ASP.NET CORE APIs

SUCCINCTLY

BY **DIRK STRAUSS**

Syncfusion®

ASP.NET Core APIs 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 Designing Your API

If you have been a developer for any amount of time, you will undoubtedly have heard the term *API*, which stands for application programming interface. For the purposes of this ebook, one way to think of these is small bits of code that allow other systems to access data and communicate. There are probably more systems out there that make use of APIs than you can shake a stick at.

When you use a website or a mobile application, these systems present data to the user in a way that is pleasing to the eye and looks all pretty on your device screen. Developers spend a lot of time making the UI of a system user-friendly. The user types a URL into the browser, and this opens a website that then displays data, videos, images, and other content formatted in a user-friendly manner. Essentially, APIs do the same thing. The only difference here is that the API does not care for formatting any of the data it returns. It is, therefore, mainly used for communication between systems.

A system can request information from an API, and the API will just return the raw data as well as other information that can be interpreted by the requesting system. This is all done without much input from a human. This means that whenever you open a mobile application (for example), chances are that the application is making use of an API call at some point in its lifecycle without you knowing it.

As a developer, another term that you've probably heard is *API integration*. This is something that APIs were designed to facilitate. If you need to take your system and integrate it with another system, chances are you're going to make use of the other system's APIs. Think of Shopify. Just Google the term "Shopify API," and you will find the online Shopify API reference documentation. If you wanted to create an application that could read data from a specific Shopify store, you would integrate your application with several of the Shopify APIs to enable the transfer of data between your system and the Shopify system.

This is my first port of call whenever I need to integrate between two systems. Searching for the API reference documentation is an essential part of understanding the system API that you are integrating with. And you better believe that not all API documentation is created equal. Some documentation needs a little more research to make sense of.

Be that as it may, the importance of APIs cannot be underestimated. Postman released the 2021 State of the API Report, which you can find [here](#).

The data was compiled by surveying 28,000 developers and combining that with what was observed on the Postman platform. There were seven key findings published in this report, which are as follows:

- The pandemic changed the world, and the world responded with APIs.
- The API ecosystem is global and growing.
- Developers are spending more time with APIs.
- API investments stay strong.
- Quality is the top priority.
- More companies are embracing the API-first philosophy.
- Being API-first pays off.

The term that jumps out to me is *API-first*. This means that developers are inclined to design and define APIs and API schemas before beginning with development. This is sometimes called bottom-up design. This makes it obvious that APIs are regarded as an essential component in many developed systems. The chances that you as a developer will be exposed to an API in some shape or form during your career is very good.

In a blog post, Postman categorizes APIs by who has access to them. These are:

- Internal APIs
- External APIs
- Partner APIs

Internal APIs are private APIs that are used by a team, company, or organization, while external APIs are APIs that are publicly accessible for anyone to use. Partner APIs are APIs that are private and only shared with specific integration partners outside of the organization.

There are also several API architectures available. Some of these include:

- REST API
- Webhooks
- SOAP API
- GraphQL API
- WebSocket API
- gRPC API

The word REST should jump out at you here. REST stands for REpresentational State Transfer. We will be taking a closer look at REST in the next section, but REST APIs are probably one of the most common APIs used today.

Unfortunately, there are some challenges when creating and consuming APIs. I alluded to the lack of proper documentation earlier in this section, and this is a very real problem. The documentation created by the API providers is often listed as the number one obstacle in helping consumers understand how to implement the API.

Another challenge is a lack of knowledge. Simply consuming an API is vastly different from actually developing an API. In this book, I will attempt to elucidate both.

You can find the GitHub repository for the code in this book [here](). Let us start by having a closer look at REST and what it means to use a REST API.

What is REST?

As mentioned in the previous section, REST stands for REpresentational State Transfer, and it is a software architectural style that developers use when developing APIs. The REST pattern provides a simple, uniform interface that can be used to access data (including media and other digital resources) via web URLs. The concepts of REST are basically:

- Separation of client and server data.
- Server requests are stateless.

- Requests can be cached.
- Requests use a uniform interface.

There are a lot of opinions as to what REST is and isn't. The basic idea, however, was summarized as follows in a [Postman blog](#):

> *REST helps you better organize your digital resources and the operations you can perform against them. It's about establishing a shared language to describe your digital capabilities and enable wider collaboration around their web usage.*

You will notice that the summary states the organization of digital resources. This is a key concept in understanding REST.

What are resources?

When talking about REST, we are referring to a URL that points to a resource. Resources are a representation of the objects in your system, such as products, purchase orders, invoices, or employees. In other words, resources are the things in your system that you might want to insert, update, delete, or read.

Some developers think of resources as entities, but it is important to note that resources and entities are not always the same thing. They could be, but resources can also denote one or more entities.

Therefore, a resource can be an employee (for example), but it can also be a sales order along with its associated sales order lines and customer information. Resources can therefore be a collection of entities or only a single entity. At the end of the day, you need some data from an API, and the API will combine all the related information across several entities (or a single entity) and return to you a single representation of this data for use in your system.

The API design

The API we will be creating will be concerned with book information for my library of books. It will contain basic book information such as ISBN, title, and description. You can make this book information as verbose as you want to, but for this project, I have kept the information about a book in my library simple.

Project setup

Our focus is creating an API, but it would be incomplete for me to not go through the initial creation and setup of the project. To this end, I will show you how to get a basic ASP.NET Core Web API project created that targets .NET 5.0. I will also show you how to set up a basic data service using Entity Framework so that we can have some data to test our API with.

For this example project, I will be using Visual Studio 2019 Version 16.9.4, but you can also use the Community (free) edition of Visual Studio.
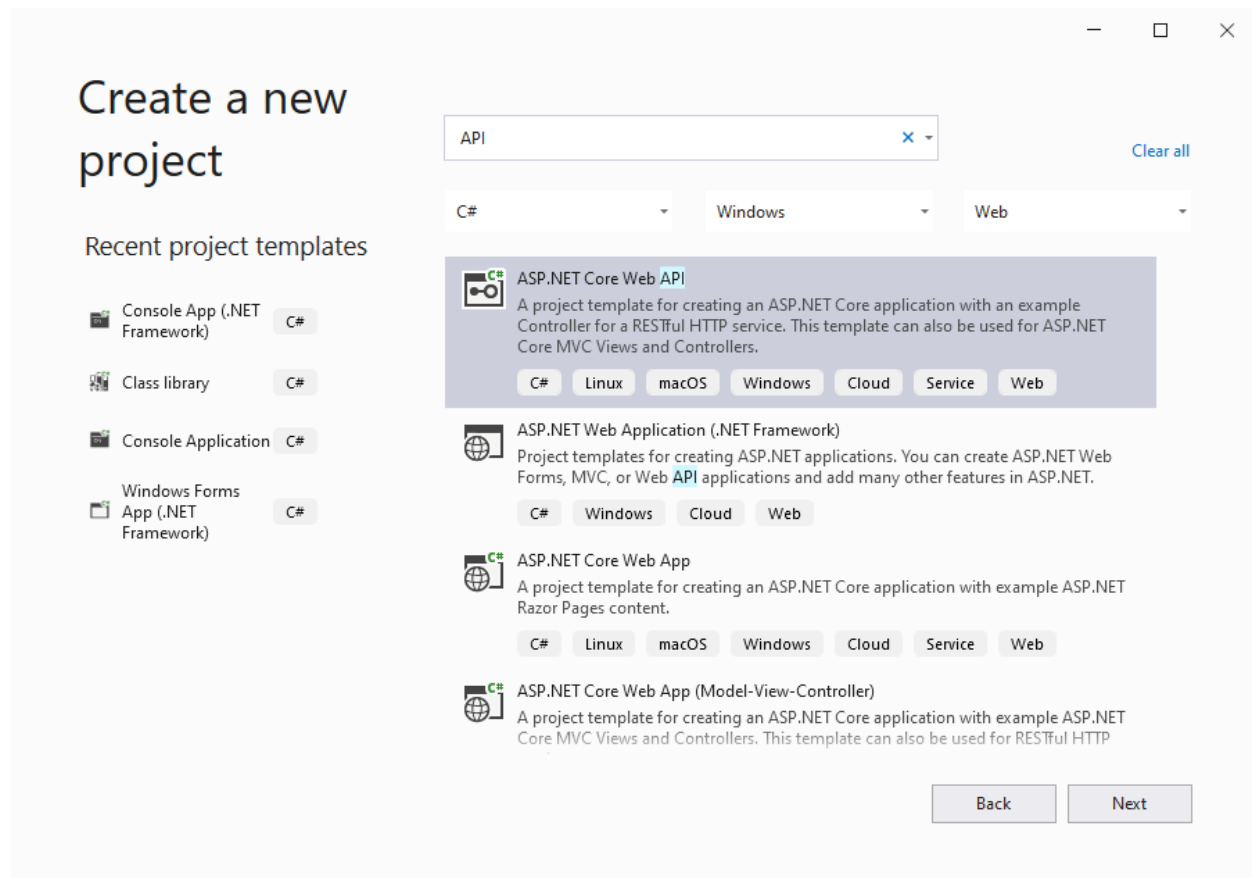
Figure 1: Creating a new project

Start by creating a new ASP.NET Core Web API application using the available project templates from the **Create a new project** screen in Visual Studio. As you can see from Figure 1, I will be creating a C# application.

On the next screen (not shown), you can choose a name for your project and solution. You can call this what you like, but I am creating a book repository API, so I'm calling the solution **BookRepositoryAPI** and the project **BookRepository**.

Figure 2: Choosing the target framework

On the next screen (Figure 2), you can choose your target framework; the one we will be using in this project is the .NET Framework 5.0. I will not be adding any authentication. You can click **Create** to create your project.

Figure 3: The boilerplate code

Visual Studio will now create a basic web API project with some boilerplate code (WeatherForecast) as shown in Figure 3.
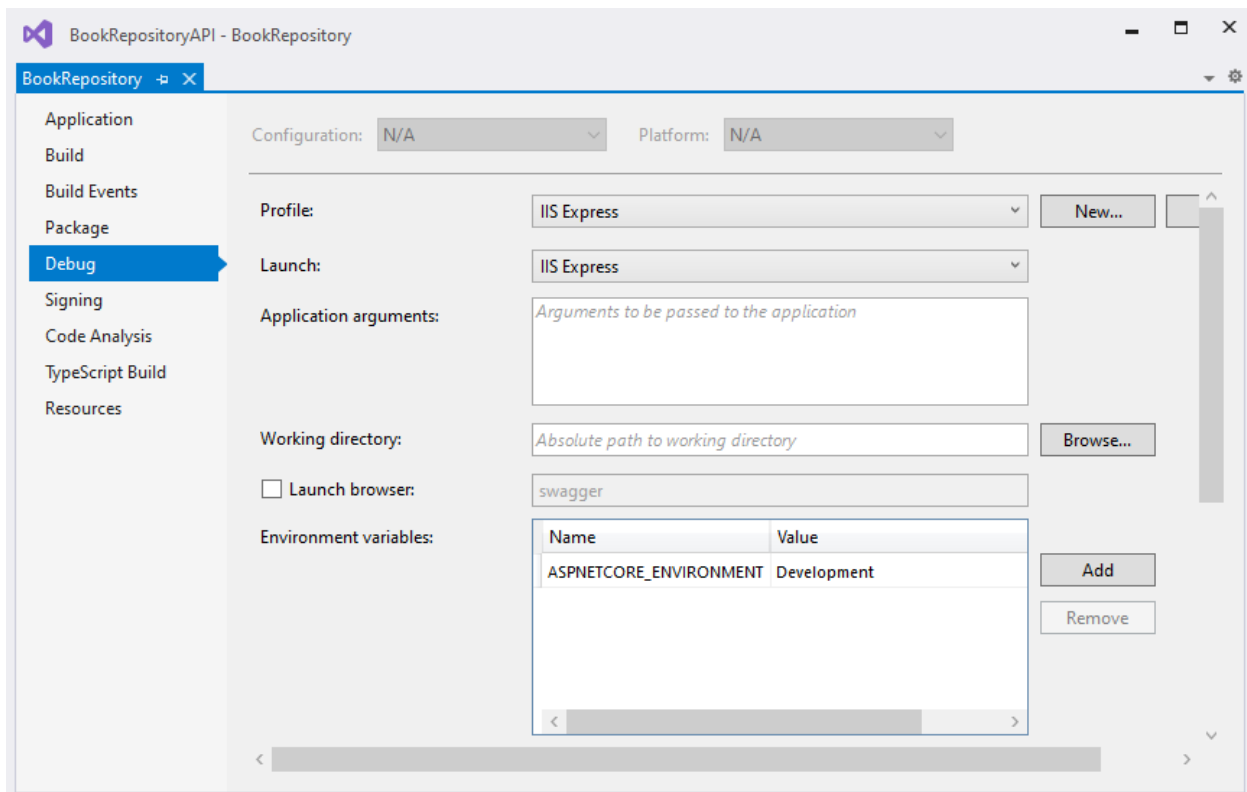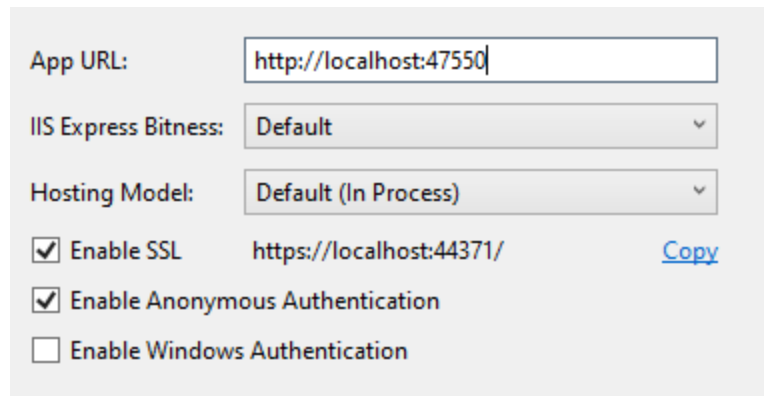
Figure 4: Uncheck Launch browser

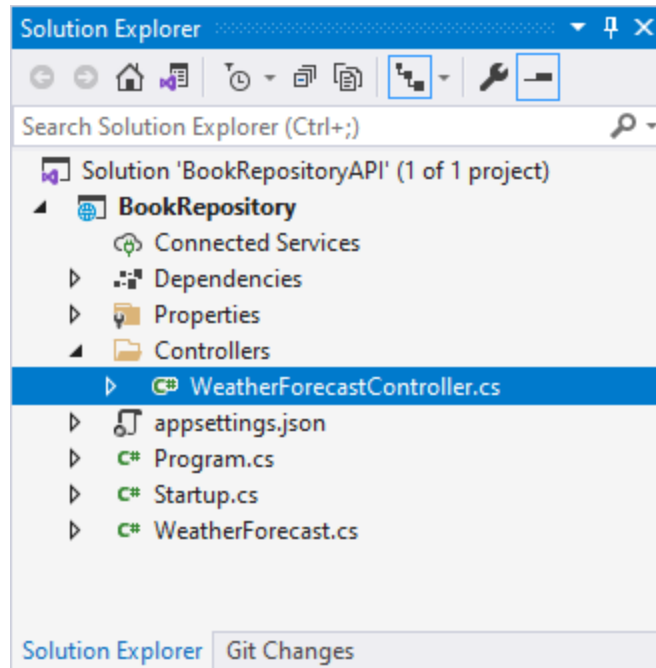Since I am creating an API, there is no user interface to debug. For this reason, I am going to uncheck the **Launch browser** option in the project properties as shown in Figure 4. To get to this option, right-click your **BookRepository** project in the Solution Explorer window and select **Properties** from the context menu. On the **BookRepository** properties page, select the **Debug** tab, and you will see the option to launch a browser. Uncheck this.

A little further down on the properties page, you will see the web server settings, as shown in Figure 5.

Figure 5: My web server settings

Pay special attention to the URL next to the **Enable SSL** checkbox. Your port will most likely be different than the one in Figure 5, which is my URL. Copy this URL, save these changes, and run your API project.

Because you have set the project not to launch a web browser, you will just see that Visual Studio is running. If you right-click **IIS Express** in your taskbar, you will see all the currently running sites.
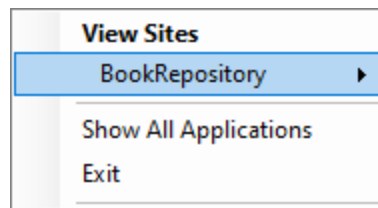
Figure 6: Running applications in IIS Express

BookRepository will be one of the running sites, as seen in Figure 6.
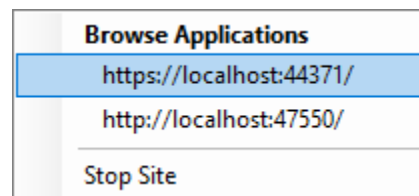
Figure 7: URLs for running applications in IIS Express

After clicking the **BookRepository** site in the running applications screen (Figure 6), you will see the URLs that can be used to access the API. Note that there are the HTTP and HTTPS URLs there that were listed on the project properties page in Figure 5.

With your application still running, open a browser and enter the URL **https://localhost:44371/WeatherForecast** into the address bar, keeping in mind to replace the port with the port listed on your property page.
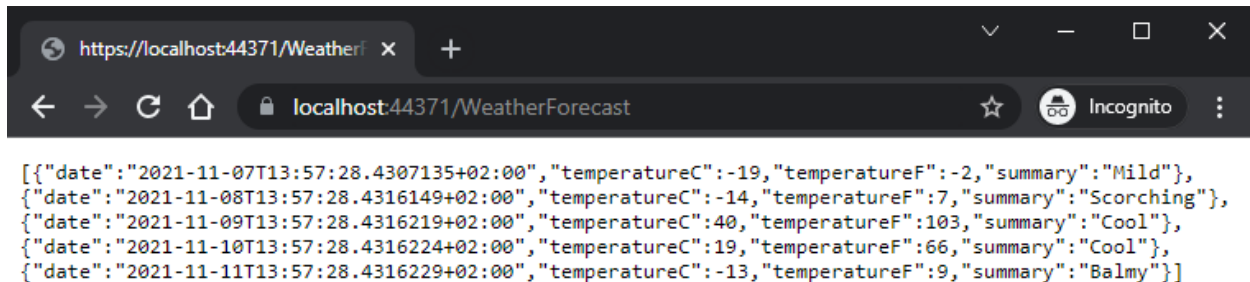
Figure 8: Calling the API in the browser

As you can see in Figure 8, the default API boilerplate code returned some JSON with weather-related data. Congratulations, you have just done a **GET** request on the API you created. It is the most common HTTP method to use. There are others and we will have a look at these in later sections of this book. For now, you have just made a request to the server and told it that you want the information on the WeatherForecast resource. The server dutifully responded by returning you JSON containing the WeatherForecast data.

Adding the data project

The goal of our API is to access data against a real database. For this reason, we will be adding a data project called BookRepository.Data to the solution. Right-click your **BookRepositoryAPI** solution and add a C# class library project.
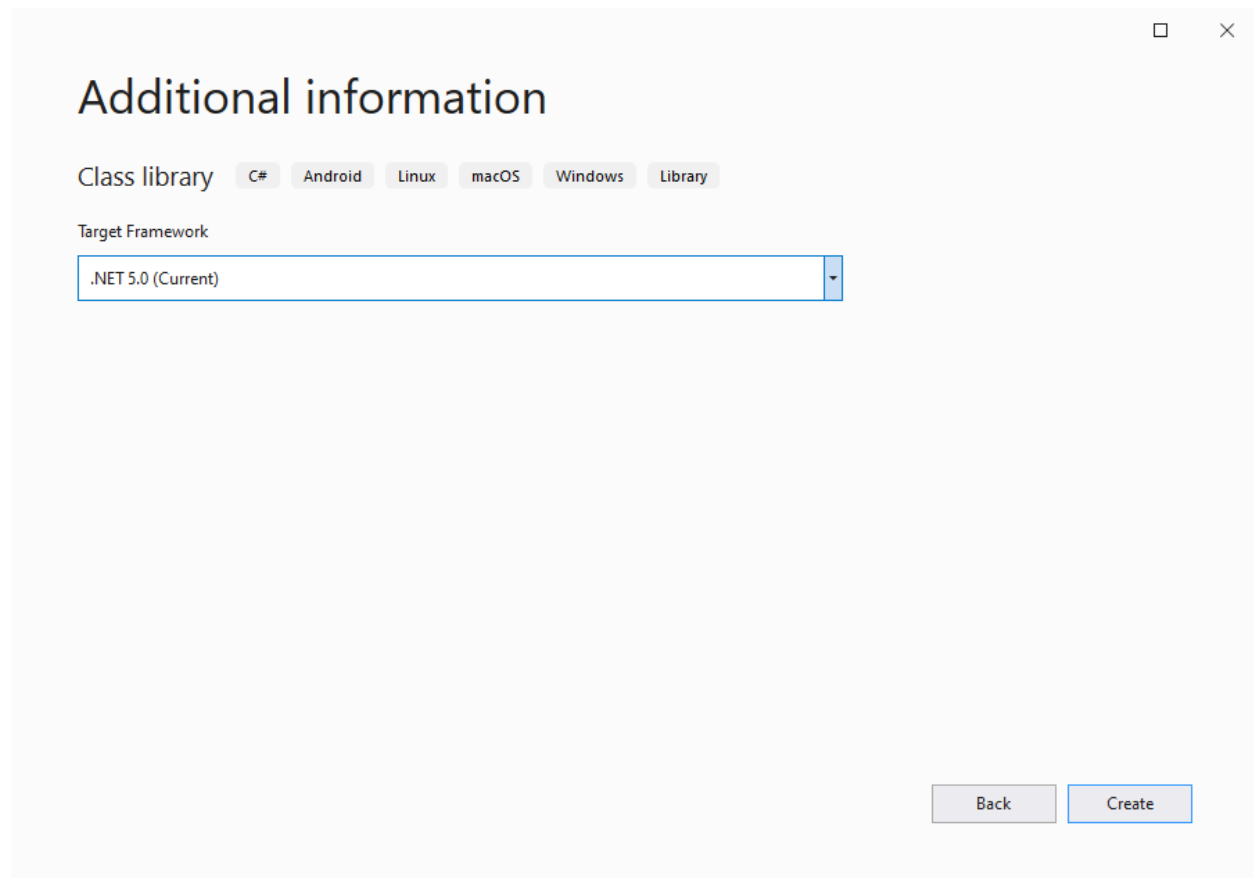
Figure 9: Targeting .NET 5.0

As you can see in Figure 9, the class library targets .NET 5.0.
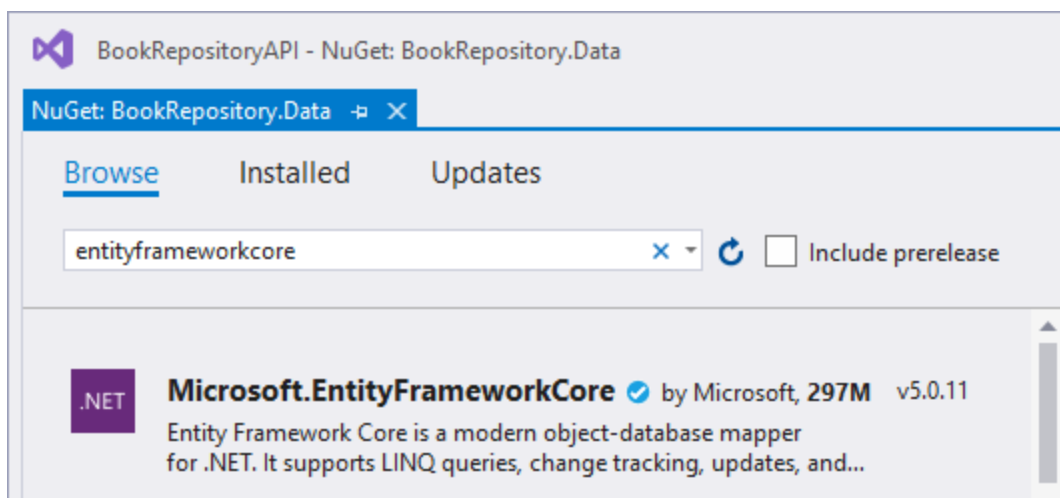
Figure 10: Adding Entity Framework Core

Once the BookRepository.Data project has been added, we need to install the NuGet package Microsoft.EntityFrameworkCore to the data project, as seen in Figure 10.
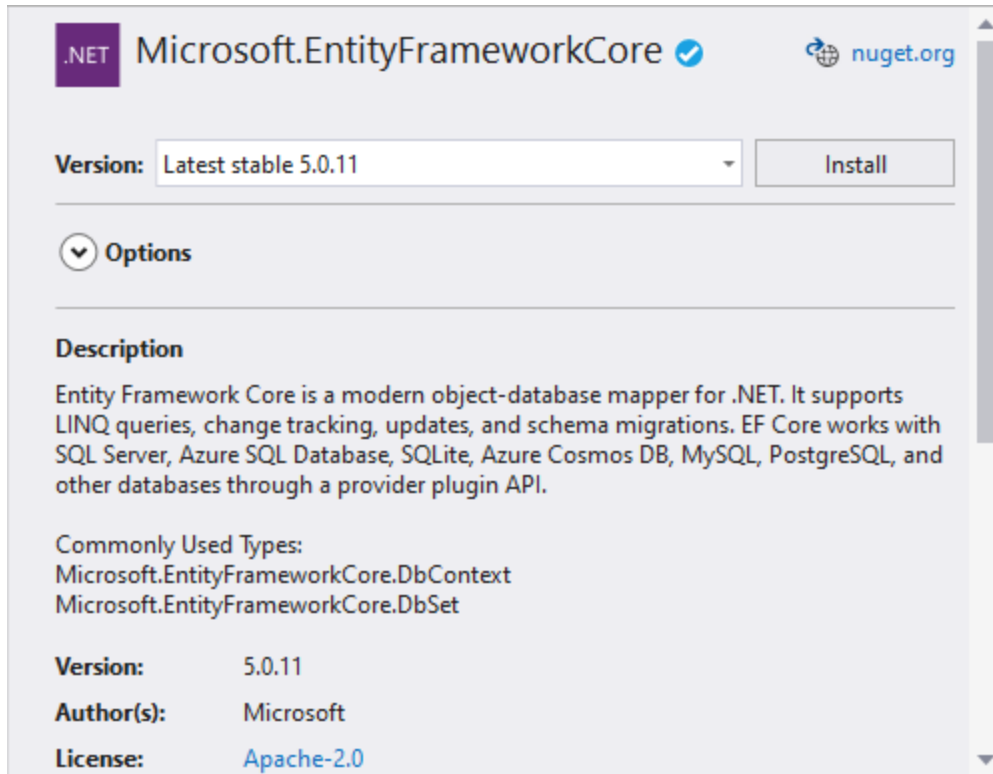
Figure 11: Install the latest stable EF Core NuGet package

Click **Install** to add **Microsoft.EntityFrameworkCore** to the data project. When all is done, your solution will look like Figure 12. Expanding the **Dependencies** under the **BookRepository.Data** project, you will see that Microsoft.EntityFrameworkCore (5.0.11) is listed as a dependency.

Entity Framework Core (EF Core) serves as an O/RM that allows developers to:

- Work against a database using .NET objects.
- Remove the need to write explicit low-level data access code.

When you use EF Core, data access is provided using a model. This comprises entity classes and a context object and represents a database session, allowing you to query and save data.
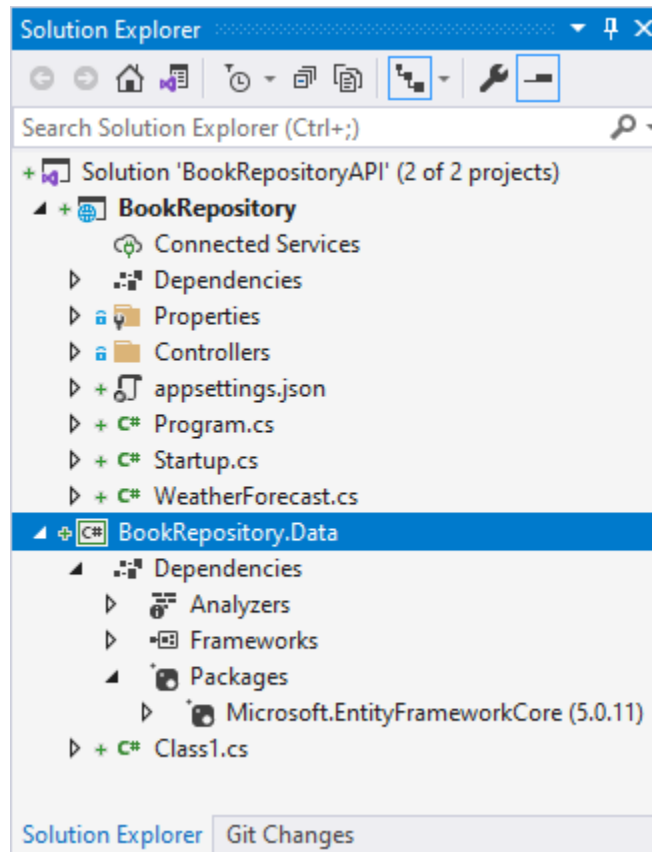
Figure 12: EF Core added to the data project references

It would not, however, make much sense if we didn't add any entities to work with. Let's do that next.

Working with entities

Earlier in this chapter, we stated that resources could contain several entities. Now, we will create some of these entities for our BookRepository project. Our project will be dealing with books, but so far we have nothing that defines exactly what a book is. This is the job of an entity. It tells our application what a book entity will look like.

As before (Figure 9), go ahead and add another class library project called **BookRepository.Core** to your solution. Again, select **.NET 5.0** as the target framework and create the project. You can delete the default `Class1.cs` class or rename it if you like. What you need to end up with is a class called `Book.cs`.

You can see this entity in Code Listing 1. It simply contains properties that describe a book in our repository. For now, we will just add some basic information about a book, such as the ISBN and title.

```
namespace BookRepository.Core
{
    public class Book
    {
        public int Id { get; set; }
        public string ISBN { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public string Publisher { get; set; }
    }
}
```

When you have added your **Book** entity, your solution will look like Figure 13.
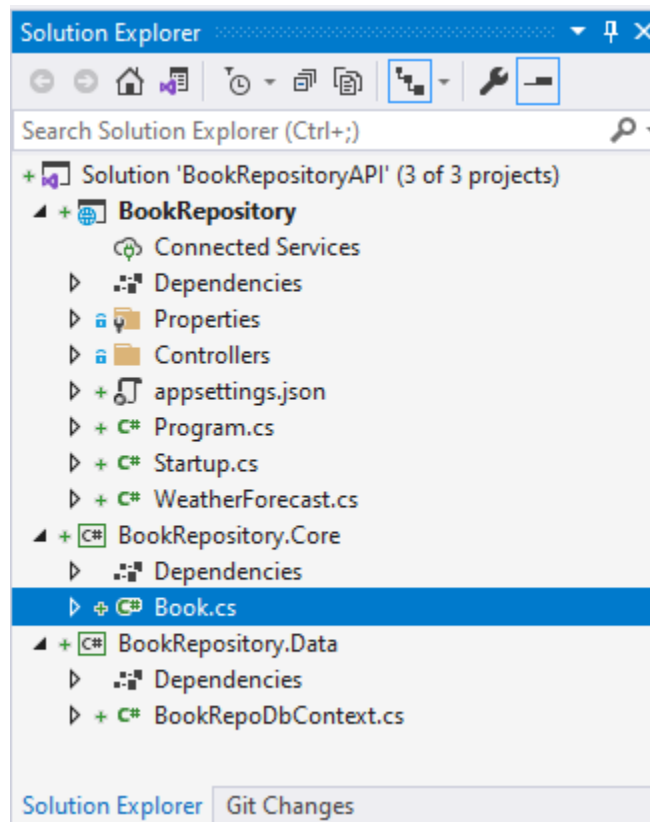
Figure 13: The Book entity

Let us swing back to our data project and flesh this out a bit by adding an interface for our book data.

Adding the IBookData interface

We need to add an interface for the book data so that we can tell our API what methods need to be implemented in the data service we want to create. Once the data service implements the **IBookData** interface, it will be up to the data service to decide exactly how to provide the implementation of the interface.

Inside the **BookRepository.Data** project, add an interface called **IBookData**. You can see the code for the interface in Code Listing 2.

Code Listing 2: The IBookData interface

```csharp
using BookRepository.Core;
using System.Collections.Generic;

namespace BookRepository.Data
{
    public interface IBookData
    {
        IEnumerable<Book> ListBooks();
        Book GetBook(int Id);
        Book UpdateBook(Book bookData);
        Book AddBook(Book newBook);
        int Save();
    }
}
```

You can see that you will have to reference the BookRepository.Core project to use the **Book** entity in your BookRepository.Data project. All this interface does is tell the data service that it must implement some logic to return a list of books, to get a specific book, to update a book, to add a book, and to save a book. By using an interface, we can decouple the data access service by allowing it to implement the **IBookData** interface. No matter what our data service looks like, as long as it implements the interface, it will be able to be injected into our services collection.

> 📝 *Note: Our interface will change when we start creating the API, specifically to include async methods. For now, I just want to get the basics in and create the DbContext.*

The next thing we need to do is implement a **DbContext**.

Implement DbContext

A **DbContext** instance represents a session with the database. It allows us to save and query entity instances. Delete the existing **Class1.cs** file in the **BookRepository.Data** project and add a new class called **BookRepoDbContext.cs**. You can see the code for the **BookRepoDbContext.cs** class in Code Listing 3.

The API will work with **Book** entities; therefore, the **BookRepoDbContext** simply has a property of type **DbSet<Book>**. This tells Entity Framework that I want to query, add, delete, and update books. You will see that I also had to add references to the BookRepository.Core project and to Microsoft.EntityFrameworkCore.

I do agree that the name **DbSet** is not particularly clear about its purpose. If you ever doubt what the purpose of a class is in .NET, you can view the metadata by clicking it and pressing F12. The code comments here are your friend in helping you understand what the particular class does.

Code Listing 3: The BookRepoDbContext

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

namespace BookRepository.Data
{
    public class BookRepoDbContext : DbContext
    {
        public DbSet<Book> Books { get; set; }
    }
}
```

The property on the **BookRepoDbContext** will allow us to work with our database. We will be using **LocalDB** as our database, and it is installed when you install Visual Studio. To check if **LocalDB** is installed, run **sqllocaldb info** in the command prompt.
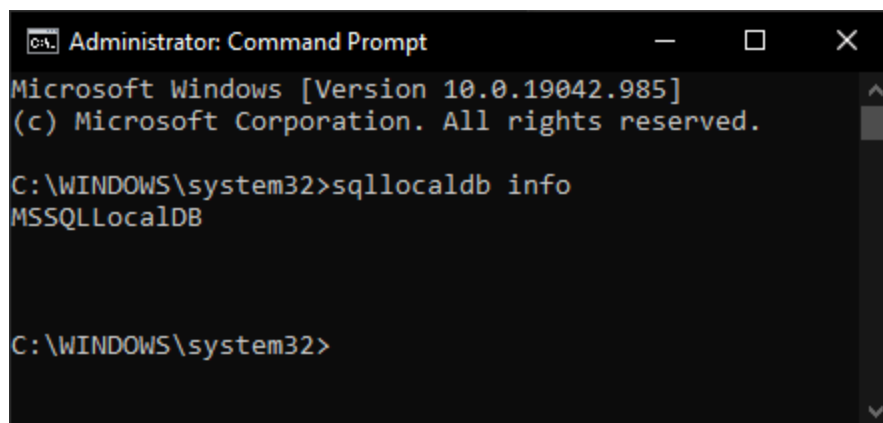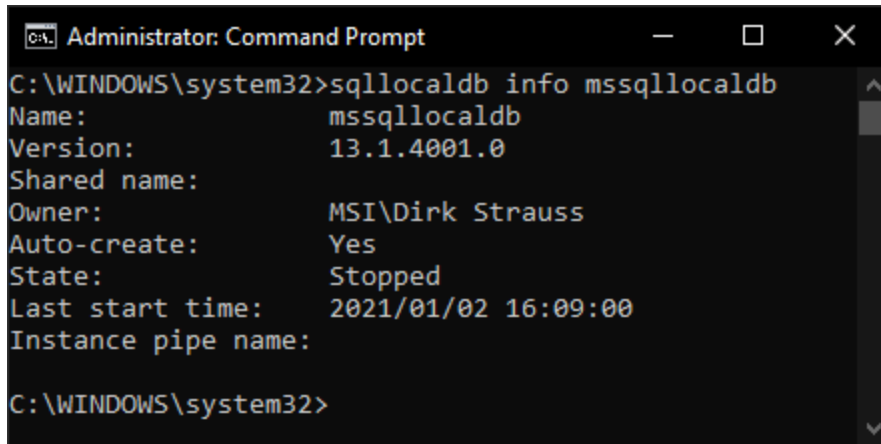
Figure 14: Check if LocalDB is installed

The API will use the built-in **LocalDB** database that is listed in the output displayed in Figure 14. To see more info about the **MSSQLLocalDB** instance, run the command **sqllocaldb info mssqllocaldb** from the command prompt, as you can see in Figure 15.

Figure 15: Info regarding MSSQLLocalDB

You can also view the **LocalDB** instance in Visual Studio by going to **View** > **SQL Server Object Explorer**. This instance may contain several databases, and it is this instance that we need to create a connection to. This we need to specify in the **appsettings.json** file that should be in your BookRepository project.

Code Listing 4: The appsettings.json file

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

The code for the **appsettings.json** file is presented in Code Listing 4. All I have done is add a new section called **ConnectionStrings** that contains a key and value pair for the various database connections we want to use. This section can contain more than one database connection, hence the section **ConnectionStrings** being plural. The key for our database connection is called **BookConn**, and the value is the connection string to the database called **BookRepo** (a database that does not exist yet).

Next, we need a way to tell the **DbContext** about this connection to the database we would like to use. We will do this in the **ConfigureServices** method of the **Startup.cs** class. Modify the **ConfigureServices** method as illustrated in Code Listing 5.

Code Listing 5: The ConfigureServices method

```csharp
public void ConfigureServices(IServiceCollection services)
        {

            _ = services.AddControllers();
            _ = services.AddSwaggerGen(c =>
            {
                c.SwaggerDoc("v1", new OpenApiInfo { Title =
"BookRepository", Version = "v1" });
            });

            _ = services.AddDbContextPool<BookRepoDbContext>(dbContextOptns
=>
            {
                _ = dbContextOptns.UseSqlServer(
                    Configuration.GetConnectionString("BookConn"));
            });
        }
```

This registers the **DbContext** as a service in the **IServiceCollection**.

The **UseSqlServer** method tells the Entity Framework about the **DbContext** being used in the API.

💡 *Tip: You will probably need to add the* Microsoft.EntityFrameworkCore *and* Microsoft.EntityFrameworkCore.SqlServer **NuGet packages to the BookRepository project.**

We also specify that **DbContext** pooling should be used, which allows for increased throughput. This is because instances of **DbContext** are reused instead of having new instances created for each request.

📝 *Note: Pay attention to the fact that the key being used for our connection in the appsettings.json file has to be an exact match to the string being passed to the* GetConnectionString **method seen in Code Listing 5.**

Now that we have registered the **DbContext** as a service in the **IServiceCollection**, we need to change the **BookRepoDbContext** class slightly to tell it about the connection string we specified along with any other options specified with the **DbContextOptionsBuilder** in the **ConfigureServices** method.

Code Listing 6: The modified BookRepoDbContext class

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

```
namespace BookRepository.Data
{
    public class BookRepoDbContext : DbContext
    {
        public BookRepoDbContext(DbContextOptions<BookRepoDbContext>
dbContextOptns) : base(dbContextOptns)
        {

        }

        public DbSet<Book> Books { get; set; }
    }
}
```

Swing back to the **BookRepoDbContext** class and add a constructor that takes
DbContextOptions as a parameter (seen in Code Listing 6). We are now finally ready to use
database migrations to create the database we specified in the connection string specified in the
appsettings.json file (seen in Code Listing 4).

Using database migrations to add a database

For this portion of the process, we will be using the dotnet tool. The dotnet tool should be
already installed; you can check by typing the command **dotnet --info** at the command
prompt. If you do not have the dotnet tool installed, you can install it by running the command
dotnet tool install –global dotnet-ef.

You can find more information on installing the dotnet tool by visiting [this website](#).

We also need to make sure that we have the Microsoft.EntityFrameworkCore.Design NuGet
package added to our BookRepository and BookRepository.Data projects. After adding the
NuGet package, your solution should now look as illustrated in Figure 16.
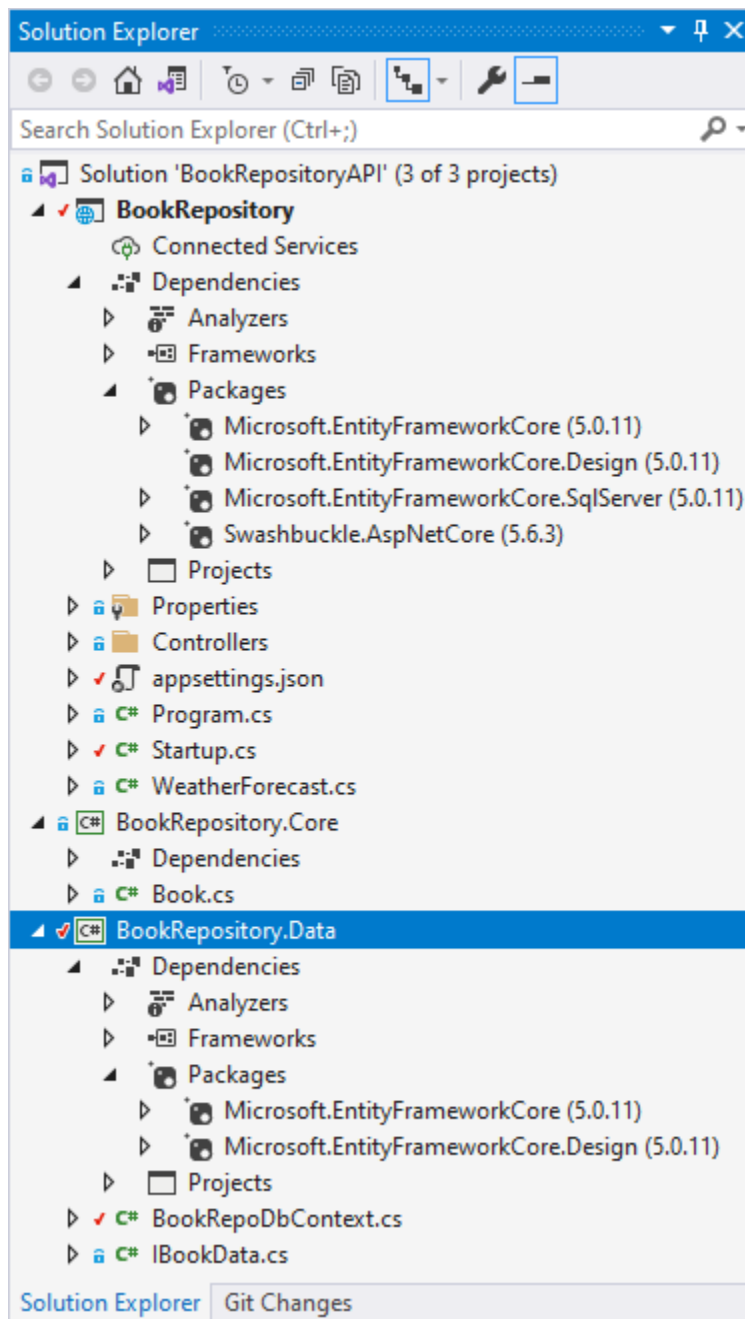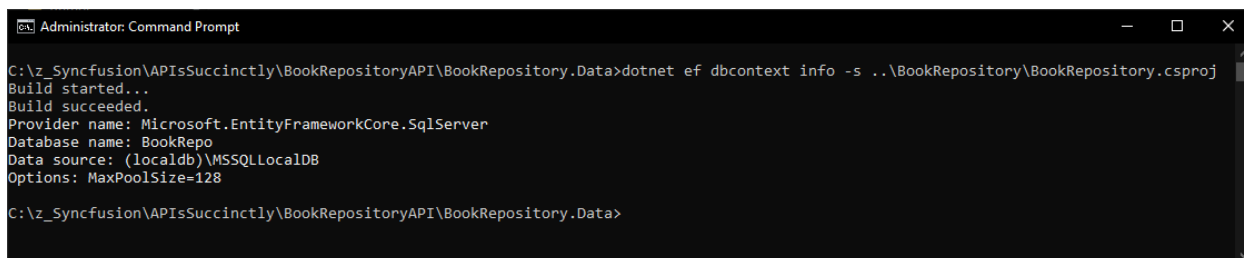
Figure 16: The BookRepositoryAPI solution with the installed NuGet packages

It is extremely important that the NuGet package versions of Entity Framework Core match between the BookRepository and BookRepository.Data projects. My versions are 5.0.11.

Before adding and running any migrations, I always run the command `dotnet ef dbcontext info`, also specifying the startup project. This way I can catch any missing NuGet packages (you might miss the addition of the Design NuGet package, for example).

Open a command prompt and change the directory to the **BookRepository.Dat**a project folder. Now run the following command: **dotnet ef dbcontext info -s ..\BookRepository\BookRepository.csproj**. The output should look as illustrated in Figure 17.

Figure 17: Running dbcontext info

It is important to specify the startup project using the **-s** option. This is because the BookRepository.Data project does not know about the **Startup.cs** class that contains the **ConfigureServices** method.

If this command produced the output as illustrated in Figure 17, we should be in a good position to add the migration to our data project.

Database migrations will enable us to keep the database in sync when a model in our project changes. EF Core will compare the current data model to a snapshot of the old model and figure out what has changed. It will then generate migration files and apply those migrations to the database and record the history in a table. This is nice because it allows you to see which migrations have been applied.

Keeping the command prompt pointed to the **BookRepository.Data** project, run **dotnet ef migrations** from the command line.

Code Listing 7: Available commands

```
Usage: dotnet ef migrations [options] [command]

Options:
  -h|--help        Show help information
  -v|--verbose     Show verbose output
  --no-color       Don't colorize output
  --prefix-output  Prefix output with level

Commands:
  add     Adds a new migration
  list    Lists available migrations
  remove  Removes the last migration
  script  Generates a SQL script from migrations

Use "migrations [command] --help" for more information about a command.
```

From the output, you will notice that we have a few commands available to us. These are:

- `add`
- `list`
- `remove`
- `script`

In our case, we want to add a new migration. From the command prompt, run the **dotnet ef migrations add --help** command.

Code Listing 8: The add command arguments and options

```
Usage: dotnet ef migrations add [arguments] [options]

Arguments:
  <NAME>  The name of the migration.

Options:
  -o|--output-dir <PATH>                  The directory to put files in.
Paths are relative to the project directory. Defaults to "Migrations".
  --json                                  Show JSON output. Use with --
prefix-output to parse programatically.
  -n|--namespace <NAMESPACE>              The namespace to use. Matches the
directory by default.
  -c|--context <DBCONTEXT>                The DbContext to use.
  -p|--project <PROJECT>                  The project to use. Defaults to
the current working directory.
  -s|--startup-project <PROJECT>          The startup project to use.
Defaults to the current working directory.
  --framework <FRAMEWORK>                 The target framework. Defaults to
the first one in the project.
  --configuration <CONFIGURATION>         The configuration to use.
  --runtime <RUNTIME_IDENTIFIER>          The runtime to use.
  --msbuildprojectextensionspath <PATH>   The MSBuild project extensions
path. Defaults to "obj".
  --no-build                              Don't build the project. Intended
to be used when the build is up-to-date.
  -h|--help                               Show help information
  -v|--verbose                            Show verbose output.
  --no-color                              Don't colorize output.
  --prefix-output                         Prefix output with level.
```

From the output in Code Listing 8, you will notice that you can give the migration a name when adding it. I am simply going to call this migration **Initial** to show that it was the first one created. To add this migration, run the command listed in Code Listing 9 from the command prompt.

```
dotnet ef migrations add Initial -s ..\BookRepository\BookRepository.csproj
```

Notice that we still need to specify the startup project by specifying the **-s** option after the migration name **Initial**.

Code Listing 10: Migration successfully added

```
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```

After the migration has been added, have a look at the BookRepository.Data project in Visual Studio.
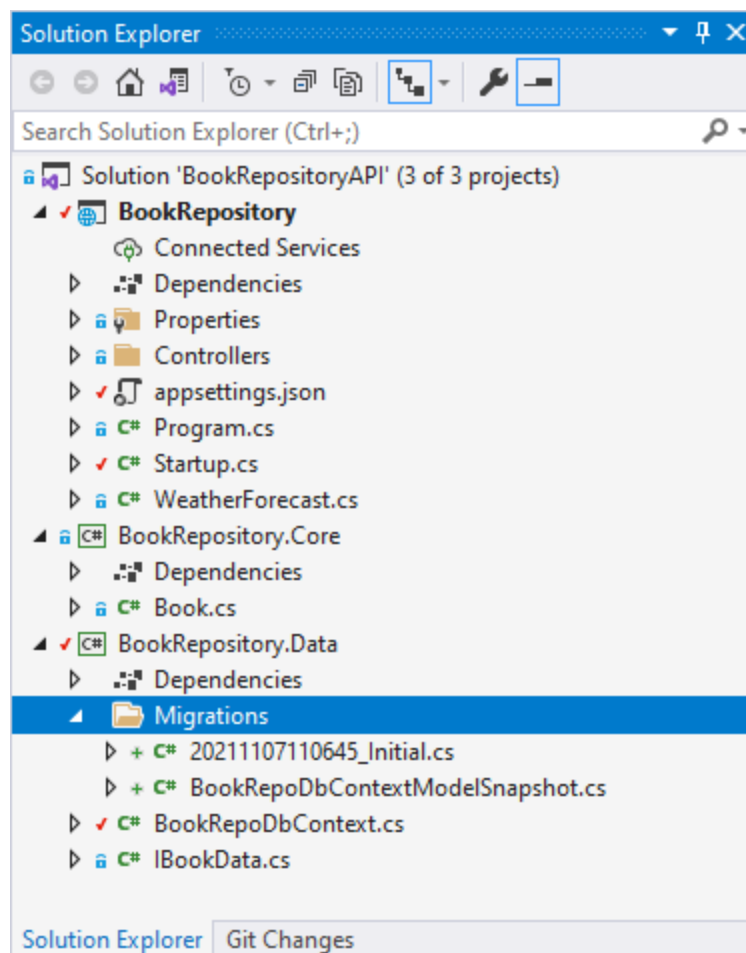
Figure 18: Migrations added to BookRepository.Data project

A Migrations folder has been added to the BookRepository.Data project. You will also see the **Initial** migration created, called **20211107110645_Initial.cs**.

We are now ready to create the database in the **LocalDB** instance.

> 📝 *Note: It was here that I found out that the BookRepository.Data project required NuGet packages called Microsoft.EntityFrameworkCore.Relational and Microsoft.EntityFrameworkCore.SqlServer. Just add these NuGet packages to the data project and you should be good to go.*

From the command prompt, run the **dotnet build** command. If you get a successful build, go ahead and run the command as listed in Code Listing 11.

Code Listing 11: Create the database on LocalDB

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

When the command has been completed, you can open the SQL Server Object Explorer and expand the **Databases** folder under the **MSSQLLocalDB** instance to see the created database as illustrated in Figure 19.
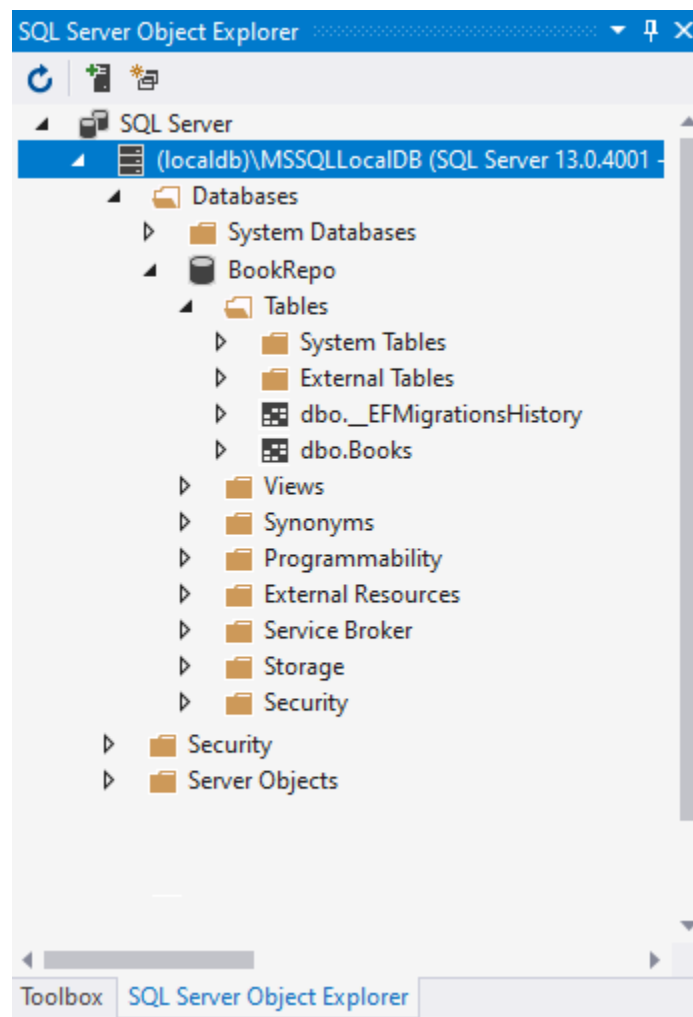
Figure 19: The created BookRepo database

Expanding the **Tables** folder, you will see the **__EFMigrationsHistory** table as well as the **Books** table.
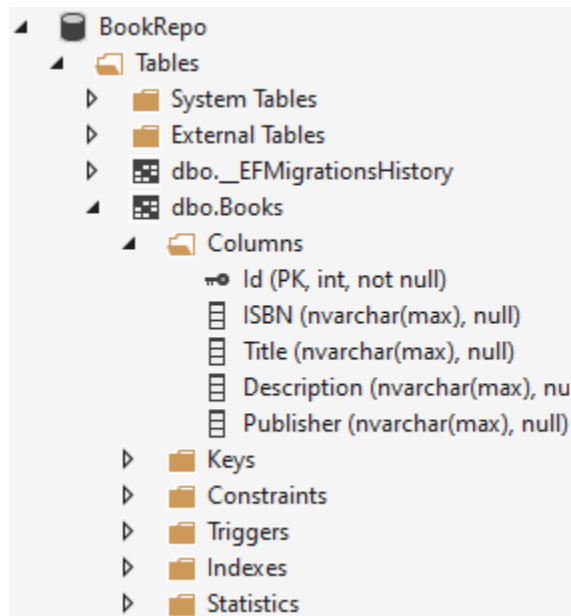
Figure 20: Viewing the Books table columns

Expanding the columns, I see that I have forgotten to add a column for Author. How do I add this column to my database? It is here that database migrations show their true worth.

Adding a column to the database

Previously I saw that I had forgotten to add an Author column. This is a problem because it is information that I will need going forward. Luckily, updating the database is simple. It only requires that we update our **Book** class in the BookRepository.Core project, as seen in Code Listing 12.

Code Listing 12: The updated Book class

```
namespace BookRepository.Core
{
    public class Book
    {
        public int Id { get; set; }
        public string ISBN { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public string Publisher { get; set; }
        public string Author { get; set; }
    }
}
```

All I have done is add a new property for **Author**. The next thing I need to do is add a new migration with this change. As mentioned earlier in this section, database migrations enable us to keep the database in sync when a model in our project changes. EF Core will compare the current data model to a snapshot of the old model and figure out what has changed.

Code Listing 13: Adding the new database migrations

```
dotnet ef migrations add BookModelUpdate1 -s
..\BookRepository\BookRepository.csproj
```

Run the command in Code Listing 13 to create a new database migrations file called **BookModelUpdate1**, and then you will see that it has been added to the Migrations folder in the BookRepository.Data project, as seen in Figure 21.
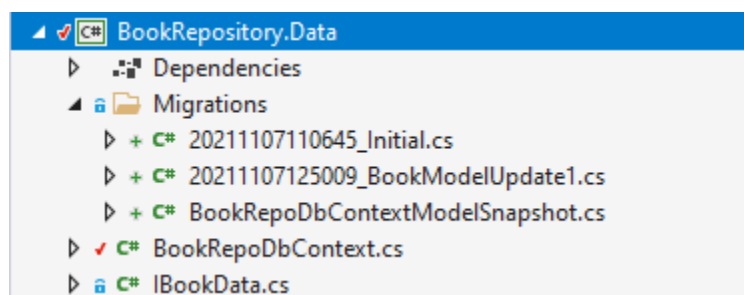
Figure 21: The BookModelUpdate1 database migration

All that remains to be done is to run the database migrations by executing the command in Code Listing 14 in the command prompt.

Code Listing 14: Updating the database

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

After this has been completed, refresh the **Books** table in the database and you will see that the Author column has been added, as shown in Figure 22.
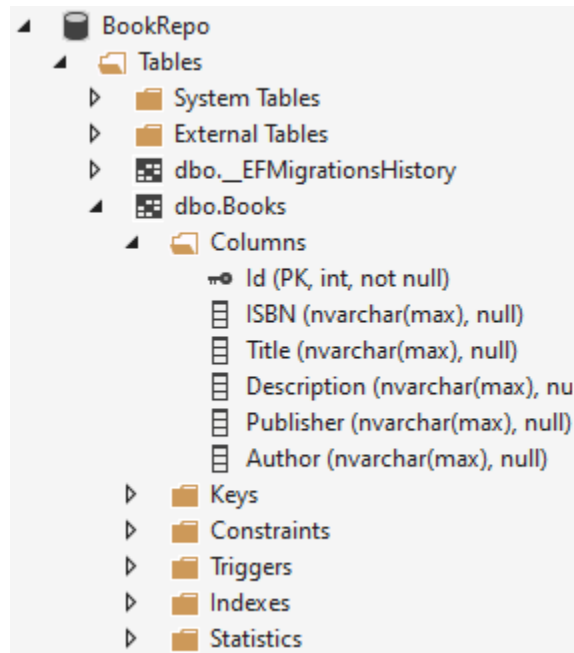
Figure 22: The updated Books table

This is generally the process that you will follow when modifying and adding to the database when our models change in the project. All that remains for us to do now is to add a data access service.

Adding the data access service

Create a new class called **SqlData** in the **BookRepository.Data** project, and let it implement the **IBookData** interface. The complete code is found in Code Listing 15, so I will not go through this in any detail. Suffice to say that it is quite basic and all that we need to start building our API.

Code Listing 15: The SqlData data service class

```csharp
using BookRepository.Core;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;

namespace BookRepository.Data
{
    public class SqlData : IBookData
    {
        private readonly BookRepoDbContext _database;

        public SqlData(BookRepoDbContext database)
        {
            _database = database;
```

```csharp
        }

        public Book AddBook(Book newBook)
        {
            _ = _database.Add(newBook);
            return newBook;
        }

        public Book GetBook(int Id)
        {
            return _database.Books.Find(Id);
        }

        public IEnumerable<Book> ListBooks()
        {
            return _database.Books.OrderBy(b => b.Title);
        }

        public int Save()
        {
            return _database.SaveChanges();
        }

        public Book UpdateBook(Book bookData)
        {
            var entity = _database.Books.Attach(bookData);
            entity.State = EntityState.Modified;
            return bookData;
        }
    }
}
```

Lastly, we need to register our data access service in the **services** collection. Switch to the **BookRepository** project and open the **Startup.cs** class.

Code Listing 16: Registering the data service

```csharp
services.AddScoped<IBookData, SqlData>();
```

We are adding this service registration and telling our API that whenever something asks for an implementation of **IBookData**, give it the **SqlData** class, as seen in Code Listing 16.

Code Listing 17: The modified ConfigureServices method

```csharp
public void ConfigureServices(IServiceCollection services)
        {

            _ = services.AddControllers();
```
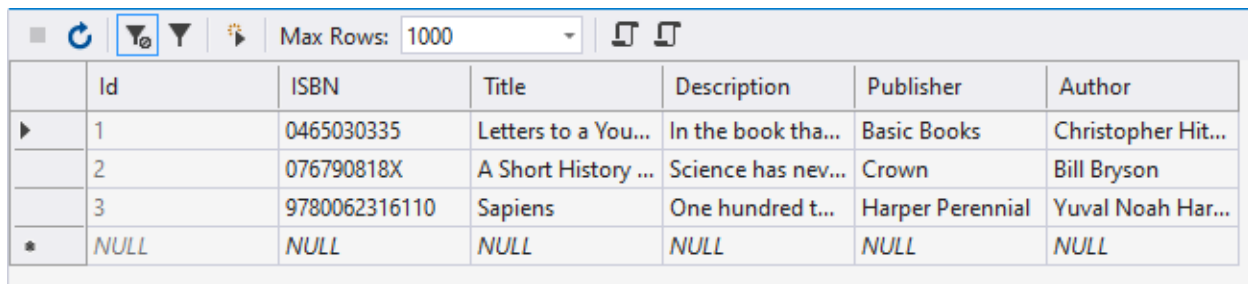
```
        _ = services.AddSwaggerGen(c =>
        {
            c.SwaggerDoc("v1", new OpenApiInfo { Title =
"BookRepository", Version = "v1" });
        });

        _ = services.AddDbContextPool<BookRepoDbContext>(dbContextOptns
=>
        {
            _ = dbContextOptns.UseSqlServer(
                Configuration.GetConnectionString("BookConn"));
        });

        _ = services.AddScoped<IBookData, SqlData>();
    }
```

This line of code is added to the **ConfigureServices** method of the **Startup.cs** class, as seen in Code Listing 17. This is all that we will need to wire up our project with a SQL database to work with. I will leave populating the **Books** table with data up to you. It is quite easy to do. Right-click on the table in SQL Server Object Explorer, select **View Data**, and start plugging in some data to work with.

Figure 23: The data in my Books table

Figure 23 shows the data that I have added to my **Books** table.

How to use Postman

Throughout this book, I will be using Postman to test the API. Postman is a free tool that you can download here. Once you have installed Postman, you can start using it to call APIs. As a quick start, I will show you how to perform a simple **GET** operation on the official public API for the public-apis project.

First, have a look at the project page on GitHub.

The URL in Code Listing 18 is the public APIs URL. All that the **GET** will return is a list of all public APIs currently cataloged in the project.

```
https://api.publicapis.org/entries
```

In Postman, make sure that you have selected a **GET** operation, and then paste the URL from Code Listing 18 into the URL field, as illustrated in Figure 24. Click **Send** to make the **GET** request.
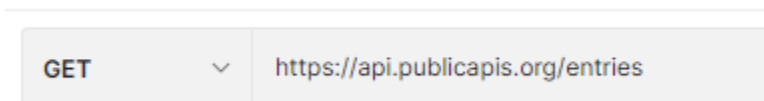
Figure 24: The URL for the GET operation

Postman will make the API call and return the response to you from the **GET** request as shown in Figure 25.
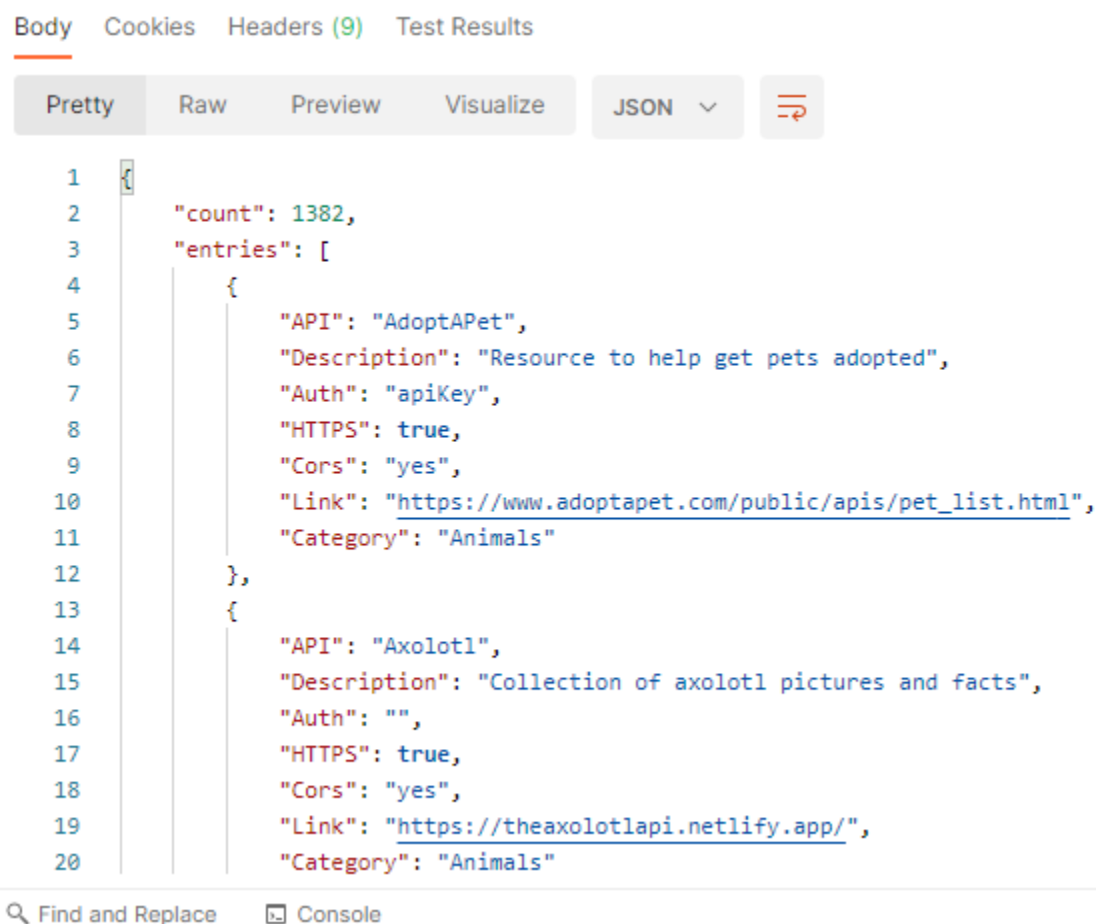
Figure 25: The GET response JSON

As you can see, it simply returns a list of APIs in its catalog. The contents of the response here are of no concern to us (except for perhaps the axolotl API—everyone loves axolotls). What is important is understanding what Postman can do for us.

Postman gives us a way to test the workings of our API, which, as you can imagine, is going to be created without a UI. This book is not going to go in depth on the topic of using Postman; the examples will illustrate the most basic usage. However, I encourage you to get to know Postman better. It is an extremely powerful tool and knowing how to use it properly will prepare you for your future API development.

Chapter 2 Returning Data with Your API

The first thing we will be doing is adding a new controller and letting it return some dummy data. This is just a way to start hooking up the bits and pieces that make up our API.

Creating actions

In Visual Studio, create a new controller called **BookController**. You can delete the default **WeatherForecastController** that was created as part of the project.
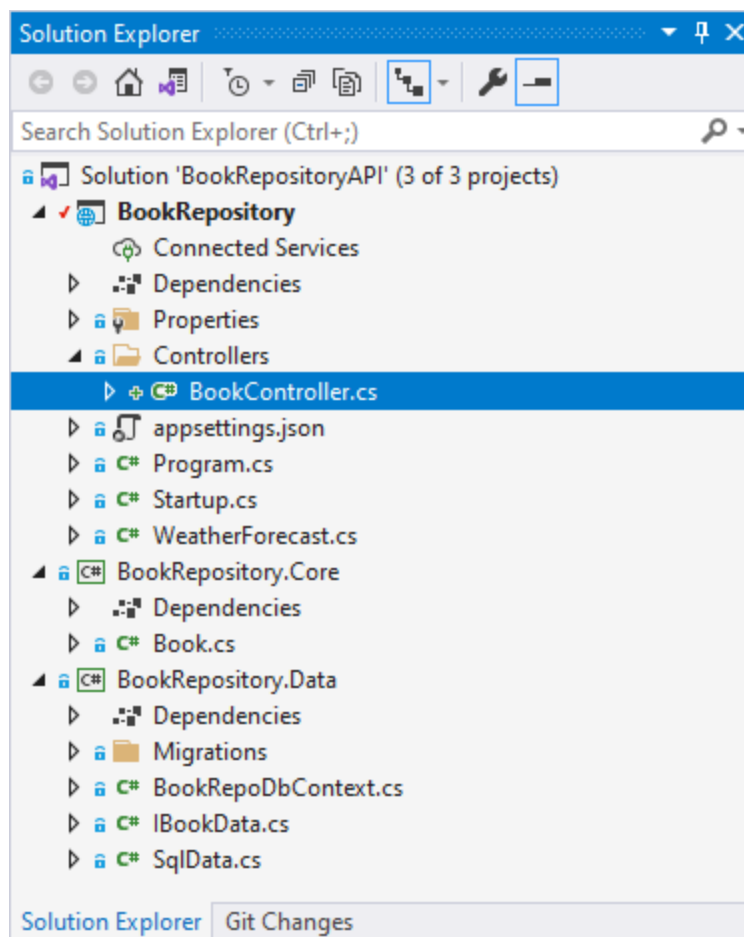
Figure 26: The new BookController

Your solution will look as illustrated in Figure 26. Open the controller and start by adding a simple **GET** action that will return an anonymous object containing a book title and ISBN. Looking at Code Listing 19, it is important to note that the **BookController** class inherits from **ControllerBase**. This is just the base class for a controller without view support. Seeing as we aren't going to have any views, this is perfect.

```
using Microsoft.AspNetCore.Mvc;

namespace BookRepository.Controllers
{
    [Route("api/book")]
    public class BookController : ControllerBase
    {
        public object Get()
        {
            return new { Title = "Sapiens", ISBN = "9780062316110" };
        }
    }
}
```

You will also notice that we are specifying the route on the **BookController** class with a route attribute of **api/book**. If you ran your API project now and performed a **GET** request in Postman using the URL **https://localhost:44371/api/book**, then your hardcoded JSON data will be returned to you.

By convention, however, we would never hard-code a route, as illustrated in Code Listing 19. We would use **[controller]** to tell the API to use whatever comes before the word **controller** as our route. Do this now by changing the route from **[Route("api/book")]** to **[Route("api/[controller]")]**, as seen in Code Listing 20.

Code Listing 20: Using a more flexible route

```
using Microsoft.AspNetCore.Mvc;

namespace BookRepository.Controllers
{
    [Route("api/[controller]")]
    public class BookController : ControllerBase
    {
        public object Get()
        {
            return new { Title = "Sapiens", ISBN = "9780062316110" };
        }
    }
}
```

Run your API in Visual Studio and perform the same **GET** request in Postman using the URL **https://localhost:44371/api/book**. As illustrated in Figure 27, the results returned are the same as before we changed the route. The only difference is that it is not hard-coded to **Book**.

Figure 27: The GET results

You will also notice that the API returned a status code of 200. This means that the **GET** request was successful. The question we now have is, what if something went wrong during the API call? How would we inform the user of this event? This is where an explanation of using status codes is required.

Using status codes

Calling an API requires a request and a response: you make a request, and the API responds. As part of the response, the API uses status codes to indicate the status of the request. Did it succeed or not; was the resource found or not; are you authorized to make this request or not?

There are quite a few status codes that can be used, but the following table lists some of the main ones that you might come across.

Table 1: Status codes

Code	Description
200	OK
201	Created
202	Accepted
302	Found
304	Not Modified
307	Temp Redirect
308	Perm Redirect
400	Bad Request
401	Not Authorized
403	Forbidden

Code	Description
404	Not Found
405	Method Not Allowed
409	Conflict
500	Internal Error

The three status code ranges that you will be most concerned about are:

- 2xx: The request worked.
- 4xx: You did something wrong.
- 5xx: There was an error on the server.

The 500 range includes:

- 503: Service Unavailable
- 504: Gateway Timeout

There is not just a single 500 status code, as listed in Table 1. For a full list of HTTP status codes, have a look at the list provided by the Internet Assigned Numbers Authority.

Status codes give us the ability to provide a clear response to the requester as to the outcome of their request. It is these status codes that we will be utilizing in our API. Looking back to Code Listing 20, modify your code as illustrated in Code Listing 21.

Code Listing 21: Modified action

```
using Microsoft.AspNetCore.Mvc;

namespace BookRepository.Controllers
{
    [Route("api/[controller]")]
    public class BookController : ControllerBase
    {
        [HttpGet]
        public IActionResult GetBooks()
        {
            return Ok(new { Title = "Sapiens", ISBN = "9780062316110" });
        }
    }
}
```

I have changed the **Get** method to return an **IActionResult** and added the **HttpGet** attribute to the method. This allows me to indicate that this method is the **GET** action on the same route that we specified on the controller. The name of the method (or action), therefore, isn't important, and we can be a bit more descriptive in naming our actions. I am also wrapping the return object with an **Ok** method that indicates what status code to return. The **Ok** method is defined in the **ControllerBase** class, so it is available to the derived **BookController** class. The **GetBooks** action is, therefore, our endpoint on this API to return a list of books.

Calling your API in Postman again, you should still receive the response as seen in Figure 27. You will see how to use different status codes later on in this book, but for now, we are just returning **Ok** which is a status 200.

Returning collections with GET

Our API project contains a data access layer. We set this up in Chapter 1. We now want to be able to use the data layer and perform API actions against our database. If you look back to Code Listing 16, you will remember that we registered our data access service in the services collection in the **Startup.cs** class. This allows us to use dependency injection to inject this service into our classes and use the service to perform actions against the database.

To inject the data access service into our controller, we will need to create a constructor. Looking at the complete code in Code Listing 22, you will see that it has changed somewhat. I have added a constructor and passed in the **IBookData** interface. This dependency injection allows me to use the data access service in my controller.

Code Listing 22: Modifying the BookController

```
using BookRepository.Data;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using System;

namespace BookRepository.Controllers
{
    [Route("api/[controller]")]
    public class BookController : ControllerBase
    {
        private readonly IBookData _service;

        public BookController(IBookData service)
        {
            _service = service;
        }

        [HttpGet]
        public IActionResult GetBooks()
        {
            try
```

```
            {
                var books = _service.ListBooks();
                return Ok(books);
            }
            catch (Exception)
            {
                return StatusCode(StatusCodes.Status500InternalServerError,
"There was a database failure");
            }
        }
    }
}
```

I have also added a **try** block with a **catch** that returns a different status code. The **StatusCodes** class is in the **Microsoft.AspNetCore.Http** namespace, so you will have to import this namespace, too.

In the **try** block, I am using the service to return the list of books. With this code in place, call the API again in Postman, and you will see that it returns the results stored in the database.

Code Listing 23: The list of books returned from my database

```
[
    {
        "id": 2,
        "isbn": "076790818X",
        "title": "A Short History of Nearly Everything",
        "description": "Science has never been more involving or entertaini
ng.",
        "publisher": "Crown",
        "author": "Bill Bryson"
    },
    {
        "id": 1,
        "isbn": "0465030335",
        "title": "Letters to a Young Contrarian",
        "description": "In the book that he was born to write, provocateur
and best-
selling author Christopher Hitchens inspires future generations of radicals
, gadflies, mavericks, rebels, angry young (wo)men, and dissidents.",
        "publisher": "Basic Books",
        "author": "Christopher Hitchens"
    },
    {
        "id": 3,
        "isbn": "9780062316110",
        "title": "Sapiens",
```

```
        "description": "One hundred thousand years ago, at least six differ
ent species of humans inhabited Earth. Yet today there is only one—
homo sapiens. What happened to the others?",
        "publisher": "Harper Perennial",
        "author": "Yuval Noah Harari"
    }
]
```

The results that are returned for you will most likely be different (because you would have added different book data than I have). In Chapter 1, , however, you can see that the data in my database table matches the data returned here in the API call. But there is still something that I don't like. I want to do this **GET** request asynchronously (and you should implement **async**). For this, we need to modify the **IBookData** interface, as seen in Code Listing 24.

Code Listing 24: The modified IBookData interface

```csharp
using BookRepository.Core;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace BookRepository.Data
{
    public interface IBookData
    {
        IEnumerable<Book> ListBooks();
        Task<IEnumerable<Book>> ListBooksAsync();
        Book GetBook(int Id);
        Book UpdateBook(Book bookData);
        Book AddBook(Book newBook);
        int Save();
    }
}
```

I have added **Task<IEnumerable<Book>> ListBooksAsync()** to the interface, and this needs to be implemented on the **SqlData.cs** class. The implementation of this async method is illustrated in Code Listing 25.

Code Listing 25: Implemented ListBooksAsync in SqlData.cs

```csharp
public async Task<IEnumerable<Book>> ListBooksAsync()
{
    return await _database.Books
        .OrderBy(b => b.Title)
        .ToListAsync();
}
```

We can now use this asynchronous method in our **Controller** action by changing the **IActionResult** to **async Task<IActionResult>** and awaiting the call to the service, as illustrated in Code Listing 26.

Code Listing 26: Calling ListBooksAsync in the controller

```csharp
using BookRepository.Data;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Threading.Tasks;

namespace BookRepository.Controllers
{
    [Route("api/[controller]")]
    public class BookController : ControllerBase
    {
        private readonly IBookData _service;

        public BookController(IBookData service)
        {
            _service = service;
        }

        [HttpGet]
        public async Task<IActionResult> GetBooks()
        {
            try
            {
                var books = await _service.ListBooksAsync();
                return Ok(books);
            }
            catch (Exception)
            {
                return StatusCode(StatusCodes.Status500InternalServerError,
"There was a database failure");
            }
        }
    }
}
```

Calling the API endpoint again will still return the same data from your database, but the call is done asynchronously.

📝 **Note: Be sure to add the** `System.Threading.Tasks` **namespace.**

This whole time, we have just been changing the logic in our controller while the actual API call remains the same.

Returning models instead of entities

There is one more thing that I want to do. I do not want to directly return the entity from my API call. Looking back to Code Listing 23, you will notice that the data returned included the database ID for the book. This is information that I do not want to expose to the user. I can use a model here to control what I return to the user.

> 📝 *Note: Sometimes you will want to filter out data for security reasons too.*

Start by adding a **Models** folder to your **BookRepository** project. Inside this folder, add a new class called **BookModel**, as illustrated in Figure 28.
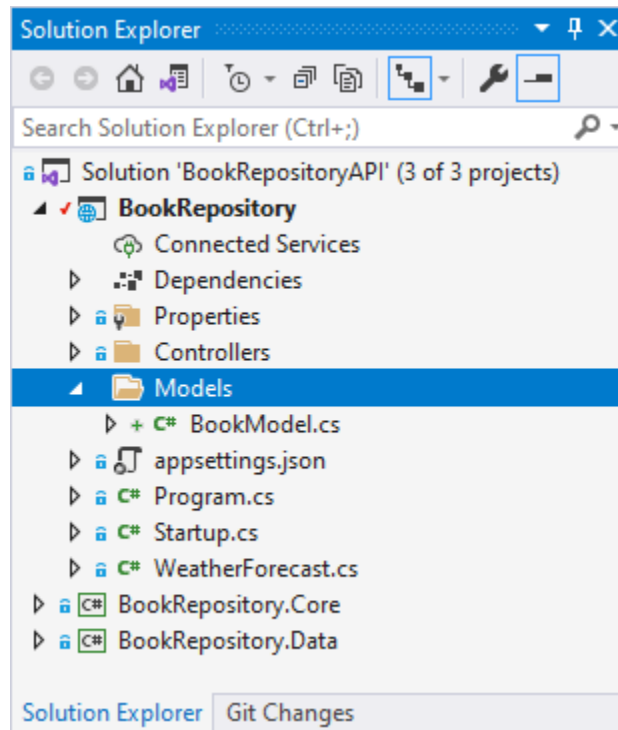
Figure 28: The Models folder and the BookModel class

The code for **BookModel** is almost the same as the **Book** entity. The only difference here is that **BookModel** does not have a property for the **Id**.

Code Listing 27: The BookModel

```
namespace BookRepository.Models
{
    public class BookModel
    {
        public string ISBN { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
```

```
        public string Publisher { get; set; }
        public string Author { get; set; }
    }
}
```

This is what we want. We do not want to return the whole **Book** entity to the user, and a model allows us to filter the data that is returned to the user.

The next change that we need to do is modify our API action on the **BookController** to return the **BookModel** instead of the **Book** entity. Now the code illustrated in Code Listing 28 is perfectly valid code. There are, however, much shorter (and in my opinion, better) ways to do this mapping between the **Book** entity and the **BookModel** model. There is a very good tool called AutoMapper that will automatically map classes.

You can find AutoMapper on [NuGet](#).

I decided that implementing AutoMapper here and going into an explanation of how to set it up and use it was beyond the scope of this book. Just be aware that there are mapping tools such as AutoMapper that can reduce the code illustrated in Code Listing 28 to a few lines.

Code Listing 28: The Modified controller action

```
[HttpGet]
public async Task<ActionResult<List<BookModel>>> GetBooks()
{
    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
                }
                select model).ToList();
    }
    catch (Exception)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, "There
was a database failure");
    }
}
```

Comparing the code for the **GetBooks** action in Code Listing 28 to the code in Code Listing 26, you will notice that I have changed the **GetBooks** action from **public async Task<IActionResult> GetBooks()** to **public async Task<ActionResult<List<BookModel>>> GetBooks()**.

I have changed the **IActionResult** to **ActionResult** that takes the return type **List<BookModel>** as a parameter.

I can now remove the **return Ok(books)** and return the mapped **BookModel** directly. Because this matches the return type of **GetBooks**, a status of 200 will be returned for us.

Calling the API again in Postman, you will see that the book data is returned without the **Id** property for each book, as illustrated in Code Listing 29.

Code Listing 29: The data returned from the BookModel

```
[
    {
        "isbn": "076790818X",
        "title": "A Short History of Nearly Everything",
        "description": "Science has never been more involving or entertaini
ng.",
        "publisher": "Crown",
        "author": "Bill Bryson"
    },
    {
        "isbn": "0465030335",
        "title": "Letters to a Young Contrarian",
        "description": "In the book that he was born to write, provocateur
and best-
selling author Christopher Hitchens inspires future generations of radicals
, gadflies, mavericks, rebels, angry young (wo)men, and dissidents.",
        "publisher": "Basic Books",
        "author": "Christopher Hitchens"
    },
    {
        "isbn": "9780062316110",
        "title": "Sapiens",
        "description": "One hundred thousand years ago, at least six differ
ent species of humans inhabited Earth. Yet today there is only one—
homo sapiens. What happened to the others?",
        "publisher": "Harper Perennial",
        "author": "Yuval Noah Harari"
    }
]
```

We have now filtered out data and given the user only the data that we wanted to expose to them.

Returning a single item

While I do not want to display a book ID to the user, I do want to use that ID so that I can return a single book from my API. Doing this is easy. As before, I will be modifying my interface to add an asynchronous call to the database to return a single book based on its **Id**.

Modify your interface as illustrated in Code Listing 30.

Code Listing 30: The IBookData interface

```
using BookRepository.Core;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace BookRepository.Data
{
    public interface IBookData
    {
        IEnumerable<Book> ListBooks();
        Task<IEnumerable<Book>> ListBooksAsync();
        Book GetBook(int Id);
        Task<Book> GetBookAsync(int Id);
        Book UpdateBook(Book bookData);
        Book AddBook(Book newBook);
        int Save();
    }
}
```

Implement that method on your **SqlData** class, as illustrated in Code Listing 31.

Code Listing 31: The GetBookAsync implementation in SqlData

```
public async Task<Book> GetBookAsync(int Id)
{
    return await _database.Books.FindAsync(Id);
}
```

In your controller (Code Listing 32), add a new action called **GetBook** and include the **HttpGet** attribute. In addition to **HttpGet**, include a route value called **Id**. This will bind the route value to the **Id** parameter passed to the **GetBook** action, and it will look for it after the slash of the **[controller]** section in the controller route **api/[controller]**.

```csharp
[HttpGet("{Id}")]
public async Task<ActionResult<BookModel>> GetBook(int Id)
{
    try
    {
        var result = await _service.GetBookAsync(Id);
        return result == null
            ? NotFound($"The book with ID {Id} was not found")
            : new BookModel()
        {
            Author = result.Author,
            Description = result.Description,
            Title = result.Title,
            Publisher = result.Publisher,
            ISBN = result.ISBN
        };
    }
    catch (Exception)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, "There
was a database failure");
    }
}
```

The action then calls the **GetBookAsync** method on the data service, and if the result is **null**, it will return a **NotFound** status.

Run your API and in Postman make a **GET** request to the URL **https://localhost:44371/api/book/1**, replacing the port **44371** that I am using with the port you are using.

Code Listing 33: The book result returned

```json
{
    "isbn": "0465030335",
    "title": "Letters to a Young Contrarian",
    "description": "In the book that he was born to write, provocateur and
best-
selling author Christopher Hitchens inspires future generations of radicals
, gadflies, mavericks, rebels, angry young (wo)men, and dissidents.",
    "publisher": "Basic Books",
    "author": "Christopher Hitchens"
}
```

Looking back to [Figure 23](#) in Chapter 1, you will see that this is the book with **Id = 1** that I added to my database table. Changing your API call to specify an invalid book **Id** (999 for example), the API will return a Not Found status, as illustrated in Figure 29.
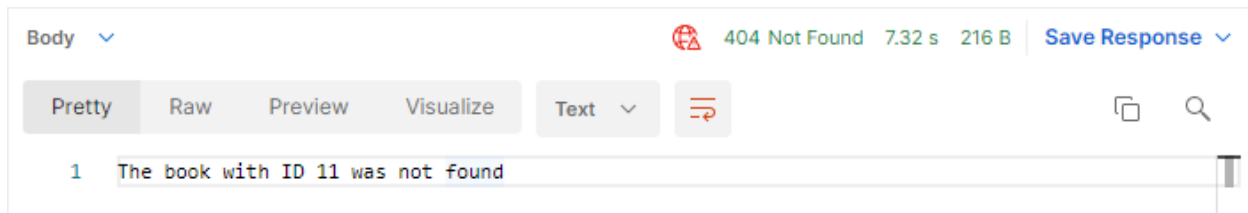

Figure 29: Book not found response

If you do get a valid book returned with **Id = 999**, I will have to congratulate you on your persistence in adding all those books. Try looking for **-1** instead to see the **NotFound** return status.

Searching data

I want to be able to give the user the ability to search books based on the ISBN. This will allow the user to find specific books because ISBNs are unique, and if you have the exact ISBN, you will find the book that you are looking for. I also want the user to be able to find books if they only enter a part of the ISBN. The API should return all books that have ISBNs starting with the number they type.

To enable searching, we will use query strings and the query string will be mapped to the **Isbn** parameter on the **SearchIsbn** action, which is illustrated in Code Listing 34. We are also telling the API that any URLs that come to the API with the word **search** after the controller section in the route must be handled by this action called **SearchIsbn**.

The **SearchIsbn** action simply does the call to the **ListBooksAsync** method on our data service, and then filters the returned book list for all books that have ISBNs starting with the value we supplied in the query string.

Code Listing 34: Adding the SearchIsbn action

```
[HttpGet("search")]
public async Task<ActionResult<List<BookModel>>> SearchIsbn(string Isbn)
{
    try
    {
        var books = await _service.ListBooksAsync();
        var results = books.Where(b => b.ISBN.StartsWith(Isbn));
        return !results.Any()
            ? NotFound()
            : (from book in results
                let model = new BookModel()
                {
```

```
                Author = book.Author,
                Description = book.Description,
                Title = book.Title,
                Publisher = book.Publisher,
                ISBN = book.ISBN
            }
            select model).ToList();
    }
    catch (Exception)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, "There
was a database failure");
    }
}
```

If no books are found, then we simply return a **NotFound** status. Assuming that you have a book with an ISBN of **076790818X** in your table, run the API and call the book search using the URL **https://localhost:44371/api/book/search?Isbn=076790818X**.

Seeing as my table does have a book with the ISBN 076790818X, the API will return that specific book result, as seen in Code Listing 35.

Code Listing 35: The result for ISBN 076790818X

```
[
    {
        "isbn": "076790818X",
        "title": "A Short History of Nearly Everything",
        "description": "Science has never been more involving or entertaini
ng.",
        "publisher": "Crown",
        "author": "Bill Bryson"
    }
]
```

If, however, I am unsure of the ISBN and only know that it starts with a **0**, I can perform a book search using the URL **https://localhost:44371/api/book/search?Isbn=0** instead.

Code Listing 36: The result for book ISBNs starting with 0

```
[
    {
        "isbn": "076790818X",
        "title": "A Short History of Nearly Everything",
        "description": "Science has never been more involving or entertaini
ng.",
        "publisher": "Crown",
```

```
            "author": "Bill Bryson"
        },
        {
            "isbn": "0465030335",
            "title": "Letters to a Young Contrarian",
            "description": "In the book that he was born to write, provocateur
and best-
selling author Christopher Hitchens inspires future generations of radicals
, gadflies, mavericks, rebels, angry young (wo)men, and dissidents.",
            "publisher": "Basic Books",
            "author": "Christopher Hitchens"
        }
]
```

In a production system, searching for an ISBN starting with **0** is not going to be useful at all (it will return too many books), but the logic here is clear. I have only three books in my database, so doing this search is easy enough to illustrate the point.

As homework, why don't you try and create a search for other book properties? Add in several books from the same author and do an author search. Your API code can be extended easily to accommodate this type of search.

Let's see how to modify data in the next chapter.

Chapter 3 Modifying Data with Your API

APIs would not be much use if they couldn't allow the modification of data to occur. We saw in the previous chapter that we can precisely control what the user sees by returning a model instead of an entity during a **GET** request. As is the case here, when modifying data, you are in control, and you can expose that functionality to the consumer of your API.

There are other HTTP verbs to accommodate the modification of data. The verbs you would commonly use with APIs are:

- **GET**: Read data from a resource.
- **POST**: Create a new resource.
- **PUT**: Update an existing resource.
- **PATCH**: Make a partial update on a resource.
- **DELETE**: Delete a resource.

You would need to think about how you would allow something such as a **DELETE**, for example. As with a **PUT** or a **PATCH**, you would most likely supply an ID of some kind when doing a **DELETE**. You would not want to allow the consumer of your API to delete all the resources at once.

Add entities using POST

As I mentioned in Chapter 1, the interface in our API will be changing somewhat throughout the development of the API. I want the ability to create a new book in my book repository, but later on, I might want to add other entity types. I not only want EF to add my entity to the **BookRepoDbContext**, but to save the entity, too. I also want the inserted ID of the entity returned to me.

Code Listing 37: Added SaveAsync in the interface

```
using BookRepository.Core;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace BookRepository.Data
{
    public interface IBookData
    {
        IEnumerable<Book> ListBooks();
        Task<IEnumerable<Book>> ListBooksAsync();
        Book GetBook(int Id);
        Task<Book> GetBookAsync(int Id);
        Book UpdateBook(Book bookData);
        void AddBook(Book newBook);
        int Save();
        Task<int> SaveAsync<T>(T entity);
```

```
    }
}
```

For this reason, I will modify my interface (Code Listing 37) to specify that a **SaveAsync** method must be implemented that takes a generic type **T** and returns an integer. I must then provide the implementation for this in the **SqlData.cs** class, as illustrated in Code Listing 38.

Code Listing 38: The SaveAsync implementation in SqlData

```csharp
public async Task<int> SaveAsync<T>(T entity)
{
    var addedEntity = _database.Add(entity);
    var entityId = -1;

    if (await _database.SaveChangesAsync() > -1)
    {
        entityId =
Convert.ToInt32(addedEntity.Property("Id").CurrentValue);
    }

    return entityId;
}
```

This **SaveAsync** method will accept an entity, add that entity to the **BookRepoDbContext**, and then attempt to save the entity to the database. I can then grab the inserted **Id** field value because I know that my tables in the database will always have an **Id** field.

> 📝 *Note: There are better ways to do the SaveAsync—specifically, surrounding the return of the inserted Id field. Because I am using a hard-coded string value for the property name, casing matters. Therefore, if you specify ID and the column on the table is called Id, you will receive an error. This book is, however, not a book on EF Core. I want you to get the most out of this book concerning APIs. I'll leave the exact mechanics of the BookRepository.Data project up to you to fine-tune.*

Entity Framework will provide this ID to you after the **SaveChangesAsync** method has been successfully called. Back in the **BookController**, we will be utilizing model binding to bind the JSON passed to the endpoint to our **BookModel**. To enable this, add the attribute **[ApiController]** to your **BookController** class, as illustrated in Code Listing 39.

Code Listing 39: The Added ApiController attribute

```csharp
namespace BookRepository.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BookController : ControllerBase
    {
```

This allows our controller to act as an API. We are telling .NET Core a lot about what we expect from this controller. It will, therefore, attempt to do model binding for us.

Code Listing 40: The Post API method

```csharp
public async Task<ActionResult<BookModel>> Post(BookModel model)
{
    try
    {
        var entityLocation = "";

        var entity = new Book()
        {
            Author = model.Author,
            Description = model.Description,
            Title = model.Title,
            Publisher = model.Publisher,
            ISBN = model.ISBN
        };

        var createdBookId = await _service.SaveAsync(entity);
        if (createdBookId > 0)
        {
            entityLocation = _linkGenerator.GetPathByAction("GetBook",
"Book", new { Id = createdBookId });
            return Created(entityLocation, model);
        }
    }
    catch (Exception)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, "There
was a database failure");
    }

    return BadRequest();
}
```

As seen in Code Listing 40, create an API method that will accept our **POST** request.

📝 *Note: The `_linkGenerator` will show a red squiggly line. I'll get back to that shortly.*

The code is a lot like before with regards to the model, but this time in reverse. The **SaveAsync** method accepts an entity of type **T**, but we know that we need to pass the database a **Book** entity.

Because the **SaveAsync** method accepts a parameter of type **T**, it is quite easy to pass the model instead of the entity which will throw an exception. This seems like a job for an interface and a constraint. I do not want to allow the accidental passing of a model to the **SaveAsync** method. It must always take an entity as a parameter.

Create a new interface in the BookRepository.Core project. Call this interface **IEntity** and give it a property called **Id**. I want to require all entities that implement **IEntity** to contain a property called **Id**. Not **ID** or **RecordId**, but **Id**. See where I'm going with this?

The code in Code Listing 41 for the **IEntity** interface is really simple.

Code Listing 41: The IEntity interface

```
namespace BookRepository.Core
{
    public interface IEntity
    {
        public int Id { get; set; }
    }
}
```

Implement this **IEntity** interface on the **Book** entity, as seen in Code Listing 42.

Code Listing 42: The Book entity implementing IEntity

```
namespace BookRepository.Core
{
    public class Book : IEntity
    {
        public int Id { get; set; }
        public string ISBN { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public string Publisher { get; set; }
        public string Author { get; set; }
    }
}
```

Swing back to the **IBookData** interface and add a constraint on the **SaveAsync** generic method to tell it to only accept types that implement **IEntity** (Code Listing 43).

Code Listing 43: Added constraint on SaveAsync

```
using BookRepository.Core;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace BookRepository.Data
{
```

```csharp
    public interface IBookData
    {
        IEnumerable<Book> ListBooks();
        Task<IEnumerable<Book>> ListBooksAsync();
        Book GetBook(int Id);
        Task<Book> GetBookAsync(int Id);
        Book UpdateBook(Book bookData);
        void AddBook(Book newBook);
        int Save();
        Task<int> SaveAsync<T>(T entity) where T : IEntity;
    }
}
```

Lastly, this needs to be implemented in the **SqlData** class on the **SaveAsync** method, as seen in Code Listing 44.

Code Listing 44: Constraint implemented in SqlData class

```csharp
public async Task<int> SaveAsync<T>(T entity) where T : IEntity
{
    var addedEntity = _database.Add(entity);
    var entityId = -1;

    if (await _database.SaveChangesAsync() > -1)
    {
        entityId =
Convert.ToInt32(addedEntity.Property("Id").CurrentValue);
    }

    return entityId;
}
```

The code we wrote for the **Post** method on our controller (as illustrated in Code Listing 40) is now much more fault-tolerant. We can only pass the data service entities to perform data modifications on. The last bit that we need to talk about is the **_linkGenerator** (which you undoubtedly see underlined with a red squiggly line in your code editor).

Because I have the **Id** of the created entity being returned to me, I want some way to tell the consumer of this API where to find this created resource. This is done with a **LinkGenerator**, which is in the **Microsoft.AspNetCore.Routing** namespace.

Code Listing 45: The modified BookController constructor

```csharp
private readonly IBookData _service;
private readonly LinkGenerator _linkGenerator;

public BookController(IBookData service, LinkGenerator linkGenerator)
{
```

```
    _service = service;
    _linkGenerator = linkGenerator;
}
```

Modify the controller's constructor (Code Listing 45) to take a **LinkGenerator** as a parameter and save that off to a field called **_linkGenerator**. Back in the **Post** method, you can use the **GetPathByAction** method of the **_linkGenerator** to return the location of the created entity. Consider the code snippet in Code Listing 46.

Code Listing 46: The getPathByAction code snippet

```
entityLocation = _linkGenerator.GetPathByAction("GetBook", "Book", new { Id
= createdBookId });
```

I am telling the **_linkGenerator** to create a valid link to the resource I have just created with the **Id** returned from the **SaveAsync** method. This resource can be found by doing a **GET** request (which calls the **GetBook** action) on the **BookController**.

When this link has been generated, return a **Created** response specifying the entity location and the model that was used to create the entity.
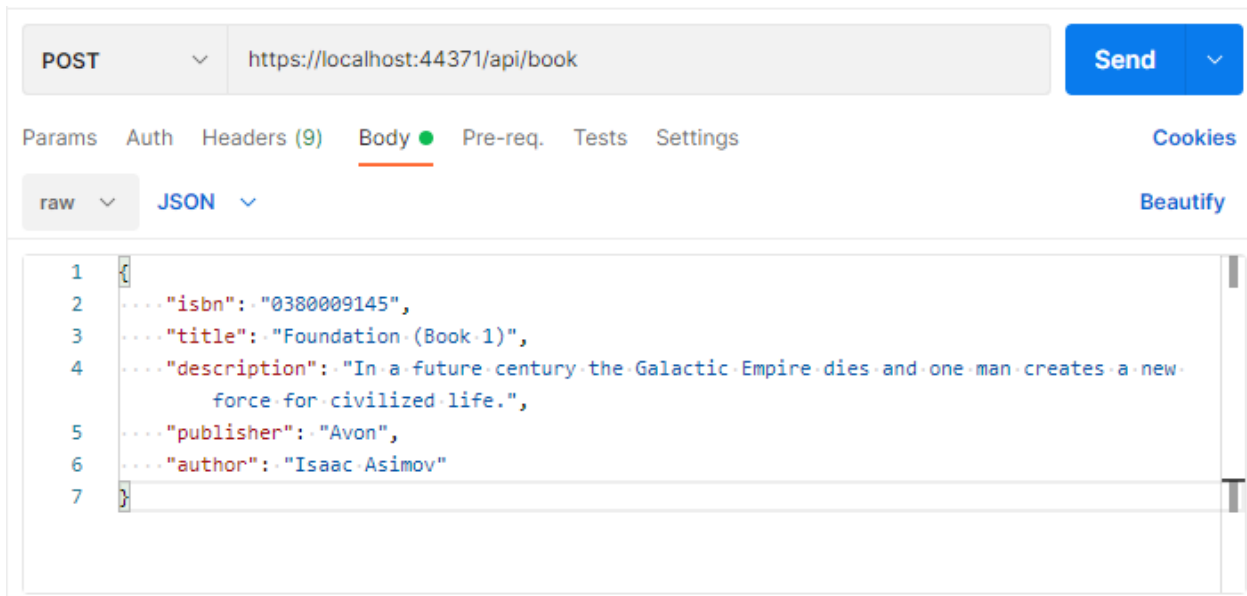

Figure 30: The Postman POST

We can now use Postman to create a new book for us. As seen in Figure 30, create a **POST** with the URL endpoint of **/api/book** and specify that you will be sending raw JSON in the body. Format the body as JSON to contain the details of the book that you want to create and click **Send**.

Figure 31: The 201 Created response specifying the location

As seen in Figure 31, Postman returned a **201 Created** response and includes the location of the created book for us. In this case, it is located at **/api/Book/13**, which means that the inserted **Id** in the table is **13**. Your **Id** returned here will differ from mine.

Go ahead and perform a **GET** request with Postman for your book **Id** returned in the **POST** you performed. You will see the newly created book returned from the **GET** request.

Performing model validation

One more thing that I want to do is ensure that any books added contain at least an ISBN and a title. Without these, the **POST** should fail.

Code Listing 47: Adding required attributes on BookModel

```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; }
    }
}
```

As seen in Code Listing 47, modify the code in the **BookModel** and add the **[Required]** attribute. You will also need to add the **System.ComponentModel.DataAnnotations** namespace.
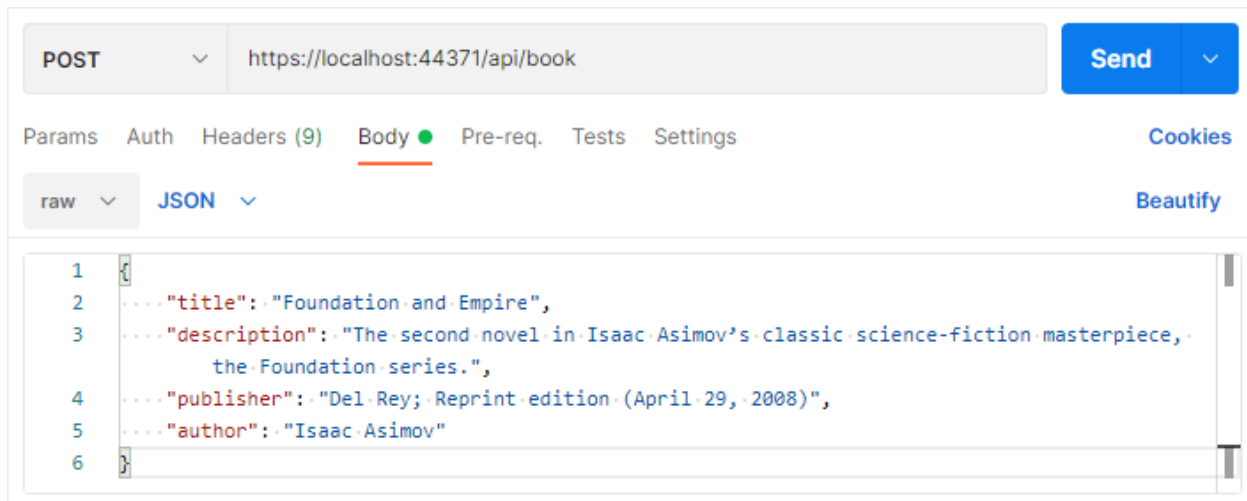

Figure 32: Posting an invalid BookModel

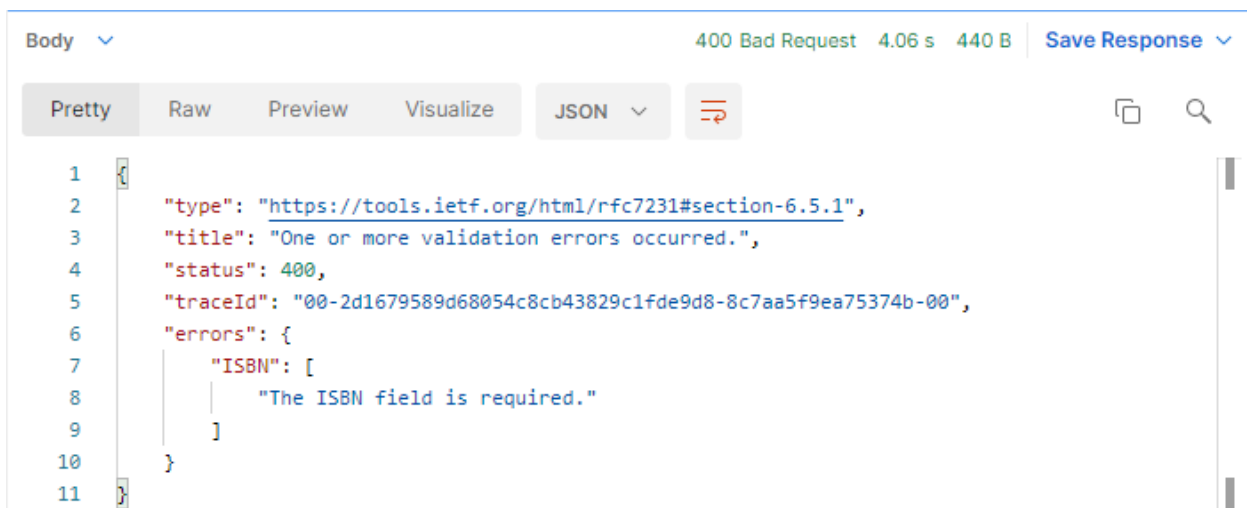In Postman, perform a **POST** for an invalid **BookModel** without an ISBN, as seen in Figure 32.

Figure 33: The model validation message for the bad request

Postman reports an error from the API, as seen in Figure 33.

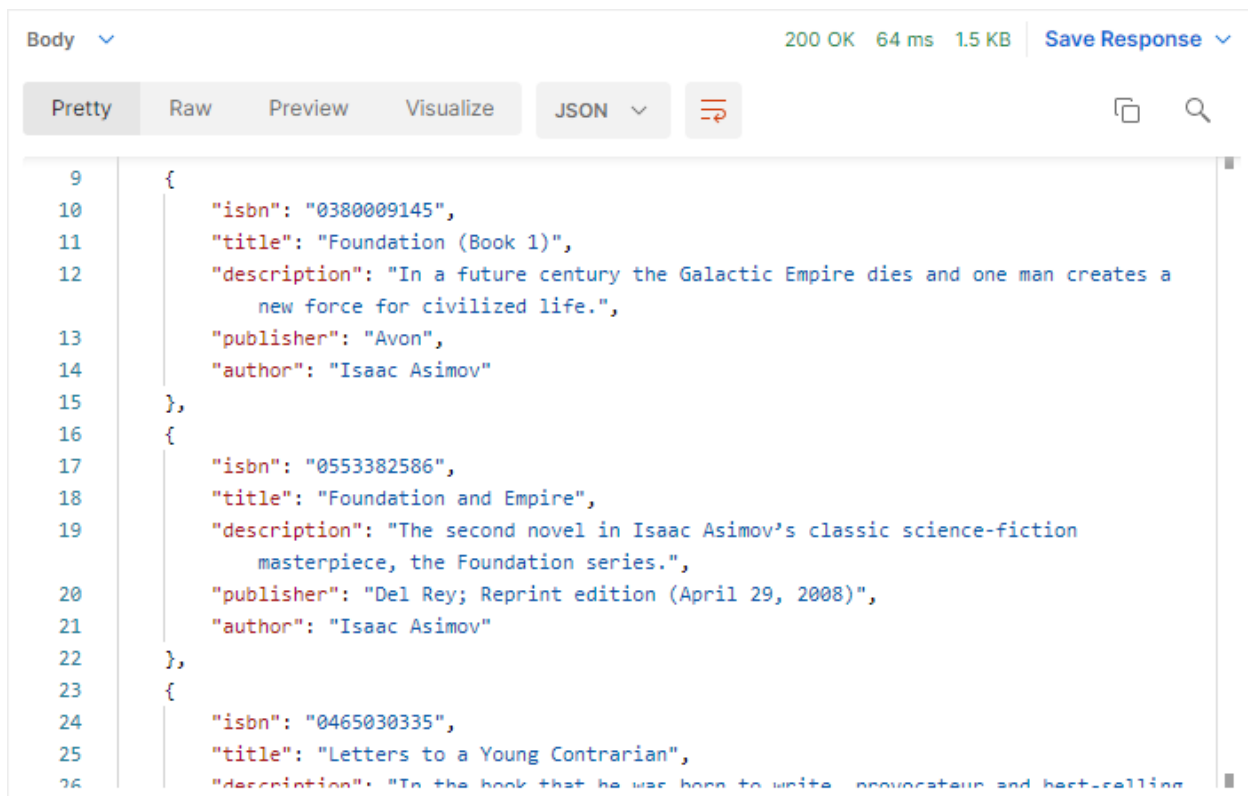💡 **Tip: You would probably also want to add some validation in your *POST* to ensure that the ISBN you are adding is unique. This can be done in the *POST* method on your *BookController*.**

With a simple attribute added to the **BookModel**, we can specify requirements for the models we use in the API. For more information on the other data annotations available, have a look at the official Microsoft documentation.

After fixing the body of the data you are posting (by adding in an ISBN), create the book via Postman. You will see that because you have specified the ISBN, your book is correctly created, and the location returned in the header of the API call in Postman. Make a note of this created book **Id**. We will use this to update the book in the next section using **PUT**.

Change entities using PUT

I enjoy reading, and I want my book repository to be as complete as possible. I have added some more books to my book repository using **POST**, as detailed earlier in this chapter. I do, however, notice that I have made a mistake on the last book that I have added. The book **Foundation and Empire** (as seen in Figure 34) should include the text **Book 2** in the title.

Figure 34: The list of books

We need to allow the updating of an entity by implementing the **PUT** verb. The code to do this is quite simple, as seen in Code Listing 48. I want to create an action decorated with the **HttpPut** attribute and tell it to expect the **Id** of a book. I also want to pass the entity to update. The code then tries to find an existing book with the **Id** we specified. This can just be done by calling the **GetBookAsync** method on the data service.

If no such book is found, then we must return a **NotFound**, as this is the most appropriate response in this case. If we do find a book with the given **Id**, then we need to apply the changes to the entity and call **UpdateAsync** on the data service.

Code Listing 48: Implementing the PUT verb

```
[HttpPut("{Id}")]
public async Task<ActionResult<BookModel>> Put(int Id, BookModel model)
{
    try
    {
        var bookToUpdate = await _service.GetBookAsync(Id);

        if (bookToUpdate != null)
        {
            bookToUpdate.Author = model.Author;
            bookToUpdate.Description = model.Description;
            bookToUpdate.Title = model.Title;
            bookToUpdate.Publisher = model.Publisher;
            bookToUpdate.ISBN = model.ISBN;

            return await _service.UpdateAsync(bookToUpdate) ? model :
BadRequest();
        }
        else
        {
            return NotFound($"Can't find book with Id {Id}");
        }
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, $"There
was a database failure: {ex.Message}");
    }
}
```

It is then the entity that we want to update that is passed to the data service's **UpdateAsync** method, as seen in Code Listing 49.

Code Listing 49: The UpdateAsync data service method

```
public async Task<bool> UpdateAsync<T>(T entity) where T : IEntity
{
    var updatedEntity = _database.Attach(entity);
    updatedEntity.State = EntityState.Modified;

    return (await _database.SaveChangesAsync() > 0);
}
```

If the update was successful, I will just return the mode. Alternatively, I will return a **BadRequest**. With this **PUT** method added, modify the book with the correct details by creating a **PUT** request in Postman using the URL that includes the **Id** of the book to update (and in your case, the **Id** you made a note of earlier).

The URL will be something similar to **https://localhost:44371/api/book/1021**, where **1021** at the end of the URL will be replaced with your book **Id**.
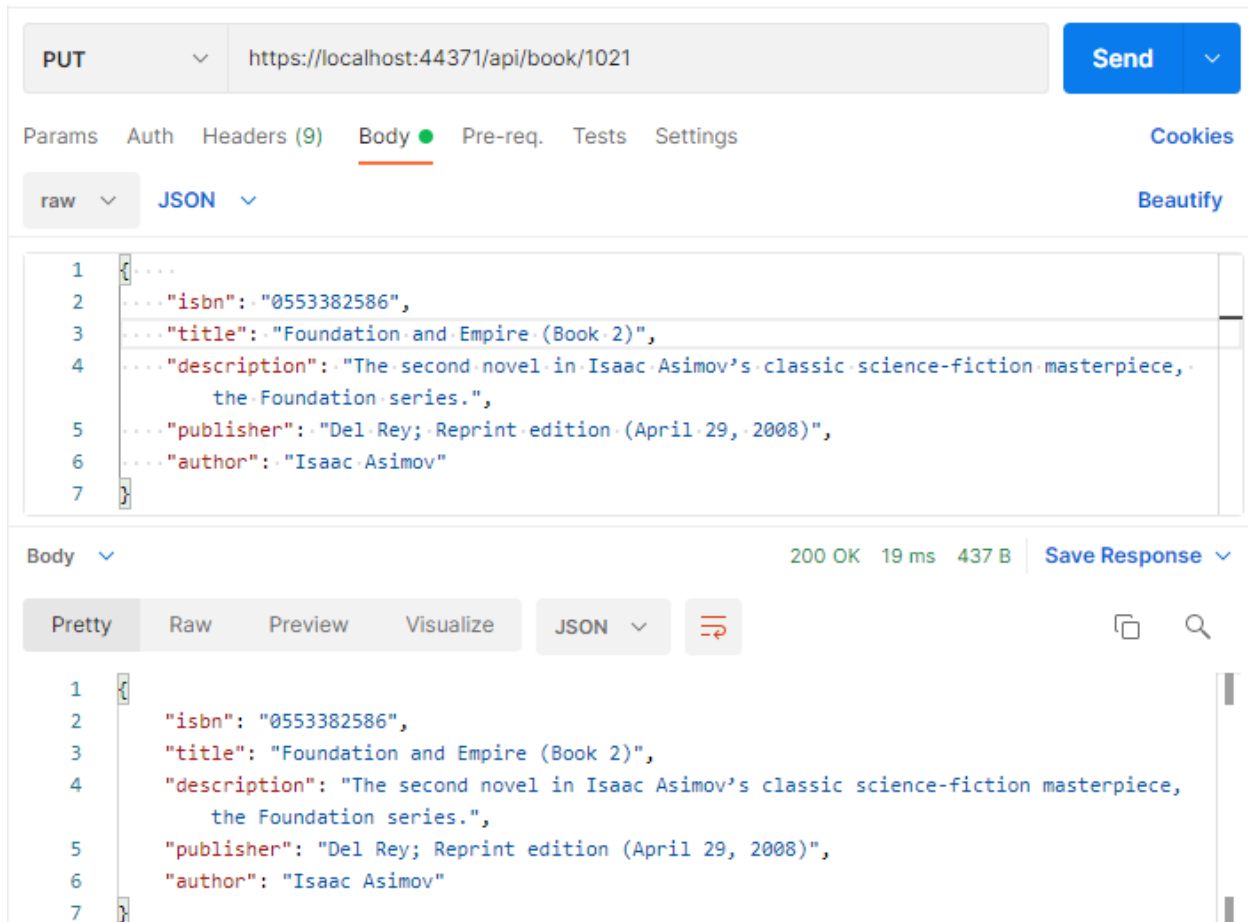
Figure 35: The PUT to update a book in Postman

To verify that the update succeeded, perform a **GET** with the URL **https://localhost:44371/api/book** and make a note of the books returned. You could also just do a **GET** for the specific book **Id** you updated. In my example, I would do a **GET** request in Postman using the URL **https://localhost:44371/api/book/1021**, but you would just use your specific book **Id** instead of mine.

Remove entities using DELETE

The last thing I want to add to my API is the ability to delete a book. By this time, you should have a few books in your book repository. There are perhaps some books that are no longer in your library, or perhaps you added a book with incorrect data that you do not wish to update.

Deleting a book is easily implemented. I have added **DeleteAsync** to my **IBookData** interface and implemented it in my **SqlData** service, as seen in Code Listing 50.

Code Listing 50: The DeleteAsync data service method

```
public async Task<bool> DeleteAsync<T>(T entity) where T : IEntity
{
    var updatedEntity = _database.Remove(entity);
    updatedEntity.State = EntityState.Deleted;

    return (await _database.SaveChangesAsync() > 0);
}
```

Back in the **BookController**, I have added a **Delete** action that expects the book **Id** of the book you want to delete (Code Listing 51). Notice how we don't pass it a model in this instance.

Code Listing 51: The Delete action in BookController

```
[HttpDelete("{Id}")]
public async Task<IActionResult> Delete(int Id)
{
    try
    {
        var bookToDelete = await _service.GetBookAsync(Id);

        return bookToDelete != null
            ? await _service.DeleteAsync(bookToDelete) ? Ok() :
BadRequest()
            : NotFound($"Can't find book with Id {Id}");
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, $"There
was a database failure: {ex.Message}");
    }
}
```

I perform the same **GetBookAsync** call to my data service as in the **PUT**, but this time if I find a book, I just call the **DeleteAsync** on my data service. If the delete works, I just return **Ok**; otherwise, a **BadRequest** is returned. If the book with that **Id** is not found, I return a **NotFound**, which is exactly what we want.

Lastly, I add the **HttpDelete** attribute to the **DELETE** action.
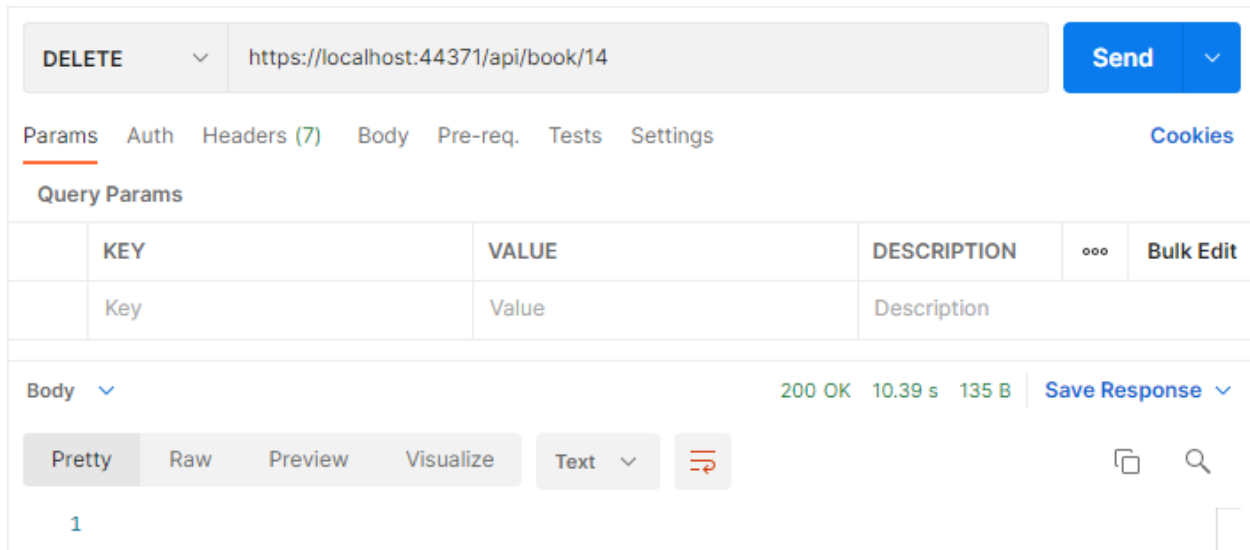
Figure 36: The DELETE in Postman

In Postman I create a **DELETE** and give it the URL **https://localhost:44371/api/book/14**, where my book **Id** is **14**. Your book **Id** will differ. This is all that I supply, and when I click **Send**, as seen in Figure 36, Postman just returns a status **200**.

If you do a **GET** for the book **Id** you just deleted, you will see that the book can't be found, as seen in Figure 37 for my book **Id** of **14**.
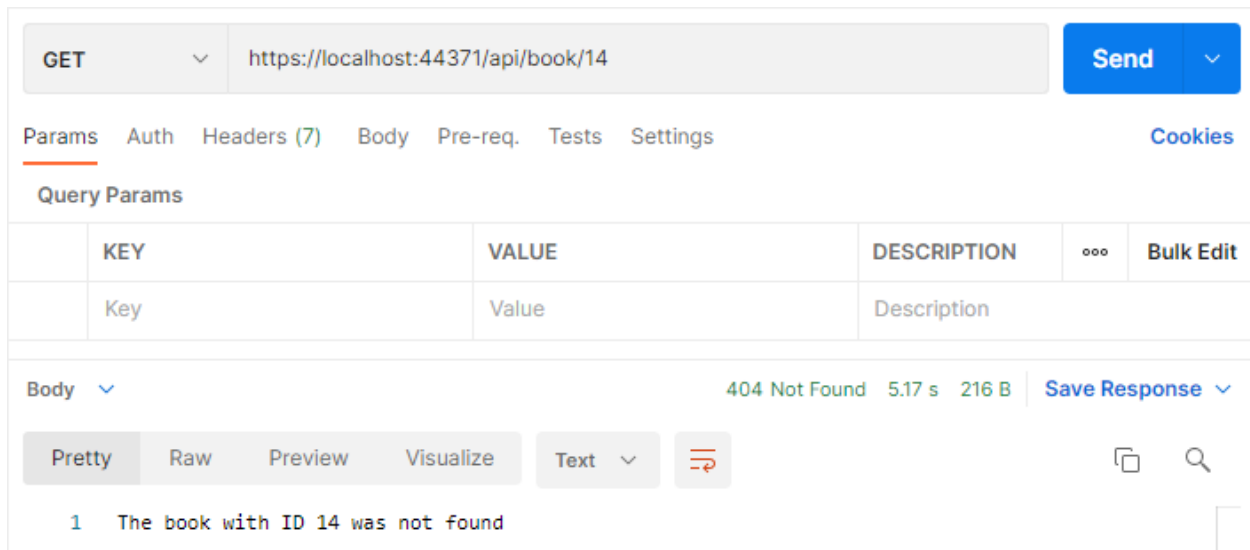
Figure 37: Book Id 14 not found

With the **DELETE** implemented, we have the basics in place for our API. In the next chapter, let's expand a bit on the functionality of the API and add more logic around versioning.

Chapter 4 Versioning Your API

In this chapter, we will be discussing various topics, one of them being versioning your API. Versioning your API is usually a good idea because this lets consumers know what data they can expect, or that they can expect some changes in the data. There are four main types of versioning. They are:

- URI versioning
- Versioning with headers
- Accept header versioning
- Content-type versioning

It does not matter which versioning method you implement in your API; what matters more is that you provide a version. You might be used to seeing an API version in the URI path, as illustrated in Code Listing 52.

Code Listing 52: Example of URI path versioning

```
https://localhost:44371/api/v2/book
```

You will also probably have seen an example of query string versioning, as illustrated in Code Listing 53.

Code Listing 53: Example of URI query string versioning

```
https://localhost:44371/api/book?v=2.0
```

Let's see how we can implement versioning into our book repository API.

Implementing versioning

Versioning does not come out of the box. You need to install a NuGet package, as shown in Figure 38.
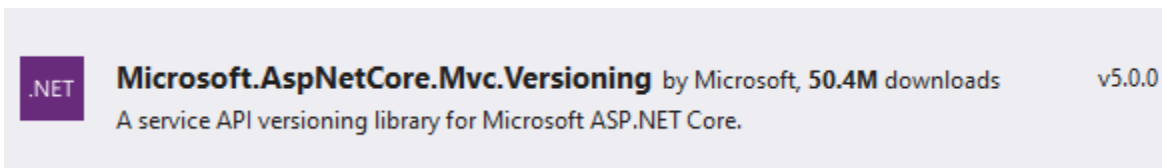

Figure 38: Adding ASP.NET Core versioning NuGet package

Ensure that you add the ASP.NET Core versioning package via NuGet, as this is built to be used with .NET Core, which our API is built on. To allow your API to use versioning, we need to make a change to the **Startup** class in the **ConfigureServices** method, as illustrated in Code Listing 54.

```
services.AddApiVersioning();
```

With this in place, run your API and call one of your API endpoints. For example, just try to call **https://localhost:44371/api/book** and have a look at the return, as illustrated in Code Listing 55.

Code Listing 55: Versioning required

```
{
    "error": {
        "code": "ApiVersionUnspecified",
        "message": "An API version is required, but was not specified.",
        "innerError": null
    }
}
```

You will receive a status 400 bad request error from the API. You must specify an API version, and this is the functionality we expect. Modify the URL that you call in Postman as follows and do the same call again: **https://localhost:44371/api/book?api-version=1.0**. This time, the call to your API will work because you specified the version in the query string, and .NET Core assumes that by default, the whole API is version 1.0.

Try and call **https://localhost:44371/api/book?api-version=1.1**, and you will see that you will receive an error again, as illustrated in Code Listing 56.

Code Listing 56: Default version 1.1 not supported

```
{
    "error": {
        "code": "UnsupportedApiVersion",
        "message": "The HTTP resource that matches the request URI
'https://localhost:44371/api/book' does not support the API version
'1.1'.",
        "innerError": null
    }
}
```

The reason for this is that ASP.NET Core expects the default version to be **1.0** because we have not specified a version for this API yet. As with almost anything in .NET Core, you can change this behavior by modifying the code we added in the **ConfigureServices** method in the **Startup** class.

As seen in Code Listing 57, modify **AddApiVersioning()** to include options and give it the default version that this API must accept.

```
services.AddApiVersioning(o =>
{
    o.DefaultApiVersion = new ApiVersion(1, 1);
});
```

If you call **https://localhost:44371/api/book?api-version=1.1**, you will see that the call will work. Calling **https://localhost:44371/api/book?api-version=1.0** will now fail because we have changed the default version to **1.1** for the API.

Make another small change to **AddApiVersioning()**, as illustrated in Code Listing 58.

Code Listing 58: Report supported API versions

```
services.AddApiVersioning(o =>
{
    o.DefaultApiVersion = new ApiVersion(1, 1);
    o.ReportApiVersions = true;
});
```

By adding **ReportApiVersions**, when you make a call to the API, the header information will return the supported API versions, as seen in Figure 39.

Body	Cookies	Headers (6)	Test Results
KEY			**VALUE**
Transfer-Encoding ⓘ			chunked
Content-Type ⓘ			application/json; charset=utf-8
Server ⓘ			Microsoft-IIS/10.0
api-supported-versions ⓘ			1.1
X-Powered-By ⓘ			ASP.NET
Date ⓘ			Fri, 03 Dec 2021 05:04:20 GMT

Figure 39: Supported API versions returned in header

We have just been tinkering with the default versioning behavior here. Let's see how to version our actions in the next section.

Version actions

Starting at the **BookController**, I want to tell it something about the versions that it must use. I want it to support versions **1.0** and **1.1** of the API. Modify your **BookController** as illustrated in Code Listing 59.

Code Listing 59: Supporting versions 1.0 and 1.1

```
[Route("api/[controller]")]
[ApiVersion("1.0")]
[ApiVersion("1.1")]
[ApiController]
public class BookController : ControllerBase
```

Next, I want to duplicate the **GetBooks** method. Make a copy of the **GetBooks** method and add the attribute **[MapToApiVersion("1.0")]** to the first one. Then, on the **ISBN** property of the **BookModel** being returned, append the text **"- for version 1.0"**.

Do the same for the second **GetBooks** method, but rename the method to **GetBooks_1_1** and give it the attribute **[MapToApiVersion("1.1")]**. Append the text **"- for version 1.1"** to the **ISBN** property of the **BookModel** being returned.

You can see the code for this change in Code Listing 60.

Code Listing 60: Version 1.0 and 1.1 of GET

```
[HttpGet]
[MapToApiVersion("1.0")]
public async Task<ActionResult<List<BookModel>>> GetBooks()
{
    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 1.0"
                }
                select model).ToList();
    }
    catch (Exception)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, "There
was a database failure");
    }
```

```
}

[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 your API and call version **1.0** by supplying the URL
https://localhost:44371/api/book?api-version=1.0 in Postman. You will see the data
returned is coming from version **1.0** of the **GET** method as illustrated in Code Listing 61.

Code Listing 61: Calling Version 1.0 of the GET

```
{
    "isbn": "076790818X - for version 1.0",
    "title": "A Short History of Nearly Everything",
    "description": "Science has never been more involving or
entertaining.",
    "publisher": "Crown",
    "author": "Bill Bryson"
}
```

Next, change the URL to call version **1.1** by calling
https://localhost:44371/api/book?api-version=1.1 in Postman. The returned data will
be coming from version **1.1** of the **GET** method, as seen in Code Listing 62.

```
{
    "isbn": "076790818X - for version 1.1",
    "title": "A Short History of Nearly Everything",
    "description": "Science has never been more involving or
entertaining.",
    "publisher": "Crown",
    "author": "Bill Bryson"
}
```

But what will happen if you call the endpoint without a version, like this:
https://localhost:44371/api/book?

Your API will return the same error as seen in Code Listing 55. We can tell our API to assume the default version (which is version **1.1** in this case) when no version is supplied by modifying the **ConfigureServices** method, as seen in Code Listing 63.

Code Listing 63: Assume a default version

```
services.AddApiVersioning(o =>
{
    o.AssumeDefaultVersionWhenUnspecified = true;
    o.DefaultApiVersion = new ApiVersion(1, 1);
    o.ReportApiVersions = true;
});
```

If you call the URL **https://localhost:44371/api/book** again, you will see that the return data has returned version **1.1** of the **GET** method, as previously illustrated in Code Listing 62. This way you can finely control what data the consumers of the API will see when they call specific versions of your API. Next, we will look at supporting a new version of the API controller.

Versioning controllers

Create a copy of the **BookController** class and call it **Book2Controller**, as illustrated in Figure 40.
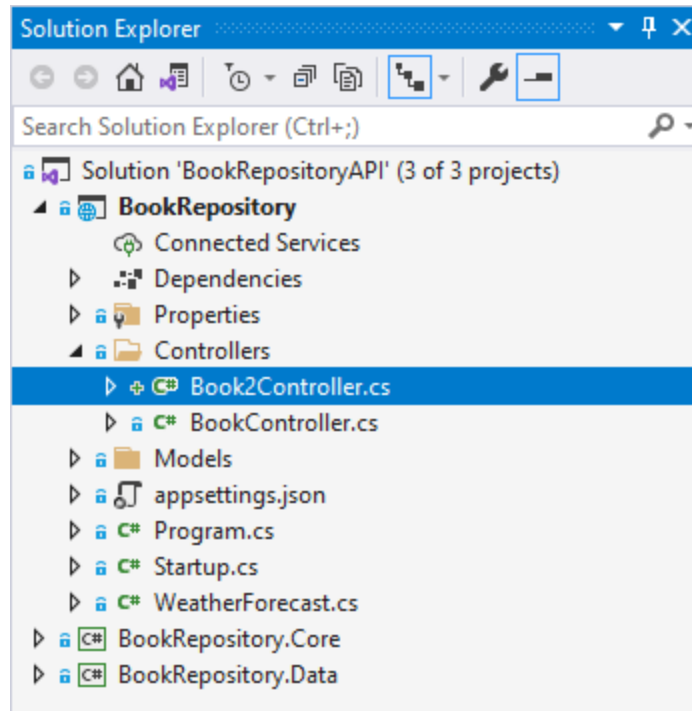
Figure 40: Adding version 2.0 of the BookController

Get rid of the **GetBooks_1_1** method and focus on the **GetBooks** method. Remove the **[MapToApiVersion("1.0")]** attribute on the **GetBooks()** method.

📝 ***Note: You can leave the rest of the class as is, since we will only be focusing on the*** *GetBooks* ***endpoint for this section.***

Modify your **Book2Controller** as illustrated in Code Listing 64.

Code Listing 64: Version 2.0 of the BookController

```
[Route("api/[controller]")]
[ApiVersion("2.0")]
[ApiController]
public class Book2Controller : ControllerBase
{
    private readonly IBookData _service;
    private readonly LinkGenerator _linkGenerator;

    public Book2Controller(IBookData service, LinkGenerator linkGenerator)
    {
        _service = service;
        _linkGenerator = linkGenerator;
    }

    [HttpGet]
```

```
    public async Task<IActionResult> GetBooks()
    {
        try
        {
            var books = await _service.ListBooksAsync();

            var results = new
            {
                Count = books.Count(),
                Books = books
            };

            return Ok(results);
        }
        catch (Exception)
        {
            return StatusCode(StatusCodes.Status500InternalServerError,
 "There was a database failure");
        }
    }
```

From this code example, you can see that the API version on the controller has changed to version 2.0 by specifying the attribute **[ApiVersion(“2.0”)]**. I have modified the **GetBooks()** method to return an anonymous type that consists of the books returned and a count of how many books were returned. Because I am returning an anonymous type, I can just change the **GetBooks** method's return type to **Task<IActionResut>**.

Run your API and make the following **GET** request using the URL **https://localhost:44371/api/book?api-version=2.0**. The data returned will be as illustrated in Code Listing 65.

Code Listing 65: GetBooks version 2.0

```
{
    "count": 5,
    "books": [
        {
            "id": 2,
            "isbn": "076790818X",
            "title": "A Short History of Nearly Everything",
            "description": "Science has never been more involving or
 entertaining.",
            "publisher": "Crown",
            "author": "Bill Bryson"
        },
        {
            "id": 13,
            "isbn": "0380009145",
```

```
            "title": "Foundation (Book 1)",
            "description": "In a future century the Galactic Empire dies
and one man creates a new force for civilized life.",
            "publisher": "Avon",
            "author": "Isaac Asimov"
        },
        {
            "id": 1021,
            "isbn": "0553382586",
            "title": "Foundation and Empire (Book 2)",
            "description": "The second novel in Isaac Asimov's classic
science-fiction masterpiece, the Foundation series.",
            "publisher": "Del Rey; Reprint edition (April 29, 2008)",
            "author": "Isaac Asimov"
        },
        {
            "id": 1,
            "isbn": "0465030335",
            "title": "Letters to a Young Contrarian",
            "description": "In the book that he was born to write,
provocateur and best-selling author Christopher Hitchens inspires future
generations of radicals, gadflies, mavericks, rebels, angry young (wo)men,
and dissidents.",
            "publisher": "Basic Books",
            "author": "Christopher Hitchens"
        },
        {
            "id": 3,
            "isbn": "9780062316110",
            "title": "Sapiens",
            "description": "One hundred thousand years ago, at least six
different species of humans inhabited Earth. Yet today there is only one—
homo sapiens. What happened to the others?",
            "publisher": "Harper Perennial",
            "author": "Yuval Noah Harari"
        }
    ]
}
```

Notice the count of books being returned for this **GET** request. Now perform the same **GET** request using the URLs **https://localhost:44371/api/book?api-version=1.1** and **https://localhost:44371/api/book?api-version=1.0**. The data returned will be coming from version 1 of the **BookController**.

Again, as with so much in .NET Core, you can change the behavior. If you do not want to use the **api-version** portion in your API URL, you can modify this in the **Startup** class. Add the **using** statement **Microsoft.AspNetCore.Mvc.Versioning** to the **Startup** class. Next, modify the section of code related to the API versioning in **ConfigureServices**, as seen in Code Listing 66.

Code Listing 66: Specify QueryStringVersionReader

```
services.AddApiVersioning(o =>
{
    o.AssumeDefaultVersionWhenUnspecified = true;
    o.DefaultApiVersion = new ApiVersion(1, 1);
    o.ReportApiVersions = true;
    o.ApiVersionReader = new QueryStringApiVersionReader("v");
});
```

This tells the API that it should expect the version of the API to be denoted by a **v** in the query string. Do a **GET** request using the following URL:
https://localhost:44371/api/book?v=2.0.

The data returned from your API is from version 2.0 of the **BookController** class.

Being able to version your controllers in this way gives you the flexibility to introduce improved code without having to affect the functionality of the API to consumers still using earlier versions.

Versioning with headers

Versioning with headers is also very easy to configure. If, for whatever reason, you can't use query strings for versioning your APIs, you can specify the version of the API you want in the header. Modify the **ConfigureServices** method in the **Startup** for the API versioning as seen in Code Listing 67.

Code Listing 67: Versioning with headers

```
services.AddApiVersioning(o =>
{
    o.AssumeDefaultVersionWhenUnspecified = true;
    o.DefaultApiVersion = new ApiVersion(1, 1);
    o.ReportApiVersions = true;
    o.ApiVersionReader = new HeaderApiVersionReader("x-version");
});
```

All I have done is tell the API to use headers to version the API instead of query strings. I have therefore replaced the **QueryStringApiVersionReader** with the **HeaderApiVersionReader**, which also takes a string parameter **x-version** as a header key to look for. Run the API and head on over to Postman. On the **Params** tab, uncheck the key **v** that was used to denote the version of the API.

Figure 41: Remove the parameter version

On the **Headers** tab, add the header key you defined in Code Listing 67 and specify the version you want as the value.

Figure 42: Specify a version in the header

Now call the book list API by doing a `GET` request for the URL `https://localhost:44371/api/book`, and you will see that the list of books returned is coming from version 2.0 of your controller. By making a small change to the API, we can switch between specifying the version in the query string and specifying the version in the header.

Versioning with headers and query strings

If you want to give the consumers of your API some flexibility when specifying the version they want, you can do this (you guessed it) in the `ConfigureServices` method of the `Startup` class. Consider the code in Code Listing 68.

```
services.AddApiVersioning(o =>
{
    o.AssumeDefaultVersionWhenUnspecified = true;
    o.DefaultApiVersion = new ApiVersion(1, 1);
    o.ReportApiVersions = true;
    o.ApiVersionReader = ApiVersionReader.Combine(
        new HeaderApiVersionReader("x-version")
        , new QueryStringApiVersionReader("version", "ver", "v"));
});
```

Here we are telling the API to accept a version specified in the query string as well as in the header. What's more, I'm telling the API to accept the following query string parameters to look for the version to call:

https://localhost:44371/api/book?v=2.0

https://localhost:44371/api/book?ver=2.0

https://localhost:44371/api/book?version=2.0

If the consumer specifies any of these query string parameters or specifies a header key called **x-version**, the API will use that to call the correct version of the API.

Versioning using the URL

Specifying the version in the URL for the API endpoint you want to use is another technique that many developers use for versioning. I quite like this because the version is enforced as part of the route. Start by modifying the **ConfigureServices** method in the **Startup** class, as seen in Code Listing 69.

Code Listing 69: Specify URL versioning

```
services.AddApiVersioning(o =>
{
    o.AssumeDefaultVersionWhenUnspecified = true;
    o.DefaultApiVersion = new ApiVersion(1, 1);
    o.ReportApiVersions = true;
    o.ApiVersionReader = new UrlSegmentApiVersionReader();
});
```

The next part is a bit of a pain but is required for this versioning method to work. Because we are going to specify the version in the URL, we need to modify the **Route** attribute on all our controllers. Start by modifying the **BookController** and tell it to expect a route that includes the version in the route after the **Api** segment, as seen in Code Listing 70. This controller is specific to version **1.0** and version **1.1** of your API.

Code Listing 70: The modified BookController

```
[Route("api/v{version:apiVersion}/[controller]")]
[ApiVersion("1.0")]
[ApiVersion("1.1")]
[ApiController]
public class BookController : ControllerBase
{
```

Next, modify the **Book2Controller** by modifying its **Route** attribute to expect a route that includes the version of the API, as seen in Code Listing 71.

Code Listing 71: The modified Book2Controller

```
[Route("api/v{version:apiVersion}/[controller]")]
[ApiVersion("2.0")]
[ApiController]
public class Book2Controller : ControllerBase
{
```

With these changes in place, run your API and call the following URLs:

https://localhost:44371/api/v1/book

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

https://localhost:44371/api/v2/book

The API still works and returns the list of books as expected. Some developers might not like this method of versioning APIs because it is less forgiving and less flexible than other methods. Some, however, might like that because specifying the version in the URL clearly states their intent. Whichever method you choose, implementing this in your API is quite simple to do.

Conclusion

In this book, we have discussed the basics of putting together an API for a book repository. There is a lot more to learn and do, but this should give you a good starting point for creating your APIs. Remember that when modifying (or adding) any entities, you will need to run the database migrations (refer to Code Listings 13 and 14). Now that you have a functional API, why not try and add some related data such as author information? Happy coding!