ASP .NET CORE 6

SUCCINCTLY

BY **DIRK STRAUSS**

Syncfusion®

ASP.NET Core 6 Succinctly

Dirk Strauss

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Table of Contents

The *Succinctly* Series of Books

Daniel Jebaraj
CEO of Syncfusion, Inc.

When we published our first *Succinctly* series book in 2012, *jQuery Succinctly*, our goal was to produce a series of concise technical books targeted at software developers working primarily on the Microsoft platform. We firmly believed then, as we do now, that most topics of interest can be translated into books that are about 100 pages in length.

We have since published over 200 books that have been downloaded millions of times. Reaching more than 2.7 million readers around the world, we have more than 70 authors who now cover a wider range of topics, such as Blazor, machine learning, and big data.

Each author is carefully chosen from a pool of talented experts who share our vision. The book before you and the others in this series are the result of our authors' tireless work. Within these pages, you will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

We are absolutely thrilled with the enthusiastic reception of our books. We believe the *Succinctly* series is the largest library of free technical books being actively published today. Truly exciting!

Our goal is to keep the information free and easily available so that anyone with a computing device and internet access can obtain concise information and benefit from it. The books will always be free. Any updates we publish will also be free.

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctlyseries@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on social media and help us spread the word about the *Succinctly* series!

About the Author

Dirk Strauss is a software developer from South Africa. He has extensive experience in SYSPRO customization, with a focus on C# and web development. He is passionate about writing code and sharing what he learns with others.

Chapter 1 Getting Started with ASP.NET Core 6.0

The .NET 6.0 framework is a major release for Microsoft and allows developers to build almost anything using one unified platform. The .NET Framework was released in 2002 for Windows development, and later evolved into what we know as .NET Core. This lightweight, cross-platform framework allows developers to build different types of applications.

Note: All the code for this book is available in this [GitHub repository](#).

With the release of .NET 5.0 in 2020, Microsoft began to solidify its unified vision for the entire platform, culminating in .NET 6.0. In this book, we will have a look at .NET 6.0 as it relates to ASP.NET Core.

Introducing .NET 6.0

Released in November 2021, .NET 6 delivered significant performance improvements as well as language improvements for C# 10 and F# 6. It also included native support for Apple Silicon (ARM64). New APIs were also added for HTTP/3, mathematics, and JSON processing using the new System.Text.Json source generator.

Here are some of the features highlighted by Microsoft program manager Richard Lander regarding .NET 6 in the [announcement blog](#):

- It will be supported for three years.
- Serves as a unified platform across browser, cloud, desktop, IoT, and mobile applications.
- Increases performance throughout the framework.
- Includes language improvements for C# 10 and F# 6.
- Supports Hot Reload in Visual Studio 2022.
- Delivers better performance for JSON APIs.
- Introduces minimal APIs.

Installing .NET 6.0

The easiest way to install .NET 6.0 is just installing Visual Studio 2022, because .NET 6.0 comes bundled with Visual Studio 2022.

Note: Visual Studio 2022 is a required update to use .NET 6.0, and all the code samples in this book will be using Visual Studio 2022.

If you are using Visual Studio Code, you will need to install .NET 6.0 directly using a separate download. If you want to use Visual Studio, you can download the Community edition (Figure 1), which is free, and will be good enough for what we want to achieve in this book.
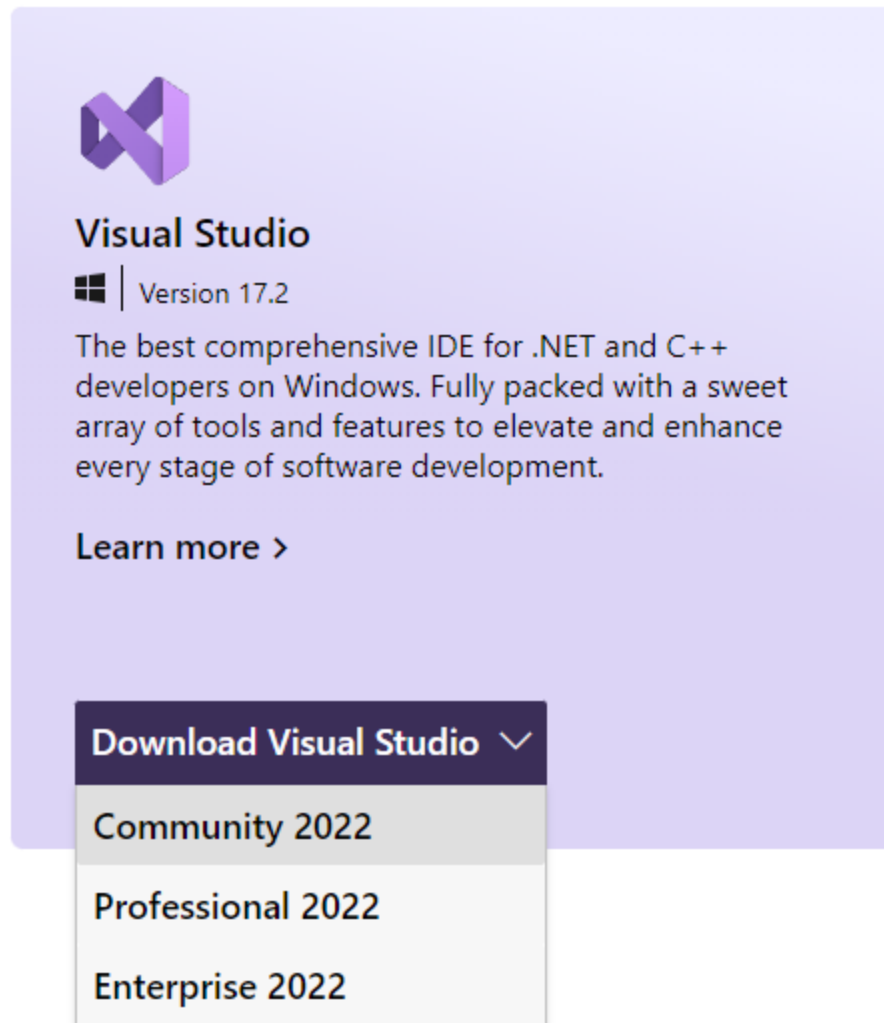
Figure 1: Download Visual Studio Community Edition

You can download a copy of Visual Studio Community 2022 [here](#).

When performing the installation, make sure that you select the **ASP.NET and web development** workload option, as shown in Figure 2. If you have already installed Visual Studio 2022, make sure that you have this workload added. If you want to install just .NET 6.0 without Visual Studio, you can download the installer [here](#).
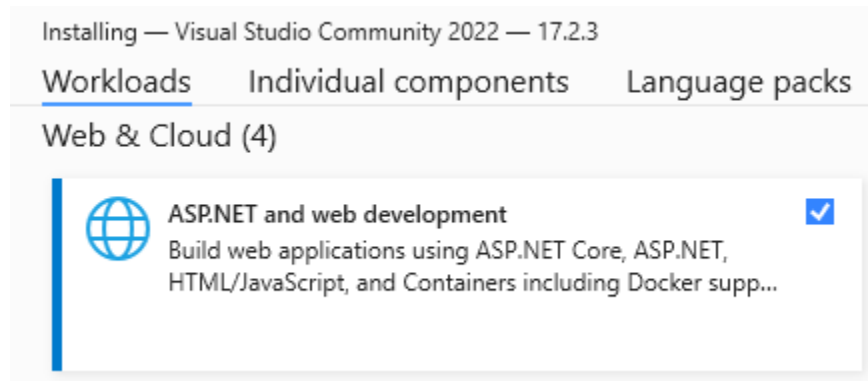
Figure 2: Select the ASP.NET and Web Development Workload

You can verify the installed versions of .NET on your machine by typing the following in the command prompt.

Code Listing 1: Verify Installed .NET Version

```
dotnet --info
```

You only need .NET 6.0 for the examples in this book, so as long as you have that installed, you're good to go.

Exploring ASP.NET Core 6.0

With all the mentioning of .NET 6.0, it is easy to confuse the fact that when it comes to ASP.NET Core 6.0, Microsoft still uses "Core" in the name. This means that while the overall platform does not use "Core" in the name, the specific application model names do when we talk about ASP.NET Core 6.0 and Entity Framework Core 6.0.

With ASP.NET Core 6.0, we can create:

- MVC/web API apps built on the MVC pattern.
- Razor pages that use a page-oriented abstraction for building web apps.
- Blazor apps that provide a way to build rich UIs, instead of using JavaScript and Razor.
- Signal R apps that use WebSockets to provide real-time communication.
- gRPC services that provide a language-agnostic, high-performance remote procedure call framework.
- Minimal APIs, a lightweight pattern for creating APIs and an alternative to creating web APIs.

We will be looking into minimal APIs in more detail in a later chapter.

Upgrading existing applications to ASP.NET Core 6.0

If your application is built on a recent version of .NET, upgrading your application to .NET 6.0 is relatively easy. Upgrading applications built on ASP.NET Core 2.0 and later requires minimal changes to references and configurations. Upgrading from any ASP.NET Core 1.x application requires some structural and configuration changes, taking more effort to accomplish. If you have an application built on the ASP.NET framework, you will have to make considerable changes to the configuration and architecture.

Some developers choose to upgrade incrementally. In such cases, if you have an ASP.NET Core 2.1 application, you'd upgrade it to ASP.NET Core 3.1 before upgrading to ASP.NET Core 6.0.

Check out Microsoft's documentation on upgrading to ASP.NET Core by visiting [this link](#). It provides useful information about upgrading earlier versions of .NET Core applications. As luck would have it, I have a book repository API that I wrote in ASP.NET Core 5.0 that I need to upgrade to ASP.NET Core 6.0.

> 💡*Tip: I would suggest going through this exercise of upgrading the book repository project to ASP.NET Core 6.0. We will be using this project again later in the book to add the data service when we look at minimal APIs. Remember, all the code is available in the [GitHub repository](#).*

Let's use this book repository and upgrade it to ASP.NET Core 6.0.

Upgrading BookRepository.Core

You will notice from Figure 3 that the solution contains three projects. Each one of these projects will need to be upgraded to ASP.NET Core 6.0.
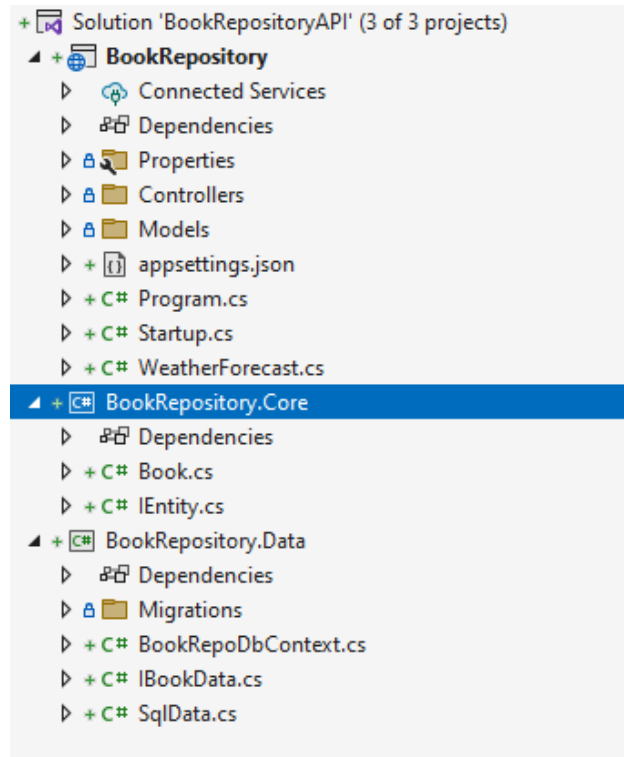
Figure 3: The BookRepositoryAPI Project

Let's start by editing the csproj files for the BookRepository.Core project. Right-click the project and choose the **Edit Project File** option.
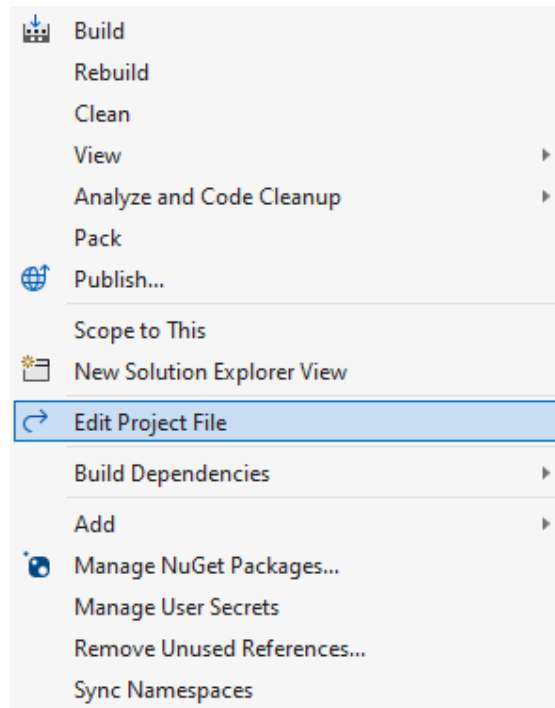
Figure 4: Edit the csproj File

In Code Listing 2, you will see the familiar **TargetFramework** tag specifying .NET 5.0. We also have a package reference to include **Microsoft.AspNetCore.Mvc.Versioning**, but at the time of writing this book, version 5.0.0 is the latest one available.

Code Listing 2: The BookRepository.Core csproj File

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

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Versioning"
Version="5.0.0" />
  </ItemGroup>

</Project>
```

Modify the **TargetFramework** from **net5.0** to **net6.0** as illustrated in Code Listing 3.

Code Listing 3: Change the Target Framework to .NET 6.0

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Versioning"
Version="5.0.0" />
  </ItemGroup>

</Project>
```

To verify that the upgrade to .NET 6.0 was successful for the BookRepository.Core project, right-click the project and perform a rebuild. If it succeeds, then the upgrade for that specific project in the solution worked as expected.

Upgrading BookRepository

The next project that I need to update is the BookRepository project. Right-click the project and edit the project file by clicking **Edit Project File** as shown in Figure 4. The project file is listed in Code Listing 4.

Code Listing 4: The BookRepository csproj File

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

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Remove="Controllers\WeatherForecastController.cs" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Versioning"
Version="5.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore"
Version="5.0.11" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
Version="5.0.11">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
Version="5.0.11" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="5.6.3" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference
Include="..\BookRepository.Data\BookRepository.Data.csproj" />
  </ItemGroup>

</Project>
```

In this project, we have a few more package references. As before, change the
TargetFramework from **net5.0** to **net6.0** and save your changes. Now go ahead and open the
NuGet package manager and view the **Updates** tab.
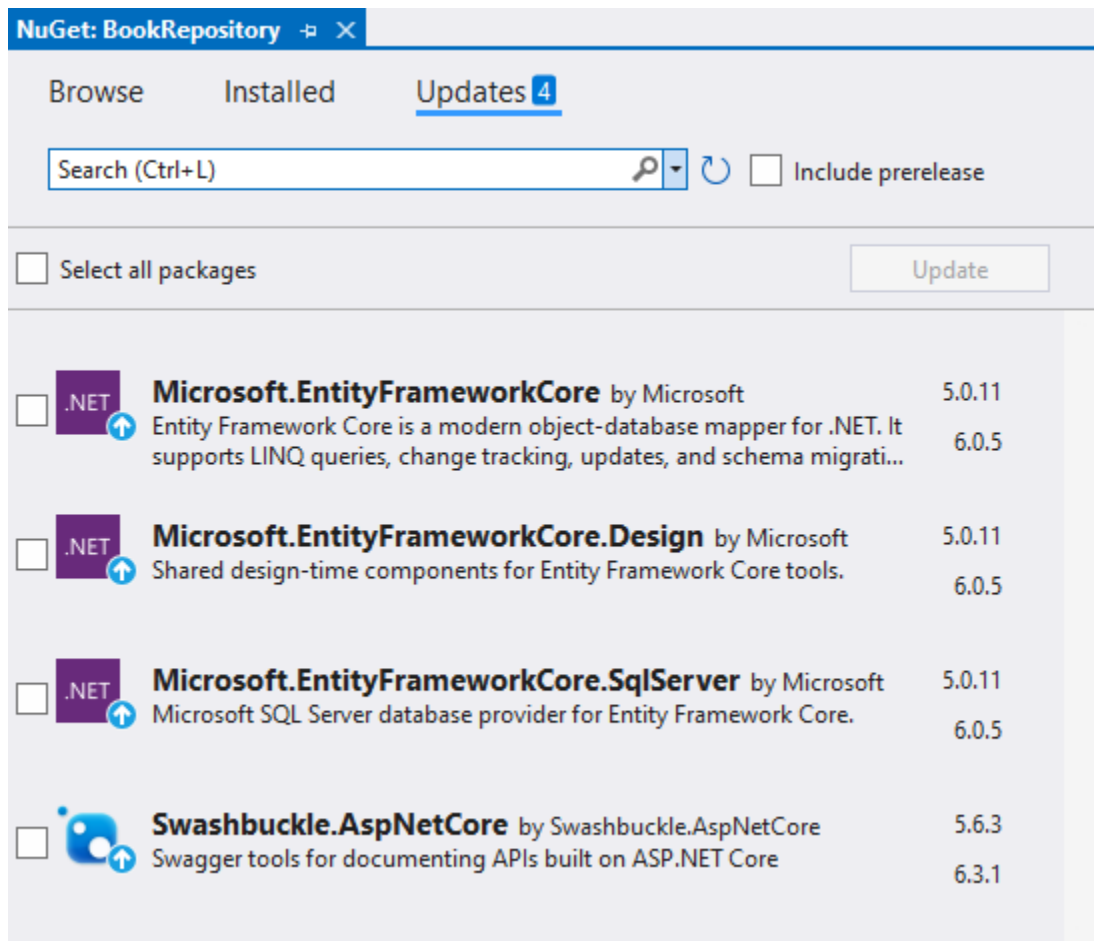
Figure 5: Update NuGet Packages for the BookRepository Project

Upgrade each of these packages to the latest versions and keep an eye on the Package Manager Output window for any errors. After updating the NuGet packages, your csproj file should look like the code in Code Listing 5.

Code Listing 5: The Upgraded BookRepository csproj File

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Remove="Controllers\WeatherForecastController.cs" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Versioning"
Version="5.0.0" />
```

```
    <PackageReference Include="Microsoft.EntityFrameworkCore"
Version="6.0.5" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
Version="6.0.5">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
Version="6.0.5" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.3.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference
Include="..\BookRepository.Data\BookRepository.Data.csproj" />
  </ItemGroup>

</Project>
```

Give the BookRepository project a rebuild and if it works, the upgrade is successful.

📝 ***Note: The versions of the package references in the csproj file are correct as of the time of writing this book. These might, however, differ for you if you are upgrading this sample project at a later time.***

The last project that I need to upgrade is the BookRepository.Data project. Let's do that next.

Upgrading BookRepository.Data

The last project that I need to update is the BookRepository.Data project. Right-click the project and edit the project file by clicking **Edit Project File** as shown in Figure 4. The project file is listed in Code Listing 6.

Code Listing 6: The BookRepository.Data csproj File

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

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Versioning"
Version="5.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore"
Version="5.0.11" />
```

```xml
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
Version="5.0.11">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Relational"
Version="5.0.11" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
Version="5.0.11" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference
Include="..\BookRepository.Core\BookRepository.Core.csproj" />
  </ItemGroup>

</Project>
```

Just like the BookRepository project, we have a few more package references in this project. Change the **TargetFramework** from **net5.0** to **net6.0** and save your changes.

Next, we need to upgrade the NuGet packages.
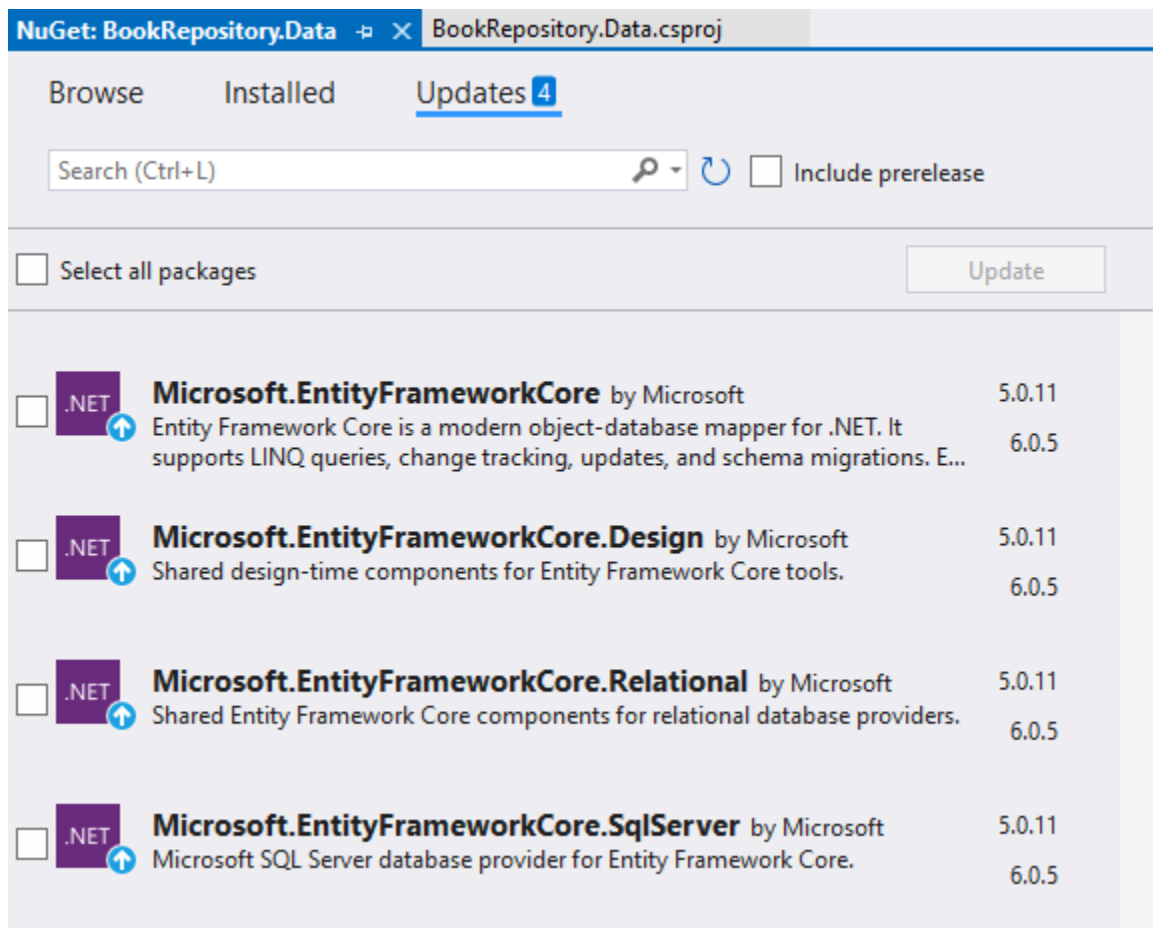
Figure 6: The NuGet Packages for the BookRepository.Data Project

As before, we can see on the Updates tab that several NuGet Packages need to be upgraded.

Note: The order in which you update the NuGet packages matters. If you tried updating the Microsoft.EntityFrameworkCore.Design package before the Microsoft.EntityFrameworkCore.Relational package, you could receive an error because Relational is a dependency of Design.

It is worth noting that each NuGet package lists any dependencies in its package description, as shown in Figure 7.
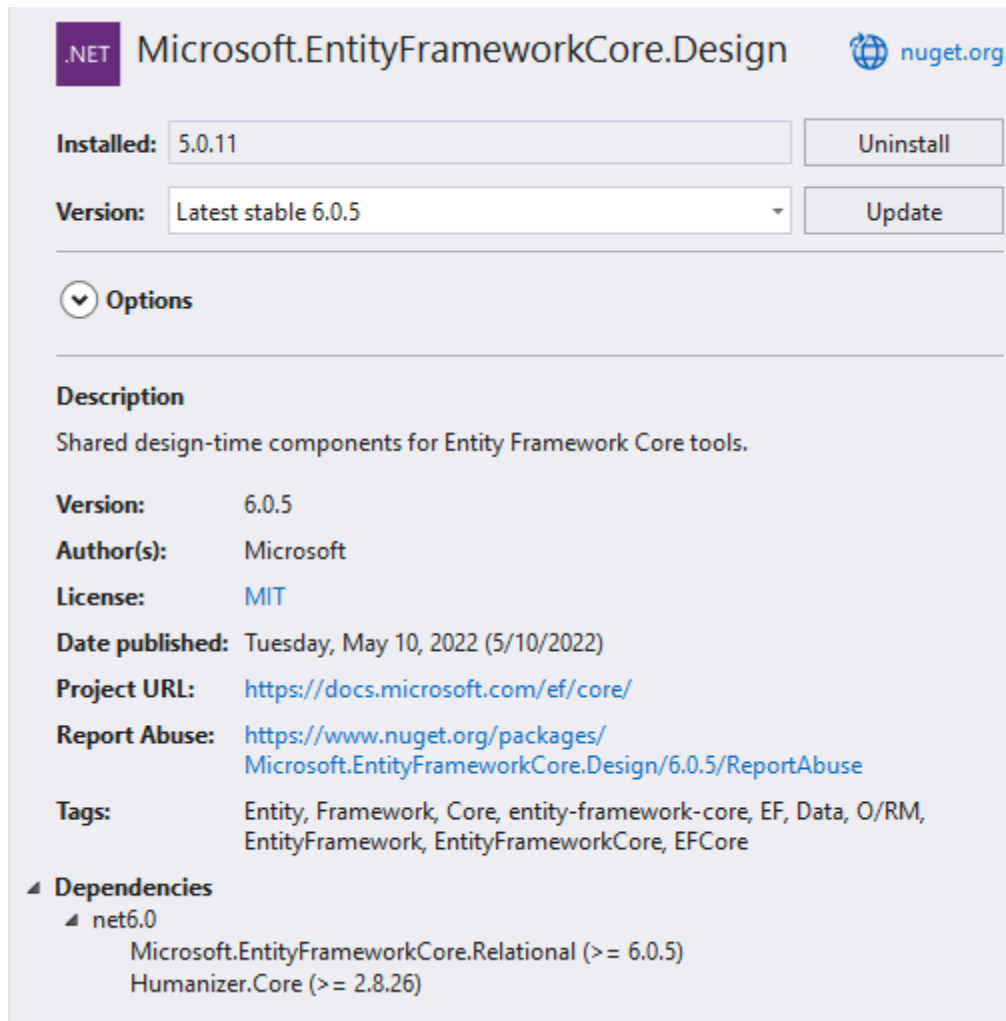
Figure 7: Listed Package Dependencies

It is therefore required that the dependencies be updated before the package is updated; otherwise, the update will fail.

Once this has been completed, your csproj file will look like the code in Code Listing 7. Rebuild the project and check that everything went as planned.

Code Listing 7: The Updated BookRepository.Data csproj File

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Versioning"
Version="5.0.0" />
```

```xml
    <PackageReference Include="Microsoft.EntityFrameworkCore"
Version="6.0.5" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
Version="6.0.5">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Relational"
Version="6.0.5" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
Version="6.0.5" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference
Include="..\BookRepository.Core\BookRepository.Core.csproj" />
  </ItemGroup>

</Project>
```

Finally, right-click the top-level solution and do a complete rebuild of the solution. If all builds successfully, you have completely upgraded the ASP.NET Core 5.0 project to ASP.NET Core 6.0.

Checking that the API still runs

The last thing I need to check is that the API still works. This portion of the chapter is for informational purposes, and only applicable if you want to run this API project on your own machine. If you do not want to run the API project, then feel free to skip to the next chapter.

The web API uses a LocalDB database called BookRepo. You can see this in Figure 8. I will not go into any detail on how to create this database and how to use migrations; I will leave this to you as homework should you be curious.
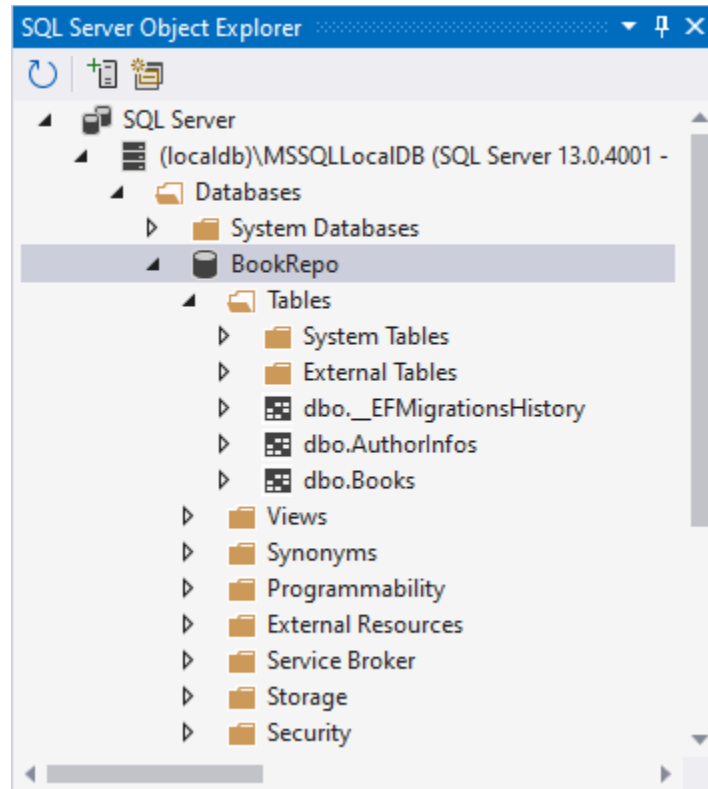
Figure 8: The SQL Server Object Explorer

The next thing I need to check is what the debug URL is. Right-click the **BookRepository** project and click **Properties**.
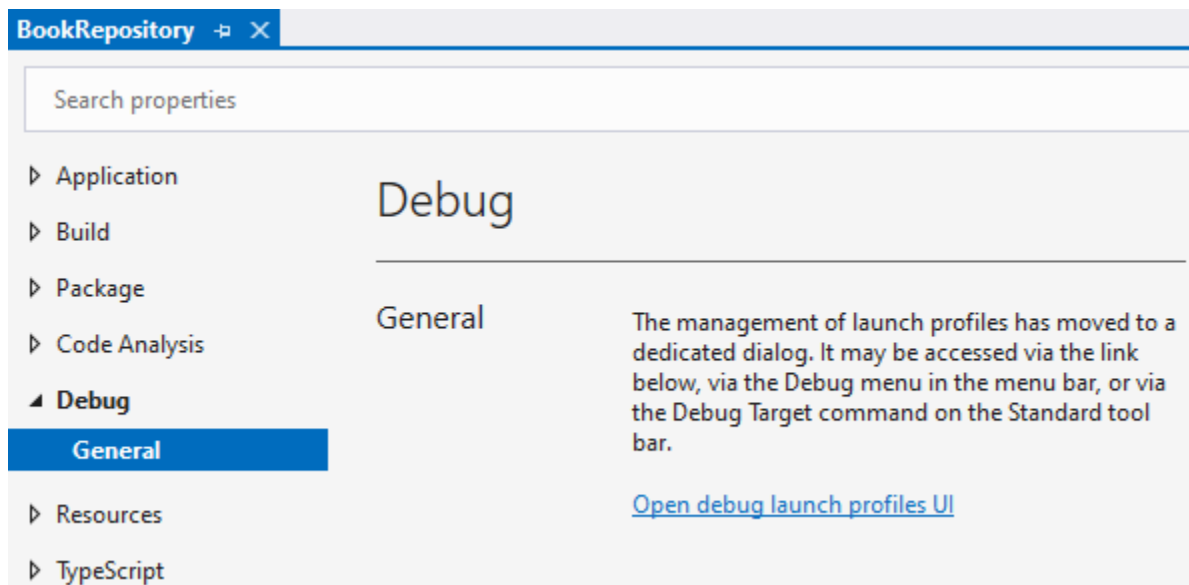

Figure 9: The BookRepository Debug Properties

You will notice in Figure 9 that the properties look slightly different. Under the **Debug** section, click the **Open debug launch profiles UI** link.
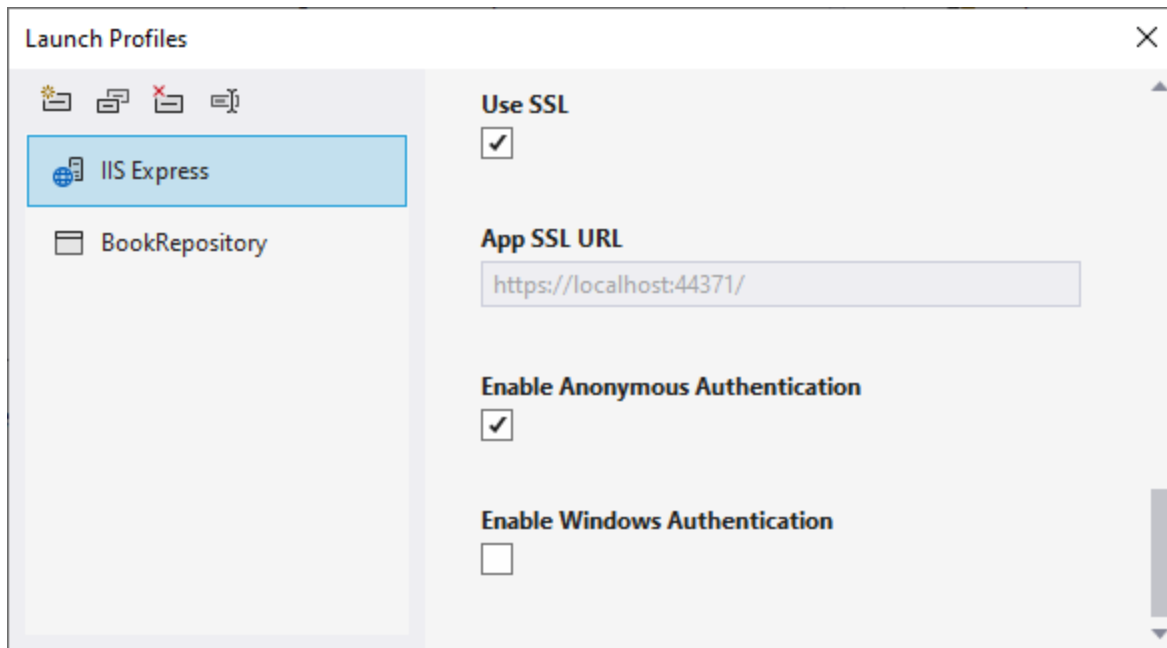

Figure 10: The Debug Launch Profiles UI

As seen in Figure 10, ensure that you:

- Select **Use SSL**.
- Make a note of the App SSL URL.
- Clear the **Launch Browser** option.

In the **BookController.cs** file in the BookRepository project, there are two **GetBooks** methods with the **[HttpGet]** verb attribute. Each **GetBooks** method has a different **MapToApiVersion** attribute. One is **[MapToApiVersion("1.0")]**, and the other is **[MapToApiVersion("1.1")]**. Depending on the version of the route I call, the method will list all the books in the database based on that particular version and logic.

I also notice that the **BookController.cs** class has a route attribute of **[Route("api/v{version:apiVersion}/[controller]")]**.

Based on this information, I know that once I run the web API in Visual Studio 2022, the browser will not be launched (because the Launch Browser checkbox was not selected), and I can list all the books in the database by calling one of the following endpoints in Postman, as shown in Code Listings 8 and 9.

Code Listing 8: Endpoint for API Version 1.0

```
https://localhost:44371/api/v1.0/book
```

To call version 1.1 of the API, just change **v1.0** to **v1.1** in the URL.

Code Listing 9: Endpoint for API Version 1.1

```
https://localhost:44371/api/v1.1/book
```

Calling one of these endpoints in Postman results in the JSON output shown in Figure 11, with the list of books returned from the **GetBooks** method.
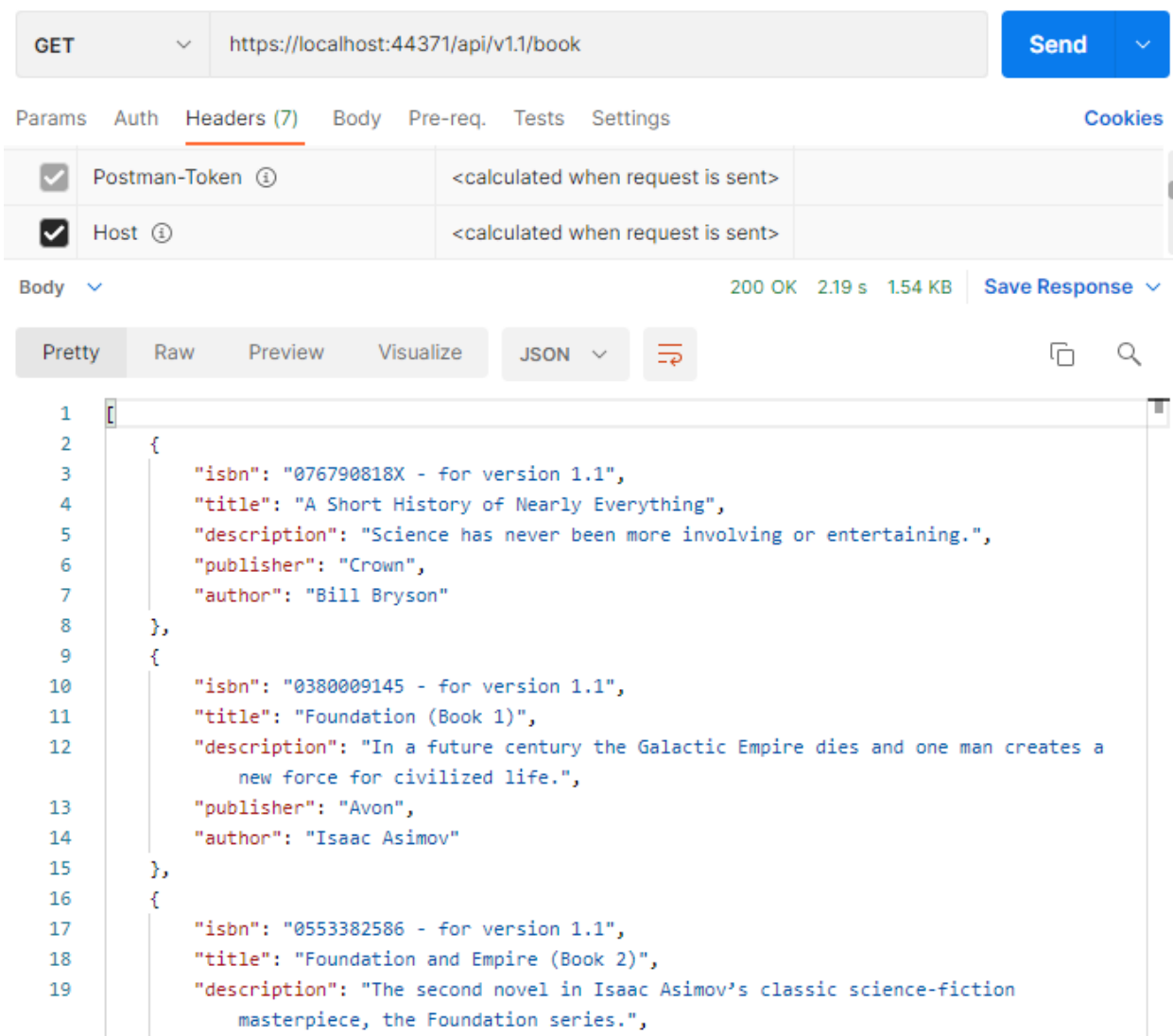
Figure 11: The Result from Postman

Now that I have tested the web API, I am satisfied that the upgrade process from ASP.NET Core 5.0 to ASP.NET Core 6.0 was successful.

Chapter 2 Working with ASP.NET Core 6.0

With the release of .NET 6.0, there have been many improvements to existing features in ASP.NET Core 6.0. There have also been a few new features added. This chapter will take a peek at some of these.

Debug running code with Hot Reload

Traditionally, before a developer made a change to an application, they had to stop debugging, modify the code, save it, and recompile it before debugging it again.
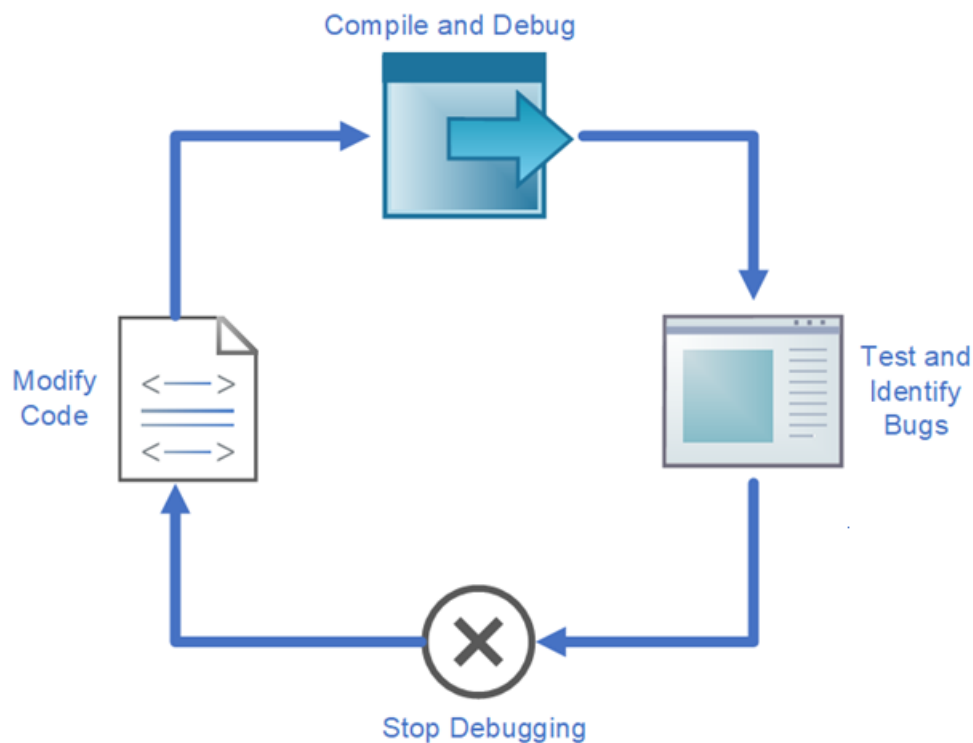
Figure 12: Traditional Debug Process

This is fine, I suppose, when you are dealing with a small application. But if you are debugging a large application consisting of multiple projects, Hot Reload becomes a welcome addition to your developer tool belt.

This is especially true when the debug startup takes several seconds. There are some limitations, and we will see them shortly.

To demonstrate Hot Reload, take a look at the BookRepository API we upgraded to .NET 6.0 in Chapter 1. In the **BookController.cs** file, there is a method called **GetBooks_1_1**. The code is illustrated in Code Listing 10.

Code Listing 10: The GetBooks_1_1 Controller Action

```
[HttpGet]
[MapToApiVersion("1.1")]
public async Task<ActionResult<List<BookModel>>> GetBooks_1_1()
{
    try
    {
        var books = await _service.ListBooksAsync();
        return (from book in books
                let model = new BookModel()
                {
                    Author = book.Author,
                    Description = book.Description,
                    Title = book.Title,
                    Publisher = book.Publisher,
                    ISBN = book.ISBN + " - for version 1.1"
                }
                select model).ToList();
    }
    catch (Exception)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, "There
was a database failure");
    }
}
```

Run the API and view the result in Postman by calling the URL
https://localhost:44371/api/v1.1/book. This will result in the output illustrated in Figure
13.
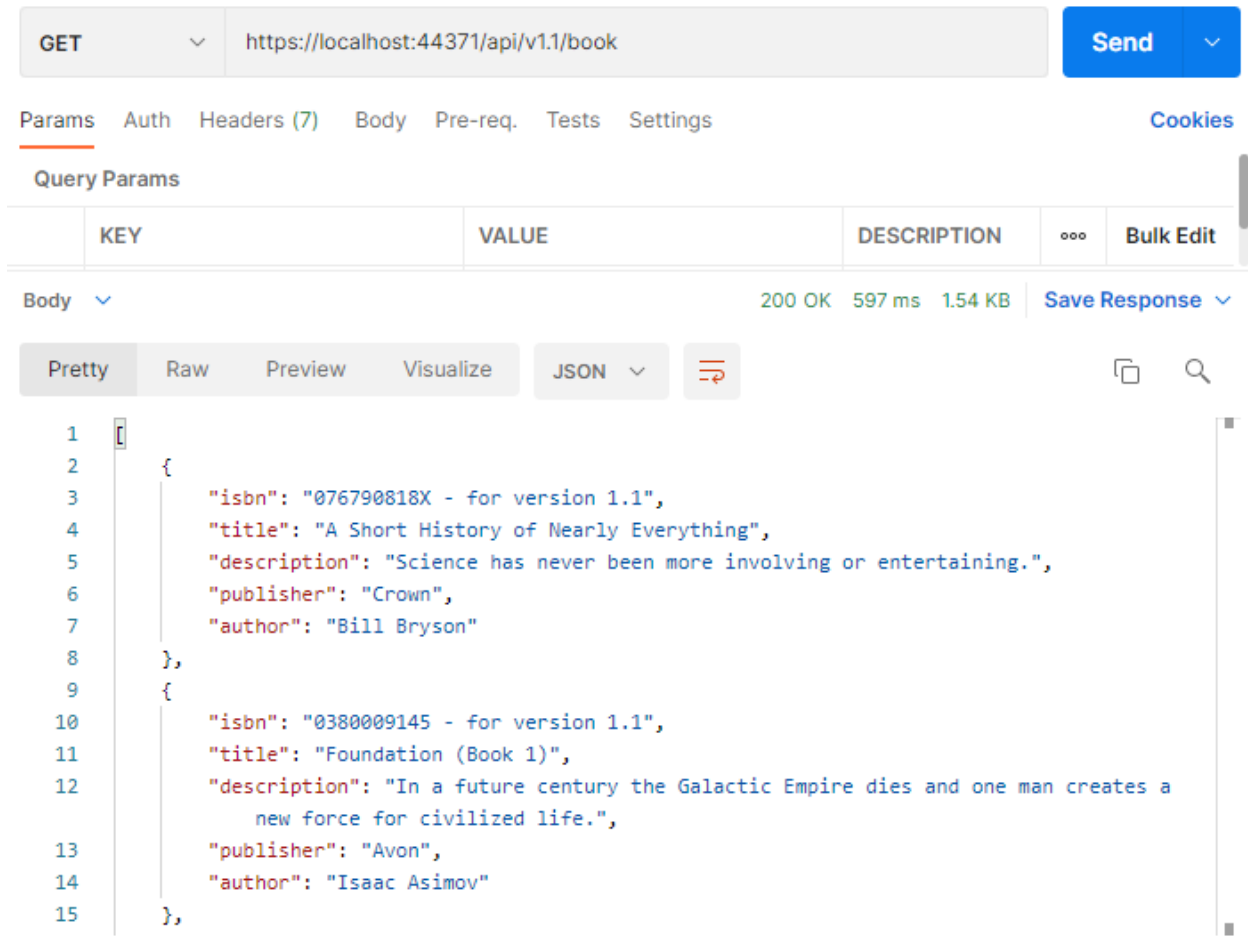
Figure 13: The Postman Result

Don't stop your debugging session just yet. Modify the **GetBooks_1_1** action and change the text **version 1.1** to **API version 1.1**, as shown in Code Listing 11.

Code Listing 11: The Modified GetBooks_1_1 Action

```
[HttpGet]
[MapToApiVersion("1.1")]
public async Task<ActionResult<List<BookModel>>> GetBooks_1_1()
{
    try
    {
        var books = await _service.ListBooksAsync();
        return (from book in books
                let model = new BookModel()
                {
                    Author = book.Author,
                    Description = book.Description,
                    Title = book.Title,
                    Publisher = book.Publisher,
```

```
                ISBN = book.ISBN + " - API version 1.1"
            }
            select model).ToList();
    }
    catch (Exception)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, "There
was a database failure");
    }
}
```

As illustrated in Figure 14, click **Save** (I know what you're thinking, you will see how to change this behavior shortly), and then click the little flame icon for Hot Reload.
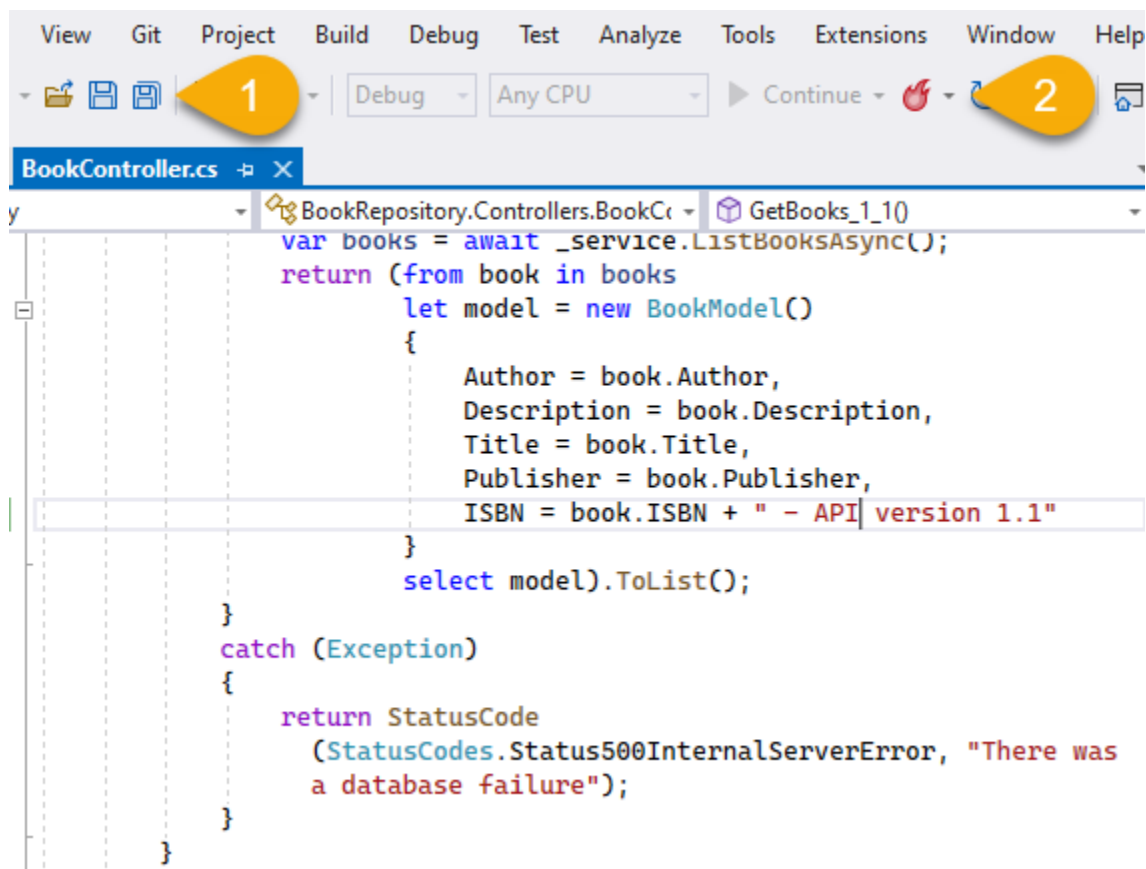

Figure 14: Save and Click on Hot Reload

Switch back to Postman, send a request to the endpoint **https://localhost:44371/api/v1.1/book**, and look at the result. You should see the result illustrated in Figure 15.
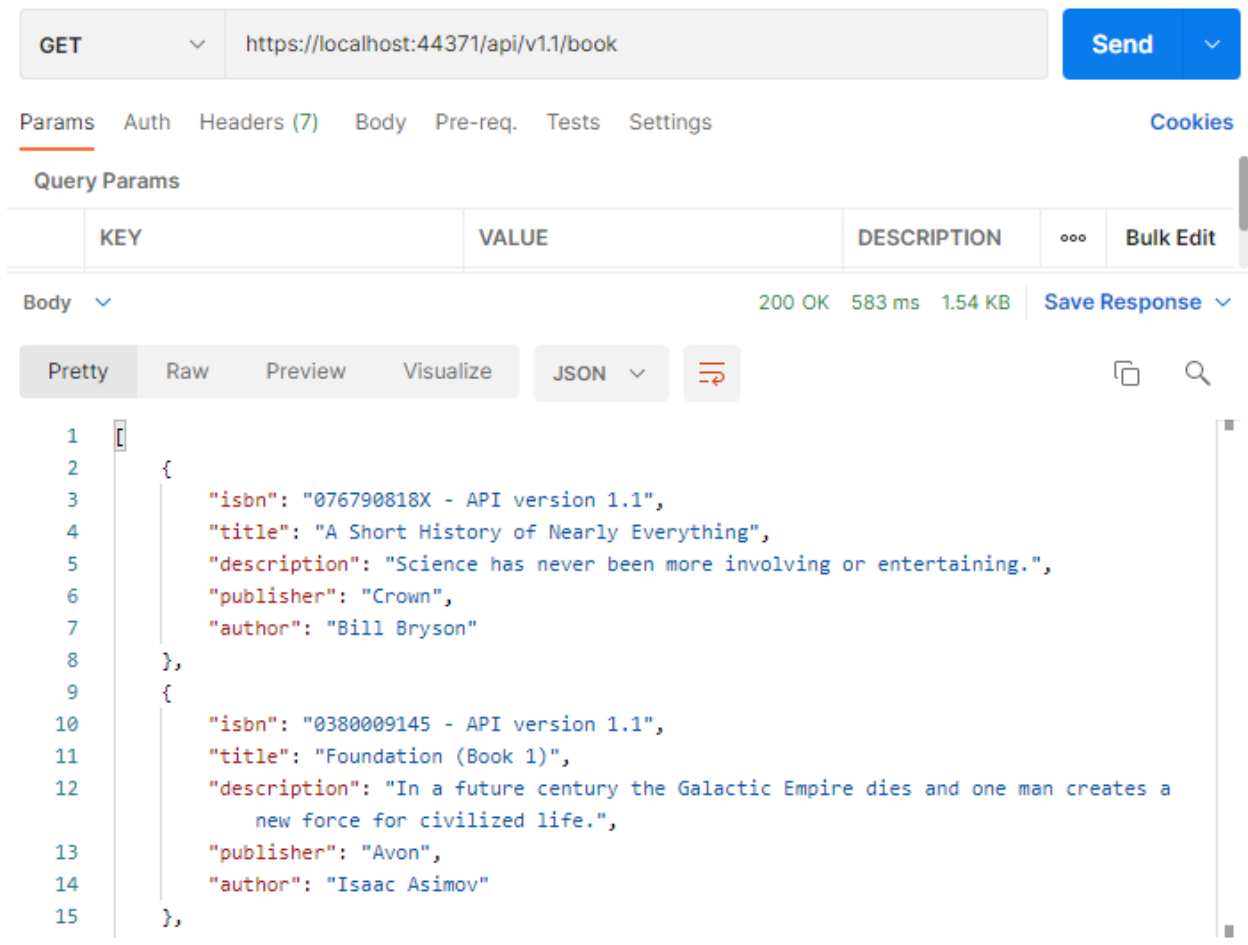
Figure 15: The Updated API Result in Postman

You can change the behavior of Visual Studio so that you won't have to click Save and then manually start the Hot Reload. In the dropdown menu next to the Hot Reload button, select **Hot Reload on File Save** (Figure 16).
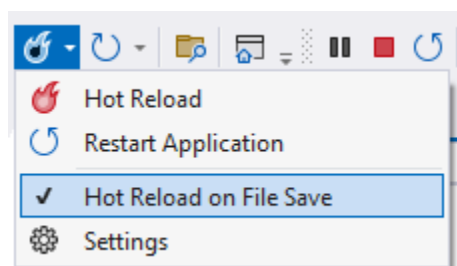
Figure 16: Check Hot Reload on File Save

With this option selected, there will automatically be a Hot Reload of your project whenever you save your code changes. Now you can save a button click before doing a Hot Reload—and you can save two button clicks by using just the keyboard shortcut Ctrl+S. This makes using Hot Reload very convenient.

Limitations of Hot Reload

Hot Reload is not, unfortunately, a magic bullet; it does have some limitations. To illustrate this, consider the code contained in the Models folder inside the **BookModel** class. The code is illustrated in Code Listing 12.

Code Listing 12: The BookModel Class

```csharp
using System.ComponentModel.DataAnnotations;

namespace BookRepository.Models
{
    public class BookModel
    {
        [Required]
        public string ISBN { get; set; }
        [Required]
        public string Title { get; set; }
        public string Description { get; set; }
        public string Publisher { get; set; }
        public string Author { get; set; }
    }
}
```

Ensure that you have selected **Hot Reload on File Save** as shown in Figure 16 and run your Book Repository API.

Make a change to the **Author** property and mark it as **Required**, as seen in Code Listing 13. After you have made the change, save it.

Code Listing 13: Changing the BookModel Class

```csharp
using System.ComponentModel.DataAnnotations;

namespace BookRepository.Models
{
    public class BookModel
    {
        [Required]
        public string ISBN { get; set; }
        [Required]
        public string Title { get; set; }
        public string Description { get; set; }
        public string Publisher { get; set; }
        [Required]
        public string Author { get; set; }
    }
}
```
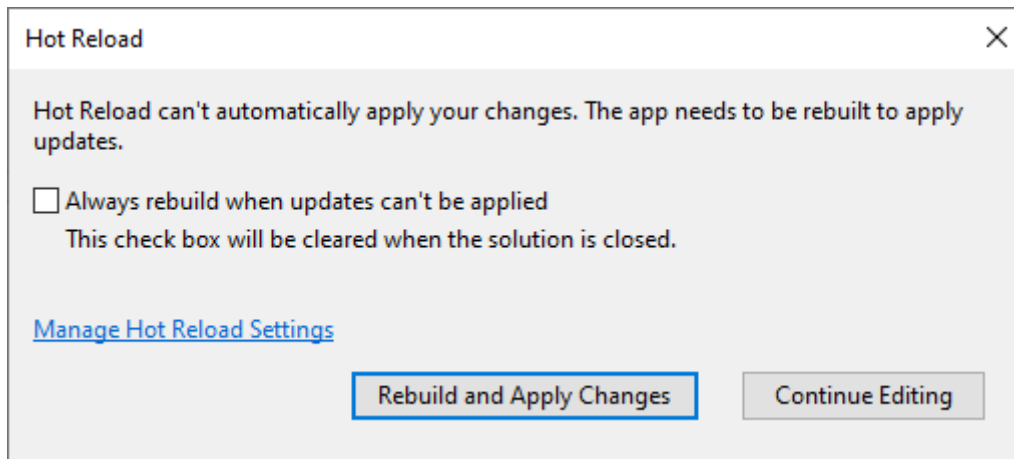
Figure 17: Limitations of Hot Reload

Visual Studio will display a message informing you that Hot Reload can't be applied against the changes you made (Figure 17). You can then rebuild and apply the changes or continue editing. For the most part, Hot Reload is a great new feature, even with the limitations of not being able to apply certain changes.

Looking at a few C# 10 features

.NET 6.0 and C# 10 bring with them a lot of new features. While this book is not a deep dive into C# 10, I will briefly touch on several of the new features. To see if the application you created is running C# 10, have a look at the project's properties. You will notice in Figure 18 that the properties have changed somewhat from previous versions of Visual Studio.
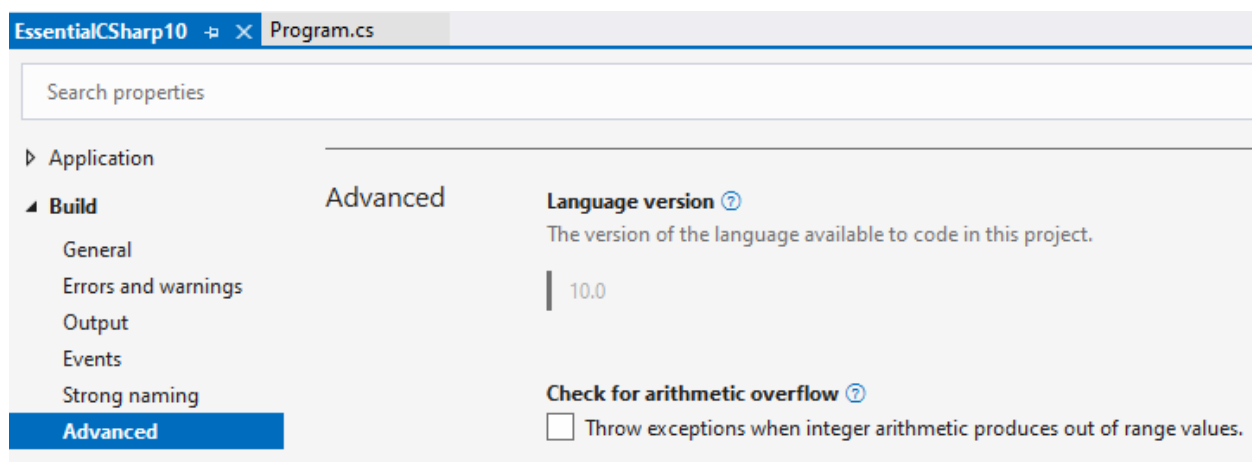

Figure 18: Checking the Available Language Version

Under **Build** > **Advanced**, you will see the Language version available. Here you will see that C# 10.0 is the version of C# that will be available to your project. Next, let's have a look at some of the new features introduced.

Global using statements

In C# 10, `using` directives have been simplified. You can now implement the new `global using` directive, and in doing so, reduce the number of usings required at the top of each file.
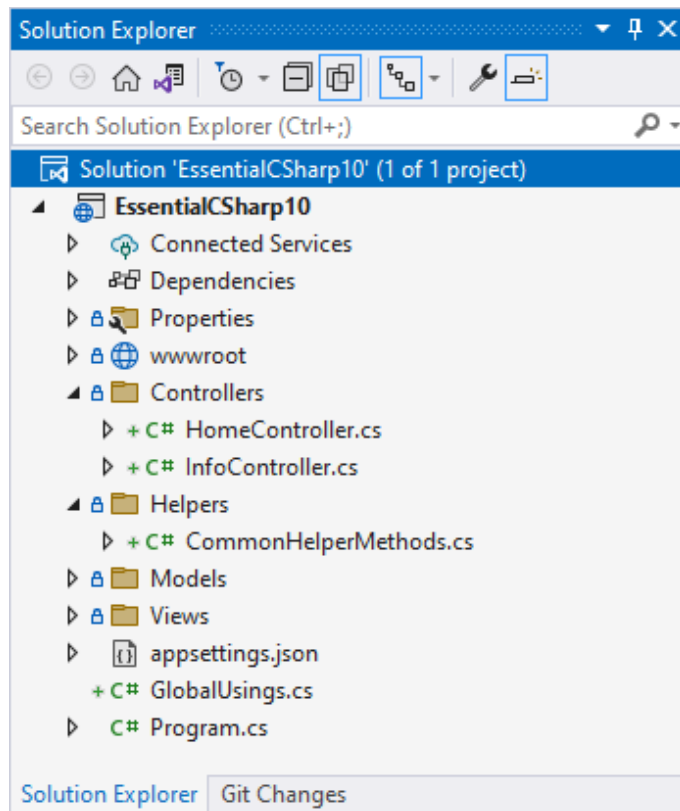

Figure 19: The Project Structure

Looking at the project in Figure 19, you will notice that I just added a file called `GlobalUsings`. I also want to use a helper class called `CommonHelperMethods` in my controllers. This means that I need to add the `using EssentialCSharp10.Helpers;` directive to each controller I want to use my helper methods in. With `global using` directives, I can add this in a single place and have it applied throughout my project.

This means that by adding the directive `global using EssentialCSharp10.Helpers;` to my `GlobalUsings` class, I can use the methods of my helper class throughout my project.

💡*Tip: It is important to note that you can put global usings in any .cs file. This includes the Program.cs file. I just added a specific file called* `GlobalUsings` *to make the intent clear.*

That being said, the `global using` directive can be added to the beginning of any source code file. It must appear:

- Before `using` directives without the `global` modifier.
- Before namespace and type declarations in the file.

It is also worth noting that you can combine the **global** modifier and the **static** modifier. This means you can add **global using static System.Console;** to your **GlobalUsings** file. You can also apply the **global** modifier to aliases in the case of **global using Env = System.Environment;**.

Implicit using statements

This feature automatically adds often-used **global using** directives for the project type that you are building. To enable this, set the **ImplicitUsings** property in your csproj file as illustrated in Code Listing 14.

Code Listing 14: Implicit Usings Defined in the csproj File

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

Now you can use the types in often-used namespaces without having to add the **using** directives yourself. You can see the generated file in your project's obj directory. In my case, the file generated in the obj directory was called **EssentialCSharp10.GlobalUsings.g.cs**, and looking inside the file reveals the **global using** directives implicitly added for my project type.

Code Listing 15: The Generated GlobalUsings File

```csharp
// <auto-generated/>
global using global::Microsoft.AspNetCore.Builder;
global using global::Microsoft.AspNetCore.Hosting;
global using global::Microsoft.AspNetCore.Http;
global using global::Microsoft.AspNetCore.Routing;
global using global::Microsoft.Extensions.Configuration;
global using global::Microsoft.Extensions.DependencyInjection;
global using global::Microsoft.Extensions.Hosting;
global using global::Microsoft.Extensions.Logging;
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Net.Http.Json;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

If you don't see the file, rebuild your solution and it will be generated.

File-scoped namespaces

Another very welcome feature is the addition of file-scoped namespaces. Consider the extremely complex helper class I created, which is illustrated in Code Listing 16.

Code Listing 16: The CommonHelperMethods Helper Class

```csharp
namespace EssentialCSharp10.Helpers
{
    public static class CommonHelperMethods
    {
        public static string SanitizeString(string sInput)
        {
            // Do some string sanitization here
            return sInput;
        }
    }
}
```

With file-scoped namespaces, I can add the **namespace** statement, followed by a semicolon, and remove the curly braces as illustrated in Code Listing 17.

Code Listing 17: The Helper Class Using a Namespace Statement

```csharp
namespace EssentialCSharp10.Helpers;
public static class CommonHelperMethods
{
    public static string SanitizeString(string sInput)
    {
        // Do some string sanitization here
        return sInput;
    }
}
```

This simplifies my code and negates an unnecessary level of nesting. You need to remember, however, that file-scoped namespaces are only available in files with a single namespace. You can't add additional namespace declarations (nested or file-scoped).

Constant interpolated strings

Developers can now create constants using string interpolation. Consider the code in Code Listing 18.

```
const string SupportEmail = "support@email.com";
const string SupportNumber = "011-1234567";
const string SupportMessage = "Please contact support at "
    + SupportNumber
    + " or email "
    + SupportEmail;
```

The **SupportMessage** constant is created by adding the other constants to the string that you build up. In C# 10, you can use string interpolation to create a constant.

Code Listing 19: Create a Constant from Constants in C# 10

```
const string SupportEmail = "support@email.com";
const string SupportNumber = "011-1234567";
const string SupportMessage = $"Please contact support at {SupportNumber}
or email {SupportEmail}";
```

The only thing you need to remember is that all the values you add in the braces **{}** must also be constant strings. You can't use other types, such as numeric or date values. This is because they are sensitive to **Culture**.

Razor pages, MVC, and web API improvements

There have been a few updates to Razor pages, MVC, and web APIs in .NET 6. CSS isolation is a new feature that makes it convenient to organize and structure your CSS files to a single page instead of throughout your entire project. This means that you effectively have a scoped CSS page. The benefit of this is that you have a single place to modify styles for a specific page in specific use cases.

Figure 20 illustrates the way developers included styles in their projects before CSS isolation. Developers defined the styles in one or more CSS files that would apply them throughout the application. To target specific elements, however, classes were used. This would result in a rather large and unwieldy CSS file, especially when the site grew in complexity and functionality. It also led to styling conflicts.
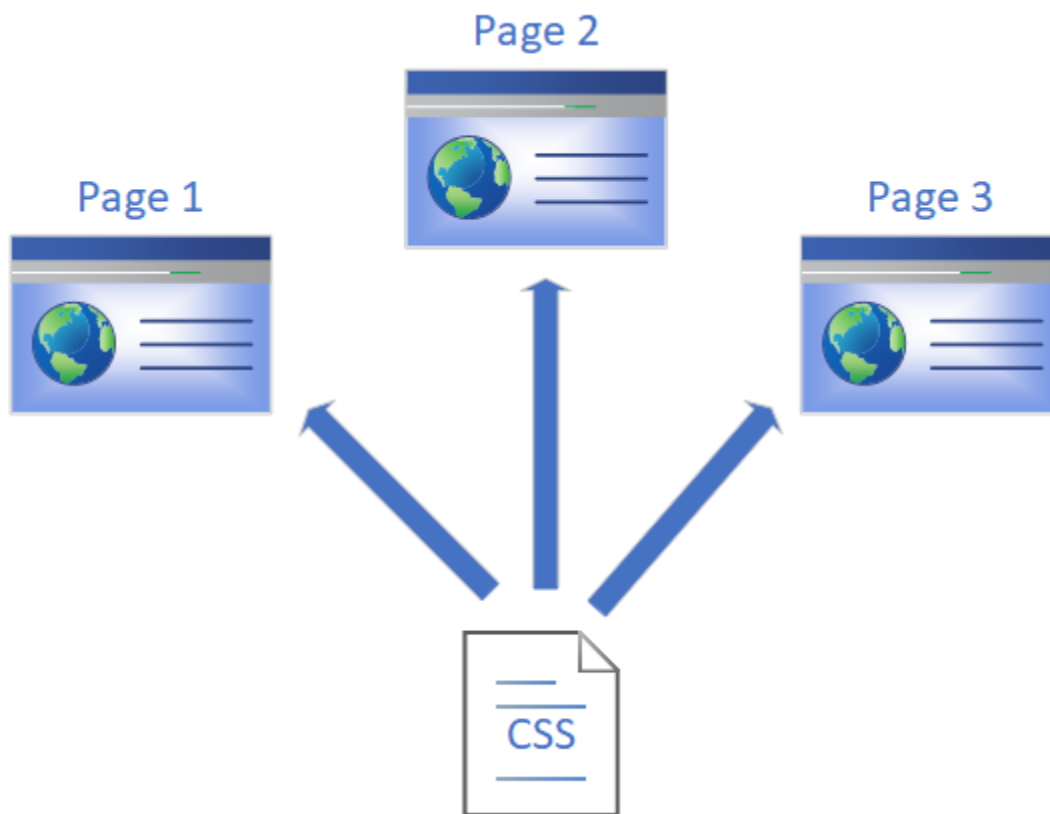
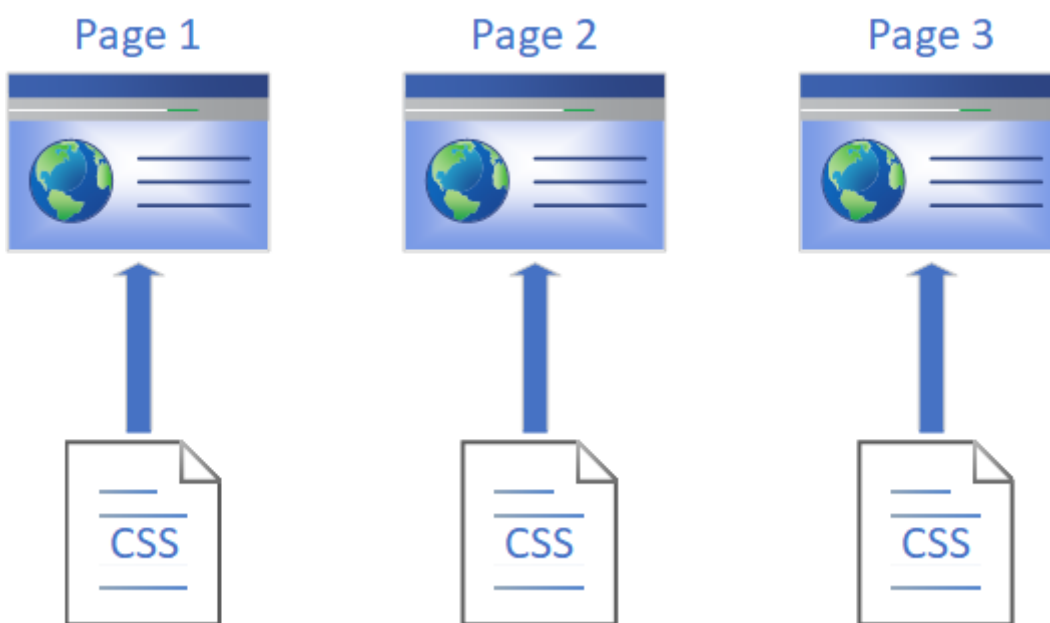*Figure 20: CSS Before CSS Isolation*

Figure 21: CSS After CSS Isolation

With CSS isolation, developers can create CSS pages scoped to specific Razor pages in the application. A global, site-wide CSS file can still be used, but developers can be more granular in their styling approach. The global CSS file can handle general styling throughout the site, while the isolated CSS files handle the styles specific to their page.

ASP.NET Core 6.0 also adds support for HTTP logging. By adding a specific middleware component to the pipeline, developers can view raw logs of the incoming and outgoing HTTP traffic. Let's have a look at these two features in the next sections.

Using CSS isolation

Let's see an example. The project illustrated in Figure 22 uses a single `site.css` file to apply some styling throughout the application.
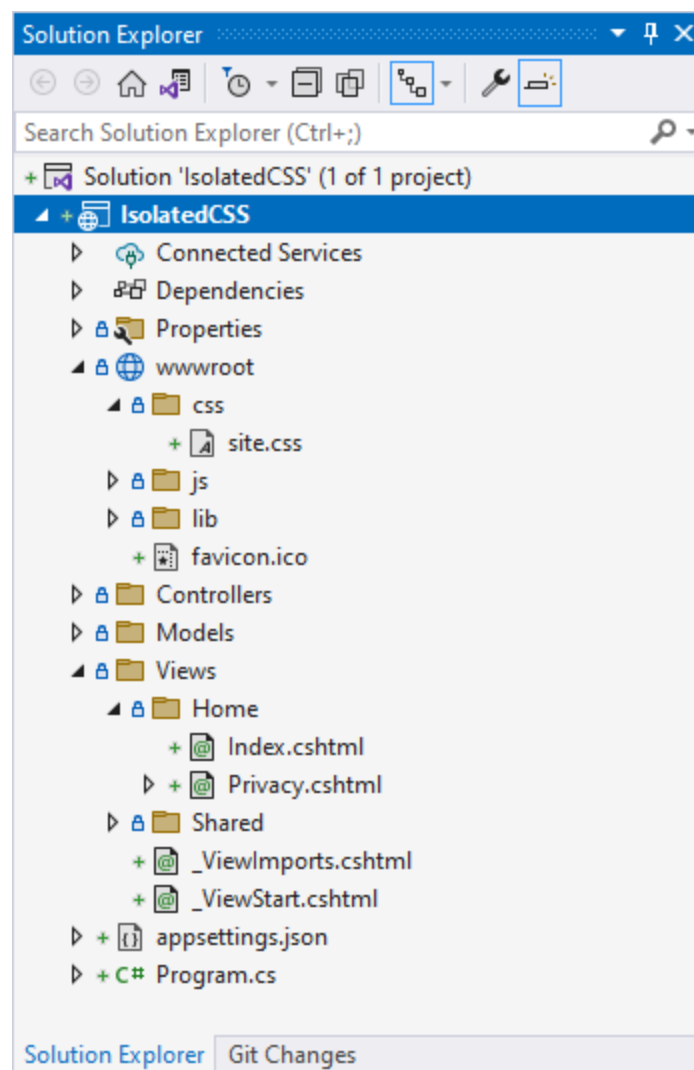

Figure 22: The Project Demonstrating a Site-Wide CSS

Looking at the CSS contained inside this file (Code Listing 20), you will see that apart from the boilerplate CSS, I have added a specific style for **H2** elements that will underline the text.

Code Listing 20: The site.css File

```css
html {
    font-size: 14px;
}

@media (min-width: 768px) {
    html {
        font-size: 16px;
    }
}

html {
    position: relative;
    min-height: 100%;
}

body {
    margin-bottom: 60px;
}

h2 {
        text-decoration: underline;
}
```

Running the web application, you will see that I have an **H2** element on the Home page (Figure 23). It is underlined because the `site.css` file specifies that all **H2** elements need to be underlined.
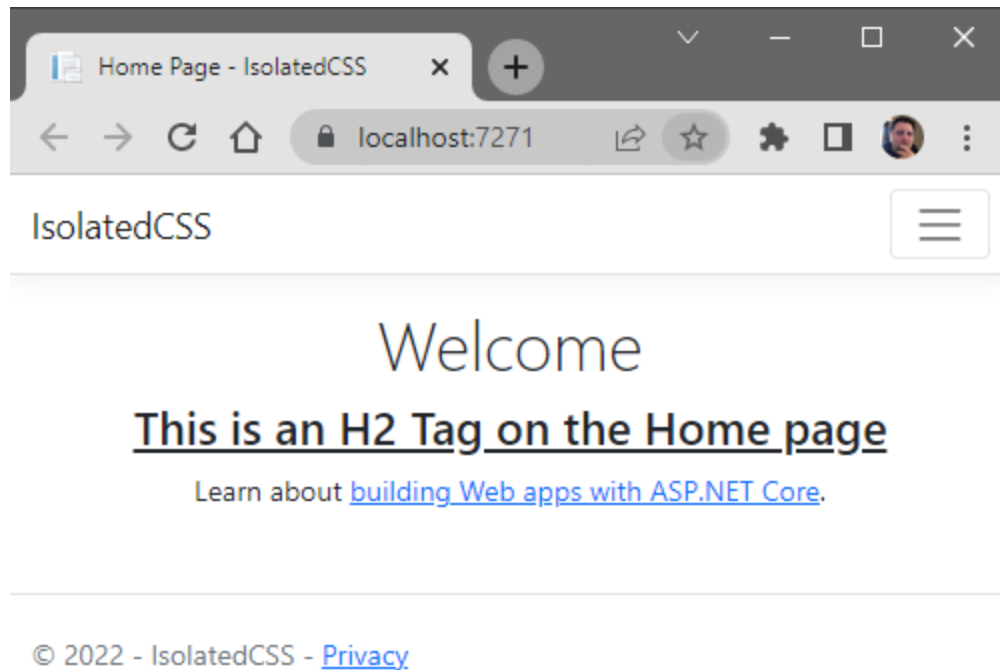
Figure 23: The Home Page Using the site.css

Heading over to then Privacy page (Figure 24), you will see that this same site-wide styling has been applied to its **H2** element.
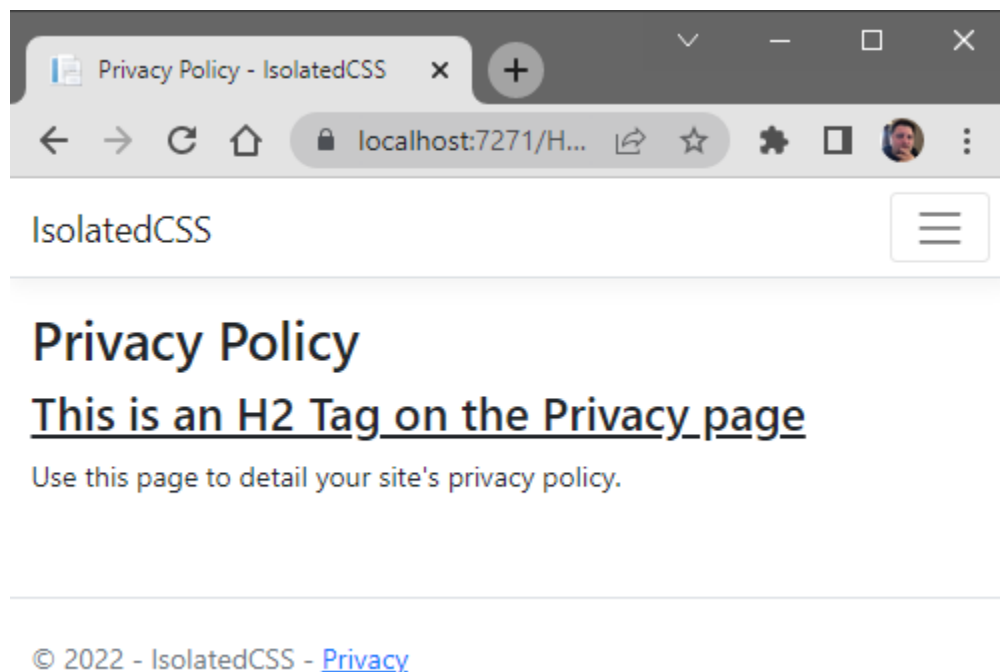

Figure 24: The Privacy Page Using the site.css

I do, however, want to style the **H2** element differently on the Privacy page. Right-click the **Home** folder and select **Add New Item** from the context menu.

Choose a style sheet and name the file **Privacy.cshtml.css**, as seen in Figure 25. The naming of this file is important. As you can see in Figure 26, when the file is added, it is nested under the Privacy.cshtml file.
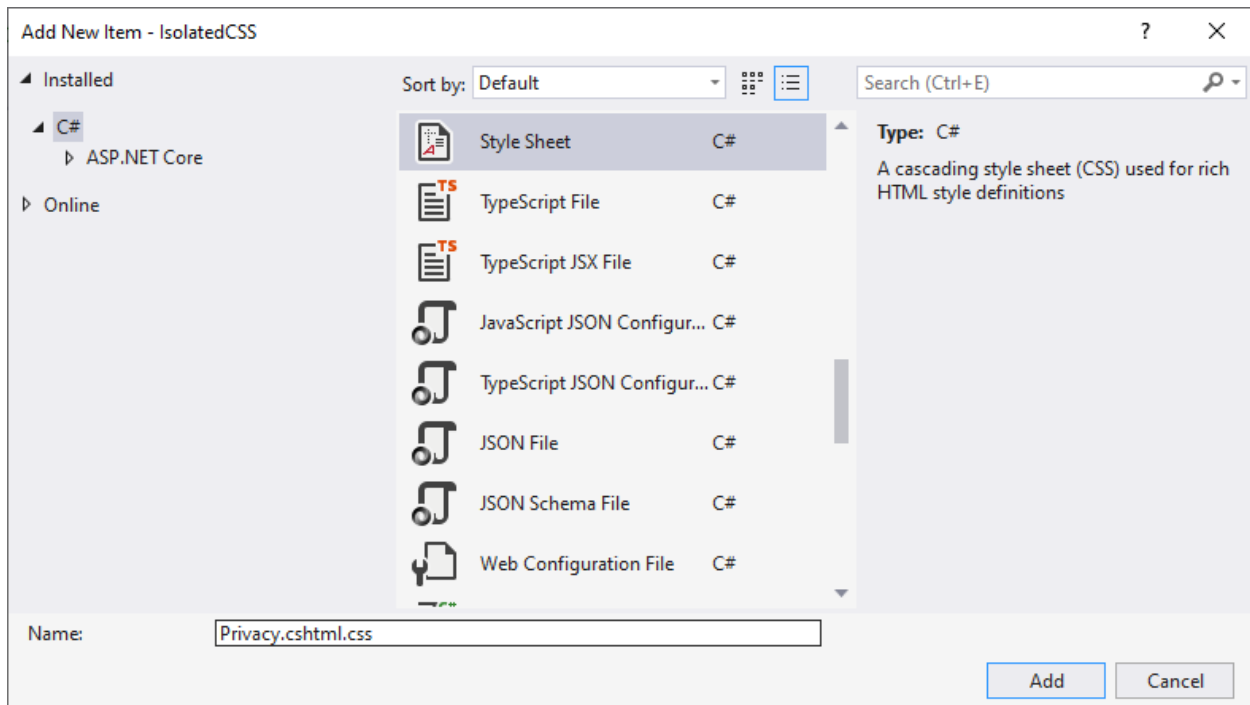

Figure 25: Adding the Isolated CSS File

It is therefore isolated to the Privacy page only, and any styles that you add to this CSS file will only apply to the Privacy page.
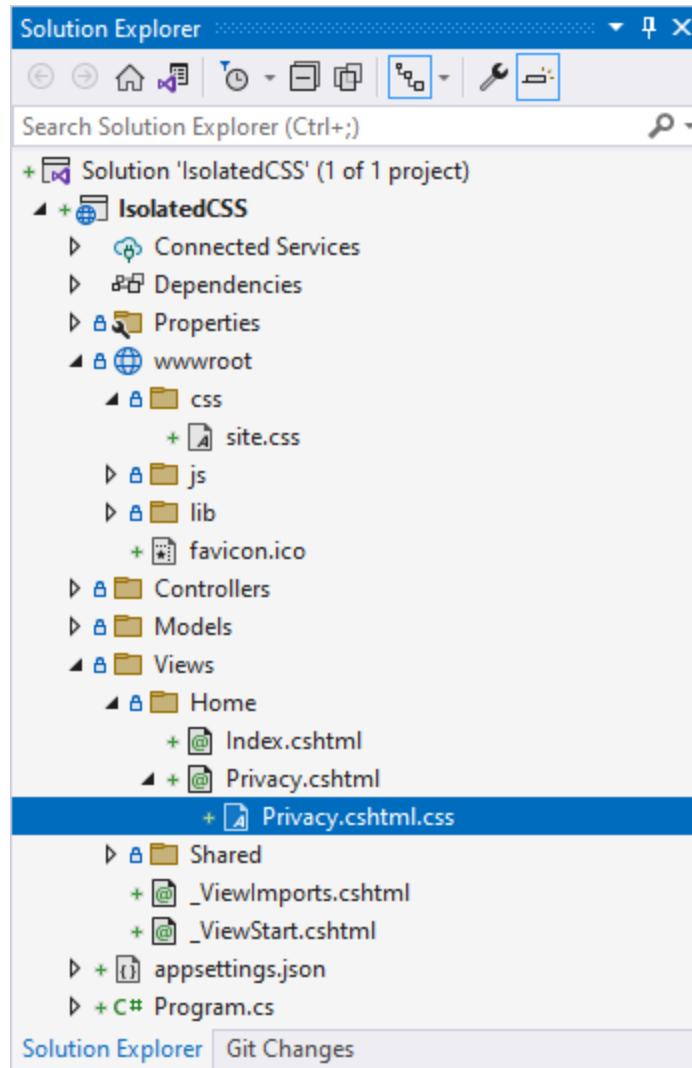
Figure 26: Added Privacy.cshtml.css Isolated File

Edit the **Privacy.cshtml.css** file and add the code in Code Listing 21.

Code Listing 21: The Privacy.cshtml.css Isolated File

```css
h2 {
    text-decoration: line-through;
}
```

You should now have CSS in the **site.css** file that adds **text-decoration: underline** to **H2** elements, and CSS in the **Privacy.cshtml.css** file that adds **text-decoration: line-through** to **H2** elements on the Privacy page only.

Run your web application, and you will see that the Home page still applies the CSS for **H2** elements as defined in the **site.css** file, as illustrated in Figure 27.
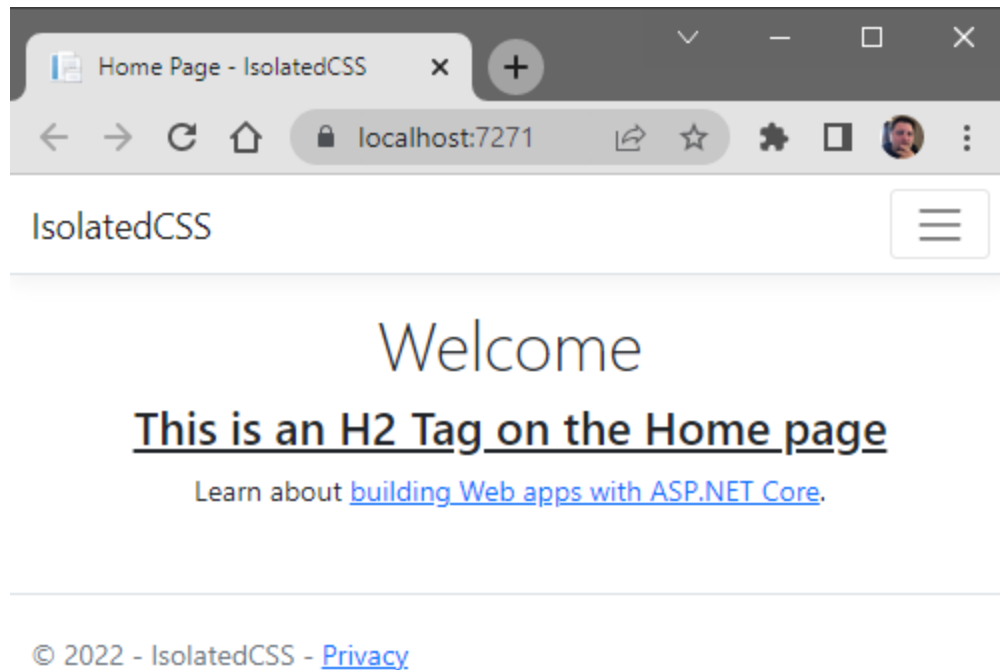
Figure 27: The Home Page Still Using the site.css

If you navigate to the Privacy page, you will see, as illustrated in Figure 28, that the isolated CSS style has been applied only to the Privacy page.
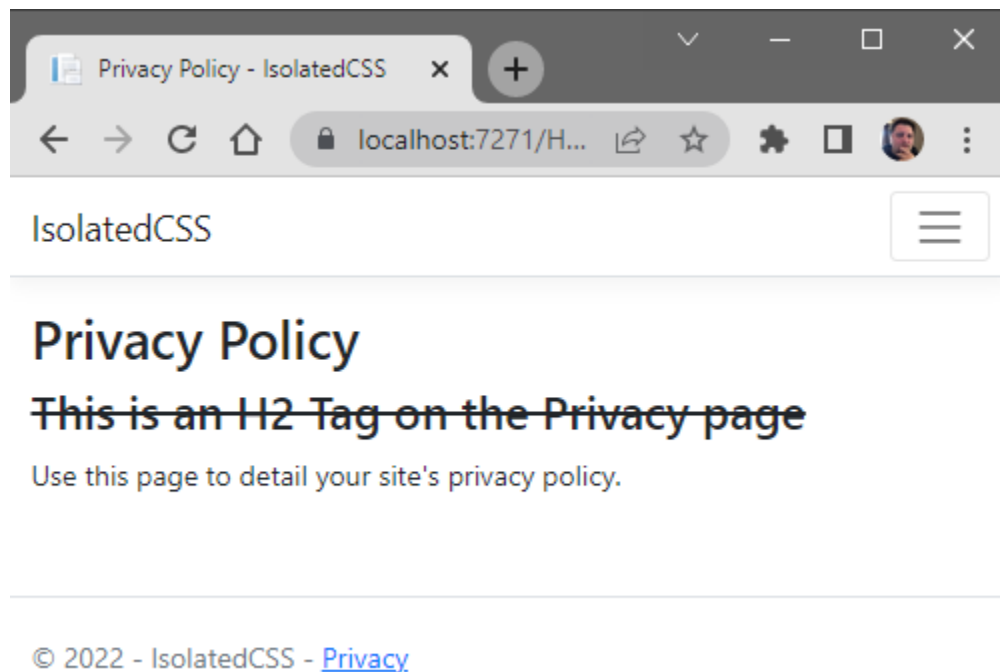

Figure 28: The Privacy Page Using the Isolated CSS File

If you view the source in the browser developer tools, you will see that ASP.NET has generated a styles.css file for you as shown in Figure 29.
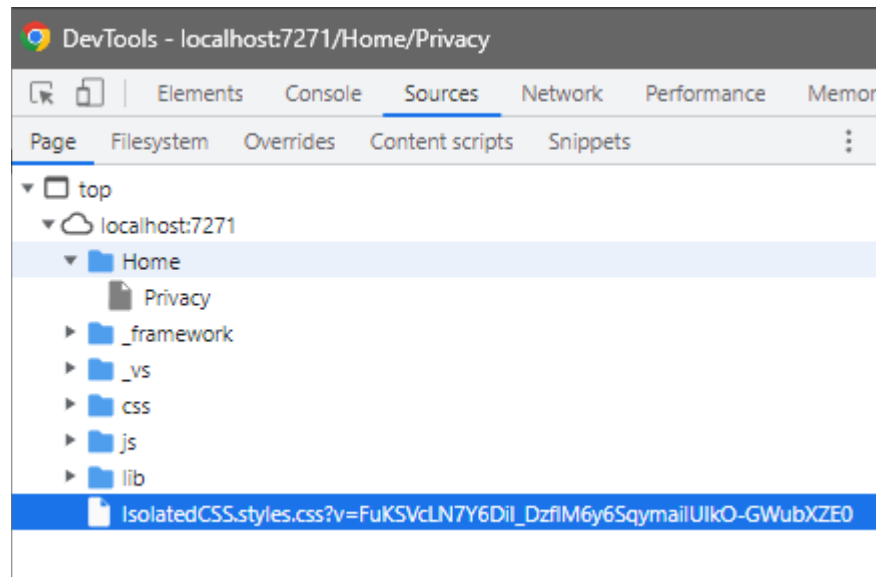
Figure 29: The Generated styles.css File

Looking at the contents of this generated file in Figure 30, you will notice that the style you added for the **H2** element in the isolated Privacy page's CSS file is contained in this generated **styles.css** file with a specific selector.

Figure 30: The Generated styles.css File Contents

The generated selector **h2[b-l8tbgmvpl4]** is applied to the **H2** element on the Privacy page, as seen in Figure 31.
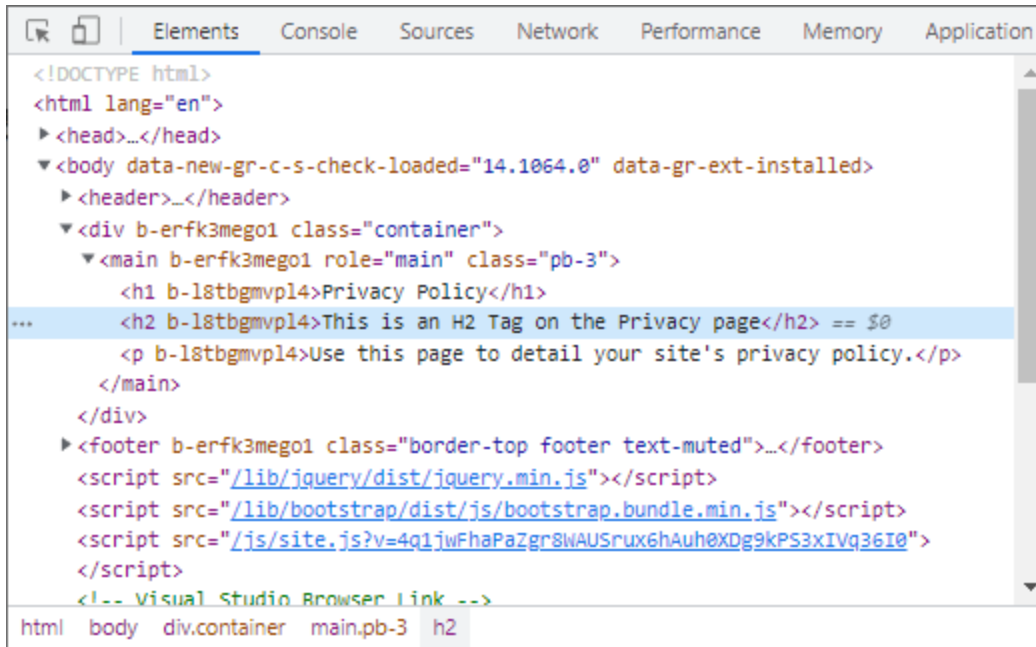
Figure 31: The Elements Seen in Developer Tools

This ensures that styles you add to isolated CSS files are always applied to only that specific page and will not conflict with other styles on your site.

HTTP logging improvements

ASP.NET Core 6.0 has added some useful features that allow developers to manage HTTP requests. In this section, you will use the BookRepository API project and add HTTP logging to it.

📝 ***Note: The book repository API project was upgraded from ASP.NET Core 5.0 to ASP.NET Core 6.0 in*** <u>Chapter 1</u>***. If you haven't done so yet, ensure that you have completed the upgrade.***

By adding HTTP logging, we can monitor the requests and responses coming into the API. To start, we will need to add some middleware to our Startup.cs file. Adding one line of code, **app.UseHttpLogging();**, will allow ASP.NET Core 6.0 to log the requests and responses.

Your **Configure** method will look like the code in Code Listing 22.

Code Listing 22: Add HTTP Logging Middleware

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

```
        _ = app.UseSwagger();
        _ = app.UseSwaggerUI(c =>
c.SwaggerEndpoint("/swagger/v1/swagger.json", "BookRepository v1"));
    }

    _ = app.UseHttpLogging();

    _ = app.UseHttpsRedirection();

    _ = app.UseRouting();

    _ = app.UseAuthorization();

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

This adds the logging to our middleware pipeline with default settings.

> **Note: Because we upgraded an ASP.NET Core 5.0 application to ASP.NET Core 6.0, it still uses an explicit** `Program` **class and** `Main` **method instead of top-level statements.**

The only other change we need to make is to ensure that the **appsettings.json** file specifies the appropriate logging level of **Information**.

In Solution Explorer, edit the **appsettings.json** file and ensure that the log levels are set to **Information**, as illustrated in Code Listing 23.

Code Listing 23: The LogLevel Set to Information

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "BookConn": "Data Source=(localdb)\\MSSQLLocalDB;Initial
Catalog=BookRepo;Integrated Security=True"
  }
}
```
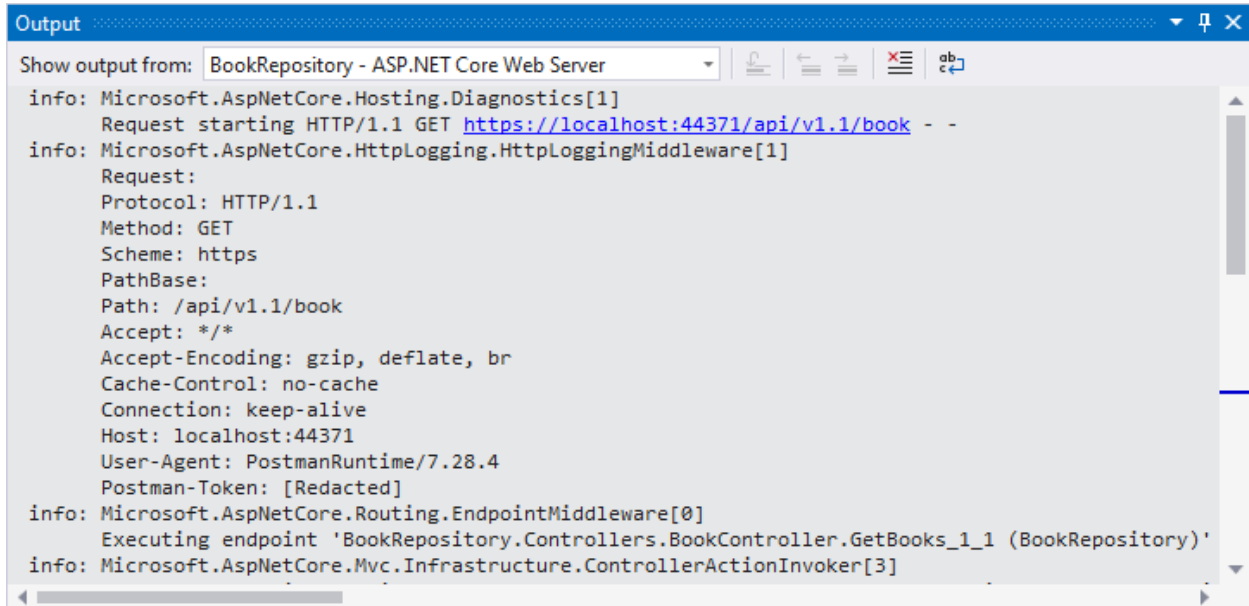
The log level determines the severity of errors that are logged out to the console. In this instance, we have lowered it to the value of `Information` instead of `Warning`.

> 📝 ***Note: When logging output like this, there are performance and security considerations to keep in mind. The danger here is that you might log sensitive request data in some instances. Therefore, these settings need to be disabled or configured correctly when moving to a production environment.***

Run your API and in Postman make a `GET` request to the URL `https://localhost:44371/api/v1.1/book`, keeping in mind that your URL port could be different.

As you will see in Visual Studio's Output window (Figure 32), the logs are displayed for the `GET` request you sent from Postman.

Figure 32: The HTTP Logs

Further down the Output window, you will see the `200` response info (Figure 33).
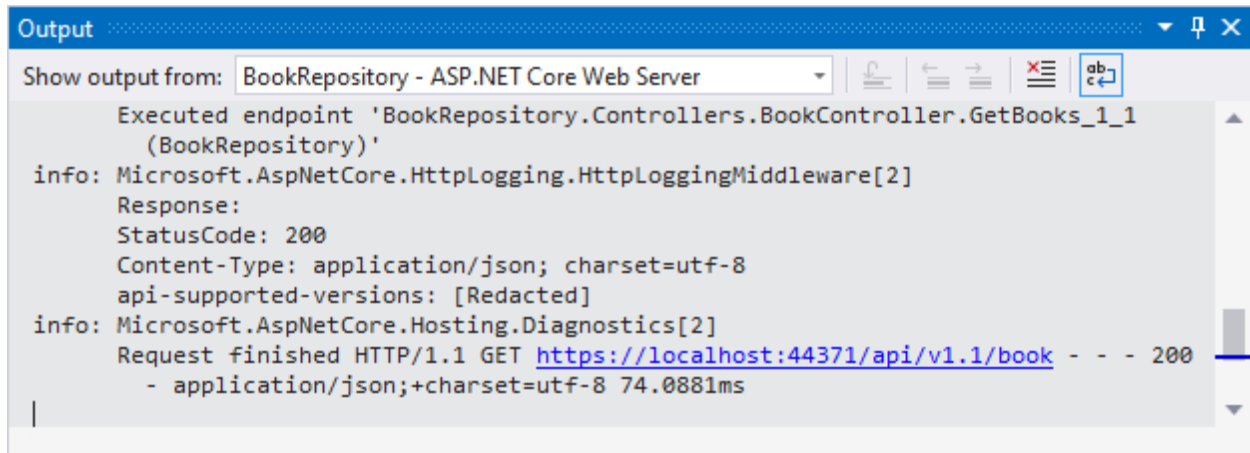
Figure 33: The 200 Response Information

The next request that I want to illustrate is a **POST**. By default, HTTP logging will not include body information in the output. We need to explicitly change this behavior by configuring some settings in the API.

Inside the **ConfigureServices** method, add the code as illustrated in Code Listing 24.

Code Listing 24: Configure Logging to Log All Fields

```
_ = services.AddHttpLogging(log =>
{
    log.LoggingFields =
Microsoft.AspNetCore.HttpLogging.HttpLoggingFields.All;
});
```

This will log everything to the Output window, including the body information. Back in Postman, construct a **POST** calling the URL **https://localhost:44371/api/v1.1/book**, passing it the JSON body in Code Listing 25.

Code Listing 25: JSON to Add a New Book

```
{
    "isbn": "978-1-61268-019-4",
    "title": "Rich Dad Poor Dad",
    "description": "What the Rich Teach Their Kids About Money That the
Poor and Middle Class Do Not!",
    "publisher": "Plata Publishing",
    "author": "Robert T. Kiyosaki"
}
```

Your **POST** in Postman should look as illustrated in Figure 34.
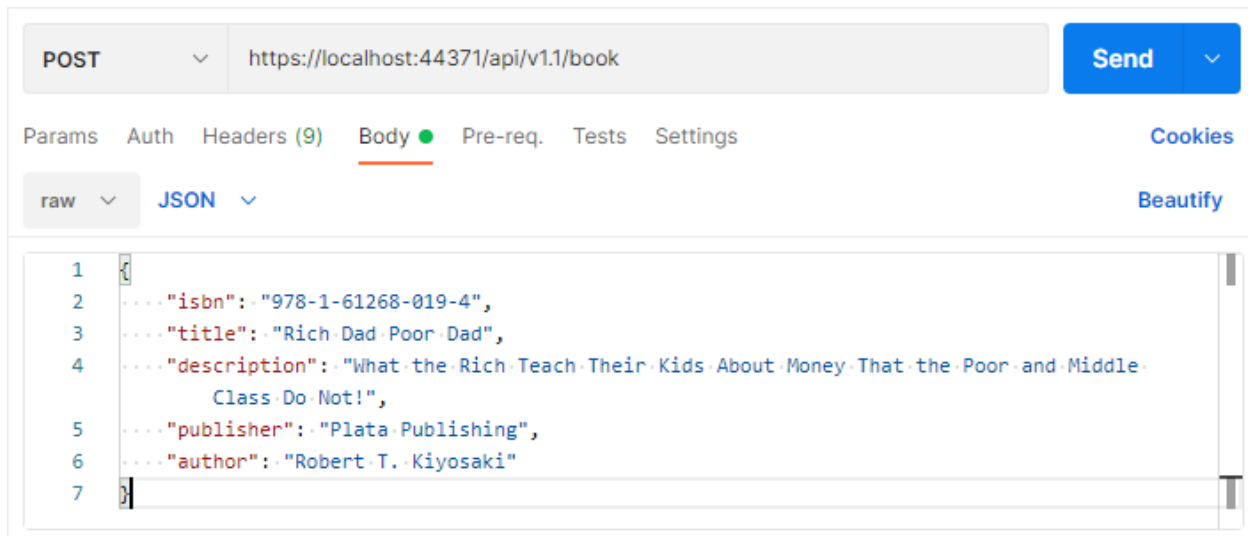
Figure 34: Add a New Book in Postman

When you perform the request, you will see it come through in the Output window of Visual Studio with a method of **POST** (Figure 35).
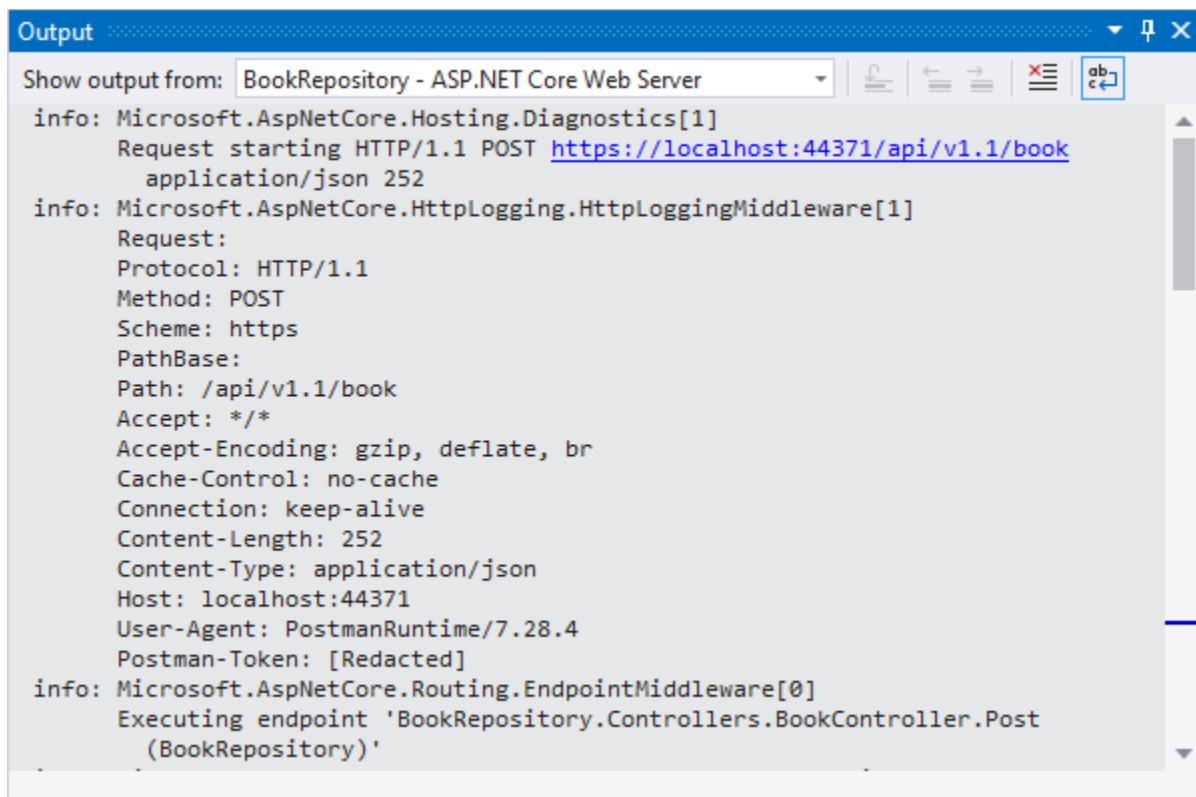
Figure 35: The POST Request

When you scroll down a bit, you will see the request body sent through (Figure 36). This will be the same as the JSON in Code Listing 25.

Figure 36: The POST Request Body

If all goes well, you will receive a **201 Created** response (Figure 37).

Figure 37: The 201 Response

To test if the book was created, perform a **GET** request using the URL
https://localhost:44371/api/v1.1/book/search?Isbn=978-1 and inspect the Output
window in Visual Studio (Figure 38).
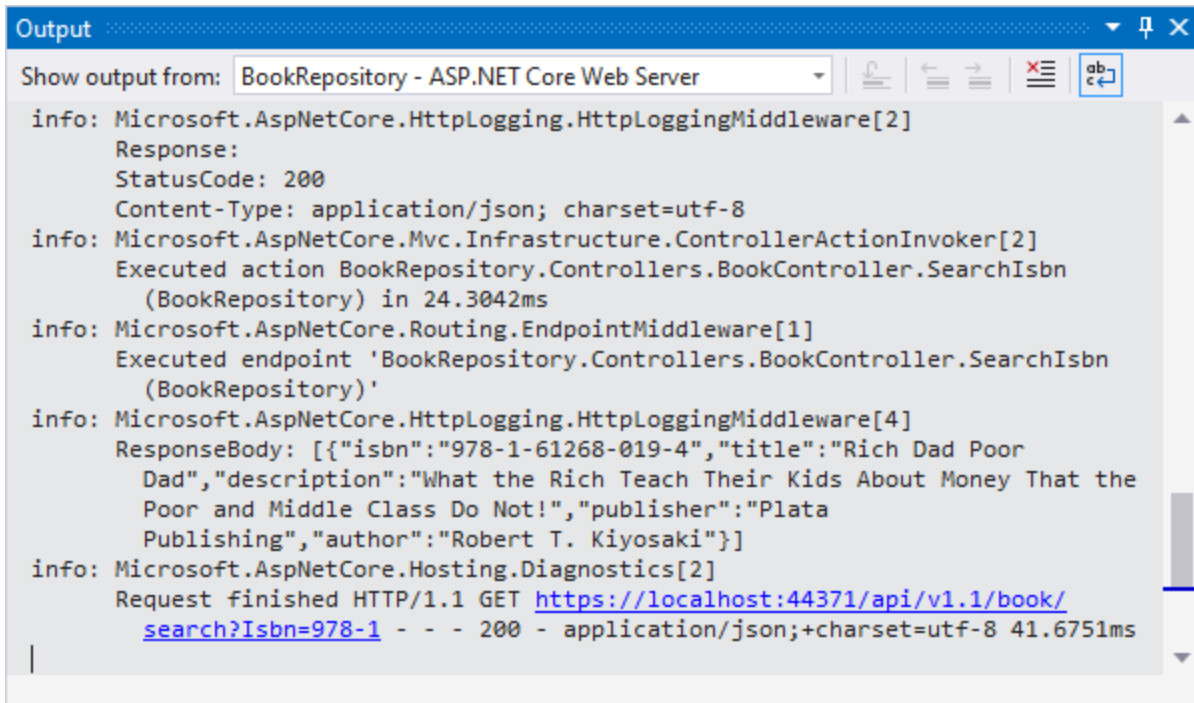
Figure 38: The 200 Response from the GET Request

You will see that the book we just added is returned in the response body. ASP.NET Core 6.0 also adds better support for managing request and response headers in a strongly typed way. Modify the **Configure** method as illustrated in Code Listing 26 and add the **AcceptLanguage** of **en-US**.

Code Listing 26: Adding a Request Header

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        _ = app.UseDeveloperExceptionPage();
        _ = app.UseSwagger();
        _ = app.UseSwaggerUI(c =>
c.SwaggerEndpoint("/swagger/v1/swagger.json", "BookRepository v1"));
    }

    _ = app.Use(async (context, next) =>
    {
        context.Request.Headers.AcceptLanguage = "en-US";
        await next.Invoke(context);
    });

    _ = app.UseHttpLogging();

    _ = app.UseHttpsRedirection();
```

```
    _ = app.UseRouting();

    _ = app.UseAuthorization();

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

Performing a simple **GET** request and looking at the Output window in Visual Studio, you will see the **Accept-Language** header is set to **en-US**.
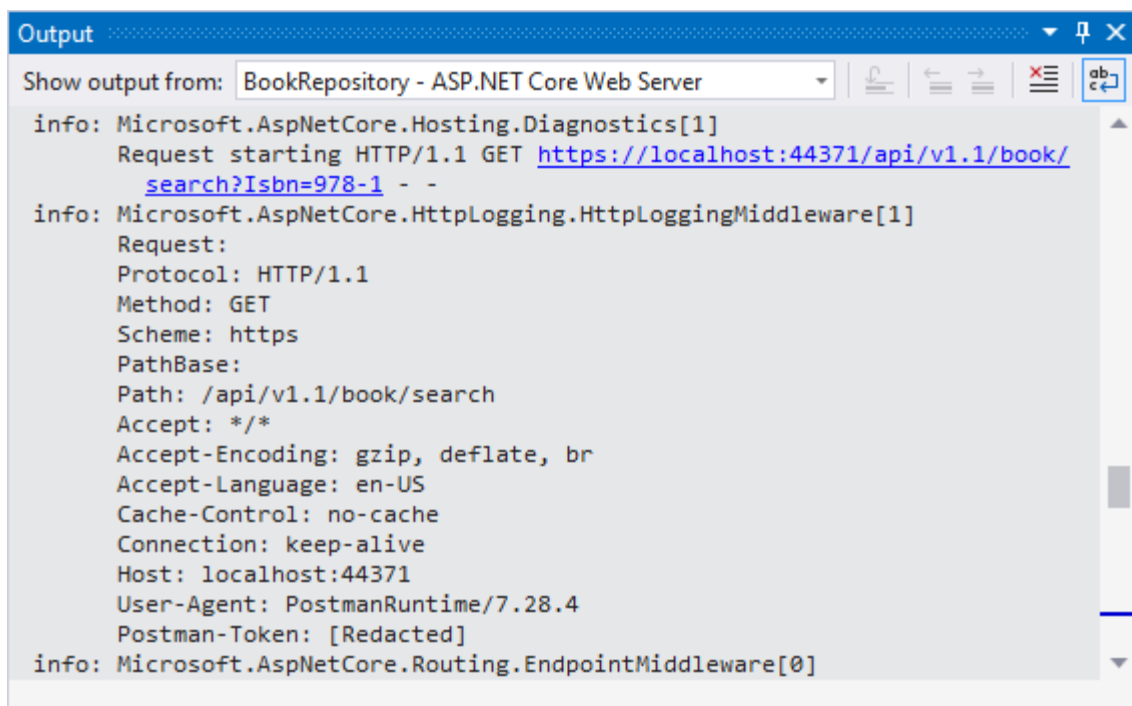
Figure 39: The Accept-Language Set in the Request Header

These logging features improve the experience of analyzing the requests and responses going in and out of your applications.

Chapter 3 Minimal APIs

Web APIs evolved from the MVC pattern that uses controllers and action methods. This has been the standard way of creating APIs. With ASP.NET Core 6.0, one of the biggest new features is the addition of minimal APIs. Minimal APIs provide an alternative API pattern for developers to use. In this chapter, we will have a closer look at what minimal APIs are and how to build one.

What are minimal APIs?

The naming of minimal APIs does not mean that these APIs are limited in features or functionality. Still supporting most of the common features of .NET APIs, minimal APIs support middleware, dependency injection, parameter binding, Swagger, and so on.

The "minimal" part of the name refers to the syntax, dependencies, and the overall structure of the feature. With this said, minimal APIs do have some limitations:

- Minimal APIs do not support filters. This means that you can't use action filters, authorization filters, result filters, or exception filters.
- Form bindings are also not supported. According to some documentation, this will be added in the future.
- Built-in validation support (for example, using the IModelValidator interface) is also not supported.

The question now becomes: when should you use minimal APIs, and when should you use controllers? The short answer is that it all depends on your use case. If you are creating a complex back-end service for an enterprise-scale application, then a controller-based approach will be best. This is especially true if you will need to use some features in your API that only controllers provide.

If, however, you want to create a small and simple API that just returns some data, then a minimal API approach should work for you. It keeps things simple and saves some time when creating the API. You can also easily migrate your minimal API to a controller-based API without too much effort.

It seems wrong to use the word "older," but if you prefer to use the older approach to create APIs, then rest assured that they are still fully supported in ASP.NET Core 6.0. With all this said, let's start by creating a minimal API in the next section.

Creating a minimal API

In Visual Studio 2022, create a new project. In the **Create a new project** dialog screen, search for **Web API** and select **ASP.NET Core Web API**, as seen in Figure 40.
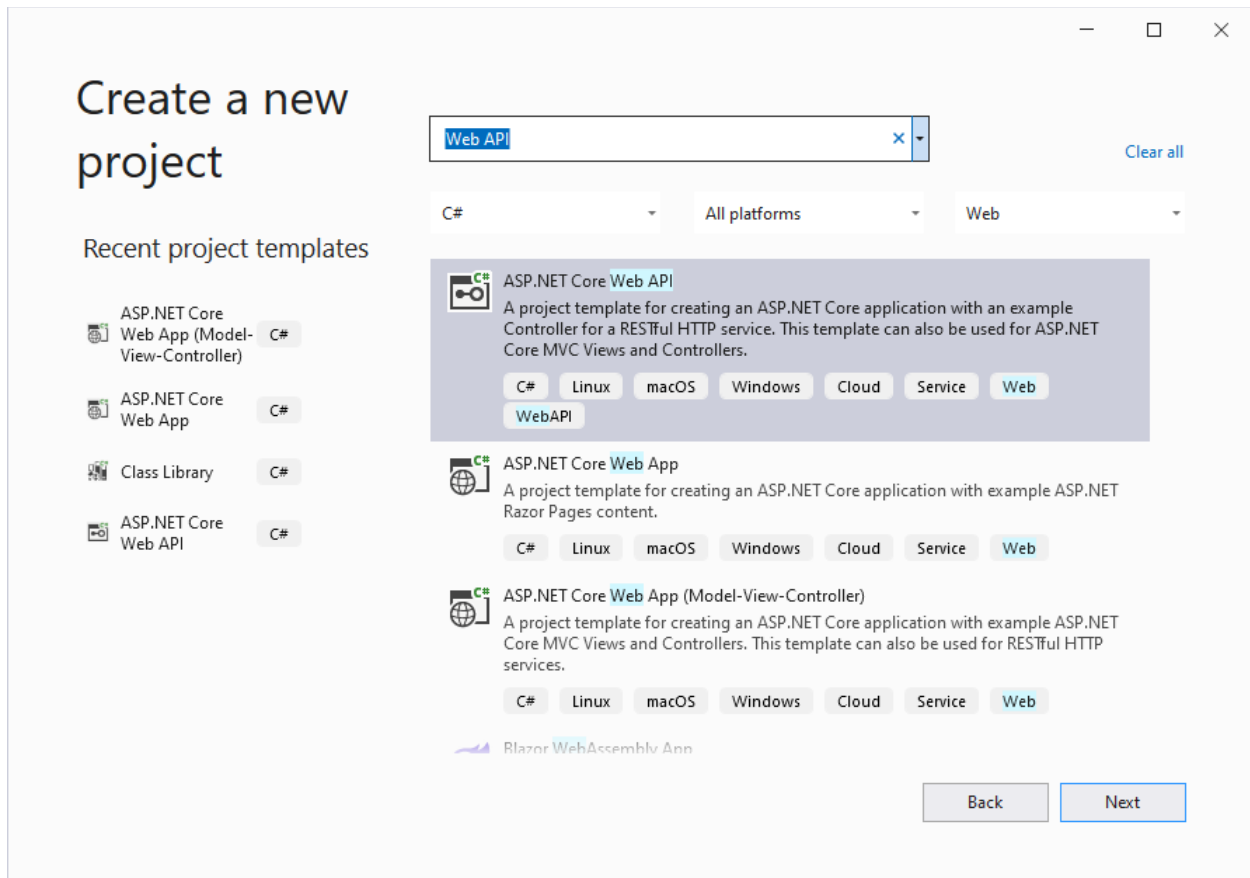
Figure 40: Search for Web API in Create a New Project Dialog

As seen in Figure 41, I am just calling this API **MinAPI**, but you can call it what you like.
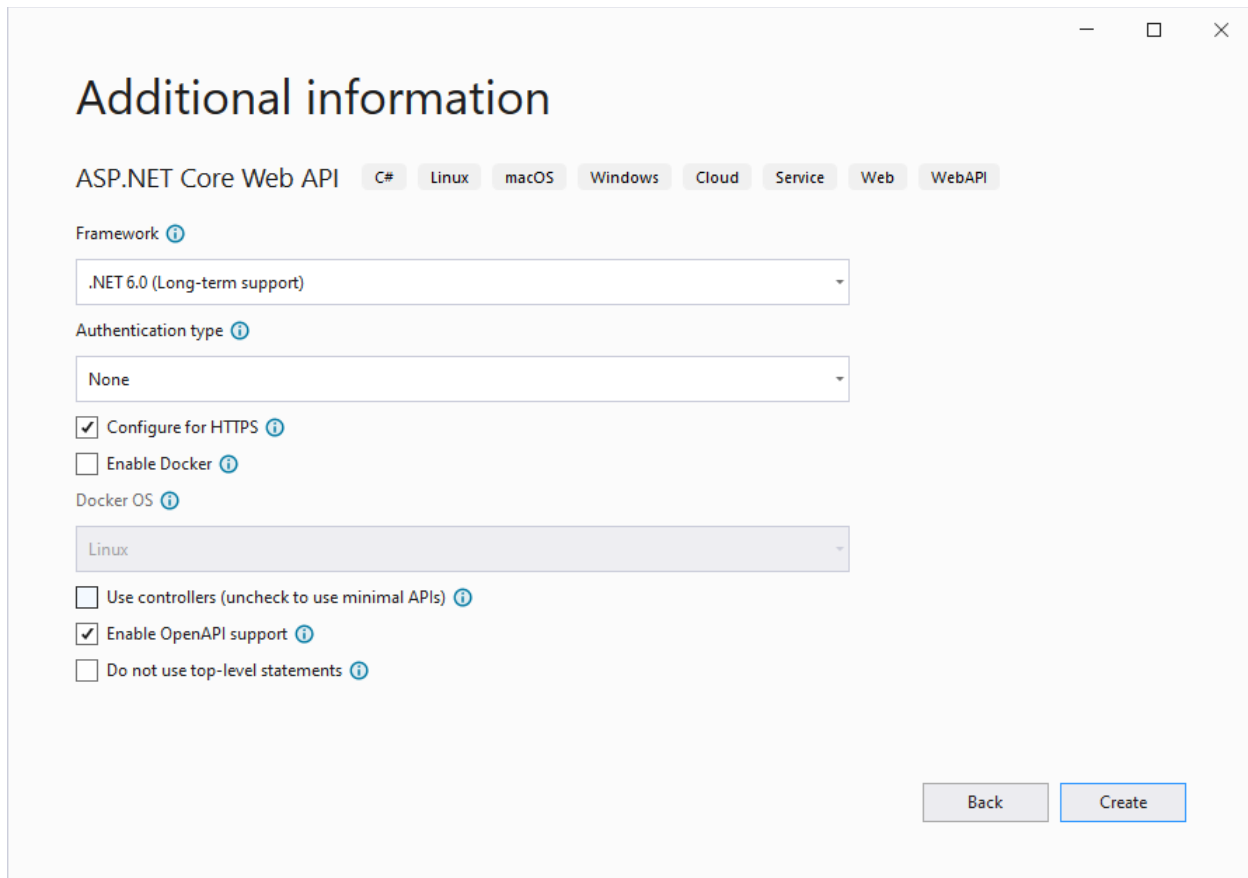
Figure 41: Choose Where to Create the Project

Click **Next**.

On the **Additional information** screen (Figure 42), you will see some options available to you. Make sure that you select **.NET 6.0** as the Framework, and very importantly, clear the option to use controllers. Doing this will use the minimal API approach. You can also select the option to allow Visual Studio 2022 to create the project without top-level statements. For this project, however, we will be using top-level statements.

Figure 42: Clear the Use Controllers Option

📝 ***Note: It does not matter if you select not to use top-level statements in your projects. ASP.NET Core 6.0 is fully compatible with using Program.cs and Startup.cs, just as it is with using a single Program.cs using top-level statements.***

Click **Create** to create your minimal API with the standard Weather Forecast boilerplate code. Have a look at the code contained in the Program.cs file (Code Listing 27).

Code Listing 27: The Minimal API Program File

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
// Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();
```

```csharp
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

var summaries = new[]
{
    "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy",
"Hot", "Sweltering", "Scorching"
};

app.MapGet("/weatherforecast", () =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateTime.Now.AddDays(index),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)]
        ))
        .ToArray();
    return forecast;
})
.WithName("GetWeatherForecast");

app.Run();

internal record WeatherForecast(DateTime Date, int TemperatureC,
string? Summary)
{
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```

Nice and minimal, isn't it? Minimal APIs do not have controllers or action methods. Requests are all handled via the methods in the Program file. Below the middleware pipeline setup (below the **app.UseHttpsRedirection();** line), you will see that an array is used to store the weather summary.

The **MapGet** method will handle incoming **GET** requests to the weather service. It has a string of **/weatherforecast** that maps the URL that the **GET** request should map to.

📝 ***Note: You will notice the*** `WithName` ***method after the*** `MapGet` ***method that takes a string*** `GetWeatherForecast`***. This names the*** `MapGet` ***method for use with OpenAPI and Swagger UI.***

Before you run your API, have a look at the project properties by right-clicking the **MinAPI** project and selecting **Properties** from the context menu.
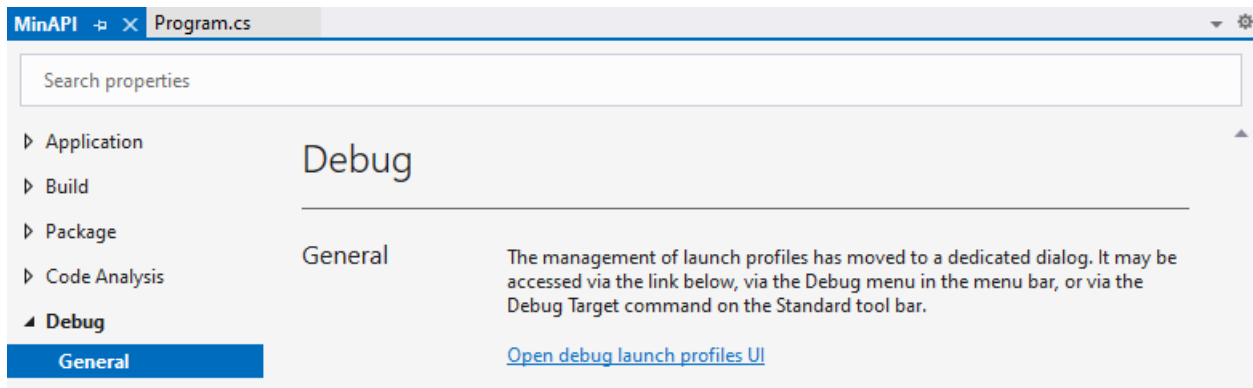
Figure 43: The MinAPI Debug Properties

The project properties will be displayed (Figure 43), and in the **Debug** > **General** tab, click **Open debug launch profiles UI**.

In the **Launch Profiles** window, select **IIS Express** and scroll down until you see the **Launch browser** option. Clear this option, as seen in Figure 44.
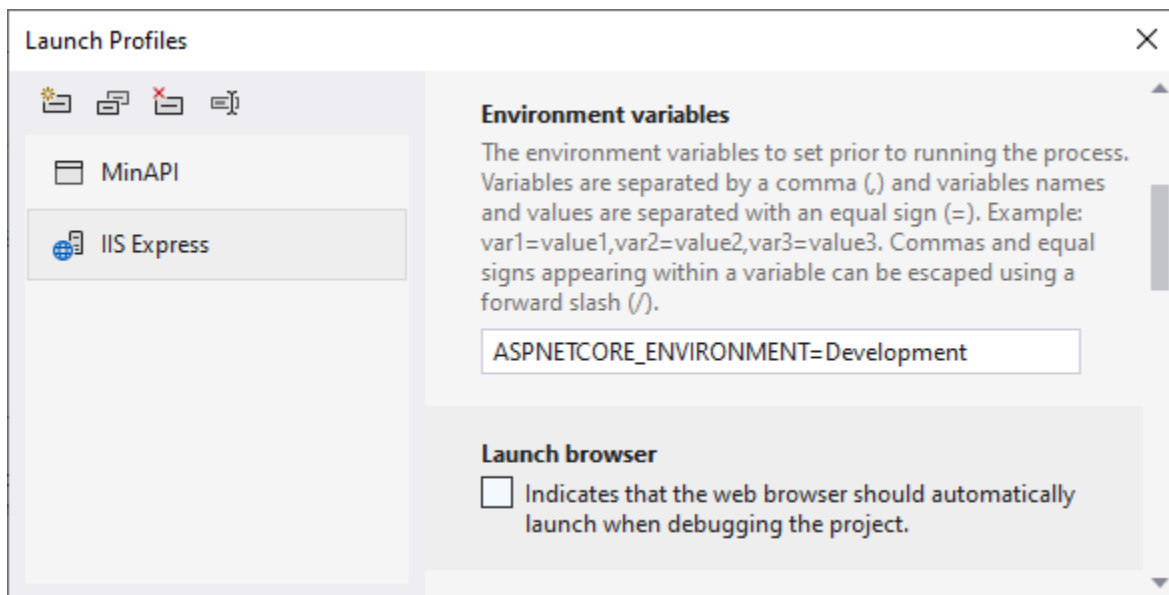
Figure 44: Clear Launch browser Option

Scrolling down a little more, you will see the option to **Use SSL**. Ensure that this option is selected, as shown in Figure 45, and make a note of the App SSL URL and port. This is the URL that you will be using in Postman.
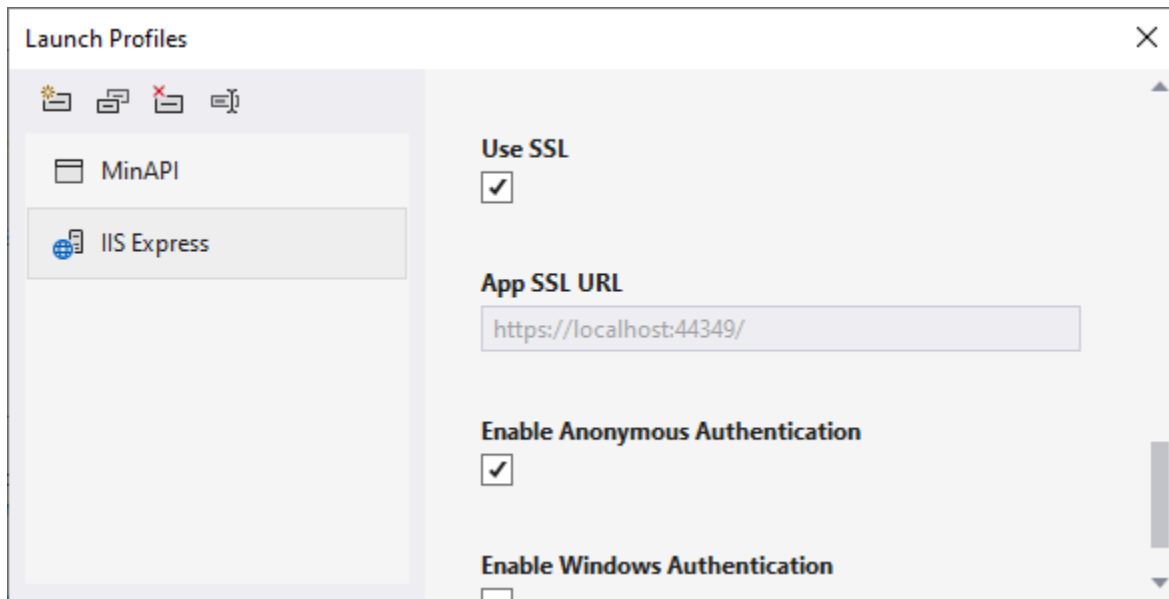
Figure 45: Note the App SSL URL

When you are done, close the Launch Profiles UI and run your API from Visual Studio, ensuring that you have **IIS Express** selected in the **Debug** dropdown.
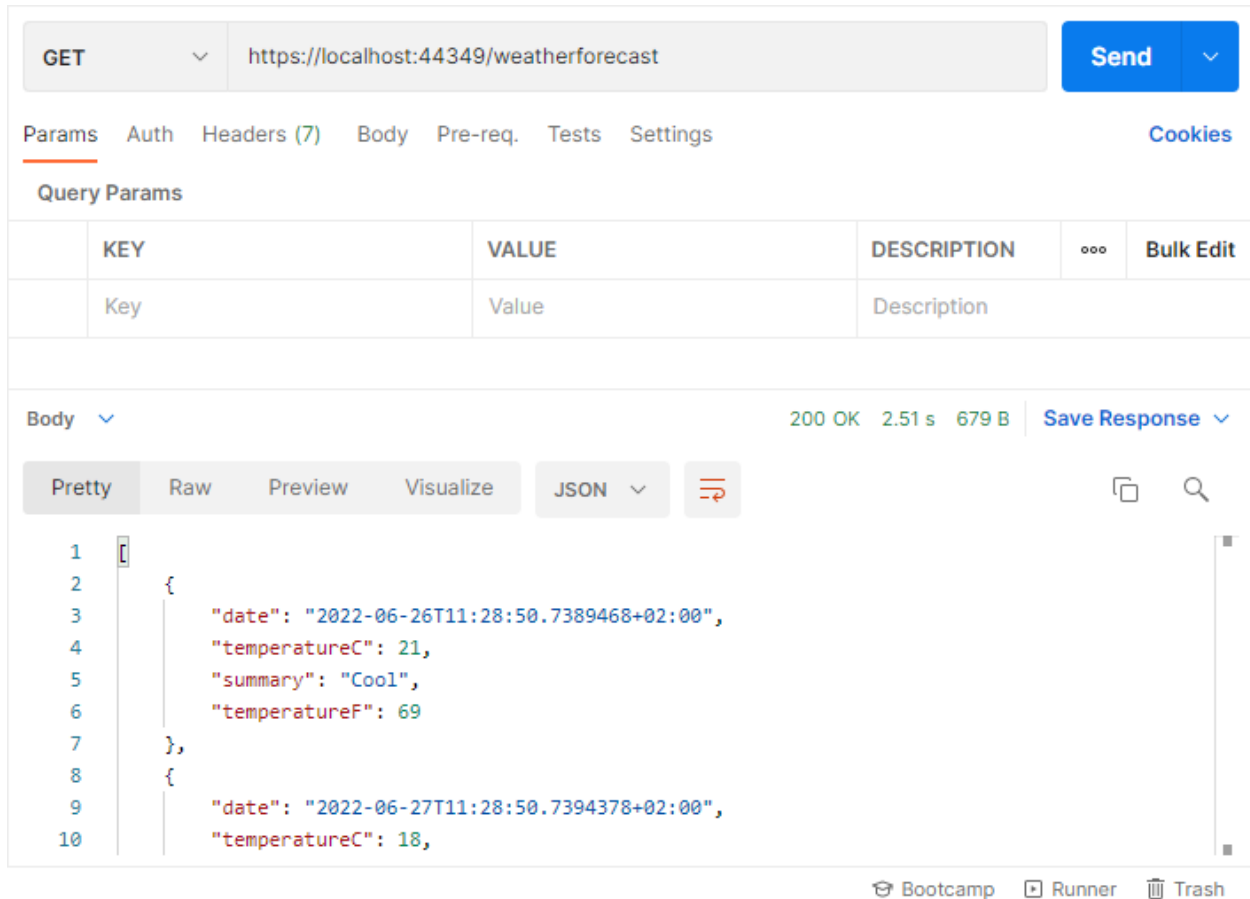
Figure 46: The GET Request in Postman

Because the App SSL URL is set to **https://localhost:44349/**, as seen in Figure 45, and the endpoint URL path on the **MapGet** method (seen in Code Listing 27) is set to **/weatherforecast**, we can create a **GET** request in Postman using the URL **https://localhost:44349/weatherforecast** (as seen in Figure 46).

This returns the weather data from the weather forecast API as expected.
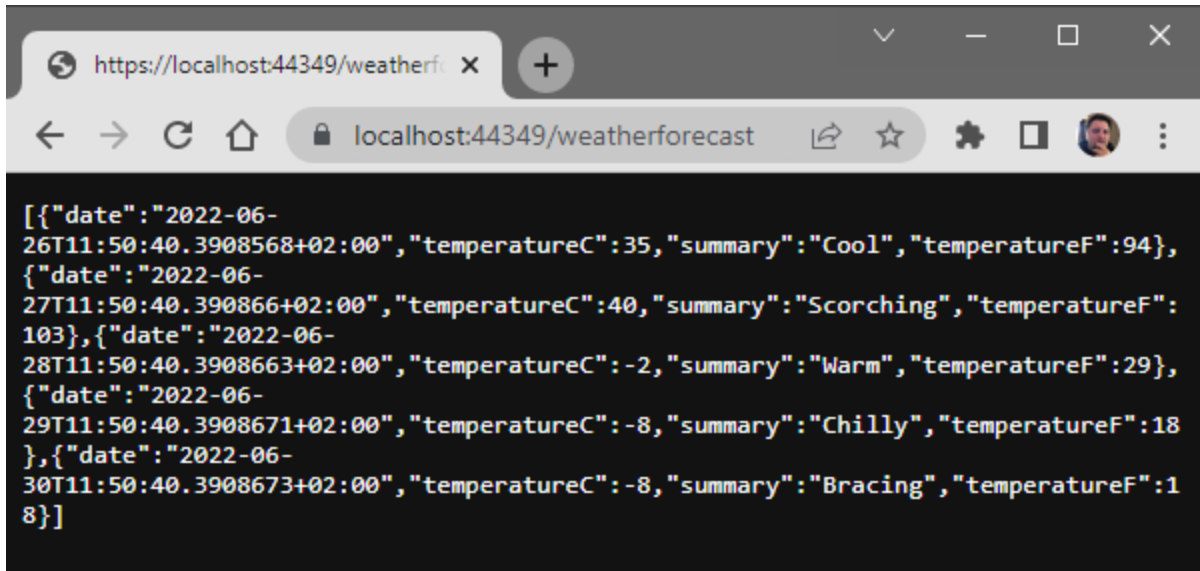
Figure 47: The Weather Forecast in Chrome

I like using Postman to test my APIs, but you can also use your browser to make the **GET** request, as seen in Figure 47. Minimal APIs also have full Swagger support, as seen in Code Listing 27, higher up in the Program file.

Go back to your properties and click to modify the Launch Profile for **IIS Express**, as shown in Figure 48. Select the option **Launch browser**.
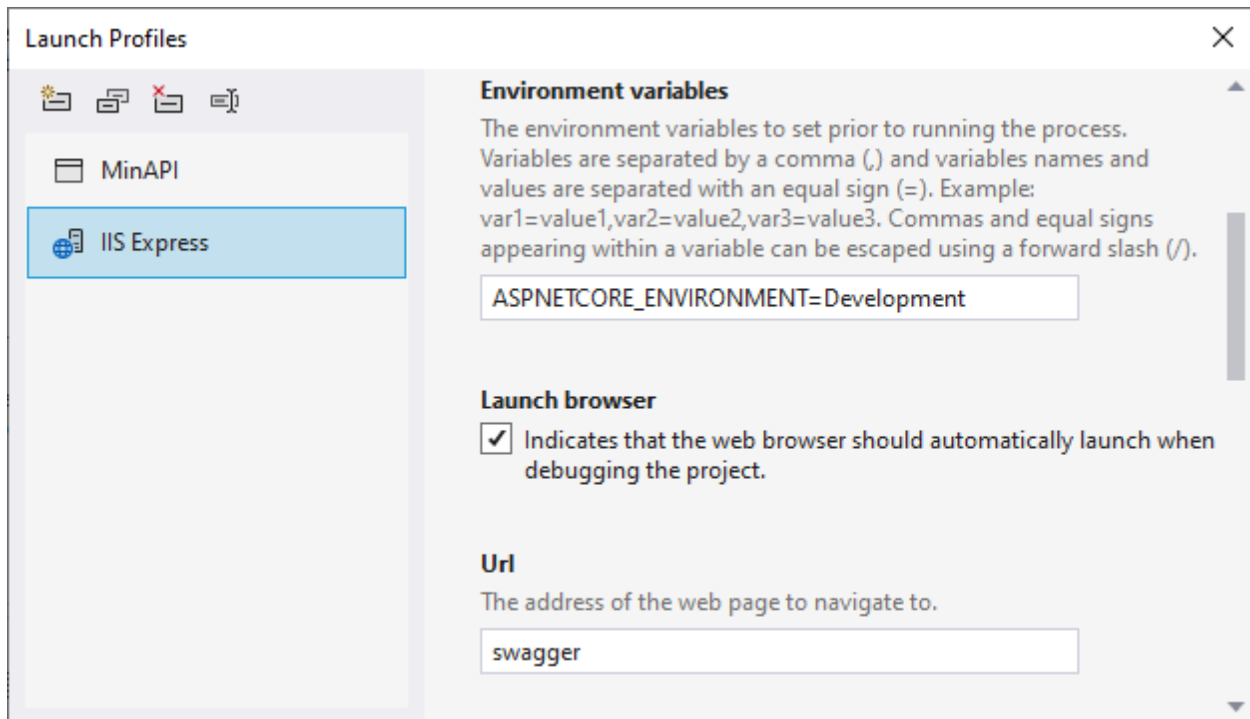
Figure 48: Select to Launch the Browser
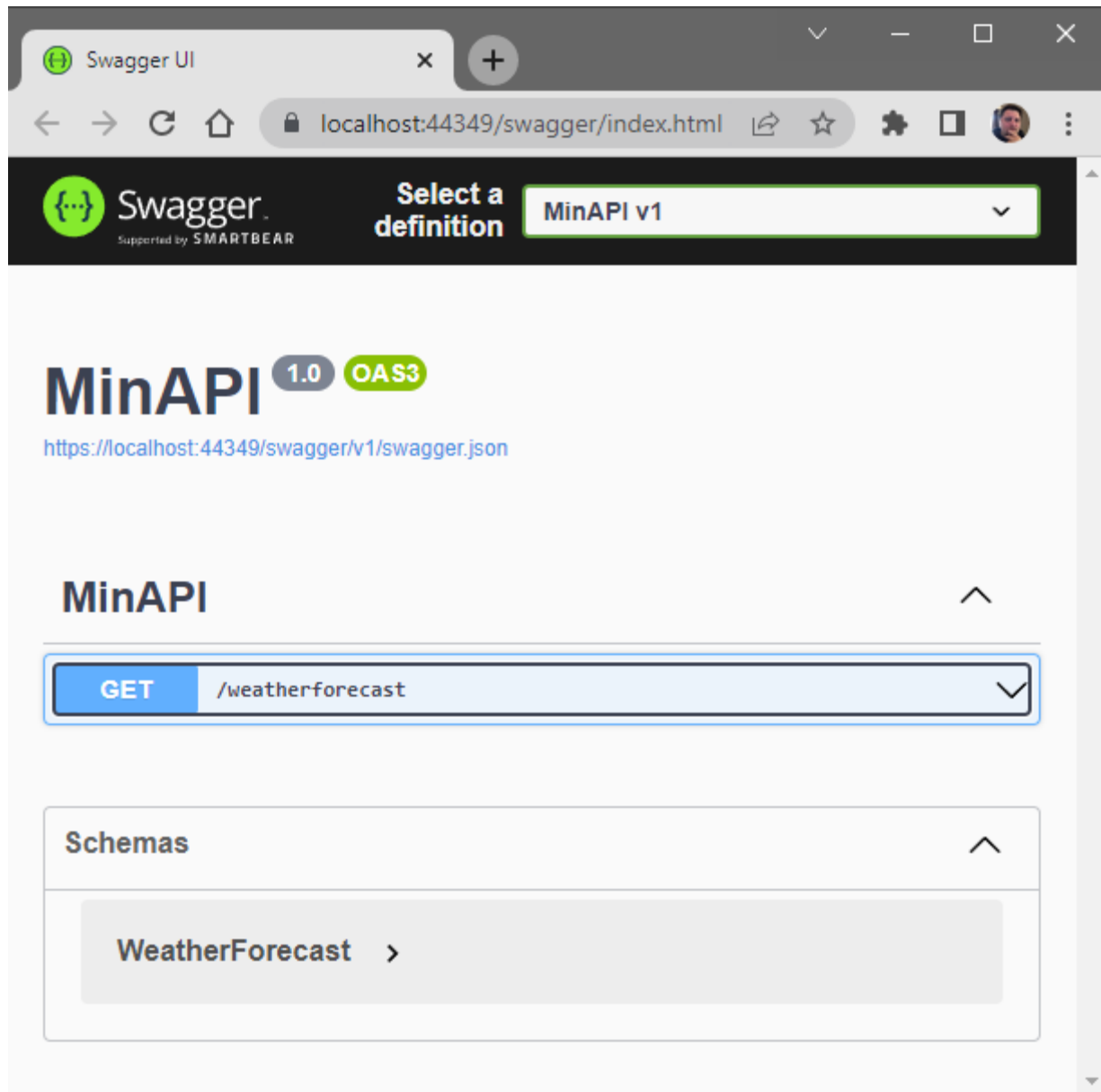
Figure 49: Viewing Your API in Swagger

Running your API again, you will see the browser launch and the Swagger page displayed for your weather forecast API. Under the **MinAPI** name at the top of our page, click the link **https://localhost:44349/swagger/v1/swagger.json** to view the Swagger JSON file. This file is shown in Code Listing 28.

The file displays the details of the minimal API based on the OpenAPI specification. You can see the details we set in the `WithName` method in Code Listing 27 show up in the `operationId` in the Swagger JSON file in Code Listing 28.

Other extension methods, such as `WithTags`, are available for use in your API to add metadata, so have a look at what there is on offer.

Code Listing 28: The Swagger JSON File

```json
{
  "openapi": "3.0.1",
  "info": {
    "title": "MinAPI",
    "version": "1.0"
  },
  "paths": {
    "/weatherforecast": {
      "get": {
        "tags": [
          "MinAPI"
        ],
        "operationId": "GetWeatherForecast",
        "responses": {
          "200": {
            "description": "Success",
            "content": {
              "application/json": {
                "schema": {
                  "type": "array",
                  "items": {
                    "$ref": "#/components/schemas/WeatherForecast"
                  }
                }
              }
            }
          }
        }
      }
    }
  },
  "components": {
    "schemas": {
      "WeatherForecast": {
        "type": "object",
        "properties": {
          "date": {
            "type": "string",
            "format": "date-time"
          },
          "temperatureC": {
            "type": "integer",
            "format": "int32"
          },
          "summary": {
            "type": "string",
            "nullable": true
```

```
        },
        "temperatureF": {
          "type": "integer",
          "format": "int32",
          "readOnly": true
        }
      },
      "additionalProperties": false
    }
  }
 }
}
```

All this was achieved without the need to write code for controllers or action methods. Minimal APIs work with a very minimalistic approach to structure and design. This is, however, still just a very basic API that only returns hardcoded data. We would ideally like to consume a service to provide this information to our API. Let's have a look at how to use services and dependency injection in the next section.

Implement dependency injection

I want to use a service to get the books from my LocalDB's BookRepo database. I can go ahead and create new projects to implement the core and data layers of my book repository, but it will be much easier to just reference the existing projects as set up in the BookRepository project that we migrated to .NET 6.0 back in Chapter 1.

To do this, right-click the **MinAPI** solution in Solution Explorer and click **Add** > **Existing Project** from the context menu, as shown in Figure 50.
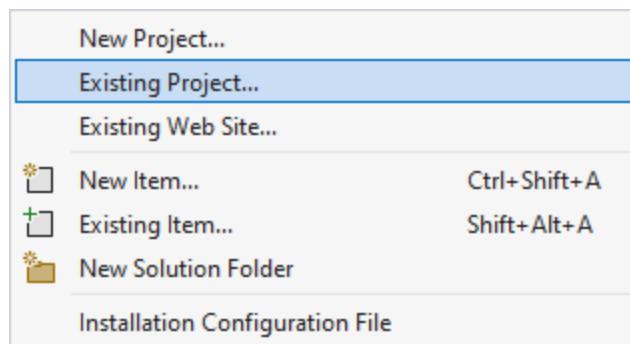
Figure 50: Add an Existing Project

Navigate to the location on your hard drive where you have saved your converted BookRepository project.

From here, add the **BookRepository.Core** and **BookRepository.Data** projects to your MinAPI project. When you have done this, your solution should look as illustrated in Figure 51.
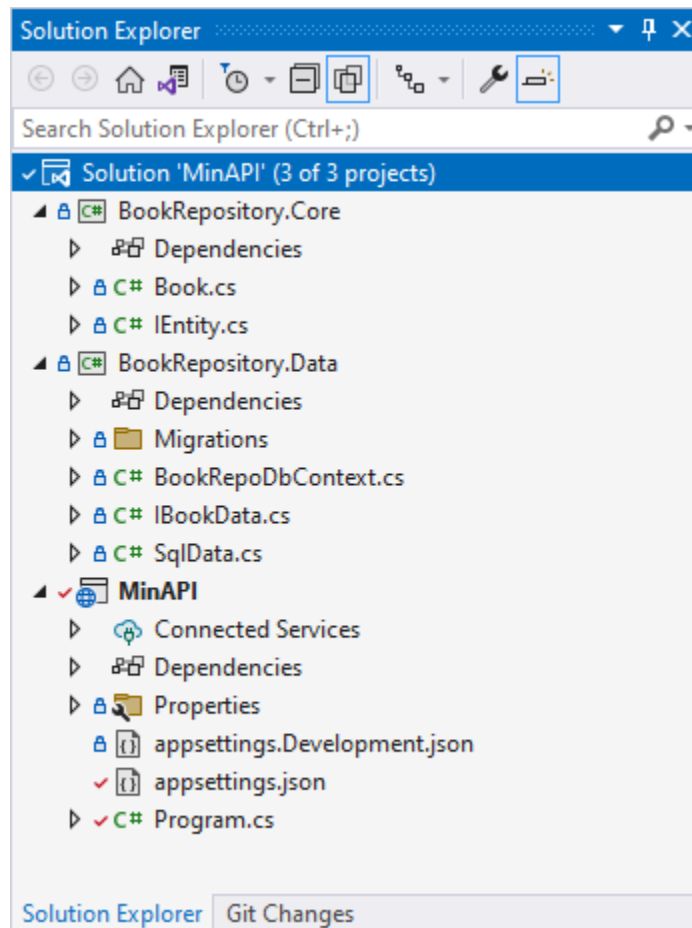
Figure 51: The Added BookRepository Projects

You can see that the MinAPI solution now contains the projects we upgraded in the BookRepository project in Chapter 1, but we can't use them without a little more work on our MinAPI project.

First, we need to add a reference to Entity Framework Core. Right-click the **MinAPI** project and click **Manage NuGet Packages** from the context menu. On the NuGet Package Manager screen, browse for and add the latest version of **Microsoft.EntityFrameworkCore** and **Microsoft.EntityFrameworkCore.Design** to your MinAPI project.

*📝 **Note: At the time of writing this book, version 6.0.6 is available, but if a newer version is available for you, be sure to add this version to your MinAPI project.***

The package references in your MinAPI solution should now look as illustrated in Figure 52.
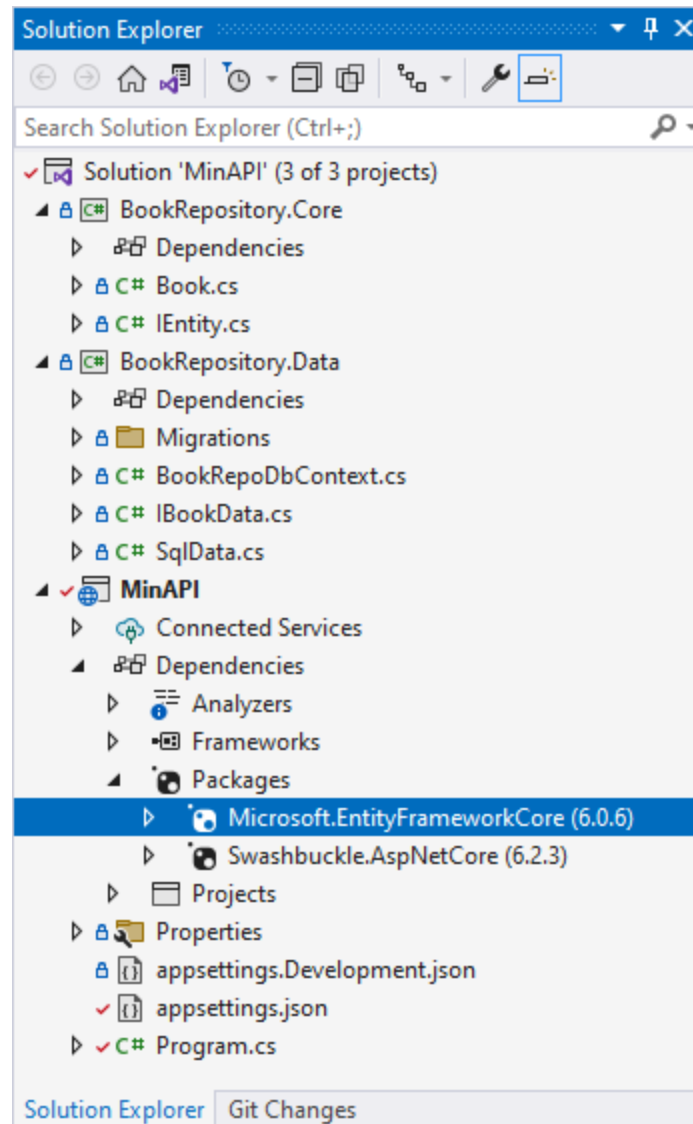
Figure 52: The Referenced EntityFrameworkCore NuGet Package

With this bit added, we have most of the components ready to use the book repository service to read the books in our database.

Open the **appsettings.json** file and add the connection string, as shown in Code Listing 29. This will be used by the service to connect to the `localdb` database.

```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "BookConn": "Data Source=(localdb)\\MSSQLLocalDB;Initial
Catalog=MinAPIBookRepo;Integrated Security=True"
  }
}
```

Now all that remains for us to do is add the service to our Program.cs file and consume it in a **MapGet** method via dependency injection. Then, before we can call the service, we need to run our database migrations to create the database.

Open the **Program.cs** file and register the data access service with the built-in dependency injection container. It is added as a scoped service in the services collection. In essence, we are telling our minimal API that whenever something in our project asks for **IBookData**, then it must provide the **SqlData** class.

We also need to tell the **DbContext** about our database connection. The **AddDbContextPool** specifies that **DbContext** pooling should be used, which allows for increased throughput because the instances of **DbContext** are reused, instead of having new instances created per request. You can see the code in Code Listing 30.

Code Listing 30: Adding the Scoped Service and DbContextPool

```csharp
builder.Services.AddScoped<IBookData, SqlData>();
builder.Services.AddDbContextPool<BookRepoDbContext>(dbContextOptns =>
{
    _ =
dbContextOptns.UseSqlServer(builder.Configuration.GetConnectionString("Book
Conn"));
});
```

Inject the service into a **MapGet** method as a parameter to our delegate. We can now use our **IBookData** service in our **MapGet** method as seen in Code Listing 31. We also need to specify the URL path of **/listbooks** and use the service to return the list of books from our data service. Lastly, add the endpoint name as **List Books**. This will now be the REST endpoint we will use to return a list of all books in our database.

```csharp
app.MapGet("/listbooks", async (IBookData service) =>
{
    return await service.ListBooksAsync();
})
.WithName("List Books");
```

The completed code for the Program.cs file is provided in Code Listing 32. If you come from the Program.cs and Startup.cs file structure before top-level statements were introduced, the code might feel slightly uncomfortable to write. It sure feels strange to me to be writing code like this using top-level statements. This, however, is not a better way of writing your code. It's just a different, more succinct way of writing your code that produces the same result as before top-level statements were introduced.

Code Listing 32: The Complete Program.cs File

```csharp
using BookRepository.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
// Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddScoped<IBookData, SqlData>();
builder.Services.AddDbContextPool<BookRepoDbContext>(dbContextOptns =>
{
    _ =
dbContextOptns.UseSqlServer(builder.Configuration.GetConnectionString("Book
Conn"));
});

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
```

```csharp
app.MapGet("/listbooks", async (IBookData service) =>
{
    return await service.ListBooksAsync();
})
.WithName("List Books");

var summaries = new[]
{
    "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy",
"Hot", "Sweltering", "Scorching"
};

app.MapGet("/weatherforecast", () =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateTime.Now.AddDays(index),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)]
        ))
        .ToArray();
    return forecast;
})
.WithName("GetWeatherForecast");

app.Run();

internal record WeatherForecast(DateTime Date, int TemperatureC, string?
Summary)
{
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```
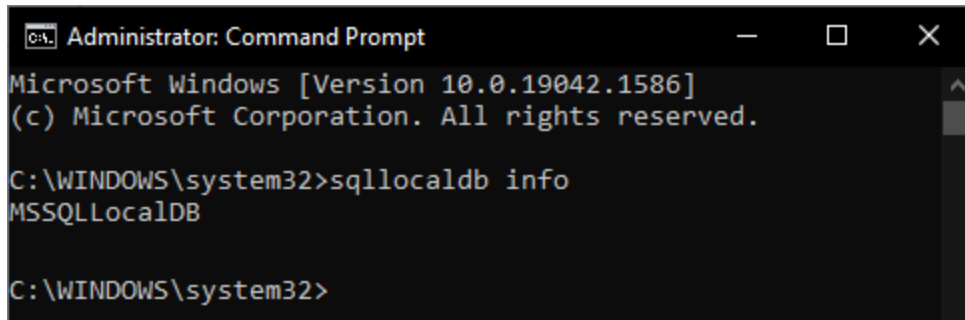
We're almost ready to call our service. If you do not already have a LocalDB instance containing the database, you need to create that before we can go any further. Luckily for us, we have our database migrations in the BookRepository.Data project.

LocalDB is usually installed when Visual Studio is installed. To check if LocalDB is installed, run **sqllocaldb info** from the command prompt, as shown in Figure 53.
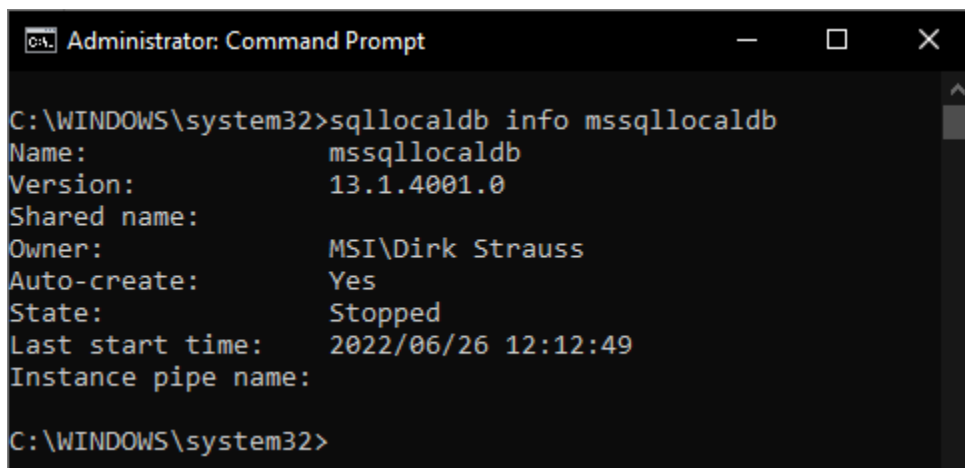
Figure 53: Check if LocalDB is Installed

You can then use `sqllocaldb info mssqllocaldb` in the command prompt to see more information about the **MSSQLLocalDB** instance (Figure 54).

Figure 54: Get More Information About MSSQLLocalDB

Next, you need to navigate to the **BookRepository.Data** directory in the command prompt.

📝 *Note: The next few steps will be dependent on where your projects exist in relation to each other. Bear this in mind when you run the commands in the following steps.*

The project locations for the projects used in this book might be different for you than for me. My projects are located under a single directory, and are as follows:

- BookRepositoryAPI
- BookRepositoryAPIv5_0
- DependencyInjection
- EssentialCSharp10
- IsolatedCSS
- MinAPI

The easiest way to get to the BookRepository.Data project is to right-click the project in your solution and select **Open Folder** in File Explorer from the context menu. Copy the directory path and change the directory to this location in the command prompt.

Then, run the following command seen in Code Listing 33, using the **-s** switch to specify the MinAPI startup project relative to the BookRepository.Data project path.
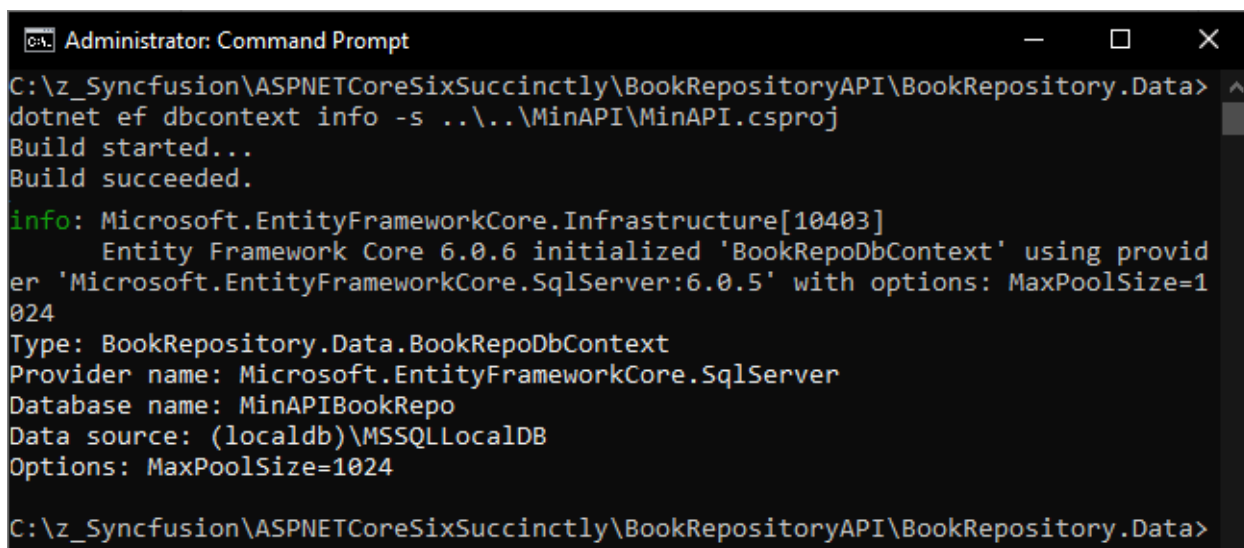
Code Listing 33: Get the DbContext Information

```
dotnet ef dbcontext info -s ..\..\MinAPI\MinAPI.csproj
```

The MinAPI project is located relative to my BookRepository.Data project at **..\..\MinAPI**.

💡 *Tip: If at any time you see the message that the Entity Framework tools version is older than that of the runtime, update the tools by typing* `dotnet tool update --global dotnet-ef` *in the Command Prompt.*

After running the command in Code Listing 33, you will see the output illustrated in Figure 55.

Figure 55: The Output from the Command in Code Listing 33

Pay special attention to the database name returned. It should be **MinAPIBookRepo**, as defined in the appsettings.json file in Code Listing 29. If you see this, you are ready to create your database by running the command in Code Listing 34 from the command prompt.

Code Listing 34: Create the MinAPIBookRepo Database

```
dotnet ef database update -s ..\..\MinAPI\MinAPI.csproj
```

After running the command, you will see the output in the command prompt, as illustrated in Code Listing 35.

Code Listing 35: The Output in the Command Prompt

```
Build started...
```

```
Build succeeded.
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.6 initialized 'BookRepoDbContext' using
provider 'Microsoft.EntityFrameworkCore.SqlServer:6.0.5' with options:
MaxPoolSize=1024
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (159ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
      CREATE DATABASE [MinAPIBookRepo];
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (62ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
      IF SERVERPROPERTY('EngineEdition') <> 5
      BEGIN
          ALTER DATABASE [MinAPIBookRepo] SET READ_COMMITTED_SNAPSHOT ON;
      END;
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (5ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      SELECT 1
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (20ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
          [MigrationId] nvarchar(150) NOT NULL,
          [ProductVersion] nvarchar(32) NOT NULL,
          CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (0ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      SELECT 1
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (12ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      SELECT [MigrationId], [ProductVersion]
      FROM [__EFMigrationsHistory]
      ORDER BY [MigrationId];
info: Microsoft.EntityFrameworkCore.Migrations[20402]
      Applying migration '20211107110645_Initial'.
Applying migration '20211107110645_Initial'.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      CREATE TABLE [Books] (
```

```
            [Id] int NOT NULL IDENTITY,
            [ISBN] nvarchar(max) NULL,
            [Title] nvarchar(max) NULL,
            [Description] nvarchar(max) NULL,
            [Publisher] nvarchar(max) NULL,
            CONSTRAINT [PK_Books] PRIMARY KEY ([Id])
        );
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
        Executed DbCommand (2ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
        INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
        VALUES (N'20211107110645_Initial', N'6.0.6');
Applying migration '20211107125009_BookModelUpdate1'.
info: Microsoft.EntityFrameworkCore.Migrations[20402]
        Applying migration '20211107125009_BookModelUpdate1'.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
        Executed DbCommand (2ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
        ALTER TABLE [Books] ADD [Author] nvarchar(max) NULL;
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
        Executed DbCommand (0ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
        INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
        VALUES (N'20211107125009_BookModelUpdate1', N'6.0.6');
Done.
```

You will notice that a few things have happened here. The database **MinAPIBookRepo** was created using the **CREATE DATABASE** command. A database table called **Books** was also created. This table will match our **Book** entity located in the BookRepository.Core project.

Lastly, you will need to create some book data in your database. You can view the database in Visual Studio by going to the **View** menu and clicking the **SQL Server Object Explorer** menu. Expanding the **Databases** folder under **MSSQLLocalDB**, you will see the created MinAPIBookRepo database. You can then expand the **Tables** folder, right-click the **Books** table, and select **View Data** from the context menu. Here you can manually add some book data to the table.

For your convenience, I have added an **INSERT** script to the GitHub repository called MinAPIBookRepoData.sql that contains some book data. Open this script and copy the **INSERT** statements. Then right-click the **MinAPIBookRepo** database and click **New Query** from the context menu. Paste the copied **INSERT** script into the query window and run the script by clicking **Execute** or pressing **Ctrl**+**Shift**+**E**.

> *Note: Phew, this was a lot of work to get here. Notice though that most of the work was setting up the data service and creating the database. Implementing dependency injection only comprised the code in Code Listings 30 and 31.*

Run your API, and when your SwaggerUI loads, you will see the **/listbooks** endpoint listed along with the **/weatherforecast** endpoint (Figure 56).
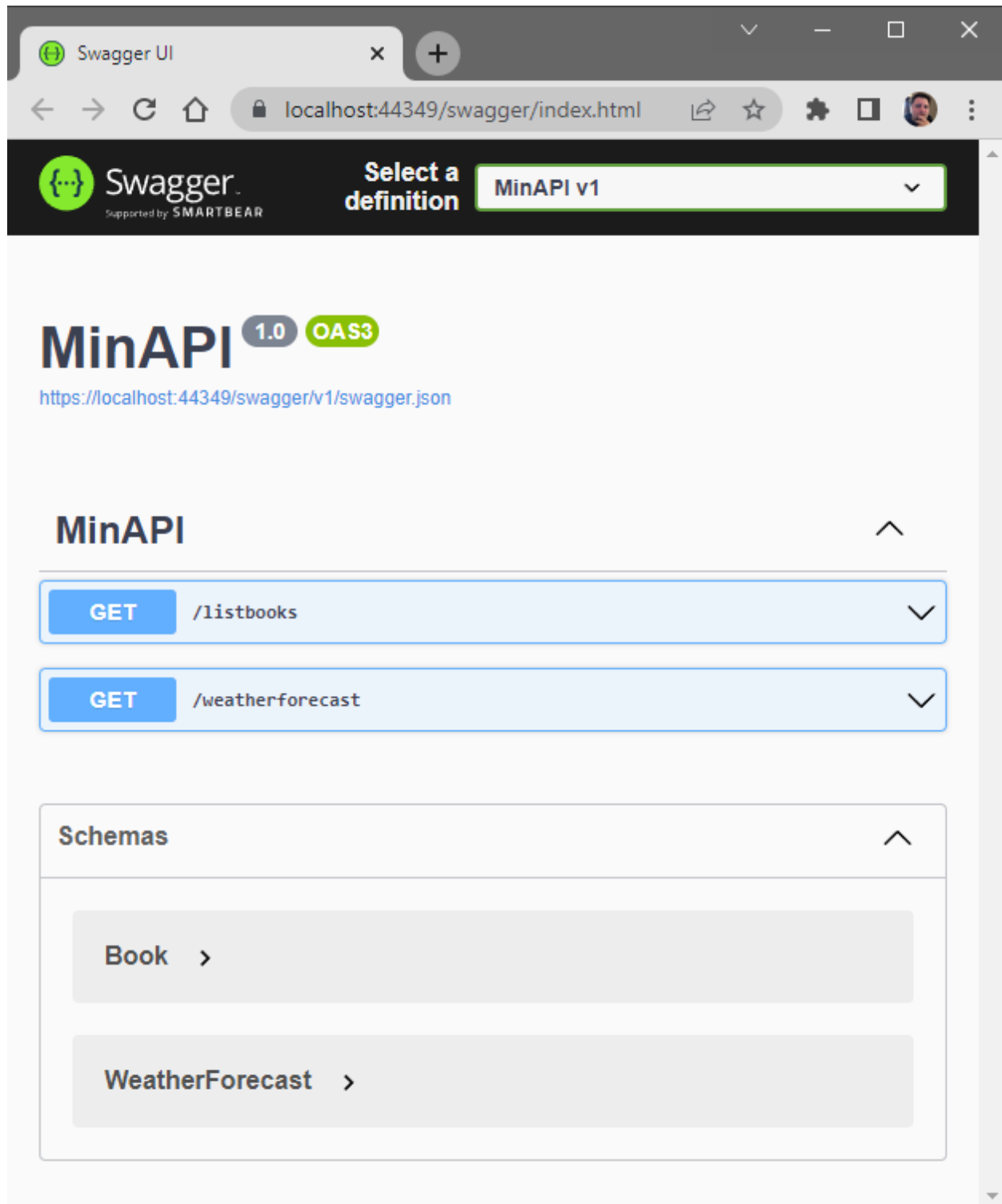
Figure 56: The ListBooks and WeatherForecast Endpoints in Swagger

Execute the **/listbooks** endpoint in Swagger, and you will see the results as illustrated in Figure 57.
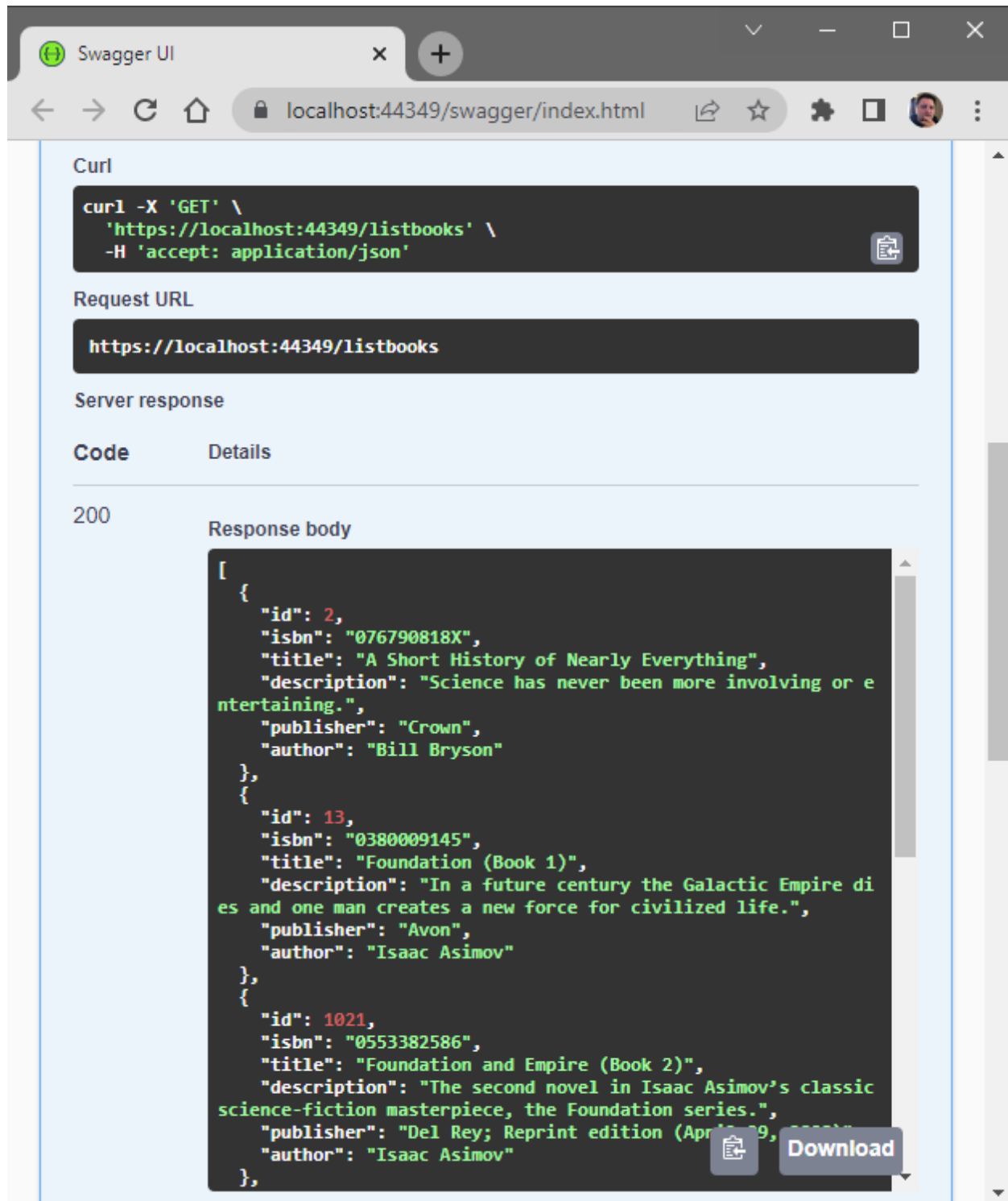
Figure 57: The ListBooks Response in Swagger

As always, I'm a fan of Postman, and while your API is running, you can perform a **GET** request in Postman by entering the URL **https://localhost:44349/listbooks** and clicking **Send**. Just note that your port might be different from mine.
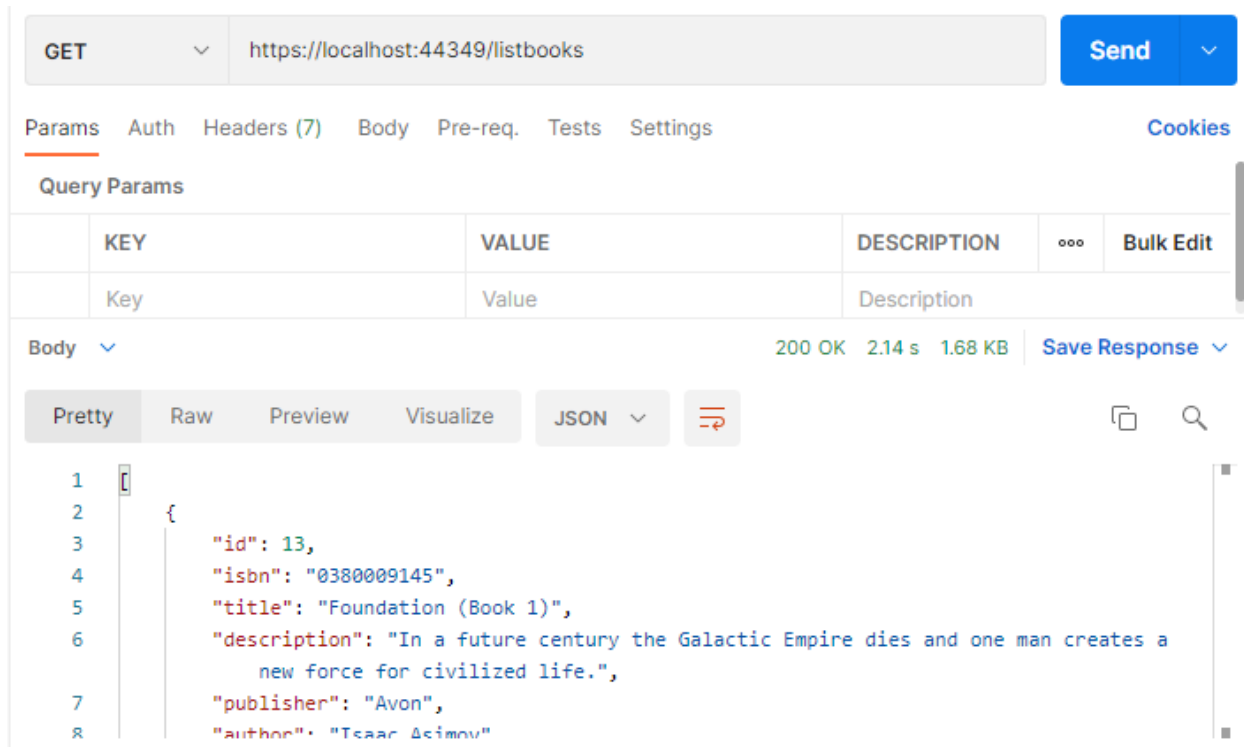
Figure 58: The ListBooks Response in Postman

With the book data returned from our book repository using our minimal API, we have a fully functioning **REST** endpoint that returns books stored in our LocalDB database. Hooking up the data service was more work than actually implementing the dependency injection of the data service. The next logical thing to do is to be able to add, update, and delete books from our book repo database. Let's have a look at how to implement these next.

Implement CRUD operations

Being able to add, edit, and delete books in our book repository is essential for completing the minimal API service. It is this functionality that will allow us to maintain our book repository and keep it up to date.

Implementing the **async** methods in our endpoints is quite straightforward. Let's take the endpoints one by one and make each one functional, starting with adding a new book to the repository.

Code Listing 36: The MapPost Endpoint to Add a Book

```
app.MapPost("/book", async (Book book, IBookData service) =>
```

```
{
    _ = await service.SaveAsync(book);
})
.WithName("Add Book");
```

As shown in Code Listing 36, `app.MapPost` will use `/book` as the URL path for the incoming HTTP verb `POST`. It also takes a parameter of `Book` that contains the book data from the `POST` request body that .NET has mapped to the `Book` object. `app.MapPost` also takes the `IBookData service`, which is required to call the `SaveAsync` method to save the book data bound to the `Book` object via model binding.

Run the application and have a look at the Swagger UI that launches. It will show the new `POST` endpoint as shown in Figure 59.
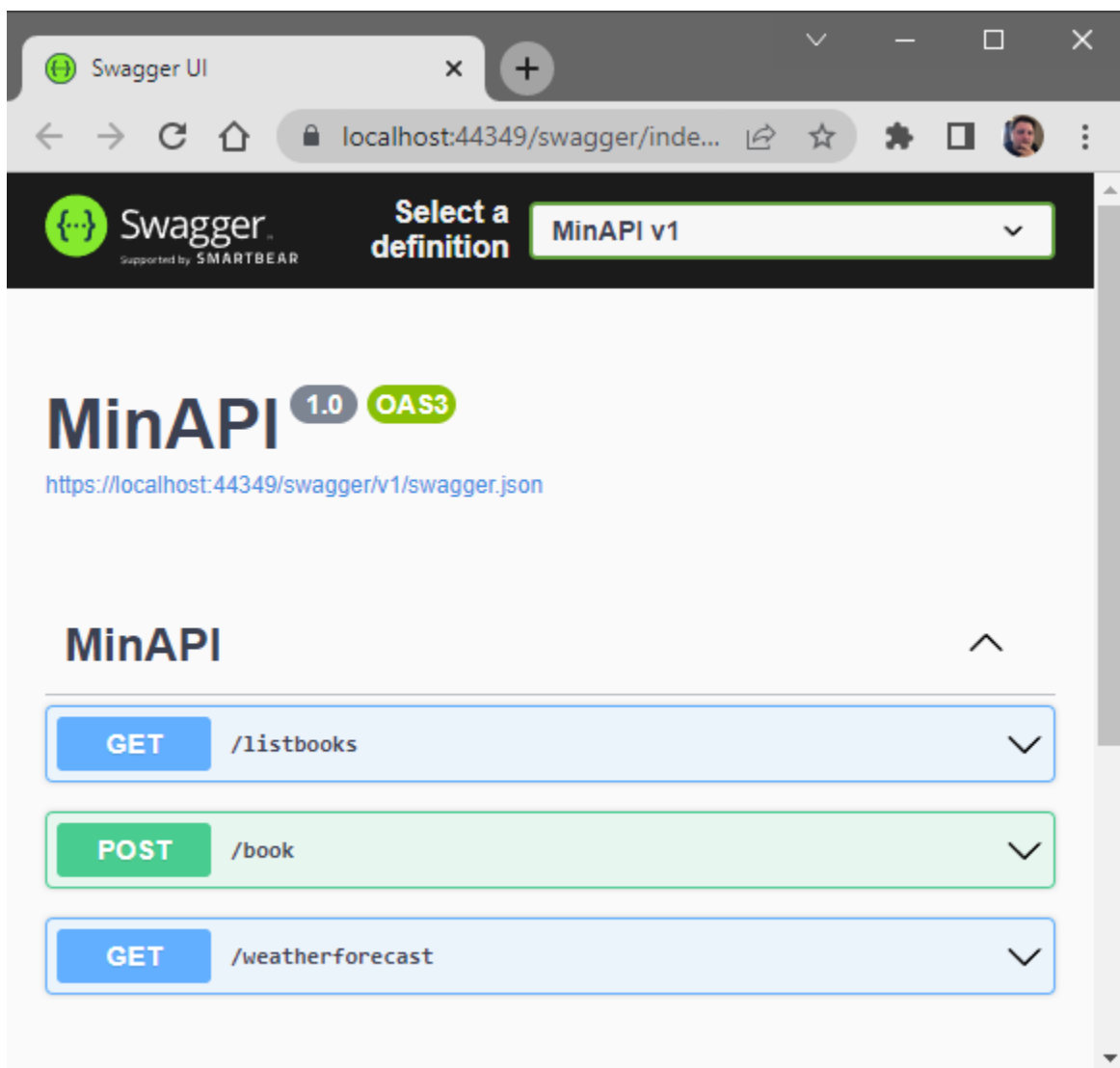
Figure 59: The New POST Endpoint Listed in Swagger UI

Expand the **POST** endpoint and you will see a sample request and expected responses, as shown in Figure 60.
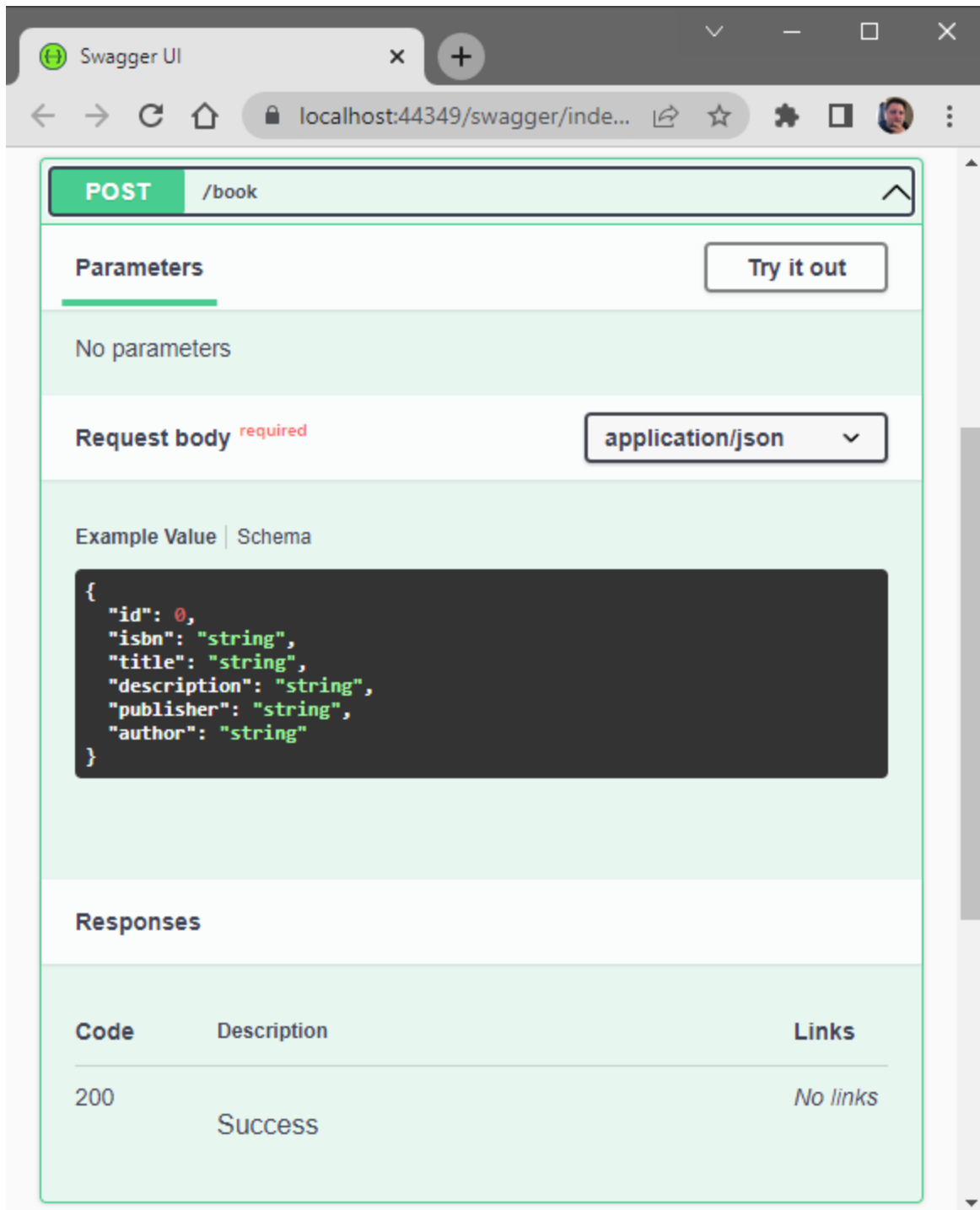
Figure 60: Example POST Request Listed in Swagger UI

Click **Try it out**, add the code in Code Listing 37, and then click **Execute**.

Code Listing 37: Adding a Dummy Book

```json
{
    "isbn": "test777",
    "title": "test777",
    "description": "test777",
    "publisher": "test777",
    "author": "test777"
}
```

The code in Code Listing 37 is just dummy data, but we want to see the book created with our minimal API. After creating a book, call the **/listbooks** endpoint, and you will see the book that you just created returned from the **GET** request.

Note: Make a note of the ID returned here. We will be using it in our PUT request later in the chapter.

The book data is somewhat silly, so we need a way to change this data.

Code Listing 38: Updating the Dummy Book

```csharp
app.MapPut("/updatebook", async (Book book, IBookData service) =>
{
    _ = await service.UpdateAsync(book);
})
.WithName("Update Book");
```

The next endpoint that we want to add is the **PUT**. This will allow us to update a book in our book repository. The code is almost identical to the **POST** in Code Listing 36. The only difference here is that the endpoint URL has changed to **/updatebook**, and the **WithName** method specifies **Update Book**.
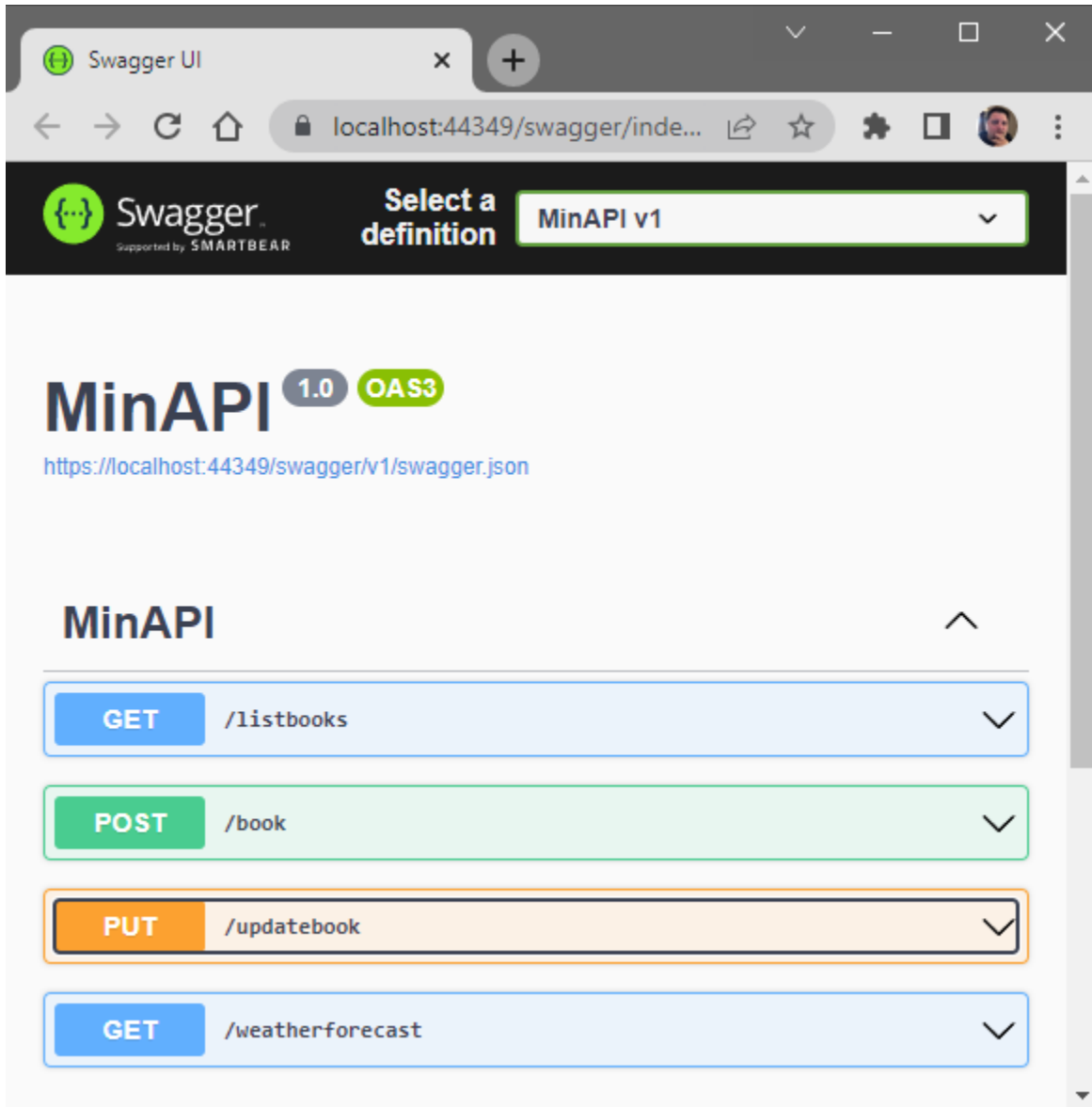
Figure 61: The PUT Endpoint Listed in Swagger UI

Running the API again, you will see the newly created **PUT** endpoint listed, as seen in Figure 61.
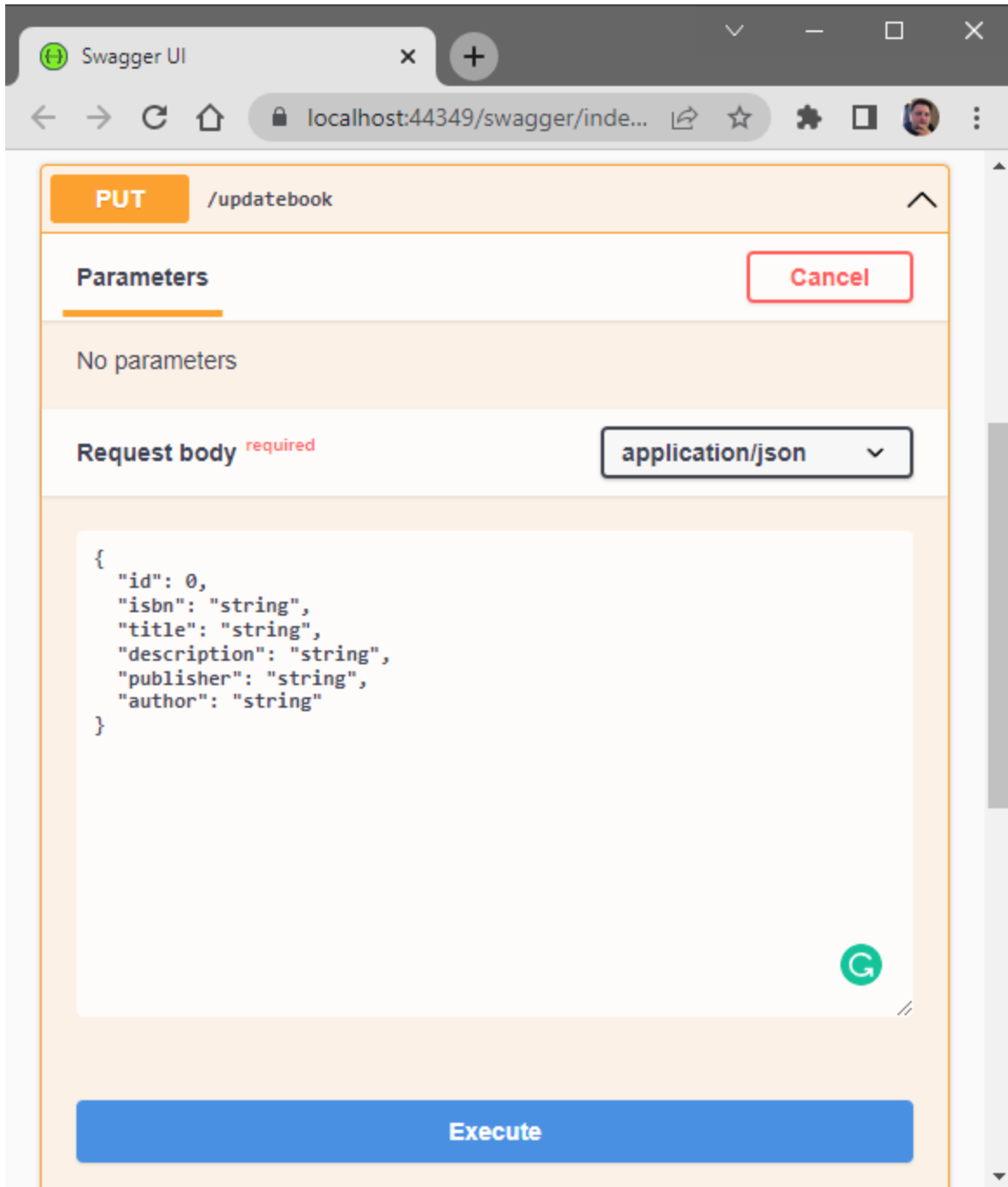
Figure 62: Sample Request Body for the PUT

Expanding the **PUT** endpoint will allow you to see a sample request body. What I want to do though, is change the book we added earlier using the **POST** in Code Listing 37. The book that I added earlier was created with an ID of **1024**. The ID of the book that you added will be different than mine. This will be the ID you noted earlier. Use this ID to perform the **PUT** request and change some of the data for the book, as seen in Code Listing 39.

```json
{
    "id": 1024,
    "isbn": "test888",
    "title": "test888",
    "description": "test888",
    "publisher": "test888",
    "author": "test888"
}
```

Calling the **GET** endpoint after executing the **PUT** will return the modified book data.

Code Listing 40: Adding the MapDelete Endpoint

```csharp
app.MapDelete("/deletebook", async ([FromBody]Book book, IBookData service)
=>
{
    _ = await service.DeleteAsync(book);
})
.WithName("Delete Book");
```

Adding the **/deletebook** endpoint is, again, similar to the **/updatebook** endpoint. Only this time, you need to add **[FromBody]** to tell .NET that the parameter should be bound using the request body.
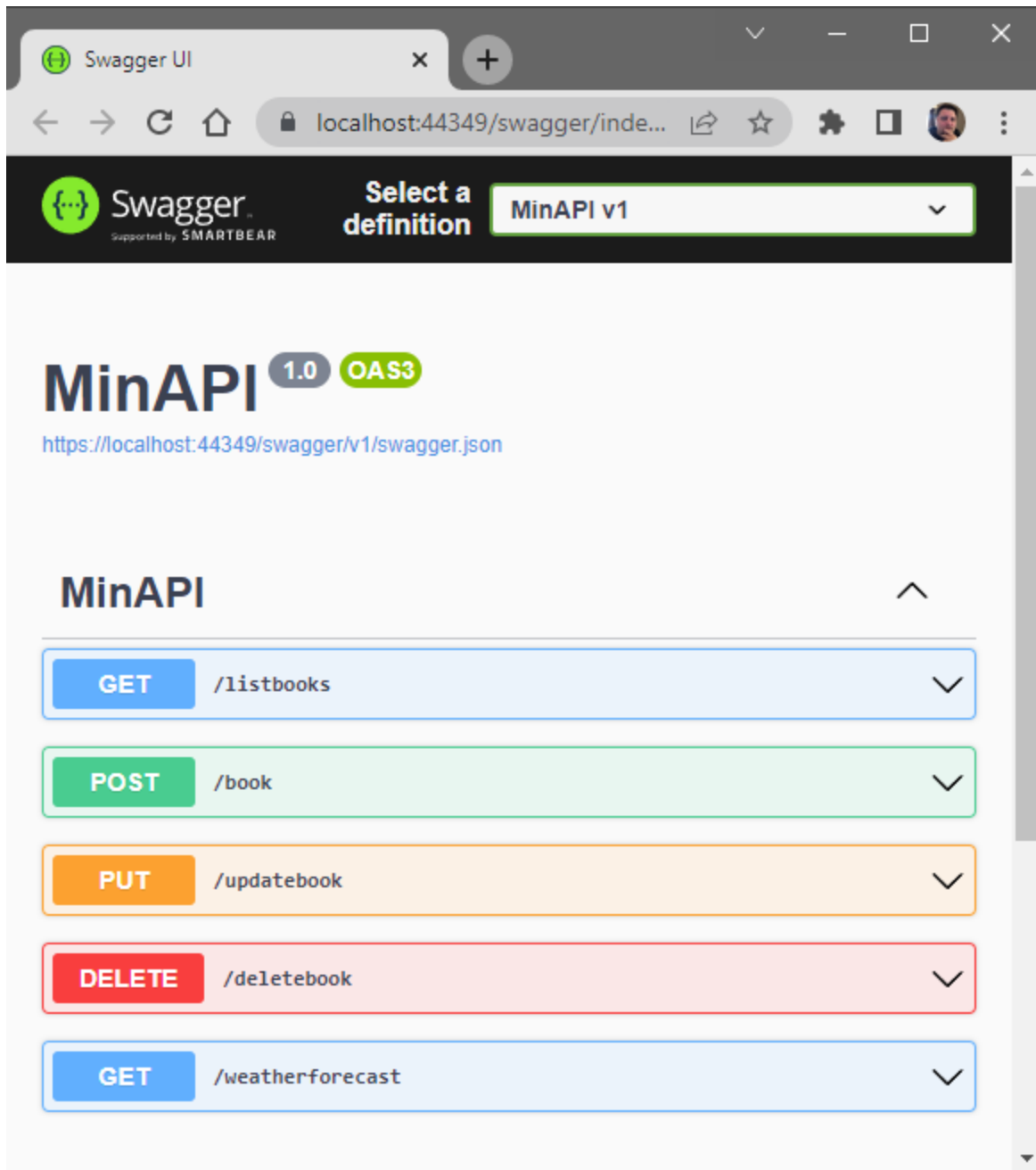
Figure 63: The Delete Endpoint in Swagger UI

Running your minimal API will display the **/deletebook** endpoint in Swagger UI, as seen in Figure 63.
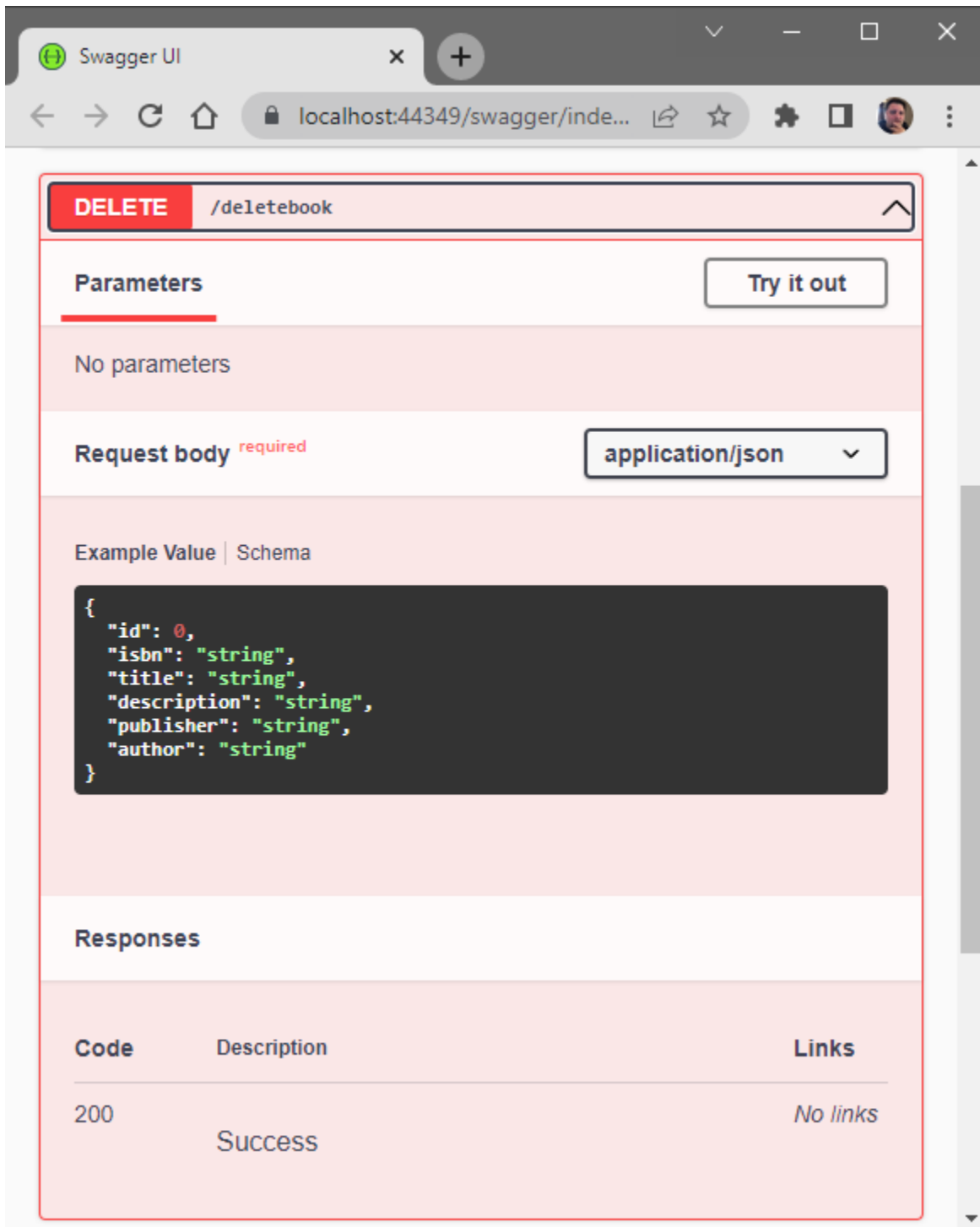
Figure 64: The Sample Request for the Delete

By expanding the endpoint and clicking **Try it out**, you can pass the same code as in Code Listing 39 to the **/deletebook** endpoint to delete the dummy book we added earlier. As before, your ID will differ from mine, and you should use the ID you noted earlier.

In reality, while this delete works, I would ideally like to pass the **/deletebook** endpoint just the ID of the book to delete. This will make it easier when calling the API. I will leave it up to you to figure this out, but I won't leave you high and dry.

Code Listing 41: Delete a Book by ID

```
app.MapDelete("/deletebook/{id}", async (int id, IBookData service) =>
{
    var result = await service.DeleteAsync(id);
    return result ? Results.Ok() : Results.NotFound();
})
.WithName("Delete Book by ID");
```

The code in Code Listing 41 is the completed **MapDelete** endpoint that accepts a book ID as a parameter. You need to complete the following:

- Modify the **IBookData** interface to tell it that any implementing classes need to contain a **DeleteAsync** method that accepts a book ID as a parameter.
- Add the implementation to the **SqlData** class to find the book entity and delete it. Also, handle errors when the supplied book ID is not found.

The complete code is on GitHub, so if you would like to see the solution, please have a look there.

Improving endpoint responses

Something that has been added to the **MapDelete** endpoint, but not any of the others, is improved responses. Ideally, when an endpoint is called, we would like the responses to tell the user what happened. Let's use the current endpoint for adding a new book, as shown in Code Listing 42.

Code Listing 42: The Current Add Book Endpoint

```
app.MapPost("/book", async (Book book, IBookData service) =>
{
    _ = await service.SaveAsync(book);
})
.WithName("Add Book");
```

If we had to add a book, the Swagger UI would display the response shown in Figure 65.
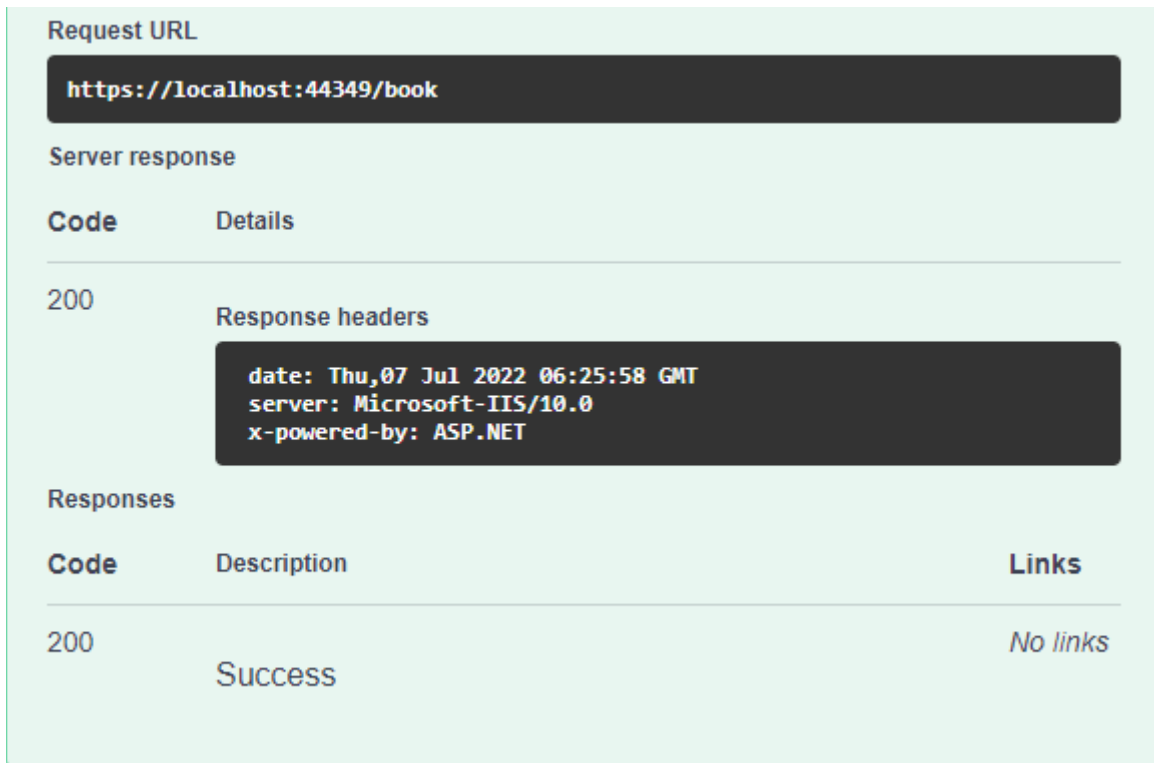
Figure 65: The Current Add Book Response

What we should be doing, though, is returning a status code for the action that took place. In the case of adding a book, we want to return a status **201** for created.

Code Listing 43: The Improved Add Book Endpoint

```
app.MapPost("/book", async (Book book, IBookData service) =>
{
    var createdBookId = await service.SaveAsync(book);
    return Results.Created($"/book/{createdBookId}", book);
})
.WithName("Add Book");
```

The modification is simple, as seen in Code Listing 43. The created book ID is returned from the **SaveAcync** method, and **Results.Created** is used to display the result to the user. Adding another book to the book repository will result in a nice **201** response, as shown in Figure 66.

Figure 66: The Improved Add Book Response

The endpoint used to update a book can also be improved. With the current endpoint, if I modify the dummy book with ID **1029** added earlier, I will receive a very generic response. The code for my modified book is illustrated in Code Listing 44.

As before, the ID of **1029** is probably not going to exist in your book repo, so be sure to update a book returned when calling your **/listbooks** endpoint.

Code Listing 44: The Modified Book with ID 1029

```
{
    "id": 1029,
    "isbn": "1455502782",
    "title": "Arguably: Essays by Christopher Hitchens",
```

```
    "description": "For nearly four decades, Hitchens has been telling us,
in pitch-perfect prose, what we confront when we grapple with first
principles-the principles of reason and tolerance and skepticism that
define and inform the foundations of our civilization-principles that, to
endure, must be defended anew by every generation.",
    "publisher": "Twelve",
    "author": "Christopher Hitchens"
}
```

Calling the **/updatebook** endpoint will return the response shown in Figure 67.

Figure 67: The Current /updatebook Endpoint Response

This endpoint response can also be improved, as illustrated in Code Listing 45. Here we are simply saying after the **PUT**, return **NoContent**, which is a **204** status code.

Code Listing 45: The Modified /updatebook Endpoint

```
app.MapPut("/updatebook", async (Book book, IBookData service) =>
{
    _ = await service.UpdateAsync(book);
    return Results.NoContent();
})
.WithName("Update Book");
```

Modifying a book with these code changes in place will result in the **204** being returned, as seen in Figure 68.

Figure 68: The NoContent 204 Response

The decision of which response to return here will likely differ from developer to developer. I like to see the resource that was modified being returned in the response, but that is just my preference. It is perfectly acceptable to also return **Results.Ok** from a **PUT**. I will leave this up to you to experiment with.

Calling an external API

Writing and consuming APIs is something every developer will do during their career. It is essential for interconnected systems. Even though people don't realize it, somewhere during their daily routine they are touching the results of an API call. Whether through a bank transaction, browsing social media, or calling an Uber, APIs make connection and integration convenient and easy.

It is, therefore, conceivable for APIs to call other APIs—that's what we'll be doing in this section. Our minimal API contains a **/weatherforecast** endpoint that just returns some dummy data. If we wanted to provide this functionality from our book repository, we would need to return real weather data.

Luckily for us, there are more weather APIs out there than you can shake a stick at, and setting up your minimal API to consume an external API is simple. In this example, I will be consuming the AfriGIS Weather API. You will find information about this API [on the AfriGIS website](#).

To talk to an external API, we need to use ASP.NET's HTTP client. At the top of our **Program.cs** file, we need to register an HTTP Client Factory as part of our setup code. The line we need to add is **builder.Services.AddHttpClient();**. Once it's added, your code should look as illustrated in Code Listing 46.

Code Listing 46: Adding the HTTP Client Factory

```csharp
using BookRepository.Core;
using BookRepository.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
// Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddScoped<IBookData, SqlData>();
builder.Services.AddDbContextPool<BookRepoDbContext>(dbContextOptns =>
{
    _ =
dbContextOptns.UseSqlServer(builder.Configuration.GetConnectionString("Book
Conn"));
});

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddHttpClient(); // Add HTTP Client Factory.

var app = builder.Build();
```

The next thing I need to do is create a **WeatherResults** entity. This will allow me to specify the target type to deserialize to when calling the weather API from my minimal API. Add the **WeatherResults** class to your **BookRepository.Core** project, as shown in Figure 69.
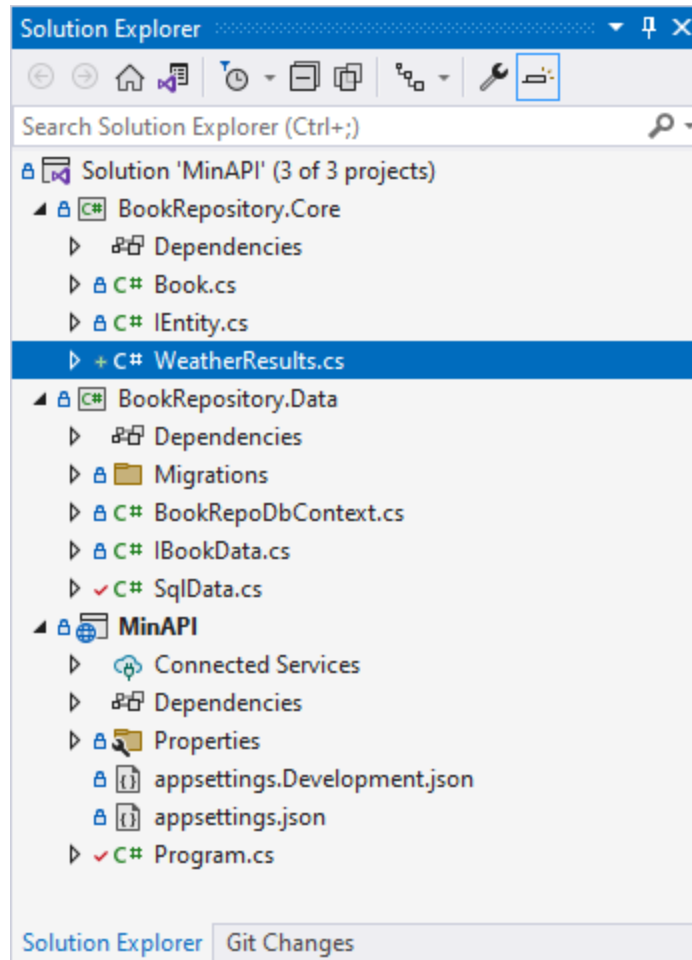
Figure 69: The Added WeatherResults Class

You can see the complete code for the **WeatherResults** class in Code Listing 47.

Code Listing 47: The WeatherResults Entity

```csharp
using System;

namespace BookRepository.Core
{
    public class WeatherResults
    {
        public int code { get; set; }
        public string source { get; set; }
        public Result[] result { get; set; }
    }

    public class Result
    {
        public Station_Details station_details { get; set; }
        public Station_Readings[] station_readings { get; set; }
```

```
    }

    public class Station_Details
    {
        public float latitude { get; set; }
        public float longitude { get; set; }
        public int distance { get; set; }
        public int synop_no { get; set; }
        public string station_name { get; set; }
    }

    public class Station_Readings
    {
        public DateTime datetime { get; set; }
        public float temperature { get; set; }
        public int humidity { get; set; }
        public float pressure { get; set; }
        public int wind_direction { get; set; }
        public float wind_speed { get; set; }
        public float last_hours_rainfall { get; set; }
    }
}
```

This will allow the JSON data that is returned from the AfriGIS Weather API to be deserialized into an object that I can use.

💡*Tip: To quickly create a class from JSON data, for example, copy the JSON you want to deserialize. Then in Visual Studio, go to the* Edit *menu and select* Paste Special > *JSON as Classes*. *This will create a class for you that is inferred from the JSON data you copied. You can do the same for copied XML. Easy peasy lemon squeezy.*

We can now start creating an endpoint that will call out to the AfriGIS Weather API in an asynchronous way. You can find the complete code in Code Listing 48.

Code Listing 48: The AfriGIS Weather API Call

```
app.MapGet("/sa-weather", async (IHttpClientFactory factory) =>
{
    var client = factory.CreateClient();
    var baseUrl = "https://saas.afrigis.co.za/rest/2";
    var endP = "weather.measurements.getByCoord/myapisamples";
    var auth = "bBFMNngfUSqQ80kFWUwmihszdPs";
    var latlong = "-25.808589,28.255833";
    var range = 10000;
    var count = 3;
```

```
    var response = await
client.GetFromJsonAsync<WeatherResults>($"{baseUrl}/{endP}/{auth}/?location
={latlong}&location_buffer={range}&station_count={count}");

    return Results.Ok(response);
})
.WithName("Get SA Weather");
```

We start with adding **app.MapGet**, and give it a route of **/sa-weather**. We then provide an **async** lambda that takes an **IHttpClientFactory** called **factory**. Using an **async** method here is a good idea, especially when making an external API call.

Inside the delegate, we have to create an instance of the client by calling **factory.CreateClient**. This creates a new **HttpClient** using the default configuration. The rest is just the mechanics to construct the external API call, and these settings, such as the base URL, authentication key, and so on, should be in a configuration file. For ease of use here, I've just added it directly to the delegate.

Lastly, we do an **async** call to the AfriGIS Weather API using our **WeatherResults** as a type parameter, and use the constructed URL to call the endpoint. We then just respond with **Results.Ok**, returning the response from the AfriGIS Weather API.

When you run your API, you will see that the Swagger UI lists the **/sa-weather** endpoint, as shown in Figure 70. Execute the **GET** in Swagger, and the JSON illustrated in Code Listing 49 will be returned.

Code Listing 49: The Data Returned from AfriGIS

```
{
  "code": 200,
  "source": "weather.measurements.api",
  "result": [
    {
      "station_details": {
        "latitude": -25.8277,
        "longitude": 28.2235,
        "distance": 3872,
        "synop_no": 68264,
        "station_name": "WATERKLOOF AIR FORCE BASE"
      },
      "station_readings": [
        {
          "datetime": "2022-07-09T07:00:00Z",
          "temperature": 13.3,
          "humidity": 67,
          "pressure": 861.4,
          "wind_direction": 26,
          "wind_speed": 0.9,
```

```
          "last_hours_rainfall": 0
        }
      ]
    },
    {
      "station_details": {
        "latitude": -25.752,
        "longitude": 28.2585,
        "distance": 6275,
        "synop_no": 68260,
        "station_name": "PRETORIA UNIVERSITY PROEFPLAAS"
      },
      "station_readings": [
        {
          "datetime": "2022-07-09T07:00:00Z",
          "temperature": 13.5,
          "humidity": 63,
          "pressure": 874.1,
          "wind_direction": 69,
          "wind_speed": 0.9,
          "last_hours_rainfall": 0
        }
      ]
    },
    {
      "station_details": {
        "latitude": -25.7663,
        "longitude": 28.2005,
        "distance": 7263,
        "synop_no": 68269,
        "station_name": "PRETORIA UNISA"
      },
      "station_readings": [
        {
          "datetime": "2022-07-09T07:00:00Z",
          "temperature": 10.7,
          "humidity": 81,
          "pressure": 866.6,
          "wind_direction": 0,
          "wind_speed": 1.1,
          "last_hours_rainfall": 0
        }
      ]
    }
  ]
}
```

As you can see, calling external APIs is simple and easy.
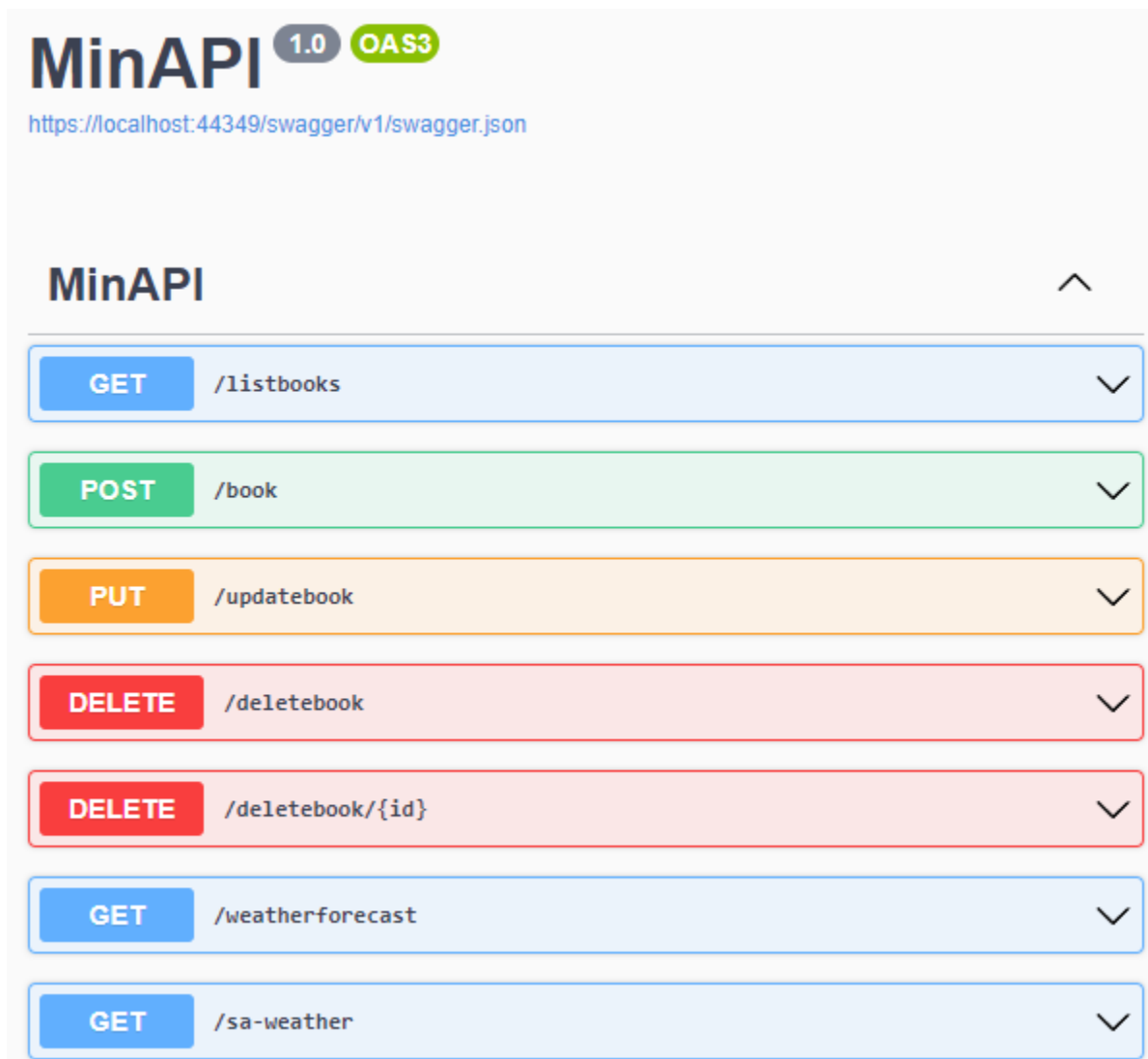
Figure 70: The AfriGIS Weather API Call in Swagger UI

Apart from the minor limitations with minimal APIs, they provide an alternative to traditional web APIs. Minimal APIs use top-level statements, a single `Program.cs` file, and implicit usings to minimize the boilerplate code required.

The `MapGet`, `MapDelete`, `MapPut`, and `MapPost` helper methods allow binding incoming requests to handler methods for common verb types. Supporting basic parameter binding from the URL, minimal APIs also support binding from the request body when using JSON.

Minimal APIs also support middleware pipelines such as cross-origin requests, logging, and authentication. Minimal APIs allow us to provide results for API calls that help to improve the developer experience. Last, but not least, dependency injection is also supported, allowing us to inject services into the handler methods. In the next chapter, we will take a closer look at dependency injection in ASP.NET Core 6.

Chapter 4 Why Use Dependency Injection?

Dependency injection (or DI) is a concept that you will come across sooner or later, if you haven't already. This design pattern supports the development of loosely coupled code. Let's have a look at how dependency injection works and why it is so beneficial.

Note: Briefly, dependency injection allows a class to accept dependencies at runtime rather than instantiating dependencies at compile time.

What is dependency injection?

Dependency injection supports the implementation of two concepts: inversion of control and the dependency inversion principle. The ASP.NET Core framework relies heavily upon dependency injection, as it is a foundational concept of ASP.NET Core. It is therefore recommended that developers make use of it when creating solutions with ASP.NET Core.

To illustrate the architectural problem that dependency injection solves, let's have a look at an application that uses the minimal API we created earlier to get the weather details from AfriGIS's Weather API. The application is a simple web application that displays the data returned from the API call as shown in Figure 71.

Note: The previous chapter dealt with the creation of the minimal API to call an external API to return weather data. The API must be deployed or running in Visual Studio if you want to test against the service in this example. You can, however, just create a dummy service that returns static data instead of doing a call to the minimal API. Also note that the example is simply a way to illustrate dependency injection, and not an example of how to correctly call an external web service (and all the error handling that goes with it).

You can follow along in the code found on GitHub, or you can create your own weather service manager to return dummy data. The real point here is to illustrate how dependency injection solves the problem we have with a tightly coupled dependency.
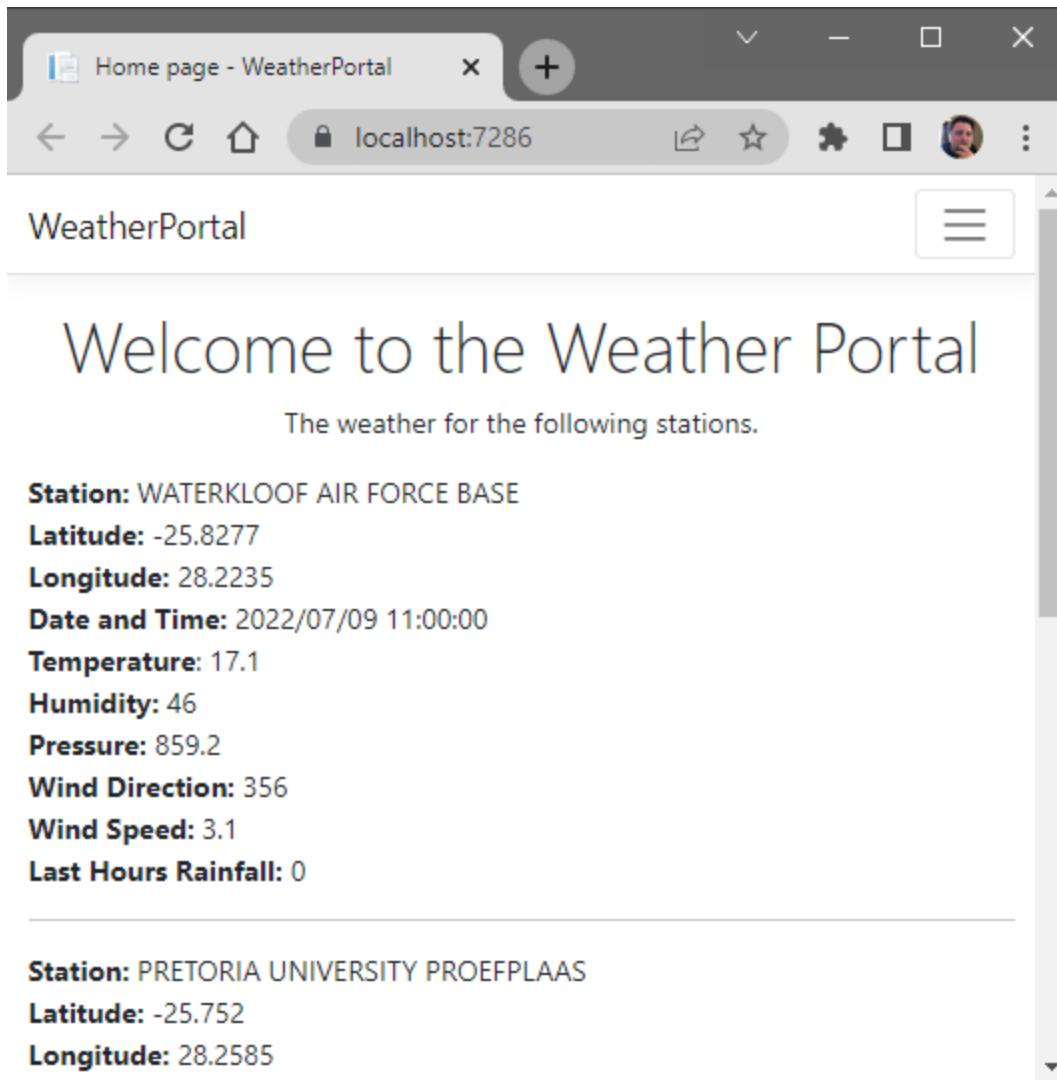
Figure 71: The Weather Portal Web Application

The Razor page model called Index.cshtml.cs is shown in Code Listing 50. The **OnGetAsync** method will populate the model that, in turn, is used to create the index page's content.

Code Listing 50: The Razor Page for the Weather Portal

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using WeatherPortal.Core;

namespace WeatherPortal.Pages
{
    public class IndexModel : PageModel
    {
        public WeatherResults WeatherResults { get; set; } = new
WeatherResults();
```

```
        public IndexModel()
        {

        }

        public async Task OnGetAsync()
        {
            var weatherManager = new WeatherManager();
            WeatherResults = await weatherManager.GetWeatherAsync();
        }
    }
}
```

The **OnGetAsync** method is currently using a class called **WeatherManager** to make a call to the minimal API via the **GetWeatherAsync** method and returning the weather details to the page. What we have here is a dependency between the **OnGetAsync** method of the **IndexModel** class and the **WeatherManager** class.

This is because the **OnGetAsync** method is responsible for creating an instance of the **WeatherManager** class on which it depends. You can see this dependency when you look at the line **var weatherManager = new WeatherManager();** in the **OnGetAsync** method.

As seen in Figure 71, the application works correctly and returns the data as expected, but there are some problems with this approach. Because the **OnGetAsync** method is responsible for the creation of the **WeatherManager** class, it is tightly coupled to the **WeatherManager** implementation.

Tight coupling is considered an anti-pattern in software development. This is because as time goes on, it becomes harder to maintain the code, and any changes (such as implementing an improved **WeatherManager** class) might require many classes to be updated that are dependent on the **WeatherManager** class. Let's see how to solve this problem. We'll start by using an interface.

Creating interfaces

According to the dependency inversion principle, classes should rely on abstractions rather than implementations. Because the **IndexModel** is tightly coupled to the **WeatherManager** implementation, it does not meet this requirement.

We will therefore have to refactor our code to create an abstraction so that we can reduce the coupling between classes and their dependencies. An interface will allow us to do this. The code for the **WeatherManager** class is illustrated in Code Listing 51.

📝 ***Note: Code Listing 51 contains hard-coded strings specific to my machine with regards to the BaseAddress URL port. If you want to use your minimal API in the*** *WeatherManager* ***class, you will need to change your port.***

The implementation code in the **WeatherManager** class does not matter. You can just return some hard-coded data if you like. I just thought it would be fun to use the minimal API created in the previous chapter.

Code Listing 51: The WeatherManager Class

```
using Newtonsoft.Json;
using System.Net.Http.Headers;

namespace WeatherPortal.Core
{
    public class WeatherManager
    {
        private static readonly HttpClient _client = new();

        public WeatherManager()
        {
            _client.BaseAddress = new Uri("https://localhost:44349/");
            _client.DefaultRequestHeaders.Accept.Clear();
            _client.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));
        }

        public async Task<WeatherResults> GetWeatherAsync()
        {
            var response = await _client.GetAsync("sa-weather");
            if (response.IsSuccessStatusCode)
            {
                string apiResp = await
response.Content.ReadAsStringAsync();
                var results =
JsonConvert.DeserializeObject<WeatherResults>(apiResp);
                return results ?? new WeatherResults();
            }
            return new WeatherResults();
        }
    }
}
```

To easily extract an interface from our **WeatherManager** class, place your cursor on the class name and press **Ctrl+.** or right-click and select **Quick Actions and Refactorings** from the context menu.

Next, click **Extract interface** (Figure 72) to create an interface called **IWeatherManager**, as seen in Code Listing 52.
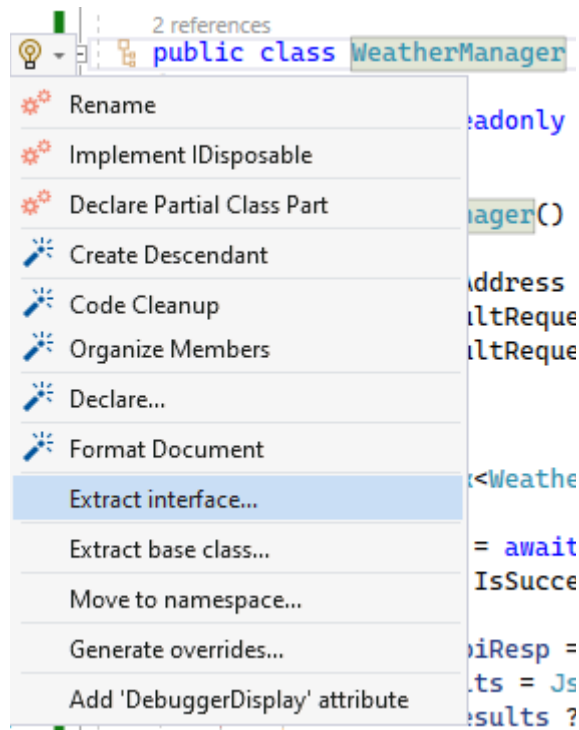
Figure 72: Extract an Interface from the WeatherManager Class

With the interface created, you will notice that the **WeatherManager** class now implements the **IWeatherManager** interface, as seen in the excerpt in Code Listing 53.

Code Listing 52: The IWeatherManager Interface

```
namespace WeatherPortal.Core
{
    public interface IWeatherManager
    {
        Task<WeatherResults> GetWeatherAsync();
    }
}
```

You will notice in Code Listing 52 that the **GetWeatherAsync** method is a member of the created **IWeatherManager** interface.

Code Listing 53: The IWeatherManager Interface Implemented

```
public class WeatherManager : IWeatherManager
{
    private static readonly HttpClient _client = new();
```

This easy refactoring is the first improvement made so that we can decouple the code moving forward. With his improvement made, we can now think about using constructor injection to inject the **WeatherManager** service into our methods.

Using constructor injections

Constructor injection will allow us to pass dependencies to our class via the constructor instead of the class having to create instances of the dependencies it requires. To achieve constructor injection, we can make dependencies parameters of the constructor. To illustrate this, consider the code in Code Listing 54 for the **IndexModel** class.

Code Listing 54: Injecting the Dependency via the Constructor

```csharp
using Microsoft.AspNetCore.Mvc.RazorPages;
using WeatherPortal.Core;

namespace WeatherPortal.Pages
{
    public class IndexModel : PageModel
    {
        private readonly IWeatherManager _service;
        public WeatherResults WeatherResults { get; set; } = new
WeatherResults();

        public IndexModel(IWeatherManager service)
        {
            _service = service;
        }

        public async Task OnGetAsync()
        {
            //var weatherManager = new WeatherManager();
            //WeatherResults = await weatherManager.GetWeatherAsync();
            WeatherResults = await _service.GetWeatherAsync();
        }
    }
}
```

Here we are applying a version of the inversion of control principle to invert control of the creation of the **WeatherManager** dependency. We allow another component to take responsibility for creating an instance of the dependency and passing that dependency to the constructor of the class that depends on the **WeatherManager**. We are now able to achieve loose coupling in our code.

A **private readonly** field called **_service** is created and assigned from within the constructor. Our code now supports the dependency injection pattern. We now need to complete the inversion of control so that ASP.NET Core can take responsibility for creating and injecting the required **WeatherManager** dependency. For that, we need to register our service. Let's do that next.

Registering services in the ServiceCollection

The last thing we need to do is use the Microsoft dependency injection container built into ASP.NET Core. The dependency injection container gets configured when the application starts and we can register the services we need via the **IServiceCollection**. If we do not register our dependencies with the **IServiceCollection**, we will experience runtime exceptions when classes try to use these injected dependencies.

Services are registered in the **Program.cs** file. It is the entry point for our application, and as seen in Code Listing 55, it contains top-level statements.

Code Listing 55: The Program.cs File with Top-Level Statements

```csharp
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

As mentioned earlier, we register services with the **IServiceCollection** of the **WebApplicationBuilder**. The single line of code that we need to add is simply **builder.Services.AddScoped<IWeatherManager, WeatherManager>();**, and the complete code for the **Program.cs** file is illustrated in Code Listing 56.

Code Listing 56: Registering our IWeatherManager Service

```csharp
using WeatherPortal.Core;

var builder = WebApplication.CreateBuilder(args);
```

```
// Add services to the container.
builder.Services.AddRazorPages();

builder.Services.AddScoped<IWeatherManager, WeatherManager>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

What we are saying here is that the container will attempt to create and return an instance of **WeatherManager** when resolving a request for **IWeatherManager**. In layman's terms, when a class uses **IWeatherManager**, give it the **WeatherManager** implementation.

Running your Weather Portal application now will correctly resolve the required dependency from within the dependency injection container and display the weather data returned from the minimal API. You have successfully changed the application to make use of dependency injection.

AddScoped vs. AddTransient vs. AddSingleton

When you register a service with the container, you need to specify the lifetime of the service—in other words, how long must a service live after it has been created. Service lifetimes are defined by using one of three extension methods on the **IServiceCollection**. These are:

- **Scoped**
- **Transient**
- **Singleton**

In our application, we used **Scoped**, but it is important to know what each of these lifetimes means and when to use them.

Transient

When specifying `Transient`, we are telling the container to create a new instance every time the service is invoked. In other words, every class that receives a transient service via dependency injection will receive its own unique instance of that dependency. This does mean a bit more work for the garbage collector, but using `Transient` is the safest choice if you are unsure of the lifetime required for your use case.

Singleton

When registering a service with a `Singleton` lifetime, we are telling the container to only create a single instance of that service for the lifetime of the container (in other words, for the lifetime of the web application). Because the service is registered for the lifetime of the container, it does not require disposal or garbage collection.

Scoped

Considering `Scoped` services, we can think of them as sitting between `Transient` and `Singleton`. A **Scoped** service instance will live for the length of the scope from which it is resolved. Therefore, in an ASP.NET Core application, a scope is created for each request that is handled.

Finally

A lot more can be said regarding dependency injection. In fact, it is a topic on its own, and does require a more detailed look. In this book, I wanted to illustrate the bare bones of what dependency injection is, and why it is something you need to use in your ASP.NET Core 6 applications.