Coding Clean, Reliable, and Safe REST APIs with ASP.NET Core 8

Develop Robust Minimal APIs with .NET 8

Anthony Giretti

Apress®

Coding Clean, Reliable, and Safe REST APIs with ASP.NET Core 8

Develop Robust Minimal APIs with .NET 8

Anthony Giretti

Apress®

Coding Clean, Reliable, and Safe REST APIs with ASP.NET Core 8:

Develop Robust Minimal APIs with .NET 8

Anthony Giretti
La Salle, QC, Canada

Table of Contents

About the Author

Anthony Giretti is a senior developer/architect at Marchex in Toronto, Canada. He appreciates learning and teaching new technologies and has a knack for web technologies (more than 17 years' experience) and a keen interest in .NET. His expertise in development and IT and his passion for sharing his knowledge allow him to deconstruct any web project in order to help other developers achieve their project goals. He loves to deal with performance constraints, high availability, and optimization challenges. Anthony is the author of *Beginning gRPC with ASP.NET Core 6* (Apress), a six-time Microsoft MVP, and a Microsoft Certified Software Developer (MCSD).

About the Technical Reviewer

Fiodar Sazanavets is a Microsoft MVP and a senior software engineer with over a decade of professional experience. He primarily specializes in .NET and Microsoft stack and is enthusiastic about creating well-crafted software that fully meets business needs. He enjoys teaching aspiring developers and sharing his knowledge with the community, which he has done both as a volunteer and commercially. Fiodar has created several online courses, written a number of technical books, and authored other types of educational content. He also provides live mentoring services, both to groups and individuals. Throughout his career, he has built software of various types and various levels of complexity in multiple industries. This includes a passenger information management system for a railway, distributed smart clusters of IoT devices, ecommerce systems, financial transaction processing systems, and more. He has also successfully led and mentored teams of software developers.

Acknowledgments

Completing this book could not have been possible without the participation and assistance of many people, and I would like to express my special thanks to them. First, thanks to my wife, Nadege, who never stopped supporting me. I love you!

Next, I would like to thank the rest of my family for their support.

This book has been written in special conditions since I was hospitalized for a severe disease that could have taken my life. I haven't given up, and I hope this book will please you; if I have completed it, it's for a good reason, I hope!

I also would like to thank my colleagues at Marchex, especially my friend (and colleague) Callon Campbell, who never stopped encouraging me.

Thanks to my friend Dominique St-Amand, who has never been stingy with comments to help me improve this book.

Last but not least, Fiodar Sazanavets! Thanks, my friend, for being part of this journey; you were essential in this new challenge I set for myself. Without you, I wouldn't have succeeded.

Prerequisites

This book is aimed at beginner and intermediate developers who want to take their Application Programming Interface (API) development skills to the next level. In this book, I assume you know the basics of .NET, C#, and, therefore, the fundamentals of Object-Oriented Programming (OOP). I also assume you've already used Visual Studio and know how to use it. As for web fundamentals, I've started from scratch, so if you don't know much about the Web, no problem!

Introduction

Dear reader friend, welcome to this book!

In my career, I have worked in various companies and on various complex APIs. Although each company had its challenges, I can assure you that they all had one thing in common: their APIs lacked a lot of love and care. They all suffered from the same problems: poor code organization due to an accumulation of minor errors over the years, lack of consistency in the definition of coding conventions, lack of technological refreshment, misinterpretations of the HyperText Transfer Protocol (HTTP) and Representational State Transfer (REST) principles, missing logging or bad logging practice, and not enough care regarding performances.

I have always enjoyed helping teams overcome these difficulties, and I have decided to write a book to share my experiences and guide you through the best practices of API implementations. This book will focus on some technical architecture of an API, but it will focus more on coding practices to help you avoid the most common mistakes in your development career. I will not cover solution architecture where an API is built around other systems, but keep assured; I will show you how to implement access to external data sources.

At the end of this book, you will know how to develop APIs with ASP. NET Core 8 properly coded, performant, resilient, secure, testable, and debuggable. You will go from a beginner/intermediate level to a senior level by learning precisely WHAT you need to know without feeling overwhelmed by a ton of information.

Let's go!

CHAPTER 1

Introducing HTTP and REST

Before we dive into ASP.NET Core 8 and API development, let's first go back to the basics of any web application. Whether a website is run from a browser or a web service (web API), it's always the same principle: a client and a server will communicate together; a client will send a request to a server, which will then respond to the client. This is all possible with the magic of the HTTP communication protocol. Under this protocol, data can be transported using different formats and constraints. Here is REST! REST is an architectural concept of data representation. Of course, these two should not be confused. In this chapter, we will cover the following content:

- HTTP
- REST architecture style

Unveiling HTTP Behind the Web

The *HyperText Transfer Protocol (HTTP)* is a network protocol for exchanging data between clients and servers. This protocol was invented in 1990 by British computer scientist Tim Berners-Lee to access the *World Wide Web (WWW)*. WWW makes it possible to consult web pages from

a browser using *HyperText Markup Language (HTML)* through *Uniform Resource Identifier (URI)* web addresses. At the beginning of HTTP's history, HTML was the language used to create pages, but since then, HTTP has evolved into a web server that can process data formats other than HTML. For example, a web server can serve (but also accept as input) *Extensible Markup Language (XML),* which is a structured language, or *JavaScript Object Notation (JSON).* Of course, a web server can serve other types of data formats, categorized as *Multipurpose Internet Mail Extensions (MIME) type*, and I will come back to this later.

HTTP follows a technical specification called *Request From Comment (RFC)*, developed by the *Internet Engineering Task Force (IETF)*. There are a ton of RFC specifications identified by numbers. The common point among them is that they define the specifications of the Internet and only the Internet. HTTP is defined by **RFC 7231**. RFC 7231 can be found at this address: www.rfc-editor.org/rfc/rfc7231.

Note   In this book, I will often refer to RFCs. The reason is that I want to teach you the good practices for using HTTP. However, in practice, the actual implementation of those RFCs may differ. Finally, while this chapter aims to teach you the good techniques with HTTP, I will not cover all the HTTP capabilities. I'll stick to what you need to know about building clean APIs with ASP.NET Core.

There are also different versions of HTTP. HTTP has evolved. I will not go into details; in the following, you can find the published versions of the protocol:

- HTTP/0.9 (obsolete)
- HTTP/1.0 (obsolete)
- HTTP/1.1 (still used)

- HTTP/2 (in use but not widely used)

- HTTP/3 (new, not much used)

In this book, I will mainly refer to HTTP/1.1 and sometimes to HTTP/2 and HTTP/3 (when approaching the performance theme).

The Characteristics of HTTP

HTTP has three essential characteristics:

1. It is **stateless**: This means that after sending a request to the server and receiving the response, neither the client nor the server retains any information on the exchanged request.

2. It is **connectionless**: An HTTP connection is open between the client and the server. Once the client has received the response from the server, the connection is closed, and the connection between the two systems is not permanent.

3. It is **independent** of the media, that is, the server can transmit any media as long as the client and the server "agree" on exchanging the content. (I will return to this when I discuss headers.)

While HTTP is stateless, transmitting information between requests may be necessary. Most web applications need to recognize the same user during a browsing session (identify this user through browsing between several web pages). To achieve this, an RFC describes HTTP cookies designed to keep user data browser-side. I won't go into this type of "persistence" in this book, but I will employ more modern techniques. However, if you are interested, consult **RFC 6265** here: www.rfc-editor. org/rfc/rfc6265.

These characteristics may seem abstract, but they will become more apparent as we read this book together. In the next section, I will give you an overview of HTTP requests and responses. This will help you understand HTTP before going into detail.

HTTP Requests and Responses

An HTTP connection works as follows: a request will receive a response unless the connection is broken. Every request and every response works the same way, and I'll go into more detail in the next section.

An HTTP request works with elements as follows:

- A client (a browser, an application) initiates an HTTP request by invoking a **URI**, the address of the requested resource on the server.

- The URI requires the use of a **verb** that will determine the action to be performed.

- Metadata will be sent in the HTTP request, called **headers**. These headers allow controlling the content negotiated with the server, such as sending authentication information and much more.

- **Parameters** are necessary to exchange content with the server and obtain the response sought, and the parameters can be in the request's body, the route, or the URI.

An HTTP response works with elements as follows:

- The server returns a response with a simple status code (HTTP status code) to determine how the HTTP request processing took place.

- The server also returns **headers** in response to the client providing with different metadata.

- Finally, the server will return (or not) a **payload** formatted in the MIME type requested by the client.

So far, I have briefly described and simplified how HTTP works. Figure 1-1, therefore, summarizes what we have previously discussed.
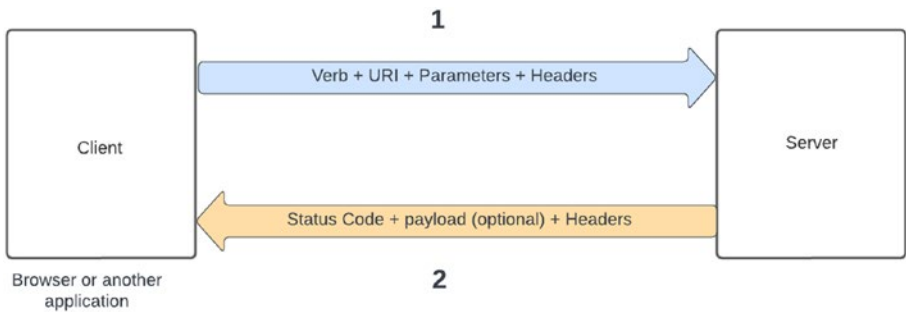
Figure 1-1. *A basic HTTP request and its response*

In the following section, I will detail the HTTP **verbs**, the **request headers**, the format of a **URI**, the different **parameters** passed in a request, the HTTP **status codes**, the **response headers**, and the **payload** formats returned to the client. Once we finish those points, I will bonify Figure 1-1 with more details.

HTTP Implementation

Let's dive into more detail to see what HTTP verbs, request headers, response headers, and HTTP status codes are and how the client passes its parameters in HTTP requests combined with the invocation of a URI.

HTTP Verbs

RFC 7231 defines the following verbs:

- **GET**: This is the most well-known. It allows you to request a resource from the server and receive a response in the desired format (defined by the headers, as we will see later in this chapter). The response is **cacheable** (retain information in memory), and we will discuss it in Chapter 6.

- **HEAD**: This verb is similar to the **GET** verb but does not return any payload; a payload is used to request metadata at the requested address. Since developers barely use it most of the time, I won't use it in this book, but it's good to know what it is used for. Like GET, the server response is also **cacheable**.

- **POST**: This verb is interesting because it serves multiple purposes. This verb allows the creation of new resources, and its payload is attached to the request's body. (I will detail what's a request body further in this chapter.) Another way to send data is to use the *form-data* technique, which will be described later in this book. The **POST** verb also allows modifying data by adding content to the data (appending data to the resource representation according to the RFC). **The server response is not cacheable unless freshness information** is added to the response headers (max-age or Expires headers). We will discuss it again in the "Request and Response Headers" section.

- **PUT**: This verb is confusing because the RFC states that it replaces a resource on the server. Very often, developers confuse **PUT** and **POST** (replace a resource vs. append data to a resource). The server response here is **not cacheable.** However, if a resource doesn't exist, **PUT** should behave as **POST** by creating the resource.

- **DELETE**: This verb is used to delete a resource. The server response is **not cacheable**.

- **CONNECT**: The verb establishes a tunnel to the server through a proxy (a server to which the HTTP request will be delegated and access the server for the requested request). This verb is used for secure requests with *Transport Layer Security (TLS)*, in other words, **HTTPS**. I will also come back to **HTTPS** later in this chapter. I never had to use this verb, and I won't talk about it in this book. The server response is **not cacheable**.

- **OPTIONS**: This verb can be helpful when you want to know what verbs are supported for a given URI. It's also used in the context of *Cross-Origin Resource Sharing (CORS)*, which has its dedicated section further in this book. In the API world, you don't necessarily need to use this verb because you will usually know the available URI for a given endpoint through the *OpenAPI* specification. This will be discussed in the "Extend Your Talent on the Web with REST Architecture Style" section of this chapter. We will also see it together in Chapter 4 when I bring up the API documentation topic. The server response is **not cacheable**.

- **TRACE**: A TRACE request sends a request to the server with no particular payload. This lets you see if intermediate servers, such as proxies, have altered the original request. In the context of APIs, this verb is not used. The server response is **not cacheable**.

RFC 7231 does not describe all the existing verbs, and there are others! **RFC 5789** defines the **PATCH** verb. This RFC can be found here: `www.rfc-editor.org/rfc/rfc5789.html`.

The **PATCH** verb can be confused with **PUT** and **POST** verbs because they all allow modifying a resource on a server. **PATCH** partially updates a resource (like **POST**) when **PUT** tends to replace a resource.

I see many developers confusing each other. Now you are aware of what the RFCs indicate about these verbs, but see that it is commonly accepted to use **POST** for resource creation or to replace **GET** verb when there are too many parameters in the URI to put them in the body of a **POST**. It's also commonly accepted to use **PUT** to entirely or partially replace a resource even if **PATCH** is made for that. Personally, I rarely use **PATCH**, only when I want to update a single property of a resource (e.g., a date). From the moment I start modifying and altering several properties of a resource (a date, a status, a description, etc.), I instead implement **PUT**.

If you recall, I briefly mentioned HTTP status codes in this section. The following section will discuss how status codes link to HTTP verbs. Some verbs are used essentially with certain HTTP statuses. In the next section, I will list the HTTP statuses and what verbs they can be associated with.

HTTP Status Codes

HTTP status codes are essential in an HTTP request/response between a server and a client. They allow the client, when the server's response has been received, to be informed of the result of the processing by the server. HTTP status codes are also governed by **RFC 7231**. I will not exhaustively

detail each HTTP status class and each HTTP code because **RFC 7231** does a pretty good job of doing so, and I won't use all of them in this book. Regarding APIs, status codes are essential for clients to understand what the server is trying to tell us. They provide us with insights on what to do next.

An HTTP status code has three digits. The first digit defines the status category, and there are five categories of HTTP status codes:

- **1xx**: They are purely informational.

- **2xx**: They express that the server received and processed the request successfully.

- **3xx**: They inform the client that the server has proceeded to a redirection, that is, the resource is not at the address indicated, but that the request will be redirected there automatically.

- **4xx**: They mean the request (client-side) is malformed and/or the client (the end user) probably made an input error in their request. 4xx are errors that the client can fix.

- **5xx**: They tell the client that the request on the server has not been completed due to an error.

RFC 7231 is not the only RFC that describes HTTP status codes. However, it describes the codes most often used. **RFC 4918** and **RFC 6585** complete the list, with other codes covering other scenarios.

Table 1-1, taken from **the following RFCs**

- **RFC 7231**: www.rfc-editor.org/rfc/rfc7231

- **RFC 4918**: www.rfc-editor.org/rfc/rfc4918

- **RFC 6585**: www.rfc-editor.org/rfc/rfc6585

lists the association between HTTP status codes and HTTP verbs commonly used as industry standards. I won't use all of them in this book; you will not need to know them by heart. On the other hand, knowing their existence is valuable since you will know their existence and how to use them when required. Later in this book, I'll dig deeper into why I'm using some of them in the code samples I provide.

Table 1-1. *List of available HTTP status codes and verbs most often used with them*

Code	Reason phrase	RFC	Associated verb
100	Continue	7231	All verbs
101	Switching Protocols	7231	All verbs
200	OK	7231	GET, HEAD
201	Created	7231	POST
202	Accepted	7231	All verbs
203	Non-Authoritative Information	7231	GET
204	No Content	7231	POST, PUT, PATCH
205	Reset Content	7231	POST, PUT, PATCH
206	Partial Content	7231	GET
207	Multi-Status	4918	All verbs
300	Multiple Choices	7231	All verbs
301	Moved Permanently	7231	GET, HEAD, DELETE
302	Found	7231	GET, HEAD, DELETE
303	See Other	7231	GET, HEAD, DELETE
304	Not Modified	7231	GET, HEAD

(*continued*)

Table 1-1. (*continued*)

Code	Reason phrase	RFC	Associated verb
305	Use Proxy (deprecated)	7231	All verbs
307	Temporary Redirect	7231	All verbs
400	Bad Request	7231	POST, PUT, PATCH
401	Unauthorized	7231	All verbs
402	Payment Required	7231	Not used yet
403	Forbidden	7231	All verbs
404	Not Found	7231	All verbs except POST
405	Method Not Allowed	7231	All verbs
406	Not Acceptable	7231	All verbs
407	Proxy Authentication Required	7231	All verbs
408	Request Timeout	7231	All verbs
409	Conflict	7231	POST, PUT, PATCH
410	Gone	7231	All verbs except POST
411	Length Required	7231	POST, PUT, PATCH
412	Precondition Failed	4918	All verbs
413	Payload Too Large	7231	POST, PUT, PATCH
414	URI Too Long	7231	GET but applies to all verbs
415	Unsupported Media Type	7231	POST, PUT, PATCH
417	Expectation Failed	7231	All verbs
422	Unprocessable Entity	4918	POST, PUT, PATCH
423	Locked	4918	GET, HEAD, POST, PUT, PATCH
424	Failed Dependency	4918	All verbs

(*continued*)

Table 1-1. (*continued*)

Code	Reason phrase	RFC	Associated verb
426	Upgrade Required	4918	All verbs
500	Internal Error	7321	All verbs
501	Not Implemented	7231	All verbs
502	Bad Gateway	7231	All verbs
503	Service Unavailable	7231	All verbs
504	Gateway Timeout	7231	All verbs
505	HTTP Version Not Supported	7231	All verbs
507	Insufficient Storage	4918	POST, PUT, PATCH

This may seem like a lot of HTTP status codes, but remember that in 99% of the cases, you will only use a handful of codes described here.

Later in this book, we will come back together to some of them, and I will explain them to you with examples of their usefulness.

Now let's move on to another essential component of an HTTP request and response, the request and response headers.

Request and Response Headers

HTTP headers are metadata that allows information to be passed between the client and the server during a request/response flow. These headers transport information but are not limited to authentication data and information on the client's browser.

In this section, I will differentiate between request headers and response headers as they differ in nature for their purpose. For both the request and response headers, **RFC 7231** defines each (some are more detailed, and some are defined in other RFCs, which **RFC 7231** refers to). As in my usual approach you've seen earlier, I will not go in depth since

RFCs describe them in detail. Remember that this book will not cover all possible use cases; specific headers are generated automatically by a browser during the request, some during the response, and by the server. You will not need to know them by heart. On the other hand, knowing they exist is excellent as you get to know they exist and you can customize them for your needs when necessary.

> **Note**   Although **RFC 7231** describes (or redirects to other RFCs) the best-known headers, in reality, there is a complete list of headers (even the most unknown, but without many details) for which you can consult **RFC 4229** here: https://datatracker.ietf.org/doc/html/rfc4229.

Request Headers

Like HTTP status codes, request headers are divided into classes, five exactly:

- Controls headers

- Conditional headers

- Content Negotiation headers

- Authentication credentials headers

- Request context headers

In the following subsections, I will tell you in what RFCs these headers are described, and I will list the links of these RFCs at **the end of this section**.

Controls Headers

There are seven headers in the Controls class. Some of them have various possible directives (key/value pair):

- **Cache-Control**: Used to specify cache duration along the request/response chain. They can handle several directives, and their name perfectly describes their use. For more details, I suggest you consult **RFC 7234**:

  - **no-cache**: Doesn't accept any value, works by itself

  - **no-store**: Doesn't accept any value, works by itself

  - **max-age**: Accepts a value in seconds, for example, Cache-Control: max-age=302400

  - **max-stale**: Accepts a value in seconds, for example, Cache-Control: max-stale=1800

  - **min-fresh**: Accepts a value in seconds, for example, Cache-Control: min-fresh=600

  - **no-transform**: Doesn't accept any value, works by itself

  - **only-if-cached**: Doesn't accept any value, works by itself

- **Expect**: Used to indicate expectations from the server to process the request correctly, for example, Expect: 100-continue. For more details, I suggest you consult **RFC 7231**.

- **Host**: Used to indicate the hostname (server) and the port (optional) from the targeted URI, for example, Host: www.example.com. For more details, I suggest you consult **RFC 7230**.

- **Max-Forwards**: Used to specify the limit of intermediate servers (proxies) that forward the request. Works only with TRACE and OPTIONS verbs and accepts integer values. Example: Max-Forwards: 1. I suggest you consult **RFC 7231** for more details.

- **Pragma**: Used as backward compatibility with HTTP 1.0 cache. This header is ignored when the **Cache-Control** header is used. Example: Pragma: no-cache. For more details, I suggest you consult **RFC 7234**.

- **Range**: Used to return a port of a document with a given range of bytes (most often), for example, Range: bytes 0-2048. For more details, I suggest you consult **RFC 7233**.

- **TE**: Used to specify the chunk transfer coding, for example, defining the compression algorithm, for example, TE: gzip. For more details, I suggest you consult **RFC 7230**.

Conditional Headers

Five conditional headers allow you to apply a condition on the target resource for completing the request. Here they are:

- **If-Match**: Used to check if the requested resource matches a current representation of the resource, for example, If-Match: * (any resource) or If-Match: "123", which targets a resource with the *ETag (Entity Tag)* "123". **ETag** represents a specific version of a resource. For more details, I suggest you consult **RFC 7232**.

- **If-None-Match**: Used to check if the requested resource does not match any current representation of the resource. Works precisely the opposite of the **If-Match** header. Example: If-None-Match: * (any resource) or If-None-Match: "123,". For more details, I suggest you consult **RFC 7232**.

- **If-Modified-Since**: Used to check if the target resource representation modification date is more recent than the provided date, for example, If-Modified-Since: Wed, 22 Aug 2022 21:56:00 GMT. For more details, I suggest you consult **RFC 7232**.

- **If-Unmodified-Since**: Used to check if the target resource representation modification date is less recent than the provided date, for example, If-Unmodified-Since: Wed, 22 Aug 2022 21:56:00 GMT. For more details, I suggest you consult **RFC 7232**.

- **If-Range**: It's a combination of If-Match and If-Modified-Since headers, for example, If-Range: "123" or If-Range: Wed, 22 Aug 2022 21:56:00 GMT. For more details, I suggest you consult **RFC 7233**.

Content Negotiation Headers

Content Negotiation headers are essential in HTTP requests. They allow the client and the server to understand each other on what format should be exchanged. They are four in number:

- **Accept**: Used to define the MIME type that the client can understand, for example, Accept: application/json or Accept: application/json, application/xhtml+xml. For more details, I suggest you consult **RFC 7231**.

- **Accept-Charset**: Obsolete. Many browsers and servers ignore this header. For more details, I suggest you consult **RFC 7231**.

- **Accept-Encoding**: Used to define the compression algorithm, for example, Accept-Encoding: deflate, gzip. For more details, I suggest you consult **RFC 7231**.

- **Accept-Language**: Used to tell the server what language the client is willing to accept, for example, Accept-Language: * (all) or Accept-Language: en-CA. For more details, I suggest you consult **RFC 7231**.

Authentication Credentials Headers

Two authentication headers are necessary to interact with resources protected by authentication. The first is particularly important because it will be developed in this book. Here they are:

- **Authorization**: Used very often to authenticate on the target server. It can handle different types of authentication, such as bearer tokens or basic authentication (both will be addressed in Chapter 9). Example: Authorization: bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9…. For more details, I suggest you consult **RFC 7235**.

- **Proxy-Authorization**: Same as Authorization, it is used to authenticate proxies. Example: Proxy-Authorization: basic YW50aG9ueWdpcmV0dGk6MTIzNA==. For more details, I suggest you consult **RFC 7235**.

Request Context Headers

Finally, **RFC 7231** describes three headers providing additional contextual data for the server. Here they are:

- **From**: Used to tell the server who, with an email address, has sent the request, for example, From: John.Doe@example.com. For more details, I suggest you consult **RFC 7231**.

- **Referrer**: Used to tell the server what URI the request comes from, for example, Referrer: https://anthonygiretti.com. For more details, I suggest you consult **RFC 7231**.

- **User-Agent**: Used to collect information about the user who has originated the HTTP request, like the browser capabilities, for example, User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.0.0 Safari/537.36. For more details, I suggest you consult **RFC 7231**.

These request headers are the most common headers you see. Don't worry about remembering them. This is why RFCs (and my book) exist. Since RFCs define standards, you will come to know how to use them when and where over time. Let's now move to response headers.

Note   Although **RFCs** describe the usage of many headers, you are free to use your custom headers. You can create a specific request header for your application and add value to it.

As promised, here is Table 1-2 that references the links for each RFC I mentioned earlier in this section.

Table 1-2. *Recap of mentioned RFCs in the current section*

RFC	Link
7230	www.rfc-editor.org/rfc/rfc7230
7231	www.rfc-editor.org/rfc/rfc7231
7232	www.rfc-editor.org/rfc/rfc7232
7233	www.rfc-editor.org/rfc/rfc7233
7234	www.rfc-editor.org/rfc/rfc7234
7235	www.rfc-editor.org/rfc/rfc7235

Response Headers

We just went over a series of headers used in an HTTP request. If it is essential to send metadata to the server, it is not less for the response headers. They are essential for informing the client (a browser or an application) of additional metadata relating to the context of the HTTP response. Some are defined in **RFC 7231** and others in **RFCs 7232**, **7233**, **7234**, and **7235**, already introduced in the previous section.

There are four response header classes. Here they are:

- Control Data headers
- Validator header fields
- Authentication Challenges headers
- Response Context headers

Control Data Headers

These headers are among the most important. These headers make it possible to enrich the information sent to the client. They are usually paired with an appropriate HTTP status code. Here is the list of these eight headers:

- **Age**: Used to tell the client when (in seconds) the response has been generated. Usually close to 0, it can be more than 0 if the response has been cached on a proxy. Example: Age: 0. I suggest you consult **RFC 7234** for more details.

- **Cache-Control**: Similar to the **Cache-Control** request header. The response header value is the same as the request header. For more details, I suggest you consult **RFC 7234**.

- **Expires**: Used to tell the client the response date and time is considered outdated, for example, Expires: Tue, 23 Aug 2022 21:35:00 GMT. For more details, I suggest you consult **RFC 7234**.

- **Date:** Used to tell the client when (date and time) the response has been generated on the server, for example, Sun, 21 Aug 2022 11:22:00 GMT. For more details, I suggest you consult **RFC 7231**.

- **Location:** Used to tell the client the URI where the resource can be found after its creation, especially before a POST request, for example, `http://contoso.com/item/52`. For more details, I suggest you consult **RFC 7231**.

- **Retry-After**: Used to tell the client when to retry (date and time or in seconds) a failed HTTP request due to a Service Unavailable (503) response, for example, Retry-After: Wed, 24 Aug 2022 08:15:00 GMT or Retry-After: 60. For more details, I suggest you consult **RFC 7231**.

- **Vary**: Used to tell the client what request parameter header influences the response from the server. "*" means that anything in the request can affect the response. Example: Vary: * or Vary: Accept-Encoding. For more details, I suggest you consult **RFC 7231**.

- **Warning**: Used to tell the client any helpful information. Not recommended since it's deprecated. For more details, I suggest you consult **RFC 7234**.

Validator Header Fields

There are two response headers allowing the addition of metadata to the representation (version) of the requested resource during the HTTP request. Here they are:

- **ETag**: Used to tell the client the version (representation) of the requested resource. It can be any string of characters. Example: ETag: "abc123". For more details, I suggest you consult **RFC 7232**.

- **Last-Modified**: Used to tell the client what date and time the requested resource has been modified for the last time, for example, Sat, 20 Aug 2022, 13:45:00 GMT. For more details, I suggest you consult **RFC 7232**.

Authentication Challenges Headers

The server (or a proxy) allows the client to be told which authentication the server (proxy) accepts, and there are two:

- **WWW-Authenticate**: Used to tell the client what authentication methods the server accepts, for example, WWW-Authenticate: basic. For more details, I suggest you consult **RFC 7235**.

- **Proxy-Authenticate**: Used to tell the client what authentication methods the proxy accepts, for example, Proxy-Authenticate: basic. For more details, I suggest you consult **RFC 7235**.

Response Context Headers

Like the client, the server can send additional contextual data related to the requested resource. There are three:

- **Accept-Ranges**: Used to tell the client what range unit the server supports for partial file download, for example, Accept-Ranges: bytes. For more details, I suggest you consult **RFC 7233**.

- **Allow**: Used to tell the client what verbs the server supports, for example, Allow: GET, POST, PUT, DELETE. For more details, I suggest you consult **RFC 7231**.

- **Server**: Used to tell the client the server technology used to handle HTTP requests, for example, Server: Kestrel. For more details, I suggest you consult **RFC 7231**.

Note   Like the request headers, you can use your custom response headers. You can create a specific response header for your application and add any value to it.

URI, URL, and More

URI

At the beginning of the chapter, I introduced you to the notion of a URI. A *URI (Uniform Resource Identifier)* is, according to **RFC 3986**

> *a compact sequence of characters that identify an abstract or physical resource.*
>
> —www.rfc-editor.org/rfc/rfc3986

In short, this is the address you type in your browser to access a resource like http://www.google.com.

This is just a simple summary, and a URI is more elaborate. A URI contains (or can include) the following:

- **Scheme** (mandatory) is the specification for accessing the remote resource.

- **Authority** (mandatory) combines optional user information, a server address (host), and a port. It must be prefixed with the characters **://**. I won't detail the user information in this book since it's not a relevant feature for this book.

- **Path** (optional) is the data identifying a resource within a particular scope. It must be prefixed with the character /.

- **Query** (optional) is the data identifying a resource within a particular scope. Must be prefixed with the character **?**.

- **Fragment** (optional) is the data that allows identifying a particular subset of the requested resource. Used for HTML pages for identifying anchors. For more explanation, you can visit the following website: https://html.com/anchors-links/. I won't detail it further in this book because it doesn't make sense since it discusses APIs and not HTML pages.

Figure 1-2 illustrates what a URI looks like with its parts.

Figure 1-2. URI structure

The structure elements in orange, **Scheme**, the characters **://**, and **Authority**, are mandatory. The following elements in blue, such as the /, **Path**, **?**, **Query**, #, and **Fragment**, are optional.

As I indicated previously, **Authority** comprises several elements, mandatory and optional. Figure 1-3 will give you an idea of the structure of **Authority**.
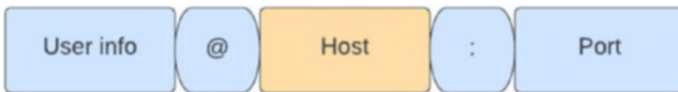
Figure 1-3. Authority structure

Only the **host** is mandatory.

To illustrate the structure of a **URI**, I will show you some examples now:

- **Example 1**: The blog's homepage without any **Path**, **Query**, or **Fragment**. Only the **Scheme** and the **Authority** are used there: `http://anthonygiretti.com`.

- **Example 2**: The search page of a blog with a **Query** parameter: `http://anthonygiretti.com/?s=http`.

- **Example 3**: The blog search page with a **Query** parameter and a **Fragment**: `http://anthonygiretti.com/?s=http#book`.

- **Example 4**: A particular page of a blog with a **Path**: `http://anthonygiretti.com/2021/08/12/asp-net-core-6-working-with-minimal-apis/`.

- **Example 5**: The same HTML page running locally on a development machine: `http://localhost:2222/2021/08/12/asp-net-core-6-working-with-minimal-apis/`.

I'll stop with the URI examples because we'll return to it in this chapter's "Extend Your Talent on the Web with REST Architecture Style" section.

Once again, if you want to learn what a URI is in depth, I suggest you read **RFC 3986**.

URL

You've probably heard of the *Uniform Resource Locator (URL)* before. Well, I bet you could have confused URI and URL like me. I will demystify this for you.

The difference between the two is subtle and little known to developers. A URI defines a resource's identity, while the URL links a resource to a specific access method defined by the **Scheme**. In the subsection "URI," I always gave the same example using the **http** Scheme value, which invokes HTTP. A URL allows invoking other protocols such as

- **File Transfer Protocol**: The Scheme value is **ftp**.

- **Gopher protocol**: The Scheme value is **gopher**.

- **Electronic email protocol**: The Scheme value is **mailto**.

- **Usenet protocol**: The Scheme value is **news**.

- **NNTP**: The Scheme value is **nntp**.

- **Telnet protocol**: The Scheme value is **telnet**.

- **Wide Area Information Server protocol**: The Scheme value is **wais**.

- **Host-specific file names protocol**: The Scheme value is **file**.

- **Prospero Directory Service protocol**: The Scheme value is **prospero**.

To learn more about them, I suggest you read **RFC 1738,** which can be found here: `www.rfc-editor.org/rfc/rfc1738`.

I will discuss URLs again in the "Extend Your Talent on the Web with REST Architecture Style" section as I will for URIs.

And Others…

Two other acronyms can be associated/confused with URI and URL, and these are the following:

- *Uniform Resource Names (URN)*, defined in **RFC 1737** here: www.rfc-editor.org/rfc/rfc1737.

- *Uniform Resource Characteristics (URC)* are not defined in any RFC, but you can find some information here: https://datatracker.ietf.org/wg/urc/about/.

These last two do not represent any interest, at least in this book, but as you certainly have an unquenchable thirst for learning, I offer you the resources to cultivate yourself further.

Parameters

Parameters… And, yes, without telling you, we have already discussed them since this chapter began. What are they for? Well, they are used to find a specific resource (not always) on the server. How? First, we can use them differently, and here they are (again). I'm sure this will remind you of something:

- **By header using custom headers**: For example, myHeader: myValue. This is not the recommended way, but it is possible if needed.

- **By the URL path**: For example, https://www.rfc-editor.org/rfc/rfc1738, where *rfc1738* is the target resource's ID, an HTML page serving the **RFC 1738** data.

- **By the Query**: For example, http://anthonygiretti.com/?s=http. Be careful here. The Query parameters do not necessarily allow access to a specific resource but a set of resources meeting the search criteria.

These last two ways of proceeding are the most adequate, and we will see why in the "Extend Your Talent on the Web with REST Architecture Style" section. Yes, I've been teasing you for a while in this section, but we'll get there soon!

Error Handling

Because HTTP has been well designed, it has been described as an elegant way of handling errors, because, in absolute terms, incorporating errors properly into an HTTP response is simply vital. There's an RFC that describes this, **RFC 7807**, which you can find here: https://datatracker.ietf.org/doc/html/rfc7807.

This RFC defines a JSON contract, named *Problem Details,* returned in response to an API client when an error occurs. *Problem Details* contains the elements described in Table 1-3, taken **as is**, from **RFC 7807**.

Table 1-3. *Recap of mentioned RFCs in the current section*

Property name	Description
type (string)	A URI reference [RFC3986] that identifies the problem type. This specification encourages that, when dereferenced, it provides human-readable documentation for the problem type (e.g., using HTML [W3C.REC-html5-20141028]). When this member is not present, its value is assumed to be "about:blank".
title (string)	A short, human-readable summary of the problem type. It SHOULD NOT change from occurrence to occurrence of the problem, except for purposes of localization (e.g., using proactive content negotiation; see [RFC7231], Section 3.4).
status (number)	The HTTP status code ([RFC7231], Section 6) generated by the origin server for this occurrence of the problem.
detail (string)	A human-readable explanation specific to this occurrence of the problem.
Instance (string)	A URI reference that identifies the specific occurrence of the problem. It may or may not yield further information if dereferenced.

Problem Details is very convenient since it is a well-detailed error contract. I won't give you any example here since **RFC 7807** already provides some. I will use Problem Details in this book while showing you how to handle errors with ASP.NET Core 8.

HTTPS, TLS, and HSTS

So far, I haven't talked to you about security concerning HTTP. Well, here we are! HTTP does have a problem with security since it allows data to be exchanged between the client and the server in the clear on the Internet. As you can imagine, this is a real problem! Transporting unencrypted data can cause the following issues:

- A hacker can "listen" and steal the data exchanged between the client and the server.

- A hacker may corrupt data.

- No security authentication ensures that the client communicates with the website requested by the client.

This is where HTTPS comes in. HTTPS is a secure version of HTTP, hence the letter **S** for **Secure**. The Scheme value will be **https**, for example, https://anthonygiretti.com. Unlike HTTP, which runs by default on port 80, HTTPS runs on port 443 by default. We will see that together later in this book, but with ASP.NET Core, we can edit the port number by configuration.

HTTPS is based on the *Transport Layer Security (TLS)* protocol, which makes it possible to overcome the problems mentioned previously with HTTP. That is, HTTPS provides

- Encryption—the data is no longer visible (in the clear) on the Internet

- Protection of data integrity—they are no longer falsifiable

29

- Authentication by ensuring the customer communicates with the website they requested

---

Note   TLS is an improved *Secure Socket Layer (SSL)* encryption–based protocol version. SSL is often associated with TLS, commonly named SSL/TLS encryption, but in reality, TLS replaces SSL/TLS since it has been updated since 1996.

So how does SSL/TLS work? A key exchange occurs between the client and the server. The latter will establish an encrypted connection named *TLS handshake.*

You don't need to know how the *TLS handshake* works, but if you are interested, you can learn more about it here: `www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/`.

Only the client and the server possessing the decryption key can encrypt/decrypt the data exchanged. To make it possible, the server needs to obtain an SSL certificate, which the server's administrator (or a developer) will install and has previously been obtained by them from a *Certificate Authority (CA)*. I won't go into detail here. This book will deal with the implementation of APIs with ASP.NET Core. The latter will automatically (but with your consent) use an SSL certificate when first run in Visual Studio 2022. However, I will show you how to force your APIs to use HTTPS when a client invokes a URL by configuring the *HTTP Strict Transport Security (HSTS)* policy mechanism in ASP.NET Core. To learn more in depth about HTTPS, TLS, and HSTS, they are described in the following RFCs:

- **RFC 2818 for HTTPS**: `https://datatracker.ietf.org/doc/html/rfc2818`

- **RFC 8446 for TLS**: `https://datatracker.ietf.org/doc/html/rfc8446`

- **RFC 6797 for HSTS**: `https://datatracker.ietf.org/doc/html/rfc6797`

I advise you to read these RFCs only if you feel the need to understand every aspect of HTTPS, but in everyday life, what you will have to remember is the need to install an SSL certificate on the server, which will allow the exchange of an encryption/decryption key on the server and the client where the data will no longer be readable in plain text or modifiable on the Internet. Figure 1-4, therefore, summarizes the situation.

Figure 1-4. *A basic HTTPS request and its response*

As simple as it may seem, this figure demonstrates (almost) exactly what anyone trying to spy on an HTTPS request sees.

Let's Put the Pieces of the Puzzle Back Together

Here we have a set of concepts described by RFCs. I introduced you to what HTTP is by starting with a basic diagram symbolizing what each concept implies in HTTP. This book deals with APIs that serve data in JSON format (remember the MIME type application/json) and not HTML or other pages, but the principle remains the same. To finally see and understand what an HTTP request does, I will redo a more detailed version of Figure 1-1 with Figure 1-5. The latter describes the invocation of the URL *www.myServiceApi.com/scope/someId* with the *GET* verb, accepting

only the *application/json* format and the *en-CA* language; accepting *gzip,* *deflate,* and *br (Brotli)* encoding; and finally authenticating with the *basic authentication*. The response returns status *200 OK* with the requested Content-Type *application/json*, the Content-Length response, the server technology (*Kestrel*), and a *JSON payload*. I voluntarily let the request and response data clear even if HTTPS appears in Figure 1-5.
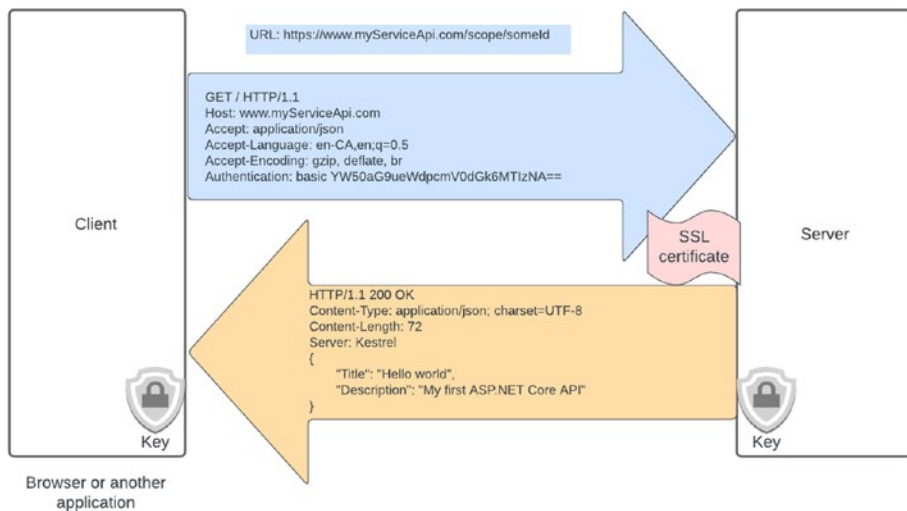
Figure 1-5. *A "real-life" HTTPS request and its response*

As simple as it may seem, this figure demonstrates (almost) exactly what anyone trying to spy on an HTTPS request sees.

Extend Your Talent on the Web with REST Architecture Style

In 2000, an American computer scientist named Roy Fielding defined the architectural style used for web service development: *Representational State Transfer (REST)*. HTTP is a protocol defined by a committee issuing RFCs; REST is a concept. A concept that does not redefine

HTTP does not add any additional functionality. REST is independent of HTTP. Unfortunately, many developers are getting mixed up. The confusion is that HTTP is a client-server communication protocol, while REST identifies constraints on how the server and client talk. I will introduce you to two different topics: REST constraints and good practices. Following REST constraints makes you respect API REST principles, and following only best practices doesn't ensure you are REST compliant.

REST Constraints

A web application should implement its business logic with all sets of object entities (e.g., a product is an entity) and possible operations (e.g., retrieve the product information based on product ID). These possible operations with these entities must be designed with four main operations or methods: *Create, Retrieve, Update, and Delete (CRUD).* These entities are called resources and are **presented or represented** in a form such as JSON, XML, or others. A client, therefore, calls the CRUD functions on the server via HTTP (or not, e.g., gRPC or SOAP, which are other communication protocols) to manipulate the **representation** of an entity on the server. This defines the term **Representational** of REST.

What does the **State Transfer** term mean? The "state transfer" from the client to the server is, for example, when the client first calls the "create product" operation, after calling what would be the next product state or product states that the "client" can call. Its status can be "retrieve created product data" or "update product data." So this is the **State Transfer** term of REST.

REST is not defined by this alone. REST is based on six constraints:

- Separation of responsibilities between client and server. (The client displays data, and the server computes them.)

- No session state (stateless). Neither the client nor the server needs the state of the other to communicate.

- Caching resources.

- Consistent communication with identifiable resources.
  In HTTP vocabulary, there must be a URL, a response
  containing a body, and a header.

- Allows the addition of intermediate layers (e.g.,
  proxies).

- Allows the client to ask the server for a piece of code
  that the client will execute.

In this book, I will mainly develop the first four points here, and we can still consider design, all along this book, of REST APIs.

REST Good Practices

Earlier in this chapter, I told you I would return to URIs and URLs. I'll talk to you here about good practices for defining these with REST.

Base URL

First of all, let's start by establishing the base URL. The base URL is the root URL of all your HTTP endpoints. For example, it is customary to use URLs with an exact domain name for your business and a path and parameters, which can become long and complicated. This is correct for a website, but simple URLs are recommended for REST APIs. Let's consider your "My company" offering a product sales website and then exposing it on the same REST API URLs. For example, this is to be avoided: *https://www.mycompany.com/home/services/rest*. Prefer the following, which is more meaningful for an API: *https://api.mycompany.com*.

Media Type

The JSON format with the header Content-Type: application/json is recommended. This is the most practical and commonly adopted. It is possible to use XML, but the JSON format is commonly adopted for practicality (JSON is not as strict as XML on syntax) and also for performance reasons. JSON is more efficient in terms of serialization/ deserialization compared with XML. As a reminder, serialization and deserialization is a process that makes it possible to transform a data structure (serialization) into a storable data format and achieve the opposite (deserialization). For example, the *Product* data structure contains an identifier, a name, and a description. Listing 1-1 shows the serialization of the *Product* data structure.

Listing 1-1. JSON serialization of a Product data structure

```
{
     "Id": 1,
     "name": "My product name,"
     "description": "My product description"
}
```

Listing 1-2 shows the same data structure serialized in XML.

Listing 1-2. Product data structure serialized in XML

```
<?xml version="1.0" encoding="utf-8"?>
<product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Id>1</Id>
  <Name>My product name</Name>
  <Description>My product description</Description>
</product>
```

Admit that it's more pleasant with JSON. It's less verbose than XML, which is heavier because of its massive content vs. JSON.

For your information, because this is not a constraint related to HTTP, using the header allows us to protect against attacks like *Cross-Site Request Forgery (CSRF)*. This attack enables the execution of client-side code. For example, JavaScript scripts execute unwanted functions in the client browser. The malicious code is blocked and cannot be executed by forcing a serialization (e.g., in JSON or XML). To learn more, I advise you to consult this post on owasp.org: `https://owasp.org/www-community/attacks/csrf`.

Note   The *Open Worldwide Application Security Project (OWASP)* is a nonprofit organization that provides resources for developers to ensure they develop secure applications.

URL Naming

Here is a subject that I defend vigorously. Why? Because by writing URLs well, we understand what they do by reading them without adding unnecessary words. Imagine that you were trying to retrieve the complete list of products that you have in your database. I have already seen URLs written like this:

```
(GET) /getAllProducts
```

Do you see what I mean? Well, this URL is a phrase on its own. I could, for example, write this instead:

```
(GET) /products
```

It's much more straightforward. I already use the GET verb, and I don't need to show it in the URL, do I? Using products in the plural is more than enough!

Now imagine that I only want ten. I'm not going to write this:

```
(GET) /getSomeProducts?limit=10
```

But I rather write

```
(GET) /products?limit=10
```

Don't you agree? Now I want to create a product. I will proceed as follows

```
POST /products
```

and not like this

```
POST /products/create
```

or

```
POST /createProduct
```

Finally, in the logic of the state transfer, the response of the POST operation will return me a status *201 Created* within the headers (or in the payload) and the ID of the product created, which will allow me to write the request GET to retrieve the information of the product created

```
GET /products/{id}
```

and not

```
GET /getProduct?id={id}
```

Do you see how practical it is? The logic is the same if I want to edit the product; I will use the URL as the POST operation and the same URL as the GET operation to delete it, which gives the following set of CRUD operations:

- **Create**: POST /products
- **Retrieve**: GET /products/{id}

- **Update**: PUT or PATCH /products/{id}

- **Delete**: DELETE /products/{id}

This logic applies in the same way to linked resources. When a data structure is linked to another, for example, a product is linked to a category, this product belongs to a certain category. The best REST practice is for the URL path to be subdivided as follows to access/manipulate products of a certain category

```
/categories/{categoryId}/products
```

and as follows to access/manipulate a particular product in a particular category:

```
categories/{categoryId}/products/{productId}
```

The CRUD operation set to manage one or more products in a particular category gives this:

- **Create**: POST /categories/{categoryId}/products

- **Retrieve**: GET /categories/{categoryId}/products/{productId}

- **Update**: PUT or PATCH /categories/{categoryId}/products/{productId}

- **Delete**: DELETE /categories/{categoryId}/products/{productId}

So, of course, handling a product without needing to access the category is possible; however, if you want to check before handling a product if it belongs to a category (it depends on your business logic), well, it's good practice to adopt, rather than using query parameters.

We'll return to this later in this book, but passing an ID in the URL path as we did is called "routing." In ASP.NET Core, the *categoryId* and the *productId* are called *route parameters*.

API Versioning

Sometimes an API evolves rapidly in terms of proposed functionalities or improvement of existing features that breaks the usual functioning (evolution of service contracts, that is, the data structures exchanged between the client and the server). Unfortunately, clients often evolve less quickly on their side, but an API must not break client applications. We must continue to maintain them while developing the API. There is a solution for this, and it is API versioning. How does it work? Well, there are as many URLs for as many versions of the API. A good practice is to insert the version number in the URL as follows—/**v{version}**/, for example:

```
https://api.mycompany.com/v1/categories
https://api.mycompany.com/v2/categories
```

And so on… We will come back to this later in this book as well.

There is another way to define a versioned API. Instead of specifying the version in the URL, headers can be used to ask for a specific API version. HTTP does not define any particular header for this. You can create your own, for example:

```
GET https://api.mycompany.com/categories
X-API-Version: 1
```

> **Note**   A good practice when creating a custom/nonstandard header is to prefix it with the **X-** characters.

I barely use API versioning from headers, but it is a valid choice if you want to use them. I will show you some examples in Chapter 4 of this book.

A third way, which is probably never used (on my end, I have never seen that before), is to use media type versioning. It implies the usage of *Accept/Content-Type* headers to define a version of your API. I won't go further with that, either. But If you want to learn more about it, you can read the Microsoft documentation here: https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#media-type-versioning.

API Documentation

One thing that is really important and almost unanimously accepted is documenting your API. What does it consist of? It is a question of exposing on a dedicated endpoint or an HTML page or in *YAML Ain't Markup Language (YAML)* or again in JSON, all the URLs that your API exposes (including the versions) with the incoming parameters, the structure's output data, the HTTP status codes used by the API, etc. The goal is to let the client know how to consume your API correctly.

There is a specification called OpenAPI to carry out this documentation task. This specification is implemented through a set of developing tools. It is named *Swagger*. I'll return to the Swagger tools for ASP.NET Core later in this book. In the meantime, if you want to know more about OpenAPI, you can consult the OpenAPI initiative website: https://spec.openapis.org/oas/latest.html.

Note   YAML is a popular serialization language. I won't use it further in this book, even if it's a language that can potentially replace JSON for configuration files, for example. To learn more about this language, you can read its specification here: https://yaml.org/.

Summary

This chapter has been long and informative. However, we will return to what we have seen in this first chapter and practice what you have learned here. After all, this chapter is a kind of introduction, a necessary step to properly implement our APIs later since it is based on RFCs, that is, a kind of "truth" that allows understanding of what HTTP is and how it works. Before going further in this book, it seemed necessary to know and understand the headers, verbs, status codes, and parameters that allow invoking URLs. Moreover, I have introduced you to good practices for developing REST APIs. However, these are not based on RFCs but are commonly accepted by the developer community worldwide. In the next chapter, we will focus on ASP.NET Core 8.

CHAPTER 2

Introducing ASP.NET Core 8

Microsoft released its first full-stack web application development framework with ASP.NET in 2002 with ASP.NET Web Forms. The years that followed were rich in developments such as ASP.NET Model-View-Controller (MVC), ASP.NET Web API, and SignalR. The framework evolved too quickly with new functionalities without changing its core, more precisely, the assembly named *System.Web*. Very quickly, new challenges appeared, such as performance, the possibility of running ASP.NET on servers other than IIS (which is the Windows-only web server designed by Microsoft), increasing its affinity with the cloud to facilitate its deployment significantly, and greatly improving its configuration by making it more flexible. ASP.NET Core was born!

ASP.NET Core is a complete overhaul of the trendy ASP.NET framework and allows you to develop many types of applications:

- Web apps, such as MVC, Razor Pages, or single-page applications with Blazor

- APIs (REST APIs, remote procedure calls, and real-time)

- Background tasks running as Windows services (or Unix daemons) or within ASP.NET Core applications

At this time, ASP.NET Core 8 (delivered with .NET 8) is the latest version. This chapter introduces you to ASP.NET Core 8, which we'll use throughout this book. ASP.NET Core 8 no longer supports ASP.NET Web Forms and *Windows Communication Foundation (WCF), which is a SOAP-based web service framework*. However, a project named CoreWCF was released in early 2022. If you are interested, you can read this post: `https://devblogs.microsoft.com/dotnet/corewcf-v1-released/`.

In this chapter, I'll teach you ASP.NET Core fundamentals and the following application types that will be used to build REST APIs:

- ASP.NET Core Web API

- ASP.NET Core minimal APIs

ASP.NET Core Fundamentals

Before diving into ASP.NET Core, let's talk about the fundamentals. Once we know the fundamentals of ASP.NET Core, we can use this knowledge to build any web application we'd like, including gRPC.

For an ASP.NET Core application, the application's entry point is the *Program.cs* file as shown in Listing 2-1. In this file, you start creating your application by instantiating a *WebApplicationBuilder* with the static method *WebApplication.CreateBuilder*. The *WebApplicationBuilder* allows customizing your application by adding the desired components (configuration) and activating them (activations).

Listing 2-1. Example of Program.cs file

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

app.MapGet("/", () => "Hello World!");

app.Run();
```

> **Note**   This is the default *Program.cs* file generated from the ASP.NET
> Core 8 template. It implements the evolved C# feature named "top-
> level statements." The same remark applies to using statements,
> and the default ASP.NET Core 8 template uses the C# "global usings"
> feature.

The *Program.cs* file has two distinct parts:

- **Services configuration** includes the type of
  application, third-party libraries, authentication,
  authorization, and the registration of services with
  dependency injection.

- **Services activation** defines the ASP.NET Core
  middleware pipeline. A middleware is a component,
  once assembled (in a particular order) into an
  application, that can handle requests and responses
  and perform operations before and after the next
  component, as shown in Figure 2-1.

Figure 2-1.   *The ASP.NET Core middleware pipeline*

Services configuration is implemented at the beginning of the file
before building the app with the *builder.Build( )* method, and services
activation occurs after the latter but **before** the *app.Run( )* method as
shown in Listing 2-2, which is a sample of a configuration of an ASP.NET
Core Razor Pages application.

Listing 2-2. Example of a configured Program.cs file

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

// Services configuration
builder.Services.AddRazorPages();

var app = builder.Build();

// Services activation
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```
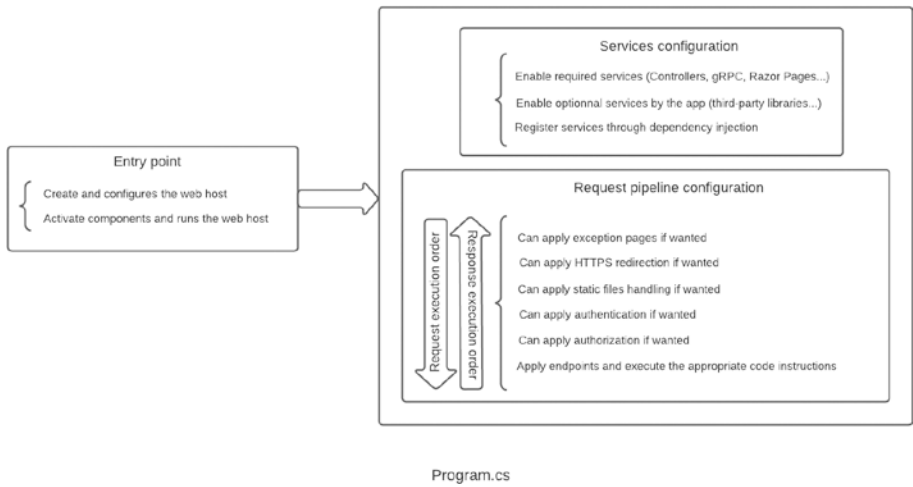
I understand this may still seem a blur to you, but the following will explain the architecture of ASP.NET Core to you more. I summarize the ASP.NET Core architecture in Figure 2-2.

Services configuration

Enable required services (Controllers, gRPC, Razor Pages...)

Enable optionnal services by the app (third-party libraries...)

Register services through dependency injection

Entry point

Create and configures the web host

Activate components and runs the web host

Request pipeline configuration

Request execution order

Response execution order

Can apply exception pages if wanted

Can apply HTTPS redirection if wanted

Can apply static files handling if wanted

Can apply authentication if wanted

Can apply authorization if wanted

Apply endpoints and execute the appropriate code instructions

Program.cs

Figure 2-2. *ASP.NET Core architecture*

First, you must understand *dependency injection* since it's central to ASP.NET Core. *Dependency injection* is a technique that weakly couples objects and service classes with each other and their dependencies. Instead of directly instantiating services in methods through constructors, the class declares what dependencies it needs. In this book, we'll use services configured with their implemented interface. These interfaces will be injected into the constructors of the classes calling these services. This decoupling allows our code to be abstracted and also facilitates testability. Later in this book, we'll see how to easily test our code and take advantage of dependency injection. The service lifetime injected by dependencies is essential. Depending on the injected services, some need to be used once or several times for the *HyperText Transfer Protocol (HTTP)* request context or even used only once for all users making an HTTP request to the server. ASP.NET Core supports three life cycles:

- **Transient**: A new service instance is created for each incoming request. This means that on the same incoming HTTP request, the developer can deal with a new instance of the same service for each HTTP request.

47

- **Scoped**: The service is instantiated once per incoming request. This is the most used lifetime. It guarantees the uniqueness of a service instance per user.

- **Singleton**: The service is instantiated once for the entire application's lifetime (as long as it is not restarted), and all users share this instance. In ASP.NET Core, singleton lifetime is thread-safe (with object construction); ASP.NET Core manages it for you if you register your service correctly in the dependency injection container. However, if you need to modify a property, such as a *Dictionary,* you'll need to use a *ConcurrentDictionary* instead.

Listing 2-3 shows how to configure the three different lifetimes. Note that the parameter on the left is the interface and the parameter on the right is the concrete class that implements this interface. A compilation error will occur if the class doesn't implement the interface to be injected by dependency.

Listing 2-3. Configure each lifetime type

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

services.AddControllers();
services.AddSingleton<ISingletonService, SingletonService>();
services.AddScoped<IScopedService, ScopedService>();
services.AddTransient<ITransientService, TransientService>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
```

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

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Listing 2-4 shows how to inject services with an MVC controller after registering to what concrete implementation they are mapped to in the *Program.cs* file.

Listing 2-4. Example of an MVC controller where services are injected by constructor

```
public class DemoController : Controller
{
    private readonly ISingletonService _singletonService;
    private readonly IScopedService _scopedService;
    private readonly ITransientService _transientService;

    public DemoController(ISingletonService singletonService,
                          IScopedService scopedService,
                          ITransientService transientService)
    {
        _singletonService = singletonService;
        _scopedService = scopedService;
        _transientService = transientService;
    }
}
```

Depending on your needs, you might sometimes want to use a service such as *Singleton*, *Scoped*, or *Transient*, but you must be aware of the scope hierarchy.

A *Transient* service can directly access a *Singleton* service or a *Scoped* service, which can directly access a *Singleton* service. The opposite is impossible because any object with a longer life than another cannot access it directly. Figure 2-3 summarizes it.
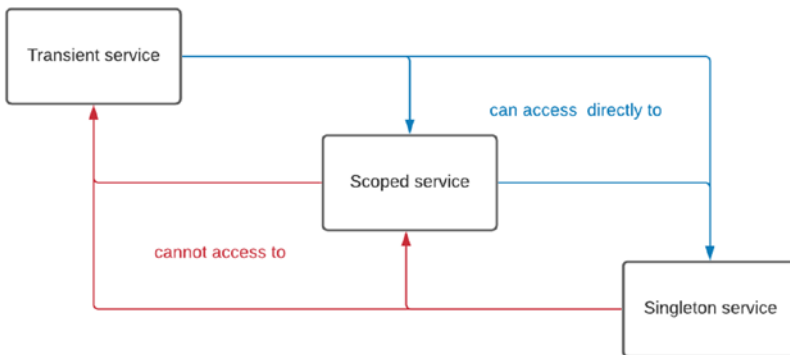
Figure 2-3. *The scope hierarchy*

ASP.NET Core provides a way to add extra configuration within your application that dependency injection can consume anywhere. You can customize and store the additional configuration in an *appsettings.json* file. You can store settings here that differ by environment. For example, in a development environment, the *appsettings.development.json* file can contain configuration specific to development mode. If a JSON key/value pair is present in both files, the more specific file *(appsettings.development.json)* will override the value presented in the main file for a given key. You can create an Options object to populate with your configuration—this is referred to as the **Options pattern**. Listing 2-5 shows an SMTP configuration in *appsettings.json* that maps the *SmtpConfiguration* object shown by Listing 2-6 and then uses the dependency injection system as shown in Listing 2-7. Finally, the *IOptions<TOptions>* interface is injected in the *DemoController* as shown in Listing 2-8.

Listing 2-5. SMTP configuration in appsettings.json

```
{
  "SmtpConfiguration": {
    "Domain": "smtp.gmail.com",
    "Port": 465
  }
}
```

Listing 2-6. SmtpConfiguration object

```
public record class SmtpConfiguration
{
     public string Domain { get; init; }
     public int Port { get; init; }
}
```

Listing 2-7. SmtpConfiguration object bound and registered in the dependency injection system

```
var builder = WebApplication.CreateBuilder(args);    services.
Configure<SmtpConfiguration>(Configuration.GetSection("SmtpConf
iguration"));
....
```

Listing 2-8. Injecting SmtpConfiguration options into DemoController

```
public class DemoController : Controller
{
     private readonly SmtpConfiguration _smtpConfiguration;

     public DemoController(IOptions<SmtpConfiguration>
     smtpConfigurationOptions)
```

```
    {
            _smtpConfiguration = smtpConfigurationOptions.Value;
    }
}
```

This is the simplest way to use options in ASP.NET Core. Depending on your needs, you can also leverage IOptionsSnapshot<TOptions> and IOptionsMonitor<TOptions>. To learn more, read Microsoft's documentation: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-8.0.

The last important thing to mention is the possibility of setting up development mode in ASP.NET Core by configuration. What is development mode? It allows developers to configure a different behavior of the application (set up, e.g., encrypted connection strings in production but not encrypted in *development mode*). Another important thing is the ability to display more detailed information about the unhandled error that occurred. Because it's more detailed, developers **should not enable development mode** in production. To enable it, you must set the *ASPNETCORE_ENVIRONMENT* environment variable to *Development* in the *launchSettings.json* file or within the project properties panel. Further in this book, you'll see a concrete example of using environment variables and encrypted connection strings. Listing 2-9 shows a *launchSettings.json* file configured for *development mode* with IIS and self-hosted mode.

Listing 2-9. Development mode enabled within the launchSettings.json file

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
```

```
      "applicationUrl": "http://localhost:57090",
      "sslPort": 44366
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "MVCDemo": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://
      localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

ASP.NET Core Web API

ASP.NET Core Web API allows you to …you guessed it …create web APIs.

A web API is an *Application Programming Interface (API)* used in conjunction with HTTP. Currently, web APIs use Representational State Transfer (REST), which is associated with the JavaScript Object Notation

(JSON) interchange format and Extensible Markup Language (XML), which is used less often. APIs use HTTP features such as a *Uniform Resource Identifier (URI)*.

Because the final Internet user is significant in terms of the variety of terminals used, we want to provide data to browsers or recent device applications in a fast, secure way; we need a web API compatible with all of this. ASP.NET Core Web API is a relevant and performant framework for building web services that many users can use.

ASP.NET Core Web API follows the *Model-View-Controller (MVC)* pattern. In traditional web apps, the *V (View)* in MVC is the web page. With the web API, it's a response in JSON, XML, or any other format. Figure 2-4 gives an overview of this pattern.
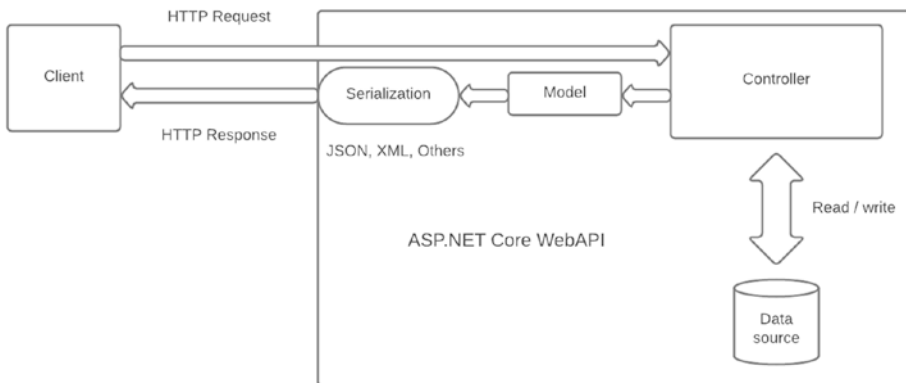
Figure 2-4. *ASP.NET Core Web API architecture*

Note   All the stuff I will create in this book is based on Visual Studio. But you can do the same things with the .NET *Command-Line Interface (CLI)* where you can find the syntax here: `https://learn.microsoft.com/en-us/dotnet/core/tools/`.


Now, let's see how to create a web API in Visual Studio 2022. As shown in Figure 2-5, select "Web" in the drop-down list to more easily find the project type you're looking for: ASP.NET Core Web API.
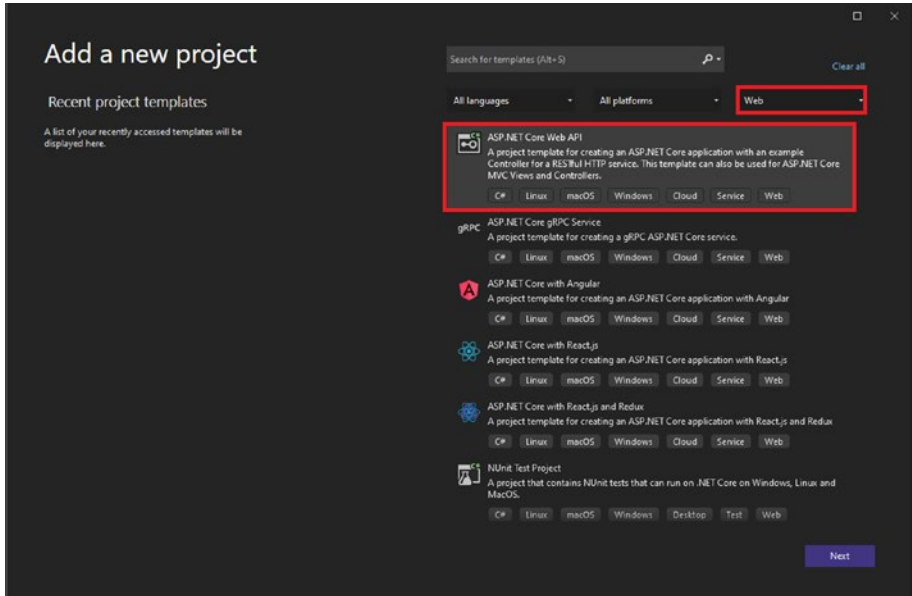

Figure 2-5. *How to find the project type: ASP.NET Core Web API*

Once you choose "ASP.NET Core Web API," you'll need to configure the project name, the location on your computer, and the solution name as shown in Figure 2-6.
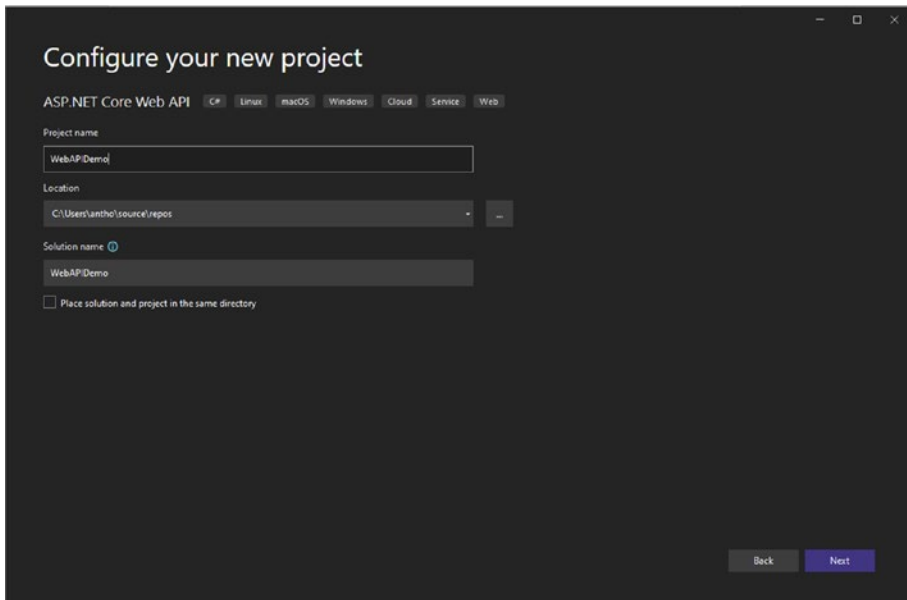
**Figure 2-6.** *How to create your new ASP.NET Core Web API project*

After that, you get the opportunity to select various options to customize your application. As shown in Figure 2-7, you can choose the runtime to run your ASP.NET Core Web API, and I strongly suggest you select the latest (.NET 8) in the drop-down; ASP.NET Core 8 can only be run by .NET 8. You can also set the authentication type (Windows, Microsoft Identity Platform, or no authentication), HTTPS, Docker, and OpenAPI support and whether you want to use controllers or not (using minimal APIs instead).
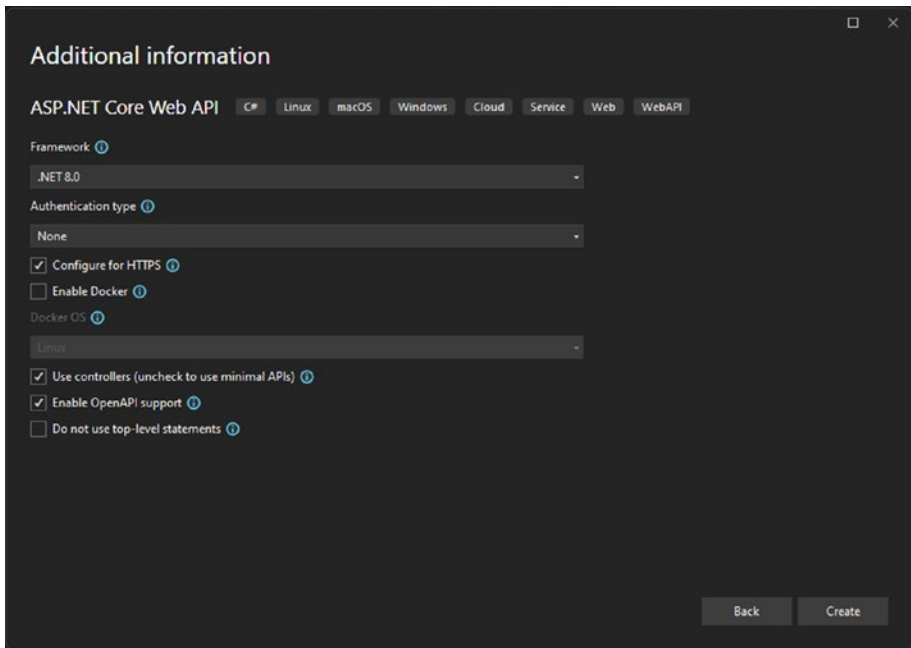
**Figure 2-7.** *How to configure the ASP.NET Core Web API*

If you aren't familiar with Docker, Docker is an open source containerization platform. *Docker* enables developers to containerize their applications that combine application source code with all the operating system (OS) libraries and dependencies required to run the code in any environment. To learn more, visit `www.docker.com/why-docker`. As for OpenAPI, it's a specification that defines a standard, language-agnostic interface to RESTful APIs, allowing humans (and the machine) to discover and understand the features of a service without reading the source code. For details, you can refer to this website: `https://swagger.io/specification/`. Swagger is the set of tools built on top of OpenAPI.

After clicking the "Create" button, Visual Studio will generate your project with a default template, including a *WeatherForecast* model and controller. Figure 2-8 shows the default project created by Visual Studio.
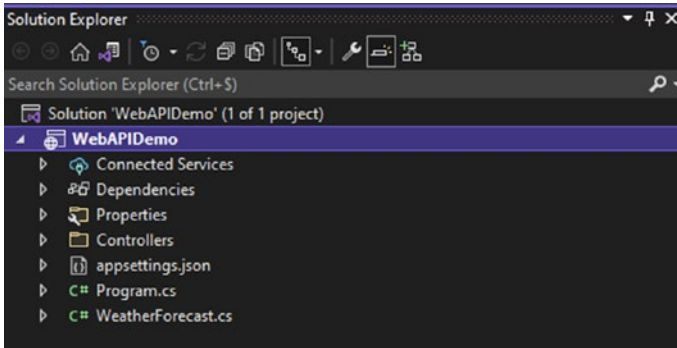
Figure 2-8. *Default ASP.NET Core Web API WeatherForecast template app*

Let's take a quick look at the controller in Figure 2-9.
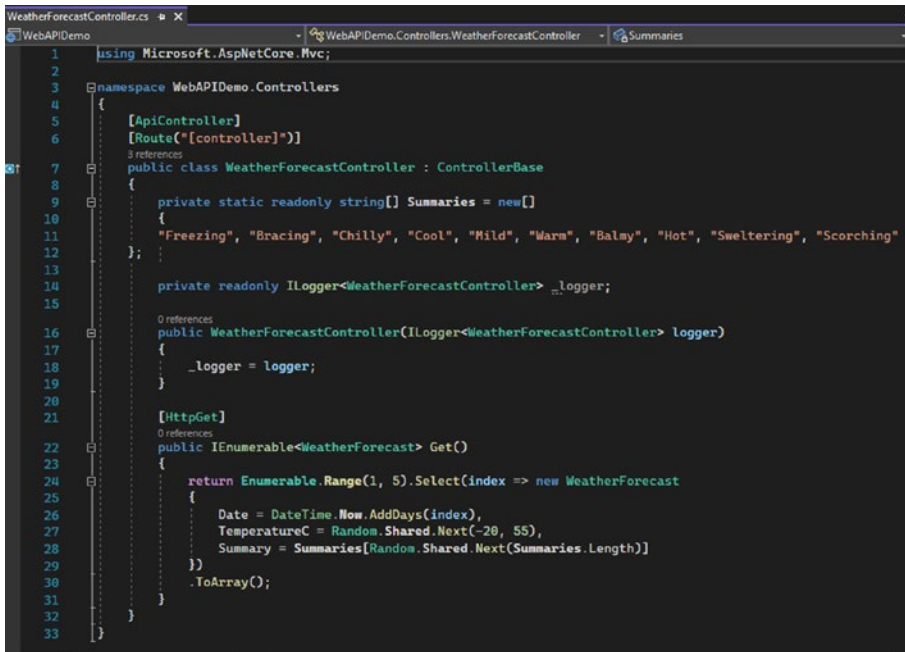
Figure 2-9. *The WeatherForecastController class*

Let's take a look at the *Program.cs* file, the entry point of the application. As you can see, we enabled OpenAPI before. Swagger UI uses this in Figure 2-10.

Figure 2-10. *Program.cs file configured with OpenAPI (Swagger)*

Visual Studio will open the browser with the OpenAPI web page and display all endpoints within the app if you run the app. At this point, it will show only the "WeatherForecast" GET endpoint. To try it, you can click the "Execute" button and view the data returned in the "Response" section, as shown in Figure 2-11.
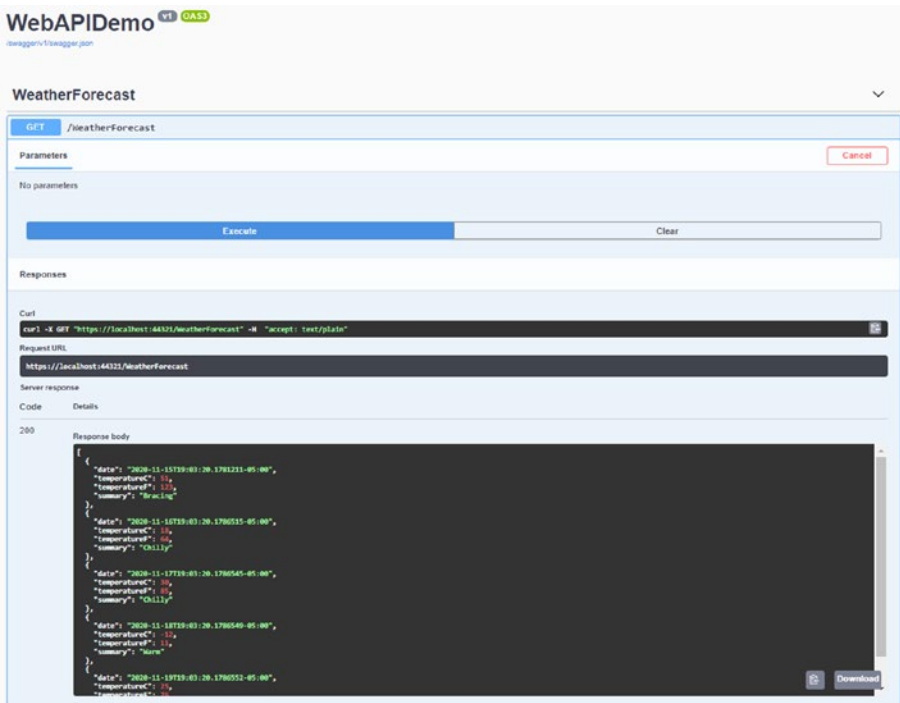
Figure 2-11. *Swagger UI web page*

The Swagger UI web page is open by default. When we enabled OpenAPI, Visual Studio configured it to open in the *launchSettings.json* with the "launchUrl" parameter, as shown in Figure 2-12.

Figure 2-12. *"launchUrl" parameter set to "swagger" value*

In addition to Swagger, it is possible to use a command-line tool, *HttpRepl (HTTP Read-Eval-Print Loop),* which is lightweight and cross-platform and can be used on ASP.NET Core APIs but also other kinds of APIs. This tool makes HTTP requests and views their results wherever the API is hosted. HttpRepl supports the following verbs: DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT.

To install it, just run the command in a PowerShell window as shown in Listing 2-10.

Listing 2-10. HttpRepl installation command

```
dotnet tool install -g Microsoft.dotnet-httprepl
```

If you want to discover all the commands supported by this tool, you can enter the command shown in Listing 2-11.

Listing 2-11. HttpRepl help command

```
httprepl --help
```

Figure 2-13 shows the commands available in the output window.

Figure 2-13. *Available commands for HttpRepl*

Figure 2-14 shows the exploration, navigation, and execution of the endpoints available in the API you want to discover. Endpoints are known because of the parsing of the *swagger.json* file, which is done automatically by typing the command (connection to the API) shown in Listing 2-12.

Listing 2-12. Connection to the local API base URL

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

Endpoint exploration, navigation, and execution shown in Figure 2-14 are pretty original since they uses *MS-DOS* such as *ls* to list endpoints (listing files in a directory in Windows) or position itself on an endpoint with the *cd* command (moving to a directory in Windows).

Figure 2-14. *Exploration, navigation, and execution of API endpoints*

Finally, you can use Postman if you do not want to use the generated Swagger web page or HttpRepl. Postman is a GUI for generating HTTP requests to test the endpoints of a given API. This tool allows you to configure all the possible request parameters, such as the URL, headers, verbs, query string, and body. To download this tool, you can go to this page: www.postman.com. Figure 2-15 shows what the Postman interface looks like.
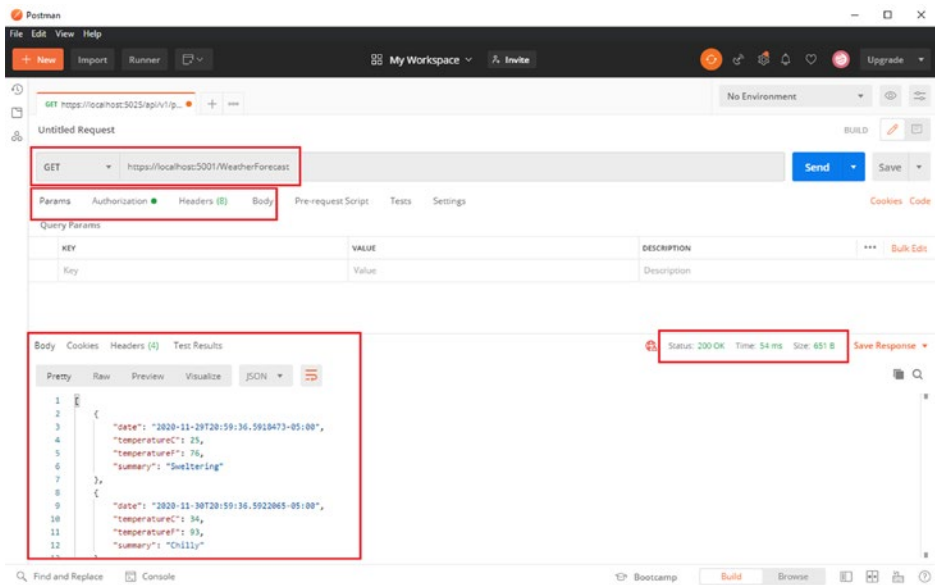
Figure 2-15. *Postman GUI tool*

The most popular of these three tools is Postman—maybe you already know it—but I admit that using HttpRepl online is quite lovely, and if you are a fan of command-line tools, this one is for you, especially if you are a Linux pro! If you are not, I hope I made you want to try it.

ASP.NET Core Minimal APIs

ASP.NET Core 8 has introduced a new feature: minimal APIs. ASP.NET Core 8 brings more functionalities to them and allows minimal APIs to catch up on web APIs. We will discuss this throughout this book.

Why do I adore them? For the simple reason that sometimes I have to write minimalistic APIs, one or two endpoints maximum with data to manipulate quite simply. How does it work? There is no need to implement controllers, and only one file is necessary: the *Program.cs* file.

As you already know, the latter allows on its own starting an application with minimal configuration. Note that all the ASP.NET Core pipeline remains the same. I mean by this the dependency injection system and the middlewares that follow one another and manage HTTP requests and responses.

To get started, create an "ASP.NET Core Empty" project as shown in Figure 2-16.
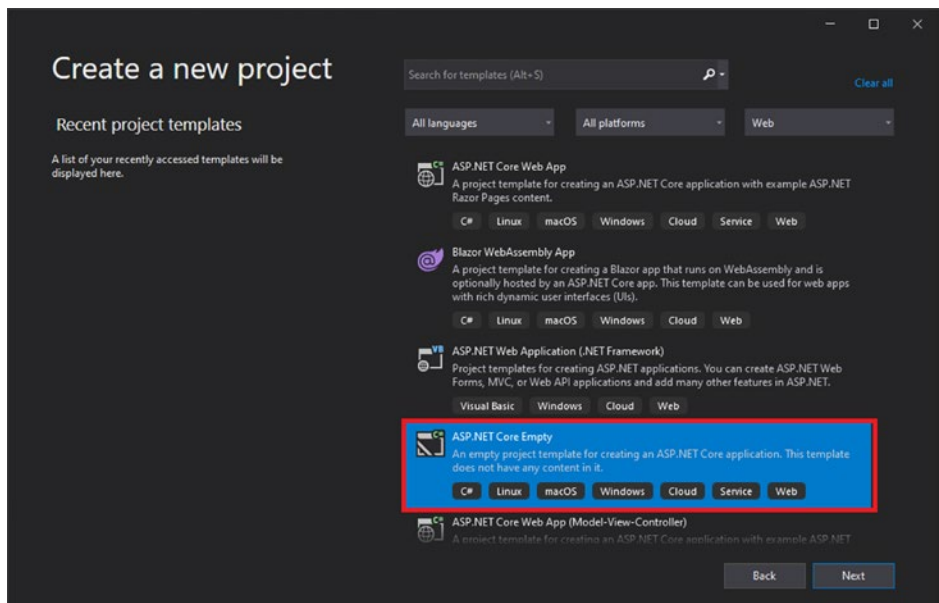
Figure 2-16. *Create an ASP.NET Core Empty project*

Note   You can also use the ASP.NET Core Web API template and unselect the "Use controllers" options to do the same.

Once you named your project, Visual Studio 2022 will create the following minimalistic project with its default endpoint, "Hello World!", as shown in Figure 2-17.
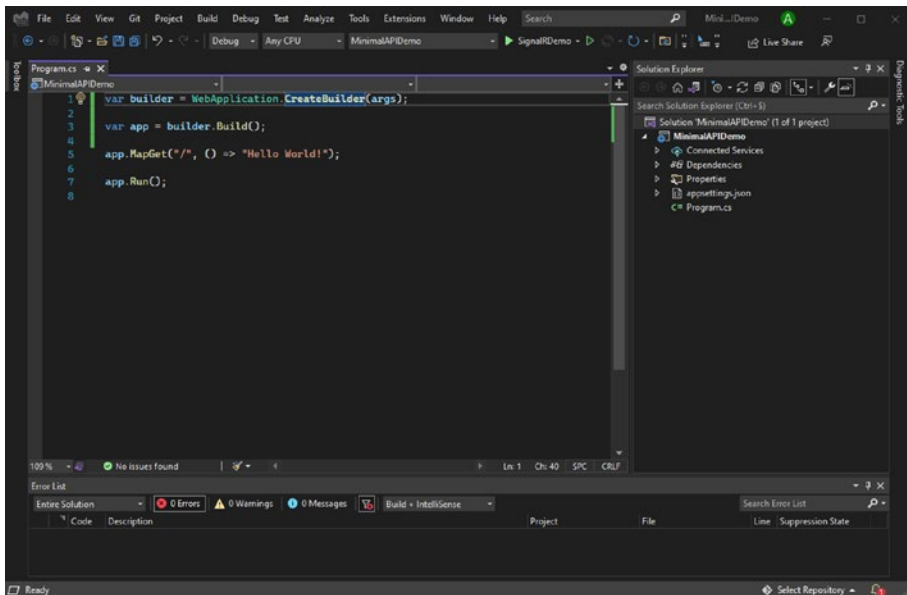
Figure 2-17. *Minimalistic ASP.NET Core project*

Figure 2-18 shows the minimal API configured to serve the Swagger documentation to reveal the *Hello* endpoint declared in the *Program.cs* file. Dependency injection is used for the *IHelloService* declared on the top of the file as a Scoped service. C# also introduces a new feature that allows developers to decorate lambda expressions with attributes, such as the *FromRoute* attribute that maps the route attribute "name" to the string parameter name.

```
IHelloService.cs        Program.cs
MinimalApiDemo
 1    using Microsoft.AspNetCore.Builder;
 2    using Microsoft.AspNetCore.Mvc;
 3    using Microsoft.Extensions.DependencyInjection;
 4    using Microsoft.Extensions.Hosting;
 5    using Microsoft.OpenApi.Models;
 6    using MinimalApiDemo.Services;
 7
 8    // Configure services
 9    var builder = WebApplication.CreateBuilder(args);
10
11    builder.Services.AddScoped<IHelloService, HelloService>();
12
13    builder.Services.AddEndpointsApiExplorer();
14
15    builder.Services.AddSwaggerGen(c =>
16    {
17        c.SwaggerDoc("v1", new OpenApiInfo { Title = "Api", Version = "v1" });
18    });
19
20    // Configure and enable middlewares
21    var app = builder.Build();
22
23    if (app.Environment.IsDevelopment())
24    {
25        app.UseDeveloperExceptionPage();
26    }
27
28    app.UseSwagger();
29    app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "Api v1"));
30
31    app.MapGet("/{name}", ([FromRoute] string name, IHelloService service) => service.Hello(name));
32
33    // Run the app
34    app.Run();
```

Figure 2-18. *An example is a minimal API that uses dependency injection attributes on lambdas and serves as an endpoint with its Swagger documentation*

Figure 2-19 shows the Swagger UI generated from the preceding code.
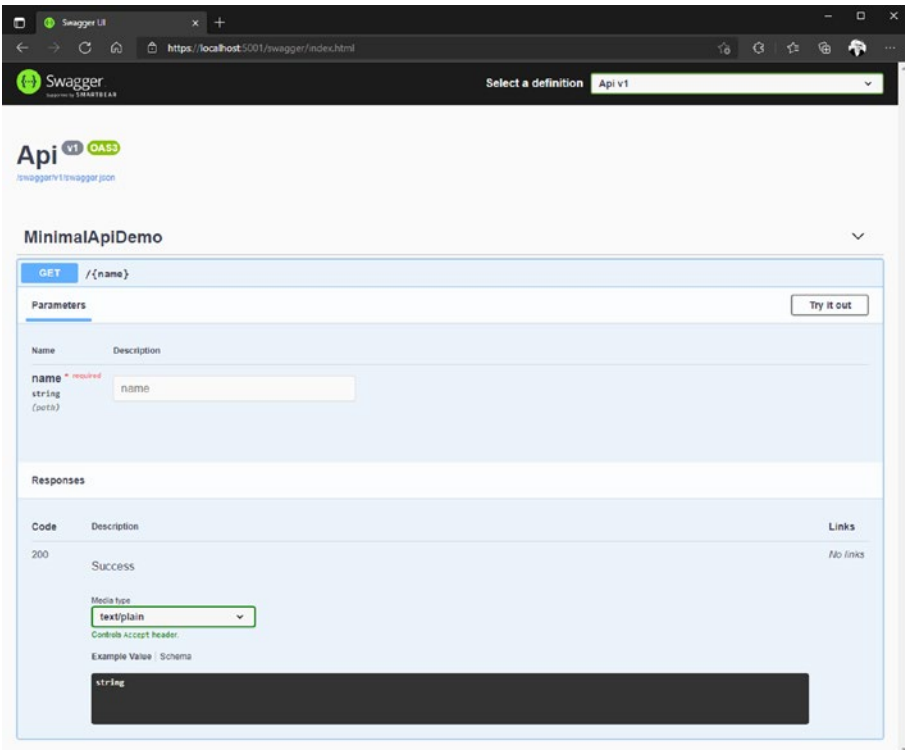
Figure 2-19. *An example of a minimal API Swagger UI*

I like this way of developing APIs. For my part, I use it almost
automatically over ASP.NET Core Web API.

Summary

In this chapter, you've learned what ASP.NET Core is, its basics, and some of the frameworks it does support. ASP.NET Core is a vibrant framework. All kinds of web technologies are supported and well documented by Microsoft. That's why it's my favorite web application development framework, and I hope you enjoy it just as much as I do.

Along with this book, you'll see how I'll build minimal APIs to show you how robust the ASP.NET Core 8 framework is.

Introduction to Application Development Best Practices

Here we are at the heart of the matter! As you might expect, we will take your time developing an API in this book. Before developing an API with ASP.NET Core 8, in this chapter, we will review the basics of any software development, whether it is an API or any other type of software, such as a mobile application or a desktop application. Ideally, any application should respect the following development fundamentals: programming basics, structuring code in a clean architecture, writing readable, maintainable (and easy to debug!) code, and keeping your application safe with the *Open Worldwide Application Security Project (OWASP)* principles.

In this chapter, you will learn a brief introduction to the following points:

- Getting the right frame of mind

- Clean architecture fundamentals

- Clean code fundamentals

- OWASP principles

> **Note**   This book assumes that you already know the fundamentals of programming, object or procedural programming, algorithms, and the basics of the C# language. Therefore, this chapter will remain more theoretical than practical since all the principles I will introduce here are well documented on the Web or in many books. This chapter aims to give you the right mindset to write clean APIs.

Getting the Right Frame of Mind

You may not know it, but *Information Technology (IT)*, particularly software development, is challenging. It requires many qualities, and I will describe them in this section.

A Basic Understanding of the Business

Here I'm talking about the minimum of the minimum, that is, understanding the basics of algorithms while knowing a programming language, such as C#, which I'll be using in this book. However, I consider that you have mastered the fundamentals of the language.

Problem-Solving Skills

Once you know the basics of algorithmics and a programming language, you must solve problems. You'll have to transpose an entire logic into a programming language to achieve your goals, and from time to time, you'll also have to deal with unexpected behavior in your software: bugs. In this book, I'll cover some techniques for debugging software.

Understanding Programming Paradigms

There are different programming paradigms, such as procedural programming and object-oriented programming. I also assume you are familiar with the basic principles of these paradigms. Throughout this book, I'll be using object-oriented programming, although I may sometimes use simple functions in the way that procedural programming requires.

Logical and Structured Thinking

This is the most complicated part! And that's what I'm going to emphasize in this book. Understanding the IT business basics, solving problems, and understanding programming paradigms can make you a programmer, but thinking logically and structurally will make you an excellent programmer! Of course, a program has to work, but more is needed. High-quality software must satisfy these rules:

1. **To be easy to understand and maintain**: What's worse than unreadable, unmaintainable code? You'll make mistakes, and lots of them, if the code isn't of the highest quality. I'm talking here about variable naming, class naming, code documentation, cyclomatic complexity, code safety, etc. I'll return to this in a moment. We're talking about clean code here.

2. **To be correctly organized**: A computer program must be divided into several layers, each containing several elements (classes) with a particular type of responsibility and reusable independently on other layers. We're talking about clean architecture here.

Clean Architecture Fundamentals

Clean architecture? If that sounds abstract, don't worry; we'll clarify it. In simple terms, it's how to organize your code and define the relationships between each piece of code. Know that there is no absolute truth about implementing a clean architecture. In this chapter, I will introduce my way of seeing things, which I have adopted (and assumed) over the years of experience I have acquired. In the industry, we often talk about the following architectures:

- Hexagonal architecture

- Onion architecture

- Domain-driven design (DDD)

- "Clean architecture"

They all describe the organization of your application layers (projects in .NET). I won't describe them here because it would be too long and could confuse you. In the meantime, I want to avoid repeating what has already been explained many times.

Also, as you may have noticed, the title of this section is called "Clean Architecture," and I mentioned this name with quotes in the preceding list. There is a difference between the two because I will talk about MY clean architecture rather than the "clean architecture" documented in other sources of information.

My clean architecture is not fundamentally different from the others. It is a preference of mine in the context of low- to intermediate-complexity API development. In the case of complex architecture, the architecture will require more attention to detail. Still, again, how I see it here will allow you to understand 99% of the web projects you will have to do in your career as an API developer.

Ultimately, the only thing that **matters** is to respect a great principle: **independence** (or weak coupling). Independence from what? Independence from the technology and from external data sources and the independence between application layers. More concretely, your application layers must be

1. **Independent of the user interface:** The user interface (API, desktop application, etc.) must function independently of your core application (business logic, data access, etc.). Throughout this book, I will use the abstraction principle to show you how to not depend on business logic.

2. **Independent of third-party libraries and frameworks:** Your application must not be strongly coupled to a particular library, or you will be dependent on it and limited in your development, especially your technical maintenance. Later in this book, I will show how to abstract these libraries and frameworks.

3. **Independent of external data access:** It must be possible to easily change databases (type of database) or switch to another type of data access, such as XML files somewhere on the network and vice versa. I will introduce some data access technologies and show you how to switch between these technologies. It also implies here the notion of abstraction.

4. **Independently testable:** Testing must be done in isolation from other software layers and technologies. To illustrate this, I will introduce unit testing later in this book, which also relies on the abstraction principle.

75

> **Note**   I mentioned several times the *abstraction* word (weak coupling). This is a crucial principle of clean code, which implies *interfaces* and *dependency injection* that I have introduced to you in Chapter 2, in the "ASP.NET Core Fundamentals" section.

If you have understood these rules, feel free to learn other architectures as mentioned previously. Again, my vision is not the absolute truth, but it works without being too complex in most cases.

Let's move on to my vision, which will convince you, I hope. The way I see it, my application is divided into a minimum of four layers, and here they are:

- A **Domain** layer will contain all our domain objects, repository interfaces (data access), and service interfaces, in other words, application contracts and abstractions. This layer does not rely on any layer. This layer is independent.

- A **Presentation** layer in this book will be an ASP.NET Core web layer that exposes APIs over HTTP. This layer is dependent on all layers. Even though the code is independent of any technology and only depends on contracts and abstractions, the application configuration needs to know what abstraction implements what concrete class. It must be done in the configuration, as I showed you in Chapter 2, in the *Program.cs* file.

- A **Business logic** (or **Application**) layer will implement business rules and orchestrate step-by-step actions from different components responsible for a particular action (e.g., data access, logging, caching data, etc.).

This layer only relies on the Domain layer. It must only know the domain contracts and abstractions. This is critical since your business logic MUST NOT depend on any technology (infrastructure). There is only one exception. This layer can depend on generic layers such as tools that help you code better in a particular situation. You can, for example, introduce a dependency on a Tools layer that implements C# classes that can be reused in any situation and don't rely on any technology. I will discuss it a little further.

- One or several **Infrastructure** layers. Infrastructure layers implement particular technology. They must rely on the Domain layer, which defines abstractions and contracts. Infrastructure layers implement them. It's always good to have one Infrastructure layer per technology. For example, if you access data via SQL and HTTP, you can design one layer for SQL data access and another for HTTP. They will be independent of each other. It's practical if you want to reuse the SQL layer and not the HTTP if you don't need it.

- Optionally, you can design a **Tools** layer, as I said before. This layer can implement anything if you are not implementing any application logic or relying on a particular technology. You can code your stuff here if you implement generic code that can be applied in any layer. For example, you can implement stuff here that transforms an array of *bytes* into a *stream* or vice versa, creates a reusable class that performs regular expressions, etc.

To give you a better idea, Figure 3-1 summarizes the interactions between the layers listed, and the **Infrastructure** layers are numbered from *1 to n*.
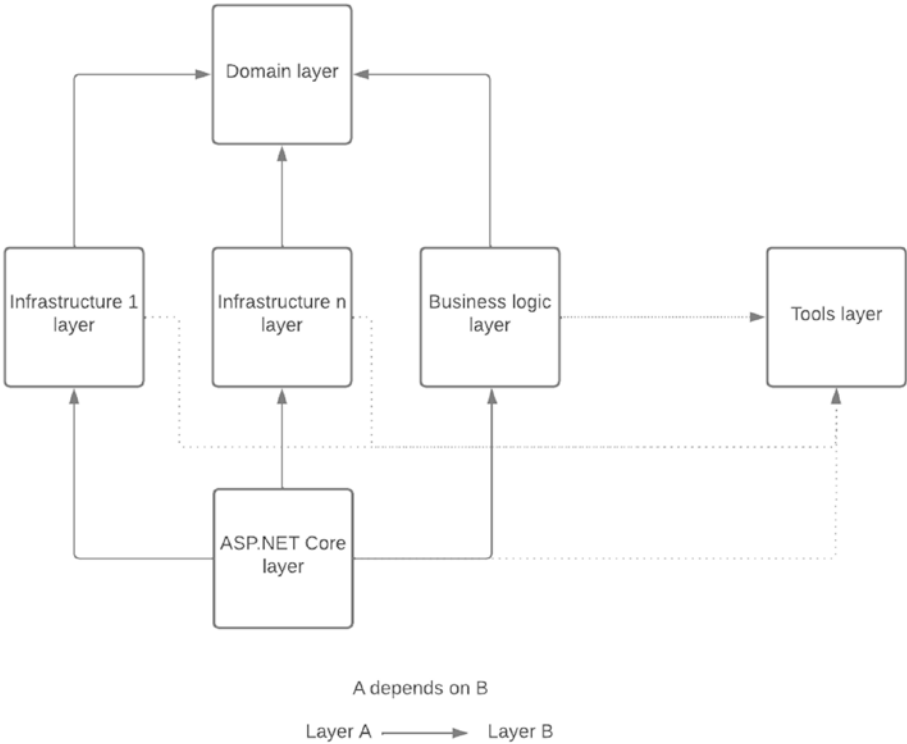
Figure 3-1. *My vision of clean architecture*

I've just explained the ideology with a diagram. I'm not going to give you a specific example at this stage, but throughout this book, I'll show you where (and how) to implement the different functionalities of your API. Once again, only the decoupling mindset matters; you'll understand that throughout this book.

I want to say a few words about design patterns. This book does not aim to teach you about design patterns, but I want to inform you that design patterns influence software architecture. A design pattern is a

specific arrangement of modules commonly accepted as best practice for solving a particular problem in software design. It sets out a way of operating a standard solution, which can be used to design different software products. There are 24 design patterns described by the *Gang of Four (GoF)*, a team of four experienced developers. Their 24 patterns are explained on their website. Don't worry; you don't need to learn them all. In this book, I'll introduce you to just a few of them, including one you already met in Chapter 2: the *Singleton*, which allows you to instantiate only one instance of a class throughout an application.

Here's the website mentioned previously: www.gofpatterns.com/.

Clean Code Fundamentals

I've already introduced you to clean architecture, which is essential and can be implemented in many ways. Still, it's also impossible for me not to make you aware of the notion of clean code. As you may have guessed, it's not enough to organize your code well, to divide it into logical and independent layers. Still, you must also ensure that the code you implement is clean. I want to look at this before we start coding beautiful REST APIs!

General Coding Fundamentals

So what's clean code? Here we go:

1. The code must be **simple**: This is the main characteristic of clean code. The simpler the code, the more readable and maintainable it will be as your software evolves. In *Information Technology (IT)* jargon, this principle is known *as Keep It Simple, Stupid (KISS)*. To achieve your goals, think effectively. No need to anticipate the unforeseeable;

that's also what we call the *You Ain't Gonna Need It (YAGNI)* principle, for example, if you try to anticipate a particular behavior when you think it won't happen. Why it's important? You will complexify your application for no valid reason!

2. Code must have a **single responsibility**: In other words, a piece of code, an instruction or a function, must have a single purpose: to solve a single problem. Why is this? Because it allows us to isolate the functionalities of an application, which will take us to point number 5 in the following. The *Single Responsibility* principle is to be found in the *SOLID* principles. I'll come back to this a little later in this section.

3. Code must **not be repeated**: This is another essential principle in programming. Don't copy and paste, and always give priority to reusability. This avoids having two identical pieces of code (which solve the same problem) evolve differently, which could lead to bugs. We call this the *Don't Repeat Yourself (DRY)* principle, which only applies if identical pieces of code solve the same problem.

4. Code must be well **isolated** from other parts of the code: Code is simple, and having only one responsibility is essential. However, for this to be entirely true, the code must meet another requirement: the *Separation of Concerns (SoC)* principle. This is not the same as the *Single Responsibility* principle, which governs the behavior of a piece of code, that is, a piece of code

is written for a specific task. For example, when you order a pizza, you choose it, pay for it, and have it delivered. The *Single Responsibility* principle means that a pizza selection function programs the "construction" of your pizza. Dispatching your order involves several functions—each of them will be responsible for making the payment and managing the delivery, which will have their independent piece of code: this is what we call the *Separation of Concerns (SoC)* principle.

5. Finally, the code must be **testable**: If all the preceding elements are respected, it should be easy to test. Testing an application, in other words, carrying out unit tests, is vital to the long-term maintainability of your application. I won't go into too much detail here, as we'll return to this in the book's last chapter.

I mentioned earlier the acronym SOLID, but I did not define it. Each letter of this acronym is the first letter of five great principles in *object-oriented programming (OOP):*

- **S**ingle Responsibility principle: I already introduced this earlier. In this book, I will use this principle as much as possible.

- **O**pen-Closed principle: This principle encourages class extension instead of modifying it when a feature needs to evolve. In other words, please create a new class and inherit it from the base class instead of reworking it. Even though it's a great principle, it's not very often applicable in an API.

- **L**iskov Substitution principle: Even though creating new classes using class inheritance is highly recommended, it's easy to overuse it (too many levels of inheritance) and destabilize a software program's functioning. This is where Liskov Substitution comes in. With this principle, a child class can remap a parent class without destabilizing the system. I can't hide the fact that this isn't always easy. Nevertheless, it's always good to know this principle, even if, in an API, it's not always applicable.

- **I**nterface Segregation principle: This principle is similar to the Separation of Concerns principle but applies to interfaces. If we take the example of ordering a pizza, we could have one interface describing the service contracts to "build" the pizza, another to pay for it, and a third to manage the delivery. In this book, I'll be using this principle, and we'll look at some concrete examples.

- **D**ependency Inversion principle: This principle aims to enforce the usage of abstraction as much as possible. The closer you are to the high level of your application (UI), the more you should rely on abstractions instead of low-level classes. This will prevent any maintenance issues if you change your low-level implementation. We have already talked about this earlier in this book; I introduced you to the dependency injection pattern. Throughout this book, I will use this pattern to make the code cleaner as much as possible.

Since this book does not intend to go deeper into OOP principles, I won't go further with this topic, but if you want to learn more about it, you can on this post, where you'll find great code samples: www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/.

Coding Style Fundamentals

Another important aspect of clean code is the coding style. Structuring your code is essential, but it must also be easy to read. What I like to do personally is to name my files correctly (I like to give explicit names to my classes, interfaces, variables, etc., well arranged in a directory).

For example, in Figure 3-2, you can see the *Download* directory containing a *Helpers* subfolder containing static classes like *AmazonS3PathBuilder.cs* and *AzureFileStoragePathBuilder.cs* and a service named *DownloadService.cs*. All files are meaningful. File names are explicit and are related to the download feature.
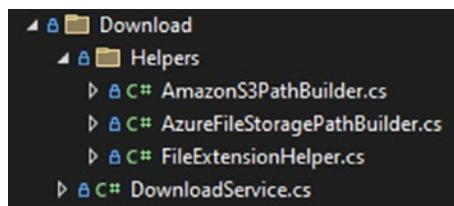
Figure 3-2. *Download directory with its classes related to the download feature*

You may have noticed that these classes have particular file names and already allow you to partially understand the intent of the implementation without looking at it. These two files contain a class for generating file paths on Amazon S3 and Azure File Storage. Amazon and Azure are both providers of software hosting solutions (essentially web-related in the cloud).

Let's look at the content of the *DownloadService.cs* file in Figure 3-3.

```
namespace Demo.Business.Download;

1 reference | Anthony Giretti, 2 days ago | 1 author, 5 changes
public class DownloadService : IDownloadService
{
    private readonly IDataStoreBusiness _dataStoreBusiness;
    private readonly IFileStorageDispatcherService _fileStorageDispatcherService;

    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions, - live
    public DownloadService(IDataStoreBusiness dataStoreBusiness,
                           IFileStorageDispatcherService fileStorageDispatcherService)
    {
        _dataStoreBusiness = dataStoreBusiness;
        _fileStorageDispatcherService = fileStorageDispatcherService;
    }

    /// <inheritdoc />
    3 references | Anthony Giretti, 17 days ago | 1 author, 2 changes | 0 exceptions, - live
    public async Task<(ProcessingStatus, string, DownloadModel)> GetFileAsync(GetFileParameters parameters)[...]
}
```

Figure 3-3. *DownloadService.cs file content*

We can see that the *DownloadService* service name is self-explanatory, as is its *IDownloadService* interface, which contains only one *GetFileAsync* contract whose intent is clear from its naming. Its incoming parameter, as is its return variable, is also explicit: a *tuple* containing a file download status, a status message, and an object containing the file data to be downloaded.

Let's now take a look at the implementation of the *GetFileAsync* function in Figure 3-4.

```
public async Task<(ProcessingStatus, string, DownloadModel)> GetFileAsync(GetFileParameters parameters)
{
    var index = parameters.FileName.IndexOf('.');
    if (index <= 0)
    {
        return (ProcessingStatus.InvalidParameters, "The file path info in the request URL is invalid", null);
    }

    (ProcessingStatus status, string message, DataStoreLoadModel dataStore) = await _dataStoreBusiness.RetrieveAndLoadAsync(parameters.dataStorePublicationId,
                                                                                                                           parameters.FileName,
                                                                                                                           parameters.CheckForAuthEnabled);

    if (status != ProcessingStatus.Success)
    {
        return (status, message, null);
    }

    // Fetch Azure
    var fileStorageProvider = _fileStorageDispatcherService.GetProvider(FileStorageTypes.AzureFileStorage);
    var downloadFileStream = await fileStorageProvider.DownloadAsync(new FileStorageParameters {
        Directory = dataStore.GetAzureFileStorageFolderName(),
        FileName = dataStore.GetAzureFileStorageFileName()
    });

    return (ProcessingStatus.Success, string.Empty, new DownloadModel
    {
        FileName = FileExtensionHelper.AdjustAudioFileExtension(dataStore.FileName, dataStore.MimeType),
        MimeType = dataStore.MimeType,
        Content = downloadFileStream
    });
}
```

Figure 3-4. *GetFileAsync method implementation*

You may have noticed that the function parameter *GetFileParameters* is unique. There is no other parameter. It's always a good practice to keep a single parameter that can take many properties because your function signature won't change if your application evolves. It helps keep your code clean.

You can see that all the variables, such as *index*, *fileStorageProvider*, and *donwloadFileStream*, are explicit.

The last point I would like to bring here is the naming convention. As you can see, I used different casing conventions for naming my variables, method parameters, methods, namespaces, classes, and fields. I used two cases: the Pascal case and the Camel case. I even used a Camel case preceded by an underscore (_) for class fields. I also use the Pascal case for properties as follows:

```
public int MyProperty { get; set; }
```

Table 3-1 shows a recap of the examples seen here.

Table 3-1. *Recap of the casing convention with examples*

Element	Casing convention	Example
Namespace	Pascal case	DownloadService
Class	Pascal case	Demo.Business.Download
Method	Pascal case	GetFileAsync
Method parameter	Pascal case	GetFileParameters
Property	Pascal case	MyProperty
Variable	Camel case	index
Method parameter name	Camel case	parameters
Field	Camel case with _	_dataStoreBusiness

Throughout this book, I'll use this way of coding as much as possible in my code examples.

Other principles of clean code can also be found in other stages of your software programming, such as

- Error handling

- Testing

- Comments (to be used sparingly)

We'll come back to these later in the book.

OWASP Principles

The *Open Worldwide Application Security Project (OWASP)* is an international organization providing recommendations on software security. OWASP develops and maintains various tools, such as documentation and videos, to make developers and companies more aware of the security of their web applications.

What interests us here from OWASP is the *OWASP Top 10*. The *OWASP Top 10* describes the most widespread attacks a web application can suffer. Without going into too much detail in this section, I'll list them here. I'll show examples in this book on how to protect your API against these attacks if it applies to the topics I will bring throughout this book. Remember that security must be your priority when designing web applications such as REST APIs! Any compromise on security shouldn't be accepted.

Here are the elements of the OWASP Top 10:

1. **Weak authentication and authorization:** Even with authentication implemented in your application, your application remains vulnerable. The most common is the brute-force attack, which consists

of trying login/password combinations hundreds or thousands of times until the right combination is found. It's effortless if a password is easy to guess. We'll discuss a solution, *Rate Limiting*, which you'll see in Chapter 5. There is also a solution to protect against this attack, two-factor authentication, but I won't discuss it in this book.

2. **Injection:** Particularly with SQL databases (but also NoSQL such as MongoDB or LDAP to identify a person on a network), there is a way of obtaining information illegally by corrupting data (which has not been verified, as a user should never be trusted), thus diverting the purpose of the initial request to the server. The best known is SQL injection. I'll discuss this in Chapter 6, when I show you how to access data. Another type of attack is *Cross-Site Scripting (XSS),* which sends data containing executable code. Data can be sent over an endpoint that saves data to be displayed further. These data may contain, for example, JavaScript code that may inject some unexpected content or, even worse, steal authentication cookies and send them to a destination that will steal and use your identity. I will provide a concrete example in Chapter 4 regarding input validation.

3. **Broken access control:** Authentication is often insufficient to protect access to sensitive data or actions. In an enterprise, not everyone can have the same privileges in an application, which is why some people, and not every application user, are given additional authorization to access sensitive

data, such as a customer's banking information. In Chapter 10, I'll show you how to implement user authentication and manage authorizations.

4. **Insufficient logging and monitoring:** Logging and monitoring can help identify attacks, such as brute-force attacks, or detect an increase in activity on a web application when activity should be low. In Chapter 8, I'll talk about observability, showing you how to log and trace HTTP requests and use them for diagnostics.

5. **Insecure data integrity:** Serialization/ deserialization of data may lead to security breaches, such as allowing an attacker to execute malicious code on the server. Except for the fact we will validate input data on API endpoints in Chapter 4, I won't go further with this kind of security issue in this book.

6. **Cryptographic failures:** Non-encryption of specific data can lead to vulnerabilities in an application, making it a prime target for hackers. I won't go into this here, as we'll transport data between a client and a server over HTTPS via an API. This doesn't necessarily mean encrypting data in the more specific context described in this book.

7. **Weak application design:** Occasionally, there are use cases where an application can be used to abuse a benefit, for example, a promotion that should only apply once but can be used several times. I remember being a student and having a prepaid SIM card with a mobile operator. To top up my

account, I used a code that was supposed to be a one-time use code but wasn't. Several people could use the same code for two minutes. This flaw took months to correct, and the company lost money. There is no specific chapter about it in this book.

8. **Weak security configuration:** An application can be vulnerable if, for example, it uses accounts whose passwords never expire or, worse, if the passwords are easy to guess or if an unused login/password pair remains active. Manage all active accounts and change their passwords regularly.

9. **SSRF:** *Server-side request forgery (SSRF) vulnerabilities* occur whenever a web application retrieves a remote resource without validating the URL provided by the user, enabling an attacker to force the application to send a specially crafted request to an inappropriate destination. I won't go any further here, as this applies more to an HTML web application than an API.

10. **Obsolete component:** Many applications often don't update their frameworks or libraries. For example, Microsoft publishes updates for .NET frameworks to close a security gap. Apart from telling you that you must always keep your application up to date, I have nothing to show you in this book for this topic.

OWASP provides the *OWASP Secure Headers Project (OSHP).* This project describes HTTP response headers that can be added to your application to make it safer. You can take a look at this address: `https://owasp.org/www-project-secure-headers/`. The good news is that there is

an implementation for ASP.NET Core, and you can find it on <u>Nuget.org</u> since it's a Nuget package. Installation is well documented and straightforward to execute. You can find it here: `www.nuget.org/packages/ OwaspHeaders.Core#readme-body-tab`.

Summary

This chapter shows the minimum acceptable clean code and architecture you must implement in an API. We can go further by using tools like code formatting or another famous tool named ReSharper (which is not free) that can help you improve your code. I want you to learn here to get the right mindset to keep code clean instead of using tools that do the job for you. You also don't need to know the 24 design patterns. Most of the time, they are overkill in many situations; they are only there to help resolve a particular implementation of a problem, which is not the intention of this book. As I said before, there is no compromise on the security purpose. It must be your obsession!

CHAPTER 4

Basics of Clean REST APIs

You're looking forward to developing APIs! Here we are. Let's look at the most basic operations with minimal APIs in ASP.NET Core 8. These are the most common features; you'll use them in every API you develop. First, you'll need to expose comprehensible URIs, validate the parameters sent by the API consumer, and do object mapping, that is, transform your objects into other objects specific to the application domain. You'll also need to manage the correct HTTP status codes for each type of operation you want to perform; you'll learn how to download and upload files, stream elements to your API's client, as well as version your API and expose your API's endpoints so that your clients understand how to invoke your endpoints. Handling *Cross-Origin Resource Sharing (CORS)* is also a challenge; we will see how to deal with that. In this chapter, you'll learn about each of the following points:

- Routing with ASP.NET Core 8

- Parameter binding

- Validating inputs

- Object mapping

- Managing CRUD operations and HTTP statuses

- Downloading and uploading files

- Streaming content

- Handling CORS

- Managing API versions

- Documenting APIs

Routing with ASP.NET Core 8

Do you remember Chapter 1, where I introduced you to HTTP and REST principles? We will put URL writing into practice using ASP.NET Core 8's routing feature. The beauty of ASP.NET Core 8 is that the framework offers improvements over previous versions of ASP.NET Core, and I'll show you how in a few lines. In this section, we'll look at two different ways of managing *routing*, the simplest of which is to write a URL applicable to a single endpoint and the second, called *RouteGroups*, of which lets you manage reusable route portions across a route group.

But first, let's define what *routing* is.

ASP.NET Core Routing

Routing is the ability to respond to an HTTP request from any client. The system, in this case, ASP.NET Core, analyzes (through pattern matching) the HTTP request and determines what to do with it, that is, find out which endpoint corresponds to a requested URL. If no endpoint is found, an HTTP 404 (Not Found) error is returned to the client. Figure 4-1 summarizes ASP.NET Core routing.
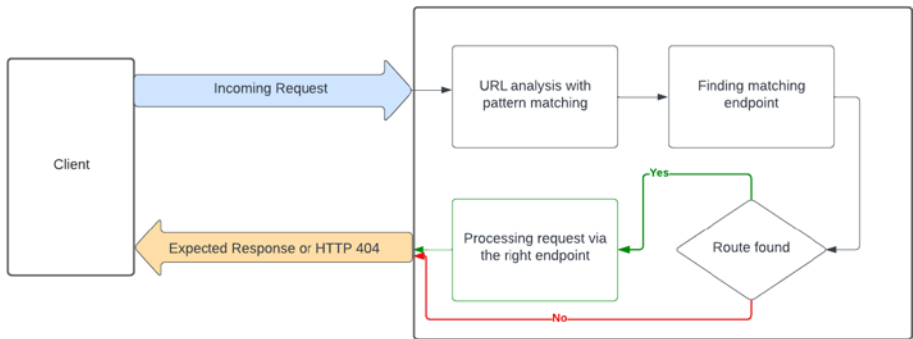
Figure 4-1. *ASP.NET Core routing*

There's no need to go into how pattern matching works here. We will see how to set up the HTTP verb that will be used to write routes and apply constraints to these routes by enforcing allowed values on route parameters.

Setting Up the Correct HTTP Verb

ASP.NET Core 8 makes it easy since each verb has its dedicated method to map a route to a specific verb. They all take as parameters a route name and a delegate. The method name is straightforward, so you can't get it wrong. Table 4-1 shows all available HTTP verbs with their methods.

Table 4-1. *HTTP verbs and their associated methods*

HTTP verb	Method
GET	MapGet
POST	MapPost
PATCH	MapPatch
PUT	MapPut
DELETE	MapDelete
Other verbs	No method

The following code snippet provides an example of the signature of the POST method:

```
app.MapPost("/yourRouteName", () =>  /*  Do action */);
```

As you can see, some verbs like OPTIONS, TRACE, and HEAD don't have their method. But don't worry; you can use the *MapMethods* method, which can take several verbs in parameters for the same route. An example is the following snippet:

```
app.MapMethods("/routeName", new List<string> { "OPTIONS",
"HEAD", "TRACE" }, () => { /* Do action */});
```

The delegate can take parameters, but I will show you further in this chapter and the next chapter when I introduce you to custom parameter binding.

Writing Routes

Writing routes is pretty straightforward. Writing pretty routes while respecting the principles can sometimes be a pain, but most of the time, it's effortless. It depends on how you want to respect REST principles. The best practice is to respect REST principles by defining meaningful routes for better readability. This is a crucial point in writing clean REST APIs. Figure 4-2 shows two examples with the GET HTTP verb. The first one, with the parameters highlighted, makes the whole route **easier** to read. Highlighting was not available before ASP.NET Core 8. Route parameters are automatically bound to the parameter(s) of the lambda function that represents the endpoint's code to respond correctly to the client's request. The second example takes no parameters, resulting in a static route. I'm showing you this code through a figure to show highlighted parameters.

```
1   var builder = WebApplication.CreateBuilder(args);
2   var app = builder.Build();
3
4   app.MapGet("/countries/{countryId}", (int countryId) => $"CountryId {countryId}");
5
6   app.MapGet("/countries", () => new List<string> { "France", "Canada", "Italy" });
```

Figure 4-2. *Basic routing*

As you can see, route names are **meaningful**. The first identifies a country from its ID among a list of countries defined by the base route */countries*. The second endpoint represents the list of countries with the route */countries*.

ASP.NET Core 8 lets you pass many parameter types in a route, obviously because of all the possibilities in terms of the type of data that routing and REST principles allow, such as the following primitive variables:

- bool

- byte

- sbyte

- short

- ushort

- int

- uint

- long

- ulong

- char

- double

- decimal

- float

It is also possible to pass more complex parameters (objects) into a route, but these can be easily serialized to a string, for example:

- DateTime

- Guid

Listing 4-1 shows routes with DateTime and Guid as parameters.

Listing 4-1. Example of routes containing DateTime and Guid

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

app.MapGet("/date/{date}", (DateTime date) => date.ToString());

app.MapGet("/uniqueidentifier/{id}", (Guid id) =>
id.ToString());
app.Run();
```


Note   Any failed attempt to bind a parameter (when a matching route with the correct HTTP verb is found) to its expected type declared in the lambda method will lead to an HTTP 400 Bad Request. Regarding DateTime, you must be more careful since the binding works only with the DateTime value in its invariant culture: yyyy-MM-dd. I will return to parameter binding in the next section of this chapter.

For example, Figure 4-3 shows what ASP.NET Core returns when a string is bound in place of an expected integer.
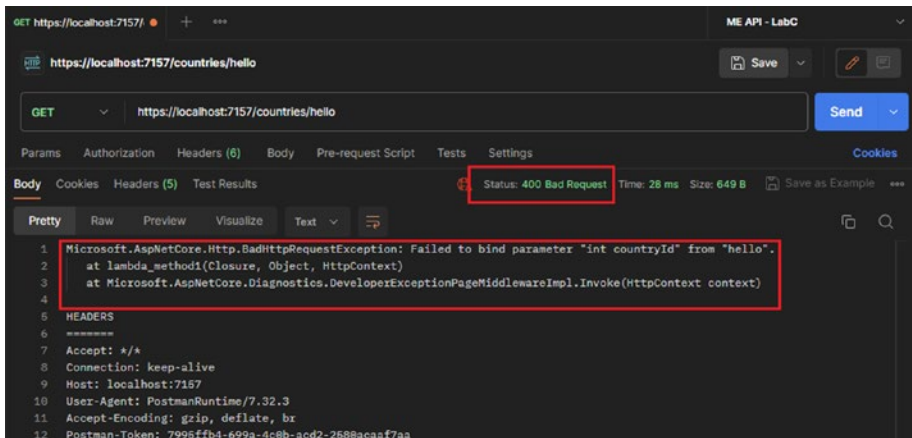
Figure 4-3. *Example of attempting to bind a string instead of an integer, which leads to the HTTP 400 Bad Request response*

On the other hand, if a route is found, but the HTTP verb does not match this route, an HTTP 405 Not Allowed error will be thrown. For example, Listing 4-2 shows an endpoint that handles PUT and PATCH verbs for the same route.

Listing 4-2. Example of an endpoint that handles PUT and PATCH verbs

```
app.MapMethods("/users/{userId}", new List<string> { "PUT",
"PATCH" }, (int userId, HttpRequest request) =>
{
    var id = request.RouteValues["id"];
    var lastActivityDate = request.Form["lastactivitydate"];
    /* code to update user */
});
```

If you try to invoke this route with the POST verb, the response will be HTTP 405 Not Allowed, as shown in Figure 4-4.
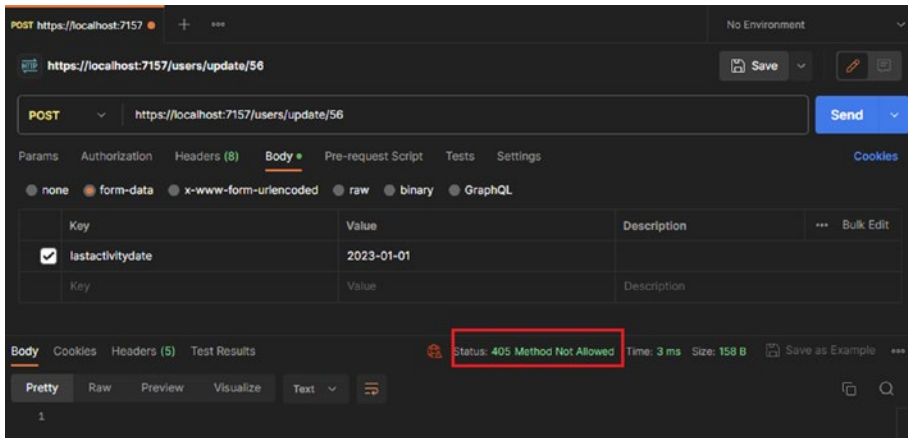
Figure 4-4. *Example of attempting to invoke a route with an incorrect verb that leads to the HTTP 405 Not Allowed response*

Another behavior remains possible, which I mentioned right at the beginning of the section; if no endpoint is found (no matching route, whatever the verb), an HTTP 404 Not Found error will be returned in the response.

Figure 4-5 shows a route that does not exist and returns an HTTP 404 Not Found.
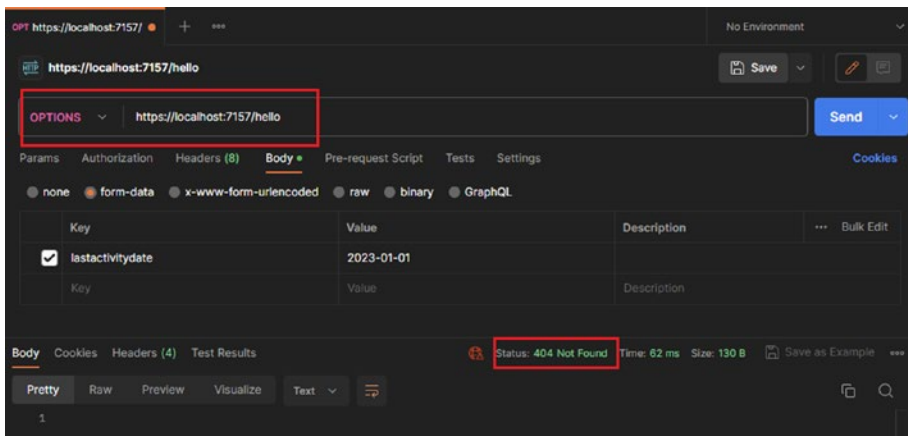
Figure 4-5. *Example of attempting to invoke a route that does not exist, which leads to the HTTP 404 Not Found response*

Table 4-2 summarizes possible behaviors after invoking an ASP.NET Core 8 minimal endpoint.

Table 4-2. *Routing behaviors with their possible responses*

Behavior	Response
The route matches, and the verb is correct but fails to bind parameters.	400 Bad Request
The route matches, but the verb is not correct.	405 Not Allowed
The route does not exist, whatever the verb.	404 Not Found
The route matches, the verb is correct, and parameter binding is working.	2XX Success

Clean REST APIs require writing proper URLs, and ASP.NET Core 8 ideally helps you respect REST principles and handle all scenarios when a mistake is made, so you can adjust yourself when an error occurs.

ASP.NET Core 8 also allows you to apply constraints to your routes, on parameters, to be precise, and this is what we will see in the following subsection.

Using Route Constraints

ASP.NET Core 8 lets you set constraints on your route parameters. These constraints allow you to filter and restrict access to your API if one or more constraints are unmet. Please note: this is not the same as validating parameters. We'll look at parameter validation in a further section of this chapter.

Listing 4-3 shows a constraint on a parameter named "provinceId", which must be an integer.

Listing 4-3. Example of a constraint applied on the "provinceId" parameter, which must match the integer type

```
app.MapGet("/provinces/{provinceId:int}", (int provinceId) =>
$"ProvinceId {provinceId}");
```

As you can see, the syntax is quite simple and always follows this pattern:

```
{ParameterName:DataType}
```

You might be wondering what happens when a constraint is not respected. Well, it's simple: ASP.NET Core 8 will return an HTTP 404 Not Found! Figure 4-6 shows the error with the previous piece code I showed you.
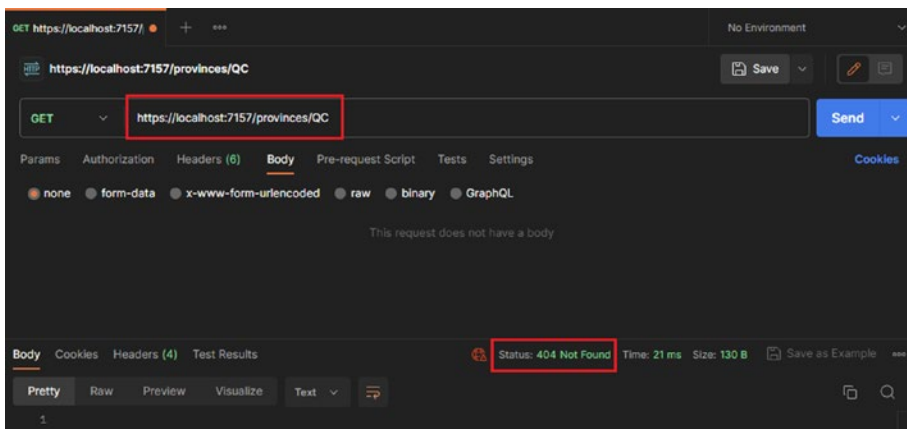
Figure 4-6. *Example of attempting to pass a string on a route that enforces an integer as a constraint, which leads to an HTTP 404 Not Found*

Why would this happen? Well, it's completely logical: a constraint defines the integrity of a route, and a constraint is just as important as the name of the route itself. Consequently, if a constraint is not respected, ASP. NET Core 8 will consider that your HTTP request cannot find the route you're looking for and will return an HTTP 404 Not Found error. You won't get an HTTP 400 Bad Request error because the binding has failed (casting a string into an integer in the example I've just shown you) because the constraint check is carried out before any parameter binding.

Now you understand why I don't recommend using constraints to validate your parameters: if you try to do this, you won't know whether your HTTP 404 Not Found is caused by a constraint error or by writing your route name. Applying constraints isn't necessarily a bad practice in itself, and it's simply to prevent you from ending up with errors that are difficult to interpret. Expecting an integer rather than a string can be helpful since your route expects an integer, so anything else would invalidate your route. However, ASP.NET Core 8 lets you apply a whole range of constraints.

Table 4-3 shows the different types of constraints that can be applied to your routes, where "p" represents the parameter to apply the constraint on, "n" represents any number, and "\\..." represents any regular expression.

Table 4-3. *All available route constraints on parameters in ASP.*
NET Core 8

Constraint	Constraint pattern	Description
int	{p:int}	Enforces an integer
bool	{p:bool}	Enforces a Boolean (true or false)
datetime	{p:datetime}	Enforces a DateTime
decimal	{p:decimal}	Enforces a decimal
double	{p:double}	Enforces a double
float	{p:float}	Enforces a float
guid	{p:guid}	Enforces a Guid
long	{p:long}	Enforces a long
minlength	{p:minlength(n)}	Enforces a minimum length
maxlength	{p:maxlength(n)}	Enforces a maximum length
length	{p:length(n)}	Enforces a precise length
length (min ,max)	{p:length(n1, n2)}	Enforces a range of acceptable length
min	{p:min(n)}	Enforces a minimum integer value
max	{p:max(n)}	Enforces a maximum integer value
range	{p:range(n1, n2)}	Enforces a range of acceptable integer values
alpha	{p:alpha}	Enforces alphabetical (non-case sensitive) characters
regex	{p:regex(\\...)}	Enforces a regular expression
required	{p:required}	Enforces a non-nullable parameter

From the *minlength* constraint to the end of the table, you can see that you can perform some validation of parameters, and I don't recommend using them as I said before; it can lead to unexpected behavior, and you might be confused more than it should help you design routes. On my end, I only allow myself to use constraints on the parameter type, validating that I'm expecting an integer, DateTime, etc.

ASP.NET Core 8 allows you to chain constraints; the following code snippet chains two constraints. The first one enforces the parameter to be an integer, and the second one enforces the maximum value of 12:

```
app.MapGet("/provinces/{provinceId:int:max(12)}", (int
provinceId) => $"ProvinceId {provinceId}");
```

Once again, I don't recommend it since it's a parameter validation, and it's not the routing feature's responsibility to validate parameters.

ASP.NET Core 8 also allows you to write custom constraints, but I won't teach you this since it's a lousy practice to me. If you still want to learn about it, you can find the Microsoft tutorial here: https://learn. microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspne tcore-8.0#custom-route-constraints.

To conclude, I strongly suggest not abusing route constraints since parameter validation MUST return an HTTP 400 Bad Request, according to RFCs.

RouteGroups

When you expose a certain number of endpoints, especially those belonging to the same functionality—for example, you have a list of endpoints whose scope is to manage countries—using the route grouping functionality in ASP.NET Core can be helpful. This can be useful because it allows to

- Isolate your routes in a specific function (and the implementation of your route if you wish).

- Take advantage of this grouping to establish access rules to these endpoints, such as defining a common URL trunk and others, such as a specific authorization, but we'll see about that in the next chapter.

We'll keep it basic here to introduce you to the functionality and then develop it further with the features I'll introduce later. Let's get back to the definition of a common URL trunk. Imagine three endpoints sharing the same scope, countries, as I said, in which the URL trunk is identical, that is, starting with */countries*. Here's how we could isolate the three endpoints in a separate function—Listing 4-4 shows a *GroupCountries* extension method on the *RouteGroupBuilder* object grouping the following endpoints:

1. List of countries.

2. Get a country by its ID.

3. Get a country's languages.

The code is deliberately kept simple to explain the grouping functionality.

Listing 4-4. Example of three different endpoints that manage countries' data

```
namespace AspNetCore8MinimalApis.RouteGroups;
public static class MyGroups
{
    public static RouteGroupBuilder GroupCountries(this
    RouteGroupBuilder group)
    {
        var countries = new string[]
```

```
    {
        "France",
        "Canada",
        "USA"
    };
    var languages = new Dictionary<string, List<string>>()
    {
        { "France", new List<string> { "french" } },
        { "Canada", new List<string> { "french",
        "english" } },
        { "USA", new List<string> { "english",
        "spanish" } }
    };
    group.MapGet("/", () => countries);
    group.MapGet("/{id}", (int id) => countries[id]);
    group.MapGet("/{id}/languages", (int id) =>
    {
        var country = countries[id];
        return languages[country];
    });
    return group;
    }
}
```

As you can see, all these methods are regrouped in the *GroupCountries* extension that must be now registered on the ASP.NET Core pipeline, in the *Program.cs* file, as shown in Listing 4-5.

Listing 4-5. Registering the countries' route group with the GroupCountries method

```
using AspNetCore8MinimalApis.RouteGroups;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGroup("/countries").GroupCountries();

app.Run();
```

Before registering the group of routes defined previously, we need to define the URL trunk, and we can achieve that with the *MapGroup* extension method, which takes the URL trunk name as a parameter. Once defined, all routes will inherit from the same trunk. It will give the following URLs:

- /countries

- /countries/{id}

- /countries/{id}/languages

You may have noticed the slash ("/") by itself on the first endpoint; if your route (the right part after the trunk, in fact) does not contain any characters, you can keep the "/" or omit it. On my end, I prefer to keep the "/" just by convention. I like to have a slash at the beginning of any portion of the route. It's only my preference.

It's convenient, but we can go further again with the route grouping. What I mean there is we can not only assign a common trunk to a URL for the same group, but we can also reuse constraints on several endpoints. Have you seen the two last endpoints? They share the same constraint on the ID: {id}. We can variable the constraint on the ID by creating the *idGroup* variable obtained from the *MapGroup* method, which takes the constraint definition method, which gives the following in Listing 4-6.

Listing 4-6. Variable the constraint on the ID of a country within the GroupCountries method

```
var idGroup = group.MapGroup("/{id}");
idGroup.MapGet("/", (int id) => countries[id]);
idGroup.MapGet("/languages", (int id) =>
{
    var country = countries[id];
    return languages[country];
});
```

You only have to reuse the *idGroup* variable on the other endpoint, and they will inherit from the constraint on the ID.

It's an exciting feature since you can chain constraints or URL portions ad infinitum. But be careful not to overdo it, or you'll end up with the opposite of the desired effect: readability or maintainability.

I find route grouping extremely practical, and I use it when I have a lot of endpoints to implement, to avoid rewriting identical code or route names. It simply applies the *KISS* principle I mentioned earlier in this book.

Parameter Binding

In the previous section, I told you a little about parameter binding. I gave you the example of the HTTP 400 Bad Request error when a route parameter cannot be bound to the function's parameter executing your request. That was the most simplistic example I could think of, but in this section, I will introduce you to the fundamentals of parameter binding so that you can understand how it works.

What's Precisely Parameter Binding?

Parameter binding means that ASP.NET Core takes the parameters of an HTTP request and converts them into typed parameters passed to the function that will handle an HTTP request. In the previous section, I showed an example with an integer, but ASP.NET Core 8 is fully capable of binding primitive parameters (see the previous section for the list) as well as more complex types such as

- Collections (lists, dictionaries, arrays)

- Any complex object, except those that contain a recursion, that is, an object that contains its type as a property (applies on minimal APIs only)

- Services that can be injected by dependency (I will show an example in the next section)

Listing 4-7 shows the *Address* class that can't be bound because of the recursivity of the *Address* type.

Listing 4-7. Address class that contains itself as a property where data binding can't be made on minimal APIs

```
public class Address
{
    public int StreetNumber { get; set; }
    public string streetName { get; set; }
    public string StreetType { get; set; }
    public string City { get; set; }
    public string Country { get; set; }
    public int PostalCode { get; set; }
    public Address AlternateAddress { get; set; }
}
```

> **Note**   I declared the Address class as a class. It could be instead a record class or a struct since Adress stands for storing data only. It's up to you to choose what you want to use.
>
> Parameters must be primitive types or classes; records, objects that behave like value types and not like reference types in C#, are not supported.

Parameter Binding by Example

Now that you know what kind of data you can bind from your HTTP requests, I'll show you from which element of an HTTP request you can bind your parameters. You already know the first one: route parameters. ASP.NET Core 8 (minimal APIs again) supports parameters from

- Routes

- QueryString

- Body as JSON-only data

- Body as form data (key/value pair)

- Headers

- Others, including class instances from the dependency injection system and custom binding (I'll come back to this later in this book)

ASP.NET Core 8 allows you to bind parameters explicitly, which means you can annotate parameters from different sources with attributes. It is also possible to combine several parameters in a single function. Still, you will have to separate your parameters explicitly from others that don't

come from the same source. I will show you some examples. First, I want to show you Table 4-4, which summarizes the different bindings with the explicit attribute to be used.

Table 4-4. *Parameter binding with the proper binding attribute on ASP.NET Core 8 minimal APIs*

Data source	Binding attribute
Routes	FromRoute
QueryString	FromQuery
Headers	FromHeaders
Body	FromBody
Forms	FromForm

Note   The query string and headers only support *arrays* as parameters, while the body as JSON data and body as form data support *arrays*, *list*, and *dictionaries*.

I'll show you a series of examples to illustrate what I mean. Listing 4-8 shows a POST request attempting to create an address whose data come from the request body, using the same class as the previous Listing 4-7, without the *Address* class recursion as an *AlternateAddress* property that I've removed.

Listing 4-8. Creating an address from a POST request where data come from the request body as JSON data

```
app.MapPost("/Addresses", ([FromBody] Address address) => {
    return Results.Created();
});
```

Ignore for now the result returned by the endpoint (*Results.Created()*); I will go back to this in a further section in this chapter since I will introduce you to the static *Results* class that allows you to return the proper HTTP status. If you try to put a breakpoint to see if the parameter binding worked, you should observe the following in Figure 4-7.

Figure 4-7. *Example of parameter binding on an Address object from the request body as JSON data*

Figure 4-8 shows the request performed by Postman.
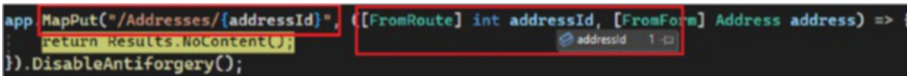
Figure 4-8. *Example of a POST request trying to create an address with Postman*

Now let's combine a route parameter with parameters whose values come from an HTML form. Listing 4-9 shows a PUT request attempting to update an *address*, with the address ID in the route and the data to be updated from the *body form data* (from an HTML form).

Listing 4-9. Updating an address from a PUT request where parameters come from the request form and the route
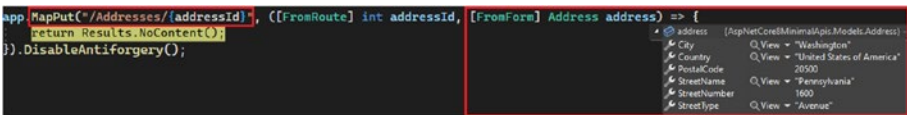
```
app.MapPut("/Addresses/{addressId}", ([FromRoute] int
addressId, [FromForm] Address address) => {
    return Results.NoContent();
}).DisableAntiforgery();
```

As you can see, the parameters from the *Route* and *Form* are explicitly separated, and the parameter binding works like a charm, as shown in Figures 4-9 and 4-10.

Figure 4-9. *addressId parameter correctly bound when explicitly treated as a route parameter*

Figure 4-10. *address parameter correctly bound when explicitly treated as a form parameter*

You may have noticed the presence of the *DisableAntiForgery* extension method. Any request made from an HTML form (form data) that requires the *FromForm* attribute on ASP.NET Core must handle the *AntiForgery* feature. The *AntiForgery* feature prevents *Cross-Site Request*

Forgery (XSRF/CSRF) attacks in ASP.NET Core, and Microsoft had made the AntiForgery feature mandatory at the very last moment when I was about to release this book. Consequently, I have updated any code that involves the *FromForm* attribute at the last moment by adding the *DisableAntiForgery* extension method to disable the AntiForgery feature. Any endpoint that implements form data will crash (in development mode) or generate a warning (in production mode) unless you disable the AntiForgery feature or you implement the AntiForgery by generating a token as shown in the following link: https://devblogs.microsoft. com/dotnet/asp-net-core-updates-in-dotnet-8-preview-6/#complex-form-binding-support-in-minimal-apis. If you try to put them in the same object, with the binding attribute on each property, as shown in Listing 4-10, it won't work and will lead to unresolved parameters. ASP. NET Core 8 (for minimal APIs) cannot bind multiple sources in the same object and will prioritize data from the *body* or the *form*. Initially, I didn't want to tackle the subject, because I didn't want to create an endpoint to return a validation token to the client to validate the forms, as these are often generated in JavaScript with libraries such as React, Angular, or VueJs and not by ASP.NET Core, and this would have required an additional call to our API to retrieve a validation token. I don't want to do this for performance reasons, as I like to save on HTTP calls.

Listing 4-10. Parameter binding property by property, mixing data sources

```
public class Address
{
    [FromRoute]
    public int AddressId { get; set; }

    [FromForm]
    public int StreetNumber { get; set; }
```

```
    [FromForm]
    public string StreetName { get; set; }

    [FromForm]
    public string StreetType { get; set; }

    [FromForm]
    public string City { get; set; }

    [FromForm]
    public string Country { get; set; }

    [FromForm]
    public int PostalCode { get; set; }
}
```
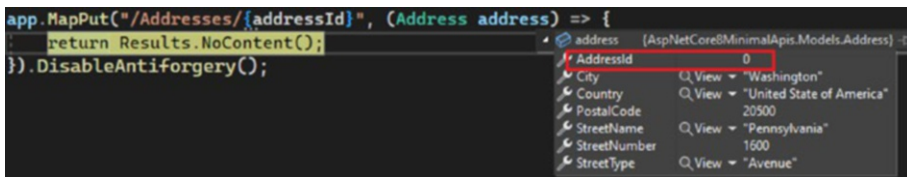
To illustrate the fact some parameters are not bound, let's take a look at Figure 4-11, where you can see the *AddressId property* remaining unbound and its value remaining 0 when I attempted to pass an integer with a value greater than 0.

Figure 4-11. *AddressId is unbound when different parameter binding is performed in the same object*

You may have noticed I did not put any parameter binding attribute on the *Address* class name in the lambda method. ASP.NET Core 8 supports the declaration of parameter binding on attributes and not necessarily on the class itself. On my end, I prefer to add a single attribute to the class in the lambda method. The preceding example works fine if I remove the

Id property and keep other attributes as is. Finally, Figure 4-12 shows the PUT request performed with Postman.
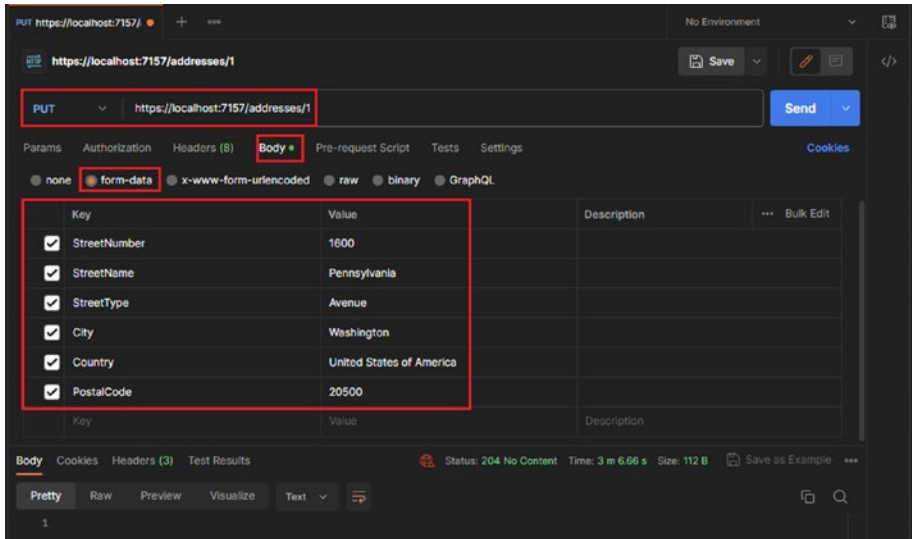
Figure 4-12. *Example of a PUT request trying to update an address from form data with Postman*

Let's use the query string and headers parameters with a GET request. Let's imagine an endpoint that returns a list of addresses based on GPS coordinates, a *coordinates* parameter passed in the headers, and a parameter passed in the query string. This *limitCountSearch* parameter determines the maximum number of elements the query returns. Listing 4-11 shows what this might look like.

Listing 4-11. Example of a GET request where parameters come from headers and queryString
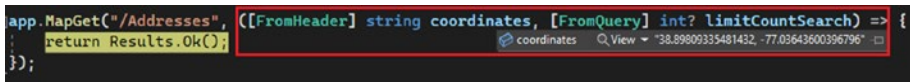
```
app.MapGet("/Addresses", ([FromHeader] string coordinates,
[FromQuery] int? limitCountSearch) => {
```

```
    return Results.Ok();
});
```

QueryString parameters tend to be optional. Because they are not mandatory to make the route work, I strongly suggest you annotate them nullable when the expected type as a parameter is not nullable. In my example, I added the question mark to the int parameter *(limitCountSearch)* since query parameters are not mandatory. Figures 4-13 and 4-14 illustrate the preceding example.

Figure 4-13. *coordinates parameter bound from the headers*

Figure 4-14. *limitCountSearch parameter bound from the queryString*

To finish, Listing 4-12 shows how to pass in the query string and headers arrays of primitive data.

Listing 4-12. Example of GET requests where an array of parameters come from the headers and the query string

```
// Represents ?id=1&id=2
app.MapGet("/Ids", ([FromQuery] int[] id) =>
{
    return Results.Ok();
});
```

```
app.MapGet("/Languages", ([FromHeader(Name = "lng")]
string[] lng) =>
{
    return Results.Ok();
});
```

As you can see, it's pretty straightforward. Regarding the IDs on the queryString, you only have to pass IDs as follows: `?id=1&id=2&id=3.`

However, you may notice that I added the property `Name = "lng"` for the second example to let ASP.NET Core 8 know that the language parameters (an array) in the headers are named *lng*. I did not tell you before, but ASP.NET Core 8 allows you to customize parameter names to get bound. This applies to any parameter binding attributes. If you want to customize a single property on an object, you can use the preceding example with the *Address* class, where I added the *FromFrom* attribute on properties and not on the class itself in the lambda function.
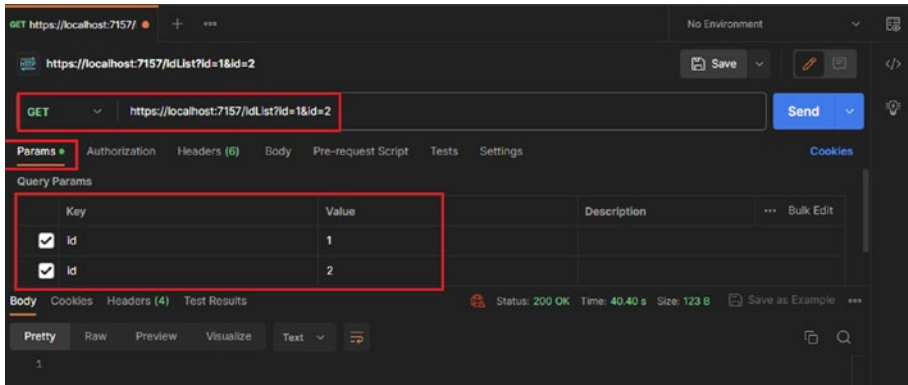
Figures 4-15 and 4-16 show the data binding operating on the GET /Ids and the GET /Languages endpoints.

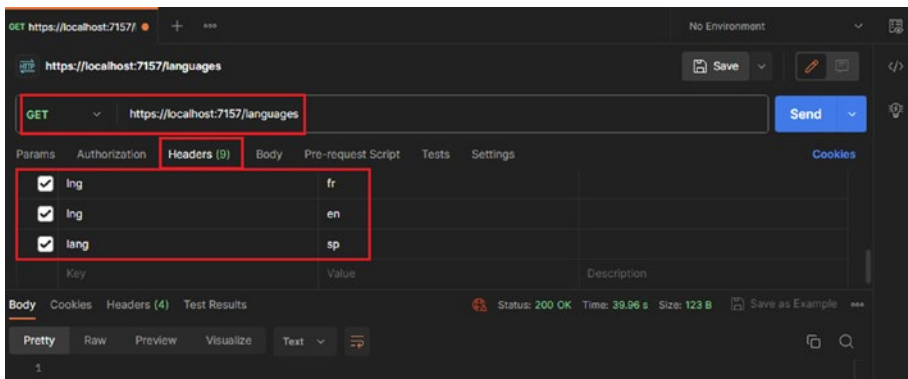*Figure 4-15. id parameter (array) bound from the query string*

Figure 4-16. lng parameter (array) bound from the headers

Now let's take a look at what the Postman request looks like for each in Figures 4-17 and 4-18.

***Figure 4-17.** Passing an array of IDs in the query string with Postman*

***Figure 4-18.** Passing an array of strings in the headers with Postman*

Regarding Figure 4-18, you may notice that the third parameter, *lang*, has not been bound if you check Figure 4-16. I voluntarily changed the language from *lng* to *lang* to show you how the customized data binding by name works.

All these examples represent the most common and frequently used cases. You could pass complete objects in the headers and the query string

in less frequent cases. However, passing complete objects as parameters to an HTTP request, for example, is for when you need to perform multi-criteria searches. Now that you've understood the principle of parameter binding, I don't need to show you any more examples. I'll return to this in Chapter 5 when it comes to showing you customized examples of parameter binding.

There's just one more thing I'd like to talk to you about. Using parameter binding attributes explicitly is not mandatory, except for *FromForm* (to distinguish it from data received from the request body, handled with the *FromBody* attribute) and *FromHeader*. However, I strongly recommend always using explicit binding to make your code clear and readable. You never have to think about where your parameters come from!

Validating Inputs

We're making progress! In the previous two sections, we've just seen how to invoke routes on ASP.NET Core 8 and how to bind parameters to functions managing the actions to be executed. Well, now we can move on to another critical point in API development: validating inputs. Why validate inputs? Well, after all, never trust your users! Your users can be dizzy or even ill-intentioned. Validating your inputs will allow you to check the following:

1. Ensure that the data passed to your API complies with your business rules. For example, suppose your users must register for a service by passing their email address. In that case, you'll need to check that their email address is valid, that it's an email address, so you want to ensure that your business rules are respected. Another example is when you receive an HTTP URL and want to ensure

that you only receive an HTTPS address and refuse HTTP URLs.

2. Consider that you're receiving data you will only use for display purposes, most likely on a web page. You'll have to be wary of what the user sends you. Remember, in Chapter 3, I talked about SQL injections, and we'll come back to this in Chapter 6, as I said, but you can also have *XSS* injections. A user could send you a malicious JavaScript script to display a message. Here you need to check whether the information the user passes contains HTML tags.

Let's look at how to manage this validation in ASP.NET Core minimal APIs. If you're familiar with *DataAnnotations* from ASP.NET Core MVC, Razor Pages, or Web API, this is not supported in the minimal APIs. Listing 4-13 shows a *Country* class on which two of its three properties, *Name* and *FlagUri* properties, contain validation rules, *Required* and *RegularExpression*.

Listing 4-13. The Country class that requires validation on Name and FlagUri properties

```
using System.ComponentModel.DataAnnotations;

namespace AspNetCore8MinimalApis.Models;

public class Country
{
    [Required]
    [RegularExpression("^[a-zA-Z0-9]+$")]
```

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

    public string Description { get; set; }

    [Required]
    [RegularExpression("^(https:\\/\\/.)[-a-zA-Z0-9@:%._\\+~#=]
    {2,256}\\.[a-z]{2,6}\\b([-a-zA-Z0-9@:%_\\+.~#?&//=]*)$")]
    public string FlagUri { get; set; }
}
```

As you can see, the *Name* property is required because of the *Required* annotation. It also must contain only alphanumeric characters, defined by the *RegularExpression* annotation. DataAnnotations, on regular expressions, can't detect the "not match"; it cannot decline the input when an HTML tag is detected. The annotation enables only matches. This is why I applied the alphanumeric match to this regular expression; matching alphanumeric means there are no HTML tags on the value of the *Name* property. Same reasoning on the *FlagUri* property, except the latter must match an HTTPS URL, once again using a regular expression.

As I said, this is not supported by ASP.NET Core 8 on minimal APIs. So we must find an alternative to this, and there is one! We will use a fascinating library called FluentValidation! FluentValidation lets you define built-in or custom validation rules and personalized error messages. To do this, run the following command from Package Manager Console in Visual Studio:

```
Install-Package FluentValidation.DependencyInjectionExtensions
```

To open the console, click "View" in the horizontal hat menu in Visual Studio, then "Other Windows," and finally "Package Manager Console," as shown in Figure 4-19.
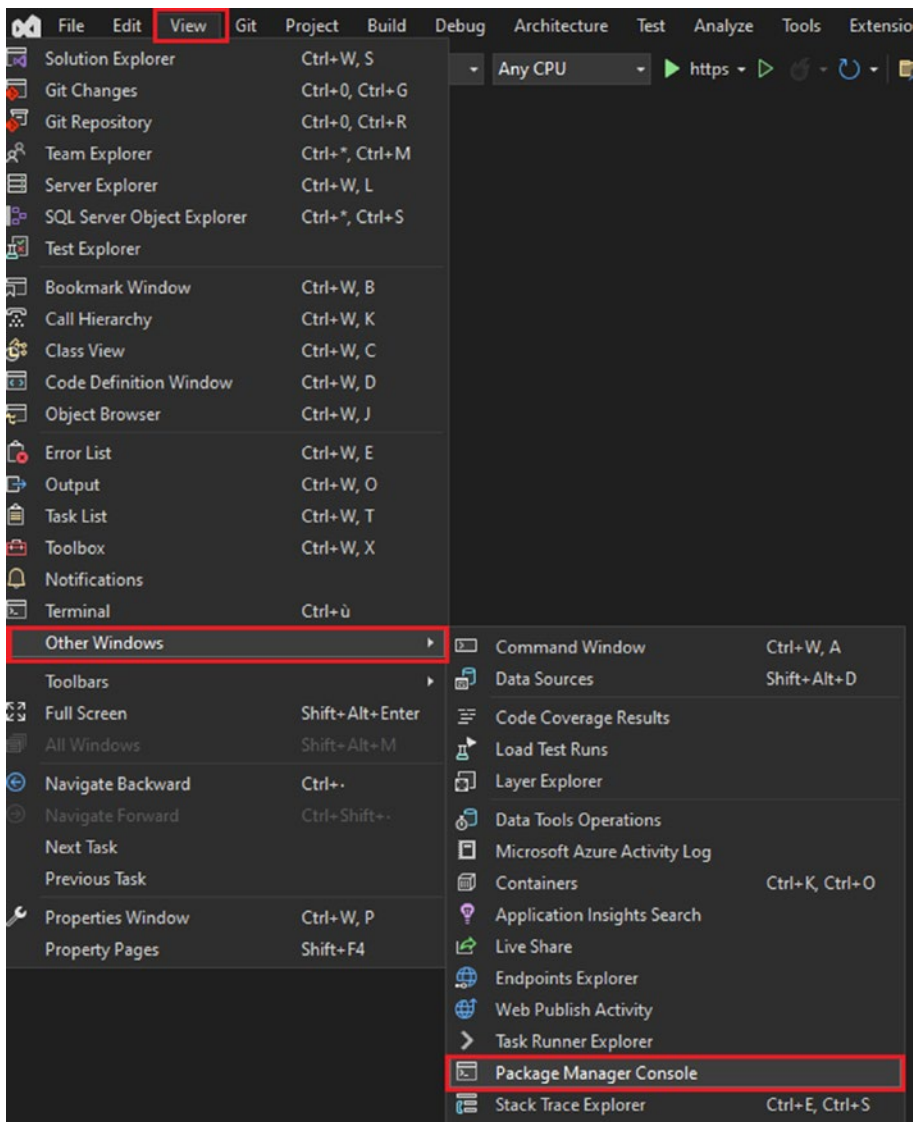
Figure 4-19. *Open Package Manager Console in Visual Studio 2022*

Then type the command and press the "Enter" key as shown in Figure 4-20.

Figure 4-20. *Executing the FluentValidation.DependencyInjection NuGet package installation in Package Manager Console*

Once installed, we can now write a validator with FluentValidation. Let's create a validation class named *CountryValidator* that inherits from *AbsctractValidator<T>* where *T* is the *Country* class. In the constructor, we will use the following methods:

- **RuleFor**: Defines the property where to apply the validation rule.

- **NotEmpty**: Defines a rule where the input value getting validated must not be empty.

- **WithMessage**: Applies an error message when the rule is not satisfied. It takes the *{PropertyName}* variable in the error message, which will be replaced with the property name tested against the validation rule. You can hard-code the property name in the message as well.

- **Custom**: Defines a custom rule, for example, a regular expression that matches an expression. It can trigger an error with the *AddFailure* method, which takes the property name and the error message as parameters. It is used, for example, when you want to raise an error when a match is found. This is precisely what was not possible with DataAnnotations.

- **Matches**: Defines a matching rule with a regular expression.

All these validation methods (except the *AddFailure* method) can be chained since they are all extension methods on the *IRuleBuilderOptions* interface.

Listing 4-14 shows the validation on the *Country* class where *Name* and *FlagUri* are required. *Name* must not match any HTML tag; else, it raises an error. And *FlagUri* must match an HTTPS URL.

Listing 4-14. FluentValidation validator applied on the Country class

```
using AspNetCore8MinimalApis.Models;
using FluentValidation;
using FluentValidation.Results;
using System.Text.RegularExpressions;

namespace AspNetCore8MinimalApis.Validators;

public class CountryValidator : AbstractValidator<Country>
{
    public CountryValidator()
    {
        RuleFor(x => x.Name)
        .NotEmpty()
```

```
    .WithMessage("{PropertyName} is required")
    .Custom((name, context) =>
    {
       Regex rg = new Regex("<.*?>"); // Matches HTML tags
       if (rg.Matches(name).Count > 0)
       {
          // Raises an error
          context.AddFailure(
             new ValidationFailure(
             "Name",
             "The parameter has invalid content"
             )
          );
    }});

    RuleFor(x => x.FlagUri)
    .NotEmpty()
    .WithMessage("{PropertyName} is required")
    .Matches("^(https:\\/\\/.)[-a-zA-Z0-9@:%._\\+~#=]
    {2,256}\\.[a-z]{2,6}\\b([-a-zA-Z0-9@:%_\\+.~#?&//=]*)$")
    .WithMessage("{PropertyName} must match an HTTPS URL");
  }
}
```

To make it up and running (being able to use the validator by dependency injection), we must register, with a single scan in the ASP.NET Core application assembly, any FluentValidation validator defined in this assembly. Listing 4-15 shows the registration by using the *AddValidatorsFromAssemblyContaining<Program>* method.

Listing 4-15. Registering all FluentValidation validators in the same assembly of the Program class

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddValidatorsFromAssemblyContaining<P
rogram>();
var app = builder.Build();
```

Using the *Program* class as a generic parameter will allow the discovery of any FluentValidation validator in the ASP.NET Core application since the *Program* class belongs to the ASP.NET Core application assembly. In other words, the *Program* allows discovering the assembly name used to scan any FluentValidation validator in this assembly. We can now write an endpoint that posts a *Country* object, the POST /countries endpoint, as shown in Listing 4-16.

Listing 4-16. /countries POST endpoint with validation of the Country object sent from the request body

```
app.MapPost("/countries", ([FromBody] Country country,
IValidator<Country> validator) => {
    var validationResult = validator.Validate(country);

    if (validationResult.IsValid)
    {
        //Do something
        return Results.Created();
    }
    return Results.ValidationProblem(validationResult.
    ToDictionary(), statusCode: (int)HttpStatusCode.
    BadRequest);
});
```

Registering all validators with the *AddValidatorsFromAssemblyContaining* method allows to pass in any minimal endpoint, by dependency injection, the *IValidator<T>* interface where *T* is the *Country* class. The latter will automatically instantiate the *CountryValidator* validator. The validation is easy. You must invoke the *Validate* method, which takes the *Country* object passed to the endpoint as a parameter. The result allows you to verify if the validation passed by using the *IsValid* property and return a successful HTTP status of a list of errors that needs to be cast to a *Dictionary* to the *ValidationProblem* method, which handles the JSON response to the client with a detailed validation error payload.

Note   The *ValidationProblem* method returns an HTTP 400 Bad Request, similar to the *ProblemDetails response payload,* which is more generic. *ValidationProblem* is not defined by any RFC but extends the *ProblemDetails* response by adding the list of the errors encountered during the input validation.


To illustrate the preceding example, in Figure 4-21, let me show you what the output gives when I try to add a malicious JavaScript *Alert* function within a script tag in the *Name* property and if I omit the HTTPS in the *FlagUri* property.
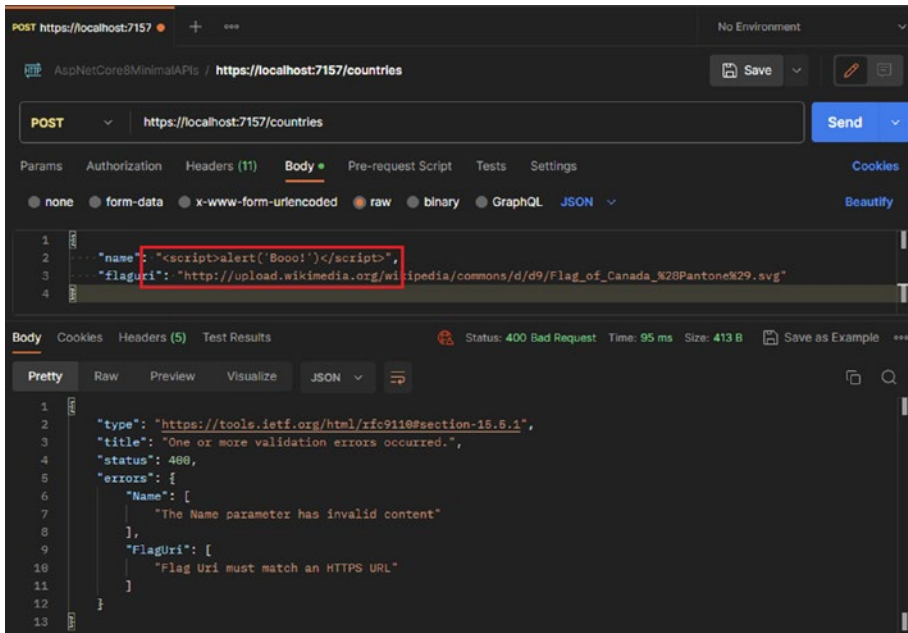
Figure 4-21. *Output with a ValidationProblem payload when validation fails on the Name and FlagUri properties passed to the POST /countries request*

As you can see, errors are pretty well detailed!

If the validation succeeds, you get instead a successful HTTP response, as shown in Figure 4-22.
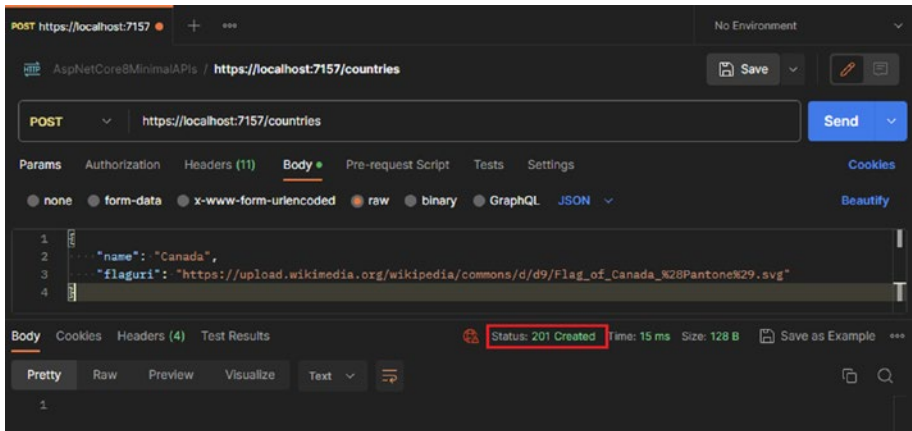
Figure 4-22. Successful validation on the Country JSON object passed to the POST /countries request

FluentValidation is a powerful library for performing any validation. However, I've used simple validation examples. If you'd like to see how powerful this library is, I invite you to learn more here: `https://docs.fluentvalidation.net/en/latest/`.

It's free!

The most important thing to remember here, and I hope this is the case, is that you should never trust a user. This can save you a lot of trouble! Validating all input will protect you from malicious attacks and is a good practice to implement on your APIs!

Object Mapping

Well done! We've seen how to validate our data! Now we can put them to use. In Chapter 3, I discussed the *Separation of Concerns (SoC)* principle and *abstraction*. Well, I will show you an example of how to apply this logic here in this section. The input parameters of your endpoints are specific to your web layer, your API to be exact, and must only be used in this layer.

So how will we transport the information received from your API endpoint to the repository enabling you to make database requests? We'll map input parameters, offering you my way of mapping your endpoint input to *Data Transfer Objects (DTOs)* or *domain objects*. These DTOs, or domain objects, are part of your application's domain and are defined in a different layer, as you'll recall from Chapter 3, a layer generally called Domain, which is used by all application layers. We're going to put into practice what we saw in Chapter 3 by doing the following:

1. Create a Domain layer in which we'll create a DTO.

2. Create an interface to **abstract** an object API input parameter/DTO mapping class and its implementation.

This interface describing the mapping to be performed and its implementation will be defined in the API layer. As I said earlier, the input parameters of your APIs are specific to this layer and must not be visible elsewhere than in this layer. DTOs, on the other hand, are accessible in your API layer.

The API layer, therefore, depends on the Domain layer, not the other way around.

Figure 4-23 shows the API and Domain layers, each containing elements that are specific to their respective responsibilities.
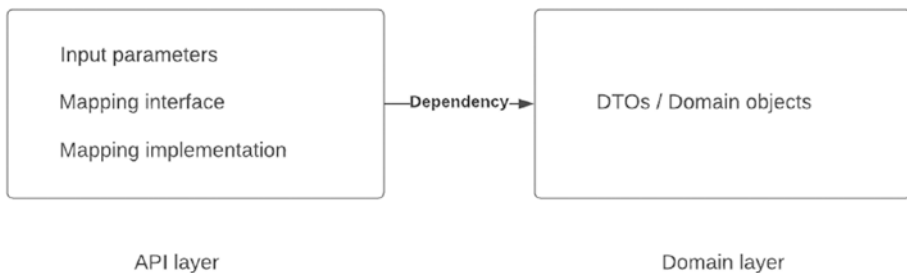
Figure 4-23. *API and Domain layers with their respective responsibilities*

Now let's create a class *CountryDto* (by first creating the Domain layer) and map it from our *Country* input parameter class in the API layer.

Listing 4-17 shows the definition of the *CountryDto* class.

Listing 4-17. The CountryDto class

```
namespace Domain.DTOs;

public class CountryDto
{
    public string Name { get; set; }
    public string Description { get; set; }
    public string FlagUri { get; set; }
}
```

The class is strictly identical to the *Country* class, which serves as an input parameter. I'm not duplicating any code here, but remember that just because the class signatures are identical doesn't mean you should create a single class for your input parameters and your domain objects because their responsibilities differ and may evolve differently. It's crucial to understand this.

As I think you already know how to create a project in Visual Studio 2022, I won't show you how it's done, but you can see how I've structured my Domain layer in Figure 4-24.
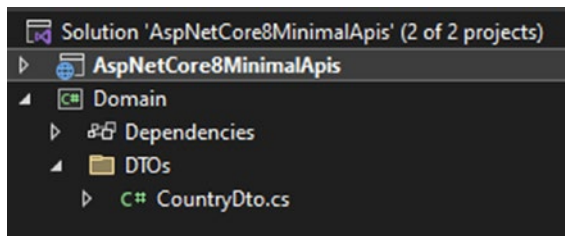
Figure 4-24. *API layer and Domain layer with their respective responsibilities*

As you can see, the Domain layer is structured to contain folders to put things in the right place. In this case, I created a folder called *DTOs* and put my *CountryDto* class within. Don't forget to reference your Domain layer. We are now ready to write our mapper class in the API layer!

Listing 4-18 shows the *ICountryMapper* interface.

Listing 4-18. The ICountryMapper interface

```
using AspNetCore8MinimalApis.Models;
using Domain.DTOs;

namespace AspNetCore8MinimalApis.Mapping.Interfaces;

public interface ICountryMapper
{
    public CountryDto? Map(Country country);
}
```

It's pretty straightforward. I defined a method *Map* that takes as a parameter a *Country* object and returns a *CountryDto*, which can be null for any reason. Now let's implement the mapper. Listing 4-19 shows the *CountryMapper* class that implements the *ICountryMapper* interface.

Listing 4-19. The CountryMapper class

```
using AspNetCore8MinimalApis.Mapping.Interfaces;
using AspNetCore8MinimalApis.Models;
using Domain.DTOs;

namespace AspNetCore8MinimalApis.Mapping;

public class CountryMapper : ICountryMapper
{
    public CountryDto? Map(Country country)
    {
```

```
        return country is not null ? new CountryDto
        {
            Name = country.Name,
            Description = country.Description,
            FlagUri = country.FlagUri,
        } : null;
    }
}
```

Once again, it's straightforward. Remember, here, the goal is not to learn how to map an object to another but to understand the importance of abstraction and SoC ☺. If you want to use a library that helps map your objects, since we abstracted the mapping operation, you can use any implementation you want. Suppose you don't want to use manual mapping like I did (which is the fastest way in terms of performance compared with any mapping library). In that case, you can keep the interface (no changes are needed) and change the implementation by using instead

- AutoMapper, which can be found here: https://automapper.org/

- Mapster, which can be found here: https://github.com/MapsterMapper/Mapster

Let's register the pair *ICountryMapper*/*CountryMapper* in the dependency injection system, and we can inject the mapper anywhere in the application. Listing 4-20 shows the registration as *Scoped* since we don't need to keep the mapper's instance (and properties) in memory until the application shuts down. Scoped will do the job perfectly.

Listing 4-20. Registration of the ICountryMapper interface and its
implementation CountryMapper

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddScoped<ICountryMapper, CountryMapper>();
var app = builder.Build();
```

Listing 4-21 shows the POST /countries endpoint updated with the
mapper injected by dependency.

Listing 4-21. The POST /countries endpoint update with the
ICountryMapper injected

```
app.MapPost("/countries", ([FromBody] Country country,
IValidator<Country> validator, ICountryMapper mapper) => {
    var validationResult = validator.Validate(country);

    if (validationResult.IsValid)
    {
        var countryDto = mapper.Map(country);

        //Do some work here
        return Results.Created();
    }
    return Results.ValidationProblem(validationResult.
    ToDictionary());
});
```

If we execute the code, it should work like a charm, as shown in
Figure 4-25.

```
app.MapPost("/countries", ([FromBody] Country country, IValidator<Country> validator, ICountryMapper mapper) => {
    var validationResult = validator.Validate(country);

    if (validationResult.IsValid)
    {
        var countryDto = mapper.Map(country);
                            • ⌖ countryDto    [Domain.DTOs.CountryDto]  {-}
        // do the thi    ⚲ Description   Q View ▾ "Maple leaf country"
        return Result    ⚲ FlagUri      Q View ▾ "https://upload.wikimedia.org/wikipedia/commons/d/d9/Flag_of_Canada_%28Pantone%29.svg"
    }                    ⚲ Name         Q View ▾ "Canada"

    return Results.ValidationProblem(validationResult.ToDictionary());
});
```

Figure 4-25. *The POST /countries endpoint execution with the*
ICountryMapper injected

Note   Another benefit of abstraction is the ease of unit testing a
piece of code. I will return to this in the last chapter of this book.

Managing CRUD Operations and HTTP Statuses

With what we have discussed so far, we can code any endpoint. This
is commonly referred to as *Create, Retrieve, Update, Delete (CRUD)*
operations. As we saw in Chapter 1, there are different types of verbs for
manipulating an entity, and we're going to take a look at them:

- **POST**: Creates an entity known as *Create (C)* from the
  CRUD acronym

- **GET**: Retrieves an entity or a collection of entities,
  known as *Retrieve (R)* from the CRUD acronym

- **PUT**: Replaces an entity (or creates an entity if it does
  not exists), known as *Update (U)* in the CRUD acronym

- **PATCH**: Updates part of an entity, known as the *Update
  (U)* element in the CRUD acronym

- **DELETE**: Allows you to delete an entity, known as
  Delete (D) in the CRUD acronym

> **Note**   To keep this section simple with the basics of CRUD
> operations, I will use the *Country* class as the input and output
> parameters in the API and the *CountryDto* as an input and output
> in the Domain layer. There is no need to duplicate them for input/
> output flow.

Managing CRUD operations also implies managing the HTTP status in
the response, and I will show how in the following subsection.

Handling HTTP Statuses

Another aspect of managing CRUD operations is to handle HTTP response
statuses correctly. If you remember Chapter 1, I introduced them to you,
and I want to show you how to handle them with the static *Results* class.
The *Results* class exposes many methods to allow you to use the most
common HTTP statuses.

I won't go into detail for each one, but I will explain why I use them
in every further example throughout this book. Some of them also have
overloads that I won't go into detail. Table 4-5 shows them.

Table 4-5. *Results class methods with their associated HTTP status*

Method	HTTP status code produced
Accepted	202
AcceptedAtRoute	202
BadRequest	400
Bytes	200, 206, 416
Challenge	401
Conflict	409

(*continued*)

Table 4-5. (*continued*)

Method	HTTP status code produced
Content	200
Created	201
CreatedAtRoute	201
File	200, 206, 416
Json	200
Forbid	403
Redirect	301, 302, 307, 308
RedirectToRoute	301, 302, 307, 308
SignIn	200
SignOut	200
Stream	200, 206, 416
Text	200
Unauthorized	401
UnproccessableEntity	422
ValidationProblem	400 (BadRequest + ValidationProblemDetail payload)
StatusCode	Any HTTP status since it takes as a parameter the integer value of any status
Empty	200

As you can see, Microsoft helps you a lot with predefined methods. If you miss any status to handle your request, you can still use the *StatusCode* method, which takes any HTTP status code as a parameter. If you can't wait for some explanations from me regarding some of them I will cover in this book, you can learn from the Microsoft documentation here: https:// learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http. results?view=aspnetcore-8.0.

Creating the Services to Handle CRUD Operations

We'll create a service that handles *CountryDto* objects to implement the preceding CRUD operations. We need to implement a repository between the API and Infrastructure layers. Once again mindful of the principles of abstraction and *Separation of Concern (SoC)*, I'm going to create an *ICountryService* service interface in the Domain layer, along with its *CountryService* implementation defined in the *Business logic layer (BLL)*, as this is where the business logic is implemented. I won't detail the implementation of the *CountryService* class here, as that's not important here, but rather how the minimal endpoints managing the *Country* entity will consume this service. Figure 4-26 shows the API layer, Domain layer, and BLL with their respective responsibilities.
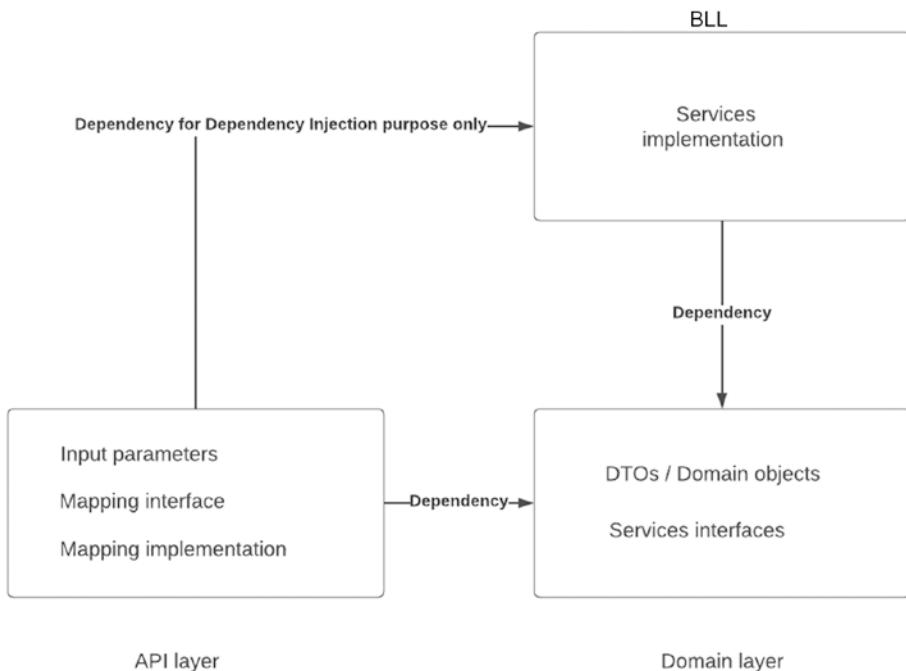
Figure 4-26. *API layer, Domain layer, and BLL with their respective responsibilities*

Listing 4-22 shows the *ICountryService* interface signature.

Listing 4-22. The ICountryService interface

```
using Domain.DTOs;

namespace Domain.Services;

public interface ICountryService
{
    CountryDto Retrieve(int id);
    List<CountryDto> GetAll();
    int CreateOrUpdate(CountryDto country);
    bool UpdateDescription(int id, string description);
    bool Delete(int id);
}
```

Don't forget to register the service as *Scoped* in the dependency injection system as follows:

```
builder.Services.AddScoped<ICountryService, CountryService>();
```

To make CRUD operations on the *CountryDto*, I added an ID to this class, which is nullable since the country may have or not have an ID (null before its creation, filled after its creation) as shown on Listing 4-23.

Listing 4-23. The CountryDto class updated with a nullable ID

```
namespace Domain.DTOs;

public class CountryDto
{
    public int? Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public string FlagUri { get; set; }
}
```

I did the same on the *Country* input parameter in the API. This is required to identify a country with a unique ID instead of using its name. As you can see, we can find all CRUD operations on the service needed to implement all CRUD endpoints in the API layer. Some explanations are needed to clarify how CRUD operations will operate:

- The *Retrieve* method takes a country ID as a parameter and must return CountryDto.

- The *GetAll* method takes no parameter and returns a collection of *CountryDto.*

- The *CreateOrUpdate* method takes a *CountryDto* as a parameter. If the country's ID is null, that means the country is not identified and should be created, while if the ID is not null, the country should be updated. It's not the absolute truth since the ID can be wrong and may identify a nonexisting country, which could be null, but the country already exists. Let's keep the example simple and assume that a country is correctly identified when an ID is provided and not defined when the ID is null.

- The *UpdateDescription* method takes a parameter of the country ID and the country description to get updated. It acts as a partial update and returns a Boolean that indicates whether the update has been performed.

- The *Delete* method takes the country ID as a parameter and returns a Boolean that indicates whether the deletion has been successfully performed.

Now it's time to write our CRUD endpoints.

Creating the Endpoints to Handle CRUD Operations

I have updated the validator and mapper classes to handle new validation and mapping rules; I will show you their implementation after showing you the endpoint implementations. I also injected on each the *ICountryService* interface. Listing 4-24 shows the CRUD endpoints.

Listing 4-24. CRUD endpoint implementations on the CountryDto domain object through the Country input/output parameter

```
// Create
app.MapPost("/countries", (
          [FromBody] Country country,
          IValidator<Country> validator,
          ICountryMapper mapper,
          ICountryService countryService) => {
    var validationResult = validator.Validate(country);

    if (validationResult.IsValid)
    {
        var countryDto = mapper.Map(country);
        return Results.CreatedAtRoute(
                "countryById",
                new {
                        Id = countryService.CreateOrUpdate(
                                countryDto
                            )
                    }
        );
    }
    return Results.ValidationProblem(
            validationResult.ToDictionary()
```

```
            );
});

// Retrieve
app.MapGet("/countries/{id}", (
            int id, ICountryMapper mapper,
            ICountryService countryService) => {
    var country = countryService.Retrieve(id);

    if (country is null)
        return Results.NotFound();

    return Results.Ok(mapper.Map(country));
}).WithName("countryById");

// Retrieve
app.MapGet("/countries", (
            ICountryMapper mapper,
            ICountryService countryService) => {
    var countries = countryService.GetAll();
    return Results.Ok(mapper.Map(countries));
});

// Update
app.MapPut("/countries", (
            [FromBody] Country country,
            IValidator<Country> validator,
            ICountryMapper mapper,
            ICountryService countryService) => {
    var validationResult = validator.Validate(country);

    if (validationResult.IsValid)
    {
        if (country.Id is null)
```

```
            return Results.CreatedAtRoute(
                    "countryById",
                    new {
                            Id =                 countryService.
                                                 CreateOrUpdate(
                                                 mapper.
                                                 Map(country)
)
    });
        return Results.NoContent();
    }
    return Results.ValidationProblem(
        validationResult.ToDictionary()
    );
});

// Update
app.MapPatch("/countries/{id}", (
            int id,
            [FromBody] CountryPatch countryPatch,
            IValidator<CountryPatch> validator,
            ICountryMapper mapper,
            ICountryService countryService) => {
    var validationResult = validator.Validate(countryPatch);

    if (validationResult.IsValid)
    {
        if (countryService.UpdateDescription(
            id,
            countryPatch.Description
        ))
            return Results.NoContent();
```

```
        return Results.NotFound();
    }
    return Results.ValidationProblem(
            validationResult.ToDictionary()
    );
});

// Delete
app.MapDelete("/countries/{id}", (
            int id,
            ICountryService countryService) => {
    if (countryService.Delete(id))
        return Results.NoContent();

    return Results.NotFound();
});
```

Let's make some explanation!

MapPost("/countries", () => {})

Since it's an HTTP POST request, the country passed in the request should be created since its ID is null here. After validating the input and mapping it into a *CountryDto* object, I'm invoking the *CreateOrUpdate* service method since it handles creation and update operations. The expected response should be Created (201), produced by the *CreatedAtRoute* method, or Bad Request (400), produced by the *ValidationProblem* method. The *CreatedAtRoute* method takes as a parameter a route name, *countryById*, which indicates at what route you can retrieve the created resource, and the ID of the created country used to populate the *countryById* route parameter. The *countryById* route name must exist in your API, and we will jump into it right after showing the expected output when the creation succeeds in Figure 4-27.
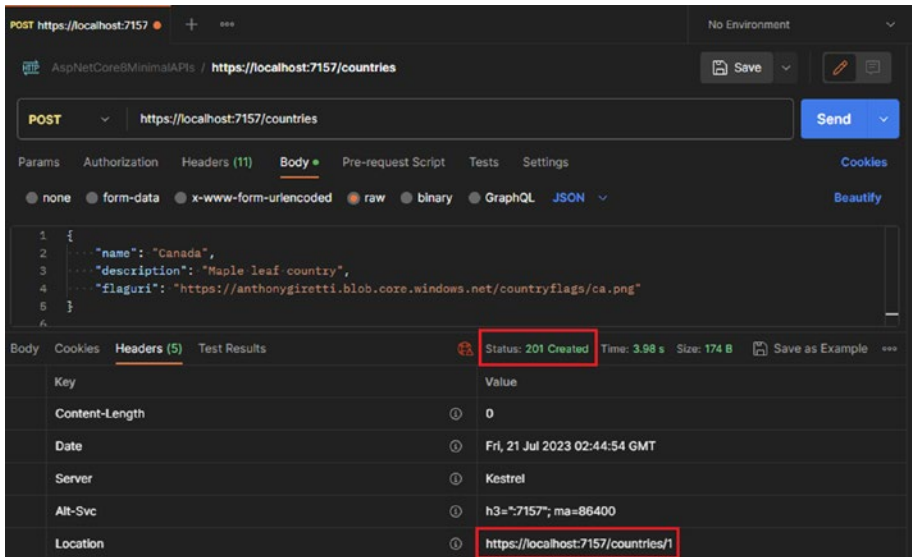
Figure 4-27. *Expected response when creating a country using the POST /countries endpoint*

As you can see, the newly created route is sent to the client over the **Location** header.

MapGet("/countries/{id}", () => {})

This is a GET endpoint, so two cases are possible: the resource is found, where the data serialized in JSON with an OK (200) response is returned, or Not Found (404). The result methods implied are, respectively, *Ok* and *NotFound*. They are pretty well named, so you can't get confused while selecting the correct one to handle the response. You may notice I attached the "countryById" route name to this endpoint to tell ASP.NET Core how to build the location URL of the newly created country. To do this, I applied on the endpoint the *WithName* method. The latter allows you to give any endpoint the name you want. It's a unique ID, useful to identify a route as we did with the country creation. Figure 4-28 shows the expected response when the request is successful.
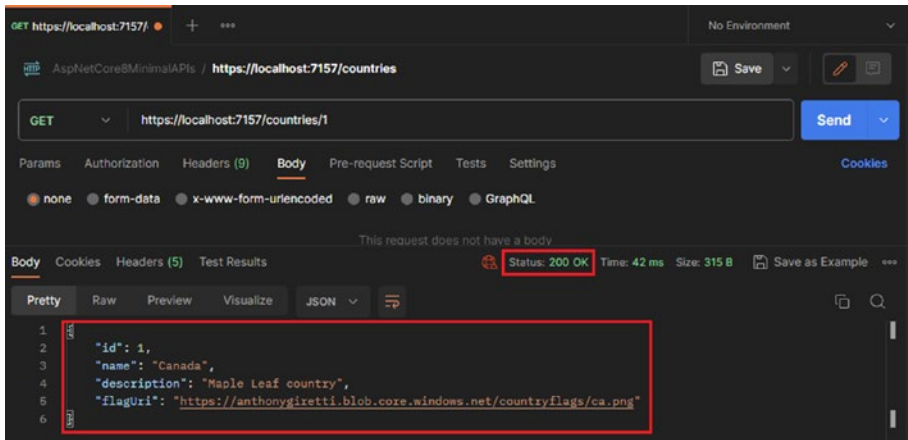
Figure 4-28. *Expected response when retrieving a country using the /countries/[id} GET endpoint*

MapDelete("/countries/{id}", () => {})

The delete endpoint is the simplest of all. It takes the country's ID you wish to delete as a route parameter. If the country exists, the response will be No Content (204) or Not Found (404) otherwise. I don't need to explain which methods I used, as their names speak for themselves. Figure 4-29 shows the expected result when the delete operation is successful.
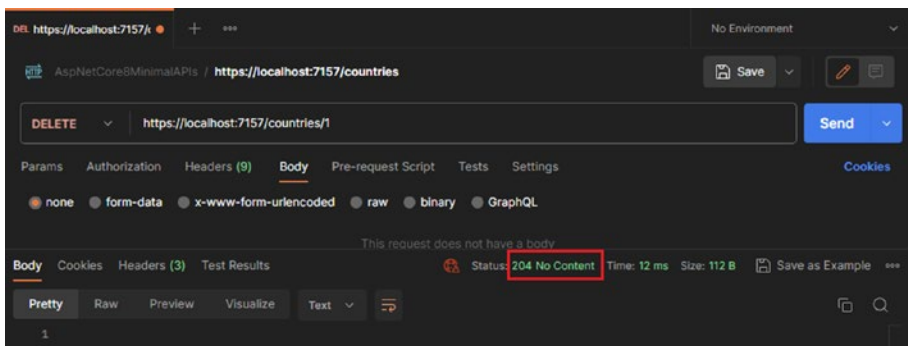
Figure 4-29. *Expected response when deleting a country using the /countries/[id} DELETE endpoint*

MapPut("/countries/{id}", () => {})

The PUT endpoint is identical to the POST endpoint since it creates a
country when it does not exist or updates it when it does. Validation
remains the same and may return a Bad Request (400) when validation
doesn't meet the requirements. The only difference resides in the fact if
the country has an ID, it's supposed to be updated and NoContent (204)
is returned when the operation succeeds, but when the country does not
have an ID set, Created (201) is returned. Figure 4-30 shows a successful
update. For a successful creation, you can refer to Figure 4-27.
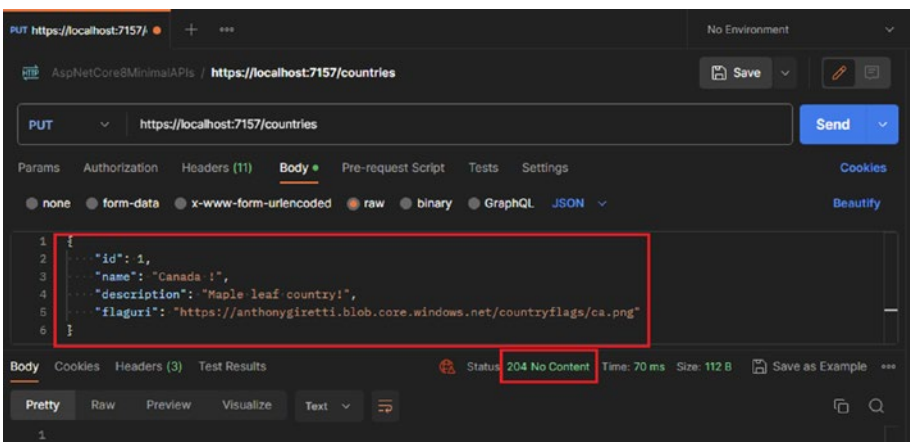
Figure 4-30. *Expected response when updating a country using the*
PUT /countries endpoint

You may have noticed that the route doesn't contain any ID. The ID is in
the request body. No RFC says that the route of a PUT request should contain
any ID of the target resource (a country here). It makes sense since PUT is
idempotent; its route doesn't change as long as the payload in the request
body doesn't change. It will always be a country resource to be updated or
created, depending on whether the ID is null or filled, which will indicate
what to do: create or update. When an update succeeds, RFC allows you to
return an OK (200) response. I prefer to return No Content. It's up to you.

MapPatch("/countries/{id}", () => {})

The PATCH method has been implemented as uncomplicated as
possible since we want to update only the description of a country. The
flow is straightforward: A validation is performed on the *CountryPatch*
input parameter, which differs from the *Country* input parameter
because we only want to update the description. Because of that, I
created a validator that applies to the *CountryPatch* input parameter
(*IValidator<CountryPatch>).* If the validation meets the requirements
and the update succeeds, No Content (204) response is returned; else,
a Not Found (404) is produced since PATCH does not create a resource
when it does not exist. So an ID in the route is necessary for it to identify
the resource. If the validation fails, a Bad Request (404) is returned with a
ProblemDetails payload in the response.

Note   This vision of implementing a PATCH request is the most
common way to perform a partial update and is allowed by RFCs.
However, it's not the absolute truth since there is a better way to
perform PATCH requests according to **RFCs 5789** and **6902**.

If you want to learn more about it, you can go on the Microsoft
documentation, which explains how to perform PATCH properly (but it's
more complicated and less often used). In the meantime, this Microsoft
documentation provides the link to the mentioned RFCs: `https://learn.`
`microsoft.com/en-us/aspnet/core/web-api/jsonpatch?view=aspne`
`tcore-8.0`.

As promised, Listings 4-25 and 4-26, respectively, show the updated
CountryMapper class and the *CountryPatchValidator* class I used to make
CRUD operations earlier.

Listing 4-25. The CountryMapper class

```csharp
using AspNetCore8MinimalApis.Mapping.Interfaces;
using AspNetCore8MinimalApis.Models;
using Domain.DTOs;

namespace AspNetCore8MinimalApis.Mapping;

public class CountryMapper : ICountryMapper
{
    public CountryDto? Map(Country country)
    {
        return country != null ? new CountryDto
        {
            Id = country.Id,
            Name = country.Name,
            Description = country.Description,
            FlagUri = country.FlagUri,
        } : null;
    }

    public Country? Map(CountryDto country)
    {
        return country != null ? new Country
        {
            Id = country.Id,
            Name = country.Name,
            Description = country.Description,
            FlagUri = country.FlagUri,
        } : null;
    }
```

```
    public List<Country> Map(List<CountryDto> countries)
    {
        return countries.Select(Map).ToList();
    }
}
```

Listing 4-26. The CountryPatchValidator class

```
using AspNetCore8MinimalApis.Models;
using FluentValidation;
using FluentValidation.Results;
using System.Text.RegularExpressions;

namespace AspNetCore8MinimalApis.Validators;

public class CountryPatchValidator : AbstractValidator<Cou
ntryPatch>
{
    public CountryPatchValidator()
    {
        RuleFor(x => x.Description)
        .NotEmpty()
        .WithMessage("{ParameterName} cannot be empty")
        .Custom((name, context) =>
        {
            Regex rg = new Regex("<.*?>"); // Matches HTML tags
            if (rg.Matches(name).Count > 0)
            {
                // Raises an error
                context.AddFailure(new ValidationFailure(
                "Description",
                "The description has invalid content")
                );
```

```
            }
        });
    }
}
```

Note   For ease of reading, I've duplicated the regular expression
to test whether any string contains HTML tags. If you recall, this
regular expression is also used in the *CountryValidator* class we saw
previously.

To finish, I just wanted to mention that I showed the happy path
in the CRUD operations earlier. In real life, error management is a bit
more complex, and we will see in the next chapter how to handle error
management when something is going wrong in the application.

Downloading and Uploading Files

As a developer, you will almost certainly have to manage files. Whether it's
downloading or uploading, you'll be doing this regularly. Although this
is part of CRUD operations in general, I've decided to make it a separate
section in this chapter, as there's much to say about it. Let's start with file
downloading.

Downloading Files

Downloading a file is relatively simple—you need three things:

1. Know the MIME type of your file.

2. Transform the contents of your file into a byte array.

3. Give your file a name.

And ...use the *File* method of the *Results* class.

Before we do that, let's take a quick look at the MIME type of a file, which I introduced in Chapter 1. There's a whole panoply of MIME types, and ...there are many of them. If you're interested, consult the complete list defined by the *Internet Assigned Numbers Authority (IANA)* at this address: www.iana.org/assignments/media-types/media-types.xhtml. For your information, *IANA* is responsible for coordinating the rules that keep the Internet at a standard of use acceptable to all.

After redesigning the *ICountryService* interface by adding a method named *GetFile* to support file download, we can look at how to write a download endpoint in the API. Listing 4-27 shows the updated *ICountryService* interface. Once again, the implementation of the *CountryService* class is not relevant to be shown since data are mocked. But you will see the implementation in the source code provided by Apress on a GitHub repository.

Listing 4-27. The ICountryService interface updated with the GetFile method

```
using Domain.DTOs;

namespace Domain.Services;

public interface ICountryService
{
    CountryDto Retrieve(int id);
    List<CountryDto> GetAll();
    int CreateOrUpdate(CountryDto country);
    bool UpdateDescription(int id, string description);
    bool Delete(int id);
    (
        byte[] fileContent,
        string mimeType,
```

```
    string filename
  ) GetFile();
}
```

As you can see, I chose to return a tuple, which contains three pieces of information, as I mentioned before:

1. The file content as an array of bytes

2. The file MIME Type

3. The file name

Then I designed the GET "/countries/download" endpoint as shown in Listing 4-28.

Listing 4-28. The GET countries/download endpoint

```
app.MapGet("/countries/download", (
          ICountryService countryService) => {

  (
      byte[] fileContent,
      string mimeType,
      string fileName) = countryService.GetFile();

   if (fileContent is null || mimeType is null)
      return Results.NotFound();

   return Results.File(fileContent, mimeType, fileName);
});
```

I grab from *ICountryService* all information needed by the *File* method, as mentioned previously, and we test it with Postman; it should give what (the debug step) is shown in Figure 4-31.
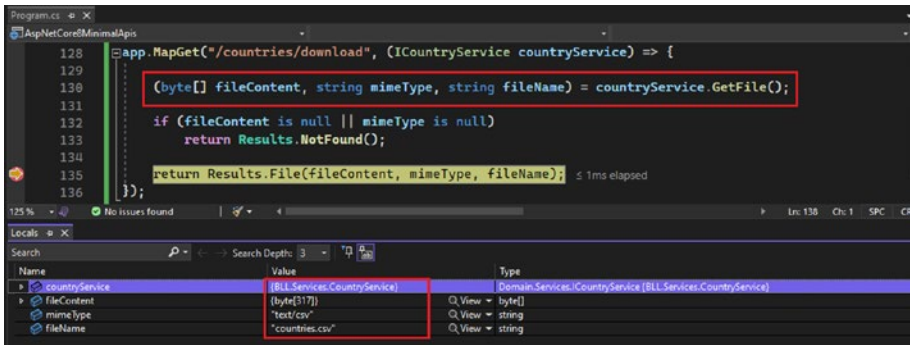
Figure 4-31. *Invoking the GET countries/download endpoint*

You can notice the file name is *countries.csv*. Since it's a CSV file, the MIME type is *text/csv*.

Figure 4-32 shows the result in Postman.
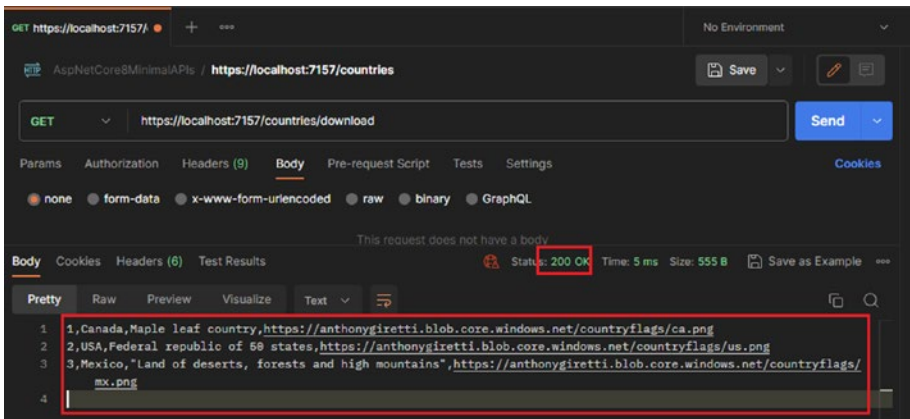
Figure 4-32. *The expected response when downloading the countries. csv file from the GET countries/download endpoint*

Postman, when it can interpret file content, will display it, but don't worry—if you try to download it from a browser, it will get done correctly, as shown in Figure 4-33.
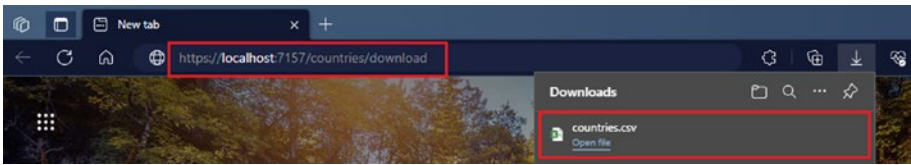
Figure 4-33. *Downloading the countries.csv file from a browser*

Finally, You can see what headers have been sent to the client, such as *Content-Length, Content-Type*, or *Content-Disposition,* as shown in Figure 4-34.
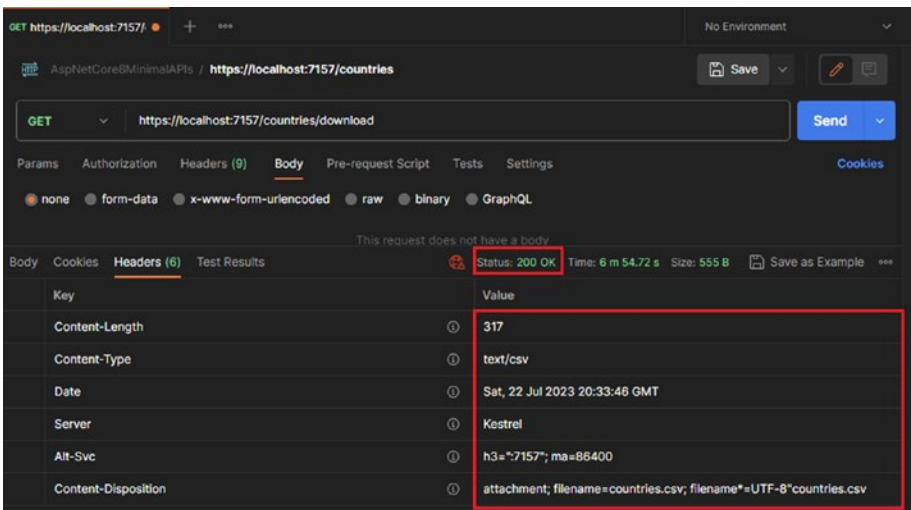
Figure 4-34. *Headers sent from the browser while downloading the countries.csv file*

Uploading Files

Let's move on to the reverse process. Upload a file to the server. This operation is a little trickier; we must perform a few validations on the file to ensure its integrity. I'll show you how to validate an uploaded file. Then, it's essential to know that file uploads work differently if

1. You're uploading a single file or many files without a payload.

2. You're sending a single file or many files with metadata (a payload in the request body).

Uploading a Single File or Many Files Without Any Payload

Let's start by establishing the possibility of uploading a file to a minimal API with ASP.NET Core 8. I will name the endpoint "/countries/upload" associated with the POST verb and use the *IFormFile* interface to accept an uploaded file. The posted file will be bound in *IFormFile* properties. Listing 4-29 shows the POST /countries/upload endpoint.

Listing 4-29. The POST countries/upload endpoint

```
app.MapPost("/countries/upload", (IFormFile file) =>
{
    return Results.Created();
});
```

I chose not to add the *[FromForm]* attribute since you can't get confused about where the files come from since it always comes from the form-data request body.

Figure 4-35 shows the *countries.csv* file getting uploaded via Postman.
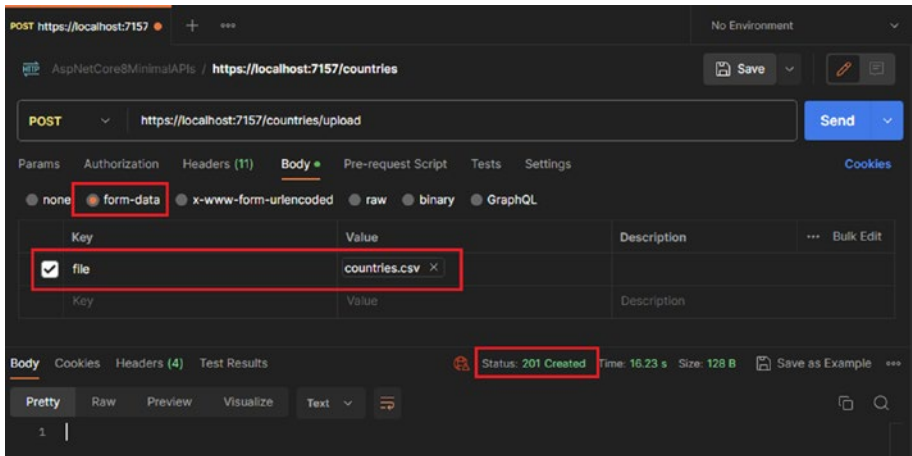
Figure 4-35. *Upload the countries.csv file to the POST /countries/ upload endpoint*

You may notice that a file must be sent over the form data and given a name identical to the input file parameter in the API; it's case insensitive. Here I chose the name *file.* This is mandatory for parameter binding purposes. If we take a look at the headers sent by Postman, we'll see two crucial headers: *Content-Type*, which contains the *multipart/form-data* value that is mandatory to send files via HTTP, and *Content-Length*, which tells the server the size of the uploaded file. Figure 4-36 shows the headers sent to the server (I voluntarily chose not to show you all headers since there is a bunch, but only the ones involved in the file upload).
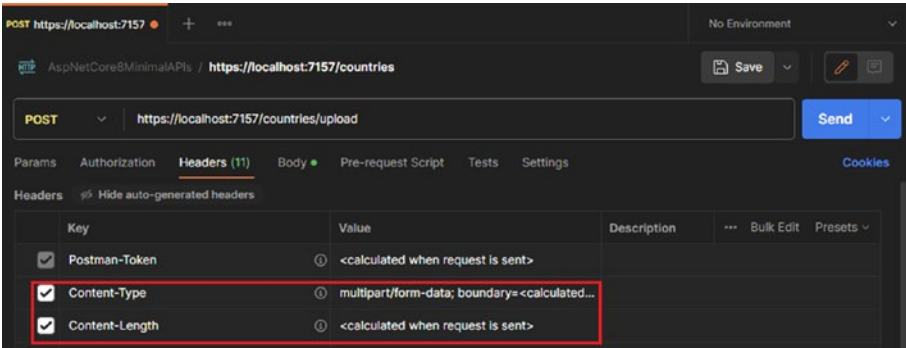
Figure 4-36. *Headers sent to the server when uploading the countries.csv file to the POST /countries/upload endpoint*

Figure 4-37 shows the *IFormFile* content while debugging the endpoint.
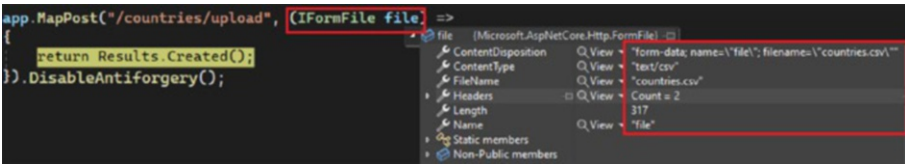
Figure 4-37. *IFormFile content when uploading the countries.csv file to the POST /countries/upload endpoint*

Let's move on to uploading several files. Yes, it's supported by ASP. NET Core 8 minimal APIs, and here's how: instead of using the *IFormFile* interface as an input parameter, you have to use the *IFormFileCollection* interface instead, as shown in Listing 4-30. I named the route /countries/ uploadmany.

Listing 4-30. The POST countries/uploadmany endpoint

```
app.MapPost("/countries/uploadmany", (IFormFileCollection
files) =>
{
    return Results.Created();
});
```

To make it work as in the previous example, you will have to name all your form-data keys, in Postman, the same as shown in Figure 4-38.
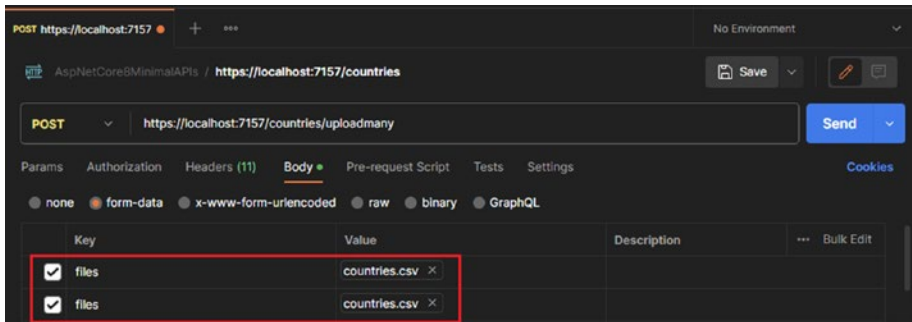
Figure 4-38. *Upload several files to the POST /countries/ uploadmany endpoint*

The headers sent to the server are the same. Nothing changes except their value. For example, the header *Content-Length* value will be higher than the previous one since we send two files instead of one. Figure 4-39 shows the endpoint while it receives the files.
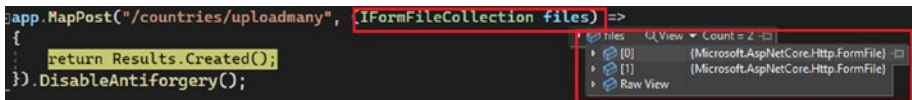
Figure 4-39. *IFormFileCollection content when uploading two files to the POST /countries/uploadmany endpoint*

Manipulating uploaded files with *IFormFileCollection* remains the same as *IFormFile*. You have to loop on each file (each file in the collection implements the *IFormFile* interface) and perform any action on it. The logic there is up to you. You can send them to a service or save them on a file disk. How you can upload files with ASP.NET Core 8 minimal APIs only matters here.

Let's see how we can handle a single file or several files sent to the server with a payload.

Note   I used the POST verb here, but the PUT and PATCH verbs also support file upload.

Uploading a Single File or Many Files with a Payload

Believe it or not, I didn't immediately understand how I was supposed to upload files and metadata simultaneously. And yet it's simple; I hadn't thought of it.

At first, I thought, *Well, I'll send my metadata in JSON in the request body and then my files in form data*. But it's simply impossible! As soon as you choose one of the two data transport methods, the other can't work. If you send data via the form-data transport, you can no longer send JSON data in a request body. The request body will be ignored. The *Content-Type* header is set to *multipart-form* data, not *application/json*. You must therefore send your payload via form data. Listing 4-31 shows the two previous endpoints updated with an additional input parameter named *CountryMetaData* to which I've associated the *[FromForm]* attribute, the latter being mandatory, as I announced earlier in this chapter.

Listing 4-31. The POST /countries/upload and /countries/ uploadmany endpoints updated with metadata coming from the request body

```
app.MapPost("/countries/uploadwithmetadata", (
[FromForm] CountryMetaData countryMetaData, IFormFile file) =>
{
    return Results.Created();
}).DisableAntiForgery();

app.MapPost("/countries/uploadmanywithmetadata", (
[FromForm] CountryMetaData
countryMetaData,    IFormFileCollection files) =>
{
    return Results.Created();
}).DisableAntiForgery();
```
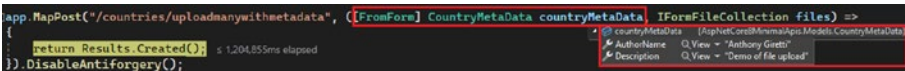
If I take the endpoint that uploads several files (both endpoints have the same behavior), the Postman request will look like that shown in Figure 4-40.
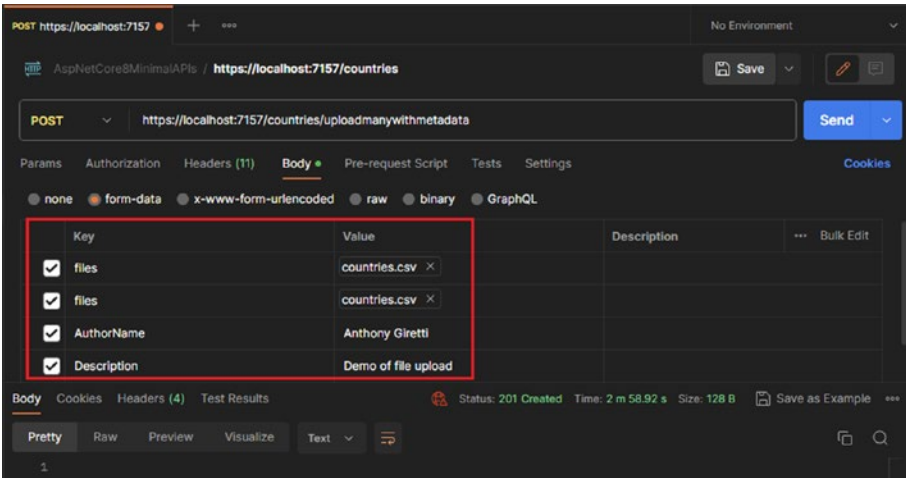
Figure 4-40. *Upload files and metadata simultaneously*

If we take a look server-side in the API endpoint, we should see the metadata property bound as shown in Figure 4-41.

Figure 4-41. *IFormFileCollection content when uploading several files with their metadata properly bound*
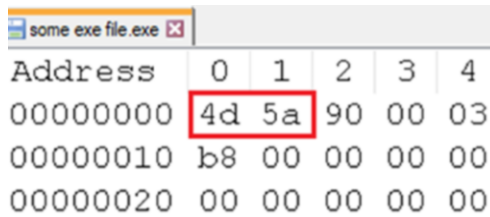
Validating an Uploaded File

Again, for security reasons, you'll need to check that the file you've uploaded poses no threat.

I'll show you what to do here. The first thing to remember here is to validate the following:

1. The file name should contain only **alphanumeric** characters, possibly with **hyphens** or **underscores**. Having special characters like slashes could induce unwanted behavior when you store your files. No file should contain a slash since slashes are used for directories.

2. Regarding file extensions, for a CSV file, we expect the **.csv** extension.

3. Regarding file MIME types, for a CSV file, expect the **text/csv** MIME type.

4. The file signature is the hexadecimal characters at the beginning of the file, and they characterize the file. For example, an **exe** file will always have the following hexadecimal character sequence at its beginning: `4D 5A` or `5A 4D`. We will test this sequence against the *countries.csv*. Since a CSV does not have a particular sequence because it's a plain text file, we must exclude some dangerous files such as **.ex**e (executable). Why are we doing that? Some hackers send files with a correct file extension, but it's not the expected file since **the extension can be renamed**. For your information, this validation is commonly named *Magic bytes* validation. Figure 4-42 shows the first two bytes in their hexadecimal representation of an executable file.

5. For file contents, I won't show you an example here, as we saw how to validate a string with a regular expression earlier in this chapter.

Figure 4-42. Executable file signature

Let's create a validator that will only authorize CSV files. If we keep the input parameters as is (refer to Listing 4-31), the validation will be applied to the *IFormFile* input parameter because the validator signature would be *IValidator<IFormFile>*. Since the latter may apply to any uploaded file, we'll face a problem if we want to validate other file types on another endpoint, for example. The best solution is to encapsulate the file to be uploaded and its metadata in a specific class where the validation will be performed exclusively for this one. Consider the *CountryFileUpload* class as the new input parameter for our endpoint that allows a single file to get uploaded (refer to Listing 4-32).

Listing 4-32. The CountryFileUpload class that encapsulates IFormFile and its metadata

```
public class CountryFileUpload
{
    public IFormFile File { get; set; }

    public string AuthorName { get; set; }
    public string Description { get; set; }
}
```

You may notice I put the file and its metadata at the same level to lower the complexity of the class.

Now we can write a specific validation on the *CountryFileUpload* class where we expect only a CSV file. We'll also validate the content of the metadata, *AuthorName* and *Description* properties. Listing 4-33 shows the *CountryFileUploadValidator* class.

Listing 4-33. The CountryFileUploadValidator class

```
using AspNetCore8MinimalApis.Models;
using FluentValidation;
using FluentValidation.Results;
using System.Text.RegularExpressions;

namespace AspNetCore8MinimalApis.Validators;

public class CountryFileUploadValidator : AbstractValidator<Cou
ntryFileUpload>
{
    public CountryFileUploadValidator()
    {
        RuleFor(x => x.File).Must((file, context) =>
        {
            return file.File.ContentType == "text/csv";
        }).WithMessage("ContentType is not valid");

        RuleFor(x => x.File).Must((file, context) =>
        {
            return file.File.FileName.EndsWith(".csv");
        }).WithMessage("The file extension is not valid");

        RuleFor(x => x.File.FileName).Matches(^[A-
        Za-z0-9_\\-.]*$").WithMessage("The file name is not
        valid");
```

```
RuleFor(x => x).Must((file, context) =>
{
    // string representation of hexadecimal signature of an
    execute file
    var exeSignatures = new List<string> {
                        "4D-5A",
                        "5A 4D"
                    };
        BinaryReader binary = new BinaryReader(file.File.
        OpenReadStream());
        byte[] bytes = binary.ReadBytes(2); // reading
        first two bytes
        string fileSequenceHex = BitConverter.
        ToString(bytes);
        foreach (var exeSignature in exeSignatures)
            if (exeSignature.Equals(
                    fileSequenceHex,
                    StringComparison.OrdinalIgnoreCase
            )
                return false;
        return true;
    }).WithName("FileContent")
      .WithMessage("The file content is not valid");

    RuleFor(x => x.AuthorName)
    .NotEmpty()
    .WithMessage("{PropertyName} is required")
    .Custom((authorName, context) =>
    {
        Regex rg = new Regex("<.*?>"); // Matches HTML tags
        if (rg.Matches(authorName).Count > 0)
        {
```

```
        // Raises an error
        context.AddFailure(
            new ValidationFailure(
                "AuthorName",
                "The AuthorName parameter has invalid
                content"));
    }
});

RuleFor(x => x.Description)
.NotEmpty()
.WithMessage("{PropertyName} is required")
.Custom((name, context) =>
{
    Regex rg = new Regex("<.*?>"); // Matches HTML tags
    if (rg.Matches(name).Count > 0)
    {
        // Raises an error
        context.AddFailure(new
            ValidationFailure(
                "Name",
                "The AuthorName parameter has invalid
                content"));
    }
});
    }
}
```

Since most parts of the validation are straightforward, I will explain the *Magic bytes* validation more. As mentioned previously, an executable file (**.exe**) signature may start with two different sequences. So I stored in a list their hexadecimal representation as a string. Each hexadecimal sequence

has two numbers representing the first two bytes, separated by a hyphen because the string representation of the hexadecimal numbers contains a hyphen between numbers. After opening the file as a *stream* with the *OpenReadStream* method, then I transform the *stream* into an *array of bytes* with the *BinaryReader* class and read the first two bytes represented in a string with the *BitConverter.ToString* method. Finally, I compare this string representation with the forbidden sequences.

Figure 4-43 shows a failed validation of the CSV file when a **.exe** file extension has been renamed into a **.csv** extension.
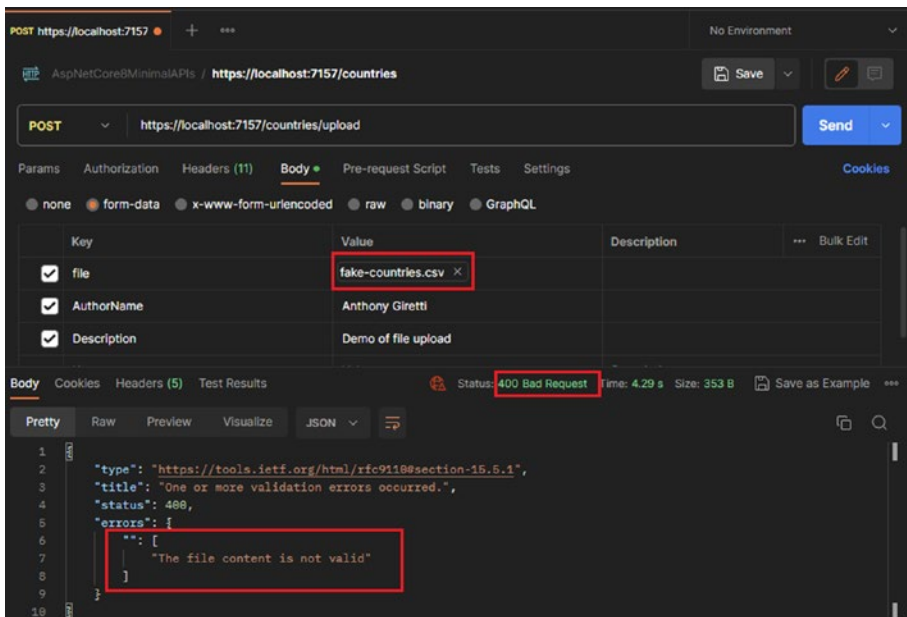
Figure 4-43. *Failed validation of an executable file renamed into a CSV file*

Note   At this time of writing, the *IFormFile* attribute is incompatible with the *FromForm* attribute when they are both nested in a custom class in minimal APIs. This should be fixed on the final release:

https://github.com/dotnet/aspnetcore/issues/49526.

You'll need to invoke the validator in a loop to validate each file if you upload several files simultaneously.

As you can see, file uploading can be tricky, whereas downloading is pretty straightforward. If you want to learn more about file signatures or *Magic bytes*, you can look at all possible signatures characterizing each type of file here: https://profilbaru.com/article/List_of_file_signatures.

Streaming Content

Content streaming is a CRUD operation because streaming content is a separate functionality, even if it's similar to downloading a file. Streaming is downloading a file temporarily stored on the device (browser, mobile application, etc.), with which it is possible to consume the content before it has been fully downloaded. Downloading, on the other hand, offers temporary storage of any file. Generally speaking, downloading doesn't allow you to consume content at the same time as downloading. To be able to stream a video, for example, you'll need to have in your hands two things:

1. The video stream (*Stream* object)

2. The file MIME type (e.g., "video/mp4")

Let's consider the *IStreamingService* interface, which exposes the *GetFileStream* method and returns the file stream and its MIME type encapsulated in a tuple, as shown in Listing 4-34.

Listing 4-34. The IStreamingService interface

```
namespace Domain.Services;

public interface IStreamingService
{
    Task<(Stream stream, string mimeType)> GetFileStream();
}
```

Let's inject it into the GET /streaming endpoint, and let's assume the implementation returns a .mp4 (video/mp4) video stream as shown in Listing 4-35.

Listing 4-35. The GET /streaming endpoint

```
app.MapGet("/streaming", async (IStreamingService
streamingService) =>
{
    (Stream stream, string mimeType) = await streamingService.
    GetFileStream();
    return Results.Stream(stream, mimeType,
    enableRangeProcessing: true);
});
```

As you can see, we need to use the *Results.Stream* method that takes, optionally, another parameter, *enableRangeProcessing*, set to true. The latter allows the server to manage partial content when a client requests a specific range of content, such as a video. It often happens when a client uses a media player to access a specific video part. If the client requests it, the response HTTP status code will be Partial Content (216) instead of OK (200).

If you remember in Chapter 1, I discussed the *Accept-Ranges*, *Range*, and *Content-Range* headers. Figure 4-44 shows the video being streamed (it also works with Postman).
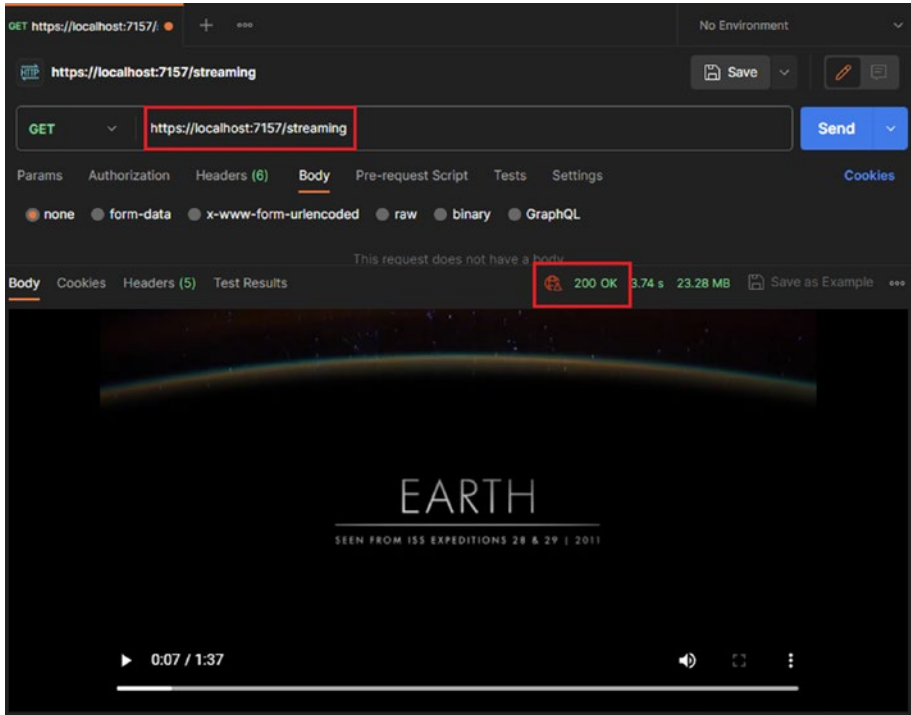
Figure 4-44. *The GET /streaming endpoint streaming a .mp4 video*

As you can see, it's pretty straight and effortless. If you want to build your streaming server, now you know how!

Handling CORS

When a website is launched, all data requested from the server must come from the same source, that is, from the same server or at least from the same HTTP domain (or sub-domain) (as several servers can hide behind

the same domain name). A security feature, *Same-Origin Policy (SOP)*, prohibits data loading from other domain names. This security applies to scripts like JavaScript that initiate HTTP requests to retrieve data.

To find out whether your API is authorized to serve a client from another domain, the browser will perform what's known as a *preflight*, that is, a preliminary request to the server. This preflight is performed via an HTTP request via the OPTIONS verb. The browser is then informed whether or not the server authorizes the HTTP request.

Various headers allow the browser to know the origin and purpose of the HTTP request received. These headers all have part of their name in common: they all start with *Access-Control-*.*

Here's the complete list of headers:

- **Access-Control-Allow-Origin**: Header returned by the server to tell the client which domains are authorized.

- **Access-Control-Allow-Credentials**: Header returned by the server to indicate to the client whether requests requiring credentials are authorized, like authorization headers. True or false will be returned to the client.

- **Access-Control-Allow-Headers**: Header returned by the server to indicate to the client which headers are authorized.

- **Access-Control-Allow-Methods**: Header returned by the server to indicate to the client which HTTP request methods are authorized.

- **Access-Control-Expose-Headers**: Header returned by the server to indicate to the client which headers will be returned in the HTTP response to a request.

- **Access-Control-Max-Age**: Header returned by the server indicating how long, in seconds, the response to the preliminary request will be cached.

- **Access-Control-Request-Headers**: Header sent by the client during the preliminary request, telling the server which headers will be sent to it.

- **Access-Control-Request-Method**: Header sent by the client during the preliminary request, telling the server which HTTP verbs will be sent to it.

To learn more about these headers, look at the Mozilla documentation here: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.

ASP.NET Core 8 allows you to set these headers, and we're going to concentrate on four of them, which are the most important and most frequently used:

1. Access-Control-Allow-Origin

2. Access-Control-Allow-Methods

3. Access-Control-Allow-Headers

4. Access-Control-Allow-Credentials

Let's start with a basic configuration of *CORS* in ASP.NET Core 8. Listing 4-36 shows a configuration named *AllowAll*, a policy that authorizes any origin, HTTP verb, or header. There is no need to set up the *Access-Control-Allow-Credentials* header when any origin is allowed.

Listing 4-36. Basic CORS configuration allowing any origin, verb, or header

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAll",
        builder =>
```

```
    {
        builder.AllowAnyHeader()
                .AllowAnyMethod()
                .AllowAnyOrigin();
    });
});
```

Suppose you try simultaneously allowing credentials with the *AllowCredentials* method when allowing any origin with the *AllowAnyOrigin* method; in that case, it will lead to an error, as shown in Figure 4-45.
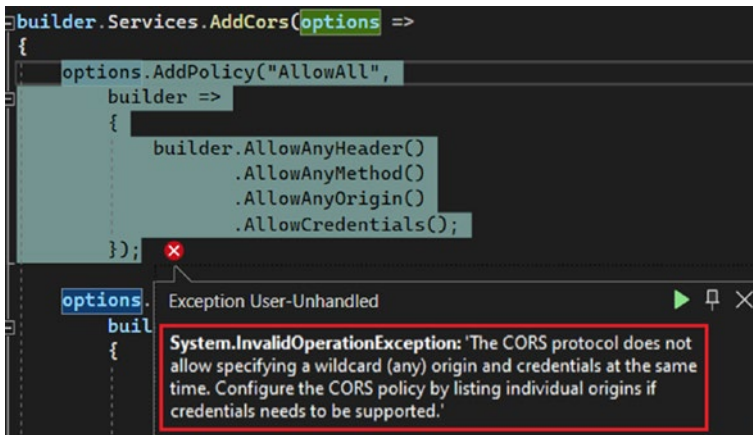
Figure 4-45. *Allowing credentials and any origin leads to an error*

This is only the configuration; this won't work if it's not enabled in the ASP.NET Core pipeline. Listing 4-37 shows how to enable the *CORS* middleware by using the *UseCors* method that takes as a parameter the *AllowAll* policy.

Listing 4-37. Enabling the "AllowAll" CORS policy

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

app.UseCors("AllowAll");
```

You can probably guess this is not the configuration to use for production. In production, you'll have to be more restrictive. Consider the following elements:

1. You'll want to authorize any header and credentials.

2. You'll want to authorize only the following verbs: GET, POST, PUT, and DELETE only, thus prohibiting all other verbs.

3. Above all, you'll want to filter the domains your customers can access by authorizing only the following domains: https://mydomain.com and https://myotherdomain.com.

Listing 4-38 shows this restricted configuration. The policy will be named "Restricted".

Listing 4-38. Restricted CORS configuration

```
options.AddPolicy("Restricted",
        builder =>
        {
            builder.AllowAnyHeader()
                    .WithMethods("GET", "POST", "PUT", "DELETE")
                    .WithOrigins("https://mydomain.com",
                    "https://myotherdomain.com")
                    .AllowCredentials();
        });
```

Let's try it out. Consider a JavaScript script, pure JavaScript, that executes an HTTP request to the server (it will try to reach my local API at the https://localhost:7157 address) with a not-allowed origin (http://localhost:5150) as shown in Listing 4-39.

Listing 4-39. JavaScript script attempting an HTTP request with a not-allowed origin

```
<script>
    function getCountries()
    {
        var xmlHttp = new XMLHttpRequest();
        xmlHttp.onreadystatechange = function() {
            if (xmlHttp.readyState == 4 && xmlHttp.
            status == 200)
                callback(xmlHttp.responseText);
        }
        xmlHttp.open("GET", "https://localhost:7157/
        countries", true);
        xmlHttp.send(null);
    }
    getCountries();
</script>
```

Since the origin (http://localhost:5150) is not allowed, the browser will return a *CORS* error, as shown in Figure 4-46.
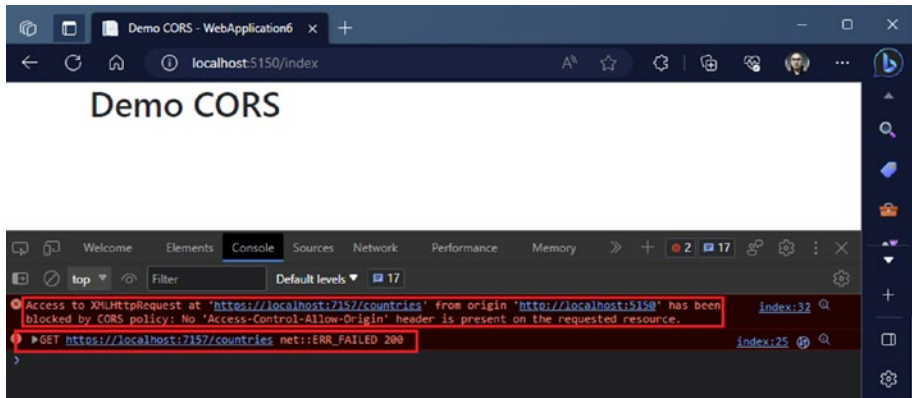
Figure 4-46. *Failed HTTP request made over JavaScript due to a not-allowed origin*

Managing *CORS* in your API is necessary for security reasons when a client uses scripts. I've shown you how to filter HTTP requests to prevent anyone from making potentially dangerous connections, thus reducing the risks associated with requests from origins other than your API. To learn more about *CORS*, you can read Mozilla's lovely documentation: `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`.

API Versioning

When you program an API, you can expect it to evolve over time. In many cases, this evolution will have no consequences for the customer, who can use the API without changing anything. In other words, it's retro-compatible. However, if your API changes its behavior, that is, if the data contract exposed to the customer changes, the customer will be blocked from using the API as before the update. To solve this problem, you'll need to version your API, that is, each endpoint (or some of them) will be assigned version numbers, and each endpoint version will have its exchange contract with the client and its back-end behaviour. So, for a

given endpoint, you'll have as many endpoints to maintain as there are versions of that endpoint. I'll show you how to do it properly.

There are three ways to manage an API version:

1. Using **headers**, for example, with a custom header **api-version** : 2

2. Using the **route**, for example, **/v2**/countries

3. Using the **queryString**, for example, /countries**?api-version=2**

I will not talk about versioning using the query string, because, in my opinion, it's the least clean solution, as it makes the URL less readable, with parameters depending on the URL. Using the route is still pretty readable, and using headers is even cleaner.

Versioning by Headers

First off, to be able to manage API versioning, we'll need to download the following Nuget package:

```
NuGet\Install-Package Asp.Versioning.Http
```

By doing that, we are now ready to configure versioning in an ASP. NET Core 8 minimal API. Listing 4-40 shows the configuration by using the *AddApiVersioning* extension method.

Listing 4-40. The API versioning configuration

```
builder.Services.AddApiVersioning(options =>
{
    options.DefaultApiVersion = new ApiVersion(1, 0);
    options.ReportApiVersions = true;
    options.AssumeDefaultVersionWhenUnspecified = true;
```

```
    options.ApiVersionReader = new HeaderApiVersionReader("api-
    version");
});

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

The important thing here is to define the configuration **before** invoking the *Build* method.

This configuration does four things:

- **DefaultApiVersion:** Specifies the default version of the API to fall back to when no version is specified, version 1.0 in this example

- **ReportApiVersions:** Specifies in the response headers the supported versions by the API, true in this example

- **AssumeDefaultVersionWhenUnspecified:** Allows to fall back to the version defined with the option DefaultApiVersion when a version is not specified, true in this example

- **ApiVersionReader:** Specifies the header's name in the request when a client wants to select a specific version, *api-version* in this example

After that, we need to define all the versions available. Let's say we will expose two versions of the API: 1.0 and 2.0. Versions 1.0 and 2.0 are a *version set* of the API. This version set must be declared **after** the *Build* method.

Listing 4-41 shows the configuration of the version set made with the *NewApiVersionSet* extension method.

Listing 4-41. The 1.0 and 2.0 version set defined

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

var versionSet = app.NewApiVersionSet()
                    .HasApiVersion(1.0)
                    .HasApiVersion(2.0)
                    .Build();
```

The *HasApiVersion* method allows us to declare each version we want to manage in the API, and then we have to build the version set with the *Build* method. As a result, the *versionSet* variable will be assigned to endpoints.

Listing 4-42 shows four endpoints using the version set obtained previously and assigned to each endpoint with the *WithApiVersionSet* method.

Listing 4-42. Assigning the version set

```
app.MapGet("/version", () => "Hello version 1").WithApiVersion
Set(versionSet).MapToApiVersion(1.0);
app.MapGet("/version", () => "Hello version 2").WithApiVersion
Set(versionSet).MapToApiVersion(2.0);
app.MapGet("/version2only", () => "Hello version 2 only").With
ApiVersionSet(versionSet).MapToApiVersion(2.0);
app.MapGet("/versionneutral", () => "Hello neutral version").
WithApiVersionSet(versionSet).IsApiVersionNeutral();
```

As you can see, I declared twice the GET /version endpoint, but each declaration is assigned to a specific version, 1.0 and 2.0, using the *MapToApiVersion* extension method. The GET /version2only endpoint has only a single version assigned, 2.0. If we don't specify any version, it will try to fall back to version 1.0 as we configured it earlier, but since it has not been assigned to this endpoint, it will lead to a Bad Request (400) error.

If we try to pass version 1.0 in the headers, it will lead to the same error. Finally, the GET /versionneutral will only handle the versions defined in the version set and will remain idempotent (same behavior) regarding the version passed in the headers as long as it is declared in the version set. If any version is passed in the headers and is not declared in the version set, it will lead to a Bad Request (400) error. Omitting the version in the headers will fall back to version 1.0 and return an OK (200), since the latter is declared in the version set.

Suppose you want to make any endpoint insensitive to any version. In that case, you will have to remove the *WithApiVersionSet* and any other function related to versioning in the endpoint declaration as follows:

```
app.MapGet("/versioneutral", () => "Hello neutral version")
```

Figure 4-47 shows the result of the execution of the GET /version endpoint bound to version 1.0 by passing the *api-version* header value 1.0.
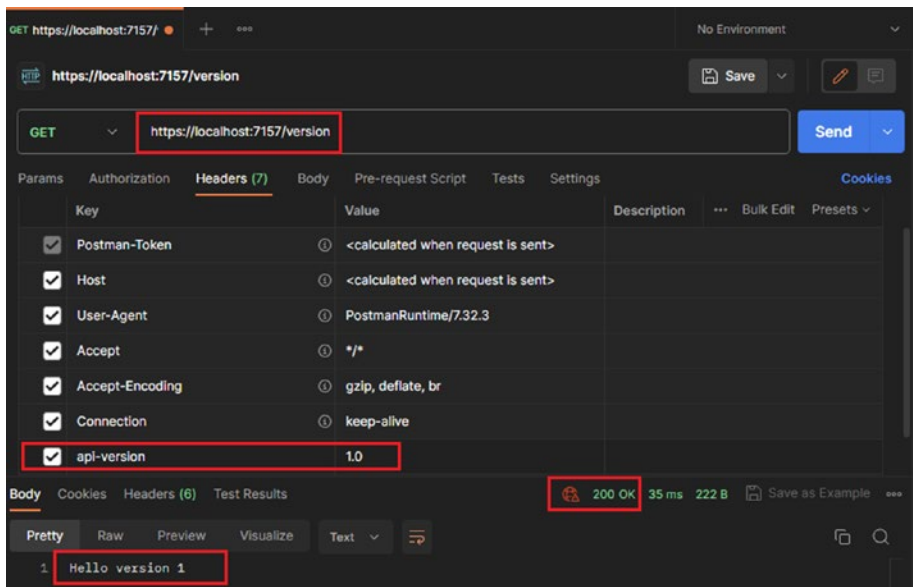
Figure 4-47. *The GET /version endpoint execution result bound to version 1.0*

As you can see, the correct output, "Hello version 1", has been returned to Postman.

Figure 4-48 shows the output of the GET /version endpoint bound to version 2.0.
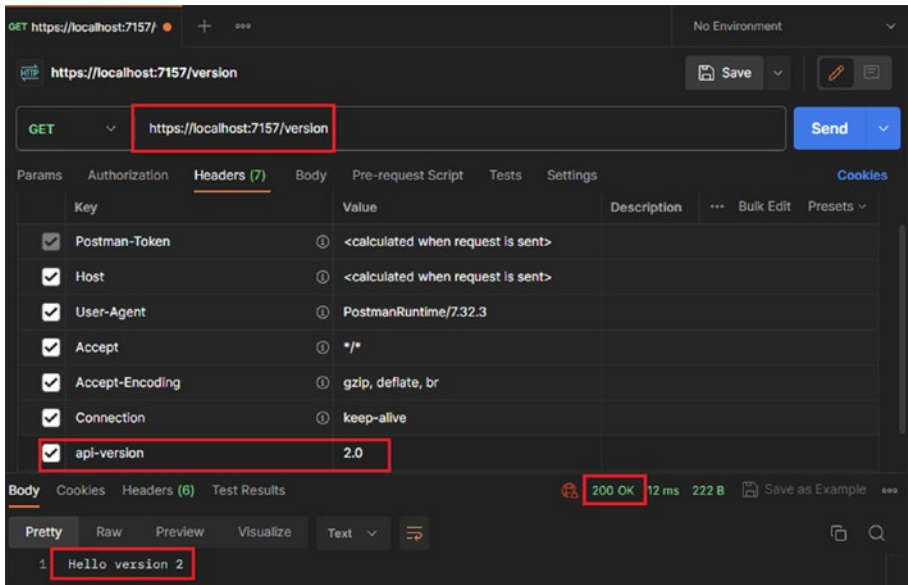
Figure 4-48. *The "/version" endpoint execution result bound to version 2.0*

Again, the correct output, "Hello version 2", has been returned.

Let's now try the GET /version2only, which is only bound to version 2.0. Figure 4-49 shows the output when trying to omit the *api-version* header.
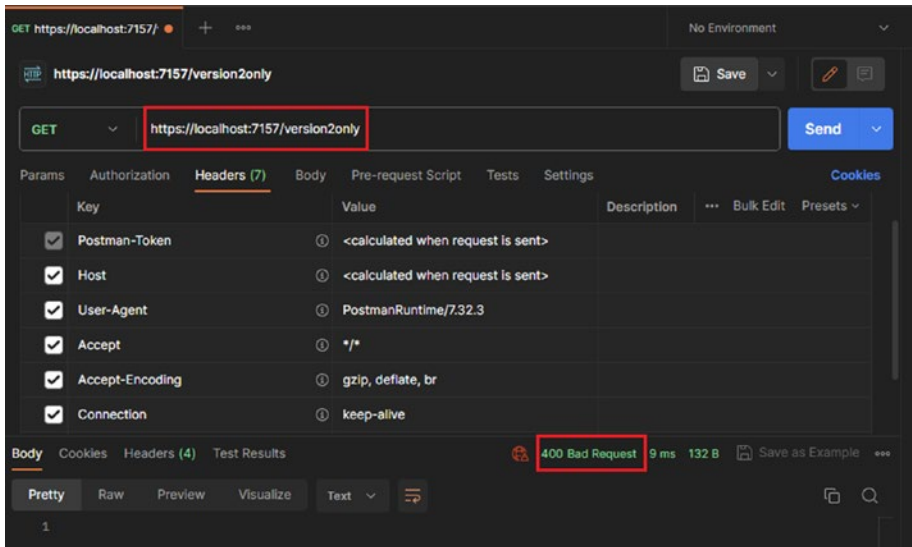
Figure 4-49. *The "/version2only" endpoint not bound to any version execution result*

The output is as expected, as Bad Request (400) has been returned. Figure 4-50 shows the output when the same endpoint is bound to version 1.0.
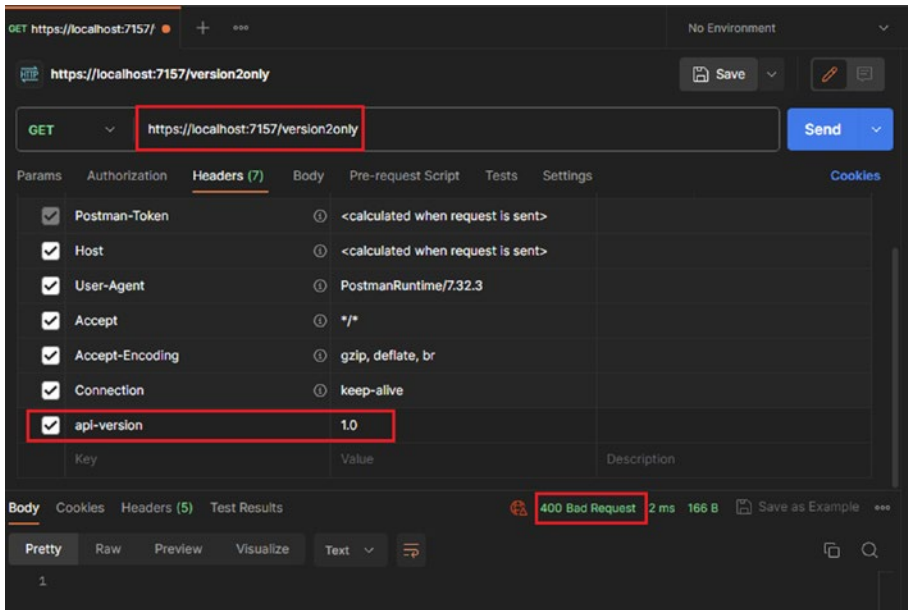
Figure 4-50. *The "/version2only" endpoint bound to the version 1.0 execution result*

As expected, we get here a Bad Request (400). Figure 4-51 now shows the output when version 2.0 is passed to the headers.
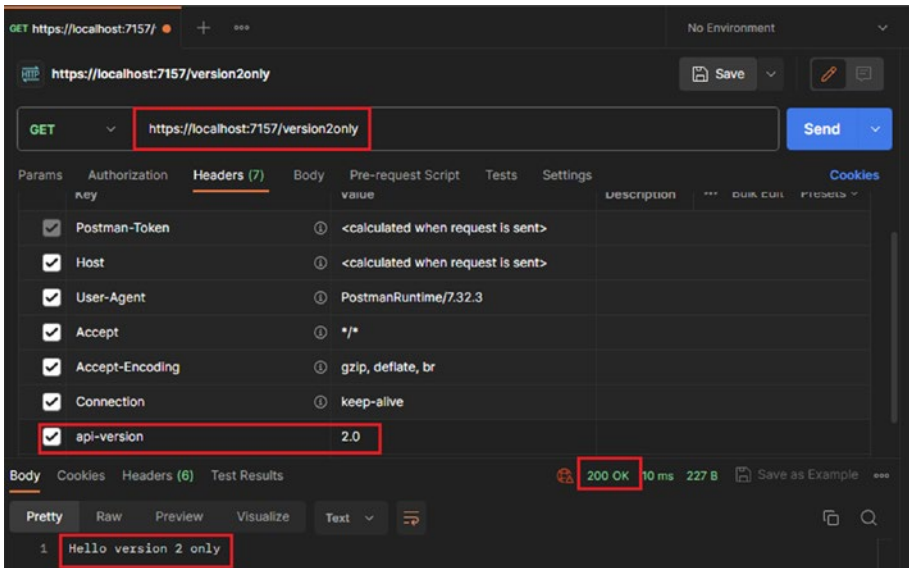
Figure 4-51. *The "/version2only" endpoint bound to the version 2.0 execution result*

As expected, we get there an OK (200) response.

Let's try now to see the output when passing any version (or nothing) to the GET /versionneutral endpoint. Figures 4-52, 4-53, and 4-54 show, respectively, the output when no version is passed in the headers, when version 2.0 is placed in the headers (declared in the version set), and, finally, when version 3.0 is passed in the headers (not declared in the version set).
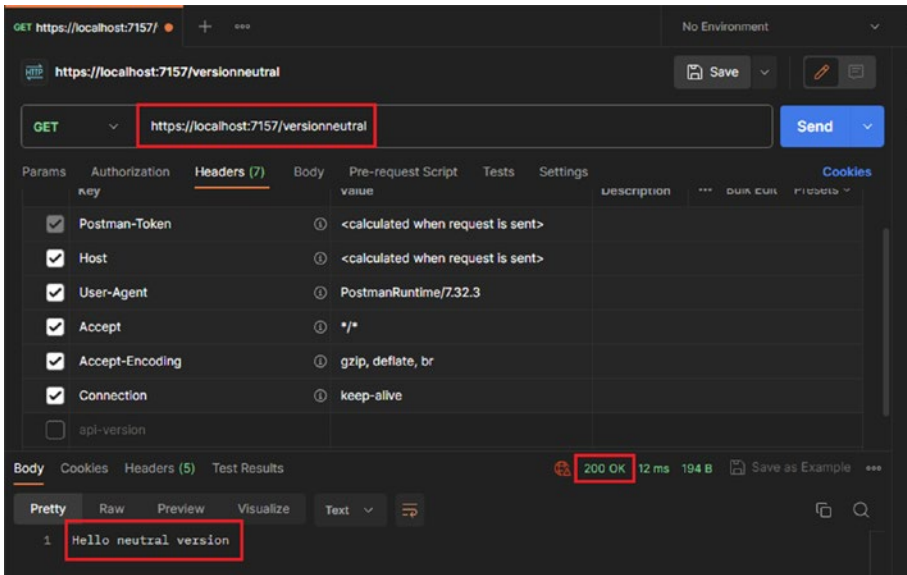
Figure 4-52. *The GET /versionneutral endpoint not bound to any version execution result*
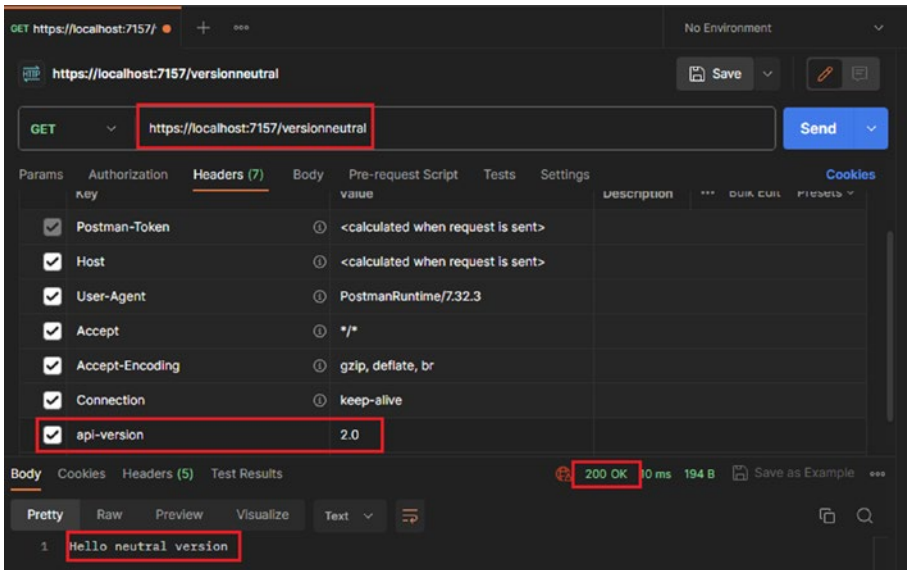
Figure 4-53. *The GET /versionneutral endpoint bound to the version 2.0 execution result*
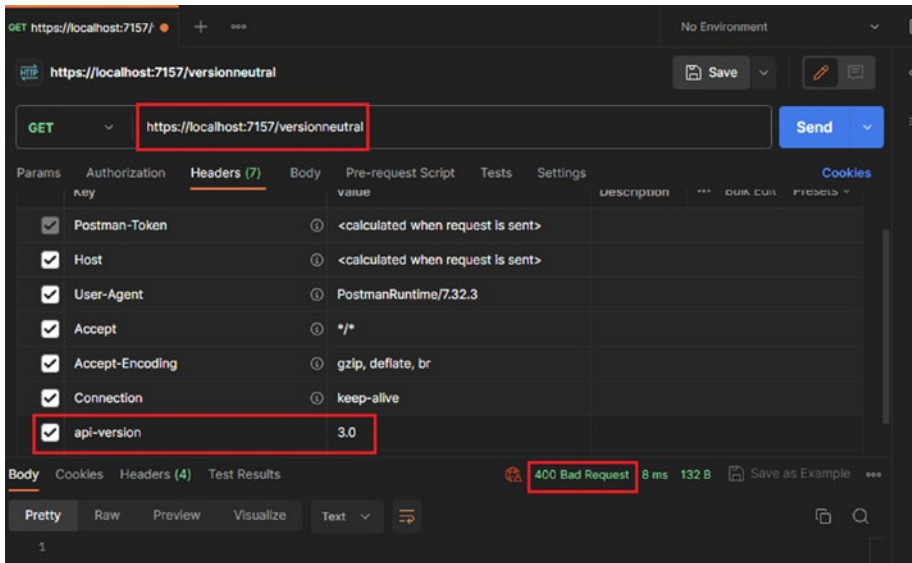
Figure 4-54. *The GET /versionneutral endpoint bound to the version 3.0 execution result*

All the output results meet the requirements as discussed earlier.

Passing the version in the headers is probably the best way to handle the version and cleaner since you don't have to modify the URL, but I prefer to manage it in the route. I will show you how in the following subsection.

When I bring back the API documentation topic, I will show you how to make version header input available on the Swagger page.

Versioning by Route

I'll now show you my favorite way of managing my API versioning. We will rely on *RouteGroups* obtained by defining routes on the *RouteGroupBuilder* class. You'll love it! Let's consider the groups of API, *GroupVersion1* and *GroupVersion2 methods*, where the GET /version and /version2only endpoints are declared within, as shown in Listing 4-43.

Listing 4-43. The GET /version and /version2only endpoints defined in RouteGroups

```
namespace AspNetCore8MinimalApis.RouteGroups;

public static class VersionGroup
{
    public static RouteGroupBuilder GroupVersion1(this
    RouteGroupBuilder group)
    {
        group.MapGet("/version", () => $"Hello version 1");
        return group;
    }

    public static RouteGroupBuilder GroupVersion2(this
    RouteGroupBuilder group)
    {
        group.MapGet("/version", () => $"Hello version 2");
        group.MapGet("/version2only", () => $"Hello
        version 2 only");
        return group;
    }
}
```

I have made the GET /version endpoint available in the two groups. Each one corresponds to a specific version of their endpoints. In group 2 only, defined by the *GroupVersion2 method*, I made the GET /version2only endpoint available. As I showed you earlier in this chapter, we must add them to the ASP.NET Core pipeline. Listing 4-44 shows the ASP.NET Core pipeline registration, and I gave them a version name by defining a URL trunk for each, */v1* and */v2*.

Listing 4-44. The GET /version and /version2only endpoints
defined in RouteGroups under the /v1 and /v2 URL trunks

```
app.MapGroup("/v1").GroupVersion1();
app.MapGroup("/v2").GroupVersion2();

app.Run();
```

As you can see, it's straightforward. There is also another reason I love
managing versioning in this way; the reason is that if you put the version
in the route and you get wrong with the specified version, it won't lead to
a Bad Request (400) but to a Not Found (404) error, which is more evident
to me. After all, it's normal because we manage API versioning by route.
Do you feel me? Figures 4-55 and 4-56 show, respectively, the output when
invoking the GET /v1/version2only (which has never been defined) and
GET /v2/version2only endpoints.
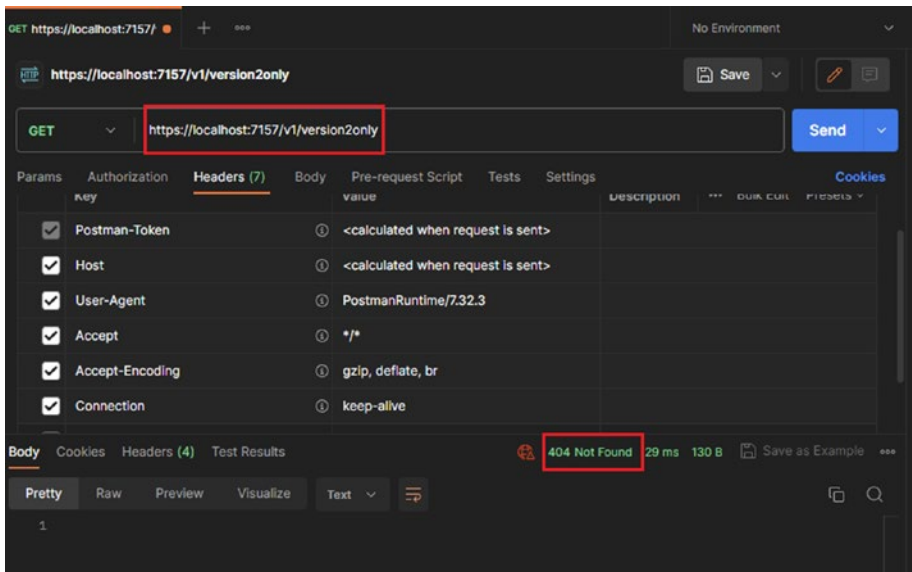
Figure 4-55. *The GET /v1/version2only endpoint bound, in the route,
to the version 1 execution result*
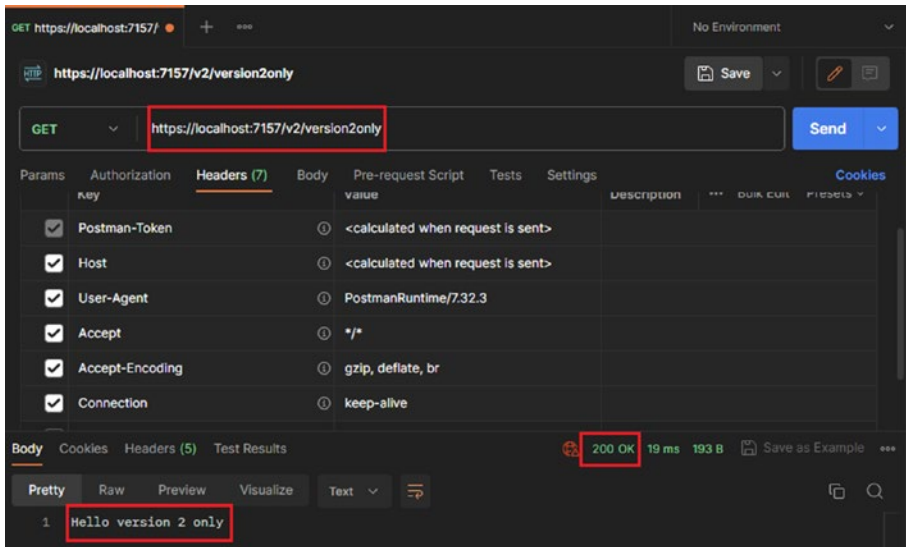
Figure 4-56. *The GET /v2/version2only endpoint bound, in the route, to the version 2 execution result*

The behavior of these endpoints is meeting the expectations. I'm sure now, and you will agree, that facing Not Found (404) errors is more evident!

One more thing! *Route groups* are compatible with Swagger; by that, Swagger will display the different registered routes in the route groups, and I will show that to you in the next section.

Documenting APIs

In Chapter 2, I introduced you to API documentation using the OpenAPI specification, which I also used to introduce Swagger and its implementation in ASP.NET Core 8.

Documenting your API is fundamental insofar as it's the documentation your client will use to be able to consume your API. In this section, I'm going to show you options for customizing your documentation:

1. Managing API versions (with header versioning only)

2. Adding comments on endpoints

3. Grouping your endpoints with tags

4. Other customizations

First off, ensure that the following Nuget packages are installed if they are not yet:

- Asp.Versioning.Http

- Asp.Versioning.Mvc.ApiExplorer

- Microsoft.AspNetCore.OpenApi

- Swashbuckle.AspNetCore

I understand it doesn't sound very clear because there are many packages. Let me explain to you why. As we saw previously, the package *Asp.Versioning.Http* enables you to version your minimal endpoints. The *Asp.Versioning.Mvc.ApiExplorer* will allow you to let ASP.NET Core (its API Explorer) know that several versions are available for an endpoint, as we did in the previous section. *Swashbuckle.AspNetCore* and *Microsoft.AspNetCore.OpenApi* help make available some Swagger features and some customizations. As I showed you in Chapter 2, these two packages are automatically brought on the ASP.NET Core Web API project if you click the "Enable OpenAPI support" option while creating a new project. Don't forget to unselect the "Use controllers" to work only with minimal APIs.

Managing API Versions in Swagger

As a first step, we will need to create a class that will be registered as Swagger options, and these options will allow us to register in the ASP.NET Core pipeline the generation of the Swagger documentation. Listing 4-45 shows the *SwaggerConfigurationsOptions* class that inherits from the *IConfigureOptions<SwaggerGenOptions>,* where the *SwaggerGenOptions* interface parameter comes from the *SwashBuckle* assembly.

Listing 4-45. The SwaggerConfigurationsOptions class

```
using Asp.Versioning.ApiExplorer;
using Microsoft.Extensions.Options;
using Microsoft.OpenApi.Models;
using Swashbuckle.AspNetCore.SwaggerGen;

namespace AspNetCore8MinimalApis.Swagger;

public class SwaggerConfigurationsOptions : IConfigureOptions<S
waggerGenOptions>
{
    private readonly IApiVersionDescriptionProvider _
    apiVersionDescriptionProvider;

    public SwaggerConfigurationsOptions(
            IApiVersionDescriptionProvider
            apiVersionDescriptionProvider)
    {
        _apiVersionDescriptionProvider =
        apiVersionDescriptionProvider;
    }

    public void Configure(SwaggerGenOptions options)
    {
```

```
        foreach (var description in _apiVersionDescription
        Provider.ApiVersionDescriptions)
        {
            options.SwaggerDoc(description.GroupName, Create
            OpenApiInfo(description));
        }
    }

    private static OpenApiInfo CreateOpenApiInfo(
                        ApiVersionDescription description
                    )
    {
        var info = new OpenApiInfo()
        {
            Title = "ASP.NET Core 8 Minimal APIs",
            Version = description.ApiVersion.ToString()
        };

        return info;
    }
}
```

Now, let's take the existing GET /version, /version2only, and /versionneutral endpoints again and try to run them through Swagger. To get it done, let's rewrite the Program.cs file by completing the API versioning with the Swagger feature, as shown in Listing 4-46.

Listing 4-46. The Program.cs file configured with customized Swagger documentation for API versioning

```
using Asp.Versioning;
using Asp.Versioning.Conventions;
using AspNetCore8MinimalApis.Swagger;
using Swashbuckle.AspNetCore.SwaggerGen;
```

```
using Microsoft.Extensions.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddApiVersioning(options =>
{
    options.DefaultApiVersion = new ApiVersion(1, 0);
    options.ReportApiVersions = true;
    options.AssumeDefaultVersionWhenUnspecified = true;
    options.ApiVersionReader = new HeaderApiVersionReader("api-
    version");
})
.AddApiExplorer(options =>
{
    options.GroupNameFormat = "'v'VV"; // Formats the version
as follow: "'v'major[.minor]"
});

builder.Services.AddSwaggerGen();
builder.Services.AddSingleton<IConfigureOptions<SwaggerGenOptio
ns>, SwaggerConfigurationsOptions>();
var app = builder.Build();

// var apiVersionDescriptionProvider = app.Services.GetRequired
Service<IApiVersionDescriptionProvider>(); Not working properly
in ASP.NET Core 8 preview
app.UseSwagger().UseSwaggerUI(c =>
{
    // Workaround, hardcoding versions to be displayed
    in Swagger
    c.SwaggerEndpoint($"/swagger/v1.0/swagger.json",
    "Version 1.0");
```

```
    c.SwaggerEndpoint($"/swagger/v2.0/swagger.json",
    "Version 2.0");

      // Not working correctly in ASP.NET Core 8 preview
    //foreach (var description in
    apiVersionDescriptionProvider.ApiVersionDescriptions.
    Reverse())
    //{
    //     c.SwaggerEndpoint($"/swagger/{description.GroupName}/
         swagger.json",
    //          description.GroupName.ToUpperInvariant());
    //}
});

var versionSet = app.NewApiVersionSet()
                    .HasApiVersion(1.0)
                    .HasApiVersion(2.0)
                    .Build();

app.MapGet("/version", () => "Hello version 1").
WithApiVersionSet(versionSet).MapToApiVersion(1.0);

app.MapGet("/version", () => "Hello version 2").
WithApiVersionSet(versionSet).MapToApiVersion(2.0);

app.MapGet("/version2only", () => "Hello version 2 only").
WithApiVersionSet(versionSet).MapToApiVersion(2.0);

app.MapGet("/versionneutral", () => "Hello neutral version").
WithApiVersionSet(versionSet).IsApiVersionNeutral();

app.Run();
```

As you can see, the Swagger feature is enabled through the *AddSwaggerGen* method and the *SwaggerConfigurationsOptions* class registered in the pipeline. You may have noticed the *AddApiExplorer* extension method applied to the *AddApiVersioning* method. The latter enables any versions declared in the API to get discovered by Swagger, and it also takes as an option the *GroupNameFormat* option, which allows configuring the versions formatted as 1.0 or 2.0 or x.0.

UseSwagger and *UseSwaggerUI* are the middlewares that allow running the Swagger interface; *UseSwagger* activates the generation of the *swagger.json* URL (that I customized because of versioning), which contains all the OpenAPI data in JSON format to get displayed on a web HTML page activated by the *UseSwaggerUI* method. By default, the Swagger HTML page is reachable at the address: /swagger/index.html. This URL is customizable, but I don't think it's relevant to show you how; since the HTML page allows you to see your API documentation, it's easier to remember the default page. At this time of writing (I'm using ASP. NET Core 8 preview), the *UseSwaggerUI* method cannot find all the API versions from the *IapiVersionDescriptionProvider* service. Still, it **could find all versions** of the API when the *SwaggerConfigurationsOptions* class was executed to register the documentation of each version. In other words, All versions of the documentation are registered, but because of a probable bug in the preview of ASP.NET Core 8, displaying all of them in Swagger does not work. I hard-coded all versions to be displayed.

To illustrate this, Figures 4-57 and 4-58 show the output on the /swagger.html endpoint for versions 1 and 2, respectively.
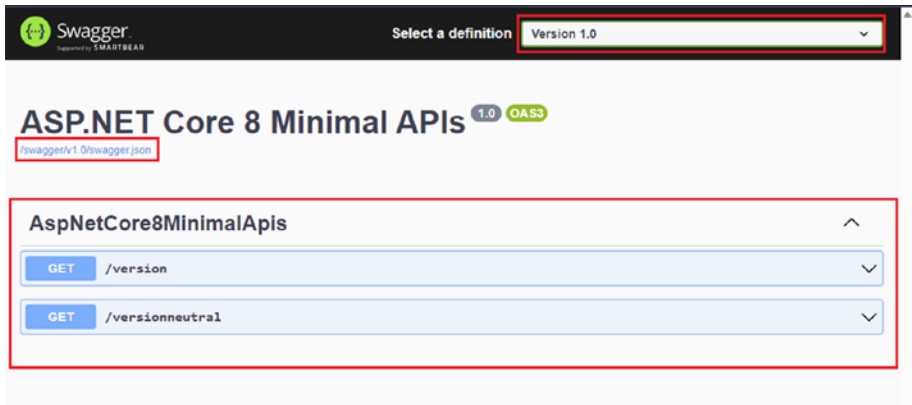
Figure 4-57. *Version 1.0 of the API displayed in Swagger*
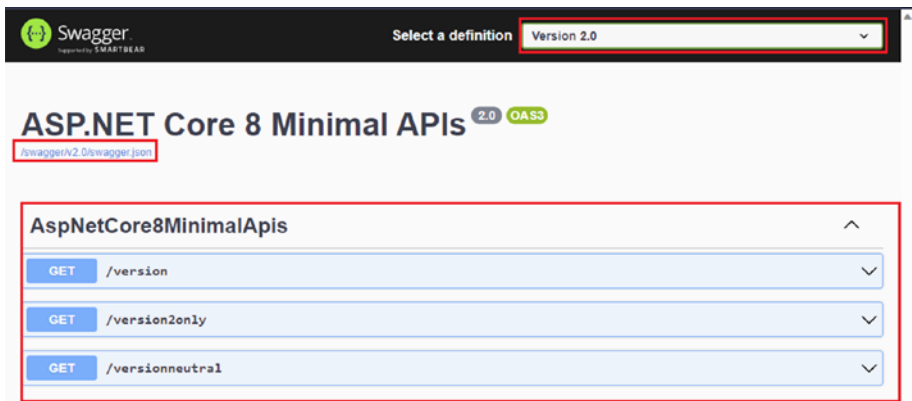
Figure 4-58. *Version 2.0 of the API displayed in Swagger*

From this, you can select the API version you want to see in the drop-down on the top right of the Swagger HTML page. By default, version 1.0 is displayed since it was what we configured. You can also notice that our GET /version2only endpoint is only available on version 2.0. It's exactly what we expected!

Finally, the *swagger.json* file on the top left is clickable, and if you click it, you'll see its content, as shown in Figure 4-59.
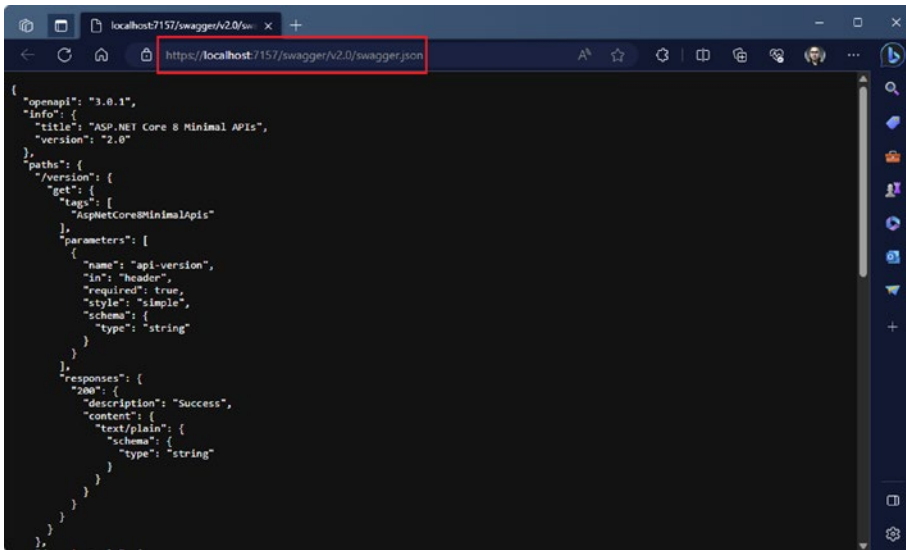
Figure 4-59. *Version 2.0 of the API displayed in JSON with the swagger.json file*

You'll remember that we used to choose the API version with the *api-version* header, right? Well, magic trick! Swagger automatically displays an input allowing you to pass a value to this header (using the *AddApiExplorer* extension method), as shown in Figure 4-60.

Figure 4-60. *Swagger displaying an input for the api-version header automatically*

Adding Comments on Endpoints

If by any chance you were tempted to add XML comments to your methods' names and display them in Swagger to improve their documentation, this is not possible as with web APIs! Adding comments via /// <summary> is unavailable with minimal APIs on the method's name. Since the endpoint methods are lambda expressions, comments on lambda are not handled, even if you encapsulate them in static methods in a separate class. But it works on custom endpoint input parameter objects (e.g., not on primitives), and I will show you how.

Note   Any object passed by dependency injection won't be handled regarding XML comments. XML comments only work on input parameters.

199

To make it work on parameters, let's create the *AddXmlComments* extension method on the *SwaggerGenOptions* class within the *SwaggerXmlComments* static class, as shown in Listing 4-47.

Listing 4-47. The SwaggerXmlComments static class with its AddXmlComments extension method

```
using Swashbuckle.AspNetCore.SwaggerGen;
using System.Reflection;

namespace AspNetCore8MinimalApis.Swagger;

public static class SwaggerXmlComments
{
    public static void AddXmlComments(this SwaggerGenOptions
    options)
    {
        var xmlFile = $"{Assembly.GetExecutingAssembly().
        GetName().Name}.xml";
        var xmlPath = Path.Combine(AppContext.BaseDirectory,
        xmlFile);
        options.IncludeXmlComments(xmlPath);
    }
}
```

This method finds the XML documentation generated (by scanning the current assembly) after configuring your API's *.csproj* file as shown in Listing 4-48.

Listing 4-48. Editing the API .csproj file to enable the XML comments

```
<PropertyGroup>
    <GenerateDocumentationFile>true</
    GenerateDocumentationFile>
    <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

We need now to register the XML comments feature in the *AddSwaggerGen* as an option, as shown in Listing 4-49.

Listing 4-49. Registering the XML comments feature in the Swagger configuration

```
builder.Services.AddSwaggerGen(options =>
{
    options.AddXmlComments();
});
```

Now XML comments are enabled. If we get back to the POST /countries endpoint we discussed earlier in this chapter, you can see the *Country* input parameter class as shown in Listing 4-50.

Listing 4-50. The POST /countries endpoint with the Country input parameter class

```
app.MapPost("/countries", ([FromBody] Country country,
IValidator<Country> validator, ICountryMapper mapper,
ICountryService countryService) => {
    /// code
}).WithApiVersionSet(versionSet)
  .MapToApiVersion(1.0);
```

I will add XML comments on its properties, and we can display them in Swagger. Remember, since *IValidator<Country>*, *ICountryMapper*, and *ICountryService* services are injected by dependency, they won't show up in Swagger and any XML comments will appear. Listing 4-51 shows comments on *Country* class parameters.

Listing 4-51. The Country class updated with summary comments

```
using System.ComponentModel.DataAnnotations;

namespace AspNetCore8MinimalApis.Models;

public class Country
{
    /// <summary>
        /// The country Id
        /// </summary>
    public int? Id { get; set; }

    /// <summary>
        /// The country name
        /// </summary>
    public string Name { get; set; }

    /// <summary>
        /// The country description
        /// </summary>
    public string Description { get; set; }

    /// <summary>
        /// The country flag URI
        /// </summary>
    public string FlagUri { get; set; }
}
```

If we run now the API and open the Swager HTML page, we should see the comments as shown in Figure 4-61.
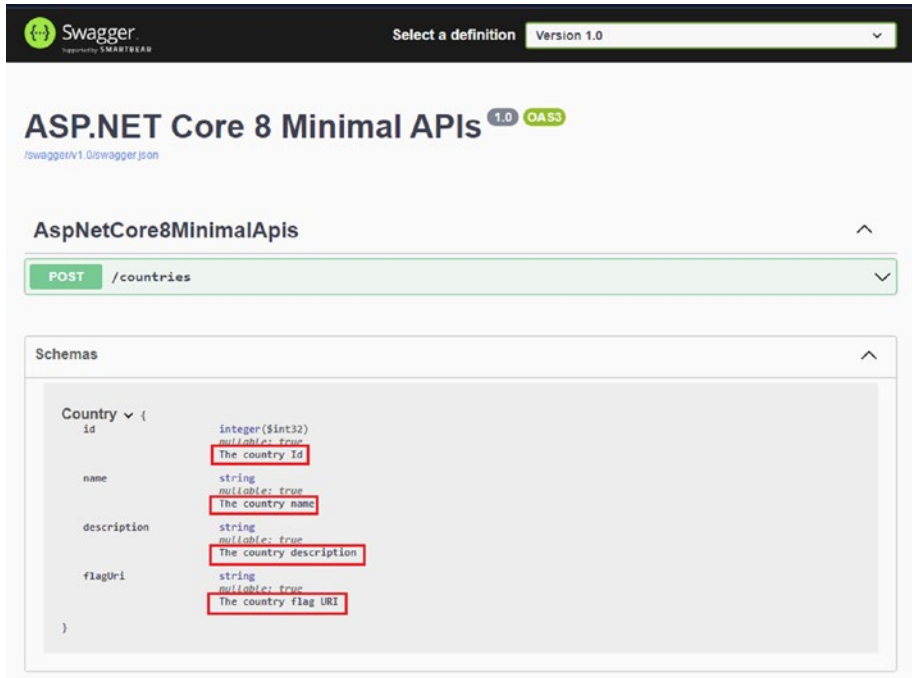
Figure 4-61. *The Country class displaying the XML comments*

If you want to add comments to your endpoints, you'll need to use Swagger annotations, so I suggest you download the Nuget package *Swashbuckle.AspNetCore.Annotations.* Once installed, add *EnableAnnotations* as an option to the *AddSwaggerGen* method, as shown in Listing 4-52.

Listing 4-52. Enabling annotations in Swagger

```
builder.Services.AddSwaggerGen(options =>
{
    options.EnableAnnotations();
    options.AddXmlComments();
});
```

We can now add annotations on minimal endpoints. To achieve this, we can use the *SwaggerOperation* attribute to replace the XML comments, as shown in Listing 4-53.

Listing 4-53. Adding the SwaggerOperation attribute for commenting endpoints (summary and description)

```
app.MapGet("/versionneutral", [SwaggerOperation(Summary =
"Neutral version", Description = "This version is neutral")] ()
=> "Hello neutral version")
.WithApiVersionSet(versionSet)
.IsApiVersionNeutral();
```

I put "Neutral version" as a summary and "This version is neutral" as a description on the GET /versionneutral endpoint, and it displays perfectly when running Swagger, as shown in Figure 4-62.
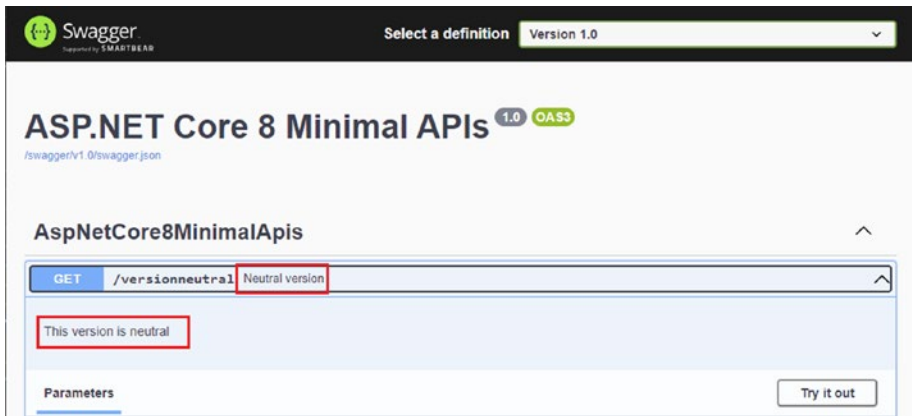
Figure 4-62. *The "/versionneutral" enhanced with comments (summary and description)*

If you dislike using Swagger annotation, you can use the *Microsoft. AspNetCore.OpenApi* assembly, which also enables the possibility to add a summary and a description on an endpoint by using the *WithOpenApi* extension method as shown in Listing 5-54.

Listing 4-54. Using the WithOpenApi extension method for commenting endpoints

```
app.MapGet("/versionneutral",() => "Hello neutral version")
.WithApiVersionSet(versionSet)
.IsApiVersionNeutral();
.WithOpenApi(operation => new(operation)
{
    Summary = "This is a summary",
    Description = "This is a description"
});
```

The output in Swagger will be the same as the *SwaggerOperation* usage, as shown previously in Figure 4-62.

205

Grouping Endpoints by Tag

Swagger lets you use tags to group your endpoints. What does this mean in concrete terms? When the Swagger HTML page is displayed, you'll have different sections with a title, and the name you give to the tag will be the section's name. Personally, I see two uses for tags:

1. Grouping endpoints by version, if you're using versioning by route with *RouteGroups*

2. Grouping by feature, for example, all endpoints linked to countries, the others linked to another feature

As an example, let's take versioned route groups, to which we'll assign a tag using the *WithTag* extension method that will define a section for each version of the route groups, as shown in Listing 4-55.

Listing 4-55. Adding the WithTag extension method on route groups

```
app.MapGroup("/v1")
    .GroupVersion1()
    .WithTags("V1");

app.MapGroup("/v2")
    .GroupVersion2()
    .WithTags("V2");
```

If we execute the Swagger page, it should display the sections defined by the *V1* and *V2* tags, as shown in Figure 4-63.

Figure 4-63. *Route groups split by version tag*

I don't know what you think, but it's a simple and effective way of managing versioning by route with route grouping.

Other Customizations

The *Microsoft.AspNetCore.OpenApi* assembly (from the Nuget package with the same name) gives more features. I will introduce you to my favorites:

- **The possibility to hide any endpoint from being exposed in Swagger, which often happens when you don't want to expose to your client the existence of a particular endpoint**: The reason could be that the hidden endpoint is used for a remote system that makes regular HTTP requests on your API to verify if the latter responds correctly. It's the case, for example, on Microsoft Azure. Some services (like Azure API

Management) need to reach the API regularly to verify its state. I won't explore more details here since it's not the book's topic.

- **The possibility to mark any endpoint as deprecated**: It happens more often than you think, especially when you manage several versions of a bunch of endpoints. Before removing the endpoints that you don't want to maintain in the near future, you will, for a certain time, annotate them as deprecated to warn your client that these endpoints will be removed soon.

- **The possibility to describe endpoint responses**: Any endpoint may return a different response depending on the behavior defined in its logic. Let's say that an endpoint you designed may return an OK (200) when everything is going well, a Timeout (408), or even an Internal Error (500), which may happen when something is going wrong.

Hiding an Endpoint

If you want to hide an endpoint or many of them, you can use the *ExcludeFromDescription* extension method. Listing 4-56 shows the *ExcludeFromDescription* extension method on a route group, which handles a bunch of endpoints tagged with the *V1* version.

Listing 4-56. Adding the ExcludeFromDescription extension method on a route group

```
app.MapGroup("/v1")
    .GroupVersion1()
    .WithTags("V1")
    .ExcludeFromDescription();
```

```
app.MapGroup("/v2")
    .GroupVersion2()
    .WithTags("V2");
```

As expected, the group of endpoints tagged with the "V1" version doesn't show up in Swagger, as shown in Figure 4-64.
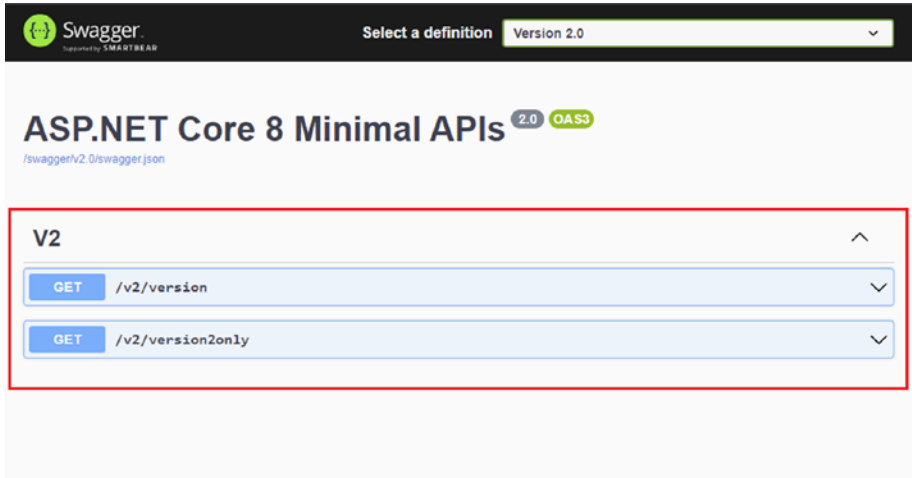
Figure 4-64. *Excluding the V1 route group from the Swagger documentation*

Deprecating an Endpoint

Let's say we want to annotate an endpoint, or a group of endpoints, as deprecated. We have to use the *WithOpenApi* extension method with the *Deprecated* option set to true, as shown in Listing 4-57.

Listing 4-57. Adding the `WithOpenApi` extension method on a route group with the Deprecated option

```
app.MapGroup("/v1")
    .GroupVersion1()
    .WithTags("V1")
```

```
.WithOpenApi(operation => new(operation)
    {
         Deprecated = true
    });
```

The group of deprecated endpoints will still show up in Swagger, but the text will be grayed out with the addition of "Warning: Deprecated," as shown in Figure 4-65.
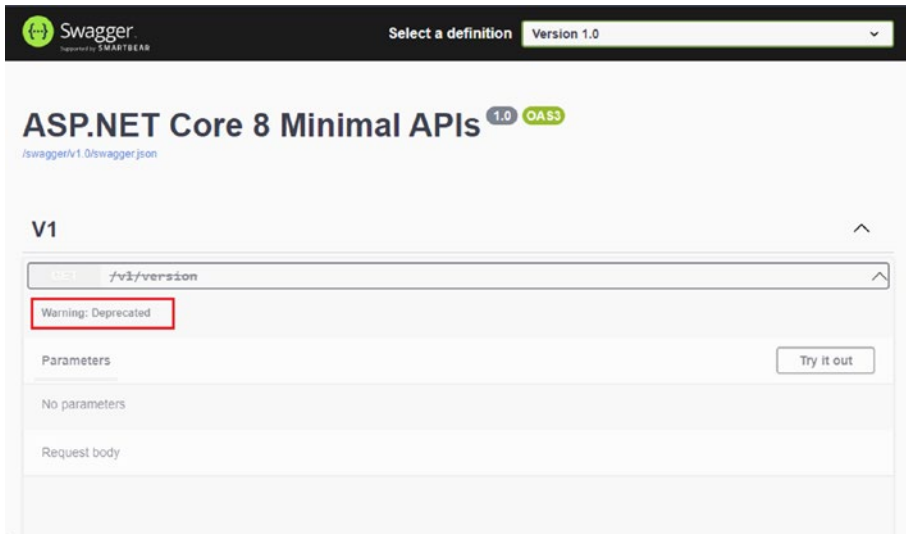
Figure 4-65. *Annotate the V1 route group as deprecated*

Describing Endpoint Responses

Describing the endpoint's possible responses is a great way to improve the client experience when reading the Swagger documentation. Remember that your client will need to handle the possible errors on their end and rely on the possible errors you will expose to them. To illustrate this, let's rework the GET /countries/download endpoint by enriching it with four possible responses:

- OK (200) that returns a *stream* (any file download is a *Stream* object) and video/mp4 MIME type

- Not Found (404)

- Internal Error (500)

- Request Timeout (408)

You may have noticed that only OK (200) and Not Found (404) are handled in the minimal endpoint. It does not mean it will only return these two errors. The code can still return unhandled errors, such as Internal Error (500) or Request Timeout (408). In the next chapter, I will return to unhandled errors and show you how to "catch" them. Listing 4-58 shows the *Produces* extension method applied on the endpoint for each possible status mentioned.

Listing 4-58. Adding the Produces extension method on the GET /countries/download endpoint

```
app.MapGet("/countries/download", (ICountryService
countryService) =>
{
    (byte[] fileContent, string mimeType, string fileName) =
    countryService.GetFile();

    if (fileContent is null || mimeType is null)
        return Results.NotFound();

    return Results.File(fileContent, mimeType, fileName);
})
.Produces<Stream>(StatusCodes.Status200OK, "video/mp4")
.Produces(StatusCodes.Status404NotFound)
.Produces(StatusCodes.Status500InternalServerError)
.Produces(StatusCodes.Status408RequestTimeout);
```

The output in Swagger is shown in Figure 4-66.
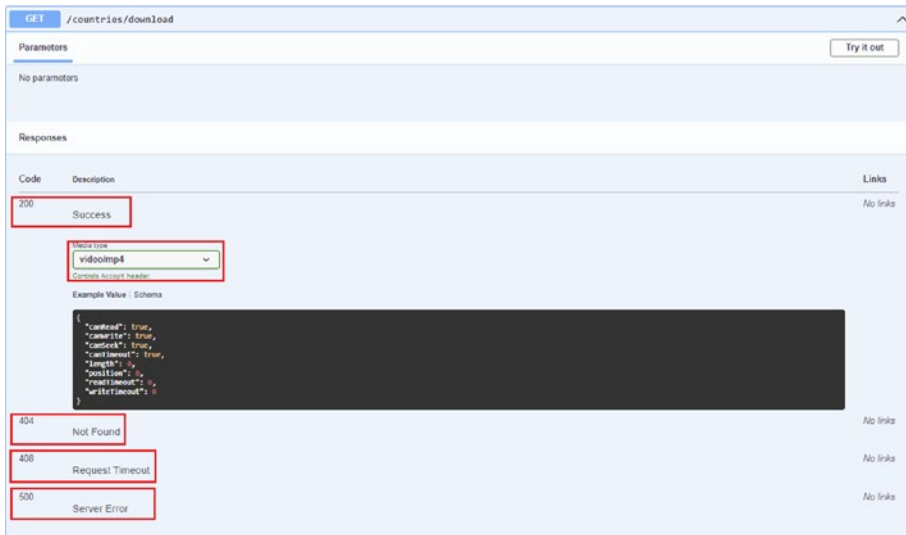
Figure 4-66. *The output in Swagger when adding endpoint response description*

Summary

This chapter taught you the basics of clean REST APIs and ASP.NET Core 8 minimal APIs. I know this chapter was huge, but it was the strict minimum knowledge needed to learn to develop clean REST APIs with ASP.NET Core 8 minimal APIs. Since it's the strict minimum, I strongly suggest you move forward with the next chapter to learn more about ASP.NET Core 8 minimal APIs and REST APIs in general. The next chapter will introduce more advanced features you may need to develop in your career. Even though they won't apply to every API you will code, you face these challenges often. Let's go to Chapter 5!

CHAPTER 5

Going Further with Clean REST APIs

Congratulations, reader! You made it through Chapter 4, which was packed with content! I showed you all the basics you could implement in an API in your career. In this new chapter, we'll take you a step further with ASP. NET Core 8. Everything we've covered in Chapter 4 is still valid. Still, I suggest you enhance your API with features that will enable you to make your API more structured, more elegant, and easier to evolve, offering your customers a better user experience and making them safer. In this chapter, I'm going to teach you the following:

- Encapsulating minimal endpoint implementation

- Implementing custom parameter binding

- Using middlewares

- Using Action Filters

- Using Rate Limiting

- Global error management

Encapsulating Minimal Endpoint Implementation

One thing I always do first in a minimal API (or any other application) is to structure my code. So far, I've shown you lambda functions that were directly executed in a minimal endpoint using the *MapGet*, *MapPost*, and other functions.

We will create static functions in static classes to make our code more structured. How do we do this? The code will be much more readable, and we'll separate the API-coupled code (*MapXXX* functions) from the execution of our logical business. Once again, we return to the SoC principle I've already explained. We will take advantage of this decoupling to make our software more testable. I'll return to unit (and integration) testing in the last chapter of this book. To begin with, let's go back to the POST /countries endpoint, which was used to create a country, with all its dependencies, as shown in Listing 5-1.

Listing 5-1. Recap of the POST /countries endpoint

```
app.MapPost("/countries", (
[FromBody] Country country,
IValidator<Country> validator,
ICountryMapper mapper,
ICountryService countryService) => {
    var validationResult = validator.Validate(country);

    if (validationResult.IsValid)
    {
        var countryDto = mapper.Map(country);
        return Results.CreatedAtRoute(
                "countryById",
                new {
```

```
                Id = countryService.CreateOrUpdate(countryDto)
                });
    }
    return Results.ValidationProblem(
            validationResult.ToDictionary()
    );
});
```

What I like to do instead is to create an "Endpoints" directory and then a static class named *CountryEndpoints* in the API project, similar to the structure of *Controllers* in a web API, but at least the code is tidy!

There's no need to move the code to another layer because, whether we like it or not, we're still somewhat coupled to the Web here, as we have a *FromBody* attribute that depends on the *Microsoft.AspNetCore.Mvc* assembly. Figure 5-1 shows the structure of the API project as described.
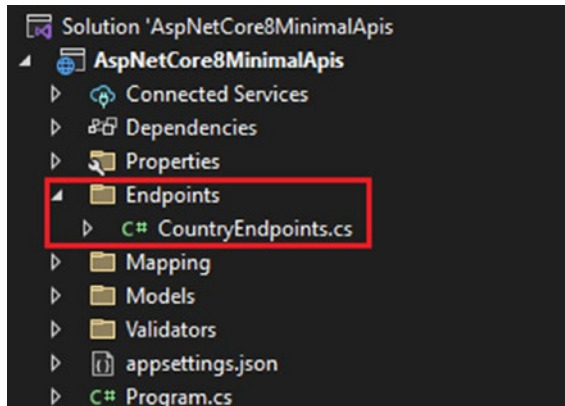
Figure 5-1. *API structure with the creation of a folder dedicated to endpoint functions*

After creating a static method named *PostCountry,* which is the result of the concatenation of the HTTP verb and the route (it's a convention I like to use) and takes as parameters the same parameters as the previous lambda, it gives the code as shown in Listing 5-2.

Listing 5-2. The CountryEndpoints class and its method
PostCountry

```
using AspNetCore8MinimalApis.Mapping.Interfaces;
using AspNetCore8MinimalApis.Models;
using Domain.Services;
using FluentValidation;
using Microsoft.AspNetCore.Mvc;
namespace AspNetCore8MinimalApis.Endpoints;

public static class CountryEndpoints
{
    public static IResult PostCountry(
    [FromBody] Country country,
    IValidator<Country> validator,
    ICountryMapper mapper,
    ICountryService countryService)
    {
        var validationResult = validator.Validate(country);

        if (validationResult.IsValid)
        {
            var countryDto = mapper.Map(country);
            return Results.CreatedAtRoute(
                    "countryById",
                    new {
                        Id = countryService.CreateOrUpdate
                        (countryDto)
                    });
        }
        return Results.ValidationProblem(
                validationResult.ToDictionary()
```

```
        );
    }
}
```

Listing 5-3 shows the *Program.cs* file updated with the static function instead of any lambda on endpoints.

Listing 5-3. The Program.cs file updated with the static method instead of lambdas

```
using AspNetCore8MinimalApis.Endpoints;
using AspNetCore8MinimalApis.Mapping;
using AspNetCore8MinimalApis.Mapping.Interfaces;
using BLL.Services;
using Domain.Services;
using FluentValidation;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddValidatorsFromAssemblyContaining<P
rogram>();
builder.Services.AddScoped<ICountryMapper, CountryMapper>();
builder.Services.AddScoped<ICountryService, CountryService>();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1.0",
                new OpenApiInfo {
                    Title = "ASP.NET Core 8 minimal APIs"
                });
});

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

```
app.UseSwagger().UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1.0/swagger.json",
                      "Version 1.0");
});
```

app.MapPost("/countries", CountryEndpoints.PostCountry);

```
app.Run();
```

It is much cleaner, right? What if I tell you, you can make it more cleaner? If you want, you can mix the power of route groups and the usage of static functions; what would it give? Listing 5-4 shows the *CountryGroup* static class route group that allows you to register all endpoints of the country group using the static functions.

Listing 5-4. The CountryGroup static class

```
using AspNetCore8MinimalApis.Endpoints;

namespace AspNetCore8MinimalApis.RouteGroups;

public static class CountryGroup
{
    public static void AddCountryEndpoints(
    this WebApplication app)
    {
        var group = app.MapGroup("/countries");

        group.MapPost("/", CountryEndpoints.PostCountry);
        // Other endpoints in the same group
    }
}
```

You may have noticed that I've created an *AddCountryEndpoints* extension method, which extends the *WebApplication* class. This extension simplifies the registration of endpoints in the *Program.cs* file, as shown in Listing 5-5.

Listing 5-5. The Program.cs file again more simplified

```
... Code
app.AddCountryEndpoints();
app.Run();
```

Next, you can extend this logic to the rest of your API by creating a class by purpose (a new class for a route group and another for static functions), and your API structure will look great!

Implementing Custom Parameter Binding

In Chapter 4, I told you about parameter binding and said I'd return to the topic. The reason I'm coming back to the subject is that, in fact, more often than you think, you'll have to deal with strange cases, strange because your customers will ask you for things that are out of the ordinary. Yes, your customers will sometimes ask you to implement endpoints for them in a certain way because they'll send you strangely formatted data. Don't ask yourself too many questions, though: more often than not, they're maintaining ancient legacy systems and manipulating data in a way that's no longer done. To counter this, I suggest you implement custom parameter binding that accepts your customers' data formats and transforms them into usable data.

ASP.NET Core 8 offers two types of custom parameter binding:

1. Data from headers, query strings, or routes

2. Data from the body (and form data)

And I will show you both of them.

Example of Custom Parameter Binding from Headers

Let's consider that a customer wants to send you (via an HTTP GET request) a list of country identifiers through headers concatenated by a dash like this: *1-2-3*. We'll handle this by creating a *CountryIds* class containing a *List<int>* Ids property that we'll try to bind from the string of IDs concatenated in the headers. For parameter binding to work, we'll implement a static *TryParse* method that ASP.NET Core will automatically recognize and execute. I've created a *CountryIds* class precisely to implement this *TryParse* method, as it wouldn't have worked otherwise. Listing 5-6 shows the conversion of the string passed in the headers into an int list.

Listing 5-6. The CountryIds class

```
namespace AspNetCore8MinimalApis.Models;

public class CountryIds
{
    public List<int> Ids { get; set; }

    public static bool TryParse(
    string? value,
    IFormatProvider? provider,
    out CountryIds countryIds)
    {
```

```
        countryIds = new CountryIds();
        countryIds.Ids = new List<int>();
        try
        {
            if (value is not null && value.Contains("-"))
            {
                countryIds.Ids = value.Split('-').Select(int.
                Parse).ToList();
                return true;
            }
            return false;
        }
        catch
        {
            return false;
        }
    }
}
```

Then consider the GET /countries/ids endpoint, which takes as input parameter the CountryIds class as shown in Listing 5-7.

Listing 5-7. The GET /countries/ids endpoint with the CountryIds class as parameter

```
app.MapGet("/countries/ids", ([FromHeader] CountryIds ids) =>
{
     Results.NoContent();
});
```

Because the *TryParse* method has no idea where the data comes from, I had to decorate the *CountryIds* class with the *FromHeader* attribute. Let's execute it. Figure 5-2 shows the Postman request.
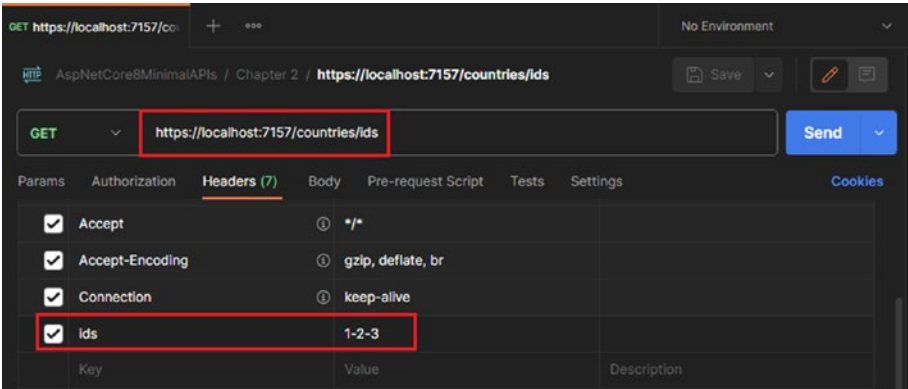
Figure 5-2. *The GET /countries/ids request in Postman*

If we add a breakpoint and take a look into the post-binding operation, we should see a list of integers correctly bound, as shown in Figure 5-3.
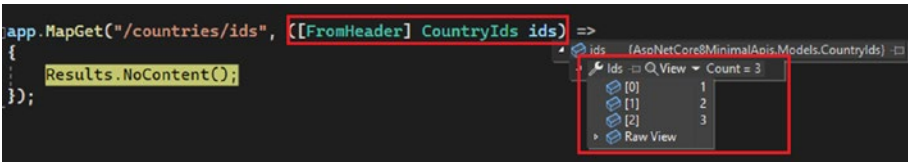
Figure 5-3. *The GET /countries/ids endpoint execution*

As you can see, it's straightforward and efficient! I hope you won't have to do this too often ☺.

Example of Custom Parameter Binding from the From Data

Custom parameter binding from body elements (and form data) works slightly differently. Here we'll apply the same principle as before, but we won't be using the *TryParse* method but rather the static *BindAsync* method, which takes the HTTP context as a parameter. Having the HTTP context here will enable us to fetch form data elements directly from it,

allowing us to omit attributes like *FromForm* and *FromBody* on input parameters. Let's imagine that your client wants to upload a file and pass metadata, but as you'd like, that is, in the form data, they'll pass you the file as expected, but instead of passing you the properties of your input parameter, for example, a *Country* object (the same as in Chapter 4) in JSON format in a single property named *Country* with the following value *{"Id" : 1, "Description" : "Canada", "FlagUri" : "", "Name" : ""}.*

Consider the *Country* class, which implements the *BindAsync* method by fetching data from the form data and deserializing it into JSON format using the *System.Text.Json* API, as shown in Listing 5-8.

Listing 5-8. The Country class

```
using System.Reflection;
using System.Text.Json;

namespace AspNetCore8MinimalApis.Models;

public class Country
{
    /// <summary>
    /// The country Id
    /// </summary>
    public int? Id { get; set; }

    /// <summary>
    /// The country name
    /// </summary>
    public string Name { get; set; }

    /// <summary>
    /// The country description
    /// </summary>
    public string Description { get; set; }
```

```
/// <summary>
/// The country flag URI
/// </summary>
public string FlagUri { get; set; }

public static ValueTask<Country> BindAsync(HttpContext
context, ParameterInfo parameter)
{
    var countryFromValue = context.Request.Form["Country"];
    var result = JsonSerializer.Deserialize<Country>
    (countryFromValue);

    return ValueTask.FromResult(result);
}
}
```

If we take a look at the POST /countries/upload we used previously, but with separated input parameters (they are not encapsulated in a single object) in the minimal endpoint, you can see there is no attribute on any input parameters as shown in Listing 5-9.

Listing 5-9. The POST /countries/upload endpoint

```
app.MapPost("/countries/upload", (
IFormFile file,
Country country) =>
{
    Results.NoContent();
});
```

The Postman request is shown in Figure 5-4.
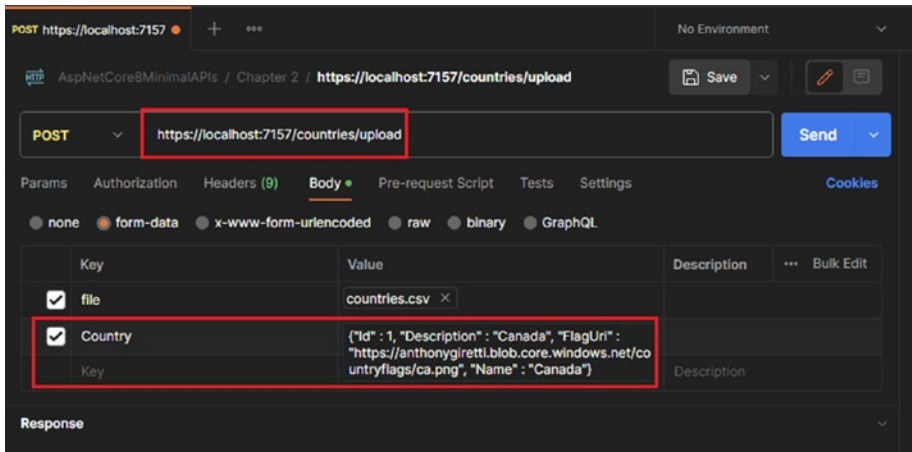
Figure 5-4. *The POST /countries/upload request in Postman*

If we debug the POST /countries/upload endpoint, you will notice that the custom binding worked as expected, as shown in Figure 5-5.
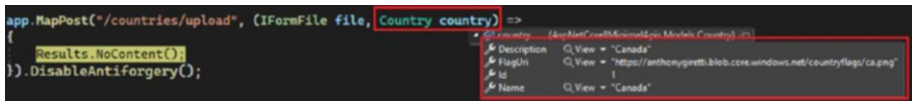
Figure 5-5. *The POST /countries/upload endpoint execution*

It's really easy! Keep this technique in mind if you're forced to manipulate data sent by your customers in unconventional ways!

Using Middlewares

In Chapter 2, in section "ASP.NET Core Fundamentals," I introduced you to middlewares. We will dig deeper here. I will teach you how to implement your middlewares. There are various kinds of middlewares, some of which you're already familiar with, such as *MapGet*, *UseSwagger*, *Run* (yes, that's a middleware), and so on.

And then there are those whose behavior you can define yourself. There are several types of customizable middlewares:

- Map

- MapWhen

- Run

- Use

- UseWhen

- UseMiddleware<T>

They can be divided into three categories:

1. **Middlewares that create another branch of middlewares by short-circuiting the execution of the main pipeline, such as *Map* or *MapWhen***: These can be nested ad infinitum if you wish.

2. **Middlewares that launch a continuously running process that's hosts the application**: It's the responsibility of the *Run* middleware. Executing the *Run* middleware is mandatory; otherwise, it will lead to an application crash.

3. **Middlewares that run on the current pipeline branch (the main or a new branch) and don't create a new one**: A pipeline branch is a flow of consecutive middlewares that will run. They also don't stop pipeline execution, and they are three: *Use, UseWhen,* and *UseMiddleware<T>*. *UseMiddleware<T>* is similar to *Use,* the only difference being that *Use* allows you to run inline code, whereas *UseMiddleware<T>* allows you to run code in a separate class. The generic type *T* is

the class that implements the middleware logic. All
of them **must** execute the *next* function to keep the
current pipeline running.

You can also mix different types of middlewares. The only requirement
is to use *Run* to terminate the execution of the pipeline. ASP.NET Core
automatically adds it at the end of the *Program.cs* file, but you can define
it elsewhere, for example, to mark the end of another pipeline branch
initiated with *Map* or *MapWhen*. **Any *Usexxx* middleware will run before
any *Mapxxx* endpoint, whatever their order.**

Tip   You can nest middlewares into the *UseWhen* middleware,
and it won't initiate a new pipeline branch, BUT, if you add a *Run*
middleware on it, the main pipeline will be short-circuited, and the
main pipeline branch won't run.

Caution   Don't be confused! Middlewares such as *MapGet*,
MapPost, etc. won't initiate a new pipeline branch as the *Map*
Middleware. **They are different.**

Finally, middlewares whose name ends with "When" have the same
function as their counterpart without the "When," the only difference
being that they execute conditionally: if the condition you define is met,
the middleware will execute; else, not. The *Map* middleware is mapped to
a specific route (takes a route as the first parameter). If the route matches,
the *Map* middleware will initiate a new pipeline branch.

Figure 5-6 summarizes middleware behavior.
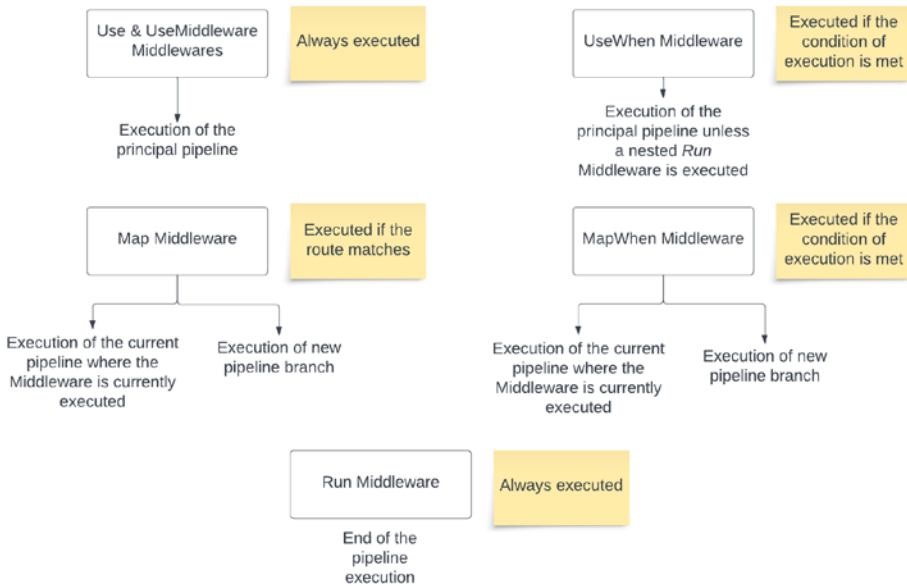
Figure 5-6. *Middleware behavior*

To illustrate this, I will show you some examples. Let's consider the GET /test endpoint being executed; the *MapWhen* middleware will execute if the request path contains the parameter *q* in the query string. The latter contains another branch of middlewares (*Use* and *Run*). Since the *MapWhen* middleware initiates a new pipeline branch when executed, the position of the GET /test endpoint in the pipeline doesn't matter. Listing 5-10 shows the code described in this scenario.

Listing 5-10. The GET /test endpoint positioned before the MapWhen middleware

```
app.MapGet("/test", () =>
{
    return Results.Ok("Test endpoint has been executed");
});
```

```
app.MapWhen(ctx => !string.IsNullOrEmpty(ctx.Request.
Query["q"].ToString()),
builder => {
    builder.Use(async (context, next) =>
    {
        app.Logger.LogInformation("New middleware pipeline has
        been invoked");
        await next();
    });
    builder.Run(async context =>
    {
        app.Logger.LogInformation("New pipeline initiated will
        end here");
        await Task.CompletedTask;
    });
});// Stops the execution if the condition is met because the
new branch contains Run Middleware

app.Run();
```
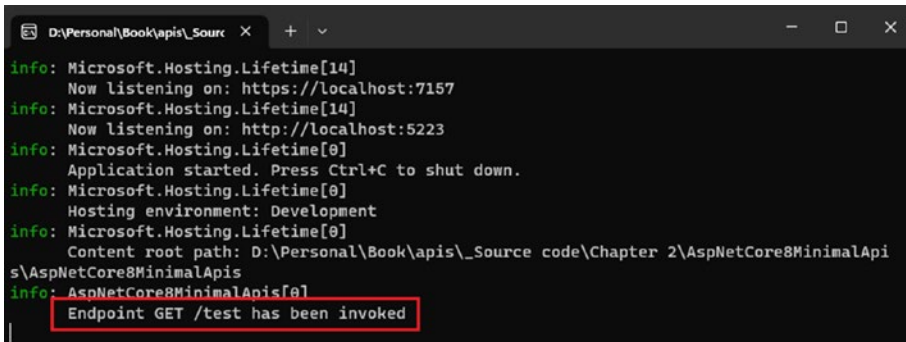
Note   I'm using logging (in the console) for my upcoming tests in this section. I will explain logging in Chapter 8.

Let's test it! Let's invoke the route GET /test without the parameter *q* in the query string, as shown in Figure 5-7.
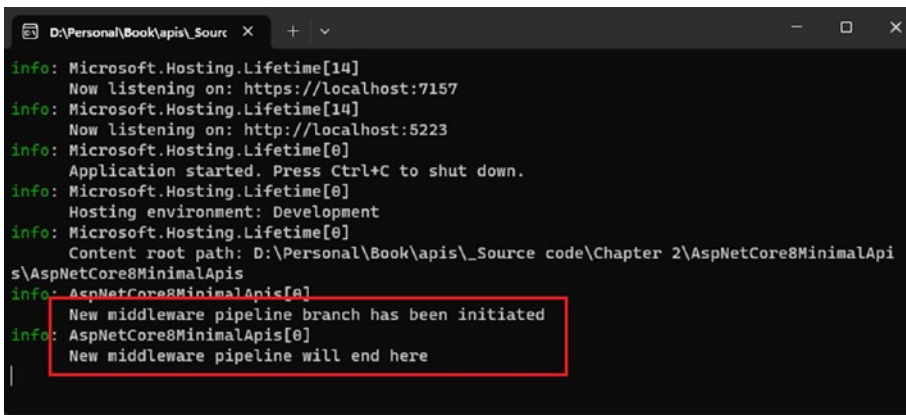
Figure 5-7. *The GET /test endpoint invoked without the query string*
parameter q

The expected behavior was that the main pipeline got executed
without initiating a new branch since the *MapWhen* middleware hadn't
met the condition to get executed.

If we had in the query string the *q* parameter, the *MapWhen*
middleware would be executed and initiate another branch of the pipeline.
Since the *MapGet(/test)* middleware is declared in the main pipeline, the
pipeline should not be executed, as shown in Figure 5-8.

Figure 5-8. *The GET /test endpoint invoked with the query string*
parameter q

When matching the route /test (whatever the verb), the *Map* middleware will lead to the same behavior as the *MapWhen* middleware seen earlier. Listing 5-11 shows the code of the *Map* middleware matching the /test route.

Listing 5-11. The Map middleware

```
app.Map(new PathString("/test"),
builder =>
{
    builder.Use(async (context, next) =>
    {
        app.Logger.LogInformation("New middleware pipeline
        branch has been initiated");
        await next();
    });
    builder.Run(async context =>
    {
        app.Logger.LogInformation("New middleware pipeline will
        end here");
        await Task.CompletedTask;
    });
});// Stops execution
```

Now let's see how *Use* and *UseWhen* middlewares behave. Listing 5-12 shows a *Use* middleware executed before the GET /test endpoint, another one executed after, and a *UseWhen* middleware at the end before the final *Run* middleware.

Listing 5-12. The GET /test endpoint positioned between the Use middlewares and before the UseWhen middleware

```
app.Use(async (context, next) =>
{
    app.Logger.LogInformation("Middleware 1 executed");
    await next();
});

app.MapGet("/test", () =>
{
    app.Logger.LogInformation("Endpoint GET /test has been
    invoked");
    return Results.Ok();
});

app.Use(async (context, next) =>
{
    app.Logger.LogInformation("Middleware 2 executed");
    await next();
});

app.UseWhen(ctx => !string.IsNullOrEmpty(ctx.Request.
Query["p"].ToString()),
builder => {
    builder.Use(async (context, next) =>
    {
        app.Logger.LogInformation("Nested middleware
        executed");
        await next();
    });

    builder.Run(async (context) =>
    {
```

```
        app.Logger.LogInformation("End of the pipeline end");
        await Task.CompletedTask;
    });

});
// Stops the execution if the condition is met because the
UseWhen contains Run Middleware

app.Run(); // Final
```

Two scenarios are possible if we execute this code. The first scenario is that the *UseWhen* is not running if we don't pass any *p* parameter in the query string when we call the GET /test endpoint; in this case, the pipeline won't end at the *Run* middleware declared in the *UseWhen* middleware. The second scenario is passing a *p* parameter in the query string; consequently, the pipeline will end because of the *Run* middleware execution in the *UseWhen* middleware. In any case, the *Run* middleware, whatever its position in the pipeline, will execute. Figure 5-9 shows the GET /test endpoint invoked without the *p* query string parameter.

Figure 5-9. *The GET /test endpoint invoked without the p query string parameter*

As you can see, the two *Use* middlewares ran in order before the execution of the *MapGet* middleware. The *UseWhen* hasn't been executed as expected.

If we pass the *p* parameter in the query string, it should give that shown in Figure 5-10.

Figure 5-10. *The GET /test endpoint not invoked because the p query string parameter triggered the UseWhen middleware execution, which ran a nested Run middleware*

Since all *Usexxx* middlewares ran before any *MapXXX* middleware and the *UseWhen* ran a nested *Run* middleware, the GET /test endpoint has not been invoked. If we remove the nested *Run* middleware, the GET /test should run.

Listing 5-13 shows the UseWhen middleware without the nested Run middleware.

Listing 5-13. The UseWhen middleware without its nested Run middleware

```
app.UseWhen(ctx => !string.IsNullOrEmpty(ctx.Request.
Query["p"].ToString()),
builder => {
    builder.Use(async (context, next) =>
    {
        app.Logger.LogInformation("Nested middleware
        executed");
        await next();
    });
});
```

Figure 5-11 shows the pipeline execution.

Figure 5-11. *The GET /test endpoint invoked with the p query string parameter with the UseWhen middleware, without the nested Run middleware*

Let's see together a last example with the *UseMiddleware* middleware. As I taught you earlier, it behaves like a *Use* middleware, but its code is encapsulated in a separate class. I like to use this way of coding middlewares since it appears **cleaner** to me. Let's code the same functionality as the *Use* middleware we saw earlier (which performs only logging) and create a *LoggingMiddleware* class as shown in Listing 5-14.

Listing 5-14. The LoggingMiddleware class

```
namespace AspNetCore8MinimalApis.Middlewares;

public class LoggingMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<LoggingMiddleware> _logger;

    public LoggingMiddleware(RequestDelegate next,
    ILogger<LoggingMiddleware> logger)
    {
        _next = next;
        _logger = logger;
    }
    public async Task Invoke(HttpContext context)
    {
        _logger.LogInformation("LoggingMiddleware executed");

        await _next(context);
    }
}
```

Implementing a middleware like this obliges you to implement the *Invoke* method, where you can put your logic and execute the *next* function, a *delegate* that represents the next middleware to get executed in the ASP.NET Core pipeline.

236

Let's replace the *Use* middleware with the *UseMiddleware<LoggingMiddleware>* middleware as shown in Listing 5-15.

Listing 5-15. The LoggingMiddleware registered in the pipeline with the UseMiddleware middleware

```
app.Use(async (context, next) =>
{
    app.Logger.LogInformation("Middleware 1 executed");
    await next();
});// Don't stop the execution

app.MapGet("/test", () =>
{
    app.Logger.LogInformation("Endpoint GET /test has been
    invoked");
    return Results.Ok();
});
```

app.UseMiddleware<LoggingMiddleware>();// Doesn't stop the execution

```
app.Run(); // Final
```

If we run it, and since *UseMiddleware* has the same behavior as the *Use* middleware, we should expect the same order of execution as seen before when no *p* parameter was passed in the query string, as shown in Figure 5-12.

Figure 5-12. *The GET /test endpoint invoked after the Use and UseMiddleware<LoginMiddleware> middlewares*

As expected, the *Use* and *UseMiddleware<LoginMiddleware>* middlewares are executed before the *MapGet* middleware.

Middlewares are a powerful feature of ASP.NET Core 8. They allow you incomparable flexibility in terms of code execution. I tried to keep the examples simple by using the logging feature (I promise we will come back to this later), but middlewares don't stop with the logging feature. Even if it's the most common scenario that justifies the usage of middlewares, you can implement any code you want to get executed in your pipeline. Later in this book, I will return to this topic when it comes to talking about data collection (metrics) in the application.

Using Action Filters

ASP.NET Core 8 takes endpoint management one step further. For a given endpoint, it is possible to perform any action **before** and **after** its execution. This is useful when, for example, you want to measure an

endpoint's execution time while ignoring the rest of the pipeline, assess its performance when in doubt, or validate an endpoint's input parameters more elegantly than in the previous chapter. I'll show you an example of each scenario in this section.
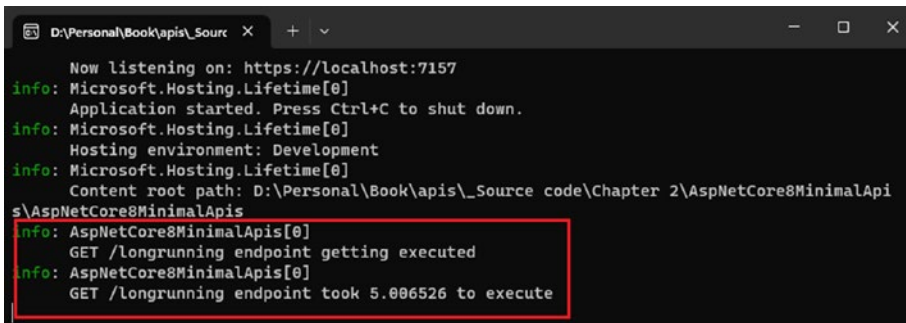
First, implementing an endpoint filter is similar to the *Use* middleware. They both implement the delegate *next*.

Let's take the example of the GET /longrunning endpoint whose execution time we want to measure. We'll apply the *AddEndpointFilter* extension method, which takes a delegate as a parameter, just like middleware Use. We'll simulate an execution time of 5 sec with the *Task. Delay* method and measure the execution time with the *Stopwatch* class, starting the timer before the endpoint execution, represented by the *next* delegate, getting the execution result, stopping the timer, then logging the execution time in the console, and finally returning the response as shown in Listing 5-16.

Listing 5-16. The GET /longrunning endpoint attached with an endpoint filter measuring its execution time

```
app.MapGet("/longrunning", async () =>
{
    await Task.Delay(5000);
    return Results.Ok();
}).AddEndpointFilter(async (filterContext, next) =>
{
    long startTime = Stopwatch.GetTimestamp();
    var result = await next(filterContext);
    TimeSpan elapsedTime = Stopwatch.GetElapsedTime(startTime);
    app.Logger.LogInformation($"GET /longrunning endpoint took
    {elapsedTime.TotalSeconds} to execute");
    return result;
});
```

Figure 5-13 shows what the console output looks like.

Figure 5-13. *The GET /longrunning endpoint output after execution with an endpoint filter measuring its execution time*

It works like a charm, as you can see. I suggest encapsulating endpoint filters into a separate class, like middlewares, for cleaner code. What you have to do is to inherit your class from the *IEndpointFilter* interface. Listing 5-17 shows the *LogPerformanceFilter* class, which implements the *InvokeAsync* method, which does the same job as the inline I showed you previously.

Listing 5-17. The LogPerformanceFilter class

```
using System.Diagnostics;

namespace AspNetCore8MinimalApis.EndpointFilters;

public class LogPerformanceFilter : IEndpointFilter
{
    private readonly ILogger<LogPerformanceFilter> _logger;
    public LogPerformanceFilter(ILogger<LogPerformance
    Filter> logger)
    {
        _logger = logger;
    }
```

```
public async ValueTask<object?> InvokeAsync(EndpointFilterI
nvocationContext context, EndpointFilterDelegate next)
{
    _logger.LogInformation($"GET /longrunning endpoint
    getting executed");
    long startTime = Stopwatch.GetTimestamp();
    var result = await next(context);
    TimeSpan elapsedTime = Stopwatch.
    GetElapsedTime(startTime);
    _logger.LogInformation($"GET /longrunning endpoint took
    {elapsedTime.TotalSeconds} to execute");
    return result;
}
}
```

Then let's attach to the *AddEndpointFilter<T>* method overload as shown in Listing 5-18.

Listing 5-18. The GET /longrunning endpoint attached with the LogPerformanceFilter measuring its execution time

```
app.MapGet("/longrunning", async () =>
{
    await Task.Delay(5000);
    return Results.Ok();
}).AddEndpointFilter<LogPerformanceFilter>();
```

Much cleaner, isn't it?

Let's go further with a very convenient usage of endpoint filters. Let's combine the power of FluentValidation, as I introduced in the previous chapter, with endpoint filters. Let's write a generic endpoint filter, the *InputValidatorFilter<T>* class, that validates endpoint input parameters as shown in Listing 5-19.

Listing 5-19. The InputValidatorFilter<T> class

```
using FluentValidation;

namespace AspNetCore8MinimalApis.EndpointFilters;

public class InputValidatorFilter<T> : IEndpointFilter
{
    private readonly IValidator<T> _validator;
    public InputValidatorFilter(IValidator<T> validator)
    {
        _validator = validator;
    }

    public async ValueTask<object?> InvokeAsync(
    EndpointFilterInvocationContext
    context,        EndpointFilterDelegate next)
    {
        T? inputData = context.GetArgument<T>(0);

        if (inputData is not null)
        {
            var validationResult = await _validator.
            ValidateAsync(inputData);
            if (!validationResult.IsValid)
            {
                return Results.ValidationProblem(
                  validationResult.ToDictionary()
                );
            }
        }

        return await next.Invoke(context);
    }
}
```

Let's take the POST /countries endpoint from Chapter 4 and let's apply it to the *InputValidatorFilter<T>* endpoint filter, which will take here as a generic parameter the *Country* class as shown in Listing 5-20.

Listing 5-20. The POST /countries endpoint attached with the Inpu tValidatorFilter<Country>

```
app.MapPost("/countries", ([FromBody] Country country) => {

    return Results.CreatedAtRoute("countryById", new {
    Id = 1 });
}).AddEndpointFilter<InputValidatorFilter<Country>>();
```

Obviously, the behavior remains the same as when we passed the *IValidator* by dependency injection in the endpoint. Once again, it's simply cleaner than coding the validation inline. I hope you will use this feature as much as you can; I really enjoy it on my end. I'm pretty sure you will find more usage scenarios on your own!

Using Rate Limiting

Here's an interesting feature of ASP.NET Core 8: Rate Limiting. As the name suggests, this feature lets you limit access to your API for obvious reasons:

> **Protect the system**: Rate Limiting helps prevent *Denial of Service (DOS)* attacks by limiting the number of requests a user or application can send.

> **Guarantee quality of service**: By limiting throughput, Rate Limiting ensures fair service quality for all users and applications and helps keep good performance by limiting access to resources.

>   **Provide different accesses to your customers with
>   a pricing tier**: For example, free but limited access
>   or a paid subscription with no limits.

ASP.NET Core 8 offers four categories of limiters:

1. **Fixed window:** Limiting is based on two
    parameters: the number of authorized requests
    and a time window during which said number of
    requests is authorized. Each authorized request
    decreases the authorized request counter, and each
    time a time window elapses, the authorized request
    counter is reset. This model allows a certain number
    of requests to be queued (until the counter is reset)
    before the others are rejected.

2. **Sliding window**: Similar to the **Fixed window**,
    but works differently. The idea here is to divide a
    time window into segments during which a certain
    number of requests are authorized. At the end
    of each period during which a segment handles
    requests, the remaining number of authorized
    requests is transferred to the next segment within
    the maximum limit defined at the start. Once the
    global time window has elapsed, the number of
    authorized requests is reallocated to the maximum
    number of requests authorized minus the number
    of requests authorized by the previous segment.
    This model allows a certain number of requests
    to be queued (until the selection segment has
    been reallocated a certain number of authorized
    requests) before the others are rejected.

3. **Token bucket:** This is similar to **Fixed window**, except the notion of token and bucket is introduced. This means that we define the number of tokens in a bucket, that is, each time a request is authorized, one token will be used, thus reducing the number of tokens in the bucket, except that this model allows you to define a maximum number of tokens available and that a number of tokens will be reintroduced into the bucket during a specific period. The number of tokens reintroduced into the bucket cannot exceed the maximum number of available tokens. This model offers regular, controlled, but limited access to your application. This model doesn't allow you to queue requests waiting for available tokens, as the rejection of a request in the absence of an available token will be automatic.

4. **Concurrency:** This is the simplest model and refers to the number of simultaneous requests allowed. This model allows a certain number of requests to be queued (until the number of simultaneously authorized executed requests has been exceeded) before the others are rejected.

Each of these models allows you to define (or not) a partition key, that is, a criterion on which to limit requests. This can be a user ID, an IP address, or anything you like. The limitation will be global if you don't define a partition key. When a request is rejected, ASP.NET Core 8 returns the error Service Unavailable (503), which is incorrect. Fortunately, ASP. NET Core 8 allows you to set the status code and to be HTTP compliant; I suggest you use Too Many Requests (429) instead.

To define a Rate Limiting rule, you have to use the *AddRateLimiter* extension method, and to enable it, use the *UseRateLimiter* middleware as shown in Listing 5-21.

Listing 5-21. The AddRateLimiter extension method and the UseRateLimiter middleware

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

builder.Services.AddRateLimiter(options =>
{
    // Code here
});

var app = builder.Build();

// Your ASP.NET Core pipeline

app.UseRateLimiter();

// Your ASP.NET Core pipeline

app.Run();
```

The Fixed Window Model

Listing 5-22 shows an implementation of the Fixed window limiter, which defines a limit of 50 requests during a window of 15 seconds. If the limit is reached, ten requests will be queued, and the others will be rejected. The partition key is the client's IP address. It returns a Too Many Request (429) and custom error messages.

Listing 5-22. The Fixed window limiter implementation

```
builder.Services.AddRateLimiter(options =>
{
    options.RejectionStatusCode = (int)HttpStatusCode.
    TooManyRequests;
    options.OnRejected = async (context, token) =>
    {
        await context.HttpContext.Response.WriteAsync("Too many
        requests. Please try again later.");
    };
      options.GlobalLimiter = PartitionedRateLimiter.
      Create<HttpContext, string>(httpContext =>
        RateLimitPartition.GetFixedWindowLimiter(
            partitionKey: httpContext.Connection.
            RemoteIpAddress.ToString(),
            factory: _ => new FixedWindowRateLimiterOptions
            {
                QueueLimit = 10,
                PermitLimit = 50,
                Window = TimeSpan.FromSeconds(15)
            }));
});
```

The function *PartitionedRateLimiter.Create* allows you to bring the *HttpContext* object that allows you to get any contextual information from the client, their IP address in this example, but it could be their userId if they are authenticated. (I will return to authentication in Chapter 10.)

The rate limit applies globally on any endpoint because I have assigned the limiter to the *GlobalLimiter* options. If you don't want the rate limiter getting applied on an endpoint, you have to use the *DisableRateLimiting* extension method as shown in Listing 5-23.

Listing 5-23. Disabling the global Rate Limiting feature with the DisableRateLimiting extension method

```
app.MapGet("/notlimited", () =>
{
    return Results.Ok();
}).DisableRateLimiting();
```

Another interesting thing is that you can create as many limiters as you wish, and they can be identified by a policy, which must be explicitly applied to the endpoint you want. Listing 5-24 shows the *ShortLimit* policy combined with the global rate limiter we saw previously.

Listing 5-24. The global rate limiter and the ShortLimit policy combined

```
builder.Services.AddRateLimiter(options =>
{
    options.RejectionStatusCode = (int)HttpStatusCode.
    TooManyRequests;
    options.OnRejected = async (context, token) =>
    {
        await context.HttpContext.Response.WriteAsync("Too many
        requests. Please try again later.");
    };
    options.GlobalLimiter = PartitionedRateLimiter.
    Create<HttpContext, string>(httpContext =>
    RateLimitPartition.GetFixedWindowLimiter(
        partitionKey: httpContext.Connection.RemoteIpAddress.
        ToString(),
        factory: _ => new FixedWindowRateLimiterOptions
        {
            QueueLimit = 10,
```

```
        PermitLimit = 50,
        Window = TimeSpan.FromSeconds(15)
    }));

    options.AddPolicy(policyName: "ShortLimit", context =>
        {
            return     RateLimitPartition.
            GetFixedWindowLimiter(context.Connection.
            RemoteIpAddress.ToString(),
             _ => new FixedWindowRateLimiterOptions
            {
                PermitLimit = 10,
                Window = TimeSpan.FromSeconds(15)
            });
        });
});
```

If you want to apply it on one or many endpoints, add the *RequireRateLimiting* extension method taking the policy name as a parameter, as shown in Listing 5-25.

Listing 5-25. The RequireRateLimiting extension method

```
app.MapGet("/limited", () =>
{
    return Results.Ok();
}).RequireRateLimiting("ShortLimit");
```

Caution   If you define a global limiter and a specific limiter defined by a policy, **both** will execute if you tell your endpoints they must execute the policy you assigned them with the *RequireRateLimiting* extension method. The global limiter will execute **first**.

It's flexible, as you can see, and it can be more flexible. Let's implement a pricing tier and create limiter rules depending on the pricing tier. Let's say we create a service that returns the pricing tier depending on the client's IP address (let's assume that the pricing tier is bound to the IP address), as shown in Listing 5-26.

Listing 5-26. The IPricingTierService service

```
using Domain.Enum;

namespace Domain.Services;

public interface IPricingTierService
{
    public PricingTier GetPricingTier(string ipAddress);
}
```

The *IPricingTierService* returns a *PricingTier* enum as shown in Listing 5-27.

Listing 5-27. The PricingTier enum

```
namespace Domain.Enum;

public enum PricingTier
{
    Free = 0,
    Paid = 1
}
```

Register the *IPricingTierService* service with its *PricingTierService* implementation (the implementation does not matter here) as follows: `builder.Services.AddScoped<IPricingTierService, PricingTierService>();`.

Since the **HttpContext** is exposed, we can access any registered service through the dependency injection system as follows: `httpContext.RequestServices.GetRequiredService<T>,` where T is the service we want to access.

We can rewrite the global rate limiter as shown in Listing .

Listing 5-28. The global rate limiter updated with the pricing tier

```
builder.Services.AddRateLimiter(options =>
{
    options.RejectionStatusCode = (int)HttpStatusCode.
    TooManyRequests;
    options.OnRejected = async (context, token) =>
    {
        await context.HttpContext.Response.WriteAsync("Too many
        requests. Please try again later.");
    };
    options.GlobalLimiter = PartitionedRateLimiter.
    Create<HttpContext, string>(httpContext =>
    {
        var priceTierService = httpContext.RequestServices.GetR
        equiredService<IPricingTierService>();
        var ip = httpContext.Connection.RemoteIpAddress.
        ToString();
        var priceTier = priceTierService.GetPricingTier(ip);

        return priceTier switch
        {
            PricingTier.Paid => RateLimitPartition.
            GetFixedWindowLimiter(
                ip,
                _ => new FixedWindowRateLimiterOptions
                {
```

```
                QueueLimit = 10,
                PermitLimit = 50,
                Window = TimeSpan.FromSeconds(15)
            }),
        PricingTier.Free => RateLimitPartition.
        GetFixedWindowLimiter(
            ip,
            _ => new FixedWindowRateLimiterOptions
            {
                PermitLimit = 1,
                Window = TimeSpan.FromSeconds(15)
            })
    };
    });
});
```

As you can see, depending on the situation, we can apply any rate limiter to any incoming request.

The Fixed window model is my favorite model; this is the one I use when I define a rate limiter in my applications. I prefer to limit incoming requests during a specific window rather than other models. That's why I have insisted on this model before showing you others. Figure 5-14 shows the HTTP Too Many Requests (429) error in Postman when a rate limiter rule has declined an incoming request.
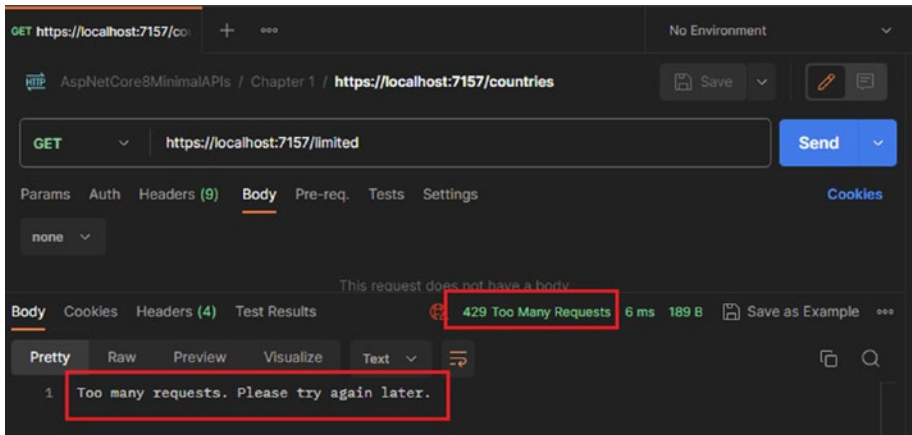
Figure 5-14. *The HTTP Too Many Requests error returned by a rate limiter rule*

The Sliding Window Model

Here we have to define the number of segments (*SegmentsPerWindow* option) that will share the limited number of requests and the window of time. The rest is only about changing classes' and functions' names, as shown in Listing 5-29.

Listing 5-29. The global rate limiter set with the Sliding window model

```
builder.Services.AddRateLimiter(options =>
{
    options.RejectionStatusCode = (int)HttpStatusCode.
    TooManyRequests;
    options.OnRejected = async (context, token) =>
    {
        await context.HttpContext.Response.WriteAsync("Too many
        requests. Please try again later.");
    };
```

253

```
options.GlobalLimiter = PartitionedRateLimiter.
Create<HttpContext, string>(httpContext =>
{
    var priceTierService = httpContext.RequestServices.GetR
    equiredService<IPricingTierService>();
    var ip = httpContext.Connection.RemoteIpAddress.
    ToString();
    var priceTier = priceTierService.GetPricingTier(ip);

    return priceTier switch
    {
        PricingTier.Paid => RateLimitPartition.GetSliding
        WindowLimiter(
            ip,
            _ => new SlidingWindowRateLimiterOptions
            {
                QueueLimit = 10,
                PermitLimit = 50,
                SegmentsPerWindow = 2,
                Window = TimeSpan.FromSeconds(15)
            }),
        PricingTier.Free => RateLimitPartition.GetSliding
        WindowLimiter(
            ip,
            _ => new SlidingWindowRateLimiterOptions
            {
                PermitLimit = 2,
                SegmentsPerWindow = 2,
                Window = TimeSpan.FromSeconds(15)
            })
```

```
        };
    });
});
```

As you can see the *GetSlidingWindowLimiter* function and *SlidingWindowRateLimiterOptions* class took the place their equivalent for the Fixed window model.

The Token Bucket Model

The Token bucket model requires the following options:

1. **TokenLimit**: Defines the maximum available tokens

2. **TokensPerPeriod**: Defines the replenished number of tokens per period

3. **ReplenishmentPeriod**: Period where tokens will get replenished

We can write the limiter as shown in Listing 5-30.

Listing 5-30. The global rate limiter set with the Token bucket model

```
builder.Services.AddRateLimiter(options =>
{
    options.RejectionStatusCode = (int)HttpStatusCode.
    TooManyRequests;
    options.OnRejected = async (context, token) =>
    {
        await context.HttpContext.Response.WriteAsync("Too many
        requests. Please try again later.");
    };
```

```csharp
options.GlobalLimiter = PartitionedRateLimiter.
Create<HttpContext, string>(httpContext =>
{
    var priceTierService = httpContext.RequestServices.GetR
    equiredService<IPricingTierService>();
    var ip = httpContext.Connection.RemoteIpAddress.
    ToString();
    var priceTier = priceTierService.GetPricingTier(ip);

    return priceTier switch
    {
        PricingTier.Paid => RateLimitPartition.GetToken
        BucketLimiter(
            ip,
            _ => new TokenBucketRateLimiterOptions
            {
                TokenLimit = 50,
                TokensPerPeriod = 25,
                ReplenishmentPeriod = TimeSpan.
                FromSeconds(15)
            }),
        PricingTier.Free => RateLimitPartition.GetToken
        BucketLimiter(
            ip,
            _ => new TokenBucketRateLimiterOptions
            {
                TokenLimit = 10,
                TokensPerPeriod = 5,
                ReplenishmentPeriod = TimeSpan.
                FromSeconds(15)
```

```
            })
        };
    });
});
```

The *GetTokenBucketLimiter* function and
TokenBucketRateLimiterOptions class took the place of their equivalent for
the Fixed window model.

Note   The Token bucket model is a bit more aggressive than other
models since it does not allow any requests to get queued if no token
is available.

The Concurrency Model

The Concurrency model is the simplest limiter to configure and only
needs to define the *QueueLimit* and *PermitLimit* options, as shown in
Listing 5-31.

Listing 5-31. The global rate limiter set with the Concurrency model

```
builder.Services.AddRateLimiter(options =>
{
    options.RejectionStatusCode = (int)HttpStatusCode.
    TooManyRequests;
    options.OnRejected = async (context, token) =>
    {
        await context.HttpContext.Response.WriteAsync("Too many
        requests. Please try again later.");
    };
```

```
    options.GlobalLimiter = PartitionedRateLimiter.
    Create<HttpContext, string>(httpContext =>
{

    var priceTierService = httpContext.RequestServices.GetR
    equiredService<IPricingTierService>();
    var ip = httpContext.Connection.RemoteIpAddress.
    ToString();
    var priceTier = priceTierService.GetPricingTier(ip);

    return priceTier switch
    {
        PricingTier.Paid => RateLimitPartition.GetConcurren
        cyLimiter(
            ip,
            _ => new ConcurrencyLimiterOptions
            {
                QueueLimit = 10,
                PermitLimit = 50
            }),
        PricingTier.Free => RateLimitPartition.GetConcurren
        cyLimiter(
            ip,
            _ => new ConcurrencyLimiterOptions
            {
                QueueLimit = 0,
                PermitLimit = 10
            })
    };
});
});
```

The *GetConcurrencyLimiter* function and *ConcurrencyLimiterOptions* class took the place of their equivalent for the Fixed window model.

Rate Limiting is a powerful and customizable feature of ASP.NET Core 8. I strongly suggest you implement it!

Global Error Management

Efficient error handling is essential when developing an application, especially when this application calls on external resources (files, databases, etc.). The role of error handling is to notify the occurrence of an error by explicitly indicating the type of error to the client consuming your API. With ASP.NET Core 8, handling errors globally and cleanly without repeating code is easy. We will rely on the ProblemDetails class I introduced in Chapter 1 to do this. This class allows you to return a correctly formed error to the client. If you remember, it's based on an RFC, so it's a norm, a standardization. As a result, your client will expect to receive errors that are correctly and possibly strongly formatted with the *ProblemDetails* RFC standard.

With ASP.NET Core 8, you must implement a class that inherits the IExceptionHandler interface. This interface signature is shown in Listing 5-32.

Listing 5-32. The IExceptionHandler interface

```
public interface IexceptionHandler
{
    ValueTask<bool> TryHandleAsync(HttpContext httpContext,
    Exception exception, CancellationToken cancellationToken);
}
```

As you can see, it defines a method named *TryHandleAsync,* which returns *ValueTask<bool>.* You must return *True* or *False.* If you return *True,* the pipeline execution will end. If you return *False,* the pipeline will continue its execution. Listing 5-33 shows the *DefaultExceptionHandler* class that handles any exception raised in the application.

Listing 5-33. The DefaultExceptionHandler class

```
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using System.Net;

namespace AspNetCore8MinimalApis.ExceptionHandlers;

public class DefaultExceptionHandler : IExceptionHandler
{
    public async ValueTask<bool> TryHandleAsync(HttpContext
    httpContext, Exception exception, CancellationToken
    cancellationToken)
    {
        await httpContext.Response.WriteAsJsonAsync(new
        ProblemDetails
        {
            Status = (int)HttpStatusCode.InternalServerError,
            Type = exception.GetType().Name,
            Title = "An unexpected error occurred",
            Detail = exception.Message,
            Instance = $"{httpContext.Request.Method}
            {httpContext.Request.Path}"
        });

        return true;
    }
}
```

Here I enforce the response in JSON format using the *WriteAsJsonAsync* method. I'm sure at 99.99% that your client expects a JSON response instead of XML or something else. To make it work, when an exception is raised, configure ASP.NET Core 8 to run it as shown in Listing 5-34.

Listing 5-34. Enabling the DefaultExceptionHandler in the ASP.NET Core pipeline

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

builder.Services.AddExceptionHandler<DefaultExceptionH
andler>();

var app = builder.Build();

// Your ASP.NET Core pipeline

app.UseExceptionHandler(opt => { });

// Your ASP.NET Core pipeline

app.Run();
```

As usual, we have to configure ASP.NET Core with an extension method, specifically the *AddExceptionHandler<T>* extension, where *T* is the handler you want to register in the pipeline. To enable it, we will add the *UseExceptionHandler* middleware, which takes a mandatory parameter, a delegate that configures the options of the handler. It can remain empty by default. We don't configure any options to make it work properly. I chose to return an Internal Server Error (500), the default status code to return when an error is raised. If we execute the GET /exception endpoint, the exception raised should be well handled and formatted. Listing 5-35 shows the GET /exception endpoint.

Listing 5-35. The GET /exception endpoint raising an exception

```
app.MapGet("/exception", () => {
    throw new Exception();
});
```

Figure 5-15 shows the output in Postman.
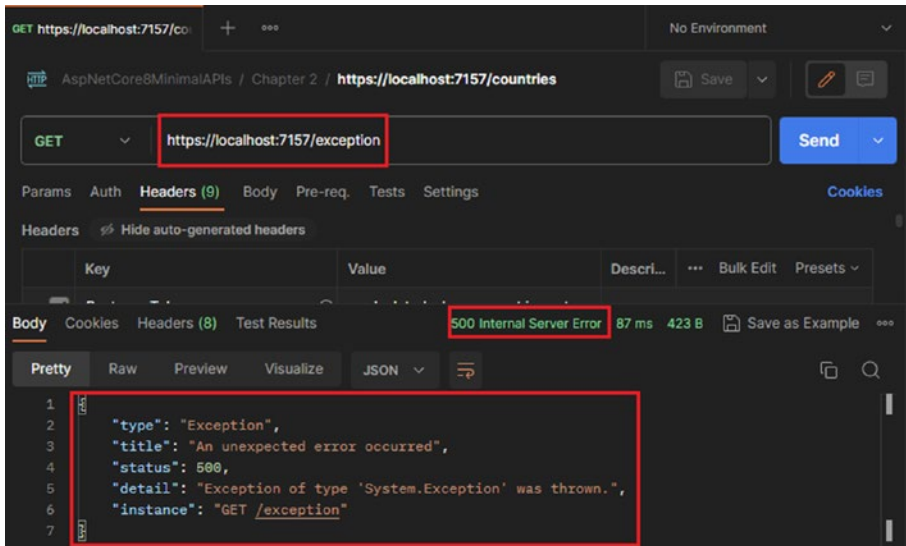
Figure 5-15. *The GET /exception endpoint output in Postman after execution*

Since it's a default exception handler, it could be great to handle more types of exceptions. How? ASP.NET Core 8 allows chaining exception handlers, which is a matching rule; in reality, it takes a test in the handler to check if the exception type matches, and then you can run your handler for a specific exception.

Let's take another example by choosing the Timeout exception handling. Why is it so important? Because it's a mistake to think that just because your API is well coded, it will give your client the response they want within a reasonable time. Usually, the HTTP Timeout (408) error is

returned, but, to me, it's **not right** to return a Timeout (408) error to the client since it's not the client's fault but rather the server's. Remember, 4xx errors involve the client, while 5xx errors involve the server. In this case, returning a Service Unavailable (503) error to the client is **more appropriate**. That's why I want to show you how to handle Timeout errors. Listing 5-36 shows the *TimeOutExceptionHandler* class, which runs only if it detects a raised in the application.

Listing 5-36. The TimeOutExceptionHandler class

```
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using System.Net;

namespace AspNetCore8MinimalApis.ExceptionHandlers;

public class TimeOutExceptionHandler : IExceptionHandler
{
    public async ValueTask<bool> TryHandleAsync(HttpContext
    httpContext, Exception exception, CancellationToken
    cancellationToken)
    {
        if (exception is TimeoutException)
        {
            httpContext.Response.StatusCode = (int)
            HttpStatusCode.ServiceUnavailable; // Manual setup
            to replace the default Internal Server error

            await httpContext.Response.WriteAsJsonAsync(new
            ProblemDetails
            {
                Status = (int)HttpStatusCode.
                ServiceUnavailable,
```

```
            Type = exception.GetType().Name,
            Title = "A timeout occurred",
            Detail = exception.Message,
            Instance = $"{httpContext.Request.Method}
            {httpContext.Request.Path}"
        });
        return true;
    }
    return false;
    }
}
```

Note   Internal Server Error is the default HTTP status returned
to the client. If you want to put another status, as I did for the
TimeOutExceptionHandler, you must set it up manually.

Since the handlers' registration order matters and all exception
handlers are evaluated (if the previous does not break the pipeline by
returning *True*), we must place the *TimeOutExceptionHandler* in the
first position; the *DefaultExceptionHandler* should be the last one to
be executed since it's the default. If a Timeout exception is raised, the
TimeOutExceptionHandler will handle it and break the pipeline to return
the response to the client, and the *DefaultExceptionHandler* won't run.
Listing 5-37 shows the *TimeOutExceptionHandler* class registration before
the *DefaultExceptionHandler* class.

Listing 5-37.   The registration of the TimeOutExceptionHandler class

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

builder.Services.AddExceptionHandler<TimeOutException
Handler>();
```

```
builder.Services.AddExceptionHandler<DefaultExceptionH
andler>();
```

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

If we execute the GET /timeout endpoint, the exception raised should be handled well by the *TimeOutExceptionHandler* class. Listing 5-38 shows the GET /timeout endpoint.

Listing 5-38. The GET /timeout endpoint raising a timeout exception

```
app.MapGet("/timeout", () => {
    throw new TimeoutException();
});
```

Figure 5-16 shows the output in Postman.
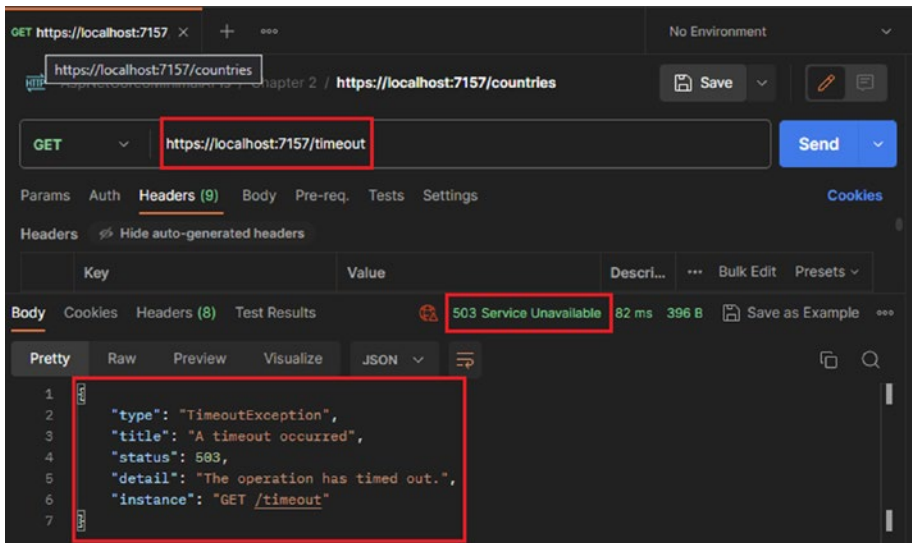
Figure 5-16. *The GET /timeout endpoint output in Postman after execution*

If you follow this principle, you can handle any exception properly and, above everything, handle as many exceptions as you want. Don't miss out on this feature!

Summary

I hope you enjoyed this chapter. I haven't gone too far into ASP.NET Core 8 here, but I've only provided you with the elements you'll need to take your APIs up a notch since Chapter 4. You can do without the features in this chapter, but if you use them, they'll make life much easier. We've seen quite a bit about minimal APIs here, so it's time to move on to what's happening behind them. The next chapter will teach you how to access data, even from different data sources, and, once again, how to structure your APIs around the external data you will have access to.

Accessing Data Safely and Efficiently

So far, we have run into ASP.NET Core functionalities but haven't yet discussed what's behind the scenes. What's going on behind the scenes? Generally, we access data from multiple sources. We usually access data from a SQL database or via HTTP by calling another remote API. In this chapter, I will teach you how to access data safely and efficiently. In this chapter, you will learn the following:

- Introduction to data access best practices

- Accessing data with Entity Framework (EF) Core 8

- Accessing data with HttpClient and REST APIs

Introduction to Data Access Best Practices

Before getting to the code, let's discuss data access best practices. It's not just about the data but how you access it. There are other types of data access, such as gRPC, OData, or NoSQL, but I won't go into them as SQL and HTTP are the most popular. Let's focus on SQL and HTTP.

In both cases, you may face problems connecting to your data source, but unfortunately, you won't have any control over this. The reasons may be multiple, such as a network problem or the remote resource not

responding because it's overloaded or unavailable for a few moments. Although you have no control over this, it is possible to manage these problems by adopting a **Retry** strategy. **Transient** errors, that is, temporary errors that can be resolved by themselves, can be replayed to avoid rendering your application non-functional. This is what we call **resilience**. It applies to SQL connections, HTTP requests, and any calls to remote resources. However, there are other specificities to each type of data access. I will use the **Polly** library.

SQL-Type Data Access

The SQL case is the trickiest because you have two things to take into account:

1. Consider the parameters you receive to carry out your queries safely. An attack allows you to corrupt character strings by injecting pieces of SQL queries to modify a query's behavior and obtain sensitive information through an unprotected query. In this chapter, I'll explain a technique to prevent SQL Injection. Using an *Object Relational Mapping (ORM)* such as Entity Framework Core will expose the SQL database through C# code using *Language Integrated Query (LINQ)*. We've already used LINQ earlier in this book, and there's an implementation with Entity Framework Core, which translates LINQ statements into SQL queries. This way, you're protected from SQL injections.

2. Manage your SQL connections. SQL databases—in this chapter, I'll be using SQL Server—require you to open a connection before performing any query, which is costly and time-consuming and requires

268

closing the connection. So we're going to save on performance by using database connection pooling, which lets you leave a connection open so it can be reused to perform another query. This is important and makes a real difference at high database traffic levels. I'll show you how in this chapter too.

HTTP Data Access

Regarding HTTP requests in .NET, we'll be making them via an HTTP client. But a particular client manages a limited resource: an HTTP connection. Like SQL requests, we'll need to pay close attention to HTTP connections. We'll use a typed HTTP client, which backs up the *IHttpClientFactory*, enabling efficient management of *HttpMessageHandler*, subject to problems such as socket exhaustion when too many instances are open. I'll take this opportunity to show you a library that simplifies your remote HTTP calls. Still based on *IHttpClientFactory*, it overlays on typed HTTP clients to enable more straightforward management of the latter, using less code. This is the **Refit** library. I will explain this in detail in the dedicated subsection further in this chapter.

Architecturing Data Access

One more thing before we get to the code.

At the beginning of the book, I introduced you to the best practices in terms of architecture. I insisted on decoupling. In the code examples I will give you, I'll create an Infrastructure layer to isolate the technology. As these are two distinct technologies (SQL and HTTP), I will isolate my code in a separate layer each time. In my opinion, each data access technique should be isolated in its layer, because we shouldn't mix technologies. Each layer depends on the Domain layer; as you already know, the

Domain layer will expose data contracts (DTOs or domain objects) and service and repository interfaces. Finally, each technology access layer implements repository interfaces and returns DTOs consumed by a service layer (BLL), itself consumed by the API layer discussed in previous chapters. Figure 6-1 summarizes the application architecture with the data access layers. You'll find this architecture in the source code supplied with the book.
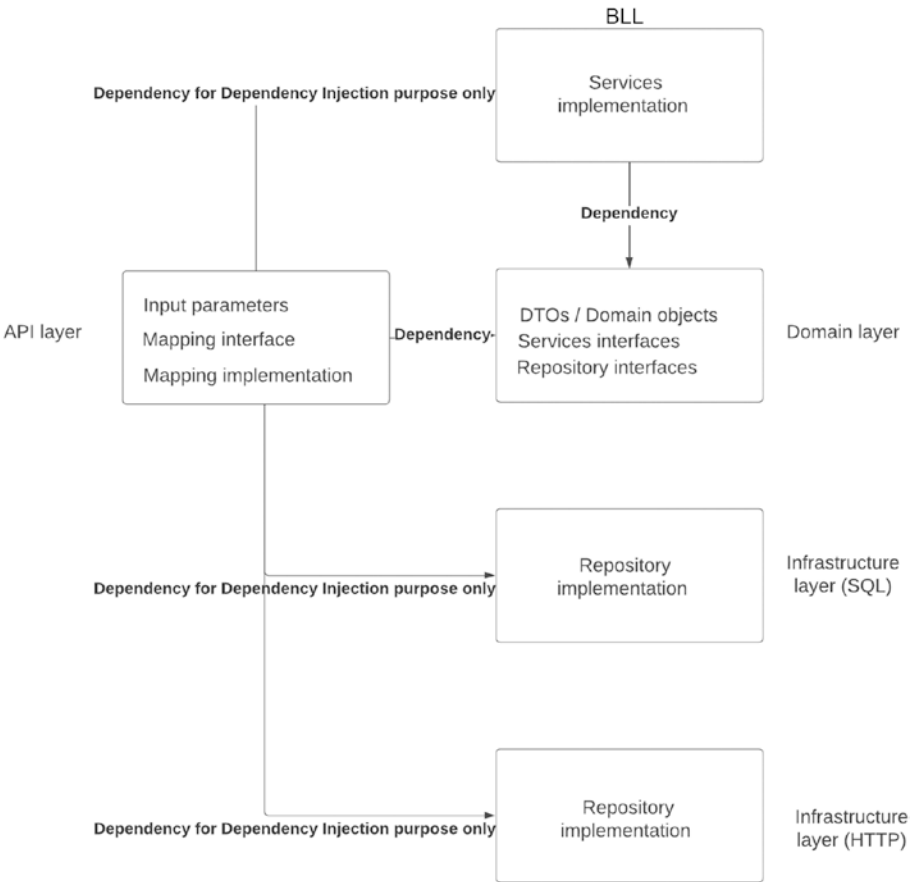
Figure 6-1. *Solution architecture with Infrastructure layers*

As you can see, each layer is decoupled from each other, except for the API that depends on each layer, for dependency injection purposes. The API needs access to both abstractions (interfaces and implementations) to register those services and repositories in the dependency injection system.

> **Note**   I won't catch and handle exceptions in these data access layers. As I showed you in the previous chapter, I will let the code crash—it does—and then let the *ExceptionFilter* classes I designed for this handle the errors. As I showed you, managing the exception type is up to you.

Accessing Data with Entity Framework Core 8

Entity Framework Core (EF Core) is a data access framework developed by Microsoft. An ORM lets you map your database to exactly C# code—entities. Each entity is mapped to a table in the SQL database (SQL Server). We will use a simple example, that is, a SQL table, to show you how to map it to an entity (class) in C# and the possible optimizations linked to this technological choice.

> **Note**   I won't go into detail with Entity Framework Core. I'll introduce it to you, as it could be the subject of an entire book. I will, however, create the database with Entity Framework and a table named *Countries*, on which we'll perform queries.

Let's create a new layer named Infrastructure.SQL. I won't detail it again; you know how to make it and download the following Nuget package from the NuGet Package Manager:

```
Microsoft.EntityFrameworkCore.SqlServer
```

Your project should look like what Figure 6-2 shows.
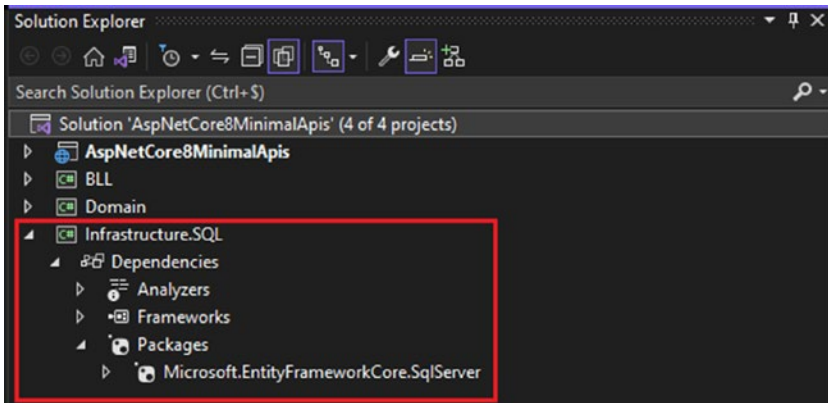
Figure 6-2. *The Infrastructure.SQL layer*

Note   Everything I will create in this section belongs to the Infrastructure layer, so you won't need to wonder where it should be created.

Step 1: Creating the CountryEntity Class

Here's our first step, creating our *CountryEntity* class. Although identical to the *CountryDto* class, this entity doesn't have the same responsibility. It's mapped to the database, whereas the *CountryDto* class is a domain object, not linked to the database. Listing 6-1 shows the *CountryEntity* class.

Listing 6-1. The CountryEntity class

```
namespace Infrastructure.SQL.Database.Entities;
public class CountryEntity
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public string FlagUri { get; set; }
}
```

We have created the entity that maps to the database. Let's configure the database context execution now.

Step 2: Creating the EF Core Context

In the EF Core universe, a database context is a class used to initialize a context (a connection and entity state). We will create a context class called *DemoContext*, inherited from the *DbContext* class. To be clear, an execution context means a *DbContext* instance that allows you to manipulate entities and generate SQL queries using LINQ statements to add, modify, retrieve, or delete instances of these entities in the database. We'll then declare a *DbSet*, a class that defines the C# entities linked to the database, a property of the *DemoContext* class. Here we'll have a *DbSet* for the *CountryEntity* entity. Listing 6-2 shows the *DemoContext* class.

Listing 6-2. The DemoContext class

```
using Infrastructure.SQL.Database.Entities;
using Microsoft.EntityFrameworkCore;

namespace Infrastructure.SQL.Database;

public class DemoContext : DbContext
```

```
{
    public DemoContext (DbContextOptions options) :
    base(options)
    {
    }
    public DbSet<CountryEntity> Countries { get; set; }
}
```

The context class is created; we will need now to configure entities.

Step 3: Configuring the CountryEntity

We've created a *CountryEntity* entity and an execution context for it. Now
we need to configure our *CountryEntity* entity. As you know, we have a
lot of SQL Server data behind us; to map our entity to a SQL table, we'll
need to tweak it so that Entity Framework Core understands the precise
mapping it needs to perform with the database. For example, EF Core
knows by default how to map an integer (int32) to an integer in SQL and
a string to a varchar, but EF Core doesn't know how to create a primary
key on its own, so we'll have to tell it which property of the *CountryEntity*
class is the primary key so that it can create it on the database side. EF
Core also doesn't know on its own what constraint you want to add to your
properties. For example, we will define a maximum length for a country
description of 200 characters and add a uniqueness constraint to the
country name, that is, on the SQL side, the country name value can't be
inserted twice. All fields will be mandatory and can't be null.

There are two ways of proceeding, either by adding attributes to the
properties of the *CountryEntity* entity or by configuring in the *DbContext*,
using a method called *OnModelCreating*. In this method, we'll configure
the name of the table mapped to the *CountryEntity* entity. Although Entity
Framework Core, by convention, can automatically give a name to a table

mapped to an entity, I prefer to explicitly give it a name and a SQL schema to which the table will belong. Listing 6-3 shows the *DemoContext* class enriched with the *CountryEntity* configuration.

Listing 6-3. The DemoContext class enriched with the CountryEntity configuration

```
using Infrastructure.SQL.Database.Entities;
using Microsoft.EntityFrameworkCore;

namespace Infrastructure.SQL.Database;

public class DemoContext : DbContext
{
    public DemoContext (DbContextOptions options) : base(options)
    {
    }

    public DbSet<CountryEntity> Countries { get; set; }

    protected override void OnModelCreating(ModelBuilder
    modelBuilder)
    {
        var builder = modelBuilder.Entity<CountryEntity>();
        builder.ToTable("Countries", "dbo");
        builder.HasIndex(p => p.Name).IsUnique(true);
        builder.Property(e => e.Id).ValueGeneratedOnAdd();
        builder.Property(e => e.Name).IsRequired();
        builder.Property(p => p.Description).HasMaxLength(200).
        IsRequired();
        builder.Property(p => p.FlagUri).IsRequired();

        base.OnModelCreating(modelBuilder);
    }
}
```

As you can see, it's pretty simple to define. The primary key, which autoincrements itself, is configured with the *ValueGeneratedOnAdd* method on the *Id* property. This will be the primary key of the *Countries* table in the *dbo* SQL schema. All fields are required and are defined using the *IsRequired* method, and the *Name* property has an index that can be used to create a uniqueness constraint using the *IsUnique* method. Finally, the *Description* property has a maximum length of 200 characters defined using the *HasMaxLength* method. The SQL column names will take the C# properties' names by default. I did not show it to you because I wanted to focus on the essentials.

We're done with the *DemoContext* configuration; in the next step, I will show you how to generate the model SQL-side.

Step 4: Generating the Database Model from C#

To generate the database, we'll need to do two things: set the database connection string and tell ASP.NET Core to create (or update) the database when the application starts.

Let's assume we have a SQL database on our local machine. A SQL Server database (LocalDb) is automatically installed when Visual Studio 2022 is installed. To do this, go to the *appsettings.json* file and add the connection string named "DemoDb", as shown in Listing 6-4.

Listing 6-4. The database

```
"ConnectionStrings": {
    "DemoDb": "Data Source=(LocalDB)\\MSSQLLocalDB;Initial
    Catalog=DemoDb;MultipleActiveResultSets=true;Encrypt=false;
    timeout=30;"
  }
```

I added a connection timeout, set to 30 seconds. It means that when a connection is unavailable to the database, ASP.NET Core will wait 30 seconds before raising an exception because a connection could not be established. It's always an excellent practice to set up a timeout since you want to limit the time the user will wait to get a response, and in the meantime, you want to give ASP.NET Core a chance to get an available connection to SQL Server. Let's write an instruction to ask ASP.NET Core when it starts to generate or update the database and also get the database connection from the *appsettings.json* file and register in the dependency injection system the *DemoContext* as shown in Listing 6-5.

Listing 6-5. The database connection configuration

```
using Infrastructure.SQL.Database;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

var dbConnection = builder.Configuration.
GetConnectionString("DemoDb");
builder.Services.AddDbContextPool<DemoContext>(options =>
options.UseSqlServer(dbConnection));

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var db = scope.ServiceProvider.GetRequiredService<Demo
    Context>();
    db.Database.SetConnectionString(dbConnection);
    db.Database.Migrate();
}

app.Run();
```

As you can see, ASP.NET Core is configured to connect to the database. I got the connection string value properly from the configuration with the *GetConnectionString* method. Then I register the DemoContext class in the dependency injection system with the *AddDbContextPool* method. The latter enables connection pooling, which will keep some connections open to the database and let them be reused, and this ensures better performances since opening/closing connections won't happen each time a connection is needed to query the database. Finally, I use the Migrate method, which will execute database migration. Database migrations are C# files that will generate SQL instructions to create/update the database each time we add or modify anything in the database model, entities themselves, or the entities' configuration in the *DemoContext*, as I showed you before. To generate database migration with EF Core, install the following package on the API layer: `Microsoft.EntityFrameworkCore.Design`.

You'll also need to install the following package on your Infrastructure. SQL layer: Microsoft.EntityFrameworkCore.Tools.

Then open Package Manager Console, select the Infrastructure.SQL layer in the drop-down list, and type the following command: Add-migration Initial.

This command will create a migration file named *Initial.cs*, as shown in Figure 6-3.
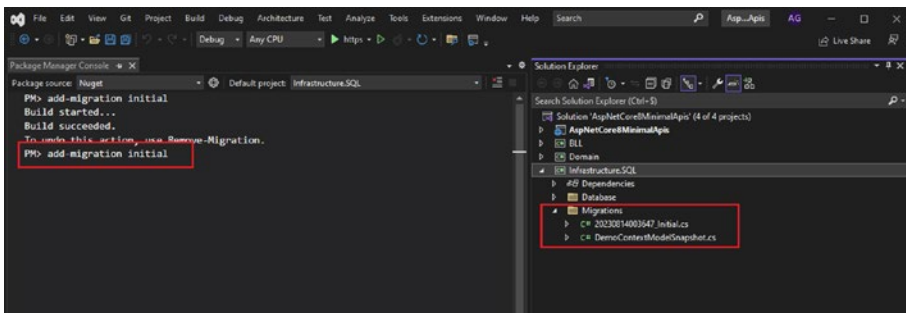
Figure 6-3. *The initial migration generation*

If we run the application, the database and the *Countries* table should be created as shown in Figure 6-4.
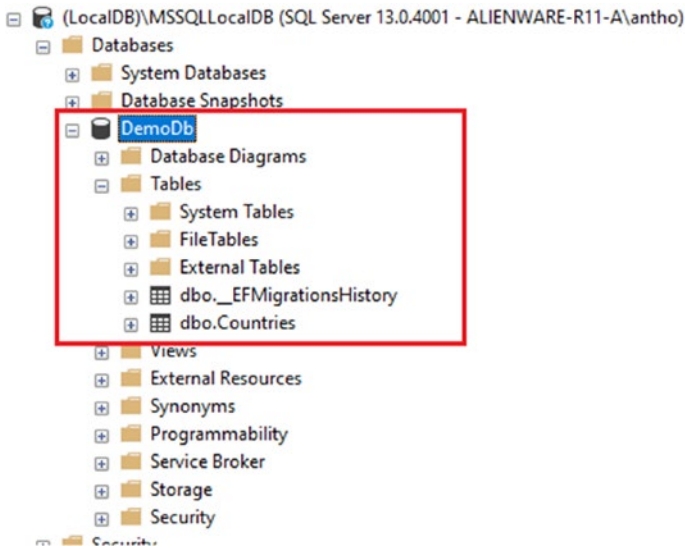
Figure 6-4. *The Demo database generated*

Note   I took the preceding picture from the SQL Server Management Studio software. You can install it from here: https://learn. microsoft.com/en-us/sql/ssms/download-sql-server- management-studio-ssms?view=sql-server-ver16.

Entity Framework Core creates a table that contains migration history. In other words, it keeps the last migration history performed in the database to avoid applying the same migration each time the application runs. If you want to update the database or add stuff, you must repeat the preceding same operation by generating a new migration and so on. The migration ID is the name of the generated file for the migration. Figure 6-5 shows the content of the history table.
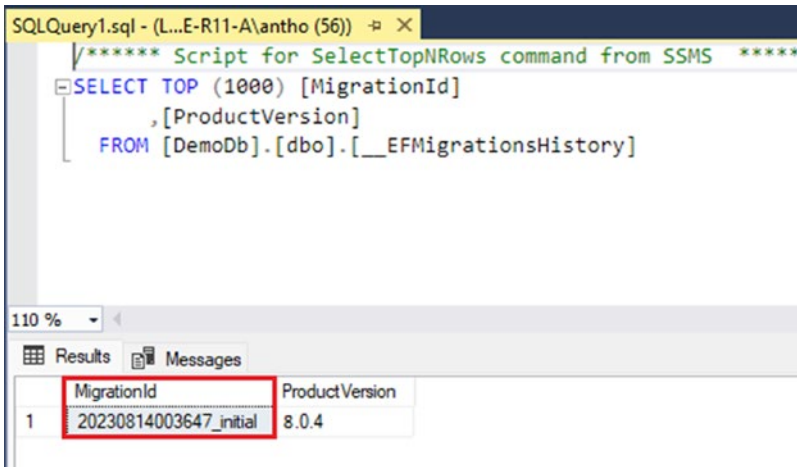
Figure 6-5. *The migration history table*

Step 5: Enabling Resiliency with Entity Framework Core

Enabling resilience on a SQL connection with Entity Framework Core is very easy. There's no need for a library like Polly, which we'll use for HTTP errors, although it can also handle SQL connection errors (and other transient errors). To do this, add the EnableRetryOnFailure option to the SQL Server options, as shown in Listing 6-6.

Listing 6-6. Enabling resiliency on SQL Server connections

```
builder.Services.AddDbContextPool<DemoContext>(options =>
    options.UseSqlServer(dbConnection,
                    sqlServerOptionsAction: sqlOptions =>
                    {
                        sqlOptions.EnableRetryOnFailure(
                        maxRetryCount: 3);
                    }));
```

As you can see here, I enabled a **Retry** strategy; this code will retry three times before raising an exception due to a transient error. You can improve this strategy by setting up the delay between retries or adding more transient errors than default ones. If you want to know the default transient errors handled by Entity Framework Core, you can visit the GitHub repository, and it shows all SQL Server transient errors: https://github.com/Azure/elastic-db-tools/blob/master/Src/ ElasticScale.Client/ElasticScale.Common/TransientFaultHandling/ Implementation/SqlDatabaseTransientErrorDetectionStrategy.cs.

I showed you the simplest way to handle transient errors here; what you need to understand here is that you should always handle them. I want to sensibilize you on this hot topic, and I showed you the way. If you want to improve the preceding example because your business needs it, you can check the Microsoft documentation here: https://learn.microsoft.com/ en-us/dotnet/architecture/microservices/implement-resilient- applications/implement-resilient-entity-framework-core-sql- connections.

Since it's hard to simulate transient errors on SQL Server, I can't show any examples here. Still, you can trust me it works like a charm, and once your database is not responsible or temporarily unavailable, retries will be performed.

Step 6: Writing the Repository on Top of the CountryEntity

It's time now to write the *CountryRepository*. First off, let's write the *ICountryRepository* interface. The latter has to be written into the Domain layer, which contains all application abstractions. Listing 6-7 shows the *ICountryRepository*.

Listing 6-7. The ICountryRepository interface

```
using Domain.DTOs;
namespace Domain.Repositories;

public interface ICountryRepository
{
    Task<CountryDto> RetrieveAsync(int id);
    Task<List<CountryDto>> GetAllAsync();
    Task<int> CreateAsync(CountryDto country);
    Task<int> UpdateAsync(CountryDto country);
    Task<int> UpdateDescriptionAsync(int id, string
    description);
    Task<int> DeleteAsync(int id);
}
```

As you can see, the methods return a *Task*. I will return to this in Chapter 7 when it comes to talking about asynchronous programming. We will now implement all these methods with Entity Framework Core. Listing 6-8 shows the implementation of the *CountryRepository* class.

Listing 6-8. The CountryRepository class

```
using Domain.DTOs;
using Domain.Repositories;
using Infrastructure.SQL.Database;
using Infrastructure.SQL.Database.Entities;
using Microsoft.EntityFrameworkCore;

namespace Infrastructure.SQL.Repositories;

public class CountryRepository : ICountryRepository
{
    private readonly DemoContext _demoContext;
```

```
public CountryRepository(DemoContext demoContext)
{
    _demoContext = demoContext;
}

public async Task<int> CreateAsync(CountryDto country)
{
    var countryEntity = new CountryEntity
    {
        Name = country.Name,
        Description = country.Description,
        FlagUri = country.FlagUri
    };

    await _demoContext.AddAsync(countryEntity);
                await _demoContext.SaveChangesAsync();

    return countryEntity.Id;
}

public async Task<int> UpdateAsync(CountryDto country)
{
    var countryEntity = new CountryEntity
    {
        Id = country.Id,
        Name = country.Name,
        Description = country.Description,
        FlagUri = country.FlagUri
    };

    return await _demoContext.Countries
                            .Where(x =>
                            x.Id == countryEntity.Id)
                            .ExecuteUpdateAsync(s =>
```

```
                                    s.SetProperty
                                    (p => p.Description,
                                    countryEntity.
                                    Description)
                                    .SetProperty
                                    (p => p.FlagUri,
                                    countryEntity.FlagUri)
                                    .SetProperty
                                    (p => p.Name,
                                    countryEntity.Name));
    }
    public async Task<int> DeleteAsync(int id)
    {
        return await _demoContext.Countries
                            .Where(x => x.Id == id)
                            .ExecuteDeleteAsync();
    }

    public async Task<List<CountryDto>> GetAllAsync()
    {
        return await _demoContext.Countries
                            .AsNoTracking()
                            .Select(x => new CountryDto
                            {
                                Id = x.Id,
                                Name = x.Name,
                                Description = x.Description,
                                FlagUri = x.FlagUri
                            })
                            .ToListAsync();
    }

    public async Task<CountryDto> RetrieveAsync(int id)
    {
```

```
        return await _demoContext.Countries
                        .AsNoTracking()
                        .Where(x => x.Id == id)
                        .Select(x => new CountryDto
                        {
                            Id = x.Id,
                            Name = x.Name,
                            Description = x.Description,
                            FlagUri = x.FlagUri
                        })
                        .FirstOrDefaultAsync();
    }

    public async Task<int> UpdateDescriptionAsync(int id,
    string description)
    {
        return await _demoContext.Countries
                            .Where(x => x.Id == id)
                            .ExecuteUpdateAsync
                            (s => s.SetProperty(p =>
                            p.Description, description));
    }
}
```

Entity Framework Core syntax is pretty straightforward, and it's LINQ plus some elements that enables optimizing a query or executing the SQL query:

- **AsNoTracking**: This method allows you to gain some performance since it tells Entity Framework Core not to track the state of an entity. I won't go into detail here. To learn more about change tracking in EF Core, read the Microsoft documentation here: https://learn. microsoft.com/en-us/ef/core/change-tracking/.

Usually, we don't need to track an entity if we don't modify it in the context of the entity requested from the database. If you retrieve an entity from the database and send it straight to the client, you don't need to track it.

- **AddAsync/SaveChangesAsync**: This pair of methods allow you to add asynchronously the entity in the *DbContext* to be inserted (asynchronously as well) into the database. After calling the *SaveChangesAsync* method, the entity is saved in the database, and the latter will populate the country ID that has been defined as the primary key and auto-incremented. To return it to the client, return the country ID.

- **FirstOrDefaultAsync:** Asynchronously triggers the query to the database and returns the first element that matches the query condition in the *Where* clause. It returns the default value of the entity (null, when it's an object) when not found.

- **ExecuteDeleteAsync:** Asynchronously triggers the query to the database and deletes all elements that match the query condition in the *Where* clause.

- **UpdateDeleteAsync:** Asynchronously triggers the query to the database and updates all elements that match the query condition in the *Where* clause.

Note   I systematically use the asynchronous methods (async/await). All the mentioned methods have their synchronous version, but I never use them. Chapter 7 will explain why.

Another interesting thing is using the *CountryDto* class in the Select statement. Why did I use *CountryDto* in the query? It's because I'm projecting the *Country* entity into the *CountryDto* object. SQL speaking, Entity Framework Core will **only request** the field I'm mapping into *CountryDto* instead of selecting all fields from the database and then mapping only those I need. This is called **projection**, and it's excellent to optimize performance because I'm bringing only the field I need from the SQL query.

We can now write the final implementation of the *CountryService* class, whose implementation I haven't yet shown you. Here it is, as shown in Listing 6-9.

Listing 6-9. The CountryService class

```
using Domain.DTOs;
using Domain.Repositories;
using Domain.Services;

namespace BLL.Services;

public class CountryService : ICountryService
{
    private readonly ICountryRepository _countryRepository;

    public CountryService(ICountryRepository countryRepository)
    {
        _countryRepository = countryRepository;
    }

    public async Task<bool> DeleteAsync(int id)
    {
        return await _countryRepository.DeleteAsync(id) > 0;
    }
```

```
    public async Task<List<CountryDto>> GetAllAsync()
    {
        return await _countryRepository.GetAllAsync();
    }
    public async Task<CountryDto> RetrieveAsync(int id)
    {
        return await _countryRepository.RetrieveAsync(id);
    }

    public async Task<int> CreateOrUpdateAsync(CountryDto
    country)
    {
        if (country?.Id is null)
            return await _countryRepository.
            CreateAsync(country);

        if (await _countryRepository.CreateAsync(country) > 0)
            return country.Id;

        return 0;
    }

    public async Task<bool> UpdateDescriptionAsync(int id,
    string description)
    {
        return await _countryRepository.
        UpdateDescriptionAsync(id, description) > 0;
    }
}
```

Finally, here are the endpoints reworked in the *Program.cs* file to accept asynchronous *ICountryService* methods as shown in Listing 6-10.

> **Note**   For code readability purposes, I removed the using statements that were a very long list here.

Listing 6-10. The Program.cs file

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

var dbConnection = builder.Configuration.
GetConnectionString("DemoDb");
builder.Services.AddDbContextPool<DemoContext>(options =>
    options.UseSqlServer(dbConnection,
                         sqlServerOptionsAction: sqlOptions =>
                         {
                             sqlOptions.EnableRetryOnFailure(
                             maxRetryCount: 3);
                         }));
builder.Services.AddValidatorsFromAssemblyContaining
<Program>();
builder.Services.AddScoped<ICountryMapper, CountryMapper>();
builder.Services.AddScoped<ICountryService, CountryService>();
builder.Services.AddScoped<ICountryRepository,
CountryRepository>();
builder.Services.AddExceptionHandler<TimeOutExceptionH
andler>();
builder.Services.AddExceptionHandler<DefaultExceptionH
andler>();

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

```
app.MapPost("/countries", async (
[FromBody] Country country,
ICountryMapper mapper,
ICountryService countryService) => {
    var countryDto = mapper.Map(country);
    var countryId = await countryService.CreateOrUpdateAsync(
        countryDto
    );
    if (countryId <= 0)
        return Results.StatusCode(
            StatusCodes.Status500InternalServerError
        );
    return Results.CreatedAtRoute(
            "countryById", new { Id = countryId }
        );
}).AddEndpointFilter<InputValidatorFilter<Country>>();

app.MapGet("/countries/{id}", async (
int id,
ICountryMapper mapper,
ICountryService countryService) => {
    var country = await countryService.RetrieveAsync(id);

    if (country is null)
        return Results.NotFound();

    return Results.Ok(mapper.Map(country));
}).WithName("countryById");

app.MapGet("/countries", async (
ICountryMapper mapper,
ICountryService countryService) => {
```

```
    var countries = await countryService.GetAllAsync();
    return Results.Ok(mapper.Map(countries));
});

app.MapDelete("/countries/{id}", async (
int id,
ICountryService countryService) => {
    if (await countryService.DeleteAsync(id))
        return Results.NoContent();

    return Results.NotFound();
});
app.MapPut("/countries", async (
[FromBody] Country country,
ICountryMapper mapper,
ICountryService countryService) => {
    var countryDto = mapper.Map(country);
    var countryId = await countryService.CreateOrUpdateAsync(
        countryDto
    );
    if (countryId <= 0)
        return Results.StatusCode(
                StatusCodes.Status500InternalServerError
            );

    if (country.Id is null)
        return Results.CreatedAtRoute(
                "countryById",
                new { Id = countryId }
            );
    return Results.NoContent();
}).AddEndpointFilter<InputValidatorFilter<Country>>();
```

```
app.MapPatch("/countries/{id}", async (
int id,
[FromBody] CountryPatch countryPatch,
ICountryMapper mapper,
ICountryService countryService) => {
    if (await countryService.UpdateDescriptionAsync(
            id,
            countryPatch.Description)
        )
        return Results.NoContent();

    return Results.NotFound();
}).AddEndpointFilter<InputValidatorFilter<CountryPatch>>();
using (var scope = app.Services.CreateScope())
{
    var db = scope.ServiceProvider.GetRequiredService<Demo
    Context>();
    db.Database.SetConnectionString(dbConnection);
    db.Database.Migrate();
}

app.Run();
```

As you can see, we've implemented the endpoints right down to the database, all correctly in each layer. Figure 6-6 shows what the global ASP.NET Core project looks like.
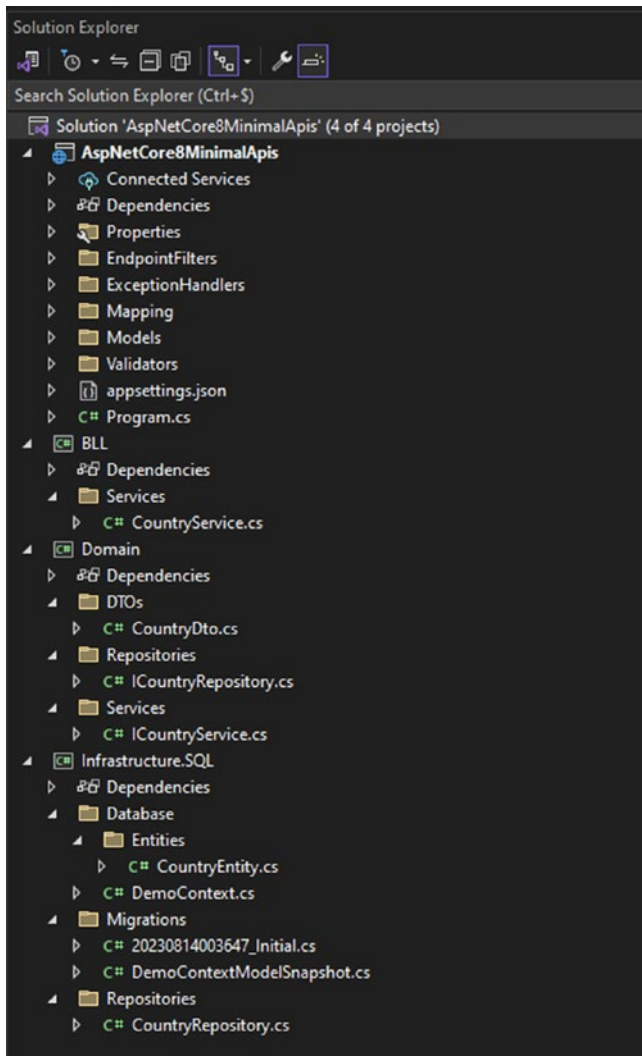
Figure 6-6. *The global ASP.NET Core solution*

Congratulations! You've come a long way! From endpoint definition to the database!

In the following subsection, we'll look at data access via HTTP, with all its constraints and optimizations.

Accessing Data with HttpClient and REST APIs

It's common for an API to access data from other than a SQL database. As REST APIs are popular, it's not uncommon for data sources to be exposed via REST. This requires access via the *HttpClient* class. We're going to instantiate the *HttpClient* class to make GET requests on an API. Listing 6-11 shows how to download the contents of an image using the *HttpClient* class.

Listing 6-11. The HttpClient class usage

```
using (var client = new HttpClient())
{
    byte[] fileBytes = await client.GetByteArrayAsync("https://
    anthonygiretti.blob.core.windows.net/countryflags/ca.png");
}
```

In reality, this practice is disastrous in terms of performance for your application. As I said earlier in this chapter, each *HttpClient* consumes an instance of the *HttpMessageHandler* class, and a large number of instances of the *HttpMessageHandler* class can lead to socket exhaustion. To avoid this, we can use the *IHttpClientFactory* interface, which will manage *HttpMessageHandler* instances for us, reusing them. There are several ways of implementing *IHttpClientFactory*, and I will show you my favorite, based on *IHttpClientFactory*. To do this, I will use the Refit library, based on *IHttpClientFactory*, which will make life much easier.

Using IHttpClientFactory to Make HTTP Requests

Before showing you Refit, here's a quick reminder of what *IHttpClientFactory* is. It's a .NET 8 (since .NET Core 1) interface for optimized HTTP requests. The *IHttpClientFactory* interface is not automatically known by .NET and its dependency injection system, so you'll need to download the following package: `Microsoft.Extensions.Http`.

Once downloaded, add the following line in the *Program.cs* file: `builder.Services.AddHttpClient();`.

Now, let's create an *IMediaRepository* in the Domain layer, followed by its implementation in the Infrastructure.Http layer. Listing 6-12 shows the signature of the *IMediaRepository* interface, which will expose a method for returning the content of an image in a byte array and its MIME type, all in *tuple* form.

Listing 6-12. The IMediaRepository interface

```
namespace Domain.Repositories;

public interface IMediaRepository
{
    Task<(byte[] Content, string MimeType)>
    GetCountryFlagContent(string countryShortName);
}
```

Here's its implementation in the *MediaRepository* class with *IHttpClientFactory*, as shown in Listing 6-13.

Listing 6-13. The MediaRepository class

```
using Domain.Repositories;

namespace Infrastructure.Http.Repositories;

public class MediaRepository : IMediaRepository
```

```
{
    private readonly IHttpClientFactory _httpClientFactory;

    public MediaRepository(IHttpClientFactory
    httpClientFactory)
    {
        _httpClientFactory = httpClientFactory;
    }

    public async Task<(
    byte[] Content,
    string MimeType
    )>
    GetCountryFlagContent(string countryShortName)
    {
        byte[] fileBytes;

        using HttpClient client = _httpClientFactory.
        CreateClient();
        fileBytes = await client.GetByteArrayAsync($"https://
        anthonygiretti.blob.core.windows.net/countryflags/
        {countryShortName}.png");

        return (fileBytes, "image/png");
    }
}
```

The implementation of the MediaRepository class is quite simple. Using *IHttpClientFactory* is quite simple. There are other forms of *IHttpClientFactory* usage in .NET 8, and all rely on this interface. For example, there are typed clients, named clients that you can find examples on the Microsoft documentation here: https://learn.microsoft.com/en-us/dotnet/ architecture/microservices/implement-resilient-applications/use-httpclientfactory-to-implement-resilient-http-requests.

I won't show you examples here because I want to focus on the Refit library, which relies on typed clients relying themselves on the *IHttpClientFactory* interface.

Using Refit to Make HTTP Requests

Refit lets you dynamically generate a typed HTTP client, which is handy when saving code.

All you need to do is declare an interface that specifies the information linked to the REST API (routes, parameters, body, headers) as attributes. To get started, download the following package: `refit`.

Let's go back to our *IMediaRepository* interface and decorate the GetCountryFlagContent method with Refit attributes to determine its behavior, as shown in Listing 6-14.

Listing 6-14. The IMediaRepository interface designed with Refit

```
using Refit;

namespace Domain.Repositories;

public interface IMediaRepository
{
    [Get("/countryflags/{countryShortName}.png")]
    Task<byte[]> GetCountryFlagContent(string
    countryShortName);
}
```

As you can see, you only need to add the *Get* attribute to define the request verb and pass it to the URL segment defining the route. There's no need to define the base URL here—we can configure it once so it applies to all members of the *IMediaRepository* interface, if any. Finally, the route parameters will be interpreted with the interface parameter values. Practical, isn't it? The only drawback here is that Refit doesn't support

tuples, for example. Refit only returns the result of the API call; we will have to handle the MIME type in the service layer instead of the repository. The next step is to register the Refit client in the *Program.cs* file, as shown in Listing 6-15.

Listing 6-15. The Program.cs file with Refit

```
builder.Services.AddRefitClient<IMediaRepository>().
ConfigureHttpClient(c => c.BaseAddress = new Uri("https://
anthonygiretti.blob.core.windows.net"));
```

To register our interface as a typed client with Refit, we had to use the *AddRefitClient* method, which required installation of the following package: Refit.HttpClientFactory.

I'm not going to show you how to use each verb. Their basic operation is almost identical, so I'll let you learn more about Refit with its beautiful documentation here: https://reactiveui.github.io/refit/.

The aim here is to show you the best practices you need to know to access data using remote APIs. I won't write a service to interface between the repository and the API endpoint. You know how to do it yourself and where to implement it.

Using Polly to Make HTTP Requests Resilient

The Polly library allows you to implement the **Retry and Circuit-Breaker** patterns. Polly works very well with HTTP requests, and I will show you an example of its use. **Retry** and **Circuit-Breaker** patterns will return an error to your client after several unsuccessful retries. If all retries fail, the **Circuit-Breaker** pattern will occur and automatically block HTTP calls for a customizable period. This prevents overloading the network when HTTP calls to the requested resource fail, and it will let the time to the remote resource recover.

What's interesting with typed clients such as Refit, and unlike the use of *IHttpClientFactory*, is that the **Retry** pattern configuration can be done outside the repository implementation that implements the *HttpClient*. It's interesting because we'll avoid polluting our data access layer with a Retry pattern. If we can define it separately, it's better in terms of Separation of Concerns and reduces the complexity of the repository that implements the *HttpClient*. This is why I will show you how to implement a **Retry** pattern with Refit only. To do so, download the following package: `Microsoft.Extensions.Http.Polly`.

Next, let's create a static class. Let's call it *RetryPolicy* class and implement the *AddDefaultHandlingPolicy* method, which is an extension method on the *IHttpClientBuilder* interface used to build typed HttpClient. Listing 6-16 shows the *RetryPolicy* class.

Listing 6-16. The RetryPolicy class

```
using Polly;
using Polly.Extensions.Http;

namespace AspNetCore8MinimalApis.Resiliency.Http;

public static class RetryPolicy
{
    public static void AddFaultHandlingPolicy(this
    IHttpClientBuilder builder)
    {
        var retryPolicy = HttpPolicyExtensions
                .HandleTransientHttpError() // Handles
                5XX and 408
                .WaitAndRetryAsync(3, retryDelayInSeconds =>
                TimeSpan.FromSeconds(3));

        var circuitBreakerPolicy =
         HttpPolicyExtensions
```

```
        .HandleTransientHttpError() // Handles 5XX and 408
        .CircuitBreakerAsync(4, TimeSpan.FromSeconds(15));

        var policy = retryPolicy.WrapAsync(circuitBreaker
        Policy);

        builder.AddPolicyHandler(policy);
    }
}
```

What happens here is that I'm handling HTTP transient errors (5xx
and 408) and I ask Polly with the *WaitAndRetryAsync* method to retry
three times every three seconds. Then I design a circuit breaker with the
CircuitBreakerAsync method, which takes place after three failed retries. I
put 4 as a *handledEventAllowedBeforeBreaking* parameter because I count
the initial HTTP call and then three retries so that the circuit breaker would
occur after four failed attempts. It will automatically make failing HTTP
calls for 15 seconds the time the remote resource recovers. To finish, I use
the *WrapAsync* method to merge the retry and circuit breaker policies
and add them to the *AddPolicyHandler* extension method. To apply it on
our HTTP client, designed with Refit, update the HttpClient as shown in
Listing 6-17.

Listing 6-17. The IMediaRepository Refit HttpClient updated with
retry and circuit breaker policies

```
builder.Services.AddRefitClient<IMediaRepository>()
            .ConfigureHttpClient(c => c.BaseAddress =
            new Uri("https://anthonygiretti.blob.core.
            windows.net"))
            .AddFaultHandlingPolicy();
```

We are done! So now your HTTP calls, when they fail, will retry the number of times you decide and break automatically after a determined number of failed attempts to reach the remote resources to prevent them from overloading. I strongly suggest you implement the **Retry** and **Circuit- Breaker** patterns. In terms of best practices, they are a must-do.

Summary

This chapter was, I hope, very interesting. This chapter covered the fundamentals of Entity Framework Core for accessing SQL data and the fundamentals for accessing data via HTTP, with a preview of best practices for each. Chapter 7 will take you even further, which will deal with API optimization, primarily concerning data access and more.

CHAPTER 7

Optimizing APIs

You now know how to develop endpoints, architect your applications, and access data efficiently. That's all very well, but you can go even further. I want to show you how to improve your API to allow it to scale if you have a lot of traffic on your application. The optimizations I will show you are both simple and effective, and once you know them, you can use them as often as you like when developing your APIs. Note that I won't bring up the compression topic since it's unnecessary to compress JSON data over an API. The compression efficiency is not worth it. In this chapter, you'll learn the following points:

- Asynchronous programming

- Long-running tasks with background services

- Paging

- JSON streaming

- Caching

- Speeding up HTTP requests with HTTP/2 and HTTP/3

Asynchronous Programming

I promised to get back to you on this, and now I'm about to. In some examples, I have used the following keywords: *Task<T>*, *async*, and *await*. I will explain what they mean.

Basics of Asynchronous Programming

In .NET, each operation is represented by the *Task* keyword requiring the *System.Threadings.Task* namespace. This .NET *Task* concept is an object that represents an operation that must be performed. A *Task* not only lets us know when an operation has been completed but also returns a result if necessary. To return a task, use the C# keyword as is: *Task*. If we want the *Task* to return a value, an object type must be specified, *Task<T>,* where T is an object type. A *Task* involves a user action, such as invoking an endpoint, which requests the database. This action **uses a generally blocked thread until the database response is obtained**; the longer the database response, the longer the thread is blocked. This is where asynchronous programming comes in. The *async* and *await* keywords are used to write asynchronous code. The *async* keyword is used to inform the compiler that asynchronous processing will take place on the *Task* in question, and *await* is used to wait for the *Task* in question to finish executing. During this time, the current thread is not blocked; it's even freed for computing another *Task* in your application and returns to processing the user action once the database has responded. In this way, you can ensure that all threads are not blocked simultaneously, and your application will remain available anytime. *async* and *await* are two keywords that always go together. The keyword *async* **must not** be used without *await* and vice versa. Otherwise, processing will be synchronous (the compiler will notify you of this too). As a reminder, here's some asynchronous code as shown in Listing 7-1.

Listing 7-1. The GetAllAsync method

```
public async Task<List<CountryDto>> GetAllAsync()
{
    return await _demoContext.Countries
                            .AsNoTracking()
                            .Select(x => new CountryDto
```

```
    {
        Id = x.Id,
        Name = x.Name,
        Description = x.Description,
        FlagUri = x.FlagUri
    })
    .ToListAsync();
}
```

If you remember, this is a LINQ query for accessing the database with Entity Framework Core. As you'll often see in your APIs, every time you access external data sources, you can (and should) use the asynchronous version of the proposed methods, such as the *ToListAsync()* method, which triggers an asynchronous request to the database. A synchronous version exists *(ToList)*, but I wouldn't recommend using it under any circumstances since it blocks the current thread when it's awaiting the database response. All operations with Entity Framework Core have asynchronous methods, such as *ExecuteUpdateAsync*, *ExecuteDeleteAsync*, *SaveChangesAsync,* etc. For HTTP requests, the same reasoning with the *GetAsync*, *PostAsync,* etc. is available. It's easy to identify them; they always end with the **Async** suffix. I recommend you name the asynchronous methods you are coding with the **Async** suffix. Asynchronous programming is best used when accessing an external resource whose response time you can't control for whatever reason. Both SQL and HTTP are external data sources. If we had used a file on the server to open in read mode, we could have done it synchronously, although it can also be done asynchronously, which I recommend.

You can create your *Task* when needed; you will barely need it, but it may happen when you want to perform an action where no asynchronous operation is available. It generally happens on old libraries or legacy

code where asynchronous programming is not handled. To do so, use the *Task.Run* method that will run asynchronously any synchronous code as follows:

```
var result = await Task.Run(() => DoSomething());
```

Using CancellationToken

Asynchronous programming allows you to do something convenient: cancel a task. When a task takes a long time to execute, it often happens that you want to cancel it. Whether it's a fat client like a desktop application that calls on an API or a browser, we're tempted to close the browser or the application. But what happens if the HTTP request has already invoked another API via HTTP or a SQL request and processing is still in progress? Well, the requested external resource will continue to run. To avoid this, we can handle the cancellation of an HTTP request, including the requested external resource. Let's consider the GET /cancellable endpoint, as shown in Listing 7-2.

Listing 7-2. The GET /cancellable endpoint

```
app.MapGet("/cancellable", async (ICountryService
countryService, CancellationToken cancellationToken) =>
{
    await countryService
    .LongRunningQueryAsync(cancellationToken);
    return Results.Ok();
});
```

What you have to do here is to add as a parameter to the endpoint lambda function the *CancellationToken* class, which is automatically filled by ASP.NET Core when set up as a parameter when added on HTTP endpoints. Then you will have to transmit it through all layers until the SQL query and pass it as a parameter on the *LongRunningQueryAsync* method as shown in Listing 7-3.

Listing 7-3. The CountryRepository class

```
using Domain.Repositories;
using Microsoft.EntityFrameworkCore;

namespace Infrastructure.SQL.Repositories;

public class CountryRepository : ICountryRepository
{
    private readonly DemoContext _demoContext;

    public CountryRepository(DemoContext demoContext)
    {
        _demoContext = demoContext;
    }

    public async Task LongRunningQueryAsync(
    CancellationToken cancellationToken)
    {
        await _demoContext.Database
                        .ExecuteSqlRawAsync(
                            "WAITFOR DELAY '00:00:10'",
                            cancellationToken:
                            cancellationToken);
    }
}
```

I have simulated a long-running query, ten seconds, with the *WAIT FOR DELAY* SQL command. I'm using the *Database* object, which enables you to perform SQL queries with methods like *ExecuteSqlRawAsync*.

If we run the GET /cancellable endpoint and cancel the HTTP request before it ends, remember I set up a fake slowness of ten seconds; it should cancel the SQL query as the HTTP query is cancelled. Figure 7-1 shows the cancellation handled by SQL, which returns a *SqlException*.

Figure 7-1. *The SQL exception generated after the cancellation*

The same cancellation process is available in any asynchronous task, on Entity Framework Core queries, like `ToListAsync(cancella tionToken)`, `FirstOfDefaultAsync(cancellationToken),` etc., and even the HTTP requests managed with the *IHttpClientFactory* or Refit. Listing 7-4 shows the *MediaRepository* class implemented with the *IHttpClientFactory*.

Listing 7-4. The MediaRepository class

```
using Domain.Repositories;

namespace Infrastructure.Http.Repositories;

public class MediaRepository : IMediaRepository
```

308

```
{
    private readonly IHttpClientFactory _httpClientFactory;

    public MediaRepository(
            IHttpClientFactory httpClientFactory)
    {
        _httpClientFactory = httpClientFactory;
    }

    public async Task<(
                byte[] Content,
                string MimeType)> GetCountryFlagContent(
                string countryShortName,
                CancellationToken cancellationToken)
    {
        byte[] fileBytes;

        using HttpClient client = _httpClientFactory
                                    .CreateClient();
        fileBytes = await client
.GetByteArrayAsync($"https://anthonygiretti.blob.core.windows.
net/countryflags/{countryShortName}.png",
                cancellationToken);

        return (fileBytes, "image/png");
    }
}
```

Listing 7-5 shows how to handle the cancellation with the
CancellationToken with Refit on the *IMediaRepository* interface.

Listing 7-5. The IMediaRepository interface

```
using Refit;

namespace Domain.Repositories;

public interface IMediaRepository
{
    [Get("/countryflags/{countryShortName}.png")]
    Task<byte[]> GetCountryFlagContent(
    string countryShortName,
    CancellationToken cancellationToken);
}
```

Relatively simple, isn't it? I only introduced you to the minimum you must know about asynchronous programming, but it's a vast topic. If you want to learn more about it, I suggest you read Stephen Cleary's book on concurrency. It's a great book to learn all the facets of asynchronous programming and more: https://stephencleary.com/book/.

In the following subsection, we'll look at more advanced cancellation management, using *BackgroundTask* classes to handle long-running background tasks, a great feature of ASP.NET Core.

Long-Running Tasks with Background Services

ASP.NET Core provides us with all the tools we need to efficiently execute any type of long-running background tasks directly hosted in our web application, thanks in particular to the *IHostedService* interface available in the *Microsoft.Extensions.Hosting* namespace by downloading the Nuget package *Microsoft.Extensions.Hosting.Abstractions*.

This interface won't be invoked directly; you'll need to implement the abstract BackgroundService class derived from it.

The interface, and hence the *BackgroundService* class, defines three methods:

1. StartAsync

2. StopAsync

3. ExecuteAsync

It's up to us to call the *ExecuteAsync* method. Still, the *StartAsync* and *StopAsync* methods will be executed, respectively, at application startup and shutdown, automatically without any intervention on our part. However, if we wish to perform specific actions, we can override them, as they are defined as abstract by the *BackgroundService* class.

In this section, we'll focus on the *ExecuteAsync* method, in which we'll execute long operations. Let's take a look at the *CountryFileIntegrationBackgroundService* class, which will enable us to manage the ingestion of files downloaded from the server as shown in Listing 7-6.

Listing 7-6. The CountryFileIntegrationBackgroundService class skeleton

```
using Microsoft.Extensions.Hosting;

namespace Infrastructure.BackgroundTasks;

public class CountryFileIntegrationBackgroundService :
BackgroundService
{
    public CountryFileIntegrationBackgroundService()
    {

    }
```

```
    protected override async Task ExecuteAsync(
    CancellationToken cancellationToken)
    {
        while (!cancellationToken.IsCancellationRequested)
        {
                // Do some job
        }
    }
}
```

As you may have noticed, this is only the skeleton of the background task. The latter doesn't do anything, but it does stop if the cancellation of the task is requested. As it's a background task, it won't stop if we close the browser but will if the ASP.NET Core application stops. It's perfectly possible to inject a service that will then perform processing. Still, it's a good idea to create a specific scope for the background task, as it works as a Singleton with a single service instance, unlike HTTP requests, which require a new instance for each service whose lifetime is of type Scoped. Since most of the time you'll be using Scoped instances rather than Singletons, we will inject the *IServiceProvider* interface into the background task, enabling us to fetch any type of service instance or repository. Let's consider the *ICountryService* exposing the *IngestFile* method for manipulating a file from its content (*stream*), which I'm going to instantiate from the *IServiceProvider* interface by creating a temporary scope (disposable after use) as shown in Listing 7-7.

Listing 7-7. The CountryFileIntegrationBackgroundService class enhanced with IServiceProvider

```
using Domain.Services;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
```

```csharp
namespace Infrastructure.BackgroundTasks;

public class CountryFileIntegrationBackgroundService :
BackgroundService
{
    private readonly IServiceProvider _serviceProvider;

    public CountryFileIntegrationBackgroundService(
    IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    protected override async Task ExecuteAsync(
    CancellationToken cancellationToken)
    {
        while (!cancellationToken.IsCancellationRequested)
        {
            using (var scope = _serviceProvider.CreateScope())
            {
                var service = scope.ServiceProvider
                .GetRequiredService<ICountryService>();
                // await service.IngestFile();
            }
        }
    }
}
```

You're probably wondering how our continuously running background task will receive data from an ASP.NET Core application running independently. Well, since the background task and the API share the same process, they're actually part of the same application, so these two subsets of our web application can communicate with each other!

Indirectly, of course, but via messages internal to the application. These two subsets share the same codebase, the same startup file (*Program.cs*), and the same configuration (*appsettings.json*). This is made possible by a .NET feature called *Channels*! *Channels* are part of the *System.Threading. Channels* namespace. This namespace exposes functionalities enabling you to publish a message in a *Channel*, which will be received by another part of the application to read the messages sent in the same *Channel* (Figure 7-2).

ASP.NET Core application
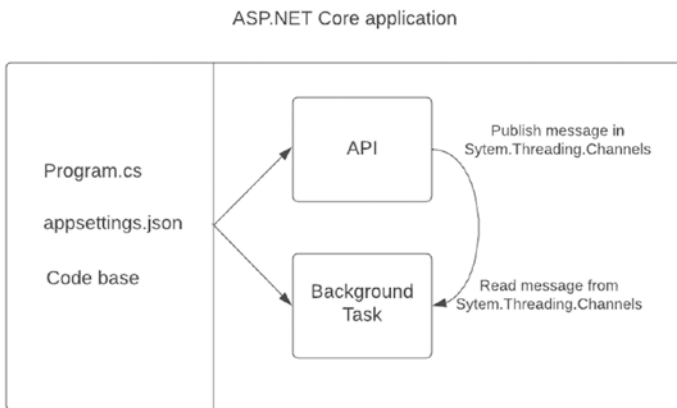
Figure 7-2. *The background task running in the ASP.NET Core application*

For your information, I've created the background task in a separate layer, Infrastructure.BackgroundTasks, still in the spirit of separating technological responsibility. I'm now going to create a *Channel* to push the content of a file (*stream*) as a message. There are different ways of doing this; I will take the simplest one. I'm going to create the interface to push a message into a Channel in the Domain layer, which doesn't change in any way compared with other chapters, and then implement this *Channel*, either directly in the API, as the endpoint in charge of uploading will send the file content directly into the *Channel*. It is also possible to implement the same thing in the BLL, by creating a specific service, which I haven't chosen to do here. Listing 7-8 shows the *ICountryFileIntegrationChannel* interface.

314

Listing 7-8. The ICountryFileIntegrationChannel interface

```
namespace Domain.Channels;

public interface ICountryFileIntegrationChannel
{
    IAsyncEnumerable<Stream> ReadAllAsync(
    CancellationToken cancellationToken);
    Task<bool> SubmitAsync(
    Stream twilioRouteProgrammerParameters,
    CancellationToken cancellationToken);
}
```

The publish method, *SubmitAsync*, is relatively straightforward, while the *ReadAllAsync* method reads all messages asynchronously with the *IAsyncEnumerable<T>* return type where *T* is a *stream*. Still, it allows us to consume messages individually as soon as they become available. Each message is a *Stream* object. Listing 7-9 shows the implementation of the *Channel*, the *CountryFileIntegrationChannel* class.

Listing 7-9. The CountryFileIntegrationChannel class

```
using Domain.Channels;
using System.Threading.Channels;

namespace AspNetCore8MinimalApis.Channels;

public class CountryFileIntegrationChannel :
ICountryFileIntegrationChannel
{
    private readonly Channel<Stream> _channel;
```

```csharp
    public CountryFileIntegrationChannel()
    {

        var options = new UnboundedChannelOptions
        {
            SingleWriter = false,
            SingleReader = true
        };

        _channel = Channel.CreateUnbounded<Stream>(options);
    }

    public async Task<bool> SubmitAsync(
    Stream fileContent,
    CancellationToken cancellationToken)
    {
        while (await _channel.Writer.WaitToWriteAsync
        (cancellationToken) && !cancellationToken.
        IsCancellationRequested)
        {
            if (_channel.Writer.TryWrite(fileContent))
            {
                return true;
            }
        }

        return false;
    }

    public IAsyncEnumerable<Stream>
    ReadAllAsync(CancellationToken cancellationToken) =>
    _channel.Reader.ReadAllAsync(cancellationToken);
}
```

The first thing to note here is the class constructor. I created and configured an *UnBoundedChannel* object. This means I'm creating a *Channel* that can receive unlimited messages because they're treated like a queue, processed one by one. Processing messages in a background task, one by one, guarantees that your application won't crash because too many tasks are executed simultaneously. Then I configured the *Channel* with the *UnboudedChannelOptions* class, which will indicate with the `SingleWriter = false` property that several publishers can publish simultaneously in the *Channel*, a publisher being an HTTP request.

On the other hand, with the `SingleReader = true` property, I indicated that there is only one simultaneous reader, and this single reader is our background task. The *SubmitAsync* method will attempt to publish the *Stream* object in the *Channel* using the *TryWrite* method and return true if it worked and false if it failed. To date, I've never had a case where publication failed. The only possibility is setting your *Channel* with the *BoundChannelOptions* option class, which allows you to limit the number of events in a queue. I keep this method for safety. The whole thing is also controlled by the *WaitToWriteAsync* method, which checks that it's possible to write a message to the Channel before doing so. This method takes a CancellationToken as a parameter. Finally, I checked that a cancellation (the application stops) has been initiated, with the *CancellationToken's IsCancellationRequested* property, in which case it doesn't publish the message in the *Channel*. If we go back to the *CountryFileIntegrationBackgroundService* class, we can now inject and integrate the *ICountryFileIntegrationChannel* interface as shown in Listing 7-10.

Listing 7-10. The CountryFileIntegrationBackgroundService class with the CountryFileIntegrationChannel injected

```
using Domain.Channels;
using Domain.Services;
using Microsoft.Extensions.DependencyInjection;
```

```
using Microsoft.Extensions.Hosting;

namespace Infrastructure.BackgroundTasks;

public class CountryFileIntegrationBackgroundService :
BackgroundService
{
    private readonly ICountryFileIntegrationChannel _channel;
    private readonly IServiceProvider _serviceProvider;
    public CountryFileIntegrationBackgroundService(
    ICountryFileIntegrationChannel channel,
    IServiceProvider serviceProvider)
    {
        _channel = channel;
        _serviceProvider = serviceProvider;
    }

    protected override async Task ExecuteAsync(
    CancellationToken cancellationToken)
    {
        await foreach (var fileContent in _channel.ReadAllAsync
        (cancellationToken))
        {
            try
            {
                using (var scope = _serviceProvider
                .CreateScope())
                {
                    var service = scope.ServiceProvider
                    .GetRequiredService<ICountryService>();
                    await service.IngestFile(fileContent);
                }
            }
```

```
        catch { }
    }
  }
}
```

Thanks to the *IAsyncEnumerable* collection returned by the *ReadAllAsync* method, we can iterate through the messages passed to the *Channel* one by one as soon as a message is available. If we want to make the whole thing work, let's go to the *Program.cs* file and configure the *Channel* and the *BackgroundService* in the dependency injection system as shown in Listing 7-11.

Listing 7-11. The registration of the CountryFileIntegrationBackgroundService class and the ICountryFileIntegrationChannel

```
builder.Services.AddSingleton<ICountryFileIntegrationChannel,
CountryFileIntegrationChannel>();
builder.Services.AddHostedService<CountryFileIntegrationBackgro
undService>();
```

The *ICountryFileIntegrationChannel* **must be** registered as *Singleton* lifetime because on the reader side, to be able to read messages, it must read messages on the **same** *ICountryFileIntegrationChannel* instance; else, it won't work, and messages will get lost. The *CountryFileIntegrationBackgroundService* must be registered with the *AddHostedService* extension method**.** If you remember, we set up the CancellationToken in the *CountryFileIntegrationChannel* class and the *CountryFileIntegrationBackgroundService* class. So if any cancellation (the application shuts down) occurs, we will let the time for any background task that is still running complete. To do this, we can set up, in the *Program.cs* file, the *ShutdownTimeout* to 60 seconds to let any in-progress processes complete during this period, as shown in Listing 7-12.

Listing 7-12. Set up the ShutdownTimeout to 60 seconds

```
builder.Services.PostConfigure<HostOptions>(option =>
{
    option.ShutdownTimeout = TimeSpan.FromSeconds(60);
});
```

Let's write the POST /countries/upload endpoint that accepts a file and passes it to the *ICountryFileIntegrationChannel* service as shown in Listing 7-13.

Listing 7-13. Set up the ICountryFileIntegrationChannel interface on the POST /countries/upload endpoint

```
app.MapPost("/countries/upload", async (IFormFile file,
ICountryFileIntegrationChannel channel, CancellationToken
cancellationToken) =>
{
    if (await channel.SubmitAsync(
        file.OpenReadStream(),
        cancellationToken))

        Results.Accepted();

    Results.StatusCode(StatusCodes.
    Status500InternalServerError);
    }).DisableAntiforgery();
```

The good practice is to return the Accepted (202) status that tells the client the server accepted the request for processing (and has not been completed). Otherwise, an Internal Error (500) should be returned.

Figure 7-3 shows the *CountryFileIntegrationBackgroundService* task execution when a message is posted in the *Channel*.

```
0 references
protected override async Task ExecuteAsync(CancellationToken cancellationToken)
{
    await foreach (var fileContent in _channel.ReadAllAsync(cancellationToken))
    {
        try
        {
            using (var scope = _serviceProvider.CreateScope())
            {
                var service = scope.ServiceProvider.GetRequiredService<ICountryService>();
                await service.IngestFile(fileContent);   ≤ 2ms elapsed
            }                                    ▶  fileContent    {Microsoft.AspNetCore.Http.ReferenceReadStream}
        }
        catch { }
    }
}
```

Figure 7-3. *The CountryFileIntegrationBackgroundService task execution*

Remarkable, isn't it? Remember that these background tasks are helpful for long operations so as not to leave your client waiting for a long process before getting a response from the server. You can declare as many background tasks as you like, but you must create a dedicated Channel for each background task so that the message corresponds to the background task for processing.

Paging

The primary purpose of an API is to expose data to customers, enabling them to read, modify, or delete information. However, when the volume of data becomes significant, and you need to expose a route returning a collection, it can be practical to paginate this information. If the client uses a limited connection, as in the case of a mobile application, volumetry will be an essential performance factor.

So I will suggest a quick and easy way to set up paging.

First, look at the GET /countries endpoint, which returns a collection of countries, and add two parameters in the query string: *pageIndex* and *pageSize*. The pageIndex parameter is the index of the data sequence we

want to query, and the pageSize is the amount of data we want to query per page. Figure 7-4 shows a collection of ten elements paged with two sequences of five items.
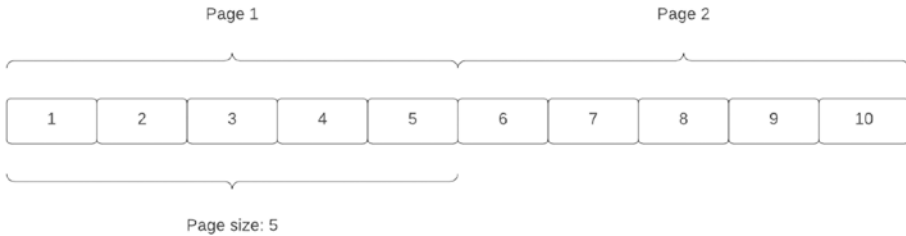
Figure 7-4. *A collection of ten elements paged with a size of five items per page*

To implement this, let's see how the code looks like as shown in Listing 7-14.

Listing 7-14. The GET /countries endpoint with paging query parameters

```
app.MapGet("/countries", async (
            int? pageIndex,
            int? pageSize,
            ICountryMapper mapper,
            ICountryService countryService) => {
    var countries = await countryService
    .GetAllAsync(
    new PagingDto {
        PageIndex = pageIndex.HasValue ? pageIndex.Value : 1,
        PageSize = pageSize.HasValue ? pageSize.Value : 10
    });
    return Results.Ok(mapper.Map(countries));
});
```

> **Note**   If you have too many query parameters, you can encapsulate them into an object to make the code more readable by using the *AsParameters* attribute, which gives the following: `app.MapGet` `("/countries",` async **([AsParameters]** `Paging paging, ...)`.

It's easy to retrieve the parameters—they're automatically bound when they come from the query string. Don't forget that query string parameters are not mandatory and must be treated as nullable. Then we pass them into an instance of the *PagingDto* class whose signature is identical to the parameters retrieved from the query string, as shown in Listing 7-15.

Listing 7-15. The PagingDto class

```
namespace Domain.DTOs;

public class PagingDto
{
    public int PageIndex { get; set; } = 1;
    public int PageSize { get; set; } = 10;
}
```

As you can see, I've given each of them a default value in case they're not set up.

Let's go back to the LINQ query, the *GetAllAsync* method we saw in Chapter 6, which offers two paging methods. The first is *Skip*, which is the index at which the query will start retrieving results, and the second is *Take*, which is the number of elements the query will retrieve. Listing 7-16 shows the paged query. Of course, paging is done on the SQL side, not on the client side, that is, we don't retrieve everything on the code side and then take a fraction of it, as this would be much less efficient.

Listing 7-16. The GetAllAsync method paged

```
public async Task<List<CountryDto>>
GetAllAsync(PagingDto paging)
{
    return await _demoContext.Countries
                .AsNoTracking()
              .Select(x => new CountryDto
              {
                  Id = x.Id,
                  Name = x.Name,
                  Description = x.Description,
                  FlagUri = x.FlagUri
              })
              .Skip((paging.PageIndex - 1) * paging.
              PageSize)
                            .Take(paging.PageSize)
              .ToListAsync();
 }
```

You see, it's easy. As far as *Refit* and *IHttpClientFactory* are concerned, it's even more accessible. All you have to do is replace the string following the same pattern as the route parameters.

When you bring back a collection of information, remember to paginate the results when it's long, as this will help maintain good performance.

JSON Streaming

ASP.NET Core 8 optimizes network bandwidth by transmitting items (streaming) to the client individually rather than a whole collection in one shot. This is very practical. A heavyweight client such as C# won't be able

to exploit items received individually, especially with an *HttpClient*. The response will be available when all items are received, just as JavaScript can display items received individually. Let's rewrite our GET /countries endpoint, returning an *IAsyncEnumerable<Country> object* to the client, as shown in Listing 7-17.

Listing 7-17. The GET /countries endpoint returning an IAsyncEnumerable<Country> object

```
app.MapGet("/countries", async (
            int? pageIndex,
            int? pageSize,
            ICountryMapper mapper,
            ICountryService countryService) => {
    async IAsyncEnumerable<Country> StreamCountriesAsync()
    {
        var countries = await countryService
        .GetAllAsync(
        new PagingDto
        {
         PageIndex = pageIndex.HasValue ? pageIndex.Value : 1,
         PageSize = pageSize.HasValue ? pageSize.Value : 10
        });
        var mappedCountries = mapper.Map(countries);
        foreach (var country in mappedCountries)
        {
            yield return country;
        }
    }
    return StreamCountriesAsync();
});
```

I made a video to show you how it works when streamed into a
JavaScript client; you can find the demo on my blog here: `https://`
`anthonygiretti.com/2021/09/22/asp-net-core-6-streaming-json-`
`responses-with-iasyncenumerable-example-with-angular/`.

Caching

Caching is a technique for storing frequently used information in memory
to avoid having to regenerate the same data at a later date.

Caching is particularly important for data coming from requests made
to a data source, as accesses to the database (out of any other data source)
are generally very costly regarding response time. I will show you how to
implement caching in ASP.NET Core to avoid this. ASP.NET Core offers
three types of caching:

1. **HTTP cache (output cache)**: Data is cached on any
   proxy servers or in the web browser.

2. **In-memory cache**: Data is stored in the
   server's RAM.

3. **Distributed cache**: Data is cached on an external
   server to which multiple applications can connect.

Output Cache

The *OutputCache* is very effective but very limited. The cache can be
stored on the proxy server or in the browser, depending on whether or
not a proxy exists. This cache only applies to **GET and HEAD requests
that return a 200 response**, which are only cached if they don't generate
a cookie and don't require authentication (identified by the presence of
the Authorization header). The other disadvantage is that the browser or a
proxy keeps data since the request doesn't reach the server because all the

HTTP response is cached. You won't be able to log any user's actions on the server because you want, for example, to generate statistics using your API. You should use this cache when you don't have any requirements, such as collecting statistics, in your application or don't have any authentication on endpoints you want to cache with (without omitting that only successful GET and HEAD requests are cacheable). Figure 7-5 summarizes the *OutputCache* workflow.
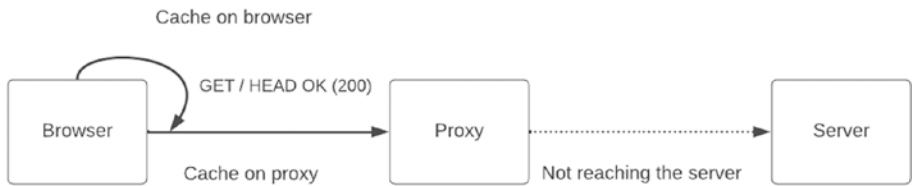
Figure 7-5. The OutputCache concept

To begin with, you need to configure your cache with the *AddOutputCache* extension method and define policies. These can be global, using *AddBasePolicy*, and apply to all requests eligible for the output cache. Alternatively, you can use a named policy to be explicitly called on your endpoints. To activate all this, you need to declare the *UseOutputCache* middleware.

Listing 7-18 shows the configuration of the *Program.cs* file.

Listing 7-18. The output cache configuration in the Program.cs file

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

...

builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder =>
      builder.Expire(TimeSpan.FromSeconds(30))
             .SetVaryByQuery("*")
```

```
    );
  options.AddPolicy("5minutes", builder =>
      builder.Expire(TimeSpan.FromSeconds(300))
            .SetVaryByQuery("*")
    );
});
```

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

```
app.UseOutputCache();
....
```

```
app.Run();
```

As you can see, this configuration contains a global policy that allows you to define a 30-second cache, which applies to all endpoints that don't define an explicit policy and applies only to a single URL, considering its query string parameters. There will be as many cached data as there are variations of the same URL with different parameters, for example, the *URL /countries?pageIndex=1* and */countries?pageIndex=2* will have different cached data.

Then I've set up a policy called "5minutes" that you'll need to apply specifically to the endpoints you want, which caches data for five minutes and also varies according to query string parameters. Listing 7-19 shows the endpoint GET / cachedcountries using the "5minutes" policy by the usage of the *CacheOutput* extension method.

Listing 7-19. The GET /cachedcountries endpoint using the "5minutes" cache policy

```
app.MapGet("/cachedcountries", async (
          int? pageIndex,
          int? pageSize,
          ICountryMapper mapper,
```

```
        ICountryService countryService) => {
    var countries = await countryService
    .GetAllAsync(new PagingDto
    {
        PageIndex = pageIndex.HasValue ? pageIndex.Value : 1,
        PageSize = pageSize.HasValue ? pageSize.Value : 10
    });
    return Results.Ok(mapper.Map(countries));
}).CacheOutput("5minutes");
```

If we execute this endpoint the first time (or if you are the first person who accesses this endpoint among other users), you won't notice anything, the response will be absolutely normal, and no headers will be set up in the response. Still, if the endpoint is cached because somebody has accessed it before you, you'll notice the presence of the *Age* header, which tells the client how long the data has been in the cache. Figure 7-6 shows the GET /cachedcountries endpoint after its execution, and the response shows data that have been cached for 15 seconds.
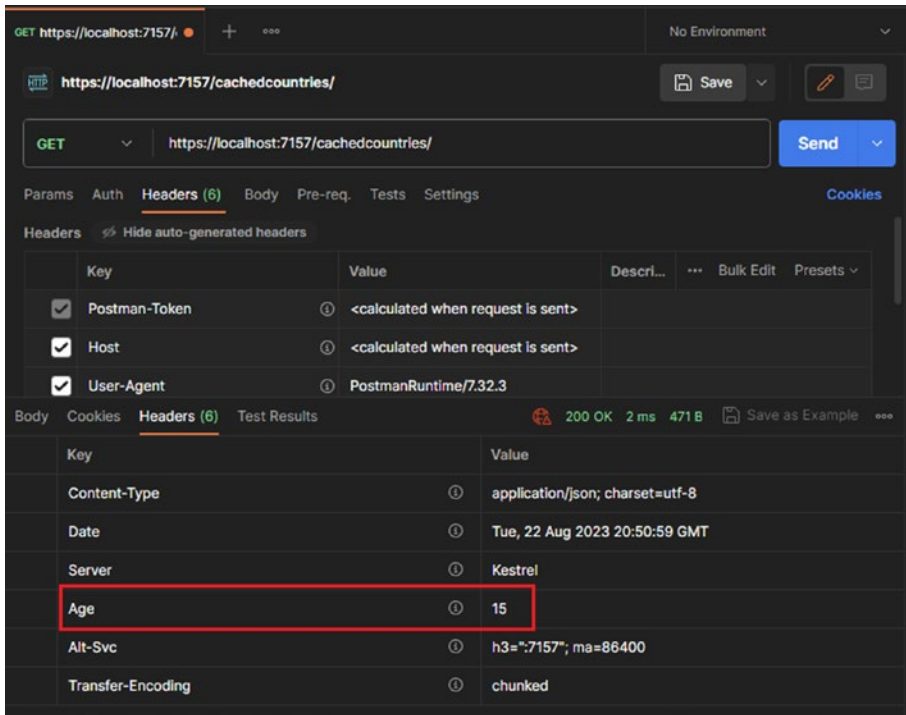
Figure 7-6. *The GET /cachedcountries endpoint output cache*

I won't go into detail here since this isn't the cache you'll often use because of its limitations. I wanted to show you its existence, and I think that if you use it, it will be to a lesser extent than I've just shown you. However, if you want to know more, you can consult Microsoft's documentation: https://learn.microsoft.com/en-us/aspnet/core/performance/caching/output?view=aspnetcore-8.0.

In-Memory Cache

This cache type is preferable to output cache, as it uses only the data you decide to cache. Here, the entire HTTP response is not cached. Still, the request reaches the server whether there is a proxy between the client and the server or not, allowing you to keep control over what happens during

the request, such as, once again, collecting statistics, logging user actions, and so on. This is the most frequently used cache when your application is not distributed, that is, when a single server exposes your API, as the in-memory cache is server-specific. The cache will be duplicated on each server if you have several servers. This happens when you have a server farm where HTTP traffic is load-balanced on this or that server. In this case, we'll talk about a distributed cache, and I'll return to this in the following subsection. Figure 7-7 summarizes the *in-memory* cache workflow.
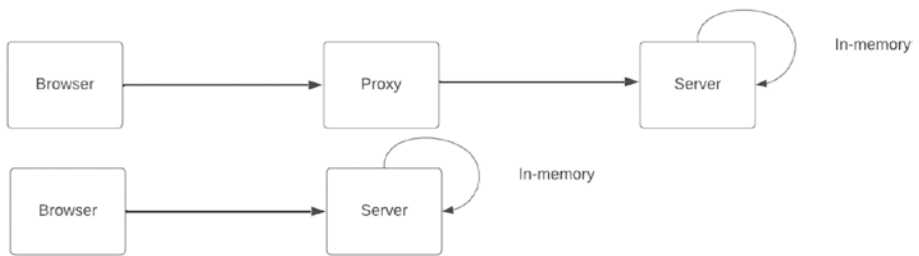
Figure 7-7. *The in-memory cache concept*

In this section, I'm going to show you two things:

1. How to use the in-memory cache.

2. For practical reasons of architecture and coding, the Single Responsibility principle (SRP) exactly, I'm going to use the *Decorator* pattern.

Why the Decorator pattern? Because I've created the *ICountryService* interface with its *CountryService* implementation that calls the database to retrieve data. But it's not its responsibility to fetch cached information. The *Decorator* pattern dynamically attaches additional responsibilities to an object. It provides a flexible alternative to inheritance for extending functionality.

We'll create a decorator class to decorate the *CountryService* class and call it *CachedCountryService*. This *CachedCountryService* class inherits from the *ICountryService* interface and is also injected by dependency on the *ICountryService* interface. As a result, this class will have its implementation while invoking the implementation of the *CountryService* class. For this to work, the dependency injection system must be configured to indicate which class is decorating the decorated class. Here, the *CountryService* class is decorated by the *CachedCountryService* class, which manages the cache.

We will create the CachedCountryService class in the BLL and need the *IMemoryCache* interface, which can be found in the Nuget package *Microsoft.Extensions.Caching.Abstractions*. We only need the *IMemoryCache* interface here, so we're downloading the version containing only the in-memory cache abstractions. In the API layer, on the other hand, we're going to download the version containing the entire in-memory cache implementation, which we'll need to activate the cache middleware. In the API layer, please install the following Nuget package: *Microsoft.Extensions.Caching.Memory*. Listing 7-20 shows the implementation of the *CachedCountryService* decorator class, including the implementation of caching using *IMemoryCache*.

Listing 7-20. The CachedCountryService decorator class

```
using Domain.DTOs;
using Domain.Services;
using Microsoft.Extensions.Caching.Memory;

namespace BLL.Services;

public class CachedCountryService : ICountryService
{
    private readonly ICountryService _countryService;
    private readonly IMemoryCache _memoryCache;
```

```
public CachedCountryService(
ICountryService countryService,
IMemoryCache memoryCache)
{
    _countryService = countryService;
    _memoryCache = memoryCache;
}

public async Task<List<CountryDto>> GetAllAsync(
PagingDto paging)
{
    var cachedValue = await _memoryCache.GetOrCreateAsync(
    $"countries-{paging.PageIndex}-{paging.PageSize}",
    async cacheEntry =>
    {
        cacheEntry.AbsoluteExpirationRelativeToNow =
        TimeSpan.FromSeconds(30);
        return await _countryService.GetAllAsync(paging);
    });

    return cachedValue;
}

public async Task<(byte[], string, string)> GetFileAsync()
{
    return await _countryService.GetFileAsync();
}

public async Task<bool> IngestFileAsync(
Stream countryFileContent)
{
    return await _countryService
    .IngestFileAsync(countryFileContent);
}
```

```
    public async Task LongRunningQueryAsync(
    CancellationToken cancellationToken)
    {
        await _countryService
        .LongRunningQueryAsync(cancellationToken);
    }
}
```

As you can see, I've implemented all the *ICountryService* interface methods because we must. However, I'm not reimplementing all the methods; I'm reusing all the *ICountryService* methods as is, that is, the methods of the decorated class (*CountryService*), because I'm only going to apply caching to the *GetAllAsync* method. I'm therefore reusing the original implementation (*CountryService*) with cache using the *GetOrCreateAsync* method, which will either create the cache key with its value if the latter doesn't exist in memory or retrieve it based on the key `$"countries-{paging.PageIndex}-{paging.PageSize}"`. This key must be unique and parameterized according to all the parameters passed to the *GetAllAsync* method. This is the only way to ensure the uniqueness of cached content in a given situation. I've set the cache duration to 30 seconds using the *AbsoluteExpirationRelativeToNow* method.

The first time the method is invoked, or when the cache is expired, the *_countryService.GetAllAsync* method is invoked again. Using the *SetSlidingExpiration* method to control cache duration instead of the *AbsoluteExpirationRelativeToNow* method is possible, but I don't recommend it, as its operation is different. Cached elements won't be refreshed until the cache is invoked for x amount of time, so you see the problem: as long as cached data is requested, its contents won't be refreshed. I never use it unless I'm sure that the cached data will never be modified, which is rarely the case. Listing 7-21 shows the configuration of the in-memory cache, the Decorator pattern for the ICountryService interface, and the GET /cachedinmemorycountries endpoint.

Listing 7-21. The in-memory cache and Decorator pattern
configuration

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

...
builder.Services.AddScoped<ICountryService, CountryService>();
builder.Services.Decorate<ICountryService,
CachedCountryService>();

builder.Services.AddMemoryCache();

...

var app = builder.Build();

...

app.MapGet("/cachedinmemorycountries", async (
ICountryMapper mapper,
ICountryService countryService) => {
    var countries = await countryService
    .GetAllAsync(new PagingDto
    {
        PageIndex = 1,
        PageSize = 10
    });
    return Results.Ok(mapper.Map(countries));
});

...

app.Run();
```

As you can see, it's easy to configure.

Simply configure the in-memory cache with the *AddMemoryCache* extension method and configure the CachedCountryService decorator class of the *CountryService* class with the *Decorate* extension method, which you can find in the Nuget *Scrutor* package. This allows you to configure the Decorator pattern in your application quickly. Then, I implemented the GET /cachedinmemorycountries endpoint to access the cache using the *GetAllAsync* method.

Note   At this time of writing, I faced issues with the *Scrutor* Nuget package, which doesn't work correctly with the .NET 8 preview 7. It should be fixed in the final version of .NET 8. For your information, you can read the details of the issue here: https://github.com/khellang/Scrutor/issues/208.

If you want to know more about design patterns, such as the Decorator pattern, you can read Fiodar Sazanavets' book about design patterns here: https://leanpub.com/the-easiest-way-to-learn-design-patterns. It's a great book. I learned a lot from it!

Distributed Cache

Distributed caching extends the traditional caching concept, where data is placed locally in temporary storage for rapid retrieval. A distributed cache is more extensive, as it is not located on the web server itself but on another machine or in a cloud service such as Microsoft Azure. Distributed caching can be implemented on top of different providers such as

1. SQL Server (distributed cache on SQL Server)

2. Redis (non-SQL in-memory database)

3. In-memory distributed cache

4. NCache (open source in-memory cache)

All meet the same criteria: they are outsourced to another server, as shown in Figure 7-8.
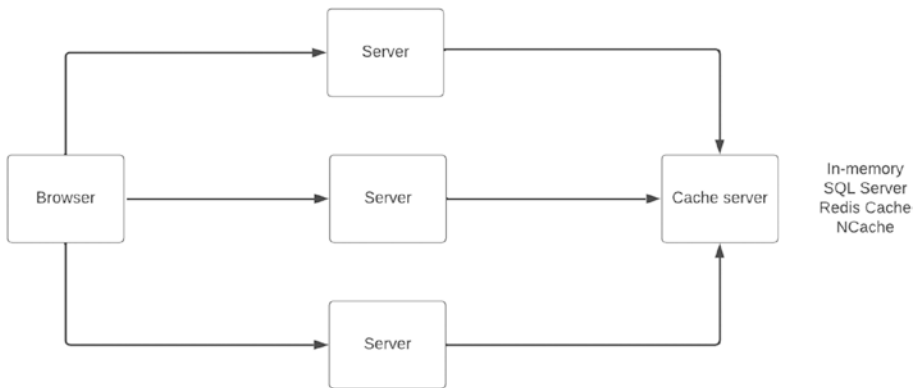
Figure 7-8. *The distributed cache concept*

I'll take Redis as an example here. Thanks to its caching algorithms, it is the most powerful caching database. It's also the most widely used caching database for distributed, high-traffic applications. To begin with, I will invite you to create an instance of Redis. I'm not going to document it here, but you can follow the following tutorial offered by Microsoft, which will enable you to create an instance in Microsoft Azure: `https://learn.microsoft.com/en-us/azure/azure-cache-for-redis/cache-configure`.

In terms of implementation, we're going to use the same principle as before, namely, to use the Decorator pattern to implement the *DistributedCachedCountryService* class, which will be injected with the *IDistributedCache* interface, present in the same Nuget package as the in-memory cache: *Microsoft.Extensions.Caching.Abstractions*. For the API layer, please download the Nuget package *Microsoft.Extensions.Caching.StackExchangeRedis* that registers an instance of the *RedisCache* class implementing the *IDistributedCache* interface. Listing 7-22 shows the implementation of the *DistributedCachedCountryService* decorator class.

337

Listing 7-22. The DistributedCachedCountryService decorator class

```
using Domain.DTOs;
using Domain.Services;
using Microsoft.Extensions.Caching.Distributed;
using System.Text.Json;

namespace BLL.Services;

internal class DistributedCachedCountryService :
ICountryService
{
    private readonly ICountryService _countryService;
    private readonly IDistributedCache _distributedCache;

    public DistributedCachedCountryService(
    ICountryService countryService,
    IDistributedCache distributedCache)
    {
        _countryService = countryService;
        _distributedCache = distributedCache;
    }

    public async Task<List<CountryDto>>
    GetAllAsync(PagingDto paging)
    {
        var key = $"countries-{paging.PageIndex}-{paging.
        PageSize}";

        var cachedValue = await _distributedCache
        .GetStringAsync(key);
        if (cachedValue == null)
        {
            var data = await _countryService
```

```
        .GetAllAsync(paging);
        await _distributedCache
        .SetStringAsync(key,
        JsonSerializer.Serialize(data),
        new DistributedCacheEntryOptions
        {
            AbsoluteExpirationRelativeToNow = TimeSpan.
            FromSeconds(30)
        });
        return data;
    }
    return JsonSerializer
    .Deserialize<List<CountryDto>>(cachedValue);
}

public async Task<(byte[], string, string)> GetFileAsync()
{
    return await _countryService.GetFileAsync();
}

public async Task<bool> IngestFileAsync(
Stream countryFileContent)
{
    return await _countryService
    .IngestFileAsync(countryFileContent);
}

public async Task LongRunningQueryAsync(
CancellationToken cancellationToken)
{
    await _countryService
    .LongRunningQueryAsync(cancellationToken);
}
}
```

As you can see, we can store any object. Still, in string form, I'm obliged to serialize/deserialize in JSON the list of countries I want to store/ retrieve to/from Redis with the *Serialize/Deserialize* methods from the *System.text.Json* assembly. As far as Redis is concerned, the *SetStringAsync* and *GetStringAsync* methods are, respectively, responsible for storing and retrieving data. The cache duration can be configured with the *DistributedCacheEntryOptions* option class, always with a relative duration known in advance. At the end of this duration, Redis automatically purges the cache key. The operation here is identical to that of the in-memory cache.

Listing 7-23 shows the configuration of the distributed cache in the Program.cs file.

Listing 7-23. The distributed cache and Decorator pattern configuration

```
...

using Microsoft.Extensions.Caching.StackExchangeRedis;

var builder = WebApplication.CreateBuilder(args);

...

builder.Services.AddScoped<ICountryService, CountryService>();
builder.Services.Decorate<ICountryService,
DistributedCachedCountryService>();

...

builder.Services.AddStackExchangeRedisCache(options =>
{
    options.Configuration = builder.Configuration
   .GetConnectionString("RedisConnectionString");
    options.InstanceName = "Demo";
});
```

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

...

app.MapGet("/cachedinmemorycountries", async (
            ICountryMapper mapper,
            ICountryService countryService) => {
    var countries = await countryService
    .GetAllAsync(new PagingDto
    {
        PageIndex = 1,
        PageSize = 10
    });
    return Results.Ok(mapper.Map(countries));
});

...

app.Run();
```

As you can see, the cache is configured using the
AddStackExchangeRedisCache method, which takes two options as
parameters:

1. The Redis connection string, which you'll have set
    up in the appsettings.json file

2. The instance name

The instance name is slightly misleading since the name you gave
will be concatenated to any Redis cache key to avoid key clashes if two
identical keys are inadvertently registered in Redis. While this isn't
necessarily bad if several applications use the same data, it poses a
problem when two applications use the same cache key for entirely
different data.

Now you know the potential of distributed caching with Redis. I invite you to take a closer look at this distributed caching technology because it's very powerful. To learn more about Redis, visit the Redis website here, `https://redis.io/`, where you'll find other examples of implementations with .NET. If you'd like to find out more about distributed caching and its possible implementation with .NET, you can consult the Microsoft documentation here: `https://learn.microsoft.com/en-us/aspnet/core/performance/caching/distributed?view=aspnetcore-8.0#recommendations`.

Speeding Up HTTP Requests with HTTP/2 and HTTP/3

The version of HTTP you need to have in mind is HTTP/1.1. I talked about it in Chapter 1 of this book. HTTP evolved and was completely redefined in 2015 with version 2 (HTTP/2). Without going into too much detail, HTTP/2 is much faster than HTTP/1.1, and ASP.NET Core supports it. However, not all browsers support HTTP/2, so you must be careful here. ASP.NET Core can handle several versions of HTTP at once. To find out more about HTTP/2, consult **RFC 9113** here: `https://datatracker.ietf.org/doc/html/rfc9113`.

If you'd like to check browser compatibility, visit the caniuse.com site for HTTP/2 here: `https://caniuse.com/http2`.

And that's not all! Microsoft has recently proposed a new version, even though all web applications and browsers have not yet adopted HTTP/2. This HTTP/3 version is even faster than HTTP/2. To learn more about HTTP/3, consult **RFC 9114** here: `https://datatracker.ietf.org/doc/rfc9114/`. Of course, HTTP/3 is even less well supported by browsers, but you can check out the evolution of its support here: `https://caniuse.com/http3`. The good news is that ASP.NET Core 8 supports HTTP/3!

How do you get ASP.NET Core 8 to support HTTP/3 and HTTP/2 while still supporting HTTP/1.1 if the client does not support the first two? Listing 7-24 explains how to configure its *appsettings.json* file.

Listing 7-24. The appsettings.json file enabling HTTP/1.1, HTTP/2, and HTTP/3 protocols

```
"Kestrel": {
  "EndpointDefaults": {
     "Protocols": "Http1AndHttp2AndHttp3"
  }
}
```

Summary

I want to congratulate you on following everything I've taught you so far. You've learned everything you need to know to implement well-coded, well-architected, and optimized APIs, along with a host of tips and tricks that will make all the difference! Now that you understand everything you can do to code your application well and make it perform well, we'll turn our attention to monitoring our application. In the next chapter, we'll look at how to perform logging as efficiently as possible, collecting metrics and actions performed on the API (tracing) to see how our application behaves when it's being used. See you in the next chapter!

Introduction to Observability

Observability is the collection, visualization, and analysis of data in an application. This is important when designing an application, as it must be possible to detect undesirable behaviors (service unavailability, errors, slow responses) and provide them with actionable information to effectively determine the cause of the problem (detailed event logs, granular information on resource use with metrics, application traces). In this chapter, with simple examples (because it's a complicated subject, let's make it simple), I'll teach you the following points:

- Basics of observability

- Performing logging

- Performing tracing and metrics data collection

- Implementing HealthChecks

Basics of Observability

When we talk about observability, we talk about *Metrics, Events, Logs, and Traces (MELT).* These are the four pillars of observability in an application:

1) **Logs**: They enable us to collect data on a specific event we've defined. We'll log, for example, an exception that has occurred in the application or other elements that are more or less indicative of what's going on in the application, for example, information that will enable us to debug a situation in a particular context.

2) **Events**: These are actions accompanied by a set of metadata, for example, the action of a user uploading a file of a specific size.

3) **Traces**: They are an overview of the user's path and interaction with the system. More specifically, when a user action triggers an HTTP request, we will trace access to external resources, such as a database, to detect any slowness at this level.

4) **Metrics**: These are used to assess the system's overall health. This could be, for example, CPU or memory usage.

In this chapter, I won't deal with events, which are rarely used on APIs, unlike the collection of metrics, logs, and traces.

Finally, observability relies on tools generally referred to as *Application Performance Monitoring (APM),* which is the practice of observing what's going on in an application. APMs are real-time application management tools aiming to anticipate problems rather than solve them quickly by

providing access to logs, traces, and metrics. In this chapter, I'll use *Application Insights* as an APM and show you how to collect logs, traces, and metrics and read them in Application Insights.

Note   Application Insights is a powerful and extensible Microsoft Azure tool for monitoring your applications. I'll be using this tool in this chapter.

Creating a Microsoft Azure account and setting up an *Application Insights* workspace are prerequisites for understanding this chapter's rest. To do this, you can visit the Microsoft documentation for creating an *Application Insights* workspace: https://learn.microsoft.com/en-us/azure/azure-monitor/app/create-workspace-resource. Once you are done, please pick up the *Application insights* connection string. You will need it further.

Note   In this book, I won't teach you how to use Application Insights, but you can read a nice tutorial on the Microsoft Learn website: https://learn.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview?tabs=net.

Performing Logging

Logging what's going on in your application is essential to detect errors and what kind of errors have occurred. It's also possible to log other contextual information that is not an error but helps debug your application in case of a problem. As you can see, you can log anything you like, except—and I've already mentioned this in connection with the OWASP principles—sensitive information. What is sensitive information? Sensitive information is, for example, personal user data such as email

addresses, telephone numbers, or, even worse, Social Security numbers, information used for identity theft. The same goes for confidential application information, such as string connections, logins, or passwords.

ASP.NET Core natively includes a logger via the *ILogger* interface and contains a default implementation that writes to the console as shown in Figure 8-1, which shows the logs automatically generated when the application is started.

Figure 8-1. *The default ASP.NET Core logs*

As you can see, ASP.NET Core writes certain types of information by default; you can see it in the left column, the *info* word. *Info* corresponds to a log level in a hierarchy of levels that we can filter by level in an APM, *Application Insights*, which I'll show you later. There are six log levels with severity levels ranging from 0 to 5. Here's the list of log levels shown with their enum values:

- **Trace = 0**: Represents a purely informative log requiring no action to be taken.

- **Debug = 1**: Represents a more detailed log aimed at diagnosing a problem.

- **Information = 2**: Represents a normal log to confirm normal software operation. Similar to Trace, but I tend to use it over Trace.

- **Warning = 3**: Represents a log warning of behavior that could potentially generate a bug, thus prompting you to better manage part of the code, by, for example, decorating it with a try/catch block.

- **Error = 4**: Represents a detailed log reporting a bug, for example, an exception raised following an action performed.

- **Critical = 5**: Represents a detailed log of a serious problem preventing the application from running, such as a database connection preventing the entire application from running.

I'm not going to use all the log types in the following examples. I'll use the ones I use most often daily: *Info*, *Error*, and *Critical*.

Before moving on to the examples, I will use another implementation than the default one proposed by Microsoft; I will use the *Serilog* logger, which we'll configure to interface with *ILogger*. What I mean by this is that only configuring the logger will give us a bit of work; invoking the logger with the *ILogger* interface will be identical.

Serilog is a library designed to facilitate application logging in .NET applications. Several sinks are available to store logs on different media (file, console, *Application Insights*, and many more).

The significant difference with other log libraries is that it offers a mechanism for obtaining metadata on events that have occurred. This makes it easier to exploit these logs than plain text logs. Most importantly, this library is compatible with structured logging. I'll return to this with some examples so you'll understand.

To start with, we're going to download two Nuget packages in the layer API:

- Serilog.AspNetCore

- Serilog.Sinks.ApplicationInsights

The first is the *Serilog* implementation for *ILogger*, while the second package sends logs to *Application Insights*.

Now, let's configure our ASP.NET Core application using the *appsettings.json* file, not forgetting to include the string connection to your Application Insights workspace. Listing 8-1 shows the *Serilog* configuration.

Listing 8-1. The Serilog configuration in appsettings.json

```
{
  "Serilog": {
    "Using": [
      "Serilog.Sinks.ApplicationInsights"
    ],
    "MinimumLevel": {
      "Default": "Information",
       "Microsoft.AspNetCore": "Warning"
    },
    "WriteTo": [
      {
        "Name": "ApplicationInsights",
        "Args": {
          "connectionString":
          "{YourApplicationInsightsConnectionString}",
          "telemetryConverter": "Serilog.Sinks.
          ApplicationInsights.TelemetryConverters.
          TraceTelemetryConverter, Serilog.Sinks.
          ApplicationInsights"
        }
      }
    ],
    "Enrich": [ "FromLogContext" ],
    "Properties": {
```

```
      "Application": "DemoAPI"
    }
  }
}
```

A few explanations are in order here. First, you need to tell *Serilog* which sink to use, so I indicate that I want to use it with the *Using* property, Application Insights. Then, I set the minimum level to *Information*. Setting *Information* as the minimum level will allow you to send *Information* up to *Critical* levels in the logs if you remember the hierarchy. This configuration is helpful if you want to limit the level of severity of the logs you want; the fewer logs you have, the more readable they'll be, and the more logs you have, the more logs you'll be drowned in. It's up to you to find the right balance. Don't forget that ASP.NET Core logs automatically, so you're in control. If you want all your logs to log the *Information* level minimally, but you want ASP.NET Core to log *Warning* minimally, you can add the name of the *Microsoft.AspNetCore* assembly to the *Using* property with the *Warning* value. You can add as many assemblies as you like to filter your logs. Next, you need to define the configuration of your Application Insights sink with its connectionString and mention the assemblies required to convert your logs to *Application Insights* format. Enriching your logs with additional metadata is possible using the *Enrich* property. I won't go into detail here, but if you'd like to know more about enriching logs with *Serilog*, you can consult their documentation here: https://github.com/serilog/serilog/wiki/Enrichment. Now let's apply this configuration with the following line of code in the *Programs.cs file*:

```
builder.Host.UseSerilog((context, configuration) =>
configuration.ReadFrom.Configuration(context.Configuration));
```

The API is correctly set up to send logs into Application Insights.

There are two ways to use logs in ASP.NET Core. The first one is to use the logger directly from the *app.Logger* object into minimal endpoints, as shown in Listing 8-2.

Listing 8-2. Using the app.Logger object to log

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

...

app.MapGet("/logging", () => {

    app.Logger.LogInformation("/logging endpoint has been
    invoked.");
    return Results.Ok();
});

...

app.Run();
```

In reality, this is not a good practice for reasons of testability because it's a variable external to the minimal endpoint's lambda. We call this *hoisting*, calling variables external to a delegate, a minimal endpoint's lambda being a delegate.

The best practice here is to use dependency injection and invoke *ILogger<T>* where *T* is a generic type, usually the class in which *ILogger* is invoked, to categorize logs according to their origin. Listing 8-3 shows the *ILogger* interface invocation by dependency.

Listing 8-3. Using the ILogger<T> interface by dependency injection

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

...

app.MapGet("/logging", (ILogger<Program> logger) => {

    logger.LogInformation("/logging endpoint has been
    invoked.");
    return Results.Ok();
});
...

app.Run();
```

Note   *ILogger* is a known type automatically injected by dependency by ASP.NET Core. No particular configuration is required to activate it to inject it by dependency, and it can, therefore, be injected into any class, just like the services and repositories we implemented together in the previous chapters.

It's cleaner and easier to test, and we'll return to this subject in the final chapter of this book.

Now, let's take a look at good logging practices. I mentioned structured logging earlier. Structured logging is recommended for more readable logging with contextual information. It all depends on how you variable your logging text. If you variable your log as shown in Listing 8-4, then your logging is structured.

Listing 8-4. Example of structured logging

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

...

app.MapGet("/countries", async (
        int? pageIndex,
        int? pageSize,
        ICountryMapper mapper,
        ICountryService countryService,
        ILogger<Program> logger) => {

    var paging = new PagingDto
    {
        PageIndex = pageIndex.HasValue ? pageIndex.Value : 1,
        PageSize = pageSize.HasValue ? pageSize.Value : 10
    };
    var countries = await countryService.GetAllAsync(paging);

    using (logger.BeginScope(
        "Getting countries with page index {pageIndex}
        and page size {pageSize}",                 paging.
        PageIndex,
                        paging.PageSize))
    {
        logger.LogInformation(
                    "Received {count} countries from the
                    query", countries.Count);
```

```
        return Results.Ok(mapper.Map(countries));
    }
});

...
app.Run();
```

On *Application Insights,* go to "Transaction Search" and find your log as shown in Figure 8-2.
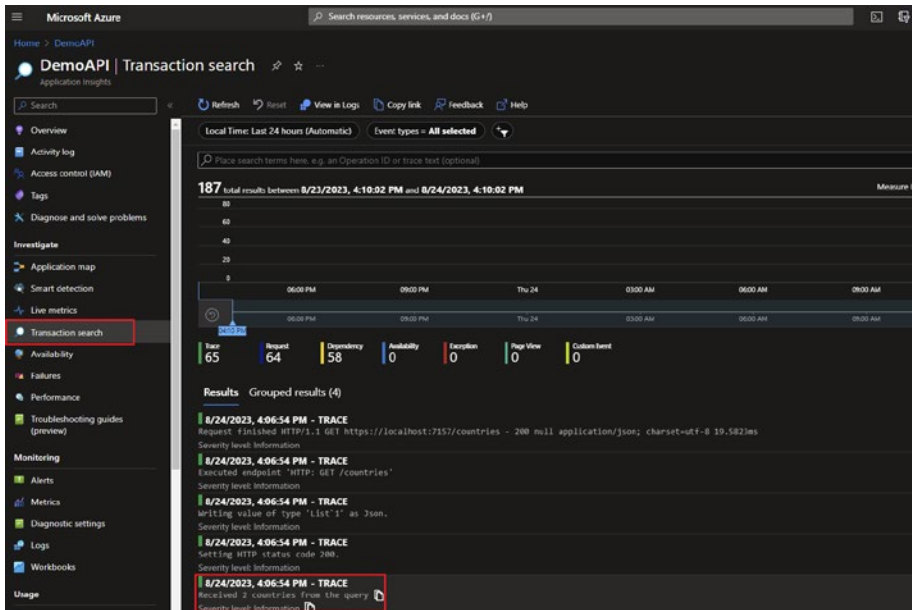
Figure 8-2. Finding logs on Application Insights

Then click it to watch details as shown in Figure 8-3.

Figure 8-3. *Log details with structured logging*

As you can see, the variables (*pageSize*, *pageIndex*, and *count*) in the preceding code sample are well displayed as "Custom Properties." Let's say I use string interpolation for my messages, as shown in Listing 8-5.

Listing 8-5. Using string interpolation

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

...

app.MapGet("/countries", async (
            int? pageIndex,
            int? pageSize,
            ICountryMapper mapper, ICountryService
            countryService, ILogger<Program> logger) => {

    var paging = new PagingDto
    {
        PageIndex = pageIndex.HasValue ? pageIndex.Value : 1,
        PageSize = pageSize.HasValue ? pageSize.Value : 10
    };
    var countries = await countryService.GetAllAsync(paging);

    using (logger.BeginScope(
    "Getting countries with page index {pageIndex} and page
     size {pageSize}",
     paging.PageIndex,
     paging.PageSize))
    {
        logger.LogInformation(
        "Received {count} countries from the query",
        countries.Count);

        return Results.Ok(mapper.Map(countries));
    }
});

...

app.Run();
```

You'll see that the log won't be the same at all and will be significantly less readable, as shown in Figure 8-4.
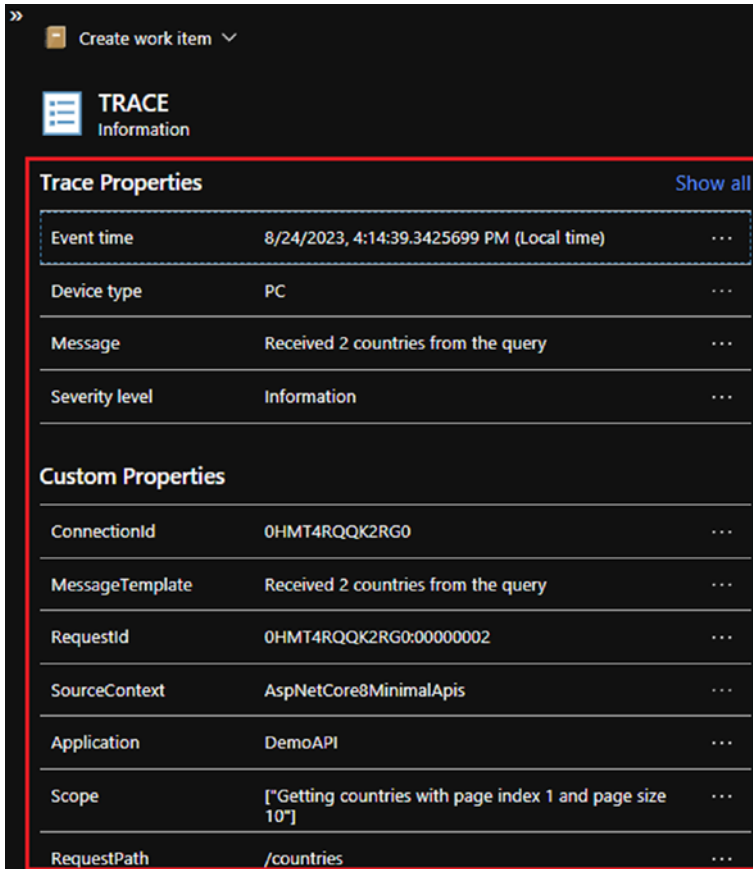
Figure 8-4. *Example of log details without structured logging*

Just a word with the using instruction `using (logger.BeginScope())`: This instruction merges all logs into a single information block spread over an APM. As you saw earlier, thanks to this method, the variables pageSize, pageIndex, and count were not part of the same log but were part of the same scope. Having them grouped in a single block, and therefore a log block in Application Insights, made the log easier to read.

Caution   I suggest you carefully use the *BeginScope* method. If any exception occurs in the *using* statement, any log will be sent to the APM if it is part of the same using block, including the message in the *BeginScope* method.

Here's the last example for logging. I'm going to update the *DefaultExceptionHandler* class, which handles application errors. In the chapter where I introduced error handling, I only handled the response to be sent to the client, but I didn't log the error. Here's the updated *DefaultExceptionHandler* class with the addition of *ILogger*, as shown in Listing 8-6.

Listing 8-6. The DefaultExceptionHandler class enhanced with ILogger

```
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using System.Net;

namespace AspNetCore8MinimalApis.ExceptionHandlers;

public class DefaultExceptionHandler : IExceptionHandler
{
    private readonly ILogger<DefaultExceptionHandler> _logger;
    public DefaultExceptionHandler(ILogger<DefaultExceptionHand
    ler> logger)
    {
        _logger = logger;
    }

    public async ValueTask<bool> TryHandleAsync(
    HttpContext httpContext,
```

```
    Exception exception,
    CancellationToken cancellationToken)
    {
        _logger.LogError(exception,
                    "An unexpected error occurred and has
                    been handled by the
                    {DefaultExceptionHandler} handler", nameo
                    f(DefaultExceptionHandler));

        await httpContext.Response.WriteAsJsonAsync(new
        ProblemDetails
        {
            Status = (int)HttpStatusCode.InternalServerError,
            Type = exception.GetType().Name,
            Title = "An unexpected error occurred",
            Detail = exception.Message,
            Instance = $"{httpContext.Request.Method}
            {httpContext.Request.Path}"
        });

        return true;
    }
}
```

If our application catches an exception, all we have to do is find it in the list of logs, as shown in Figure 8-5.
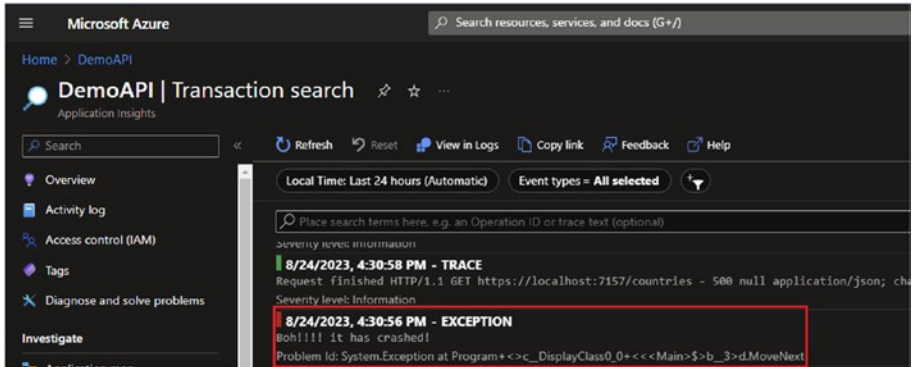
Figure 8-5. *Finding exceptions on Application Insights*

If you click it, you'll be given a whole host of information, including the exact exception we've passed as a parameter to the *LogError* method.

Figure 8-6. *Example of error log details*

After clicking on an exception you will see the error detail as shown in Figure 8-6.

Now, you know almost everything you need to do as a developer. You are now able to configure an APM and set up logs to send to Serilog. I hope this tutorial has informed you of the importance of logs in an application and that you will rigorously practice relevant logging!

Performing Tracing and Metrics Data Collection

Tracing operations on dependencies such as SQL databases and collecting metrics to monitor an application can prove decisive in the event of performance problems when your application goes into production. So as not to be overtaken by events should this happen to you, I'm going to show you how you can easily collect some helpful information on the health of your application. I won't go too far into this topic for the simple reason that it won't be up to you, the developer, to take care of this, by which I mean you'll certainly be asked to implement the data collection, but it might not be up to you to interpret it. So, to make you more aware of the subject, I've prepared a few simple examples to get you started. To get started, download the *Microsoft.ApplicationInsights.AspNetCore* Nuget package in the API layer, go to the *Program.cs* file, and add the following line:

```
builder.Services.AddApplicationInsightsTelemetry();
```

You will also have to add another configuration section in the *appsettings.json* to make the Application Insights telemetry work:

```
"ApplicationInsights": {
  "ConnectionString": "{YourConnectionString}"
}
```

The first thing you'll notice is the enrichment of logs with new telemetry elements.

Firstly, Application Insights now logs dependencies (in this case, SQL queries), HTTP requests, and other events, as shown in Figure 8-7.
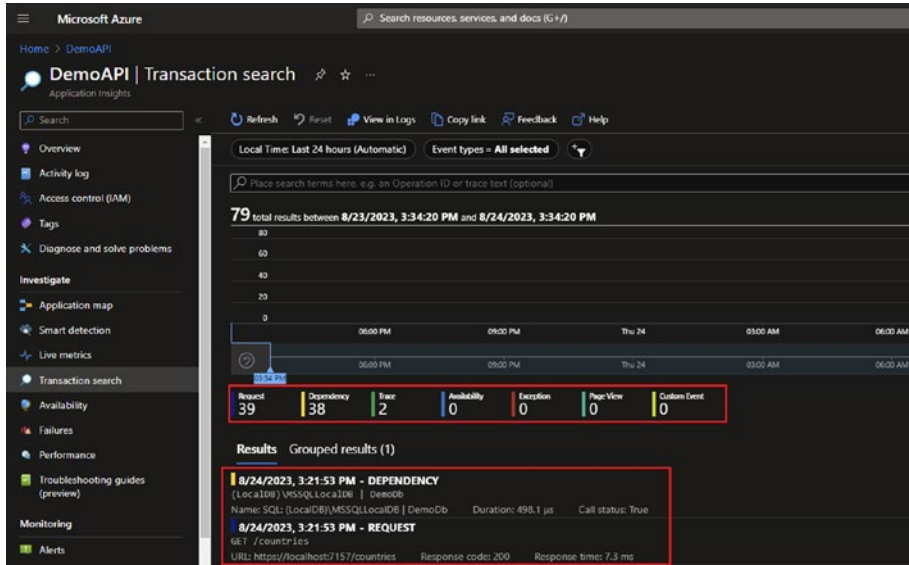
Figure 8-7. *Example of telemetry data*

As you can see, we know what HTTP request has been executed, what time it took to process the request, and what dependency has been called with what duration. Very useful, isn't it? I've got even more to show you: if you click a dependency, you'll have access to more details, such as which HTTP request triggered this call, as shown in Figure 8-8.
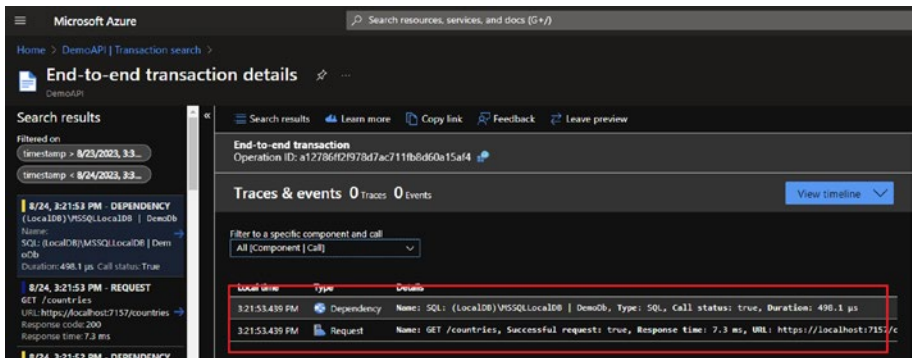
Figure 8-8. *Example of telemetry data on dependencies and HTTP requests*

I've got something even better! Exceptions are now enriched with other metadata, such as the endpoint that triggered the exception, the execution time, etc., as shown in Figure 8-9.
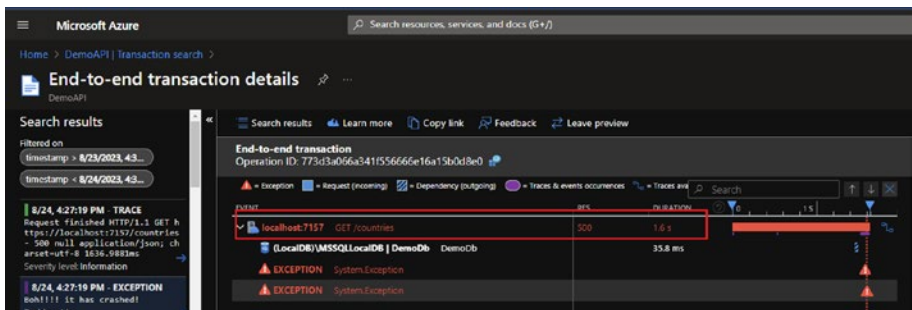
Figure 8-9. *Exceptions enriched with metadata*

Let's get down to business. Let's take a closer look at the metrics. Go to the Overview tab for an overview of metrics such as the count of failed requests, server response time, etc., as shown in Figure 8-10.
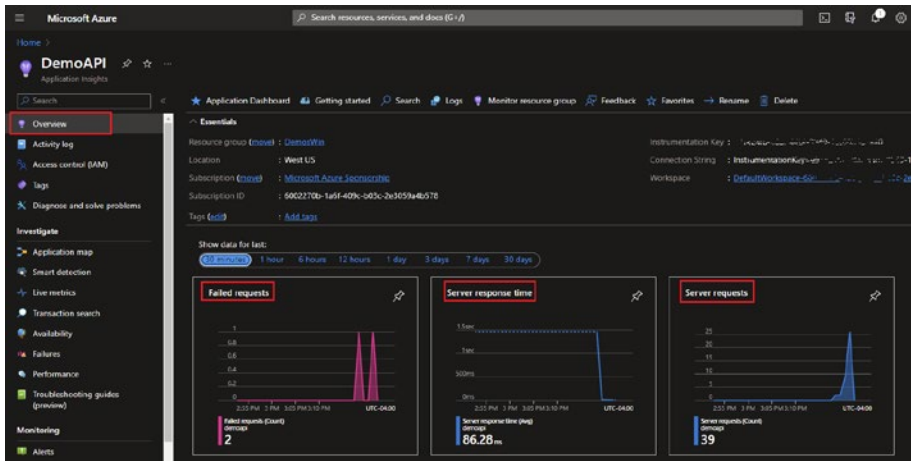
Figure 8-10. *Metrics overview*

At last, the highlight of the show! If you click the Live metrics tab, you'll find everything you need regarding metrics, such as CPU and memory usage. It's very comprehensive, as shown in Figure 8-11.
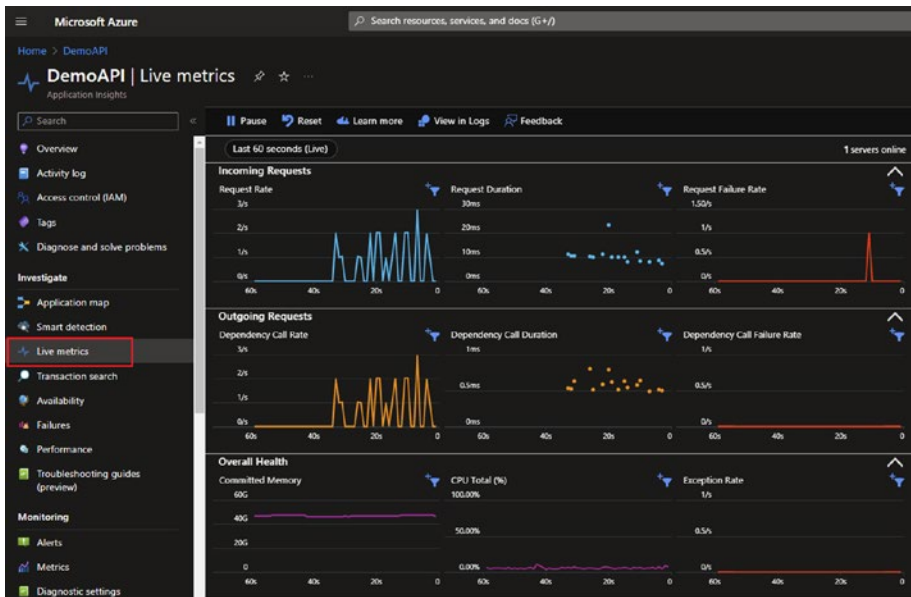
Figure 8-11. *Detailed metrics*

Of course, it's possible to customize your traces, metrics, etc. I won't talk about it here, as it's something you probably won't do. If you do, it won't be you but the architect you'll depend on, as the developer rarely implements custom traces and metrics other than those proposed by default. However, if you're interested in the subject, you can consult the Microsoft documentation here, `https://learn.microsoft.com/en-us/dotnet/core/diagnostics/metrics-collection`, but also this one, which is more concerned with the performance of your application: `https://learn.microsoft.com/en-us/azure/azure-monitor/app/performance-counters?tabs=net-core-new`.

Implementing HealthCheck

To finish with observability, I'm going to tell you about HealthCheck. But what is HealthCheck? The idea is to offer HTTP endpoints to ensure two main things:

1. The presence/proper deployment of the application

2. A report giving the status of the service and its dependencies (operational, non-operational, degraded)

There are two types of HealthChecks:

1. Readiness HealthCheck, indicating whether your application is ready for use

2. Liveness HealthCheck, indicating whether your application works or not

Without going into all the customization details, I'll show you an example of each type of HealthCheck.

Liveness HealthCheck

Since we're using SQL Server as our data access, I suggest you download the following Nuget package: *AspNetCore.HealthChecks.SqlServer.*

Then go to the *Program.cs* file and configure the HealthCheck for SQL Server with the *AddHealtchCheck* and *AddSqlServer* methods, the latter taking the database's connectionString as a parameter, and then expose the GET /health endpoint using the *MapHealthChecks* extension method, as shown in Listing 8-7.

Listing 8-7. The configuration of the GET /health endpoint for the HealthCheck

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

...

var dbConnection = builder.Configuration.
GetConnectionString("DemoDb");
builder.Services.AddHealthChecks()
                .AddSqlServer(connectionString:
                dbConnection)

...

var app = builder.Build();

app.MapHealthChecks("/health");

...

app.Run();
```

This endpoint will return *Healthy* or *Unhealthy* if the database connection works or not. Figure 8-12 shows the GET /health endpoint returning *Healthy*.
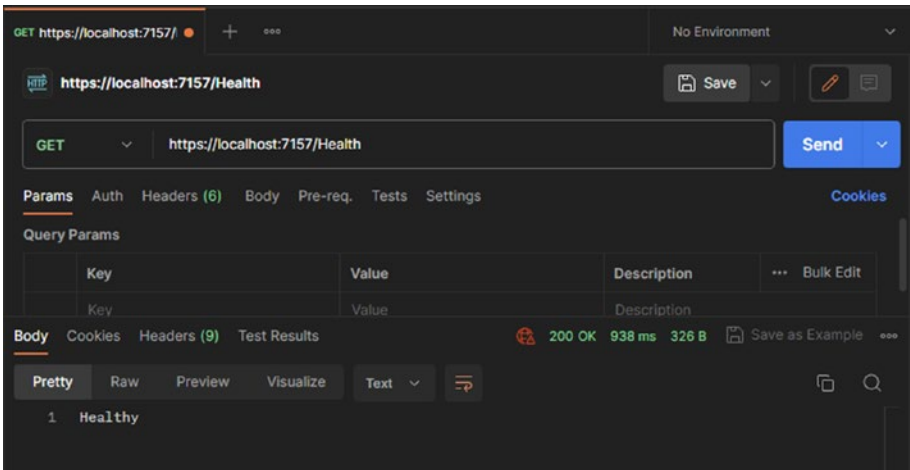
Figure 8-12. *The GET /health endpoint returning Healthy*

It is possible to test several databases at the same time, but for ASP.NET Core to be able to differentiate between them, each one must be given a name, as shown in Listing 8-8.

Listing 8-8. The configuration of the GET /health endpoint for the HealthCheck with two databases

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

...

var dbConnection1 = builder.Configuration.GetConnectionString("
DemoDb1");
var dbConnection2 = builder.Configuration.GetConnectionString("
DemoDb2");
builder.Services.AddHealthChecks()
                .AddSqlServer(name: "SQL1", connectionString:
                dbConnection1)
```

```
                    .AddSqlServer(name: "SQL2", connectionString:
                    dbConnection2);

...

var app = builder.Build();

app.MapHealthChecks("/health");

...

app.Run();
```

This example shows a liveness HealthCheck, demonstrating that the application works because of its dependency on SQL Server.

Readiness HealthCheck

Showing that our application is ready for use requires more manual work. In general, a readiness HealthCheck requires customization. Imagine that, after startup/restart, our application has had to perform some long actions to make it work as well as possible (think of initializing a cached dataset). Still, in this example, I will simulate a long action lasting ten seconds.

To do this, define a static variable, *IsReady*, on the static *Ready* class in our application that symbolizes the ready (or not ready) state, as shown in Listing .

Listing 8-9. The Ready static class

```
namespace AspNetCore8MinimalApis;

public static class Ready
{
    public static bool IsReady { get; set; } = false;
}
```

We now create the *ReadyHealthCheck* class, which implements the *IHealthCheck* interface, as shown in Listing 8-10.

Listing 8-10. The ReadyHealthCheck class

```
using Microsoft.Extensions.Diagnostics.HealthChecks;

namespace AspNetCore8MinimalApis.Healthchecks;

public class ReadyHealthCheck : IHealthCheck
{
    public Task<HealthCheckResult> CheckHealthAsync(
     HealthCheckContext context,
     CancellationToken cancellationToken = default)
    {
        var result = Ready.IsReady
            ? HealthCheckResult.Healthy()
            : HealthCheckResult.Unhealthy(
              "Application not ready");

        return Task.FromResult(result);
    }
}
```

As you can see, the *CheckHealthAsync* method reads the static *IsReady* variable of the *Ready* static class and returns *Healthy* if *IsReady* is *true* or *Unhealthy* if *IsReady* is false. Let's simulate a lengthy operation, which, at the end of its execution, will set the verifiable IsReady to true after ten seconds, as shown in the following piece of code (placed in the *Program.cs* file):

```
Task.Run(() => { Thread.Sleep(10000); Ready.IsReady = true; });
```

Let's now declare our *ReadyHealthCheck* in another endpoint to ensure we don't mix liveness and readiness states in the application. To do this, we need to group the HealthCheck with a tag, *Ready* for the readiness check and *Live* for the liveness check, with two separate endpoints attached to their respective tag as shown in Listing 8-11.

371

Listing 8-11. The readiness and liveness checks configured in separate endpoints

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

...

builder.Services.AddHealthChecks()
                .AddSqlServer(
                name: "SQL1",
                connectionString: dbConnection1,
                tags: new[] { "live" })

                .AddSqlServer(
                name: "SQL2",
                connectionString:
                dbConnection2,
                tags: new[] { "live" })
                .AddCheck<ReadyHealthCheck>(
                "Readiness check",
                tags: new[] { "ready" });

...

var app = builder.Build();

..

app.UseExceptionHandler(opt => { });

app.MapHealthChecks("/ready", new HealthCheckOptions
{
    Predicate = healthCheck => healthCheck.Tags.
    Contains("ready")
});
```

```
app.MapHealthChecks("/live", new HealthCheckOptions
{
    Predicate = healthCheck => healthCheck.Tags.
    Contains("live")
});

...

app.Run();
```

If I invoke the GET /ready endpoint before ten seconds, it will return Unhealthy, whatever the GET /live endpoint returns. And after ten seconds, it will return Healthy, whatever the GET /live endpoint returns.

Note   Since we enabled the telemetry data collection for Application Insights in the previous section, any HealthCheck endpoint invoked by a user or a system that regularly checks the state of our application will be logged as a request event as any other requests made on the application.

I haven't been too far on this ASP.NET Core feature; you only need to know how to monitor if your application is functional and ready to use. To learn more about HealthCheck configuration and customization, you can check the Microsoft documentation here: https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/health-checks?view=aspnetcore-8.0.

Summary

And that's it for this chapter. I've tried to keep it simple and concise, as this subject could be the subject of a whole book. The aim here was to make you aware of the notion of observability by giving you the basics to enable you to do the minimum you'll need to implement in your application. Many other APMs are used in many companies, such as Jaeger and Grafana; you should also consider them for monitoring your apps. Still, I've also provided you with the necessary resources to learn more if you're interested. You can be sure that what you've learned in this chapter will be very helpful! I'll see you in the next chapter to discuss configuring sensitive data (secret applications), which will also be very useful!

Managing Application Secrets

Throughout this book, I've used sensitive data, a.k.a. secrets, for example, connection strings that may have been used to connect to a database, an APM, etc. The easiest way to make these secrets available is to save them in application configuration files named by environment (*appsettings.json*, *appsettings.dev.json*, *appsettings.prod.json*, a dedicated configuration file for each environment) and then push all these files into the GIT repository. GIT is a source code manager on a remote server to store your code. However, two questions arise: How/where to store these secrets, and how to update them easily?

In this chapter, I'm going to teach you the following points:

- Introduction to application secret management

- Example with Azure Key Vault

Introduction to Application Secret Management

Anyone with access to the source code manager can view the various environments' passwords. A developer or tester may not need to know the secrets of accessing production databases, for example, and this is potentially a threat to data security, as a team member may be ill-intentioned.

Applications are often entry points for hackers. Finding sensitive information enabling access to other resources, such as a database, is a real danger. The most telling example is Uber in 2016. Although the source code was stored in private GIT repositories (not accessible to the public), a hacker managed to bypass security and gain access to the applications that contained Uber's database secrets, exposing 57 million driver accounts!

To prevent this from happening in your company, I will show you an example of protecting sensitive data with Azure Key Vault, which requires you to create an Azure Key Vault resource in Microsoft Azure as you did for Application Insights. As the aim is to inform you of the importance of protecting secrets, I won't explain how Azure Key Vault works in detail. Here is the Microsoft documentation for configuring a vault with Azure Key Vault: `https://learn.microsoft.com/en-us/azure/key-vault/general/quick-create-portal`.

Don't forget to pick up your Vault URI on Azure Key Vault's main page; you will need it further!

You'll also need your Azure Tenant ID, which can be found on the Azure Portal as shown in Figures 9-1 and 9-2. For security purposes, I have partially hidden my Tenant ID.
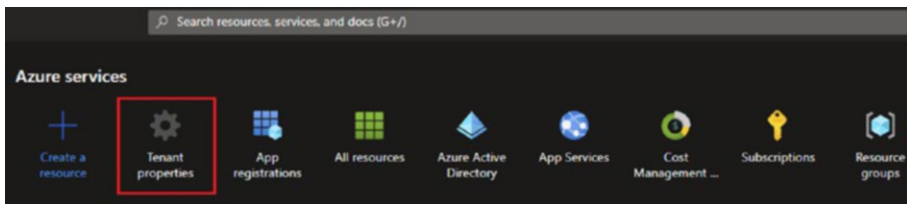
Figure 9-1. *The Tenant properties menu item*
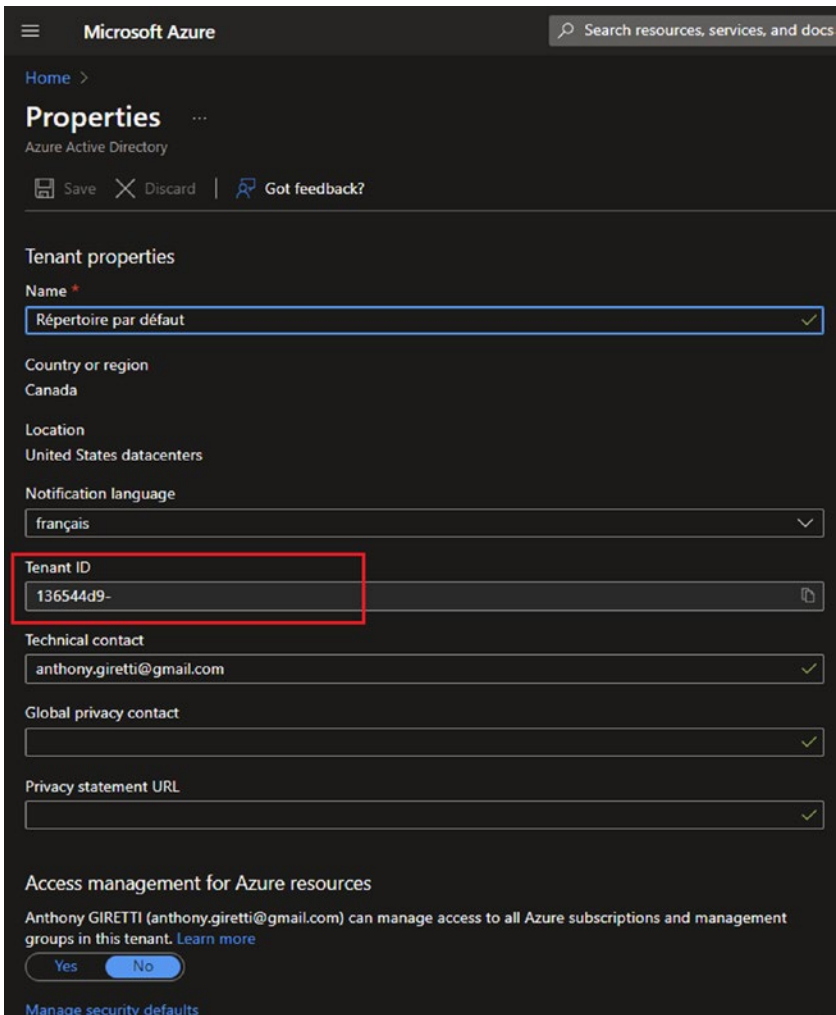
Figure 9-2. The Tenant ID

Example with Azure Key Vault

I'll show you how to protect your database connections, for example, the ones I used earlier in this book. Of course, you can repeat the same logic to store any secret or other string connections like the one in Application Insights. To do this, go back to the main page of your Azure Key Vault instance (again, I assume you've read the Microsoft documentation beforehand) and then create your keys and associated values for your connection strings, as I did for the *DemoDb1* and *DemoDb2* connection strings as shown in Figure 9-3.
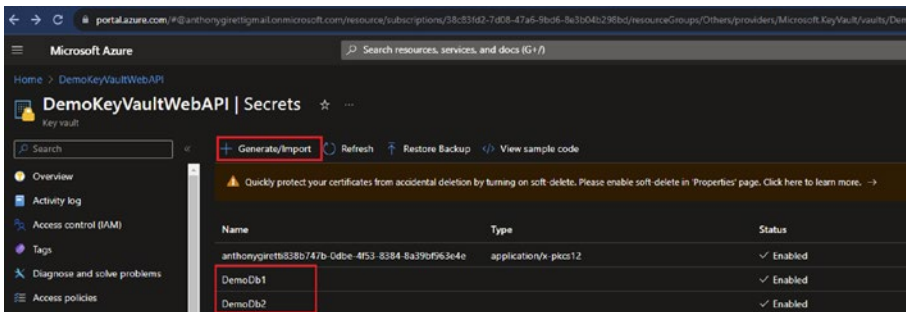
Figure 9-3. *Creating secrets in Azure Key Vault*

On the ASP.NET Core side, however, we will have a bit more work to do. You'll need to download two Nuget packages, and I'll explain why:

- ***Azure.Extensions.AspNetCore*.Configuration.Secrets**: This package allows you to retrieve secrets from ASP. NET Core on Azure Key Vault.

- ***Azure.Identity***: This package will enable us to connect to Azure Key Vault without an Azure Key Vault– specific login/password from Visual Studio using our Visual Studio account. In other words, we'll be using Microsoft's *Single Sign-On (SSO)*. I'll come back to this in the next chapter of this book.

If you've created an Azure Key Vault workspace, you have a Microsoft Azure account and, therefore, a Microsoft account. So click the top right-hand corner of Visual Studio to authenticate yourself with your Microsoft account, which must be the same as your Microsoft Azure account. Figure 9-4 shows my identity after authentication in Visual Studio.
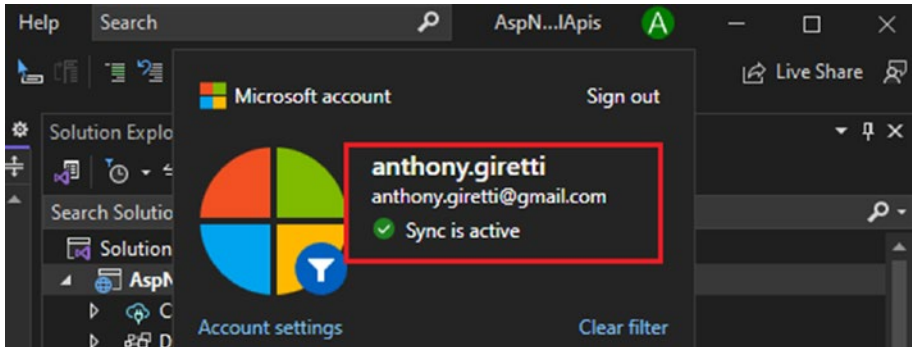
Figure 9-4. *Connection to a Microsoft account from Visual Studio*

Once authenticated, as shown previously, we can retrieve the key/value pairs of secrets stored in the Key Vault. Before configuring everything in the *Program.cs* file, Azure Key Vault requires the Tenant ID of the Microsoft Azure account associated with Azure Key Vault. We'll need to create an environment variable that the SDK of the package we downloaded earlier *(Azure.Extensions.AspNetCore.Configuration.Secrets)* will read and send to Azure Key Vault when it connects to it. To do this, go to the properties of your API project, click "Debug," and set up the *AZURE_TENANT_ID* variable as shown in Figure 9-5.
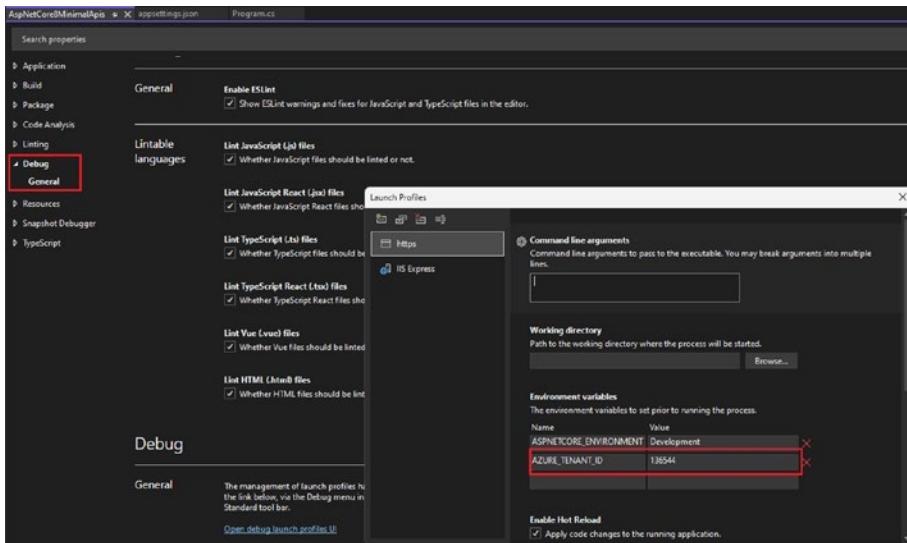
Figure 9-5. *Set up the AZURE_TENANT_ID environment variable*

Go to the *appsettings.json* file to remove the *ConnectionStrings* property you wish to protect, and then add the Vault URI you need to have on hand, as shown in Listing 9-1.

Listing 9-1. The Azure Key Vault configuration

```
"KeyVault": {
    "Uri": "{YourKeyvaultUri}"
 }
```

As you can see, I've named *KeyVault* my section containing the *Uri* property. You can configure your section as you like; I named it as such. This is not sensitive data, so it can remain in the *appsettings.json* file. On the other hand, if you're paranoid like me, you can right-click your API project and click "Manage User Secrets," which will generate a *secrets.json* file that will be stored only on your machine. When the API project runs,

this configuration will be transparently merged with the configuration in the appsettings.json file. Figures 9-6 and 9-7 show the Visual Studio process I've just described.
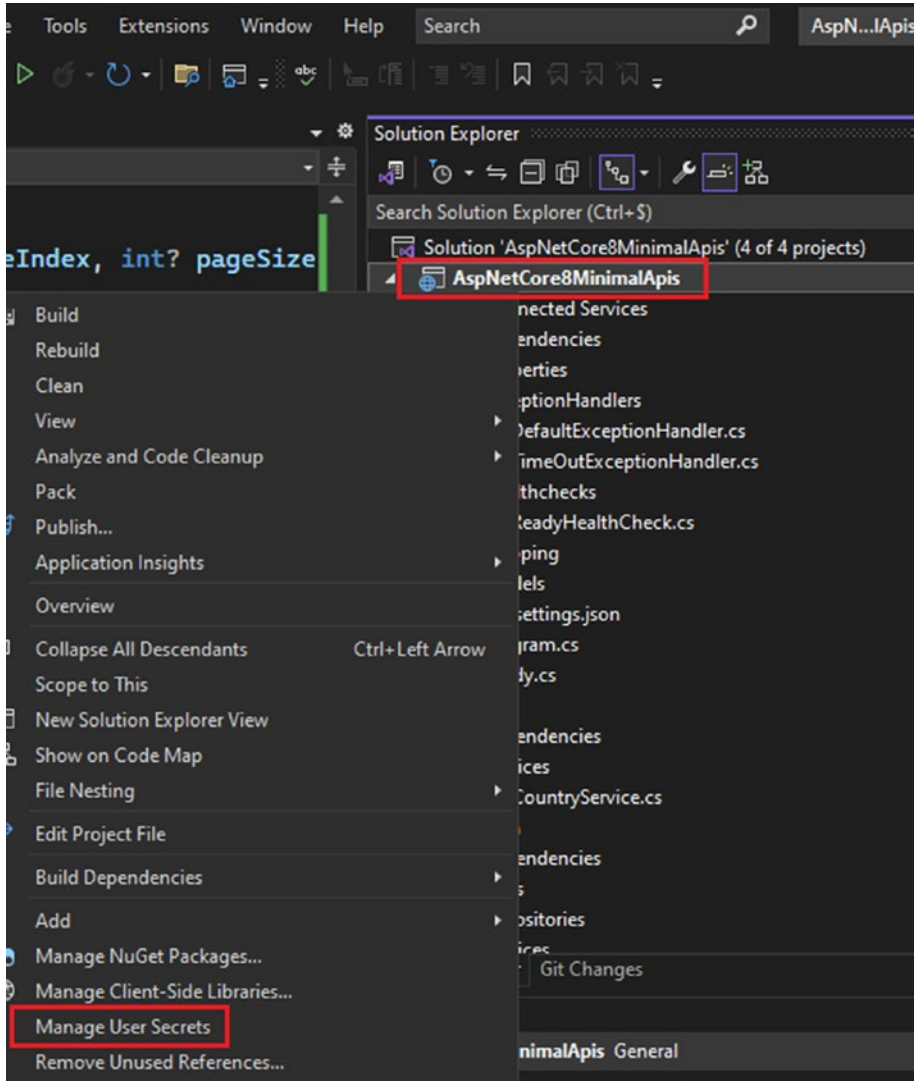
Figure 9-6. *Find the "Manage User Secrets" menu item*
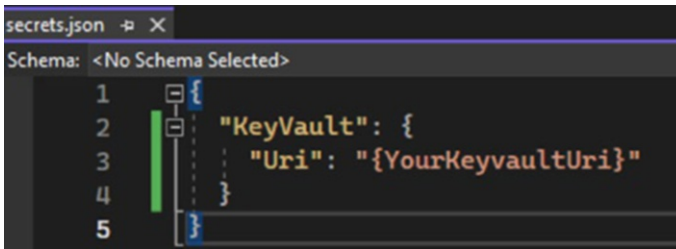
Figure 9-7. *The secrets.json file*

Let's go to the *Program.cs* file and add this piece of code to retrieve our secrets:

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

...

var keyVaultUri = builder.Configuration.GetValue<string>("Key
Vault:Uri");
builder.Configuration.AddAzureKeyVault(new Uri(keyVaultUri),
new DefaultAzureCredential());
var dbConnection1 = builder.Configuration.
GetValue<string>("DemoDb1");
var dbConnection2 = builder.Configuration.
GetValue<string>("DemoDb2");

...

var app = builder.Build();

...

app.Run();
```

It's straightforward to implement. The *DefaultAzureCredential* class used as a parameter for the *AddAzureKeyVault* method specifies that authentication to the Azure Key Vault service will be performed using

the Microsoft account authenticated in Visual Studio. Of course, when
your application is deployed in production, this will be slightly different.
A login/password will be required, or your team's cloud architect or
DevOps engineer will create a service account that will simulate the
same behavior as Visual Studio when your application runs in the cloud.
The sample code (lines 11 to 31) provided by Microsoft shows how this
should be done: https://github.com/dotnet/AspNetCore.Docs/blob/
main/aspnetcore/security/key-vault-configuration/samples/6.x/
KeyVaultConfigurationSample/Program.cs.

I won't go any further into the configuration of our production
application, as this is not the subject of this book.

To retrieve our secrets, access them using the *builder.Configuration.
GetValue<T>("YourKey")* method, where *T* is the type of secret to be
retrieved, in this case, a string, and *YourKey* is the exact name of your
secret, in this case *DemoDb1* and *DemoDb2*. If we execute our application,
it should retrieve the desired secrets, as shown in Figure 9-8.

Figure 9-8. *Retrieving secrets*

As you can see, it works well!

Summary

This chapter was short, but I hope it was full of learning! Remember
that a security flaw threatening the integrity of your data can be hidden
anywhere, and hackers will have a field day! Remember that this kind of
data leak, caused by poor secret management practices, is at the root of

many attacks, which is why this type of attack is listed in the OWASP Top 10. Suppose you don't want to use Microsoft Azure and are more into AWS. In that case, you can use the AWS Key Management Service, if you want: `https://aws.amazon.com/fr/kms/`. In the next chapter, we'll look at a final aspect of application security: authentication and authorization!

CHAPTER 10

Secure Your Application with OpenID Connect

Security is essential in an application, by which I mean that almost all applications need a mechanism to identify the user attempting to perform actions on them. This is called authentication, which should not be confused with authorization, a mechanism allowing privileges to be given to an authenticated user, that is, allowing them to perform specific actions that other users will not achieve. In this chapter, you will learn

- **I**ntroduction to OpenID Connect (OIDC)

- Configuring authentication and authorization in ASP. NET Core

- Passing a JSON Web Token (JWT) into requests and getting the user's identity

Introduction to OpenID Connect

OpenID Connect (OIDC) is an identification standard that is positioned above *OAuth 2.0*, which is itself an authorization protocol. *OpenID Connect* works on the principle of delegating user authentication: with *OpenID Connect*, this responsibility is entrusted to a third-party service. The latter uses the protocol to ensure the user is authenticated, so the application protected by *OpenID Connect* does not know how the authentication is performed. So that's it with the login forms in your code.

To be completely independent of the application, this authentication system can be transverse and reused to develop a single authentication within an information system. This is the very definition of the *Single Sign-On (SSO)* principle. We end up with an interaction between three actors:

1. The client that is a web app, for example

2. The identity provider

3. The protected resource

Figure 10-1 shows the relationship between the three actors.
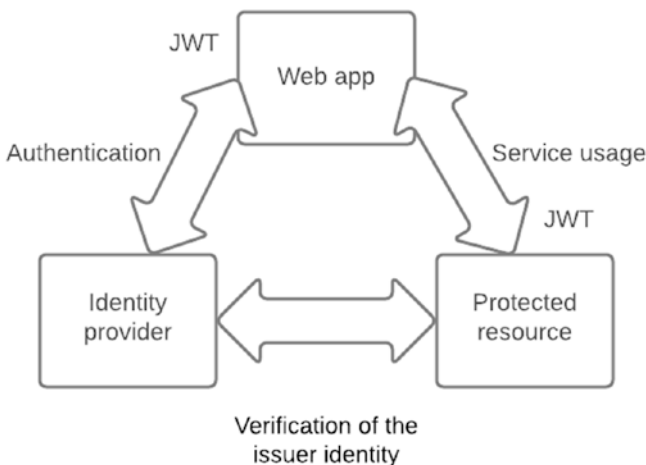
Figure 10-1. *The relationship between the three actors in OpenID Connect*

The client will authenticate with the service provider. The latter will issue a *JSON Web Token (JWT)* that will be used to access the protected resource. This resource will validate the token received by retrieving the metadata from the identity provider to certify that the latter is the issuer of the JWT. Metadata is retrieved only once, and then the application can validate the JWT autonomously. A JWT is a JSON accompanied by a signature and the reference to the key, which allows the signature to be verified. The whole is encoded in Base64, and dots separate the three parts. They are assembled as follows: the header, the content, and the signature. I will show you an example in the next section. **RFC 7519** describes the JWT standard, which can be found at this address: `https://datatracker.ietf.org/doc/html/rfc7519`.

This introduction is brief. The goal is not to teach you *OpenID Connect* in great detail but rather to understand the basic principle, the minimum, to allow you to use *OpenID Connect* to authenticate in ASP.NET Core. If you want to learn more about OpenID Connect, you can consult the official documentation for this protocol here: `https://openid.net/connect/`.

To configure ASP.NET Core with *OpenID Connect*, we must have an identity provider to achieve our ends. You may not know it, but a lot of applications use *OpenID Connect*, and I think you already know of the most often-used identity providers:

- Facebook
- Google
- Apple
- And less frequently, Microsoft

Figure 10-2 shows the canva.com website offering to authenticate with different providers.
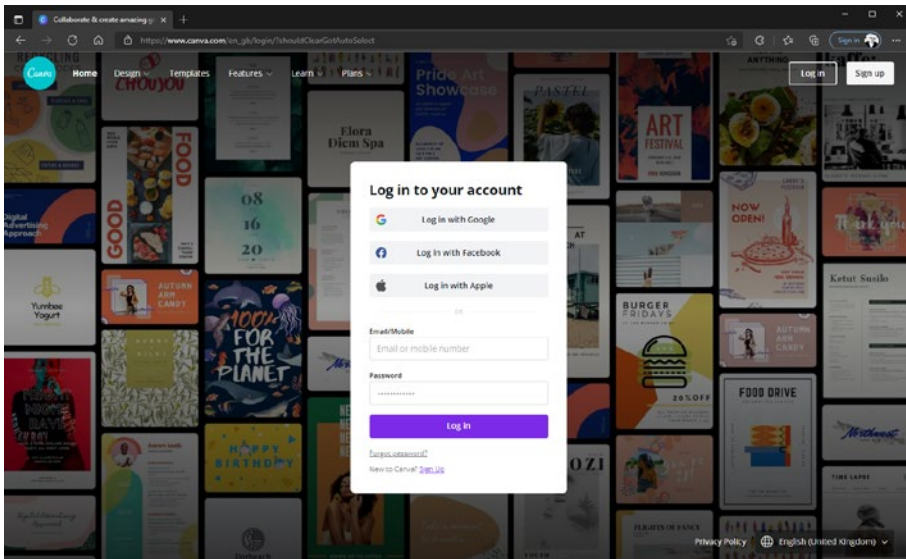
Figure 10-2. *Canva.com uses Google, Facebook, and Apple as OpenID Connect providers*

In the code samples in this chapter, I'll be using the Microsoft authentication platform based on Azure Active Directory. However, I will not go into details about its configuration. I will show you how to configure ASP.NET Core; the authentication part will be up to you. Azure Active Directory and *OpenID Connect* are big pieces. To avoid losing the line on the main subject, I invite you to learn more about the Microsoft Identity Platform here: https://docs.microsoft.com/en-us/azure/active-directory/develop/active-directory-v2-protocols. If you want to get things done quickly, you can follow my tutorial on setting up *OpenID Connect* on Microsoft Azure here: https://anthonygiretti.com/2018/02/28/using-openidconnect-with-azure-ad- angular5-and-webapi-core-introduction/.

Note   Along with this chapter, I will assume you could obtain the emitted **access_token**, which is used as a bearer token.

Configuring Authentication and Authorization in ASP.NET Core

The configuration of an ASP.NET Core API is generic, regardless of the identity provider used to emit a token. For example, the following configuration applies to minimal APIs or other ASP.NET Core applications. To get started, install the required Nuget package: *Microsoft.AspNetCore. Authentication.JwtBearer*.

Once done, go to the *Program.cs* file and configure and activate authentication and authorization.

Configuration needs several lines of code:

- The **AddAuthentication** extension method defines the authentication based on a JWT using the *JwtBearerDefaults.AuthenticationScheme* value. AddAuthentication defines the type of authentication, and it's generic. *JwtBearerDefaults. AuthenticationScheme* is specific to JWT authentication.

- The **AddJwtBearer** extension method allows the setup of the *Authority*, that is, the authentication server address, and *Audience*, that is, the target application for which the JWT is emitted. Both of these values are given by the identity provider you have chosen. Then we will configure the parameters used to validate the JWT: *ValidateLifetime* and *ValidateIssuer* both set to

> *True* and *Clockskew*, which is used to manage the time gap between the JWT issuer and the application and will be set up to 5 min. In other words, the latter allows a 5-minute gap between the JWT expiry timestamp and the application, where the token lifetime is validated.

- The **AddAuthorization** extension method allows configuring authorization in ASP.NET Core by using the *Authorize* attribute.

Activation is only about adding two middlewares in the pipeline:

- The **UseAuthentication** extension method, which registers the *Authentication* middleware in the pipeline

- The **UseAuthorization** extension method, which activates the *Authorization* middleware in the pipeline

Listing 10-1 shows the *Program.cs* properly configured. *Authority* and *Audience* are partially hidden. They are specific to my Azure Active Directory tenant on Microsoft Azure. Following their documentation, you can find the Authority and Audience configuration parameters in *any OpenID Connect* provider. Once again, if you want to try *OpenID Connect* with Microsoft Azure, you can follow the tutorial on my blog post I mentioned earlier in this chapter.

Listing 10-1. Configure and activate OpenID Connect authentication and authorization on ASP.NET Core

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

...

builder.Services.AddAuthentication(options =>
{
```

```
    options.DefaultAuthenticateScheme = JwtBearerDefaults.
    AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.
    AuthenticationScheme;
}).AddJwtBearer(options =>
{
    options.Authority = "https://login.microsoftonline.
    com/136544d9-xxxx-xxxx-xxxx-10accb370679/v2.0";
    options.Audience = "257b6c36-xxxx-xxxx-xxxx-6f2cd81cec43";
    options.TokenValidationParameters.ValidateLifetime = true;
    options.TokenValidationParameters.ValidateIssuer = true;
    options.TokenValidationParameters.ClockSkew = TimeSpan.
FromMinutes(5);
});

builder.Services.AddAuthorization();

...

var app = builder.Build();

...
app.UseCors(); // Before Authentication and Auhtorization
middlewares
app.UseMiddleware<YourMiddleware>(); // Before Authentication
and Authorization middlewares

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

...

app.Run();
```

> **Caution**   The *UseCors* middleware and other custom middlewares must be placed **before** the *UseAuthentication* and *UseAuthorization* middlewares, or else they won't work correctly since UseCors must not depend on authentication, so authentication must apply after CORS handling.

To apply authentication on your minimal endpoints, you'll need to add the *RequireAuthorization* extension method on any minimal endpoint you want to protect by authentication, like the GET /authenticated endpoint as shown in Listing 10-2.

Listing 10-2. The GET /authenticated endpoint

```
app.MapGet("/authenticated", () =>
{
    return Results.Ok("Authenticated !");
}).RequireAuthorization();
```

This example is the simplest but allows you to do the necessary to get authenticated, but it doesn't show authorization. Very often, applications require higher privileges for some users. Those users may have more responsibilities and may need to perform sensitive actions. The company will assign a particular (or several) role(s) that can be passed in a JWT when the latter is issued and handled by any ASP.NET Core application. Listing 10-3 shows how to configure a custom policy on the *AddAuthorization* method that ensures the authenticated user has the *SurveyCreator* role, and this role will be assigned to the GET /authorized endpoint that requires the *SurveyCreator* custom policy to get executed.

Listing 10-3. The GET /authorized endpoint assigned with the
SurveyCreator policy

```
...
builder.Services
        .AddAuthorization(options =>
         options.AddPolicy("SurveyCreator",
         policy =>  policy.RequireRole("SurveyCreator")
));
...
var app = builder.Build();
...
app.UseAuthentication();
app.UseAuthorization();
...
app.MapGet("/authorized", () =>
{
    return Results.Ok("Authorized !");
}).RequireAuthorization("SurveyCreator");
..
app.Run();
```

Suppose a user tries to invoke any endpoint configured with
authentication or authorization without a valid JWT. ASP.NET Core will
return the HTTP UnAuthenticated (401) error in that case. If the user is
authenticated but doesn't meet the authorization requirements, such as
missing the proper role, ASP.NET Core will return the HTTP Forbidden
(403) error.

I have shown you how to secure your application with a JWT. You must follow your company's business rules to apply the proper criteria to protect your application. Before finishing this section, I would like to show you what a JWT looks like with the *SurveyCreator* role assigned to user Anthony Giretti. First, please generate a token with your provider, and then go to the https://jwt.io website to observe the content of your JWT. Figure 10-3 shows the JWT of my decoded provider.

Figure 10-3. *A JWT decoded on the* https://jwt.io *website*

As you can see, we find the information relating to the provider and the expiration date of the JWT in the first framed block and information on the user for whom the JWT was issued in the second framed block, then followed by the role(s) that the user may have. So you will have understood that decoding your JWT will be used to debug your application if you have trouble with the expiration date of your JWT, if you are using the wrong *Audience,* or if you are not using roles correctly (or if you have poorly set up your JWTs with your identity provider).

In the next section, I'll show you how to use the token and pass it into the headers from Postman and Swagger.

Passing a JWT into Requests and Getting the User's Identity

Passing a JWT to any web application always works the same way. It's a matter of passing a header named Authorization in the headers with the value *"bearer {YourJWT}"*.

Note   In general, steps to accomplish JWT authentication with OIDC are to set up an OIDC provider, have a UI application that fetches a JWT, and then send it to the back end, such as an ASP. NET Core API. The examples in this section show you how to debug back-end applications (ASP.NET Core), assuming you already have generated a JWT. You can also set up Identity Server, an open source *OpenID Connect* provider. You can learn how to set it if you don't want to use Microsoft Azure or any other providers here: `https:// scientificprogrammer.net/2022/10/02/securing-your- signalr-applications-with-openid-connect-and-oauth/`.

If you remember, at the start of the book, I told you about Swagger, which lets you generate documentation for your API and expose it to your customers. Since Swagger is well designed, you can configure it to take a JWT and then pass it to the HTTP request. To do this, you need to configure the *AddSwaggerGen* method, as shown in Listing 10-4.

Listing 10-4. The AddSwaggerGen method configured for accepting a JWT

```
builder.Services.AddSwaggerGen(c =>
{
    c.AddSecurityDefinition("Bearer", new
    OpenApiSecurityScheme()
    {
        Name = "Authorization",
        Scheme = "Bearer",
        BearerFormat = "JWT",
        In = ParameterLocation.Header,
        Description = "JWT Authorization header required"

    });
    c.AddSecurityRequirement(new OpenApiSecurityRequirement {
        {
            new OpenApiSecurityScheme {
                Reference = new OpenApiReference {
                    Type = ReferenceType.SecurityScheme,
                        Id = "Bearer"
                }
            },
            new string[] {}
        }
    });
});
```

If we now execute the Swagger page, you should see an "Authorize" button at the top right of the page, as shown in Figure 10-4.
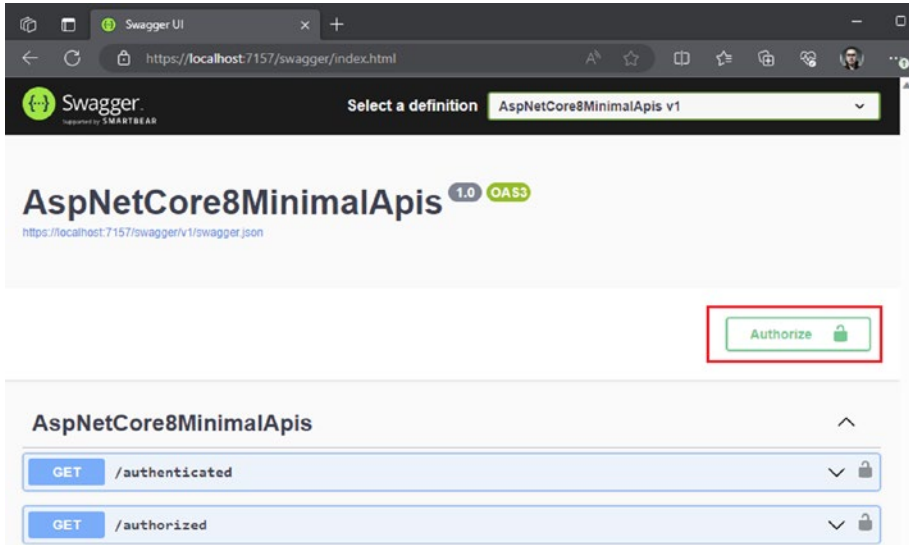
Figure 10-4. *The Authorize button*

If you click, you'll be asked to paste your JWT, taking care not to forget the bearer keyword in front of it, as shown in Figure 10-5.
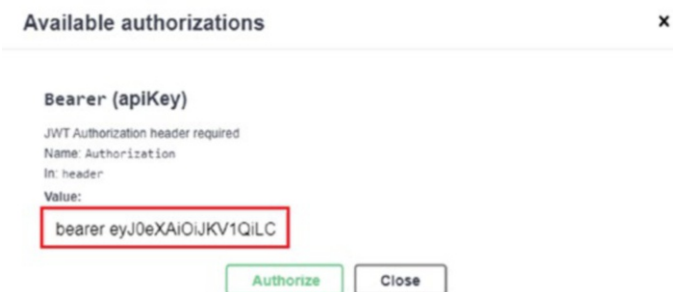
Figure 10-5. *Filling the headers with the JWT bearer Authorization*

If you run the /GET authorized endpoint, assuming the JWT is valid and contains the *SurveyCreator* role, Swagger should return a successful response as shown in Figure 10-6.
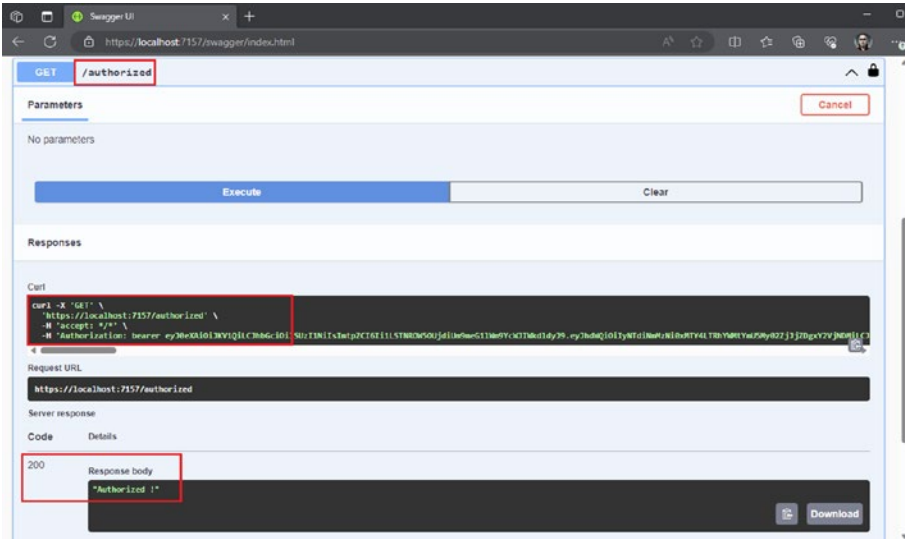
Figure 10-6. *The GET /authorized endpoint output in Swagger after passing a valid JWT*

Now, let's try Postman. **Be careful; Postman itself adds the bearer keyword,** so you can only paste your JWT in the *Auth* section, as shown in Figure 10-7.
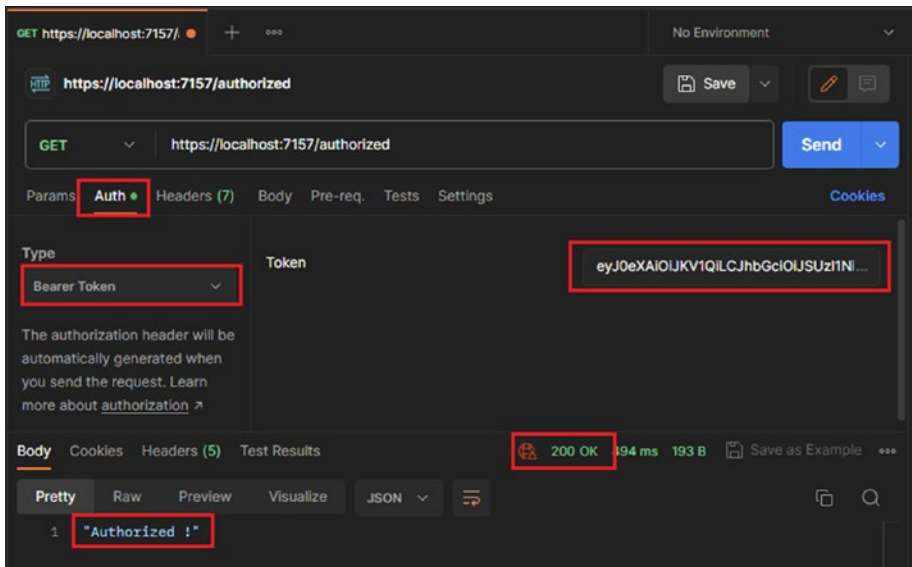
Figure 10-7. *The GET /authorized endpoint output in Postman after passing a valid JWT*

Into your minimal endpoint, inject the *ClaimsPrincipal* class, which is automatically registered in the dependency injection by ASP.NET Core, and you'll get the user's identity in the form of *Claims*, as shown in Figure 10-8.
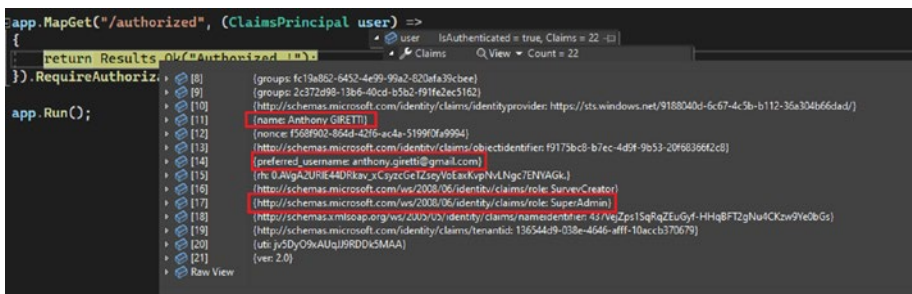
Figure 10-8. *Accessing the user's identity through the UserClaims object*

When a user is logged in, their identity is defined by *Claims*, which somehow defines their profile as a logged-in user; in other words, they are identity data issued by the identity provider. As you can see from the preceding figure, there is a whole range of claim types, such as the user's name or roles. The claims contain all the information about the user, which is customizable on the identity provider side, and I won't go into detail here. However, we will design a service that will expose user profile data more intelligibly. Let's consider the *UserProfile* class, which implements the *IUserProfile* interface and contains two properties, *Name* and *Roles* (for a simple, easy-to-understand example), as shown in Listing 10-5.

Listing 10-5.   The UserProfile class

```
using System.Security.Claims;

namespace AspNetCore8MinimalApis.Identity;

public class UserProfile : IUserProfile
{
    private readonly IHttpContextAccessor _context;
    public UserProfile(IHttpContextAccessor context)
    {
        _context = context;
    }

    public string Name => _context.HttpContext.User?.Claims.
    FirstOrDefault(x => x.Type == "name")?.Value;

    public IEnumerable<string> Roles => _context.HttpContext.
    User?.Claims.Where(x => x.Type == ClaimTypes.Role).Select(x
    => x.Value);
}
```

As you can see, I access Claims using LINQ. It is important to remember that the *ClaimsPrincipal* class is not available directly via dependency injection, as in a minimal endpoint when it's a custom service. Don't forget to register the *IUserProfile* service with its *UserProfile* implementation and the *HttpContextAccessor* service with its *AddHttpContextAccessor* extension method. Let's run the same endpoint as earlier but with the *IUserProfile* interface injected, as shown in Figure 10-9.

Figure 10-9. *Accessing the user's identity through the IUserProfile interface*

Summary

Voila! You know how to manage JWT authentication in your application. Remember that, as I write these lines, JWT authentication is the most widely used way to authenticate to an API, thanks to its reliability and the widespread adoption of *OpenID Connect*. Now you know how to do everything in a moderately complex API. There's just one thing left: test your API. And I'll give you the final chapter of this book.

Testing APIs

Testing an application reveals errors related to its quality. Whether the test is functional, performance-related or aimed at verifying the user experience. It's an essential part of software development. The test team (the whole team involved in developing the application, not just the developers) draws up a report on these aspects, enabling the developer to make any necessary corrections. In this chapter you will learn:

- Introduction to testing
- Efficient unit testing

Introduction to Testing

It's always best to test applications early (in the application development cycle) to find bugs and eliminate them before they can affect the end user. There are many different types of testing, and I will introduce them to you quickly:

1. **Unit testing**: It's a type of software testing in which individual units or components are tested. The main aim of unit testing is to validate each unit of behavior in software and determine whether it works as expected.

2. **Integration testing**: Integration testing aims to validate that all independently developed parts work well together. For example, verify if the code works correctly once connected to a database.

3. **End-to-end testing**: End-to-end testing is a technique used to verify the correct operation of an application. It involves testing the entire system, from its interface to operating mode.

4. **Functional testing**: Functional testing focuses on the process's results rather than the process's mechanics.

5. **Acceptance testing**: Acceptance testing is often confused with functional testing. This type of testing aims to verify that the application meets the end user's expectations.

6. **Performance testing**: Performance testing is a non-functional software test used to evaluate an application's performance in terms of stability and speed under heavy workloads.

7. **Smoke testing**: Smoke testing, often confused with end-to-end and functional tests, tests critical application functions such as authentication.

Unit tests and integration tests are the exclusive responsibility of the developer. In contrast, end-to-end tests can be designed by the developer, for a DevOps engineer who will automate tests, or by the quality assurance team using end-to-end testing tools. All other tests are performed by the quality assurance team or the client themself. I won't be discussing end-to-end testing here, as I generally prefer to leave this responsibility to DevOps and/or QA engineers for a single reason: the complexity of testing.

It's a complex thing for a developer to set up because you have to set up all the dependencies on external resources on the one hand. Still, above all, you have to automate the whole thing, which is, speaking only of code, very demanding in terms of energy and time. I won't go into integration tests either because if end-to-end tests are carried out by whomever, they are not, in my opinion, very relevant. In this chapter, we'll focus on unit tests essential to any developer's life.

Note   In this chapter, I am talking about unit testing, which is not *Test-Driven Development (TDD)*, and it's different: unit testing is the practice of performing automated tests on code, usually **after** it has been written. *TDD* is a set of practices whereby you write your unit tests **before** writing your code and continually perform tests while you keep coding.

Efficient Unit Testing

Unit tests enable developers to verify the operation of a unit. A unit of code, known as a *System Under Test (SUT)*, generally a function, contains a particular logic that must be tested without regard to external dependencies. Unit tests are, therefore, performed in **isolation** from the rest of the application. This is one of the characteristics of good unit tests, among others:

- **Readable**: They must be easy to understand by reading the code.

- **Specific**: We only test one behavior at a time, so we don't include conditions in the unit test to cover all use cases.

- **Fast**: Tests must run quickly. This is generally the case when they are performed in total isolation from the rest of the application.

- **Complete (or almost)**: It's often said that you should test 100% of your code. In reality, it's more involved than that. Not everything is easily testable. Aim for at least 60% code coverage. Take, for example, *the Program.cs* file, and check that the dependencies you've registered (repository, classes, middleware) are correctly registered. Well, it's a long way to test for little benefit, but an end-to-end test can verify it, and you'll quickly realize if your application crashes.

- **Immediate**: Do your tests right from the start of the project. Otherwise, you'll never do them! (I say this from experience.)

Using the Right Tools

To code unit tests efficiently, we'll need several tools and libraries. We'll create a library project in Visual Studio; reference the projects we want to test, for example, our API layer; and add the following Nuget packages:

- **Microsoft.NET.Test.Sdk**: This package is required to run unit tests in a .NET solution.

- **xunit**: This package allows you to use xUnit as the test framework. It proposes a great feature to run your tests.

- **xunit.runner.visualstudio**: This package allows Visual Studio to discover the tests in your solution. Visual Studio won't find any tests made with xUnit without this package.

- **NSubstitute**: This package is a mocking library. I will return to this while showing you the unit test example.

- **AutoFixture**: This package allows you to generate fake data to populate quickly object properties.

- **ExpectedObjects**: This package allows you to compare objects by value and not by their reference. You'll see it's convenient.

Your test project should look as shown in Figure 11-1.
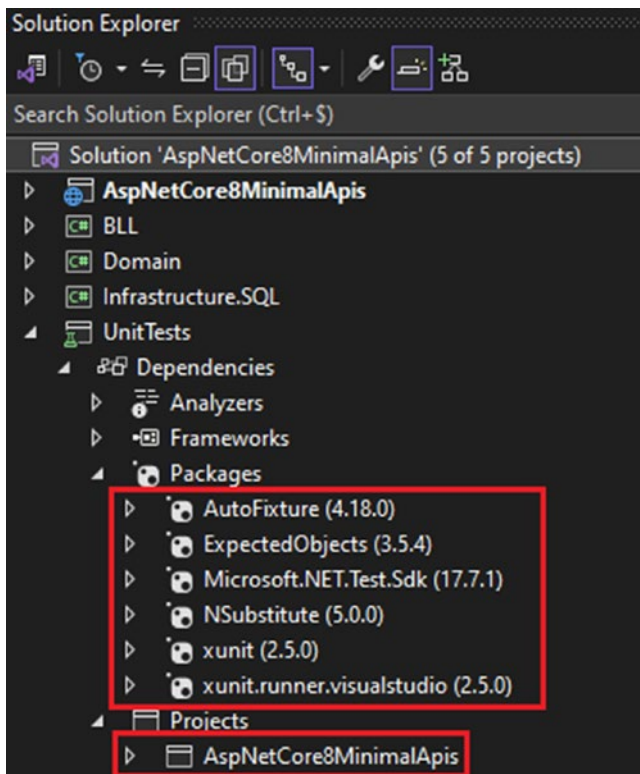
Figure 11-1. *The unit test project structure*

Testing a SUT Step-by-Step

Let's choose the minimal endpoint GET /countries I've isolated in a static *GetCountries* function (within the *CountryEndpoints* class) as the example SUT, as I showed you in Chapter 5. If you recall, I did this to facilitate the testability of the minimal endpoint. Let's test it. Here's its implementation, as shown in Listing 11-1.

Listing 11-1. The CountryEndpoints class

```
using AspNetCore8MinimalApis.Mapping.Interfaces;
using Domain.DTOs;
using Domain.Services;

namespace AspNetCore8MinimalApis.Endpoints;

public static class CountryEndpoints
{
    public static async Task<IResult> GetCountries(int?
    pageIndex, int? pageSize, ICountryMapper mapper,
    ICountryService countryService)
    {
        var paging = new PagingDto
        {
            PageIndex = pageIndex.HasValue ? pageIndex.
            Value : 1,
            PageSize = pageSize.HasValue ? pageSize.Value : 10
        };
        var countries = await countryService.GetAllAsync(paging);

        return Results.Ok(mapper.Map(countries));
    }
}
```

Identify What to Test

The first thing to do here is to identify what you want to test. The *GetCountries* function will be easy to test, as it has only four possible behaviors: the parameters *pageIndex* and *pageIndex* can each be null or not, and one of them can be null and the other not. That's four behaviors, so four tests to run. (This is good practice in unit testing: there are as many tests as possible as there are behaviors.) As each test will be similar, I'll show you how to test the *GetCountries* function when all query string parameters are null.

Let's have a look at what we're going to test:

1. The function returns an *IResult* of type *Ok*, precisely with a list of *CountryDto* objects returned by the *Map* service method.

2. We'll also test whether the query string parameters *pageIndex* and *pageSize* are assigned **1** and **10**, respectively, when null.

3. We will test that the *GetAllAsync* service method takes the *PagingDto* object as a parameter with the correct values. This is necessary to check that the parameters are used correctly; even if the *GetCountries* function returns a list of countries, it may not be the correct result, as the wrong parameters have been passed.

4. We'll use the same reasoning with the Map function, checking that it takes the list of Country objects returned by the GetAllAsync service method as parameters.

As you can see, we're not just testing the output but the entire behavior of the function. This may sound cumbersome, but it's very effective because we check that the parameters are correctly used. Unfortunately, many bugs occur because the parameters are being misused.

Create the Test Class

We will now create the test class with a convention that makes it easy to understand which SUT is being tested. I'm only going to test one SUT at a time in the same class, so I'm going to name my class *GetCountriesTests* to indicate that I'm only testing the *GetCountries* function without forgetting to place this test class in a folder whose name represents the tested class to which the *GetCountries* function belongs: *CountriesTests*. For clarity, I suffix folders and test class names with *Tests*. The skeleton of the *GetCountriesTests* test class should look as shown in Listing 11-2.

Listing 11-2. The GetCountriesTests class

```
using AspNetCore8MinimalApis.Endpoints;
using AspNetCore8MinimalApis.Mapping.Interfaces;
using AspNetCore8MinimalApis.Models;
using AutoFixture;
using Domain.DTOs;
using Domain.Services;
using ExpectedObjects;
using Microsoft.AspNetCore.Http.HttpResults;
using NSubstitute;
using Xunit;

namespace UnitTests.Countries;

public class GetCountriesTests
{
    public GetCountriesTests()
```

```
    {

    }

    [Fact]
    public async Task When_GetCountriesReceivesNullPaging
    ParametersAndGetAllAsyncMethodReturnsCountries_ShouldFillUp
    DefaultPagingParametersAndReturnCountries()
    {
        // Arrange

        // Act

        // Assert
    }
}
```

You'll notice the xUnit *Fact* attribute, which allows the Microsoft Test SDK to detect that the function decorated with this attribute is a unit test and appears in the Visual Studio Test Explorer, as shown in Figure 11-2.
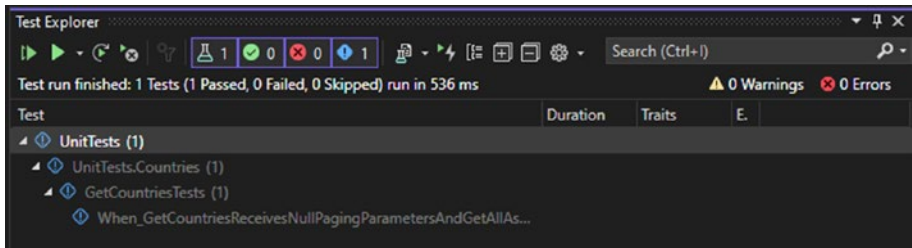
Figure 11-2. *The Visual Studio Test Explorer panel displaying discovered tests*

As for the test project, it should be arranged as shown in Figure 11-3.
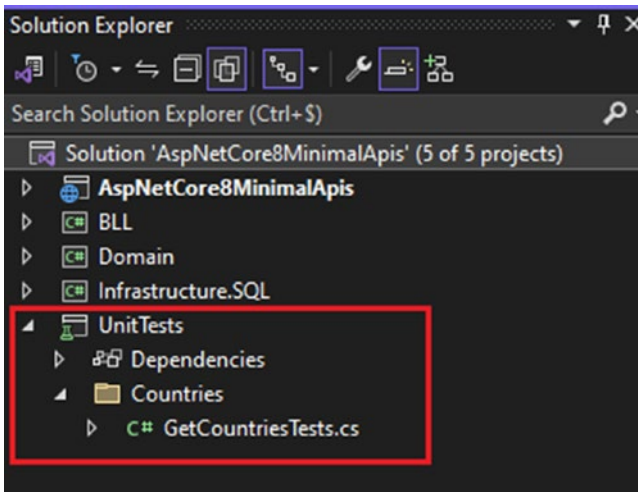
Figure 11-3. *The UnitTests project structure*

You can also see that I've named the test method with the following pattern, *When{condition}_Should{expectedBehavior}*, which allows you to understand what the function does when you read it. Don't be afraid of the length of the *When_GetCountriesReceivesNullPagingParametersAnd GetAllAsyncMethodReturnsCountries_ShouldFillUpDefaultPaging ParametersAndReturnCountries* method. By reading it, you'll quickly understand what the test will do. Finally, I've commented out three sections of this function:

1. Arrange

2. Act

3. Assert

These are the famous *AAAs* of unit testing. *Arrange* defines the variables needed to run your test. *Act* defines the SUT to be executed with the parameters defined in the *Arrange* section. Finally, *Assert* defines the checks that need to be made to validate your unit test. Remember: a unit test must be easy to read.

Write the Test

Listing 11-3 shows the final test implementation, as shown. I will also detail each step right after Listing 11-3.

Listing 11-3. The GetCountriesTests class final implementation

```
using AspNetCore8MinimalApis.Endpoints;
using AspNetCore8MinimalApis.Mapping.Interfaces;
using AspNetCore8MinimalApis.Models;
using AutoFixture;
using Domain.DTOs;
using Domain.Services;
using ExpectedObjects;
using Microsoft.AspNetCore.Http.HttpResults;
using NSubstitute;
using Xunit;

namespace UnitTests.Countries;

public class GetCountriesTests
{
    private readonly ICountryMapper _countryMapper;
    private readonly ICountryService _countryService;
    private readonly Fixture _fixture;

    public GetCountriesTests()
    {
        _countryMapper = Substitute.For<ICountryMapper>();
        _countryService = Substitute.For<ICountryService>();
        _fixture = new Fixture();
    }

    [Fact]
```

```
public async Task WhenGetCountriesReceivesNullPaging
ParametersAndGetAllAsyncMethodReturnsCountries_ShouldFillUp
DefaultPagingParametersAndReturnCountries()
{
    // Arrange
    int? pageIndex = null;
    int? pageSize = null;
    var expectedPaging = new PagingDto
    {
        PageIndex = 1,
        PageSize = 10
    }.ToExpectedObject();

    var countries = _fixture.CreateMany<CountryDto>(2).
    ToList();
    var expectedCountries = countries.ToExpectedObject();

    var mappedCountries = _fixture.CreateMany<Country>(2).
    ToList();
    var expectedMappedCountries = mappedCountries.
    ToExpectedObject();

    _countryService.GetAllAsync(Arg.Any<PagingDto>()).
    Returns(x => countries);
    _countryMapper.Map(Arg.Any<List<CountryDto>>()).
    Returns(x => mappedCountries);

    // Act
    var result = (await CountryEndpoints.
    GetCountries(pageIndex, pageSize, _countryMapper, _
    countryService)) as Ok<List<Country>>;

    // Assert
    expectedMappedCountries.ShouldEqual(result.Value);
```

```
        await _countryService.Received(1).GetAllAsync(Arg.
        Is<PagingDto>(x => expectedPaging.Matches(x)));
        _countryMapper.Received(1).Map(Arg.
        Is<List<CountryDto>>(x => expectedCountries.
        Matches(x)));
    }
}
```

Write the Constructor

First, I instantiated a fake instance of each interface (*ICountryMapper* and *ICountryService*) in the constructor using *NSubstitute*, a mocking library. We must mock these interfaces to give them a fake instance to which we'll define a precise behavior to see how our SUT reacts. This is the key to unit testing: mocking services, which must be abstracted to perform unit tests. We give a false implementation to an interface rather than using its concrete instance; for example, a unit test won't connect to a database. I've also instantiated the *Fixture* class from the *AutoFixture* library, which allows you to create filled objects automatically, a handy way to save writing time (and readability).

Note   xUnit is the unit test runner here, and it's an intelligent one. For each unit test executed, xUnit will instantiate the constructor each time, so there's no risk of service and autofix instances being altered from one test to the next, as each test will have its instances.

Write the Arrange Section

I define my parameters here. pageIndex and *pageSize* are set to null. I will test the SUT's behavior with these parameter values. Then I instantiate a *PageDto* class, initializing it with a *pageIndex* of *1* and a *pageSize* of *10*.

These values should be assigned when pageIndex and pageSize are null. I'm going to attach it to the *ToExpectedObjects* (which comes from the *ExpectedObjects* library) extension method that will allow us to test by value the *PageDto* object instance, as I want to verify further that the *GetAllAsync* service method will take the correct *PageDto* values as parameters.

Using *AutoFixture*, I will create a list instance of *Country* and *CountryDto* objects, each with a length of two elements. Here, their value doesn't matter; these instances will be defined as the return value of the *GetAllAsync* and *Map* service methods, which I'll remind you are mocked, so I'm going to define a behavior that will tell them that whatever parameters they take (using *Arg.Any<T>)*, they'll return the object list defined by *AutoFixture*. Here, it's not a question of testing the content of each object list or checking that the content is mapped by the service *Map* method but of checking that the *Map* service method takes the object list of type *Country* from the *GetAllAsync* service method as its parameter. As for the list of *CountryDto* objects returned by the service *Map* method, we're going to check that the SUT returns this same list, and we need to test this by value and not by reference, which is, as you know, how objects are compared.

Write the Act Section

In this section, I executed the SUT and retrieved its result, passing all the necessary parameters, including mocked *IMapper* and *ICountryService* service instances. In the *Assert* section, we'll test several elements.

Write the Assert Section

It's the final moment! Let's implement our checks.

First, we want to check that the SUT output (result) corresponds to what the *Map* service method returns since we're returning it as is. Before you do that, don't forget that the output result is an *IResult* type, the

Ok<ListCountryDto> type, to be exact, which is why I'm doing an implicit cast. If the latter fails, it won't generate an exception, thanks to the *as* keyword.

I then use *ExpectedObject* and its *ShouldEqual* method applied to the *expectedMappedCountries* instance and compare it with the *Value* property of the *result* output.

There's no need to test the type, as the type test is implicitly performed here: if the cast didn't work, the output (*result*) would be *null.*

Then, for each service method (*GetAllAsync* and *Map*), I check that the mocked service instance to which they belong is invoked **once**, using the *Received* extension method (from *NSubstitute*). Then I check that they receive the list of objects they're supposed to receive as parameters, using the *Arg. Is<T>* static function. Each one takes, as parameters, a delegate invoking the *Matches* extension method on the respective *expectedpaging* and *expectedCountries* objects defined with ExpectedObjects. The *Matches* extension returns a *Boolean* indicating whether the values of the expected instances match. So thanks to *ExpectedObjects*!

If we execute our test on Visual Studio, it should appear as passed, as shown in Figure 11-4.
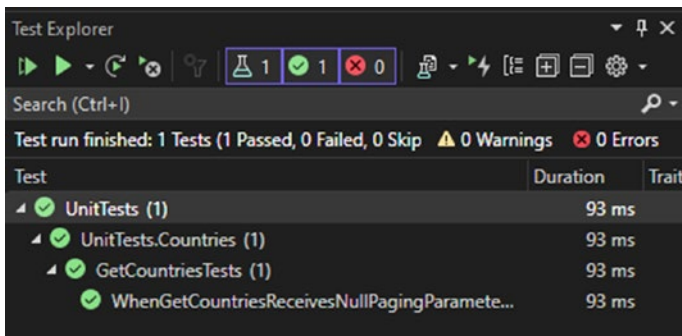
Figure 11-4. *The Test Explorer output when unit tests passed*

Summary

That's it! You've now mastered the unit test technique. You need to understand here not how to use the tools I use to do my tests but the philosophy you must adopt to test your code correctly. What I've shown you here will make your code bulletproof; we've tested everything possible to test, and we've gone beyond testing the output of a SUT. I'm confident you can extrapolate this logic to any unit test you want to perform! This book is complete, and I'd like to thank you for following me to the end!

Index

W, X, Y, Z