middleware components – `app.UseMiddleware1()`, etc. Take a look at the `Configure` method in our `Startup` class, and have a look at the *actual* middleware components that are being added.

Hint They’ll start with “app.”

The *Request Pipeline* and middleware, in general, are an expansive area which could occupy a whole chapter of the book on its own. In keeping with the “thin and wide” approach, I feel we’ve covered enough to move on and start coding (we cover the Request Pipeline in more detail in Chapter 14 when we discuss Authentication and Authorization).

CHAPTER 5

The “C” in MVC

Chapter Summary

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

When Done, You Will

- Understand what the MVC pattern is
- Understand our API application architecture, including concepts such as
 - Repositories
 - Data transfer objects (DTOs)
 - Database contexts
- Add a controller class to our API project.
- Create a Controller Action (or API Endpoint if you prefer) that returns “hard-coded” JSON.
- Place our solution under source control.

Quick Word on My Dev Setup

I just want to level set here on the current state of my development setup I’m going to use moving forward:

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

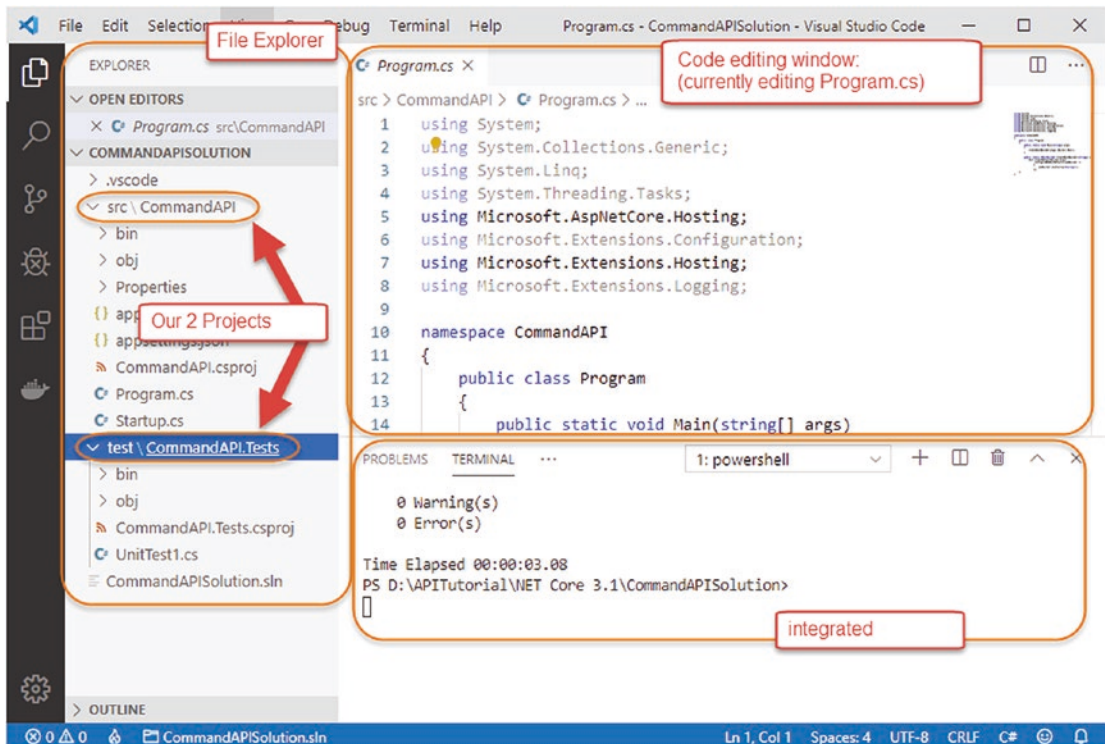


Figure 5-1. My VS Code setup

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

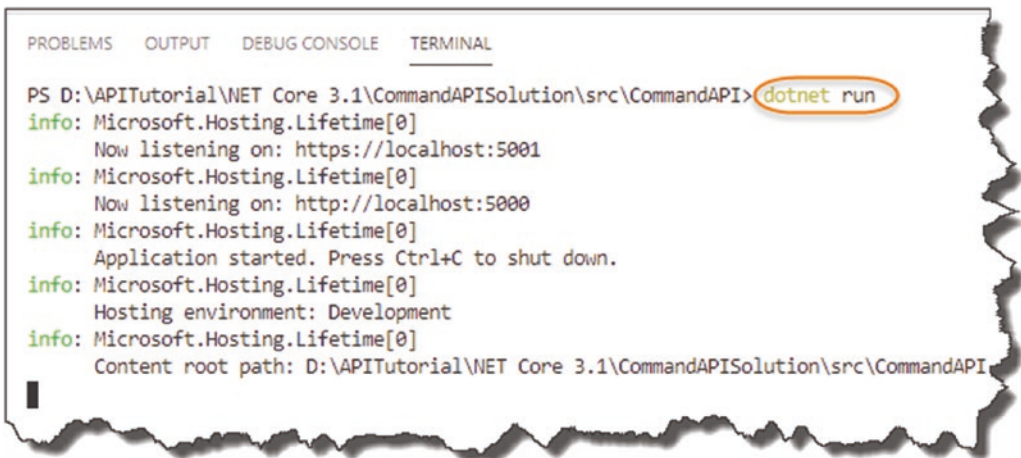
1. In VS Code hit “F1” (this opens the “command palette” in VS Code).
2. Type *shell* at the resulting prompt, and select “*Terminal: Select Default Shell.*”
3. You can then select from the Terminals that you have installed.

Start Coding!

First, let’s just check that everything is set up and working OK from a very basic startup perspective. To do this from a command-line type (ensure that you’re “in” the API project directory – *CommandAPI*)

```
dotnet run
```

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



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI> dotnet run
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI
```

Figure 5-2. Running our API for the first time

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

i To change that port allocation, you can edit the *launchSettings.json* file in the *Properties* folder; for now though there would be no benefit to that. We’ll talk more about this file when we come to our discussion on setting the runtime environment in Chapter 8.

If you go to a web browser and navigate to `http://localhost:5000` You’ll see the following.

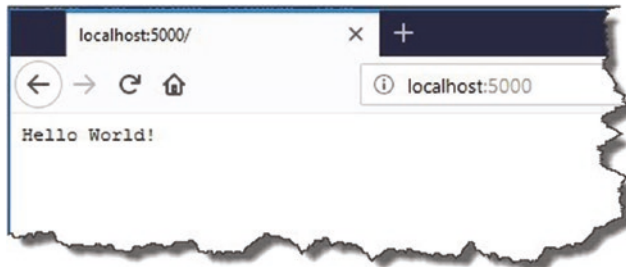


Figure 5-3. *Hello World!*

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

```

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

    app.UseRouting();

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

```

Figure 5-4. Where our greeting comes from

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

- `app.UseEndPoints`

You would have seen

- `app.Run(async)`

The previous version of the framework would also make use of

- `services.AddMvc()`: In our `ConfigureServices` method
- `app.UseMvc()`: In our `Configure` method

Further discussion on the differences between versions 2.x and 3.x of the .NET Core Framework can be found here: <https://docs.microsoft.com/en-us/aspnet/core/migration/22-to-30>.

Stop our host from listening (Ctrl+C on Windows – should be the same for Linux/OSX), and *remove* the highlighted section of code (shown previously) from our Configure method. Add the highlighted code shown next to our Startup class, making sure to update both the ConfigureServices and Configure methods:

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

namespace CommandAPI
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            //SECTION 1. Add code below
            services.AddControllers();
        }

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

            app.UseRouting();

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

What does this code do?

1. Registers services to enable the use of “Controllers” throughout our application. As mentioned in the info box, in previous versions of .NET Core Framework, you would have specified `services.AddMvc`. Don’t worry; we cover what the Model-View-Controller (MVC) pattern is below.
2. We “MapControllers” to our endpoints. This means we make use of the Controller services (registered in the `ConfigureServices` method) as endpoints in the *Request Pipeline*.

📌 Reminder The code for the entire solution can be found here on GitHub: <https://github.com/binarythistle/Complete-ASP-NET-3-API-Tutorial-Book>

As we have done before, run the project (ensure you *save the file* before doing this¹)

```
dotnet run
```

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

Call the Postman

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

So, if you’ve not done so already, go to the Postman website (www.getpostman.com), and download the version most suitable for your environment, (I use the Windows desktop client, but there’s a Chrome plugin along with desktop versions for other operating systems).

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

¹Plenty of times I’ve run code after making changes, and the changes were not reflected. Yes, that’s right - hadn’t saved the file.

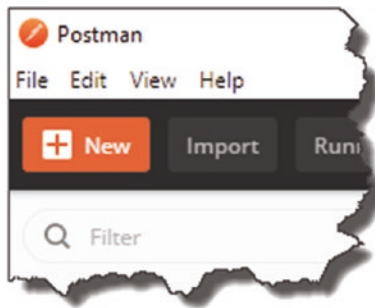


Figure 5-5. Start a New Request in Postman

The select “request.”

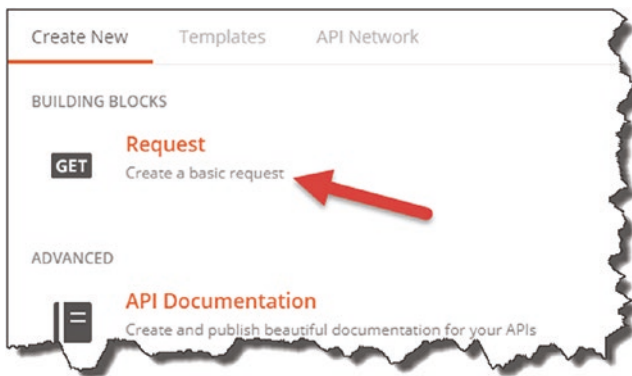


Figure 5-6. Create a basic request

Give the request a simple name, for example, “Test Request.”



Figure 5-7. Name your request

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

1. Click “+ Create Collection.”
2. Give it a name, for example, “CommandAPI.”
3. Select OK (the tick), and ensure you select your newly created collection (not shown).
4. Click Save to Command API.



Figure 5-8. Request Collection

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

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

or

```
https://localhost:5001
```

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

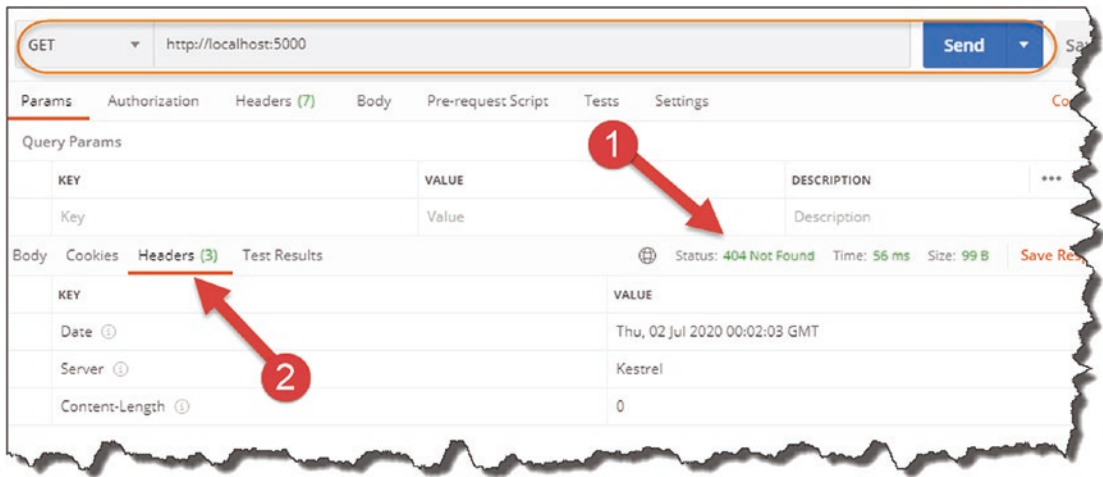


Figure 5-9. GET Request results in Postman

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

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

What have we broken?

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

What Is MVC?

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

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

I won’t go into *too* much detail. Again, I feel you’ll learn more about MVC by building a solution, rather than reading long textual explanations. I think when we cover off the Application Architecture below, things will make much more sense though.

Model–View–Controller

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

- Model (our Domain Data)
- View (User Interface)
- Controller (Requests and Actions)

In fact, to make things even simpler, as we’re developing an API, we won’t even have any **View** artefacts.² A high-level representation of this architecture for our API is shown here.

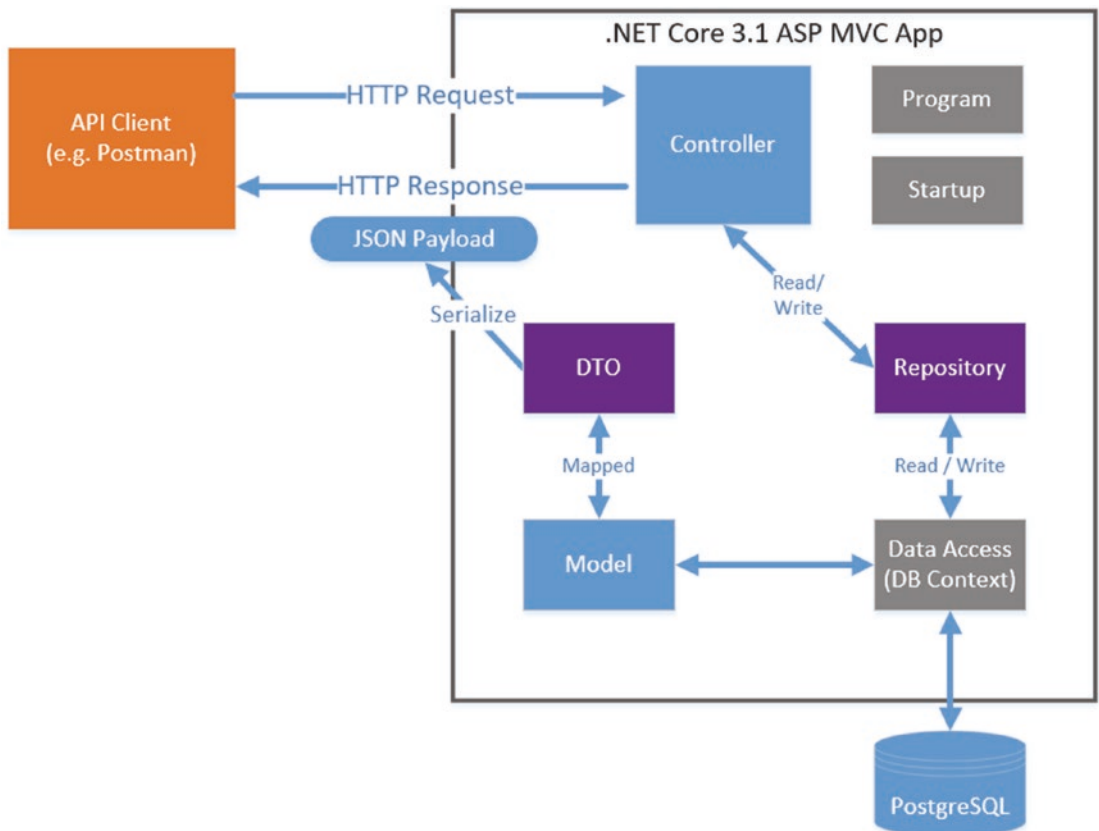


Figure 5-10. Our API Application Architecture

²You could argue that the serialized JSON payload is a “view” from a conceptual perspective.

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

Models, Data Transfer Objects, Repositories, and Data Access

You’re probably happy enough with the concept of a Model – it’s just data, right? Yes, that’s simple enough. So, looking at the architecture diagram in Figure 5-10, you’re then wondering what’s a DTO, a Repository, and a DB Context. And I don’t blame you – I struggled with the distinction between these concepts too at first. In fact, we could leave out DTOs and Repositories from our solution and it would work without them. So why include them at all? Great question; let me try and explain.

First, let me answer the “what” before I answer the “why.”

What’s the Distinction?

Let’s step through each of those classes:

- **Model:** Represents the *internal domain* data of our application (the “M” in MVC).
- **Data Transfer Objects (DTOs):** Are the *representations* of our Domain Models to our *external consumers*, meaning that we don’t expose internal implementation detail (our Models) to external concerns. This has multiple benefits as we’ll discuss later.
- **Data Access (aka DB Context):** Takes our Models and represents (or “mediates”) them down to a *specific persistence layer* (e.g., PostgreSQL, SQL Server, etc.). Going forward, I’ll refer to our Data Access class as a “DB Context” which is a technology-specific term taken from “Entity Framework Core” – don’t worry; more on that later in Chapter 7.

- **Repository:** Provides a *technology agnostic* (or persistence ignorant) view of our permanently stored data to our application.

So, what do you take from this? The main concept (which is repeated throughout the book) is that we should be *decoupling implementational detail* from the *interface* or *contract* we want to provide to consumers. But *why* is that a good thing?

Why Decoupling Is Good?

I’ve kind of alluded to that earlier, and we’ll cover it in more detail when we come to implementing these concepts, but in short decoupling our interfaces (or contracts) from our implementations provides the following benefits:

- **Security:** We may not want to expose potentially sensitive data contained in our implementation (think our Model) to our external consumers. Providing an external representation (e.g., a DTO) with sensitive information removed addresses this.
- **Change Agility:** Separating out our interface – which should remain consistent so as not to break our “contract” with our consumers – means we can then change our implementation detail without impacting that interface. We then have the confidence to react quickly to market demands without fear of breaking existing agreements. We’ll demonstrate this concept more when we come onto using *dependency injection* and our *repository*.

Bringing It Together

In the chapters that follow, we’ll leverage MVC as well as the other concepts discussed earlier to

- **Chapter 5:** Create a *Controller* to manage all our API requests (see our CRUD actions in Chapter 3).
- **Chapter 6:** Create a *Model* to internally represent our resources (in this case our library of command-line prompts)
- **Chapter 6:** Create a **Repository** to provide a technology agnostic view of our persisted data.

- **Chapter 7:** Leverage Entity Framework Core to create a **DB Context** that will allow us to persist our Model down to PostgreSQL.
- **Chapter 9:** Create **DTO** representations of our Model for external use.

Let’s wrap our architectural overview there (again, don’t worry – we’ll deep dive these concepts later) and move on to creating our Controller.

Our Controller

Making sure that you are in the main API project directory (*CommandAPI*), create a folder named “*Controllers*” underneath *CommandAPI* as a subfolder.

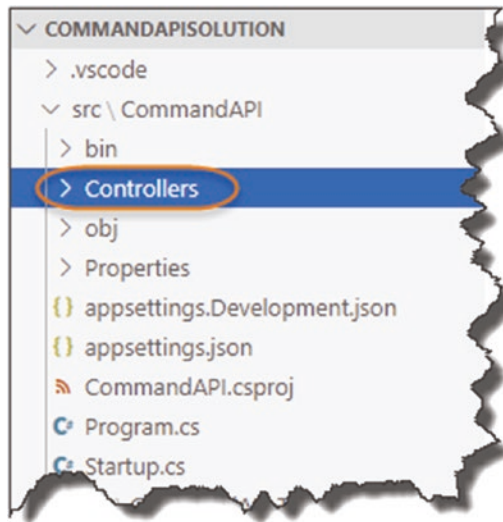


Figure 5-11. Controllers Folder in our API Project

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

i Quick Tip If you’re using VS Code, you can create both folders and files from within the VS Code directory explorer. Just make sure when you’re creating either that you have the correct “parent” folder selected.



Figure 5-12. File and folder creation in VS Code

Your directory structure should now look like this.



Figure 5-13. Our directory structure

Ensure that you postfix the **CommandsController** file with a “.cs” extension to denote it’s a C# file.

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

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

i A controller “Action” (I may also refer to it as an *endpoint*) maps to our API CRUD operations as listed in Chapter 3; our first action though will just return a simple hard-coded string.

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

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

namespace CommandAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CommandsController : ControllerBase
    {
        [HttpGet]
        public ActionResult<IEnumerable<string>> Get()
        {
            return new string[] { "this", "is", "hard", "coded" };
        }
    }
}
```

Again, if you don't fancy typing this in, the code is available here on GitHub:

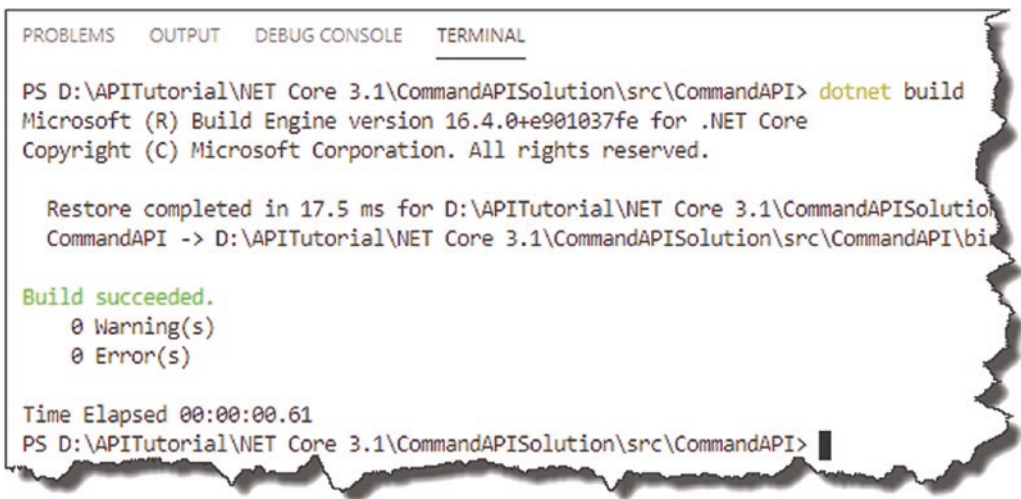
<https://github.com/binarythistle/Complete-ASP-NET-3-API-Tutorial-Book>

We'll come onto what all this means next, but first let's build it.

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

```
dotnet build
```

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



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI> dotnet build
Microsoft (R) Build Engine version 16.4.0+e901037fe for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 17.5 ms for D:\APITutorial\NET Core 3.1\CommandAPISolution
CommandAPI -> D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI\bi

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

Time Elapsed 00:00:00.61
PS D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI> |
```

Figure 5-14. Successful API run

Now, *run* the app.

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

Go to Postman (or a web browser if you like), and in the URL box, type `http://localhost:5000/api/commands`

Ensure that “Get” is selected in the drop-down (in Postman), then click “Send”; you should see something like this.

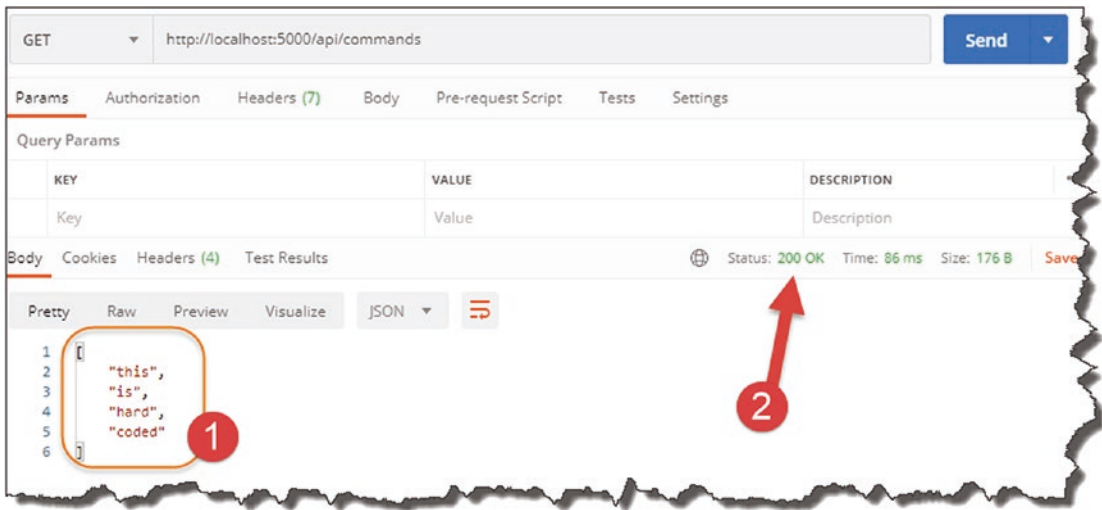


Figure 5-15. Our first API endpoint response

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

I guess technically you could say that we have implemented an API that services a simple “GET” request! Excellent, but I’m sure most of you want to take the example a little further.

Back to Our Code

OK so that’s great, but what did we actually do? Let’s go back to our code and pick it apart.

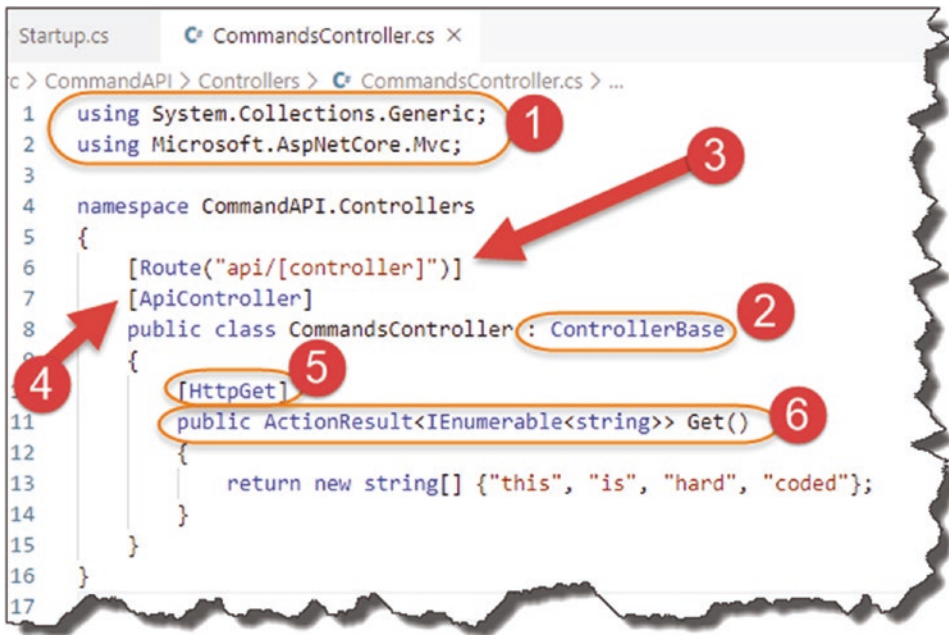


Figure 5-16. Deep dive on our first controller action

1. Using Directives

We included two using directives here:

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

2. Inherit from Controller Base

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

`ControllerBase` is further detailed on Microsoft Build.³

³<https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase?view=aspnetcore-3.1>

3. Set Up Routing

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

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

The URI convention for an API controllers is

`http://<server_address>/api/<controller_name>`

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

To enable this, we have “decorated” our `CommandsController` class with a `[Route]` attribute:

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

i You’ll notice that when we talk about the name of our controller from a *route perspective*, we use “Commands” as opposed to the fuller “Commands**Controller**”.

Indeed, the name of our controller really is “Commands”; the use of the “Controller” postfix in our class definition is an example of configuration over convention. Basically, it makes the code easier to read if we use this convention, that is, we know it’s a controller class.

⚠ Warning! We have specified our route using the `[controller]` “wildcard,” which dynamically derives that segment of the route from the name of our controller (minus the “Controller” portion) as we’ve explained before. So, in our case, this gives us the route:

api/commands

What this means is that if you *change the name of your Controller* for whatever reason, the *route will change also*. This may have quite unexpected consequences for our consumers, in effect breaking our contract - so be careful!

You can “rectify” this behavior by hardcoding the name of your route as so it would become

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

I’ll leave the semi-dynamically declared route for now as I think that is what you’ll most likely see out there in the field, but feel free to change to the hard-coded approach if you’re more comfortable with that.

4. ApiController Attribute

Decorating our class with this attribute provides the following out-of-the-box behaviors for our controller:

- Attribute Routing
- Automatic HTTP 400 Error responses (e.g., 400 Bad Request, 405 Not Allowed, etc.)
- Default Binding Sources (more on these later)
- Problem details for error status codes

It’s not mandatory to use it but highly recommended, as the default behaviors it provides are *really* useful; for a further deep dive on this, refer to the Microsoft documentation.⁴

5. HttpGet Attribute

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

Decorating our first simple action with [HttpGet] is really just specifying which verb our action responds to.

⁴<https://docs.microsoft.com/en-us/aspnet/core/web-api/index?view=aspnetcore-2.2#annotation-with-ApiController-attribute>

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

i As mentioned before, the Verb Attribute (e.g., GET) in combination with the route (e.g., `api/commands`) *should be unique* for each action (endpoint) within our API.

If you take a look back to our full list of the API endpoints in Chapter 3, you’ll notice that this is indeed the case.

6. Our Controller Action

This is quite an expansive area,⁵ and there are multiple ways you can write your controller actions. I’ve just opted for the “ActionResult” return type which was introduced as part of .NET Core 2.1.

In short, you’ll have an ActionResult return type for each API CRUD action, so we’ll end up with six by the time we’re finished.

Synchronous vs. Asynchronous?

In the recent example, our controller actions are *synchronous*, meaning that when they get called by a client (e.g., Postman), they will wait until a result is returned and in doing so *occupy a thread* (think of a thread as a small slice of a CPU’s time). Once a result is returned, that thread is then released (back to a thread pool) where it can be reused by some other operation.

The problem with synchronous operations is that if there is enough of them (think a high traffic API), eventually all the available threads will be used from the thread pool, *blocking* further operations from running. That is where asynchronous Controller Actions would come in.

An *asynchronous* controller action will not wait for a long-running operation (e.g., complex Database query or call over the network) to complete and will hand the thread

⁵<https://docs.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-3.1>

back to the pool while it waits. When the long-running operation does eventually have a result for us, a thread is reacquired by the controller action to complete the operation.

In short, asynchronous operations are really about *scalability* and not (as is sometimes claimed) speed. Just using an asynchronous controller action does nothing to improve the time the I/O operation (e.g., database query) takes to complete. It does, however, improve the situation where we may run out of threads (due to blocking) which has positive implications for scaling. There is also some nice usability implications when applied to User Interface design (have you ever used an application that “freezes” when performing a long-running operation?).

I did debate whether to use asynchronous controller actions in our example; however, in keeping with the “Thin and Wide” approach of the book, I thought it would introduce unnecessary complexity that would detract from the core thrust of the book, so I have omitted for now.

This section has already taken up enough space, so let’s move on!

Source Control

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

What is source control?

Source control is really about the following two concepts:

1. Tracking (and rolling back) changes in code
2. Coordinating those changes when there are multiple developers/contributors to the code (referred to as *Continuous Integration*; we’ll deep dive into this in Chapter 12)

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

- **Requirements Traceability:** Ensuring that the changes relate back to a requested feature/bug fix.
- **Release Notes:** Wrapping up our changes so we can publish new release notes for our app.

- **Rolling Back:** If we know what changed (and we broke something), we can either (a) fix it or (b) roll back the change – a source control system allows us to do that.

On top of tracking changes, the other primary reason for using a source control solution is to coordinate the changes to the codebase when multiple developers are working on it. If you’re the only person working on your code, you’re not going to really conflict with yourself (well not usually anyway). What about when you have more than one person making changes to the same codebase? How can that happen without things like overwriting each other’s changes? Again, this is where a source control solution comes in to play – it coordinates those changes and identifies conflicts should they arise.

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

1. To introduce you to the concept
2. So we can automatically deploy our app to production via a CI/CD⁶ pipeline – more in Chapter 12

Git and GitHub

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

What’s the difference?

Git is the source control system that

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

⁶Continuous Integration/Continuous Delivery (or Deployment).

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

- GitHub (probably the most well-recognized – and now acquired by Microsoft)
- Bitbucket (from Atlassian – the makers of Jira and Confluence)
- Gitlabs

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

Setting Up Your Local Git Repo

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

At a terminal/command line *in* the main **solution** directory, (**CommandAPISolution**), type (if your API app is still running you may want to stop it by hitting Ctrl + c)

```
git init
```

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

Now type

```
git status
```

This will show you all the “untracked” files in your directory (basically files that are not under source control); at this stage, that is everything.

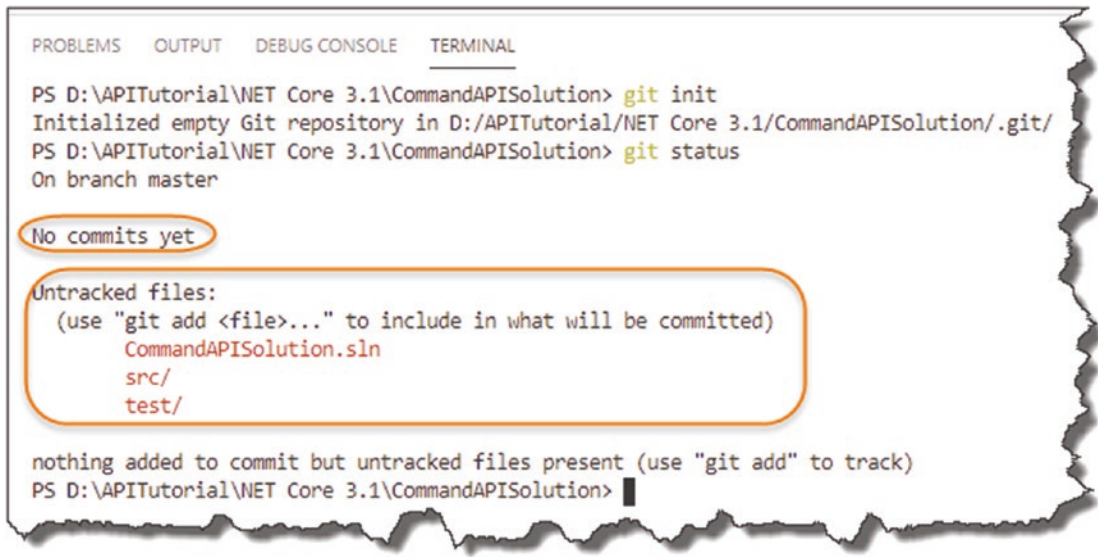


Figure 5-17. Untracked files in our new Git Repo

.gitignore file

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

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

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

# Visual Studio Code
.VS Code
```

```
# User-specific files
*.suo
*.user
*.userosscache
*.sln.docstates

# Build results
[Dd]ebug/
[Dd]ebugPublic/
[Rr]elease/
[Rr]eleases/
x64/
x86/
build/
bld/
[Bb]in/
[Oo]bj/
msbuild.log
msbuild.err
msbuild.wrn

# Visual Studio
.vs/

# Compiled Source
*.com
*.class
*.dll
*.exe
*.o
*.so
```

So if you want to use a **.gitignore** file (I recommend it – you don’t want to put compiled assets in a *source repository*), create one, and pop it in the root of your solution directory, as I’ve done here (this shows the file in VS Code).

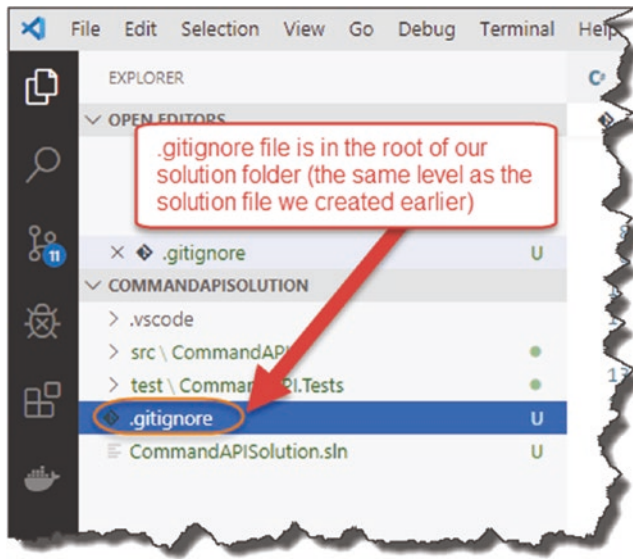


Figure 5-18. Our .gitignore file

Type `git status` again, and you should see this file now as one of the “untracked” files also.



Figure 5-19. Untracked .gitignore file

Track and Commit Your Files

OK, we want to track “everything” (except those files ignored!); to do so, type (ensure you put the trailing period “.”)

```
git add .
```

Followed by

```
git status
```

You should see the following.



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

No commits yet

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

Figure 5-20. Tracked Files ready for Commit

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

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

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

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

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS D:\APITutorial\NET Core 3.1\CommandAPISolution> git commit -m "Initial Commit"
[master (root-commit) 86576b7] Initial Commit
11 files changed, 248 insertions(+)
create mode 100644 .gitignore
create mode 100644 CommandAPISolution.sln
create mode 100644 src/CommandAPI/CommandAPI.csproj
create mode 100644 src/CommandAPI/Controllers/CommandsController.cs
create mode 100644 src/CommandAPI/Program.cs
create mode 100644 src/CommandAPI/Properties/launchSettings.json
create mode 100644 src/CommandAPI/Startup.cs
create mode 100644 src/CommandAPI/appsettings.Development.json
create mode 100644 src/CommandAPI/appsettings.json
create mode 100644 test/CommandAPI.Tests/CommandAPI.Tests.csproj
create mode 100644 test/CommandAPI.Tests/UnitTest1.cs
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> █
```

Figure 5-21. Committed Files

A quick additional `git status` and you should see the following.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS D:\APITutorial\NET Core 3.1\CommandAPISolution> git status
On branch master
nothing to commit, working tree clean
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> █
```

Figure 5-22. No further changes have occurred

🎉 Celebration Checkpoint Good job! We have basically placed our solution under *local* source control and have committed all our “changes” to our master branch in our first commit.

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

I will of course cover the necessary amount of Git to get the job done in this tutorial; further reading is purely optional!

The Git website also allows you to download the full *Pro Git* ebook; you can find that here: <https://git-scm.com/book/en/v2>

Set Up Your GitHub Repo

OK so the last section took you through the creation of a local Git repository, and that’s fine for tracking code changes on your local machine. However, if you’re working as part of a larger team, or even as an individual programmer, and want to make use of Azure DevOps (as we will in Chapters 12, 13, and 14), we need to configure a “remote Git repository” that we will

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

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

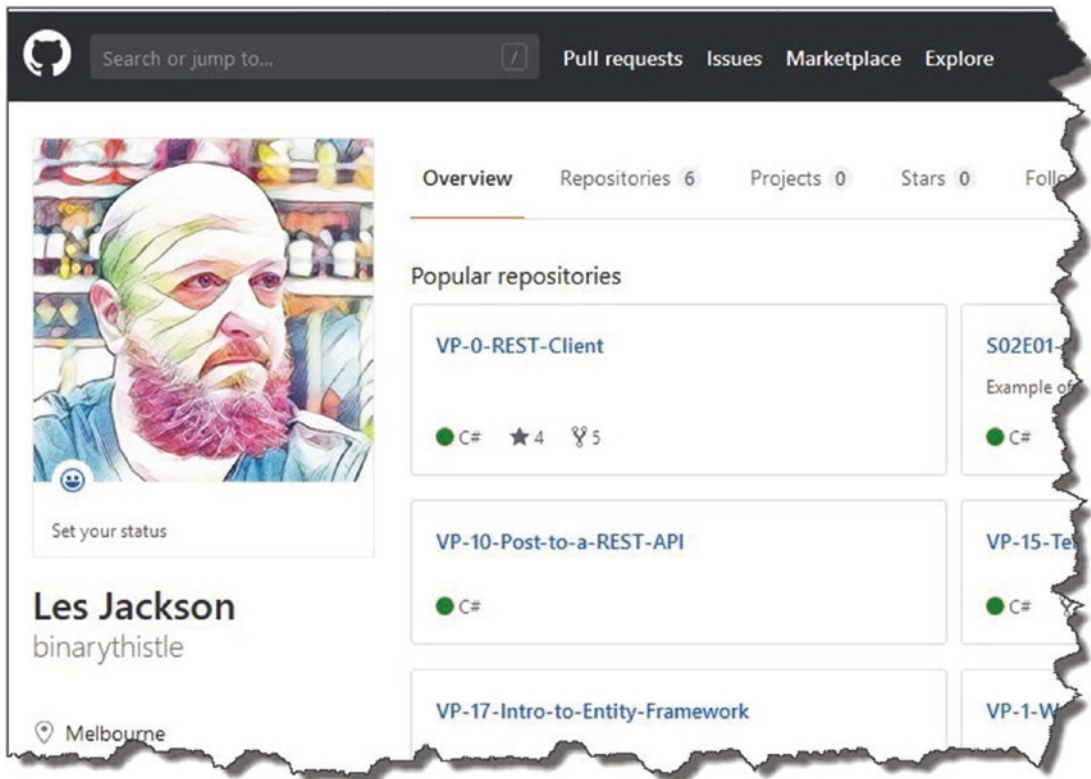


Figure 5-23. GitHub Landing Page

Create a GitHub Repository

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

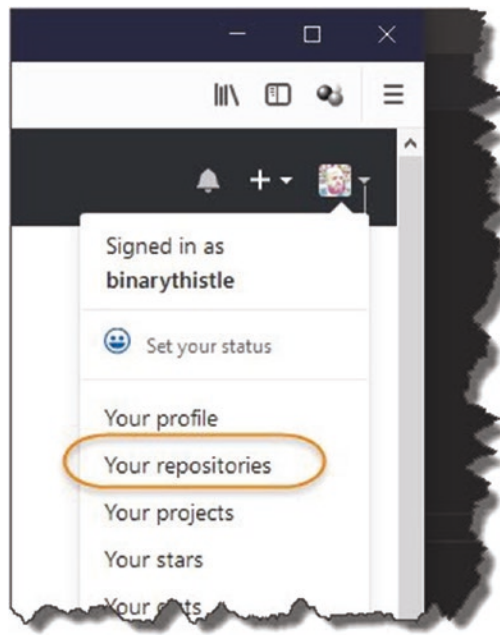


Figure 5-24. *Select your repositories*

Then click “New” and you should see the “Create a new repository” screen.

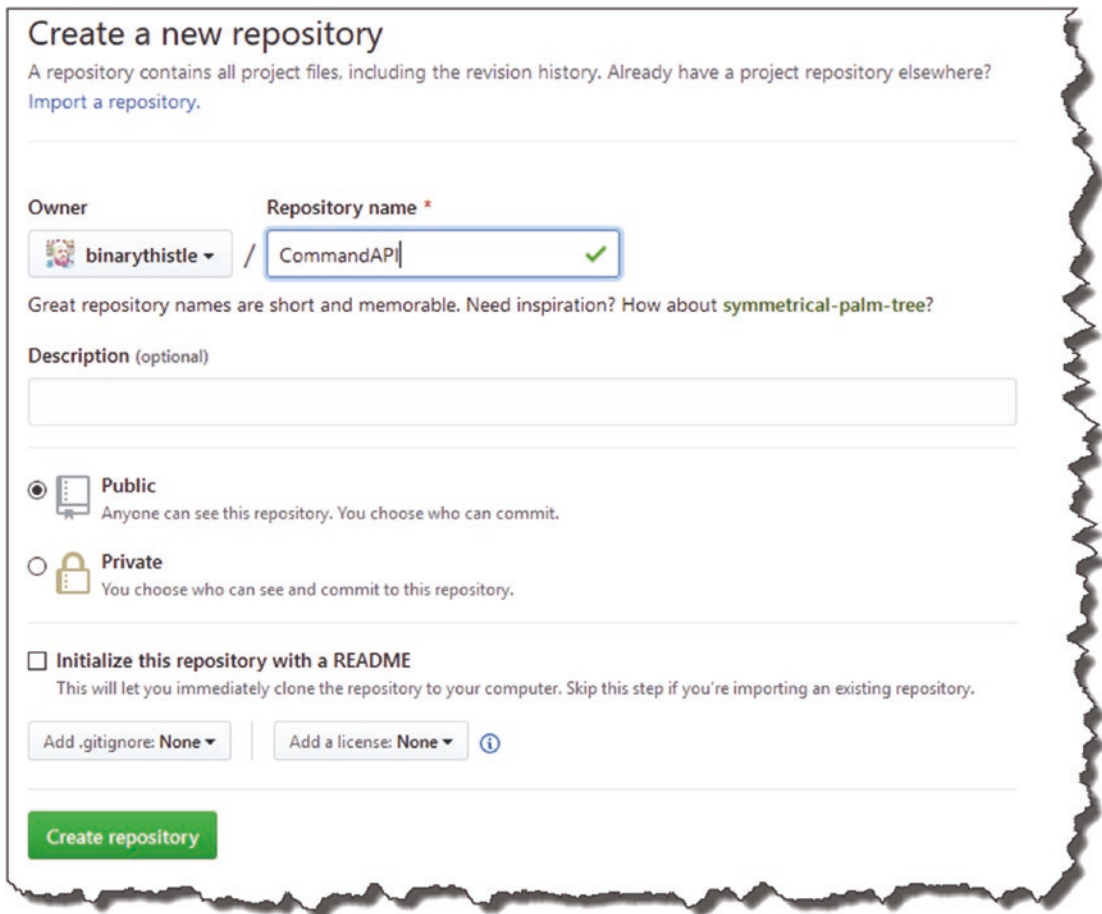


Figure 5-25. Create your repository

Give the repository a name (I just called mine *CommandAPI*, but you can call it anything you like), and select either Public or Private. For this tutorial, I strongly recommend selecting *Public*, primarily as that’s the option I’ve developed this tutorial with – and I know it works with the later sections of the book. Indeed, the option you select here is important as it has impacts when we come to set up our CI/CD pipeline in Chapter 12.

Then click “Create Repository”; you should see the following.



Figure 5-26. *GitHub repository created*

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

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> git status
On branch master
nothing to commit, working tree clean
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> git remote add origin https://github.com/binarythistle/Complete-ASP.NET-Core-API-Tutorial-2nd-Edition-Net-Core-3.1.git
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> git push -u origin master
Enumerating objects: 19, done.
Counting objects: 100% (19/19), done.
Delta compression using up to 8 threads
Compressing objects: 100% (15/15), done.
Writing objects: 100% (19/19), 3.42 KiB | 876.00 KiB/s, done.
Total 19 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/binarythistle/Complete-ASP.NET-Core-API-Tutorial-2nd-Edition-Net-Core-3.1.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> █
```

Figure 5-27. Add remote repo and push your local repo



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

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

So What Just Happened?

Well, in short

- We "registered" our remote GitHub repo with our local repo (first command).
- We then pushed our local repo up to GitHub (second command).

Note The first command line only needs to be issued once; the second one we'll be using more throughout the rest of the tutorial.

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

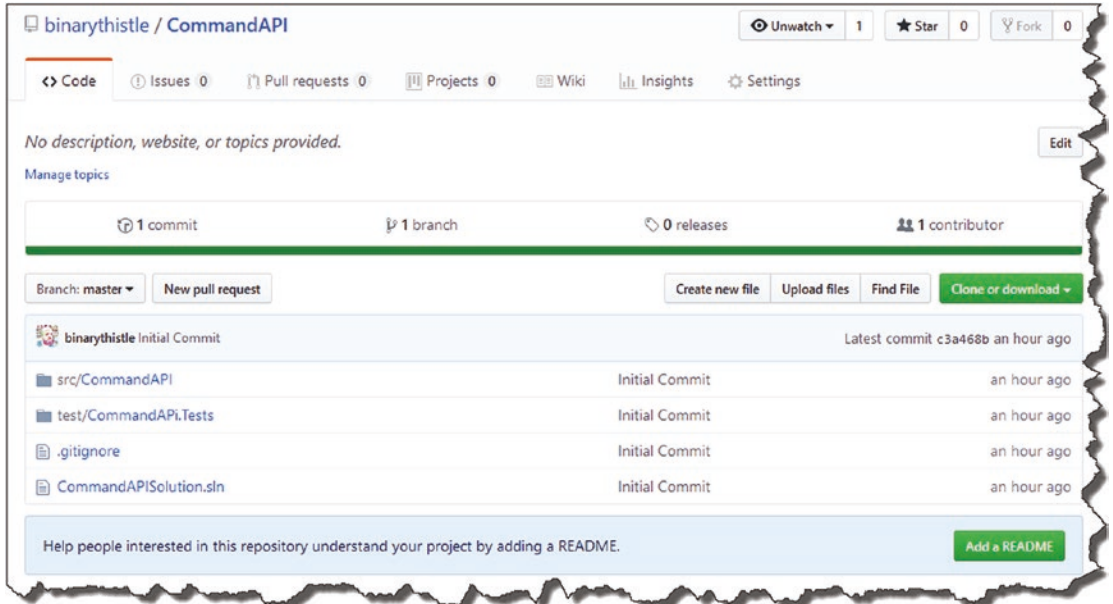


Figure 5-28. Our code is now on GitHub

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

Well that’s it for this chapter – great job!

CHAPTER 6

Our Model and Repository

Chapter Summary

In this chapter we're going to introduce "data" to our API, so we'll begin our journey with our Model and Repository classes.

When Done, You Will

- Understand what a "Model" class is and code one up.
- Define our Repository Interface, and implement a "mock" instance of it.
- Understand how we use *Dependency Injection* to decouple interfaces from implementation.

Our Model

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

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

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

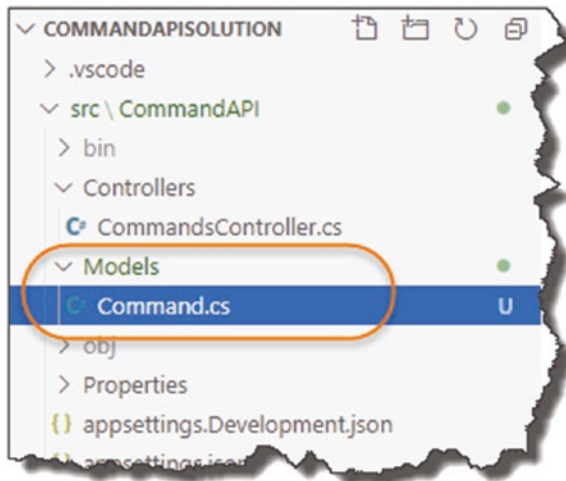


Figure 6-1. Model Folder and Command Class

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

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

As promised, very simple; just be sure that you've specified the correct namespace at the top of the code:

CommandAPI.Models

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

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

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

Data Annotations

We could leave our model class like that, but when we come to working with it, especially for creation and update actions, we want to ensure that we specify the properties of our Model that are mandatory and those that are not. For example, would there be any value in adding a command-line snippet to our solution without specifying some data for our CommandLine property? Probably not. We solve this by adding some *Data Annotations* to our class.

We can decorate our class properties with Data Annotations to specify things like

- Is it required or not?
- Maximum length of our strings
- Whether the property should be defined as Primary Key
- And so on.

In order to use them in the Command class, make the following updates to our code, making sure to include the using directive as shown:

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;}
```

```
}
```

```
}
```

The Data Annotations added should be self-explanatory:

- All Properties are required (they cannot be null).
- Our Id property is a primary key.
- In addition, the `HowTo` property can only have a maximum of 250 characters.

With our annotations in place, when we come to creating an instance of our Model later, a validation error (or errors) will be thrown if any of them are not adhered to. They also provide a means by which to generate our database schema, which we'll come onto in Chapter 7.

As we have made a simple, yet significant, change to our code, let's add the file to source control, commit it, then push up to GitHub; to do so, issue the following commands in order (make sure you're in the Solution folder *CommandAPISolution*):

```
git add .
git commit -m "Added Command Model to API Project"
git push origin master
```

You have used these all before, but to reiterate

- First command adds all files to our local Git repo (this means our new *Command.cs* file).
- Second command commits the code with a message.
- Third command pushes the commit up to GitHub.

If all worked correctly, you should see the commit has been pushed up to GitHub; see the following.

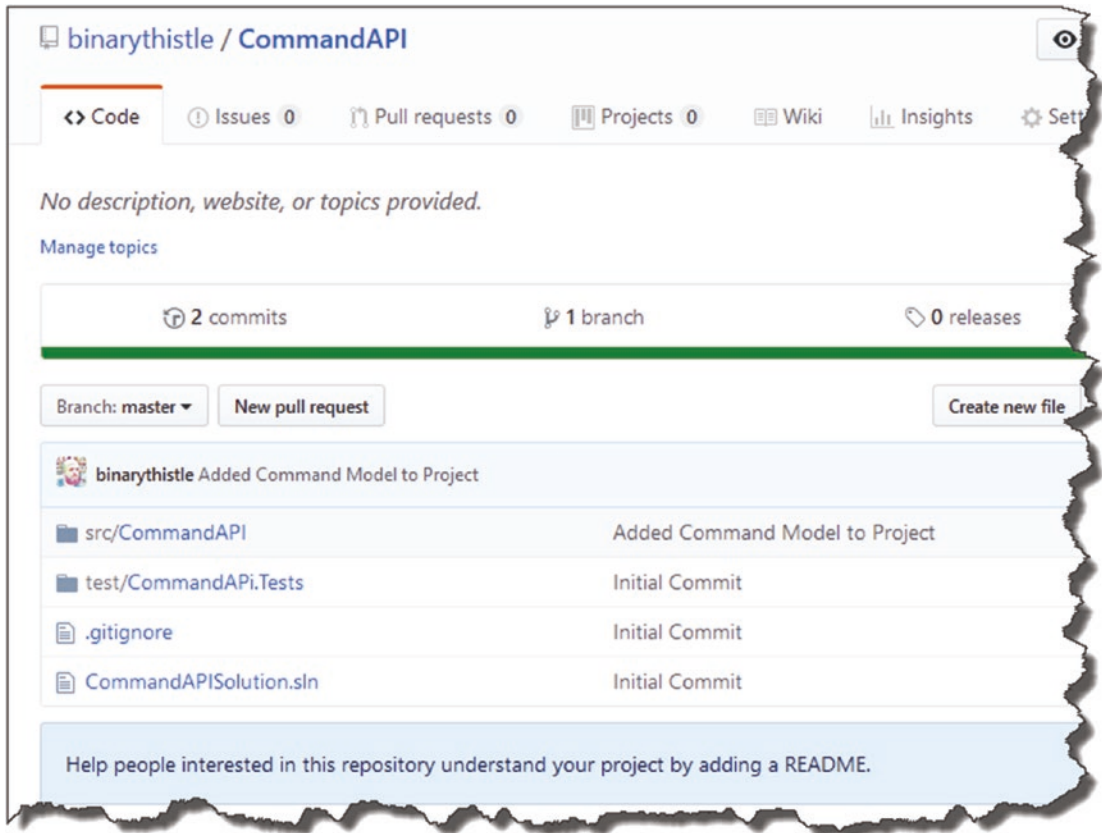


Figure 6-2. Our Committed Model Class

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

Our Repository

Taking a quick look back at our application architecture, I've outlined the components we've either started or, in the case of our Model, completed.

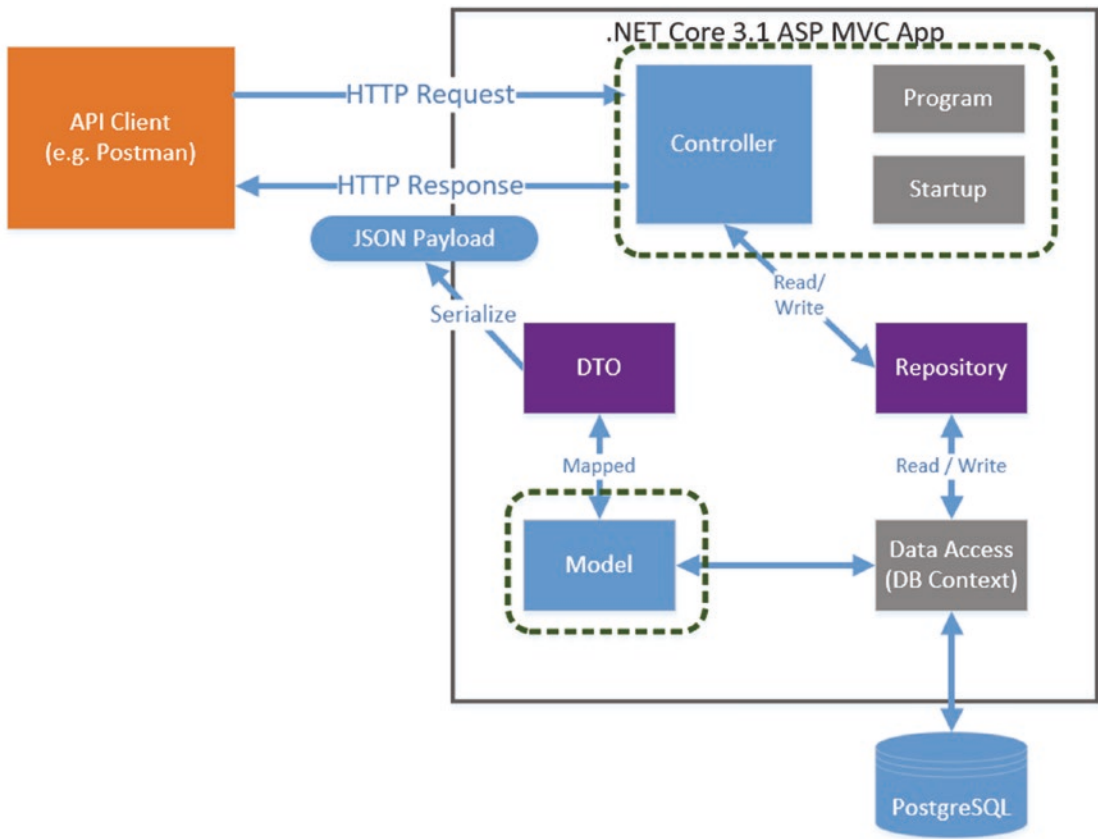


Figure 6-3. Progress through our architecture

It's all still a bit disjointed; to review we have

- Started our Controller that currently returns hard coded data
- Created our Model

The next step in our journey is to define our *Repository Interface*, which will provide our controller with a technology-agnostic way to obtain data.

What Is an Interface?

An interface is just as it sounds; it's a *specification* for *what* functionality we want it to provide (in this case to our Controller), but we don't detail *how* it will be provided – that comes later. It's essentially an agreement, or *contract*, with the consumer of that Interface.

When we think about what methods our Repository Interface *should* provide to our Controller (don't think about how yet), we can look back at our CRUD actions from Chapter 3 for some guidance.

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

In this case, they almost directly drive what our Repository should provide:

- Return a collection of all Commands.
- Return a single Command (based on its Id).
- Create a new Command Resource.
- Update an existing Command Resource (this covers PUT and PATCH).
- Delete an existing Command Resource.

To start implementing our Repository, back in the root of our API Project (in the *CommandAPI* folder), create another folder and call it *Data* as shown here.

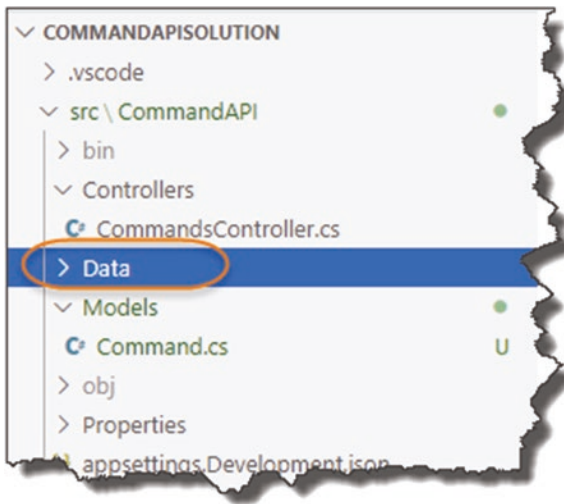


Figure 6-4. Data Folder will hold our Repository Interface

Inside this folder, create a file and name it ***ICommandAPIRepo.cs***.

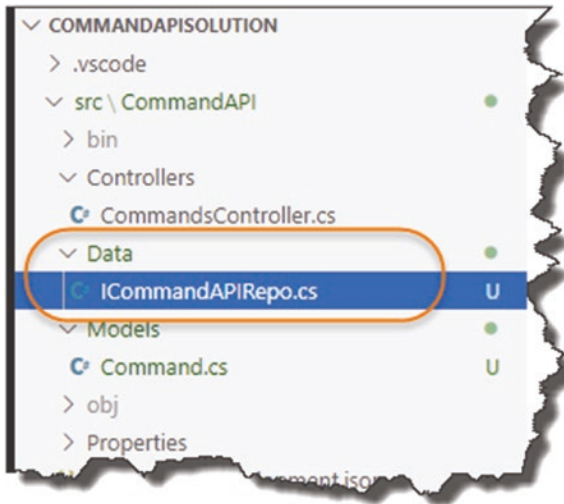


Figure 6-5. Our ICommanderRepo.cs File

Inside the file, add the following code:

```
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);
    }
}
```

Your file should look like this; make sure you **save the file**, and let's take a look at what we have done.

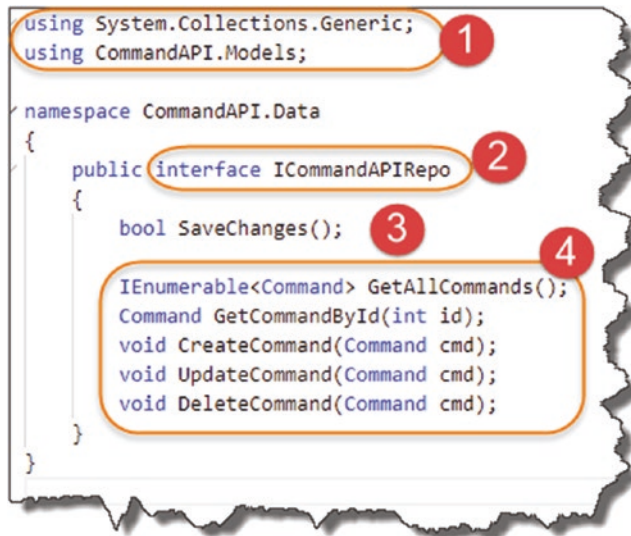


Figure 6-6. *ICommandAPIRepo Interface*

1. Using directives, noting that we have brought in the namespace for our Models.
2. We specify a public interface and give it a name starting with capital “I” to denote it’s an interface.
3. We specify that our Repository should provide a “Save Changes” method; stick a pin in that for now, we’ll revisit when we come to talking about Entity Framework Core in Chapter 7.
4. Section 4 defines all the other method signatures that consumers of this interface can use to obtain and manipulate data. They also serve another purpose, which I detail in the section below.

What About Implementation?

That’s our Repository Interface complete. Yes, that’s right; it’s done, fully complete. So, your next question (well it was my next question when I was learning about interfaces), will be: OK, but where does stuff “get done”?

Great question!

Again, to labor the point, our interface is just a specification (or a contract) for our consumers. We still need to *implement* that contract with a *concrete class*. And this is the power and the beauty of using interfaces: we can create multiple *implementations* (concrete classes) to provide the same interface, but the consumer doesn’t know, or care, about the implementation being used. All they care about is the interface and what it ultimately provides to them.

Still confused? Let’s move to an example.

Our Mock Repository Implementation

We are going to create a concrete class that implements our interface using our model; however, we’ll just be using “mock” data at this stage (we’ll create another implementation of our interface to use “real” data in the next chapter).

So, in the same **Data** folder where we placed our repository interface definition, create a new file called **MockCommandAPIRepo.cs**, and add the following code:

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

namespace CommandAPI.Data
{
    public class MockCommandAPIRepo : ICommandAPIRepo
    {
    }
}
```

You should see something like this in your editor.

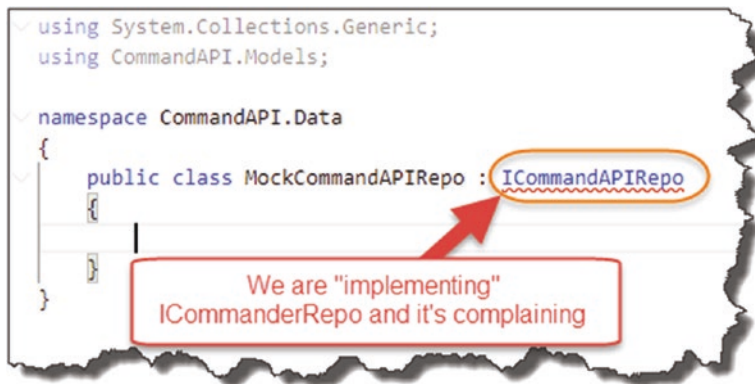


Figure 6-7. Our Concrete Class Definition is complaining

We have created a public class definition and specified that we want it to implement `ICommanderRepo`, as denoted by

```
: ICommanderRepo
```


And we can see that it's complaining; this is because we haven't "implemented" anything yet. If you're using VS Code or Visual Studio, place your cursor in the complaining section and press

CTRL + .

This will bring up some helpful suggestions on resolution; we want to select the first option "Implement Interface," as shown in the next figure.

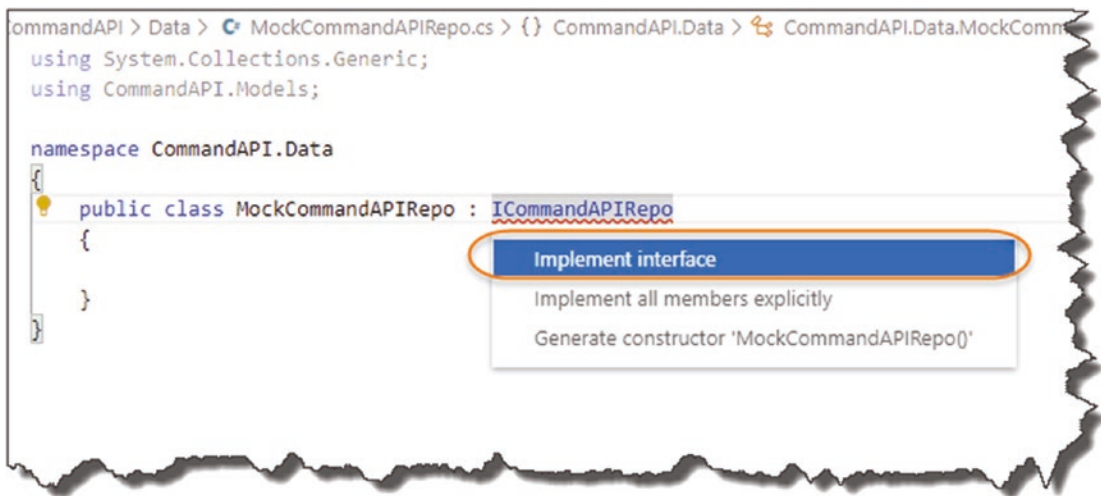


Figure 6-8. *Help is always appreciated!*

This should then generate some placeholder implementation code for our class.

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

namespace CommandAPI.Data
{
    public class MockCommandAPIRepo : ICommandAPIRepo
    {
        public void CreateCommand(Command cmd)
        {
            throw new System.NotImplementedException();
        }

        public void DeleteCommand(Command cmd)
        {
            throw new System.NotImplementedException();
        }

        public IEnumerable<Command> GetAllCommands()
        {
            throw new System.NotImplementedException();
        }

        public Command GetCommandById(int id)
        {
            throw new System.NotImplementedException();
        }

        public bool SaveChanges()
        {
            throw new System.NotImplementedException();
        }

        public void UpdateCommand(Command cmd)
        {
            throw new System.NotImplementedException();
        }
    }
}
```

We're only going to write code for these

Figure 6-9. Auto-generated Implementation code

As you can see, it has provided all the method signatures for the members of our interface and populated them with a `throw new System.NotImplementedException();`

In our example we're only going to update our two "read" methods:

- GetAllCommands
- GetCommandById

This is enough to demonstrate the core concepts of using interfaces and by extension *Dependency Injection*. So, in those two methods, add the following code as shown below, remembering to save your work when done:

```

.
.
.
public IEnumerable<Command> GetAllCommands(){
    var commands = new List<Command>
    {
        new Command{
            Id=0, HowTo="How to generate a migration",
            CommandLine="dotnet ef migrations add <Name of Migration>",
            Platform=".Net Core EF"},
        new Command{
            Id=1, HowTo="Run Migrations",
            CommandLine="dotnet ef database update",
            Platform=".Net Core EF"},
        new Command{
            Id=2, HowTo="List active migrations",
            CommandLine="dotnet ef migrations list",
            Platform=".Net Core EF"}
    };
    return commands;
}

public Command GetCommandById(int id){
    return new Command{
        Id=0, HowTo="How to generate a migration",
        CommandLine="dotnet ef migrations add <Name of Migration>",

```

```

    Platform=".Net Core EF"};
}
.
.
.

```

What this does is take our Model class and use it to create some simple mock data (again just hard-coded) and return it when these two methods are called. Not earth-shattering, but it is an implementation (of sorts) of our repository interface.

We now need to move on to making use of the ICommandAPIRepo interface (and by extension the MockCommandAPIRepo concrete class) from within our controller.

To do this we use Dependency Injection.

Dependency Injection

Dependency Injection (DI) has struck fear into many a developer getting to grips with it (myself included), but once you grasp the concept, not only is it pretty straightforward, it's also really powerful and you'll *want* to use it.

What makes it even easier in this instance is that DI is baked right into the heart of ASP.NET Core, so we can get up and running with it quickly without much fuss at all. Next, I'll take you through a quick theoretical overview; then we'll employ DI practically in our project (indeed, we'll continue to use it throughout the tutorial).

Again, as with many of the concepts and technologies in this tutorial, you could fill an entire book on DI, which I'm not going to attempt to do here. If you want a deep dive on this subject beyond what I outline below, the MSDN docs are decent¹.

Back to the Start (Up)

To talk about DI in .NET Core, we need to move back to our Startup class and in particular the ConfigureServices method.

¹<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1>

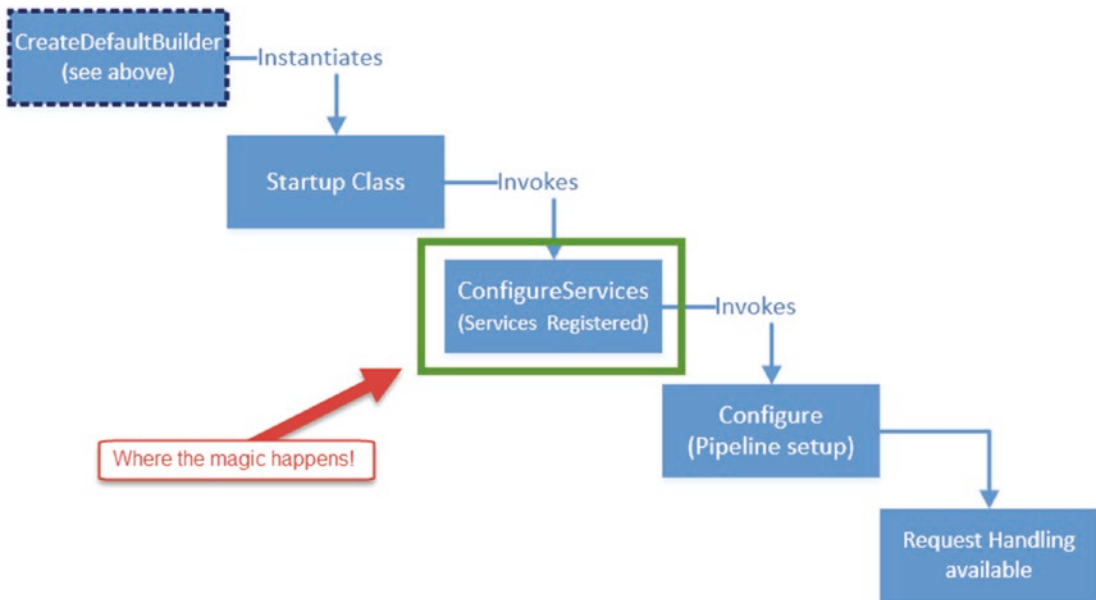


Figure 6-10. *Startup Class Sequence*

Casting your mind back, it is in the ConfigureServices method where our “services” are registered (in this case think of a service as *both* an interface and an implementation of it). But what exactly do we mean by *register*?

When we talk about registering services, what we are really talking about is something called a Service Container; this is where we “register” our services. Or to put it another way, this is where we tell the DI system to associate an interface to a given concrete class. See the following diagram.

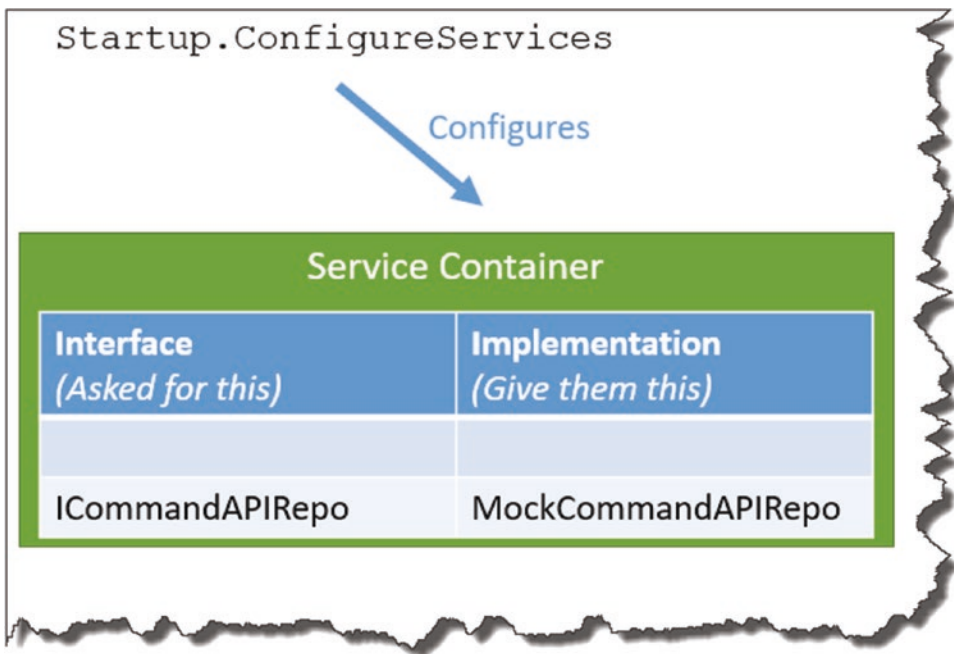


Figure 6-11. *Service Container with our Repository Service Registered*

Once we have registered our service in the Service Container, whenever we request to make use of a given interface from somewhere else in our app, the DI system will serve up, or “inject,” the concrete class we’ve associated with the interface (aka the “dependency”) in its place.

This means that if we ever need to swap out our concrete class for some other implementation, we only need to make the change in one place (the `ConfigureServices` method); the rest of our code does not need to change.

We will follow this practice in this tutorial by first registering our mock repository implementation against the `ICommandAPIRepo` interface; then we’ll swap it out for something more useful in the next chapter without the need to change any other code (except the registration).

This decoupling of interface (contract) from implementation means that our code is infinitely more maintainable as it grows larger.

Enough theory; let’s code.

Applying Dependency Injection

Back over in our API Project, open the Startup class, and add the following code to our ConfigureServices method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    //Add the code below
    services.AddScoped<ICommandAPIRepo, MockCommandAPIRepo>();
}
```

The code is quite straightforward; it uses the service collection: `services`, to register our `ICommandAPIRepo` with `MockCommandAPIRepo`. The only other novelty is the use of the `AddScoped` method.

This has to do with something called “Service Lifetimes,” which in essence tells the DI system how it should provision a service requested via DI; there are three methods available:

- **AddTransient:** A service is created each time it is requested from the Service Container.
- **AddScoped:** A service is created once per client request (connection).
- **AddSingleton:** A service is created once and reused.

Beyond what I’ve just outlined, I feel we may get ourselves off track from our core subject matter: building an API! So, we’ll leave it there for now; again refer to Microsoft Docs as mentioned earlier if you want more info.

OK, so now that we have registered our service, the next step is to make use of it from within our Controller – how do we do that?

Constructor Dependency Injection

If I’m being honest, it was this next bit that tripped me up when I was learning DI, so I’ll try and be as clear as I can when describing how it works.

We can't just "new-up" an interface in the same way that we can with regular classes; see Figure 6-12.

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

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

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

Figure 6-12. You can't write this code!

You will get an error along the lines of "Can't create an instance of an abstract class or interface." You could revert to "newing-up" a concrete instance of our `MockCommandAPIRepo` class, but that would defeat the entire purpose of what we have just been talking about. So how do we do it?

The answer is that we have to give the DI system an *entry point* where it can perform the "injection of the dependency," which in this case, it means creating a class constructor for our Controller and providing `ICommandAPIRepo` as a required input parameter. We call this *Constructor Dependency Injection*.

i Pay very careful attention to the *Constructor Dependency Injection* code pattern that follows; as you'll see, this pattern is used time and time again throughout our code as well as in other projects.

Let's implement this. Move back over to our API project, and open our `CommandsController` class, and add the following constructor code (make sure you add the new using statement too):

```
// Remember this using statement
using CommandAPI.Data;
.
.
.
namespace CommandAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CommandsController : ControllerBase
    {
        //Add the following code to our class

        private readonly ICommandAPIRepo _repository;

        public CommandsController(ICommandAPIRepo repository)
        {
            _repository = repository;
        }
    }
.
.
.
```

Let's go through what's happening.

```

using System.Collections.Generic;
using CommandAPI.Data; ①
using Microsoft.AspNetCore.Mvc;

namespace CommandAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CommandsController : ControllerBase
    {
        ② private readonly ICommandAPIRepo _repository;
        ③ public CommandsController(ICommandAPIRepo repository)
        {
            _repository = repository; ⑤
        }
    }
}

```

ICommandAPIRepo -> MockCommandAPIRepo

④

Figure 6-13. *Constructor Dependency Injection Pattern*

1. Add the new using statement to reference `ICommandAPIRepo`.
2. We create a private read-only field `_repository` that will be assigned the injected `MockCommandAPIRepo` object in our constructor and used throughout the rest of our code.
3. The Class constructor will be called when we want to make use of our Controller.
4. At the point when the constructor is called, the DI system will spring into action and inject the required dependency when we ask for an instance of `ICommandAPIRepo`. This is *Constructor Dependency Injection*.
5. We assign the injected dependency (in this case `MockCommandAPIRepo`) to our private field (see point 1).

And that's pretty much it! We can then use `_repository` to make use of our concrete implementation class, in this case `MockCommandAPIRepo`.

As I've stated earlier, we'll reuse this pattern multiple times through the rest of the tutorial; you'll also see it everywhere in code in other projects – take note.

Update Our Controller

We'll wrap up this chapter by implementing our two "Read" API controller actions using the mock repository implementation we have. So just to be clear we'll be implementing the following endpoints.

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

We'll start with implementing the endpoint that returns a collection of all our command resources, so move back into our Controller, and first *remove* our existing controller action.

```

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

        public CommandsController(ICommandAPIRepo repository)
        {
            _repository = repository;
        }

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

```

Remove or comment this action out

Figure 6-14. Removal of our old Controller Action

In its place, add the following code, remembering to add the required using statement at the top of the class too:

```
// Remember this using statement
using CommandAPI.Models;
.
.
.
namespace CommandAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CommandsController : ControllerBase
    {
        private readonly ICommandAPIRepo _repository;

        public CommandsController(ICommandAPIRepo repository)
        {
            _repository = repository;
        }

        //Add the following code
        [HttpGet]
        public ActionResult<IEnumerable<Command>> GetAllCommands()
        {
            var commandItems = _repository.GetAllCommands();

            return Ok(commandItems);
        }

        .
        .
        .
    }
}
```

I think the code is relatively straightforward but let's just step through it.

```
[HttpGet] 1
public ActionResult<IEnumerable<Command>> GetAllCommands() 2
{
    var commandItems = _repository.GetAllCommands(); 3
    return Ok(commandItems); 4
}
```

Figure 6-15. *New Controller Action using our Repository*

1. The controller action responds to the GET verb.
2. The controller action should return an enumeration (IEnumerable) of Command objects.
3. We call GetAllCommands on our repository and populate a local variable with the result.
4. We return a HTTP 200 Result (OK) and pass back our result set.

Make sure you save everything, run your code, and call the endpoint from Postman.

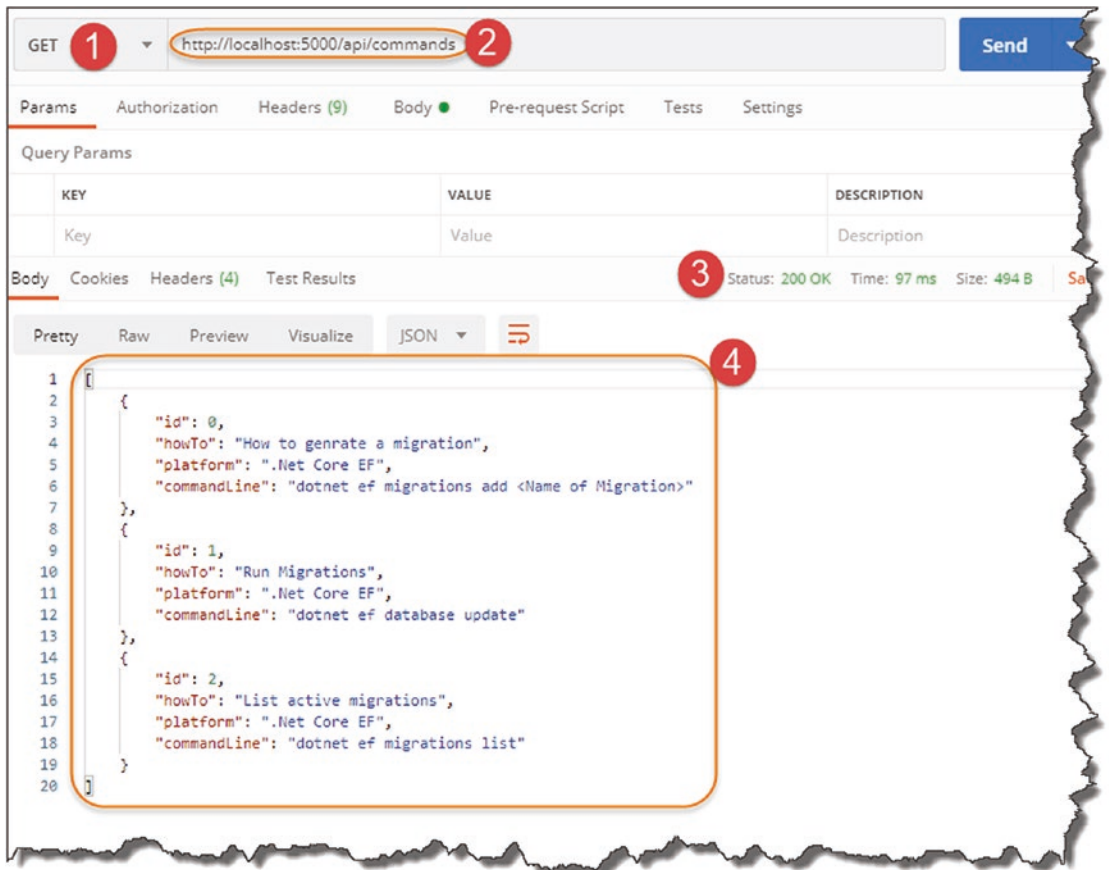


Figure 6-16. Successful API Endpoint Result

1. Verb set to GET.
2. Our URI is exactly the same as the one we have used before.
3. We get a 200 OK status result.
4. We have the hardcoded data returned from our mock repository!

🎉 Celebration Checkpoint This is actually a really important checkpoint! We have implemented our repository interface, created and used a concrete (mock) implementation of it, and used it in our Controller via Dependency Injection!

Give yourself five gold stars and a pat on the back.

We have one more controller action to implement in this section: returning a single resource by supplying its Id. Back over in the Controller, add the following code to implement:

```
.
.
.
[HttpGet]
public ActionResult<IEnumerable<Command>> GetAllCommands()
{
    var commandItems = _repository.GetAllCommands();
    return Ok(commandItems);
}
```

//Add the following code for our second ActionResult

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

```
.
.
.
```

There's a bit more going on here; let's review.

```

[HttpGet("{id}")] 1
public ActionResult<Command> GetCommandById(int id) 2
{
    var commandItem = _repository.GetCommandById(id); 3
    if (commandItem == null)
    {
        return NotFound(); 4
    }
    return Ok(commandItem); 5
}

```

Figure 6-17. *GetCommandByID endpoint*

1. The route to this controller action includes an additional route parameter, in this case the `Id` of the resource we want to retrieve; we can specify this in the `HttpGet` attribute as shown.
2. The controller action requires an `id` to be passed in as a parameter (this comes from our route; see point 1) and returns an `ActionResult` of type `Command`.
3. We call `GetCommandById` on our repository passing in the `Id` from our route, storing the result in a local variable.
4. We check to see if our result is null and, if so, return a 404 Not Found result.
5. Otherwise if we have a `Command` object, we return a 200 OK and the result.

Note Our mock repository will always return a result irrespective of what `Id` we pass in, so the null check will never return false in this case. That will change when we come to our “real” repository implementation in Chapter 7.

Let’s check our code by testing it in Postman; note that the route we’ll require is `/api/commands/n` where *n* is an integer value.

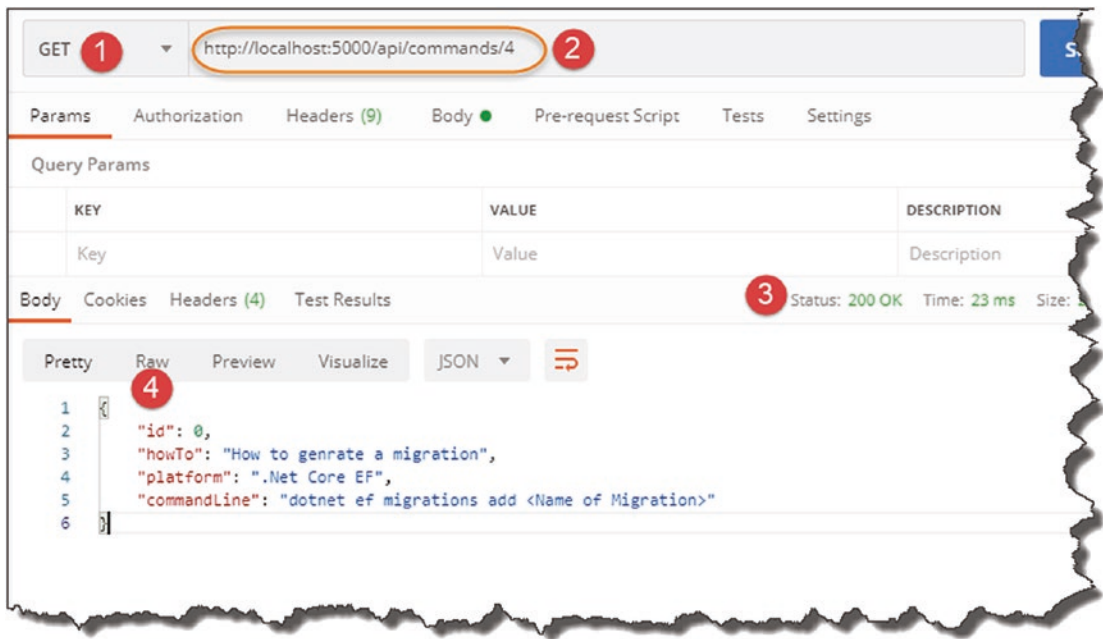


Figure 6-18. Single Command Resource Returned

1. We’re still using a GET request.
2. Our URI has changed to reflect the route we need to use to hit our endpoint.
3. 200 OK Status Retrieved.
4. Single Resource returned.

We’ll wrap this chapter up here for now as we’ve covered a lot of ground, but we will revisit these two controller actions later when we come on to discussing Entity Framework Core, Data Transfer Objects, and Unit Testing.

Before we finish here though, remember to save everything and (ensuring you’re in the main *solution* folder **CommandAPISolution**):

- `git add .`
- `git commit -m "Added Model and Mock Repository"`
- `git push origin master`

to update our Git repository (local and remote) with our changes.

In the next chapter, we move on to using “real” data that’s persisted in a database backend rather than relying on hard-coded mock data.

CHAPTER 7

Persisting Our Data

Chapter Summary

In this chapter we'll move away from mock data and implement our data access and persistence layers to store and retrieve data in a PostgreSQL database.

When Done, You Will

- Have configured a PostgreSQL instance (including setting up a new instance in Docker if required)
- Have created a Database Context (DB Context) class using Entity Framework Core
- Have used “migrations” to create the necessary schema in our database
- Have created a new implementation of our repository interface to use our DB Context
- Have used Dependency Injection to swap out our mock repository for our DB Context version

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

Architecture Progress Check

Before we move on with all of the given learning points, let's just check where we are in terms of progressing our application architecture. In the following diagram, I've outlined the components we've either started work on or in some cases completed altogether.

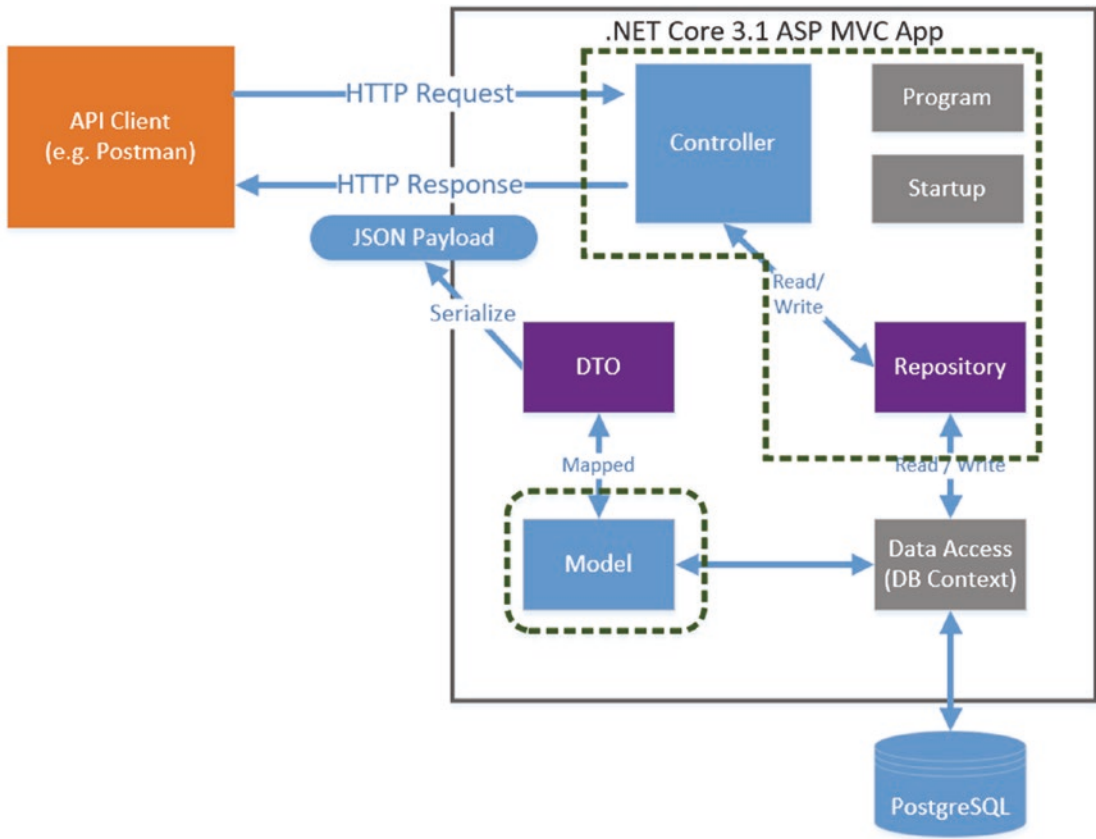


Figure 7-1. Application Architecture Progress

To review

- Our Model is completed.
- Our repository interface definition is complete.
- We’ve implemented a mock instance of our repository interface to return mock *Model* data.
- We’ve used constructor dependency injection in our controller to use our repository to implement our 2 GET controller actions. These both return mocked *Model* data.

We want to tie this altogether with a Database, DB Context, and an updated instance of our repository so we can work with real dynamic data that persists over time. So, what are we waiting for?

PostgreSQL Database

Before moving on to writing our DB Context, I first want to make sure we have an instance of PostgreSQL up and running and configured correctly.

Using Docker

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

Ensuring you have Docker installed and running (see Chapter 2) at a command prompt; simply type

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

Note This is all on one line.

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

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> docker run --name some-postgres -e POSTGRES
Unable to find image 'postgres:latest' locally
latest: Pulling from library/postgres
8ec398bc0356: Pull complete
65a7b8e7c8f7: Pull complete
b7a5676ed96c: Pull complete
3e0ac8617d40: Pull complete
633091ee8d02: Pull complete
b01fa9e356ea: Pull complete
4cd472257298: Pull complete
1716325d7dcd: Pull complete
9b625d69c7c8: Pull complete
74d8b4d9818c: Pull complete
c36f5edbeb97: Pull complete
9b38bb0fb36e: Pull complete
6b5ee1c74b9a: Pull complete
5fcc518252b4: Pull complete
Digest: sha256:3657548977d593c9ab6d70d1ffc43ceb3b5164ae07ac0f542d2ea139664eb6b3
Status: Downloaded newer image for postgres:latest
4586c4f3e83f23ab9a2a932144d54457b5b3b98ad1530713f588f5b7756f20ba
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> █
```

Figure 7-2. PostgreSQL Image Downloaded and Running

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

```
docker ps
```

should show you the number of running containers.

```
PROBLEMS OUTPUT TERMINAL ... 1: powershell + ☐ 🗑
PS D:\APITutorial\NET Core 3.1\CommandAPISolution> docker ps

```

NAMES	CREATED	CONTAINER ID	IMAGE	COMMAND
4586c4f3e83f	postgres	"docker-entrypoint.s..."	40 seconds ago	
Up 39 seconds	0.0.0.0:5432->5432/tcp	some-postgres		

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

Figure 7-3. The Docker PS Command

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

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

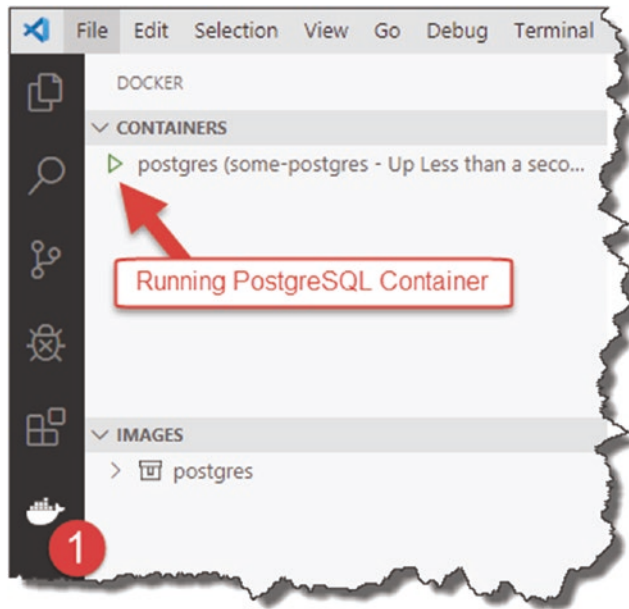


Figure 7-4. *Container Status in VS Code*

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

i When you execute the “`docker run`” command again, assuming there isn’t a later version of the PostgreSQL image on Docker Hub, Docker will not attempt to download a new image; it will simply use the cached copy locally available to you.

Docker Command Prompt

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

```
docker run
```

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

```
--name
```

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

```
-e POSTGRES_PASSWORD=mysecretpassword
```

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

```
-p [internal port] : [external port]
```

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

```
-d
```

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

```
postgres
```

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

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

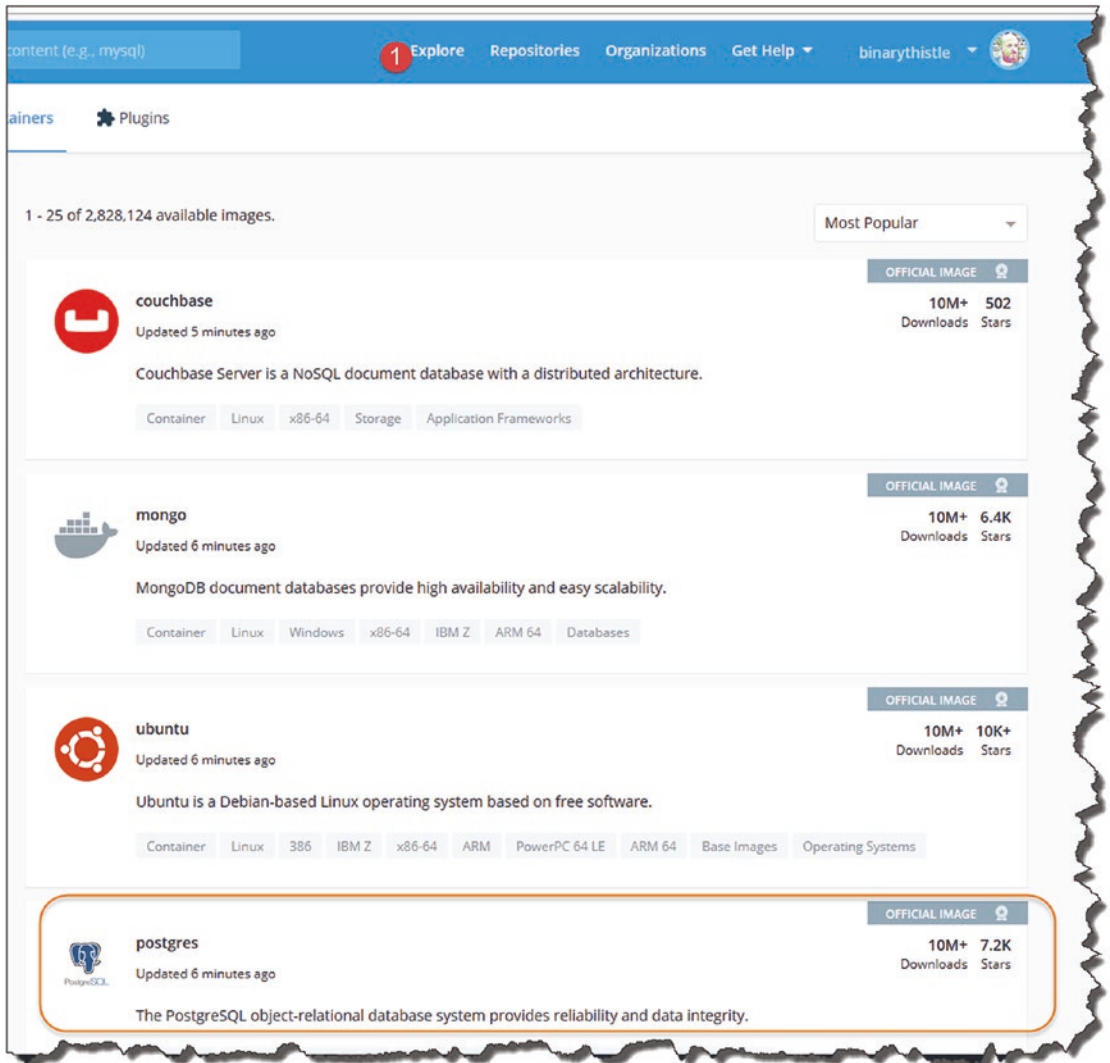


Figure 7-5. Postgres Image on Docker Hub

Here, you can see that Postgres is among the most popular image downloads.

i The other Docker commands you're likely to use are

- **docker start <container Id or Name>**: Start an existing container.
- **docker ps**: List running containers.
- **docker ps - -all**: List all containers that *have* run.
- **docker stop <container Id or Name>**: Stop a running container.

A more detailed description of these and others can be found here:

<https://docs.docker.com/engine/reference/commandline/docker/>

Or of course if you prefer a “graphical” interface to manage your containers, again I suggest the VS Code plugin.

Connecting with DBeaver

Before continuing you should either

1. Have followed along with the given Docker steps and have a running PostgreSQL Docker container
2. Have an instance of PostgreSQL running somewhere else that you can connect to from the machine you're running the API code on

Now, we want to connect in and see what we have.

Open DBeaver and

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

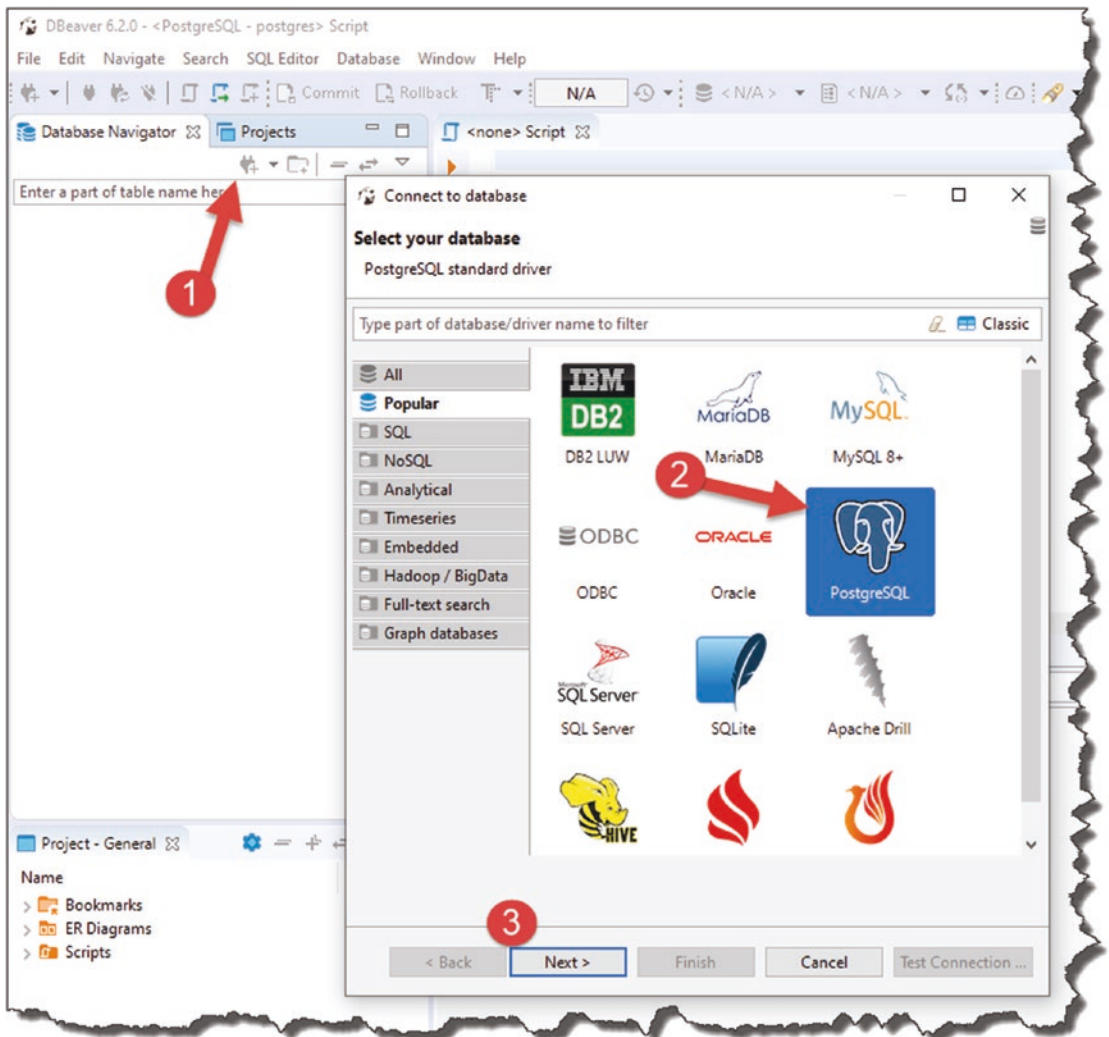


Figure 7-6. PostgreSQL Connection in DBBeaver

You'll then be presented with the Connection Configuration settings for PostgreSQL. On the Main tab, enter the details as appropriate for you; note that the details I have here are good for the PostgreSQL instance I have running in Docker (localhost is fine for the host).

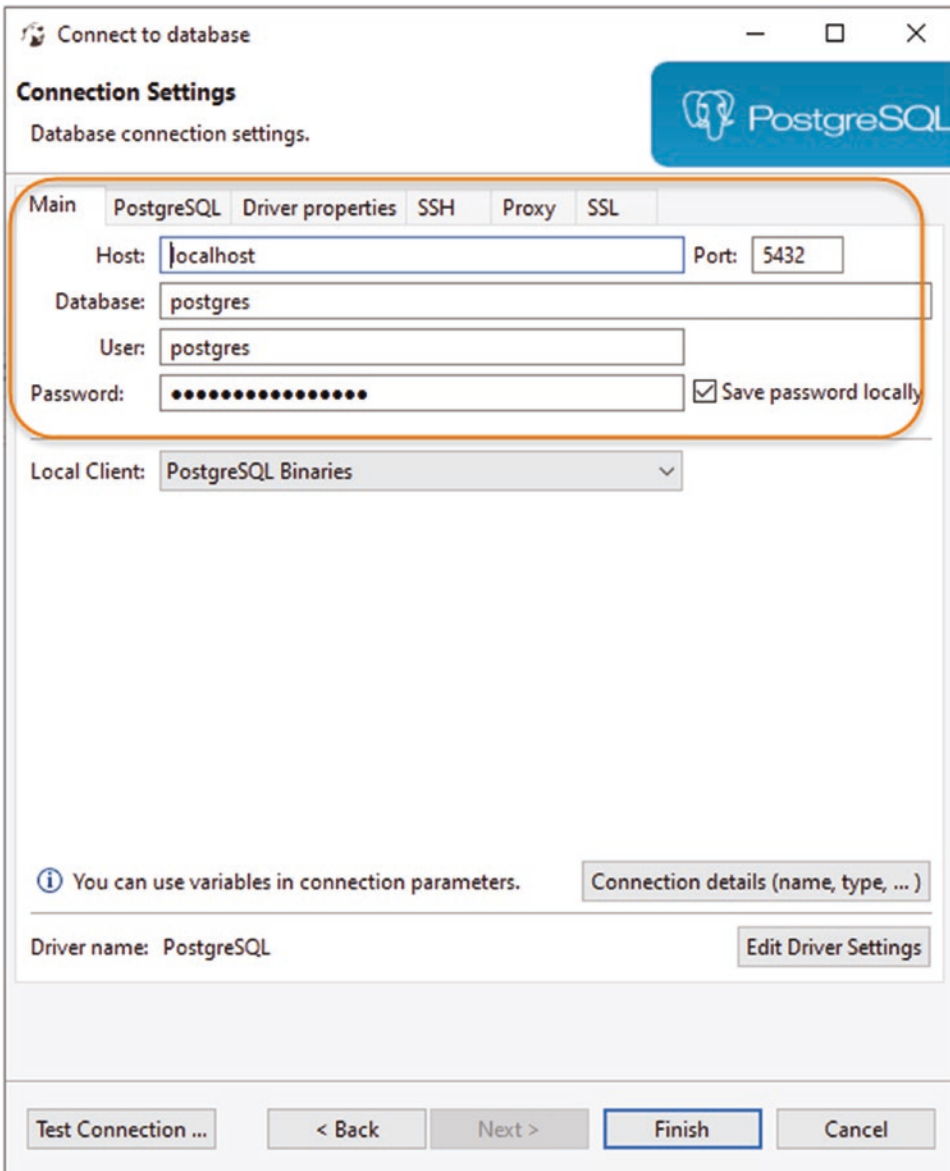


Figure 7-7. *Connecting to PostgreSQL*

Then move over to the PostgreSQL tab and tick “Show all databases.”

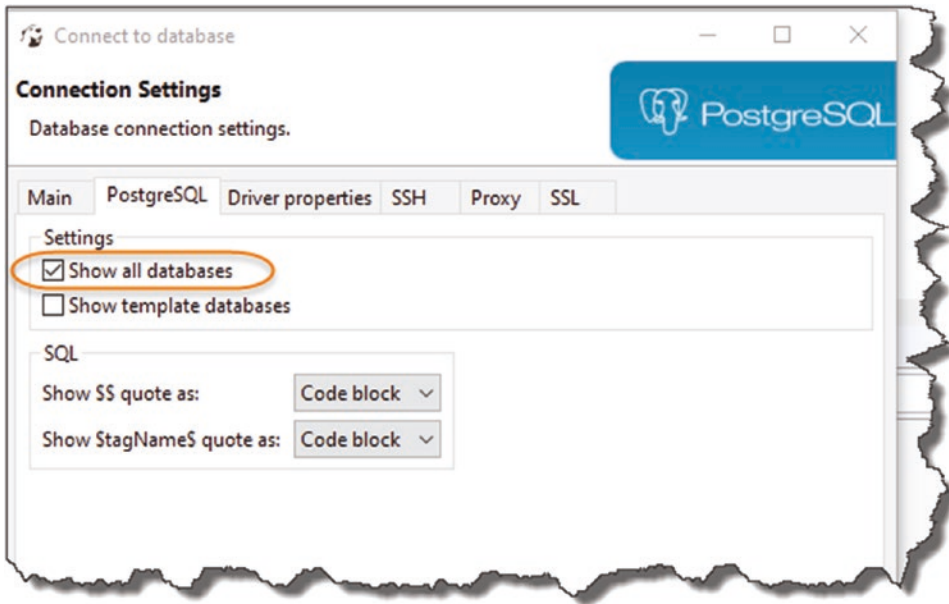


Figure 7-8. Tick Show All Databases

I'd then suggest you test the connection by clicking the "Test Connection..." button.

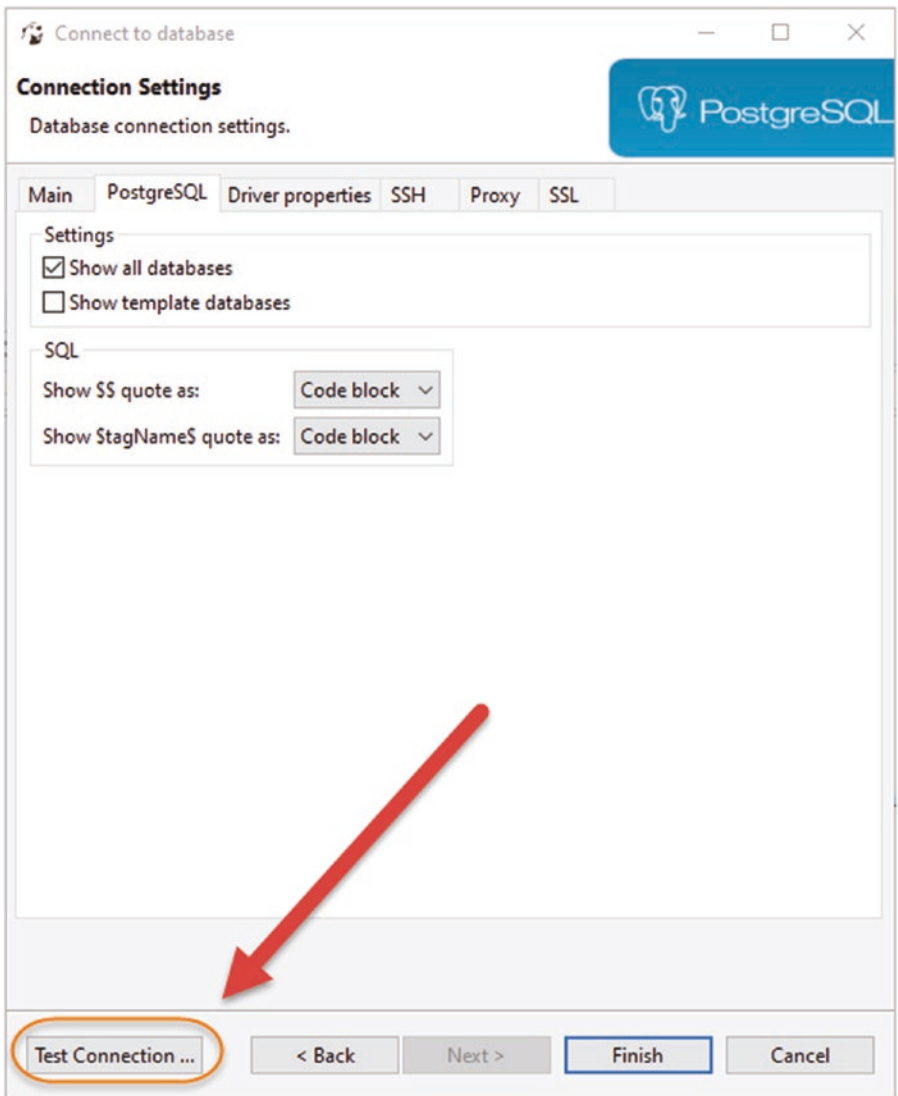


Figure 7-9. Test Connection before moving on

Assuming the connection is successful, you should see something like this.

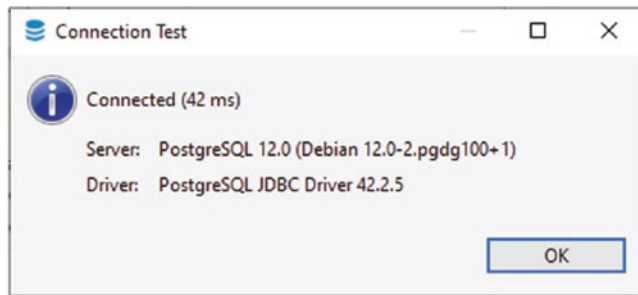


Figure 7-10. *Test Connection Successful*

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

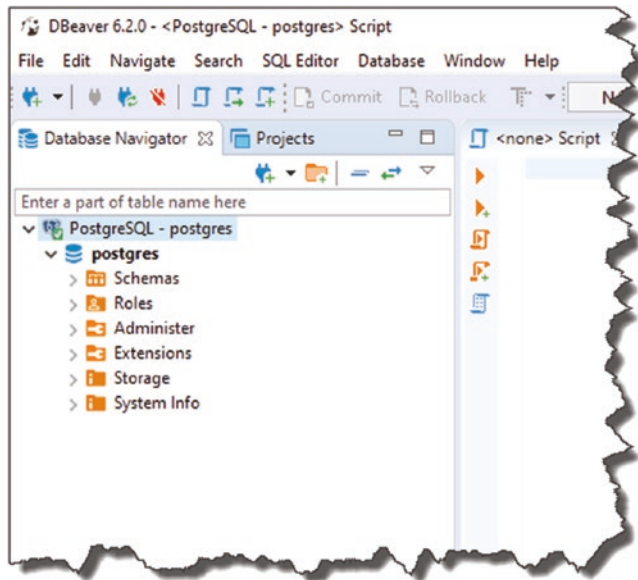


Figure 7-11. *PostgreSQL Connection in DB Beaver*

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

Connection Issues

Connection issues will most usually be down to

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

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

The default "super user" for PostgreSQL is (not surprisingly) called **postgres**. Again, depending on how you installed, the server will depend on whether you set the password value for this. There are articles on how to reset this password (assuming you have administrator/root privileges on the machine you've installed on) if you get stuck.

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

Entity Framework Core

Entity Framework Core (EF Core) is what's termed generically as an Object Relational Mapper (ORM), so what's that, and why should we use it?

To best answer that, I think you would look at the approach you'd need to take to reading and writing data to a database *without* the use of an ORM. In that case, you'd typically

- Need a (relatively low-level) working knowledge of the database schema.
- Have to write (vendor specific) SQL queries to manipulate the data set you wanted.
- Place your results into some kind of semi-proprietary result set object, ensure everything was mapped correctly (DB columns to your object attributes), and iterate through your results.

This all works, but it's a fairly manual process distracting developers from their core focus; surely there is a better way. Enter the ORM.

The What and Why of ORMs

An ORM acts as an “object wrapper” around specific database implementations, meaning

- Developers can use an object-oriented software development approach to data access.
- Developers don’t need to know the nuances of vendor SQL.
- Developers don’t need to perform proprietary mappings from database tables to code-based result sets.

This really equates to the following primary benefits:

- Speed of Development
- Code portability
- Code maintainability

We’ll be using Entity Framework Core as our ORM of choice, but there are alternatives available, so again please remember to distinguish between our *technology implementation* of choice (Entity Framework Core) and the *generic concept* of ORMs.

If that’s all still a bit abstract, as I’ve said before, I think the best way to understand and learn something new is to get our hands dirty and start coding.

Entity Framework Command-Line Tools

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

First, check if you already have them installed; to do so, type
`dotnet ef`

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

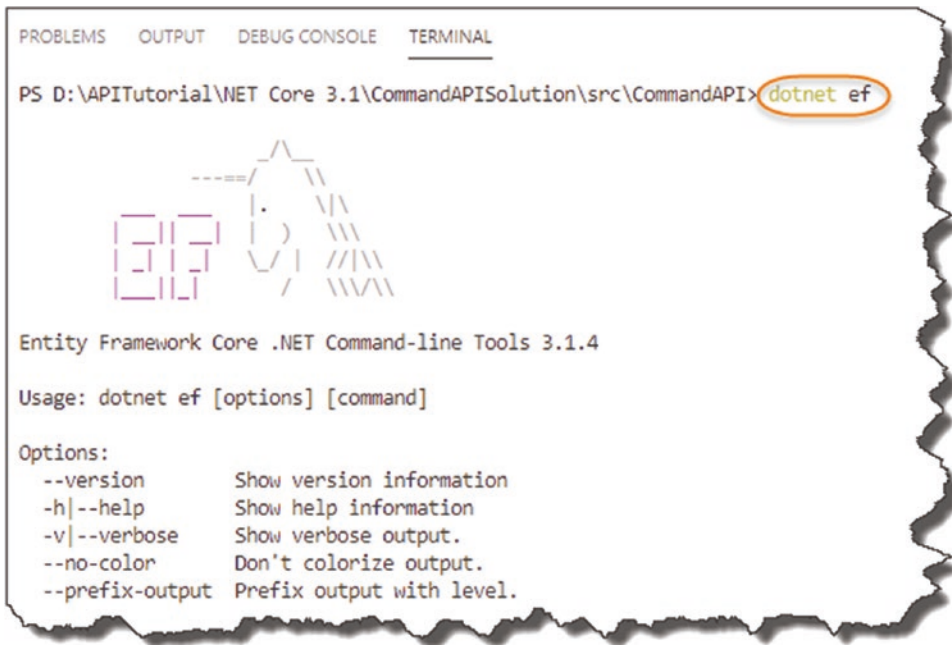


Figure 7-12. Entity Framework Command Line Tools

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

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

and this will make the tools available to you globally.

Create Our DB Context

The next step in producing the data access layer via Entity Framework Core (EF Core) is to create a Database Context Class. The DB Context class acts as a *representation of the Database* and mediates between our data Models and their existence in the DB, as shown in Figure 7-13.

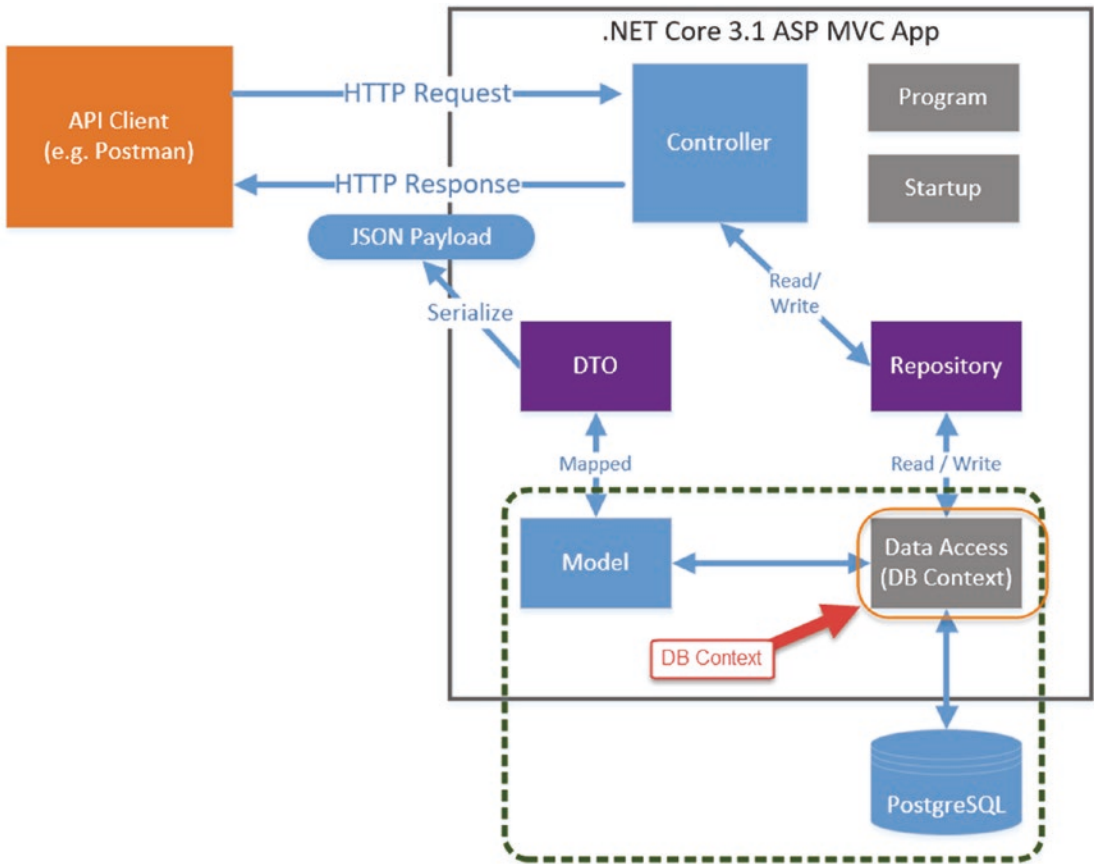


Figure 7-13. *The importance of the DB Context*

As mentioned in Chapter 5, we could use the DB Context direct from the Controller without using a repository. As you are aware though, we will be using both a repository and DB Context in this tutorial.

Reference Packages

In order to use the features of EF Core, we're going to have to add reference three packages in our API Project .csproj file:

- **Microsoft.EntityFrameworkCore:** Primary Entity Framework Core Package
- **Microsoft.EntityFrameworkCore.Design:** Design time components (required for migrations)

- **Npgsql.EntityFrameworkCore.PostgreSQL**: PostgreSQL provider for Entity Framework Core

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

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

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



```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.1">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Npgsql.EntityFrameworkCore.PostgreSQL" Version="3.1.0" />
  </ItemGroup>
</Project>
```

Figure 7-14. Package references added for persistence

With the necessary packages added, we can move on.

i As mentioned, the third package we added

(Npgsql.EntityFrameworkCore.PostgreSQL)

is the EF Core provider for PostgreSQL. If you want to use another database, then you'd add the relevant package here. For example, if you want to use SQL Server, you'd add the following package instead:

```
Microsoft.EntityFrameworkCore.SqlServer
```

We'll create the DB Context class in the "Data" folder along with our repository interface and classes, so create a new file called *CommandContext.cs*, and place it in the *Data* folder; it should look like this.

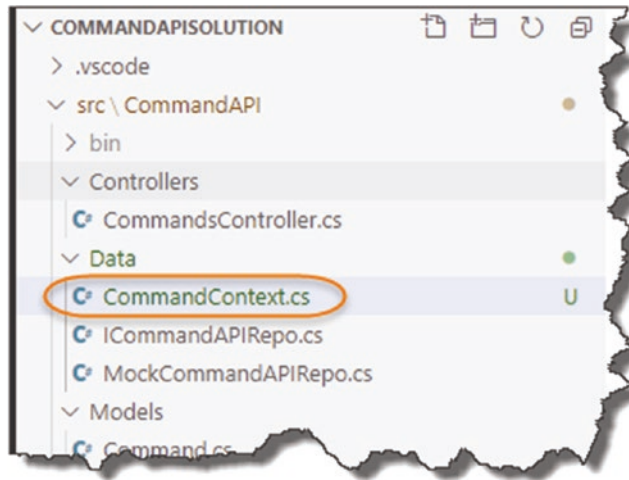


Figure 7-15. Added DB Context

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

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

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

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

Some points of note

- Ensure you have the `EntityFrameworkCore` and `CommandAPI.Models` using statements.
- Our class inherits from `DbContext`.
- It's really important that we create a `DbSet` of `Command` objects (see the following).

i While you can think of the `DbContext` class as a representation of the Database, you could think of a `DbSet` as a representation of a table in the Database. That is, we are telling our `DbContext` class that we want to “model” our `Commands` in the Database (so we can persistently store them as a table).

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

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

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

🎓 Learning Opportunity Try to remember the git commands that you need to issue in order to achieve the items previously discussed – I'm not going to detail them again.

If you can't remember, refer to [Chapter 5](#).

Update `appsettings.json`


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

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

We'll place our DB connection string in our *appsettings.json* file to begin with.

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

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

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

Creating a user can be done in one of two ways using DBeaver:

- SQL Command
- Using the Graphical Interface

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

Create User – SQL

Open DBeaver (make sure you're connected to your PostgreSQL instance), and select SQL Editor ► New SQL Editor from the menu.

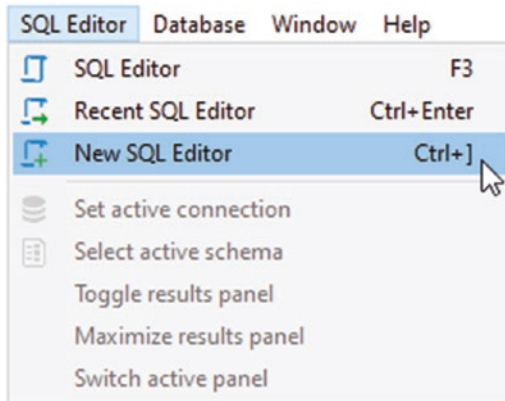


Figure 7-16. *Opening a New SQL Editor*

This should open up a new query widow; then simply enter the following SQL:
create user cmddbuser with encrypted password 'pa55w0rd!' createdb;

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

You can the hold Ctrl + Enter to execute the SQL statement or select “Execute SQL Statement” from the SQL Editor menu.

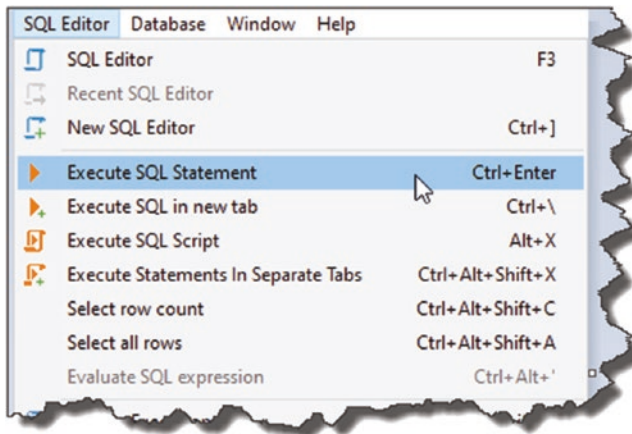


Figure 7-17. *Create our user by executing the SQL*

The command should execute successfully, and if you then expand: postgres ► Roles, you should see your newly created user. If you don't, right-click the “Roles” folder, and select “Refresh.”

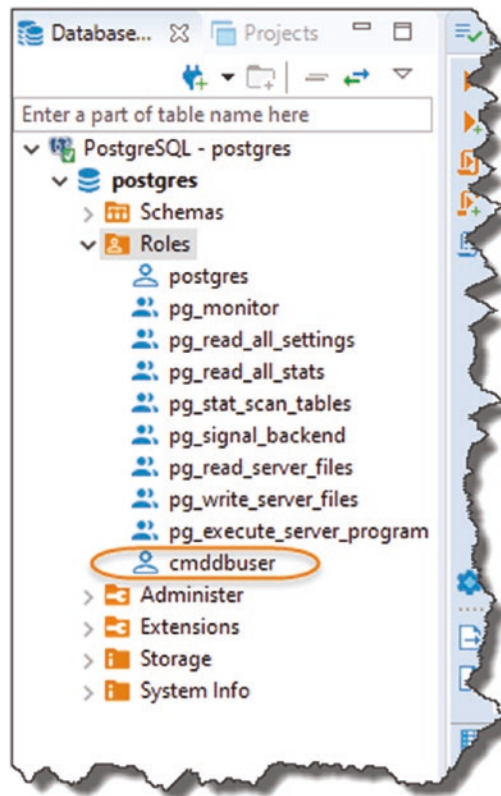


Figure 7-18. Newly created DB User for our API

Right-click the newly created role and select “View Role” (or you can just press F4).

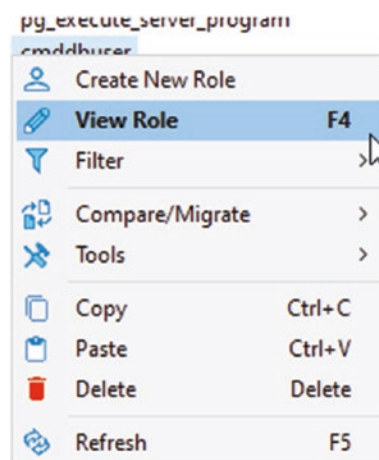


Figure 7-19. View Role Details

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

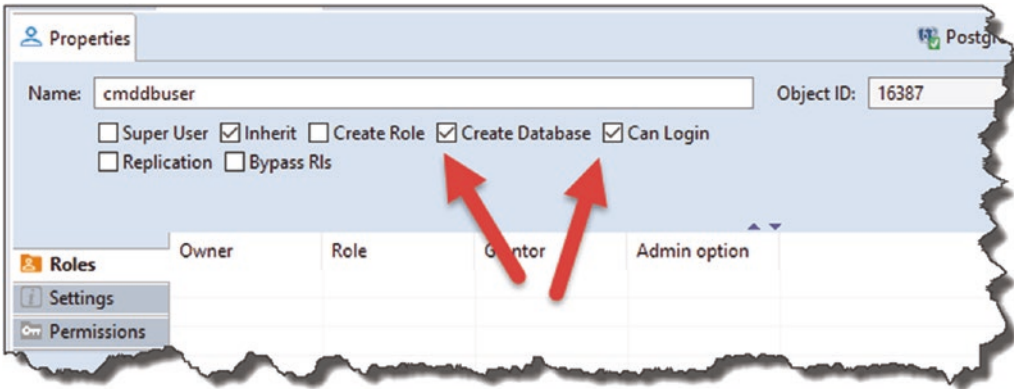


Figure 7-20. Role permissions required for our new user

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

Open *appsettings.json*, and *append* the following json string to the correct point in the file (again make sure you replace the User ID and password to match the user *you* just created):

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

So, your file should look something like this.¹

¹If you get errors copying and pasting, check the double-quote characters and ensure the connection string value is on one line.

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "PostgreSQLConnection": "User ID=cmddbuser;Password"
  }
}

```

Figure 7-21. Connection string in *appsettings.json*

Some points to note about the connection string

- The “name” of the connection string is `PosrgreSqlConnection`.
- The connection string is made up of the following components, separated by a semicolon:
 - **User ID:** The login for our Postgres Server (we created this in the last section).
 - **Password:** The password for our login – stored in plain text – not very secure!²
 - **Host:** The host name of our PostgreSQL server.
 - **Port:** The port our PostgreSQL server is listening on.
 - **Database:** This is our database (or will be our database – it does not exist yet).
 - **Pooling:** Connection pooling (essentially sharing) is being used.

²We will remedy this in the next chapter.

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

Where's Our Database?

As previously mentioned, we have specified the name of our database in our connection string (**CmdAPI**), but the actual database does not yet exist on our server; a quick look at the databases in DBeaver will confirm that.

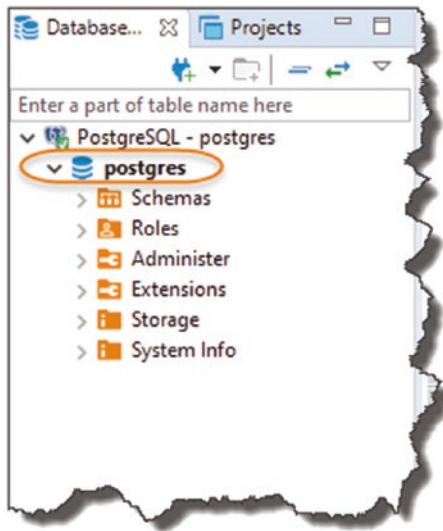


Figure 7-22. *Where's our database?*

We only have the default **postgres** database, but as yet, **CmdAPI** is not there. That is because our database will be created when we perform our first Entity Framework “migration.” I explain what this is later in this section.

Revisit the Startup Class

To recap we have

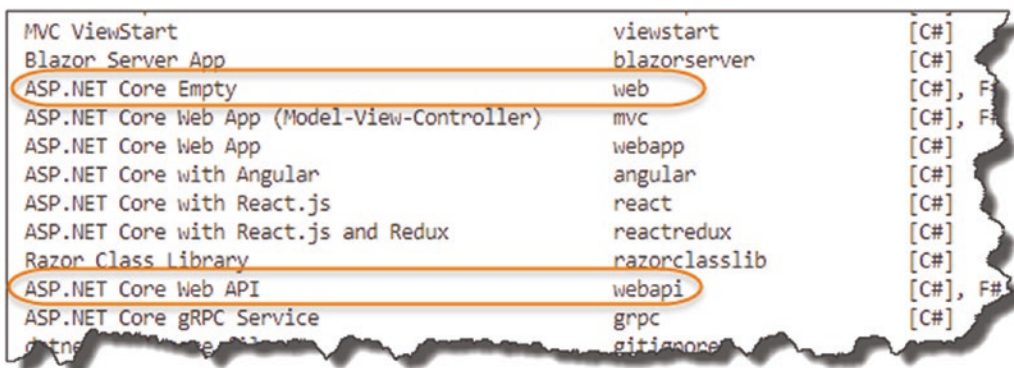
- A Database Server (but actually no **CmdAPI** “database” as yet!)
- A Model (Command)
- DbContext (CommandContext)
- DBSet (CommandItems)
- Connection String to our database server

The last few things we have to do are

- Point our DbContext class to the connection string (currently it’s not aware of it).
- “Register” our DbContext class in Startup ► ConfigureServices so that it can be used throughout our application as a “service” – seem familiar?

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

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



MVC ViewStart	viewstart	[C#]
Blazor Server App	blazorserver	[C#]
ASP.NET Core Empty	web	[C#], F#
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F#
ASP.NET Core Web App	webapp	[C#]
ASP.NET Core with Angular	angular	[C#]
ASP.NET Core with React.js	react	[C#]
ASP.NET Core with React.js and Redux	reactredux	[C#]
Razor Class Library	razorclasslib	[C#]
ASP.NET Core Web API	webapi	[C#], F#
ASP.NET Core gRPC Service	grpc	[C#]
ASP.NET Core gRPC Service	grpc	[C#]
ASP.NET Core gRPC Service	grpc	[C#]
ASP.NET Core gRPC Service	grpc	[C#]
ASP.NET Core gRPC Service	grpc	[C#]
ASP.NET Core gRPC Service	grpc	[C#]
ASP.NET Core gRPC Service	grpc	[C#]
ASP.NET Core gRPC Service	grpc	[C#]
ASP.NET Core gRPC Service	grpc	[C#]
ASP.NET Core gRPC Service	grpc	[C#]

Figure 7-23. .NET Core Project Templates

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

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

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

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

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

I’ve shown the new sections in context of the whole file here.

```

using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration; 1
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace CommandAPI
{
    public class Startup
    {
        public IConfiguration Configuration {get;}
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers();
        }
    }
}

```

Figure 7-24. Using Dependency Injection to Add the Configuration API

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

Does this pattern seem familiar? If not, maybe return to Chapter 6 and review. What this code provides for us is access to the “Configuration API” (via an implementation of the `IConfiguration` interface), which means that we can now access the configuration stored in (among other places) the *appsettings.json* file. In particular, it means we can read in our connection string and pass it to the DB Context.

For more information on the `IConfiguration` interface, refer to the Microsoft Build Docs.³

The last thing we have to do is register our `DbContext` in the `ConfigureServices` method and pass it the connection string (via the configuration API). Add the following using directive to your `Startup` class:

- `using Microsoft.EntityFrameworkCore;`

³<https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.configuration.iconfiguration?view=dotnet-plat-ext-3.1>

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

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

    services.AddControllers();
}
```

To put those changes in context, they are shown here.

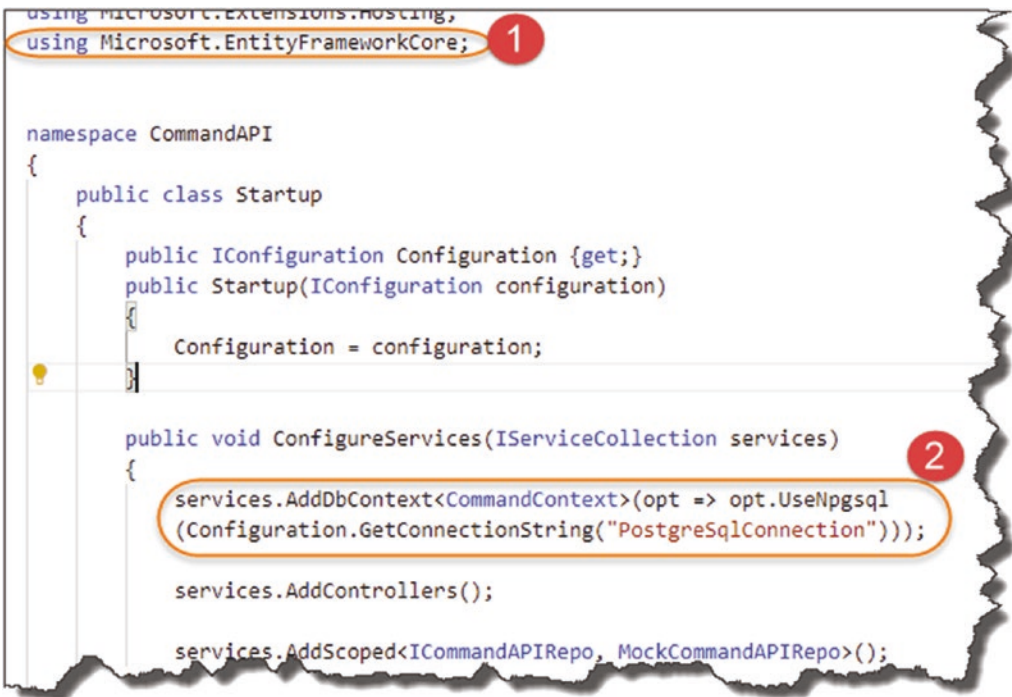


Figure 7-25. Registering our DB Context with our Services Container

You'll observe the following:

1. We include a new using directive.

2. We register our `CommandContext` class as a solution-wide `DbContext` (in the Service Container), and we point it to the connection string (`PostgreSqlConnection`) that is contained in our ***appsettings.json*** file. This is accessed via our Configuration object.

i If you're using a different database to PostgreSQL, you'd need to change the code in point 2; specifically, you'd swap out the

```
opt.UseNpgsql
```

For something else, for example, in the case of SQL Server, you'd use

```
opt.UseSqlServer
```

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

Create and Apply Migrations

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

Code First vs. Database First

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

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

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

Go to your command line, and ensure that you are “in” the API project folder (*CommandAPI*), and type the following (hitting Enter when you’re done):

```
dotnet ef migrations add AddCommandsToDB
```

Now all being well, a number of things should have happened here.

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



Figure 7-26. Create our migration files ready to run

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

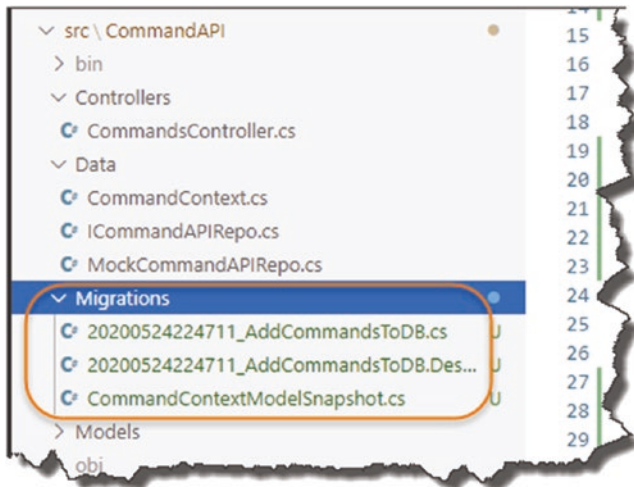


Figure 7-27. Newly created migrations

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

```
20200524224711_AddCommandsToDB.cs
```

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

```

public partial class AddCommandsToDB : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder) 1
    {
        migrationBuilder.CreateTable(
            name: "CommandItems", 2
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                .Annotation("Npgsql:ValueGenerationStrategy", NpgsqlValueGenerationStrategy.IdentityByDefaultColumn), 3
                HowTo = table.Column<string>(maxLength: 250, nullable: false), 4
                Platform = table.Column<string>(nullable: false),
                CommandLine = table.Column<string>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_CommandItems", x => x.Id);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder) 5
    {
        migrationBuilder.DropTable(
            name: "CommandItems");
    }
}

```

The data annotations we added to our Command Model have been replicated.

Figure 7-28. Contexts of our Migrations File

1. An “Up” method. This method is called to create new items.
2. The creation of a table “CommandItems” (where does this name come from?).
3. Database provider specific annotations/instructions.
4. Our table columns; note the data annotations we added to our model have been replicated.
5. A “Down” method. Used to roll back the changes made in the Up method.

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

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

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

It failed.

You can examine an “SQL Server” migrations file and look for the differences in the Source Code for an older project here on GitHub.⁴

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

Note At this stage, we *still do not* have the **CmdAPI** database created; that comes next.

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

```
dotnet ef database update
```

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

⁴<https://github.com/binarythistle/Complete-ASP.NET-Core-API-Tutorial>

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

Figure 7-29. Successfully run our migration

i Tip If you get an error at this stage, in my experience, it usually always has to do with database “connection” issues. So, I’d check:

1. Can you still connect to the PostgreSQL server using DBeaver?
2. Double-check the formatting of the connection string. For example, copy and paste it into something like jsoneditoronline.org to check for syntactical JSON errors.
3. Double-check the values you’ve put into the connection string. For example, passwords are case-sensitive.
4. Double-check if you’re using the correct connection string “name” when you’re setting up the `DbContext` on the `ConfigureServices` method.
5. Perform a `dotnet build` to check that there are no syntax errors in the code (this is actually run when you do a `dotnet ef database update` – but it’s worth checking separately).
6. Check that (a) the database user you created exists on the server and (b) check that it has the necessary permissions.

All going well, a number of things happen here:

1. Our database (**CmdAPI**) is created as it did not yet exist on the target server.

2. A table called **_EFMigrationsHistory** is created; this just stores the IDs of the migrations that have been run and allows Entity Framework to both roll back migrations to a certain point or correctly run migrations on a new endpoint server.
3. Our **CommandItems** table is created which is the persistent equivalent of our Command model.

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



Figure 7-30. Our Database and Table have been created

1. **CmdAPI** database
2. **CommandItems** table

Adding Some Data

Up until now, we’ve been using hard-coded mock data in our API code. With the establishment of our Database and DB context, we can now start to add some “real” data for use in by our API.

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

In terms of the data we should put in, I’d like to circle back to the creation and updating of the database and table; we used the following two commands:

- `dotnet ef migrations add`
- `dotnet ef database update`

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

ID ⁵	HowTo	Platform	CommandLine
1	Create an EF Migration	EF Core CLI	<code>dotnet ef migrations add</code>
2	Apply Migrations to DB	EF Core CLI	<code>dotnet ef database update</code>

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

- Open DBeaver and connect to the server.
- From the SQL Editor Menu, select “New SQL Editor.”

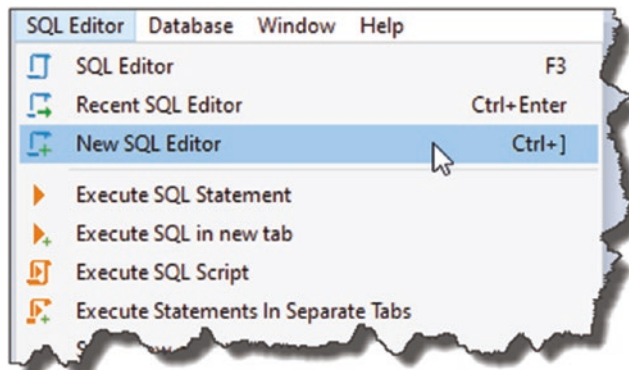


Figure 7-31. Add new Query window

⁵You do not need to provide a value for ID when you add data to the database; this is auto-created by PostgreSQL for us.

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

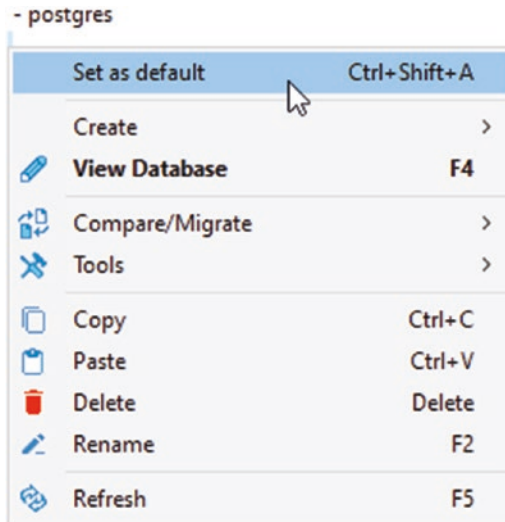


Figure 7-32. Set Default Database

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

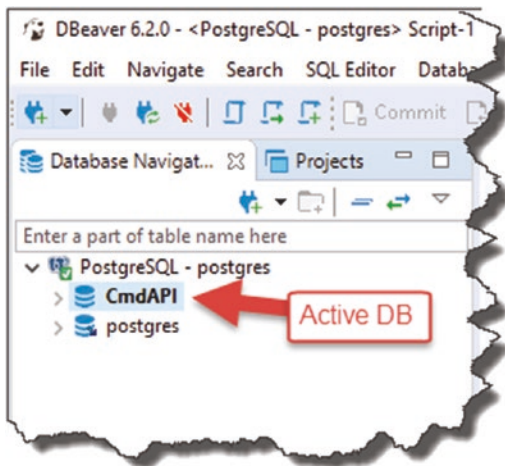


Figure 7-33. Our Default DB has been set

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

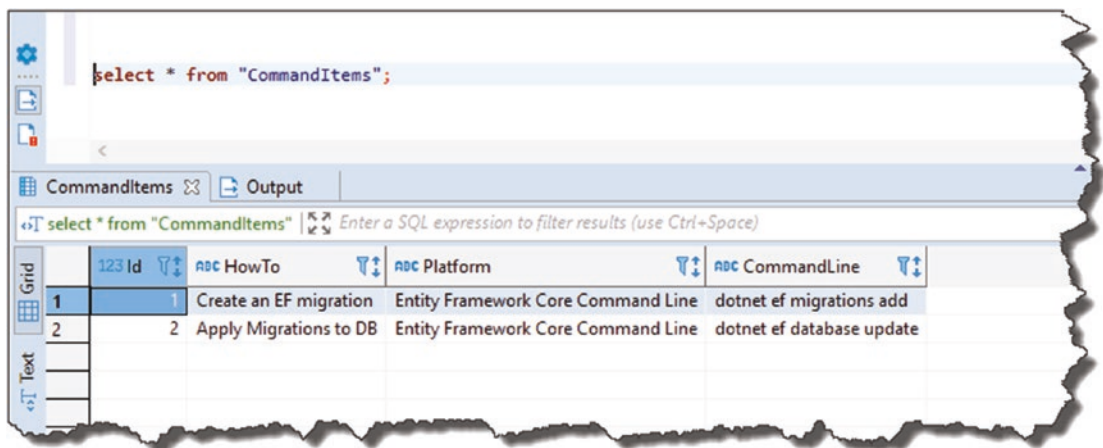
```
insert into "CommandItems" ("HowTo", "Platform", "CommandLine")
values ('Create an EF migration', 'Entity Framework Core Command Line',
'dotnet ef migrations add');
```

```
insert into "CommandItems" ("HowTo", "Platform", "CommandLine")
VALUES ('Apply Migrations to DB', 'Entity Framework Core Command Line',
'dotnet ef database update');
```

After that, press Ctrl+Enter, or select “Execute SQL Statement” from the SQL Editor menu to run the SQL – this should insert the lines into our database. To check this, clear the SQL from the window (otherwise, if you execute it again, it’ll insert two more rows, effectively duplicating the data), and type

```
select * from “CommandItems”;
```

This should return something like the following.



Grid	123 Id	HowTo	Platform	CommandLine
1	1	Create an EF migration	Entity Framework Core Command Line	dotnet ef migrations add
2	2	Apply Migrations to DB	Entity Framework Core Command Line	dotnet ef database update

Figure 7-34. Our two new rows of “real” data

If you read my blog post on Entity Framework,⁶ you’ll have noticed by now that the commands used in that tutorial are different to those used here. That’s because in that

⁶<https://dotnetplaybook.com/introduction-to-entity-framework/>

online tutorial, we’re using the “Package Manager Console” in Visual Studio to issue commands for Entity Framework (not Entity Framework Core/.NET Core Command line) – quite confusing I know!

I think, therefore, just to labor that point, let’s add two new command-line prompts in our DB.

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

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

After executing the SQL INSERT commands, perform another SELECT "all" (i.e., SELECT * ...), and you should see the following.



Figure 7-35. Commands in the DB

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

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

Tying It Altogether

A quick progress check to see where we are with our architecture, and this time I've just highlighted the *interaction* that we **have not yet** implemented.

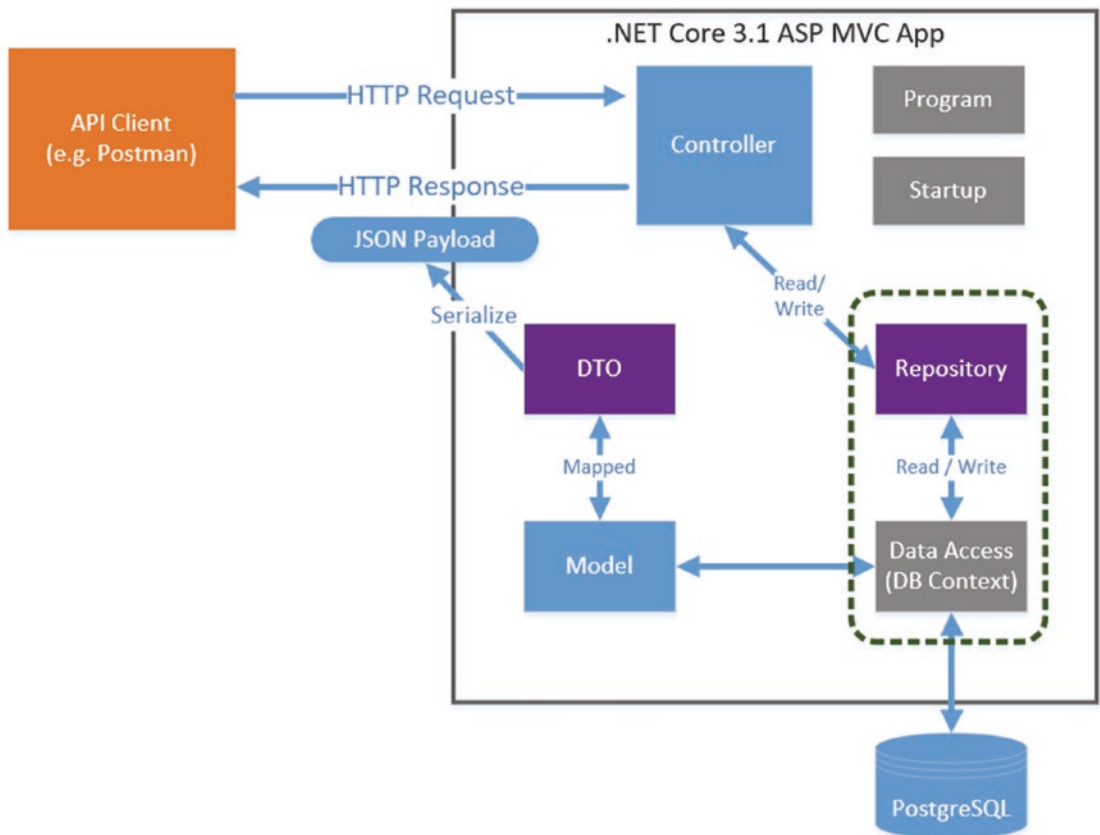


Figure 7-36. We need to make use of our DB Context Class

Currently, we are using our repository to return data using a mock implementation, so what we need to do next to make use of our “real” data (and, therefore, DB Context) is

- Create a new implementation of our repository interface (to use the DB Context).
- “Swap” out our existing mock implementation for our new one.

I’ve depicted our current state with our desired end state in Figure 7-37.

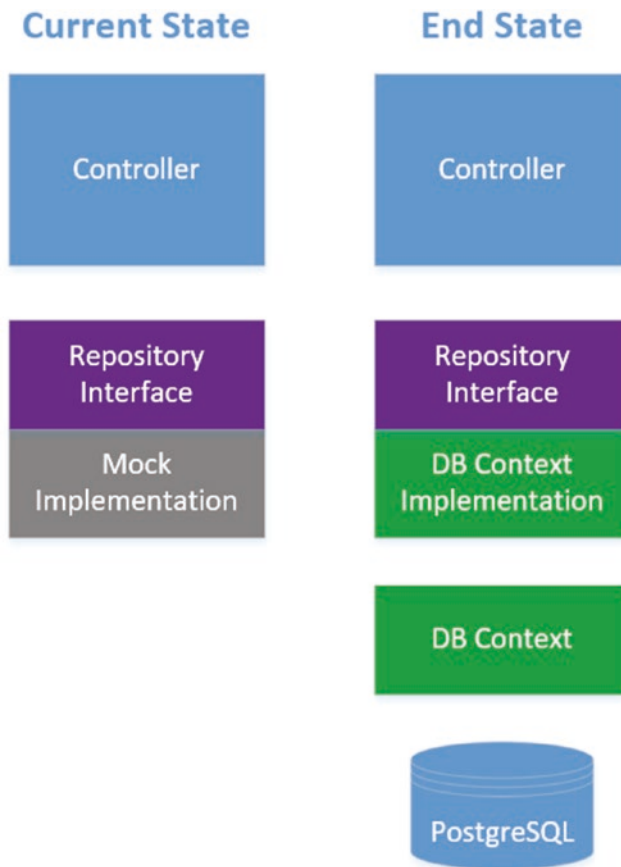


Figure 7-37. *Current State vs. End State*

Create a New Repository Implementation

The first thing we want to do is create a new *concrete implementation* of our `ICommandAPIRepo` interface, so add a new file to the **Data** folder in our API project, and call it ***SqlCommandAPIRepo.cs*** as shown in Figure 7-38.

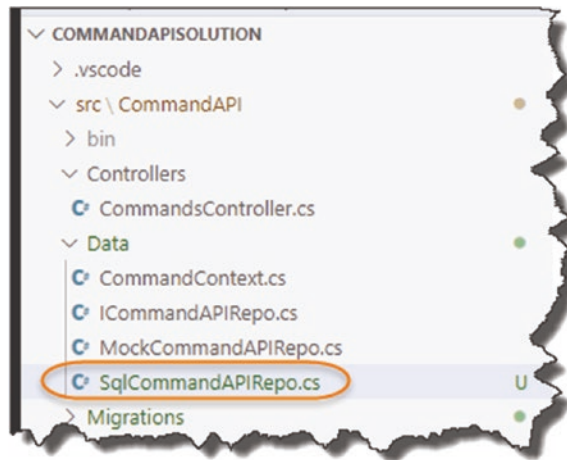


Figure 7-38. New Concrete implementation class

Add the following code to that file:

```
namespace CommandAPI.Data
{
    public class SqlCommandAPIRepo : ICommandAPIRepo
    {
    }
}
```

As you've done before with our mock implementation in Chapter 6, place your cursor "in" the `ICommandAPIRepo` statement, and press

`CTRL + .`

To auto-generate the template implementation code for our API, see Figure 7-39.

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

namespace CommandAPI.Data
{
    public class SqlCommandAPIRepo : ICommandAPIRepo
    {
        public void CreateCommand(Command cmd)
        {
            throw new NotImplementedException();
        }

        public void DeleteCommand(Command cmd)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<Command> GetAllCommands()
        {
            throw new NotImplementedException();
        }

        public Command GetCommandById(int id)
        {
            throw new NotImplementedException();
        }

        public bool SaveChanges()
        {
            throw new NotImplementedException();
        }

        public void UpdateCommand(Command cmd)
        {
            throw new NotImplementedException();
        }
    }
}
```

Figure 7-39. Auto-generated interface implementation code

For now, we’re just going to implement the same two methods that we implemented in our mock implementation:

- GetAllCommands
- GetCommandById

To begin, we're going to use Constructor Dependency Injection to inject our DB Context into our `SqlCommandAPIRepo` class (so we can use it). Remember that we registered our DB Context class with the Service Container in the Startup class so it is available for "injection."

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<CommandContext>(opt => opt.UseNpgsql(
        Configuration.GetConnectionString("PostgreSQLConnection")));
    services.AddControllers();
    services.AddScoped<ICommandAPIRepo, MockCommandAPIRepo>();
}
```

Figure 7-40. We registered our DB Context Class with the Service Container in the startup class

So, to inject our DB Context into our new concrete repository class, add the following class constructor to `SqlCommandAPIRepo`:

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

namespace CommandAPI.Data
{
    public class SqlCommandAPIRepo : ICommandAPIRepo
    {
        private readonly CommandContext _context;

        public SqlCommandAPIRepo(CommandContext context)
        {
            _context = context;
        }
    }
}
```

Again, this pattern should be familiar to you now:

- A DB Context instance is injected in via our constructor (as context).
- We then assign context to a private read-only field (`_context`) that we can use in the rest of the `SqlCommandAPIRepo` class.

We then need to update our two methods as shown by the code here (making sure to add the using statement to using `System.Linq` at the top of the file):

```
using System.Linq;
.
.
.
public IEnumerable<Command> GetAllCommands()
{
    return _context.CommandItems.ToList();
}

public Command GetCommandById(int id)
{
    return _context.CommandItems.FirstOrDefault(p => p.Id == id);
}
```

To put the changes in context, I've shown them here.

```

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

namespace CommandAPI.Data
{
    public class SqlCommandAPIRepo : ICommandAPIRepo
    {
        private readonly CommandContext _context;

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

        public void CreateCommand(Command cmd)
        {
            throw new System.NotImplementedException();
        }

        public void DeleteCommand(Command cmd)
        {
            throw new System.NotImplementedException();
        }

        public IEnumerable<Command> GetAllCommands()
        {
            return _context.CommandItems.ToList();
        }

        public Command GetCommandById(int id)
        {
            return _context.CommandItems.FirstOrDefault(p => p.Id == id);
        }

        public void SaveChanges()
    }
}

```

Figure 7-41. Concrete Implementation of our Repository

To review

1. Class constructor utilizing the injection of our DB Context.
2. We reference “CommandItems” on our DB Context (_context) and return as a List of Command objects.
3. We call the FirstOrDefault method on our “CommandItems” to return a Command object (if one exists) that matches our desired ID.

As you can see, we can reference our object collections (in this case, we just have Commands) via our DB Context with relative ease (this is the power of the “ORM”).

That’s our new repository implementation complete (for now – we’ll complete the other methods later). All that remains to do is to change our Service Container registration in the Startup class as shown next – it goes without saying that you should make these changes in your own code too.

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

    services.AddControllers();

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

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
```



Figure 7-42. We swap out our mock implementation for our new Implementation

And that’s it! That’s how easy it is to swap out implementations of our repository – we didn’t have to change a single line of code in our Controller. While our codebase is quite small, you can imagine in situations where we’re making use of our repository elsewhere in our code just how powerful (and convenient) this is.

Let’s save everything and test to see if this is working, so

```
dotnet build
```

Assuming all is well, let’s run

```
dotnet run
```

And let’s trigger some calls in Postman.

Get All Command Items

Using the URI `http://localhost:5000/api/commands/` along with GET in Postman yields the following.

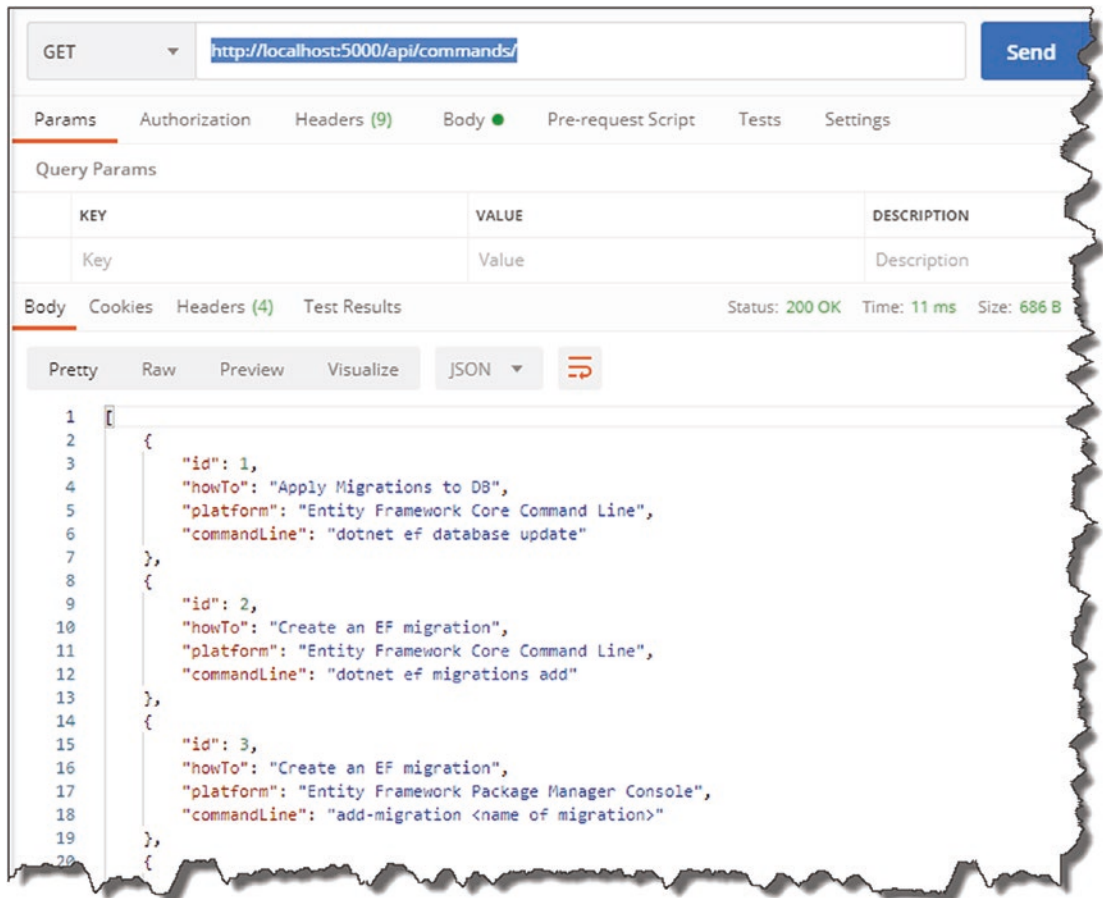


Figure 7-43. Our API working with database derived data

Get A Single Command (Existing)

Using the URI `http://localhost:5000/api/commands/1` along with GET in Postman yields the following.

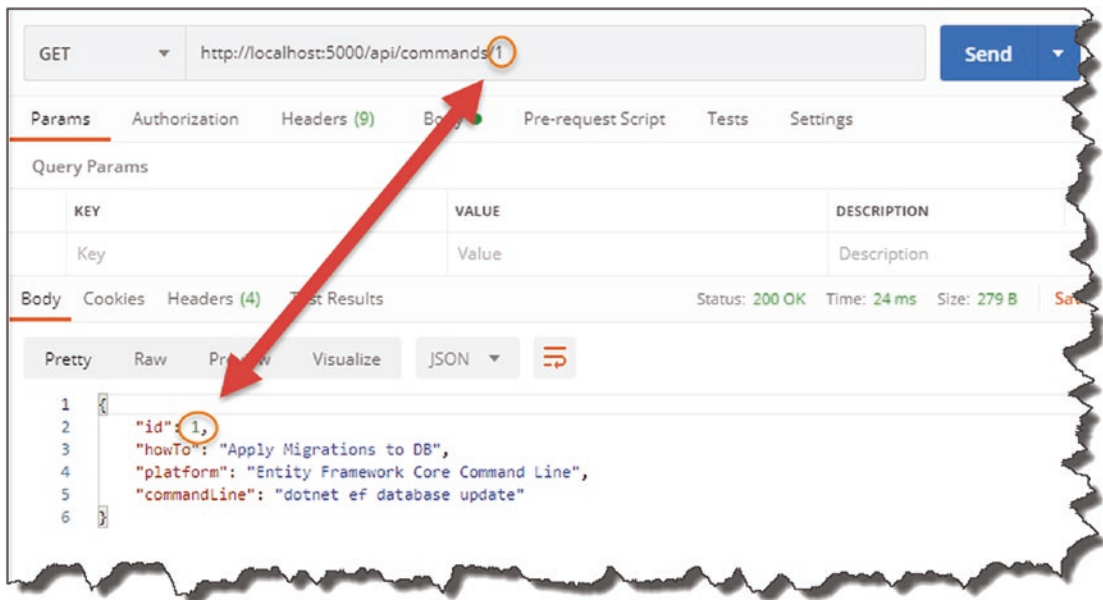


Figure 7-44. Returning a Single Resource

Get A Single Command (Not Existing)

Using the URI `http://localhost:5000/api/commands/67` along with GET in Postman yields the following.

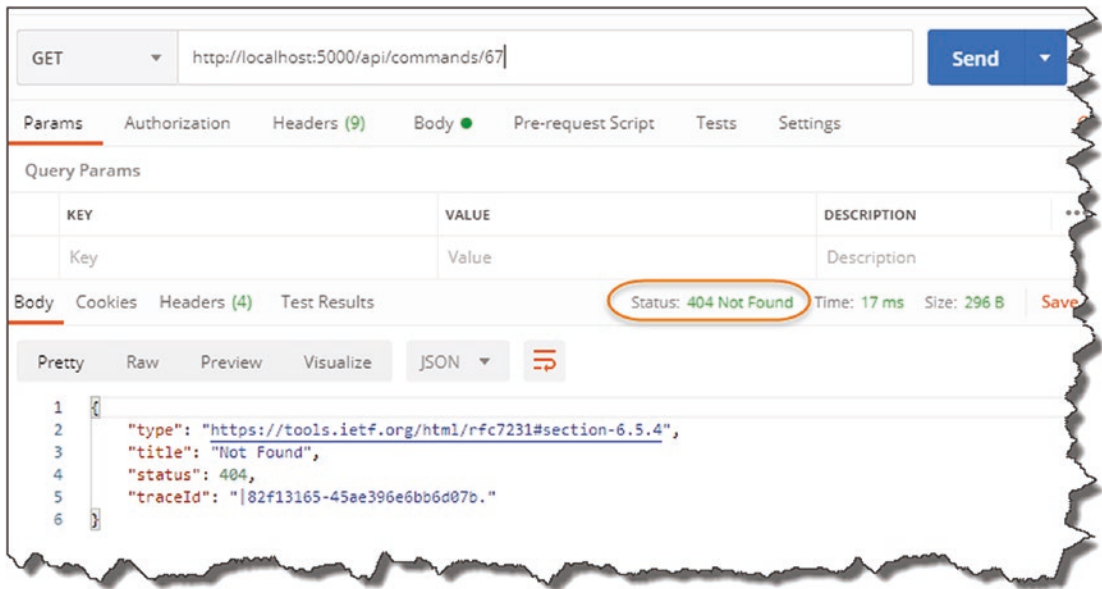


Figure 7-45. 404 Not Found Error

🎉 Celebration Checkpoint Possibly the most significant celebration in the whole book – well done! You’ve basically built a data drive API in .NET Core!

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

Wrapping Up the Chapter

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

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

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

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

Redact Our Login and Password

If your API is still running, stop it (Ctrl + C), and edit the connection string in your *appsettings.json* file, redacting or changing the values for User ID and Password to something nonsensical (note that if you run the API again, it will fail when we come to retrieve data!). See Figure 7-46.



Figure 7-46. Nonsense User ID and Password

Save the file, then perform the three steps to Add/Track, Commit, and Push your code to GitHub (remember to do this at the Solution level: *CommandAPISolution* folder).

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

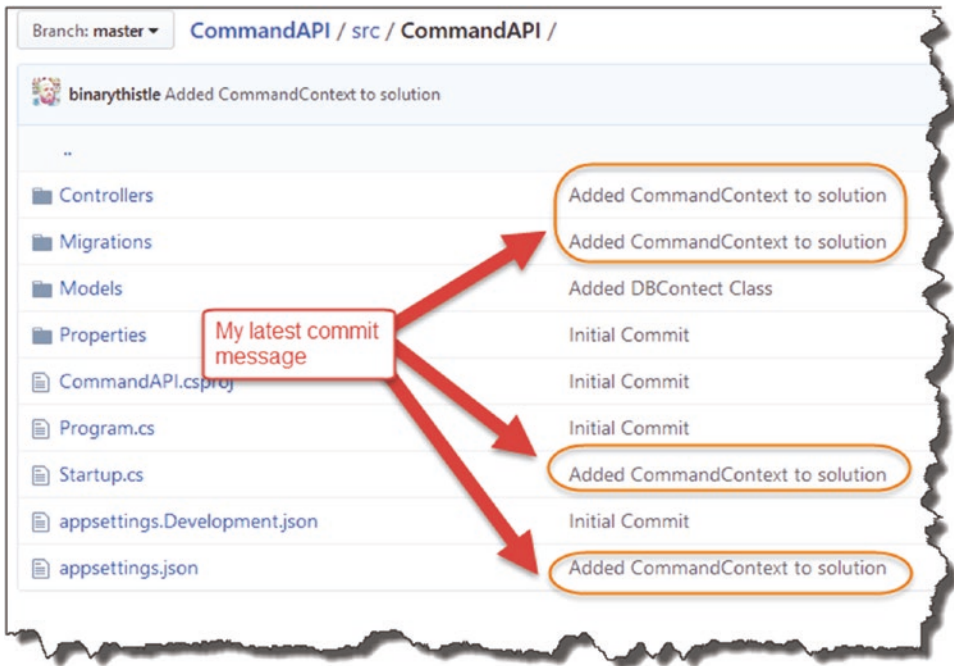


Figure 7-47. Latest Commits

See the *appsettings.json* file as it exists publicly on GitHub.



Figure 7-48. Redacted User ID and Password on GitHub

We have two major problems now:

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

Clearly, we can’t publish user IDs and password to GitHub; even if we made the GitHub repository private, this is still terrible practice. We need a way of keeping these details secret.

CHAPTER 8

Environment Variables and User Secrets

Chapter Summary

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

When Done, You Will

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

Environments

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

- Stressful for you as a developer

- Showing great irresponsibility as an application owner
- Potentially impactful to the end user

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



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

I know this from bitter experience.

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

You will typically never make code changes directly in production; indeed deployments and changes to production should be done, where possible, in as automated (and trackable) a way as possible, where the “human hand” doesn't intervene to any large extent.

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



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

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

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

Our Environment Setup

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

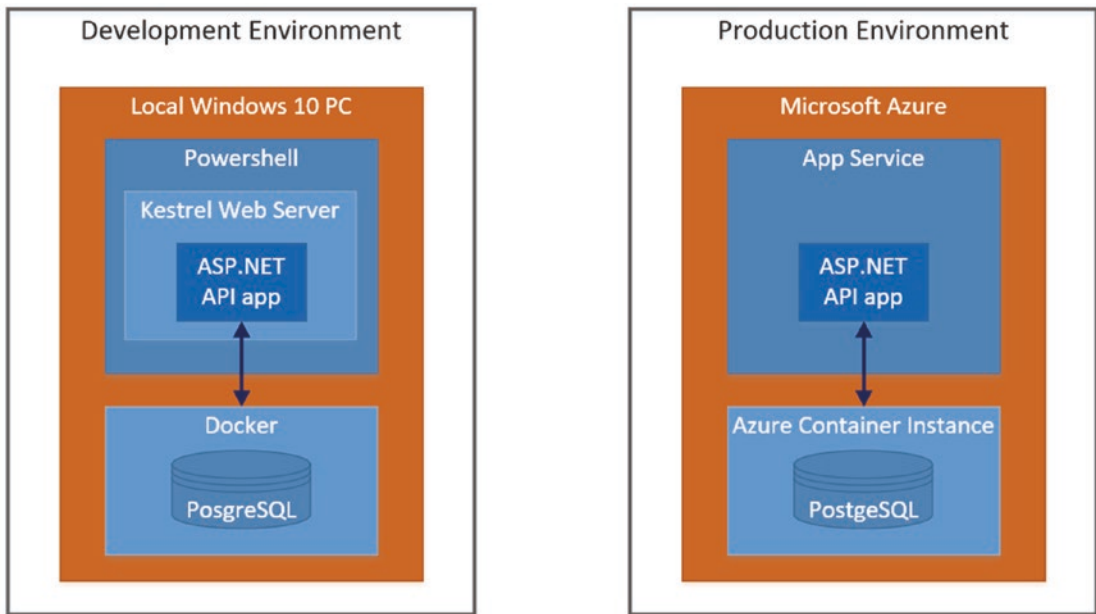


Figure 8-1. *Development and Production Environments*

As you can see, the “components” that are there are effectively the same; it’s really only the underlying platform that is different (a local Windows PC vs. Microsoft Azure).

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

The Development Environment

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

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

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

A fuller discussion on multiple environments in ASP.NET Core can be found here.¹

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

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:12662",
      "sslPort": 44343
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "CommandAPI": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Figure 8-2. *LaunchSettings.json* File

¹<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments>

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

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

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

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

- *launchSettings.json* is read (if available).
- `environmentVariables` settings override system/OS-defined environment variables.
- The hosting environment is displayed.

For example, see Figure 8-3.

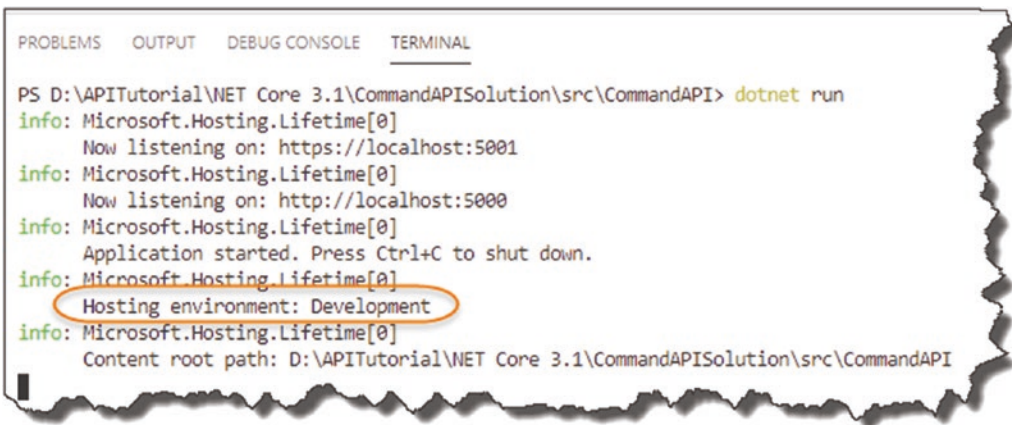


Figure 8-3. Our environment is set to Development

So What?

At this stage I hear you all saying, “*Yeah that’s great and everything, but so what?*”

Good question; I’m glad you asked that question!²

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

- Endpoints (e.g., Server Name/IP address, etc.)
- Different log-in credentials, etc.

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

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

Make the Distinction

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

1. “Manually” determine the environment in your code, and take the necessary action.
2. Leverage the power and behavior of the .NET Core *Configuration API*.

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

²Beware when you get this response from either Salesman, an Executive, or Politician – it usually means that they don’t know the answer and will either deflect the question somewhere else or lay on some major bullsh!t.

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

Figure 8-4. Code-based determination of environment

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

Order of Precedence

OK, so we're going to leverage from the behavior of the .NET Core *Configuration API* to change the config as required for our two different environments (we've already made use of this when we configured the connection string for the DB Context).

Let's quickly revisit the Program Class startup sequence for our app as covered in Chapter 4.

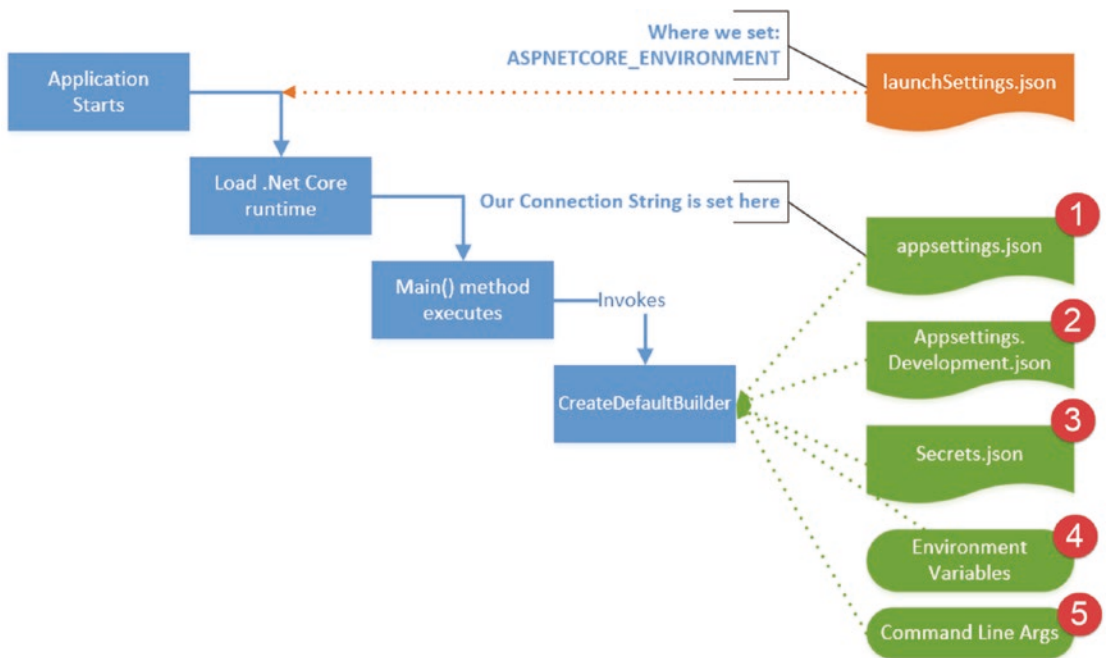


Figure 8-5. Configuration sources and order of preference

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

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

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

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

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

So be careful!

I've referenced a great Blog Post on the Order of Precedence with Configuring ASP.NET Core here,³ for a further overview.

It's Time to Move

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

- Go into your ***appsettings.json*** file, and *copy* the `ConnectionStrings` key-value pair that contains our `PostgreSqlConnection` connection string.
- Make sure you have the **correct values**⁴ for User ID and Password.
- Insert this JSON segment into the ***appsettings.Development.json*** file – see Figure 8-6.

³<https://devblogs.microsoft.com/premier-developer/order-of-precedence-when-configuring-asp-net-core/>

⁴Remember we had changed them at the end of the last chapter to avoid publishing them to GitHub.

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

```

src > CommandAPI > {} appsettings.Development.json > {} ConnectionStrings
1  {
2  |   "Logging": {
3  |     |   "LogLevel": {
4  |     |     |   "Default": "Information",
5  |     |     |   "Microsoft": "Warning",
6  |     |     |   "Microsoft.Hosting.Lifetime": "Information"
7  |     |     |   }
8  |     |     } ,
9  |     |   "ConnectionStrings":
10 |     |   {
11 |     |     |   "PostgreSQLConnection": "User ID=cmddbuser;Password=pa55w0rd!;
12 |     |     |   }
13 |     |   }
14 |   }

```

Figure 8-6. *Appsettings.Development.json*

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

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

Let's Break It

OK, so to prove the point we were previously making

- Stop your API from running (Ctrl + c).
- Go back into ***appsettings.Development.json*** file, and edit the Password parameter in the connection string so that authentication to the PostgreSQL Server will fail – see Figure 8-7.
- Save your file.

```
Commands.cs  appsettings.Development.json X
src > CommandAPI > {} appsettings.Development.json > ...
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft": "Warning",
6        "Microsoft.Hosting.Lifetime": "Information"
7      }
8    },
9    "ConnectionStrings":
10   {
11     "PostgreSqlConnection": "User ID=cmddbuser;Password=somewrongpaassword;Host=
12   }
13 }
14
```

Figure 8-7. The wrong credentials

OK, now run the app again, and try to make the API Call.

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

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
Content root path: D:\APITutorial\NET Core 3.1\CommandAPISolution\src\CommandAPI
fail: Microsoft.EntityFrameworkCore.Database.Connection[20004]
An error occurred using the connection to database 'CmdAPI' on server 'tcp://localhost:5432'.
fail: Microsoft.EntityFrameworkCore.Query[10100]
An exception occurred while iterating over the results of a query for context type 'CommandAPI
Npgsql.PostgresException (0x80004005): 28P01: password authentication failed for user "cmddbuser
at Npgsql.NpgsqlConnector.<>.DisplayClass160`0.<<DoReadMessage>>.ReadMessage_Long|0>d.Mp
ack
```

Figure 8-8. As expected, we can't connect

Fix It Up

OK, so let's fix this:

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

This means that *only* our ***appsettings.Development.json*** file now contains our connection string; your ***appsettings.json*** file should now look like that in Figure 8-9.

```

src > CommandAPI > {} appsettings.json > AllowedHosts
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft": "Warning",
6        "Microsoft.Hosting.Lifetime": "Information"
7      }
8    },
9    "AllowedHosts": "*"
10 }
11

```

Figure 8-9. Cleaned up Appsettings.json

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

🎓 Learning Opportunity What will happen if you edit the ***launchSettings.json*** file and change the value of ASPNETCORE_ENVIRONMENT to “Production”? Do this, run your app, and explain why you get this result.

We will cover our Production connection string in the Chapter 13.

User Secrets

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

We solve that here.

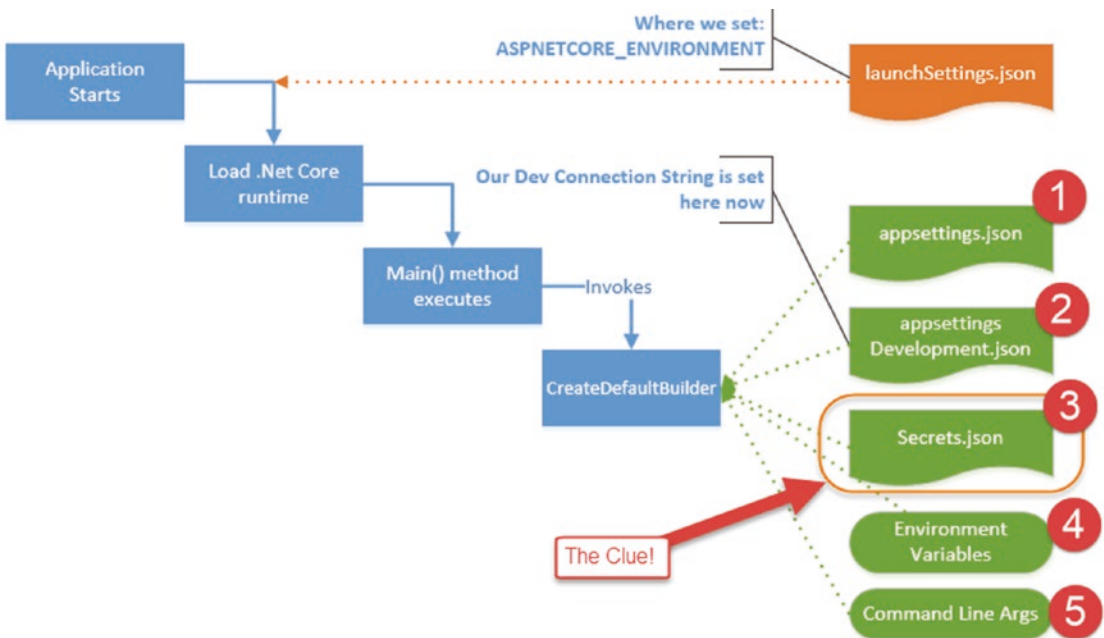


Figure 8-10. *Secrets.json in the scheme of things*

What Are User Secrets?

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

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

- User Secrets are “tied” to the individual developer – that is, you!


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

This means that individual users can store (among other things) the credentials that they use to connect to a database. As the file is secured by the local file system, they remain secure (assuming no one has log-in access to your PC).

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

Setting Up User Secrets

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

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

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

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

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <UserSecretsId></UserSecretsId>
  </PropertyGroup>
  <ItemGroup>
```

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

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

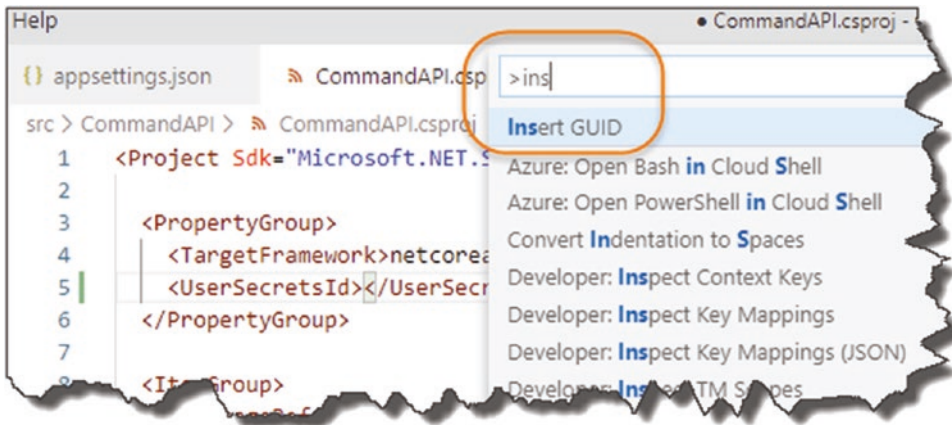


Figure 8-11. Insert GUID

- Insert GUID should appear; select it and select the first GUID Option.

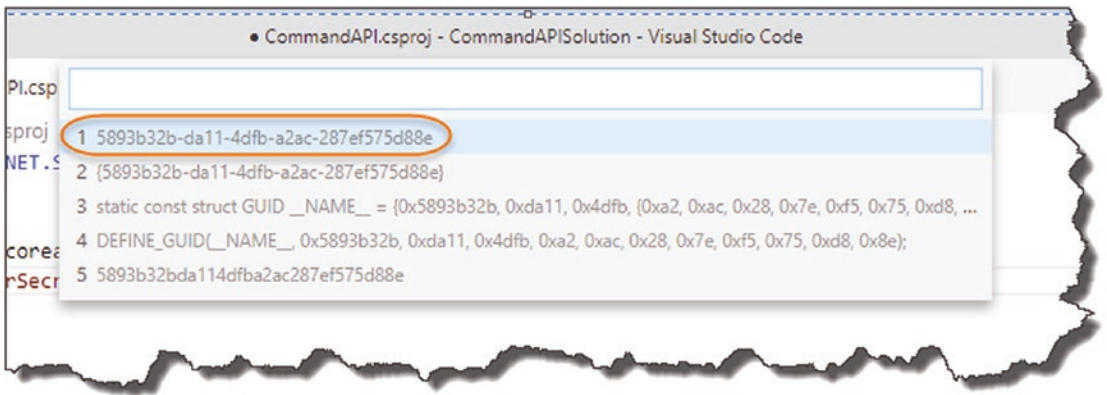


Figure 8-12. Select this GUID Format

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

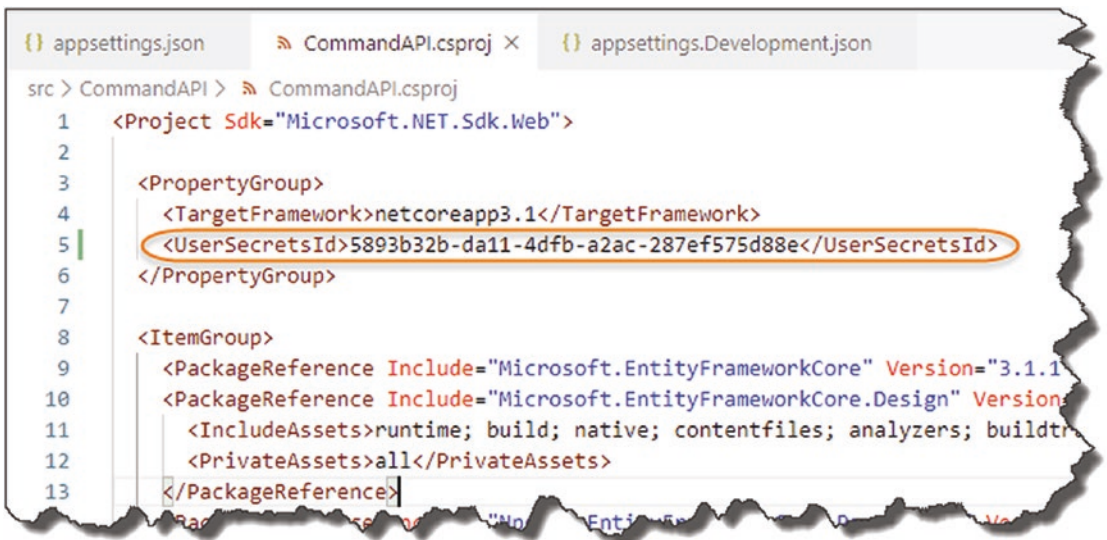


Figure 8-13. GUID Inserted into the .CSPROJ File

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.⁵

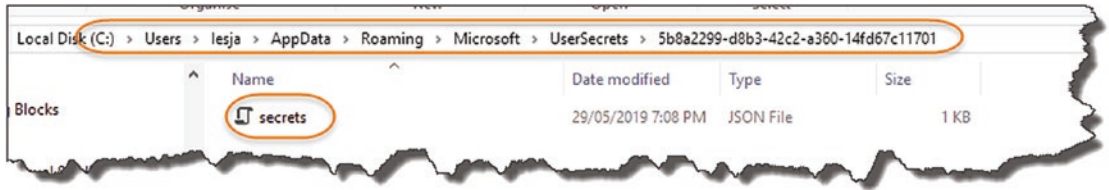


Figure 8-15. Location of Secrets.Json on Windows

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.

⁵On 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.

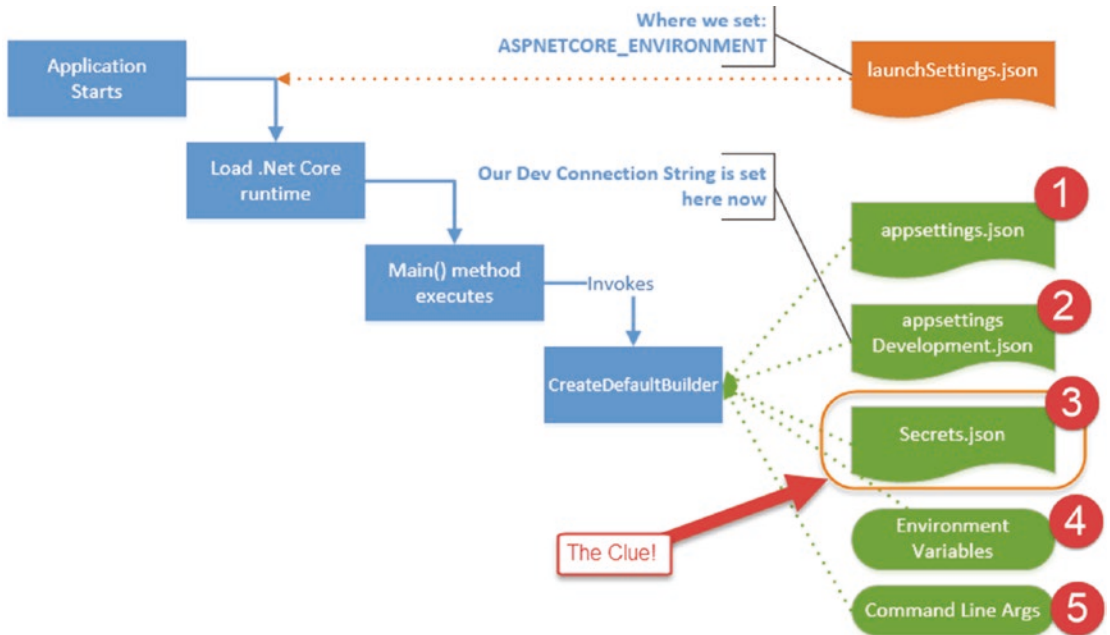


Figure 8-18. Revisit of precedence

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)!

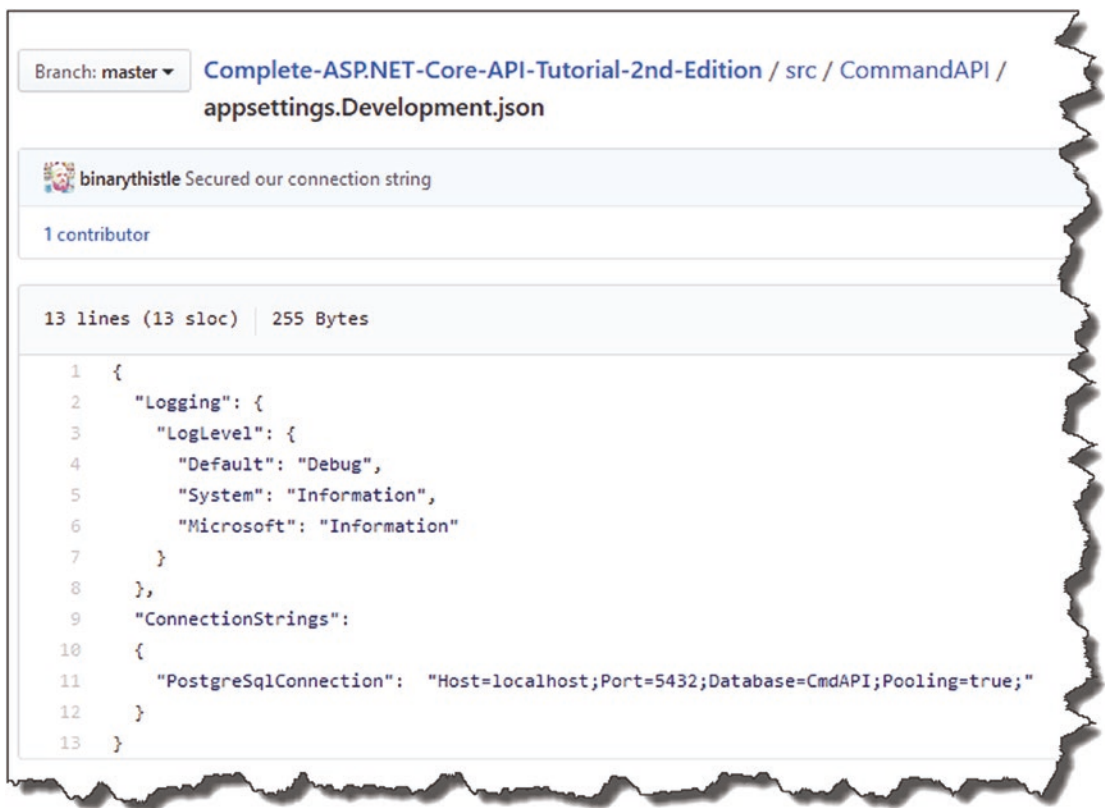


Figure 8-19. Clean Appsettings.json on GitHub

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.

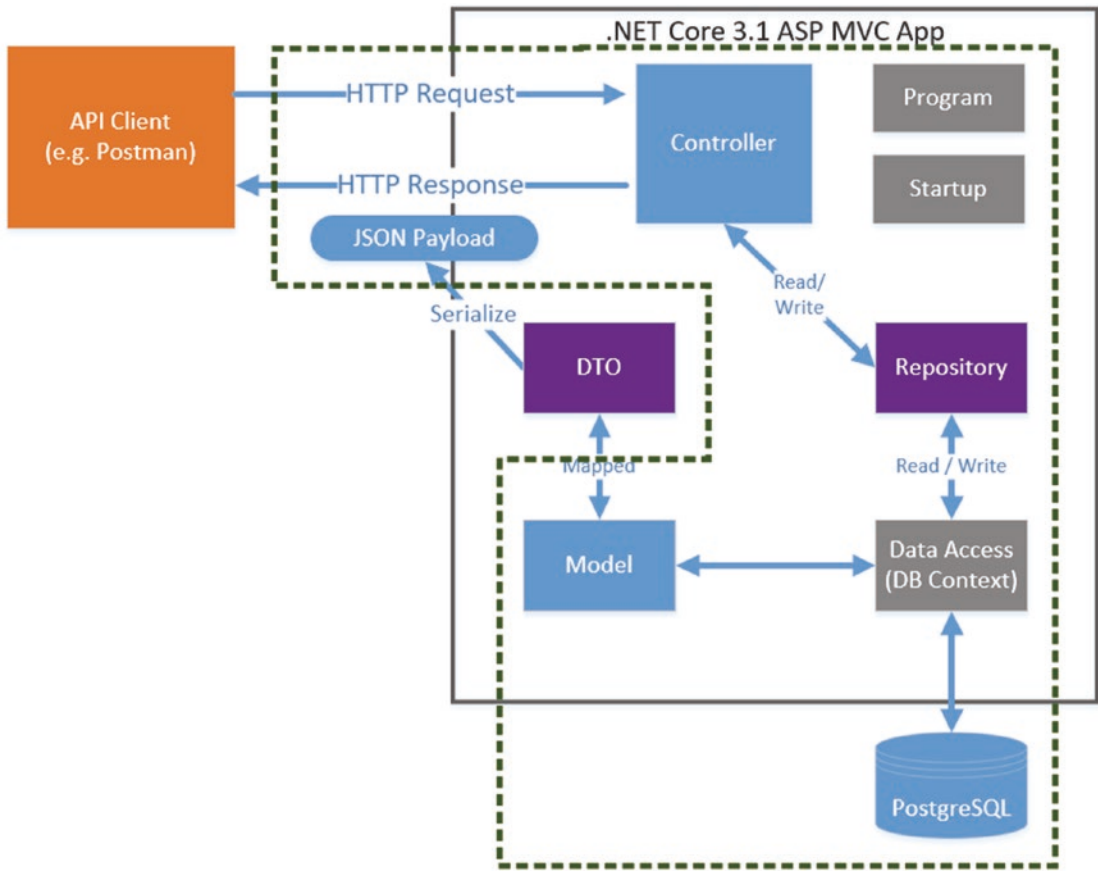


Figure 9-1. Architecture Progress

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:

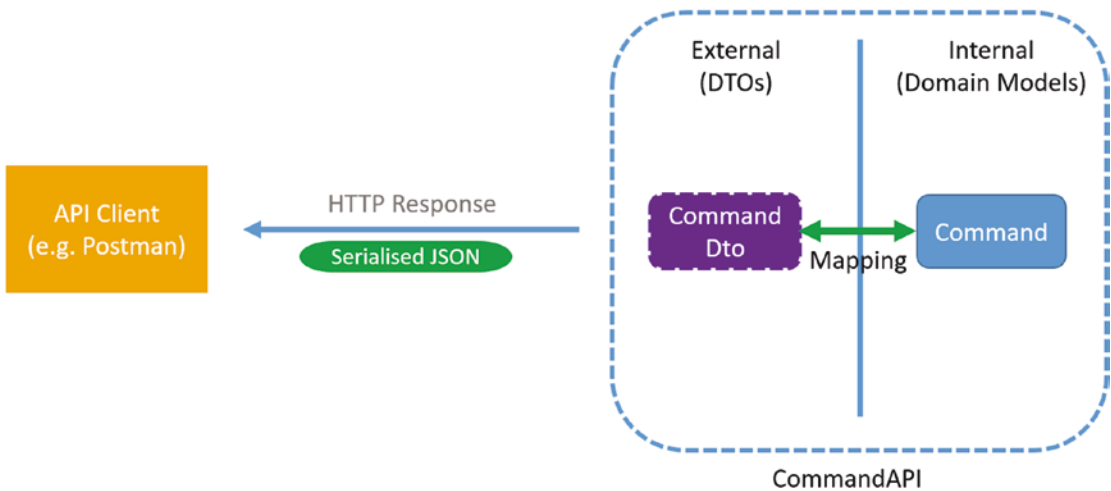


Figure 9-2. Example of Read DTO

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.

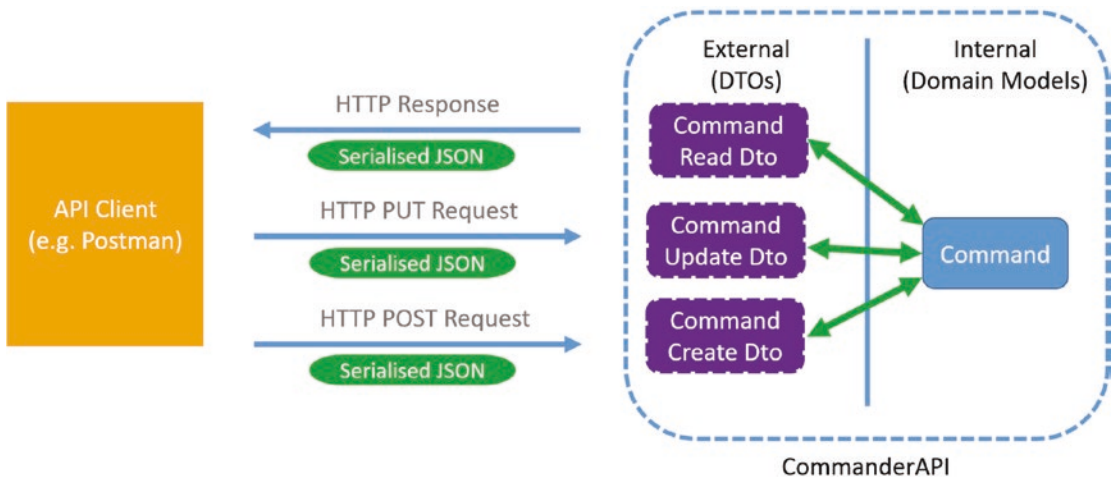


Figure 9-3. We can have DTOs for different actions

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.

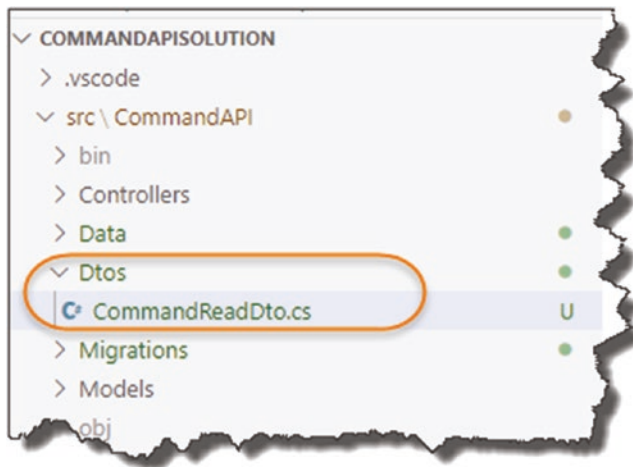


Figure 9-4. New Dtos Folder and CommandReadDto.cs file

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.

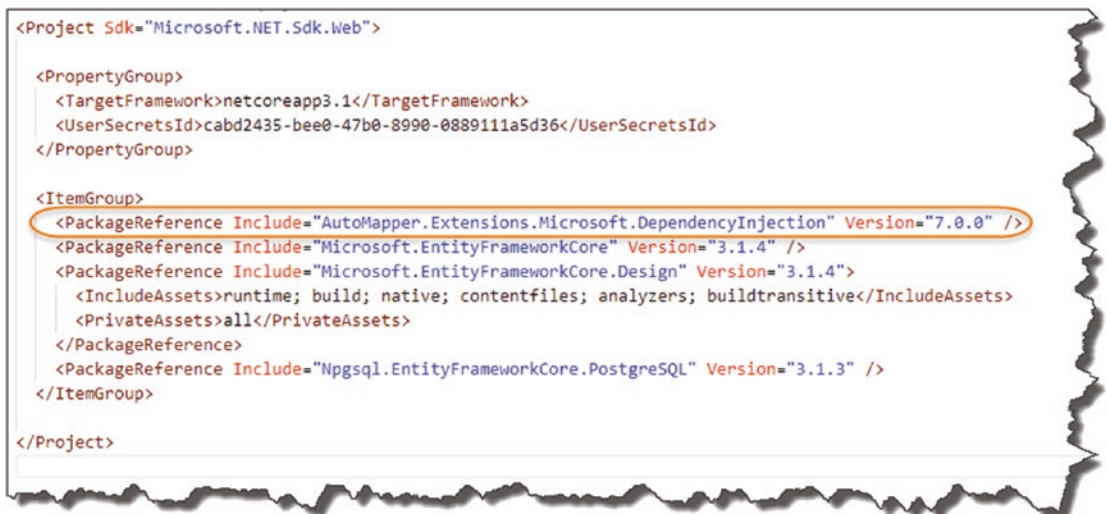


Figure 9-5. Reference to Automapper

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.¹

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

¹<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.

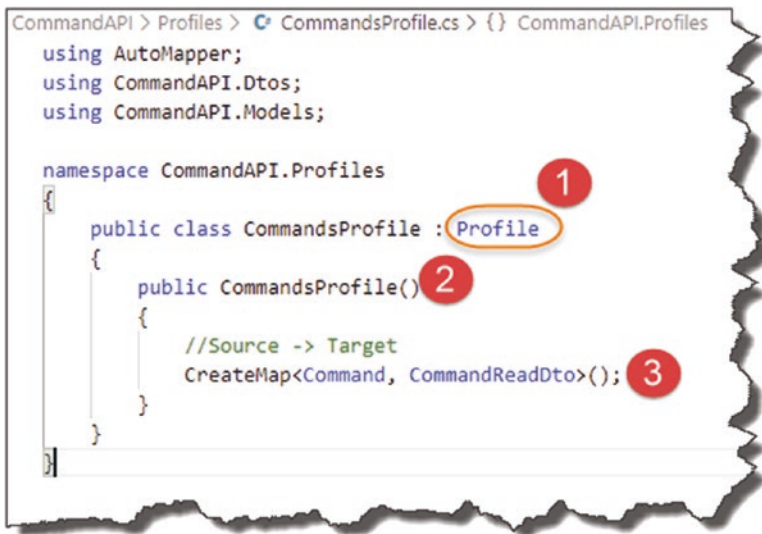


Figure 9-8. Our first AutoMapper Mapping

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.

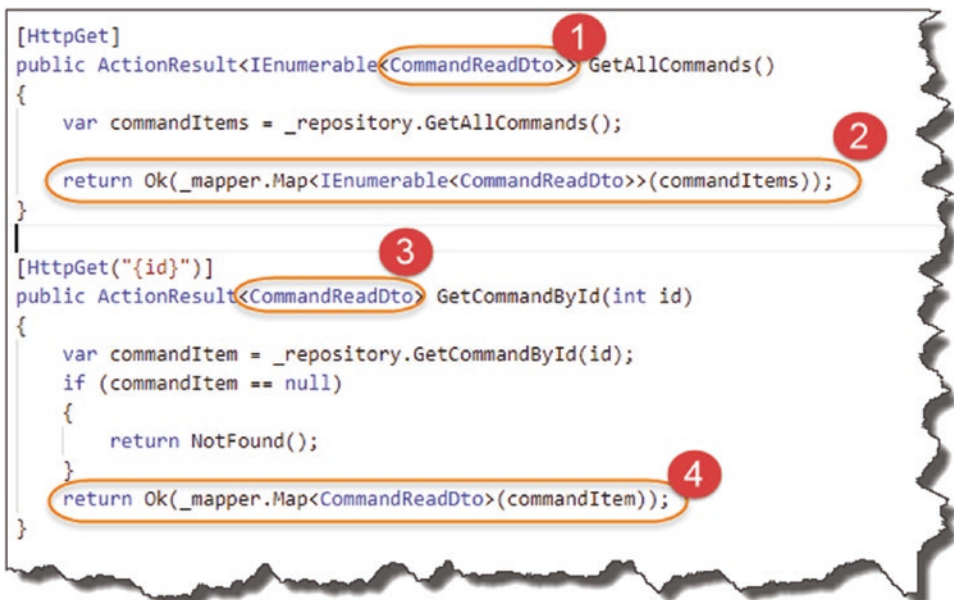


Figure 9-10. Use of AutoMapper in our 2 GET Controller Actions

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.

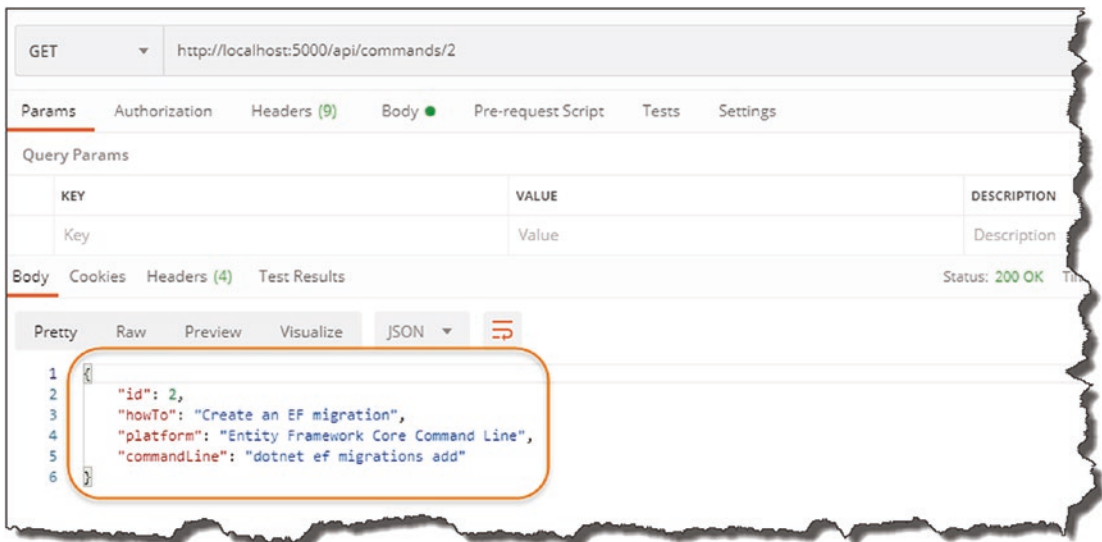


Figure 9-11. CommandReadDTO Returned

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.

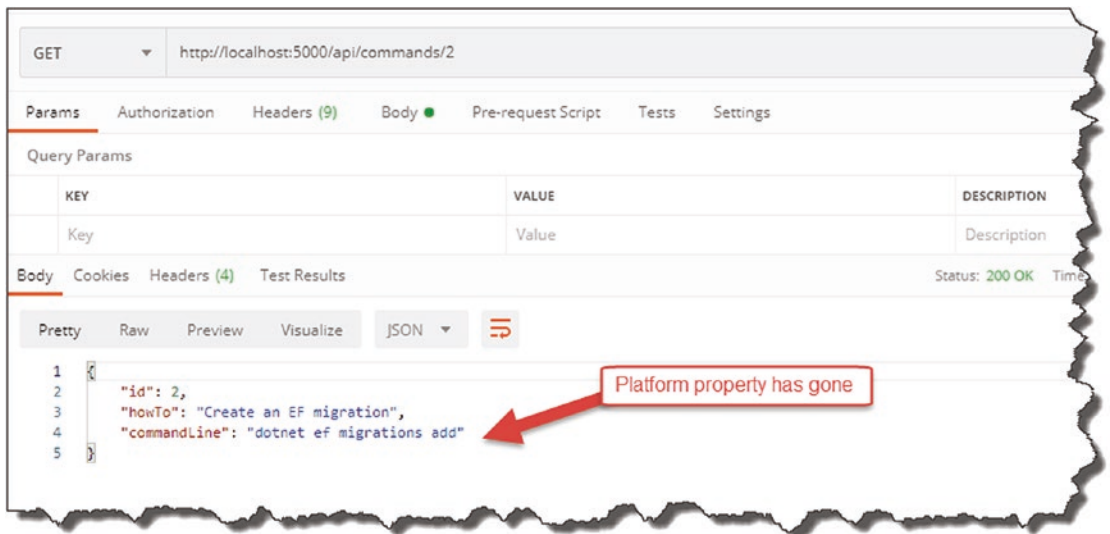


Figure 9-12. *CommandReadDto* returned with platform removed

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):

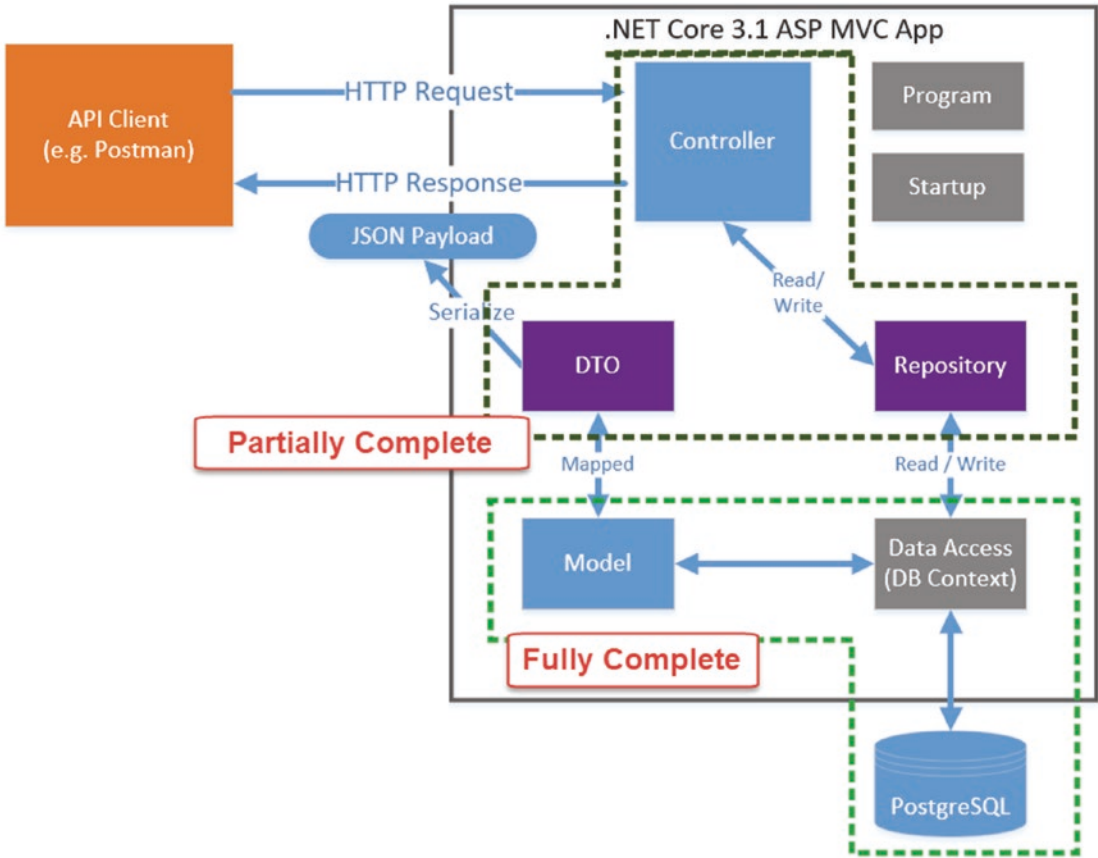


Figure 9-13. Architecture Check

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"> • HTTP 201 Created Status
Outputs	<ul style="list-style-type: none"> • Newly Created Resource (response body) • URI to newly created resource (response header)
Failure Outputs	<ul style="list-style-type: none"> • HTTP 400 Bad Request • HTTP 405 Not Allowed
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.¹

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*

¹https://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.

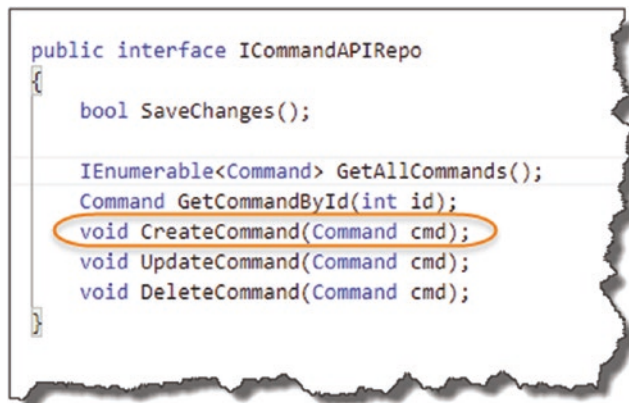
Compare that with one of our existing GET requests; we can perform the same request time and time again and get the *same result* – these *are* idempotent.

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.

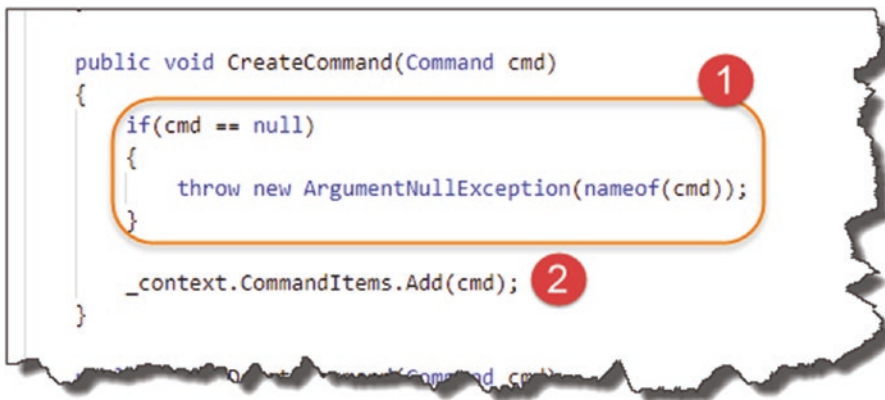


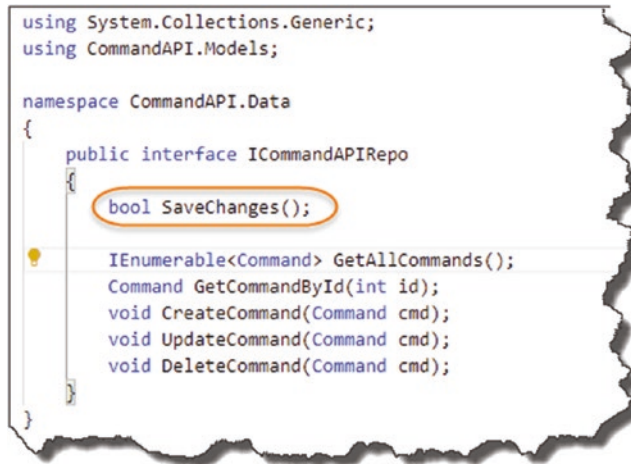
Figure 10-2. Implementation of CreateCommand

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;

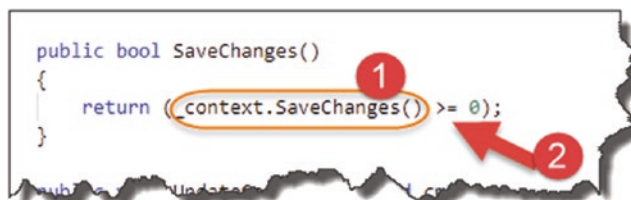
namespace CommandAPI.Data
{
    public interface ICommandAPIRepo
    {
        bool SaveChanges();
        IEnumerable<Command> GetAllCommands();
        Command GetCommandById(int id);
        void CreateCommand(Command cmd);
        void UpdateCommand(Command cmd);
        void DeleteCommand(Command 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 are²).

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.

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

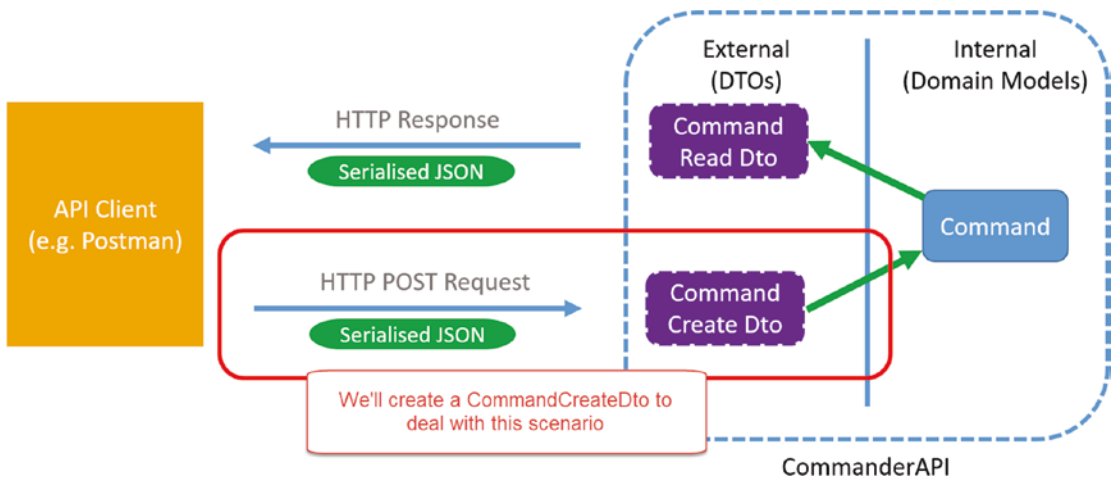


Figure 10-5. *CommandCreate DTO Example*

Create the New DTO

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

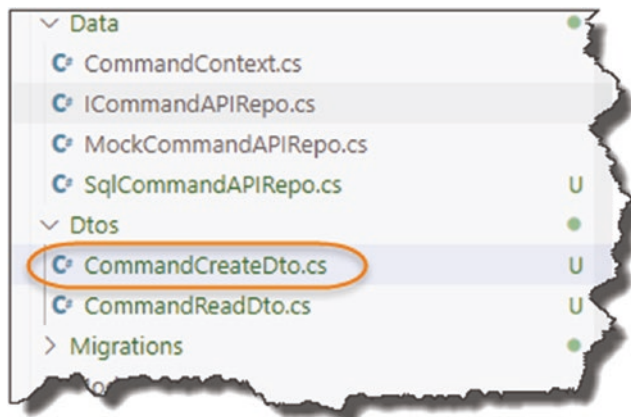


Figure 10-6. *CommandCreateDTO.cs created*

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.

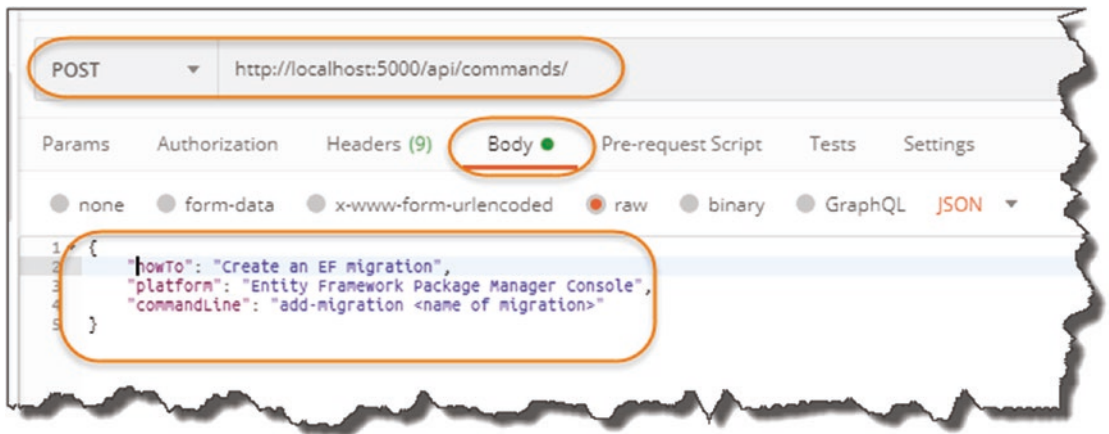


Figure 10-8. POST Request in Postman

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](#).³

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 Docs⁴) where we:

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

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

⁴<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.

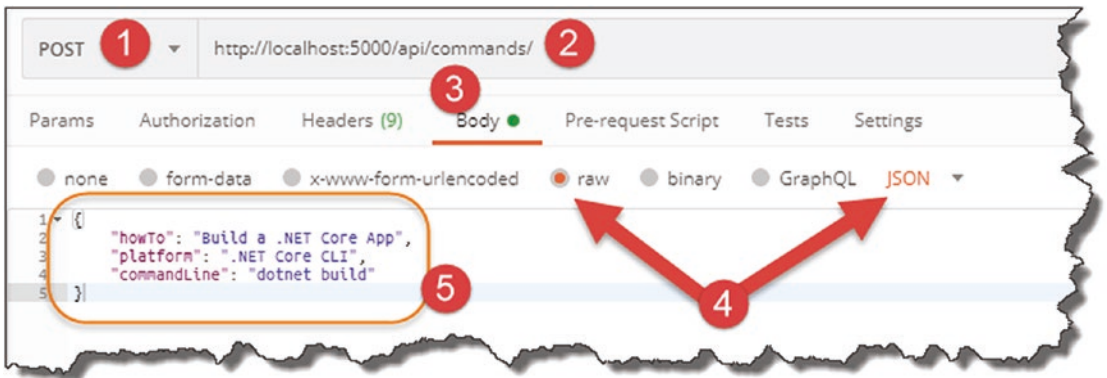


Figure 10-11. Test our *CreatCommand Endpoint*

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.

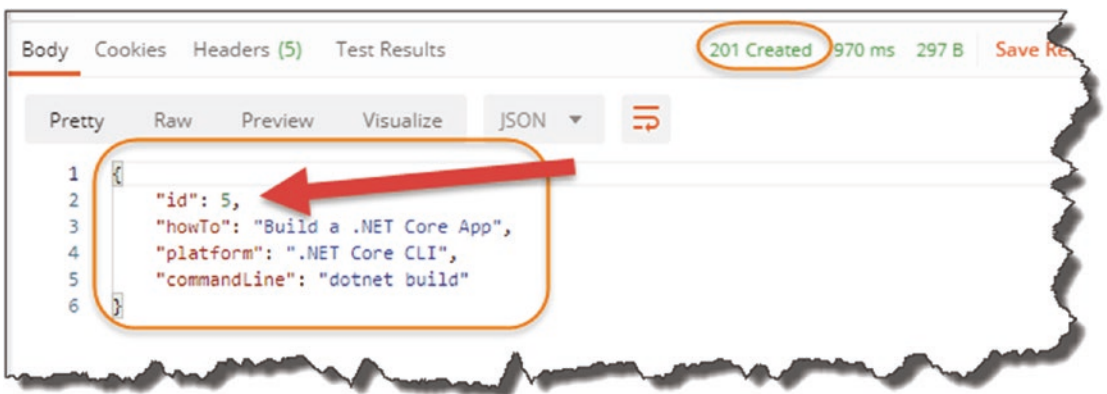


Figure 10-12. Successful 201 Result

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

Selecting the Headers Tab, you should get the following.

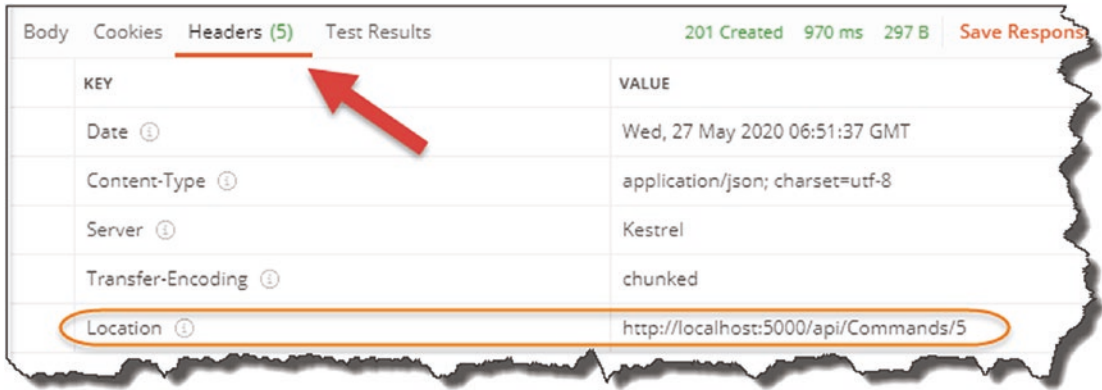


Figure 10-13. URI of our newly created resource is returned in the header

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"> • HTTP 204 No Content response code
Failure Outputs	<ul style="list-style-type: none"> • HTTP 400 Bad Request • HTTP 404 Not Found • HTTP 405 Not Allowed
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 CommandUpdateDto
    {
        [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 `CommandUpdateDto` 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`.