CHAPTER 17

Working with Data

All the examples in the earlier chapters in this part of the book have generated fresh responses for each request, which is easy to do when dealing with simple strings or small fragments of HTML. Most real projects deal with data that is expensive to produce and needs to be used as efficiently as possible. In this chapter, I describe the features that ASP.NET Core provides for caching data and caching entire responses. I also show you how to create and configure the services required to access data in a database using Entity Framework Core. Table 17-1 puts the ASP.NET Core features for working with data in context.

Note The examples in this chapter rely on the SQL Server LocalDB feature that was installed in Chapter 2. You will encounter errors if you have not installed LocalDB and the required updates.

Table 17-1. *Putting the ASP.NET Core Data Features in Context*

Question	Answer
What are they?	The features described in this chapter allow responses to be produced using data that has been previously created, either because it was created for an earlier request or because it has been stored in a database.
Why are they useful?	Most web applications deal with data that is expensive to re-create for every request. The features in this chapter allow responses to be produced more efficiently and with fewer resources.
How are they used?	Data values are cached using a service. Responses are cached by a middleware component based on the `Cache-Control` header. Databases are accessed through a service that translates LINQ queries into SQL statements.
Are there any pitfalls or limitations?	For caching, it is important to test the effect of your cache policy before deploying the application to ensure you have found the right balance between efficiency and responsiveness. For Entity Framework Core, it is important to pay attention to the queries sent to the database to ensure that they are not retrieving large amounts of data that is processed and then discarded by the application.
Are there any alternatives?	All the features described in this chapter are optional. You can elect not to cache data or responses or to use an external cache. You can choose not to use a database or to access a database using a framework other than Entity Framework Core.

Table 17-2 summarizes the chapter.

Table 17-2. *Chapter Summary*

Problem	Solution	Listing
Caching data values	Set up a cache service and use it in endpoints and middleware components to store data values	7, 8
Creating a persistent cache	Use the database-backed cache	9–14
Caching entire responses	Enable the caching middleware and set the `Cache-Control` header in responses	15, 16
Storing application data	Use Entity Framework Core	17–23, 26–28
Creating a database schema	Create and apply migrations	24, 25
Accessing data in endpoints	Consume the database context service	29
Including all request details in logging messages	Enable the sensitive data logging feature	30

Preparing for This Chapter

In this chapter, I continue to use the Platform project from Chapter 16. To prepare for this chapter, replace the contents of the `Startup.cs` file with the code shown in Listing 17-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/apress/pro-asp.net-core-3`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 17-1. Replacing the Contents of the Startup.cs File in the Platform Folder

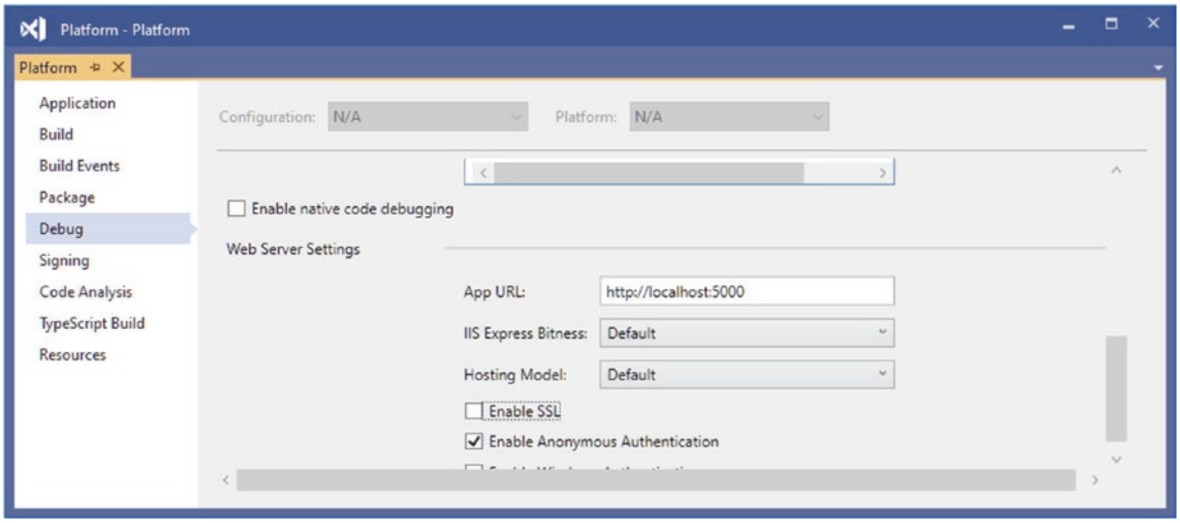
```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
            });
        }
    }
}
```

To reduce the detail level of the logging messages displayed by the application, make the changes shown in Listing 17-2 to the `appsettings.Development.json` file.

Listing 17-2. Changing the Log Detail in the appsettings.Development.json File in the Platform Folder

```
{
    "Logging": {
      "LogLevel": {
        "Default": "Debug",
        "System": "Information",
        "Microsoft": "Information",
        "Microsoft.AspnetCore.Hosting.Diagnostics": "None"
      }
    }
}
```

If you are using Visual Studio, select Project ➤ Platform Properties, navigate to the Debug tab, uncheck the Enable SSL option (as shown in Figure 17-1), and select File ➤ Save All.



Platform - Platform

Configuration: N/A Platform: N/A

Enable native code debugging

Web Server Settings

App URL: http://localhost:5000

IIS Express Bitness: Default

Hosting Model: Default

Enable SSL

[✓] Enable Anonymous Authentication

Figure 17-1. *Disabling HTTPS*

If you are using Visual Studio Code, open the `Properties/launchSettings.json` file to remove the HTTPS URL, as shown in Listing 17-3.

Listing 17-3. Disabling SSL in the launchSettings.json File in the Platform/Properties Folder

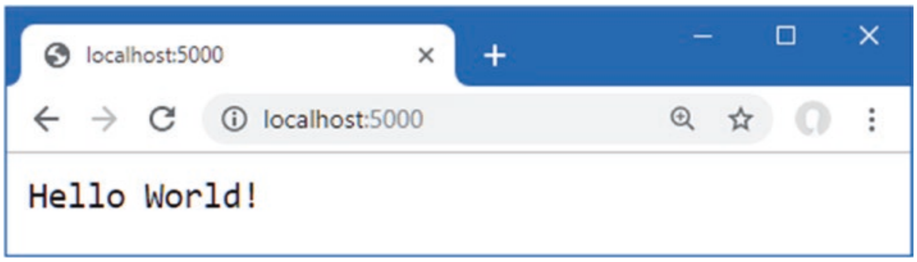
```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Platform": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000"
    }
  }
}
```

Start the application by selecting Start Without Debugging or Run Without Debugging from the Debug menu or by opening a new PowerShell command prompt, navigating to the Platform project folder (which contains the Platform.csproj file), and running the command shown in Listing 17-4.

Listing 17-4. Starting the ASP.NET Core Runtime

```
dotnet run
```

If the application was started using Visual Studio or Visual Studio Code, a new browser window will open and display the content shown in Figure 17-2. If the application was started from the command line, open a new browser tab and navigate to https://localhost:5000; you will see the content shown in Figure 17-2.



```
Hello World!
```

Figure 17-2. *Running the example application*

Caching Data

In most web applications, there will be some items of data that are relatively expensive to generate but are required repeatedly. The exact nature of the data is specific to each project, but repeatedly performing the same set of calculations can increase the resources required to host the application and drive up hosting costs. To represent an expensive response, add a class file called SumEndpoint.cs to the Platform folder with the code shown in Listing 17-5.

Listing 17-5. The Contents of the SumEndpoint.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context) {
            int count = int.Parse((string)context.Request.RouteValues["count"]);
            long total = 0;
            for (int i = 1; i <= count; i++) {
                total += i;
            }
            string totalString = $"({ DateTime.Now.ToLongTimeString() }) {total}";
            await context.Response.WriteAsync(
                $"({DateTime.Now.ToLongTimeString()}) Total for {count}"
                + $" values:\n{totalString}\n");
        }
    }
}
```

Listing 17-6 creates a route that uses the endpoint, which is applied using the `MapEndpoint` extension methods created in Chapter 13.

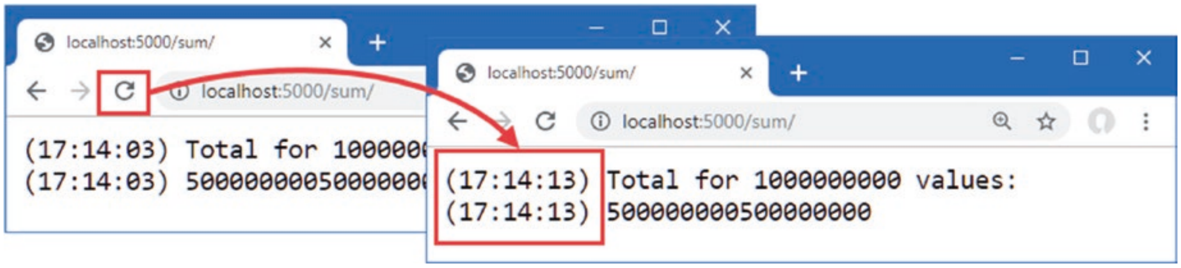
Listing 17-6. Adding an Endpoint in the Startup.cs File in the Platform Folder

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

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {

                endpoints.MapEndpoint<SumEndpoint>("/sum/{count:int=1000000000}");

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

Restart ASP.NET Core and use a browser to request `https://localhost:5000/sum`. The endpoint will sum 1,000,000,000 integer values and produce the result shown in Figure 17-3.

Reload the browser window, and the endpoint will repeat the calculation. Both the timestamps in the response change, as shown in the figure, indicating that every part of the response was produced fresh for each request.

■ **Tip** You may need to increase or decrease the default value for the route parameter based on the capabilities of your machine. Try to find a value that takes two or three seconds to produce the result—just long enough that you can tell when the calculation is being performed but not so long that you can step out for coffee while it happens.



```
(17:14:03) Total for 100000
(17:14:03) 500000000500000
```

```
(17:14:13) Total for 1000000000 values:
(17:14:13) 500000000500000000
```

Figure 17-3. *An expensive response*

Caching Data Values

ASP.NET Core provides a service that can be used to cache data values through the IDistributedCache interface. Listing 17-7 revises the endpoint to declare a dependency on the service and use it to cache calculated values.

Listing 17-7. Using the Cache Service in the SumEndpoint.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Caching.Distributed;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context, IDistributedCache cache) {
            int count = int.Parse((string)context.Request.RouteValues["count"]);
            string cacheKey = $"sum_{count}";
            string totalString = await cache.GetStringAsync(cacheKey);
            if (totalString == null) {
                long total = 0;
                for (int i = 1; i <= count; i++) {
                    total += i;
                }
                totalString = $"({ DateTime.Now.ToLongTimeString() }) {total}";
                await cache.SetStringAsync(cacheKey, totalString,
                    new DistributedCacheEntryOptions {
                        AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(2)
                    });
            }
            await context.Response.WriteAsync(
                $"({DateTime.Now.ToLongTimeString()}) Total for {count}"
                + $" values:\n{totalString}\n");
        }
    }
}
```

The cache service can store only byte arrays, which can be restrictive but allows for a range of IDistributedCache implementations to be used. There are extension methods available that allow strings to be used, which is a more convenient way of caching most data. Table 17-3 describes the most useful methods for using the cache.

Table 17-3. *Useful IDistributedCache Methods*

Name	Description
`GetString(key)`	This method returns the cached string associated with the specified key, or `null` if there is no such item.
`GetStringAsync(key)`	This method returns a `Task<string>` that produces the cached string associated with the key, or `null` if there is no such item.
`SetString(key, value, options)`	This method stores a string in the cache using the specified key. The cache entry can be configured with an optional `DistributedCacheEntryOptions` object.
`SetStringAsync(key, value, options)`	This method asynchronously stores a string in the cache using the specified key. The cache entry can be configured with an optional `DistributedCacheEntryOptions` object.
`Refresh(key)`	This method resets the expiry interval for the value associated with the key, preventing it from being flushed from the cache.
`RefreshAsync(key)`	This method asynchronously resets the expiry interval for the value associated with the key, preventing it from being flushed from the cache.
`Remove(key)`	This method removes the cached item associated with the key.
`RemoveAsync(key)`	This method asynchronously removes the cached item associated with the key.

By default, entries remain in the cache indefinitely, but the `SetString` and `SetStringAsync` methods accept an optional `DistributedCacheEntryOptions` argument that is used to set an expiry policy, which tells the cache when to eject the item. Table 17-4 shows the properties defined by the `DistributedCacheEntryOptions` class.

Table 17-4. *The DistributedCacheEntryOptions Properties*

Name	Description
`AbsoluteExpiration`	This property is used to specify an absolute expiry date.
`AbsoluteExpirationRelativeToNow`	This property is used to specify a relative expiry date.
`SlidingExpiration`	This property is used to specify a period of inactivity, after which the item will be ejected from the cache if it hasn't been read.

In Listing 17-7, the endpoint uses the `GetStringAsync` to see whether there is a cached result available from a previous request. If there is no cached value, the endpoint performs the calculation and caches the result using `SetStringAsync` method, with the `AbsoluteExpirationRelativeToNow` property to tell the cache to eject the item after two minutes.

```
...
await cache.SetStringAsync(cacheKey, totalStr,
    new DistributedCacheEntryOptions {
        AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(2)
    });
...
```

The next step is to set up the cache service in the `Startup` class, as shown in Listing 17-8.

Listing 17-8. Adding a Service in the Startup.cs File in the Platform Folder

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

namespace Platform {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddDistributedMemoryCache(opts => {
                opts.SizeLimit = 200;
            });
        }

        public void Configure(IApplicationBuilder app) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {

                endpoints.MapEndpoint<SumEndpoint>("/sum/{count:int=1000000000}");

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

AddDistributedMemoryCache is the same method I used in Chapter 16 to provide the data store for session data. This is one of the three methods used to select an implementation for the IDistributedCache service, as described in Table 17-5.

Table 17-5. *The Cache Service Implementation Methods*

Name	Description
AddDistributedMemoryCache	This method sets up an in-memory cache.
AddDistributedSqlServerCache	This method sets up a cache that stores data in SQL Server and is available when the Microsoft.Extensions.Caching.SqlServer package is installed. See the "Caching Responses" section for details.
AddStackExchangeRedisCache	This method sets up a Redis cache and is available when the Microsoft.Extensions.Caching.Redis package is installed.

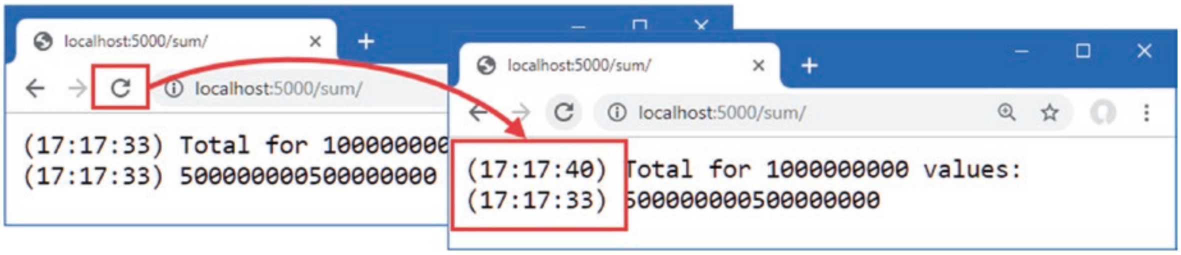
Listing 17-8 uses the AddDistributedMemoryCache method to create an in-memory cache as the implementation for the IDistributedCache service. This cache is configured using the MemoryCacheOptions class, whose most useful properties are described in Table 17-6.

Table 17-6. *Useful MemoryCacheOptions Properties*

Name	Description
ExpirationScanFrequency	This property is used to set a TimeSpan that determines how often the cache scans for expired items.
SizeLimit	This property specifies the maximum number of items in the cache. When the size is reached, the cache will eject items.
CompactionPercentage	This property specifies the percentage by which the size of the cache is reduced when SizeLimit is reached.

The statement in Listing 17-8 uses the SizeLimit property to restrict the cache to 200 items. Care must be taken when using an in-memory cache to find the right balance between allocating enough memory for the cache to be effective without exhausting server resources.

To see the effect of the cache, restart ASP.NET Core and request the `http://localhost:5000/sum` URL. Reload the browser, and you will see that only one of the timestamps will change, as shown in Figure 17-4. This is because the cache has provided the calculation response, which allows the endpoint to produce the result without having to repeat the calculation.



```
(17:17:33) Total for 1000000000
(17:17:33) 500000000500000000
```

```
(17:17:40) Total for 1000000000 values:
(17:17:33) 500000000500000000
```

Figure 17-4. *Caching data values*

If you wait for two minutes and then reload the browser, then both timestamps will change because the cached result will have been ejected, and the endpoint will have to perform the calculation to produce the result.

Using a Shared and Persistent Data Cache

The cache created by the `AddDistributedMemoryCache` method isn't distributed, despite the name. The items are stored in memory as part of the ASP.NET Core process, which means that applications that run on multiple servers or containers don't share cached data. It also means that the contents of the cache are lost when ASP.NET Core is stopped.

The `AddDistributedSqlServerCache` method stores the cache data in a SQL Server database, which can be shared between multiple ASP.NET Core servers and which stores the data persistently.

The first step is to create a database that will be used to store the cached data. You can store the cached data alongside the application's other data, but for this chapter, I am going to use a separate database, which will be named `CacheDb`. You can create the database using Azure Data Studio or SQL Server Management Studio, both of which are available for free from Microsoft. Databases can also be created from the command line using `sqlcmd`. Open a new PowerShell command prompt and run the command shown in Listing 17-9 to connect to the LocalDB server.

■ **Tip** The `sqlcmd` tool should have been installed as part of the Visual Studio workload or as part of the SQL Server Express installation. If it has not been installed, then you can download an installer from `https://docs.microsoft.com/en-us/sql/tools/sqlcmd-utility?view=sql-server-2017`.

Listing 17-9. Connecting to the Database

```
sqlcmd -S "(localdb)\MSSQLLocalDB"
```

Pay close attention to the argument that specifies the database. There is one backslash, which is followed by `MSSQLLocalDB`. It can be hard to spot the repeated letters: M**S**-**SQL**-**L**ocalDB (but without the hyphens).

When the connection has been established, you will see a `1>` prompt. Enter the commands shown in Listing 17-10 and press the Enter key after each command.

■ **Caution** If you are using Visual Studio, you must apply the updates for SQL Server described in Chapter 2. The version of SQL Server that is installed by default when you install Visual Studio cannot create LocalDB databases.

Listing 17-10. Creating the Database

```
CREATE DATABASE CacheDb
GO
```

If no errors are reported, then enter `exit` and press Enter to terminate the connection. The next step is to run the command shown in Listing 17-11 to create a table in the new database, which uses a global .NET Core tool to prepare the database.

■ **Tip** If you need to reset the cache database, use the command in Listing 17-9 to open a connection and use the command `DROP DATABASE CacheDB`. You can then re-create the database using the commands in Listing 17-10.

Listing 17-11. Creating the Cache Database Table

```
dotnet sql-cache create "Server=(localdb)\MSSQLLocalDB;Database=CacheDb" dbo DataCache
```

The arguments for this command are the connection string that specifies the database, the schema, and the name of the table that will be used to store the cached data. Enter the command on a single line and press Enter. It will take a few seconds for the tool to connect to the database. If the process is successful, you will see the following message:

```
Table and index were created successfully.
```

Creating the Persistent Cache Service

Now that the database is ready, I can create the service that will use it to store cached data. To add the NuGet package required for SQL Server caching support, open a new PowerShell command prompt, navigate to the `Platform` project folder, and run the command shown in Listing 17-12. (If you are using Visual Studio, you can add the package by selecting Project ➤ Manage Nuget Packages.)

Listing 17-12. Adding a Package to the Project

```
dotnet add package Microsoft.Extensions.Caching.SqlServer --version 3.1.1
```

The next step is to define a connection string, which describes the database connection in the JSON configuration file, as shown in Listing 17-13.

■ **Note** The cache created by the `AddDistributedSqlServerCache` method is distributed, meaning that multiple applications can use the same database and share cache data. If you are deploying the same application to multiple servers or containers, all instances will be able to share cached data. If you are sharing a cache between different applications, then you should pay close attention to the keys you use to ensure that applications receive the data types they expect.

Listing 17-13. Defining a Connection String in the appsettings.json File in the Platform Folder

```
{
    "Location": {
        "CityName": "Buffalo"
    },
    "Logging": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft": "Warning",
        "Microsoft.Hosting.Lifetime": "Information"
      }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": {
        "CacheConnection": "Server=(localdb)\\MSSQLLocalDB;Database=CacheDb"
      }
  }
```

Notice that the connection string uses two backslash characters (\\) to escape the character in the JSON file. Listing 17-14 change the implementation for the cache service in the Startup class to use SQL Server with the connection string from Listing 17-13.

Listing 17-14. Using a Persistent Data Cache in the Startup.cs File in the Platform Folder

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

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration {get; set;}

        public void ConfigureServices(IServiceCollection services) {

            services.AddDistributedSqlServerCache(opts => {
                opts.ConnectionString
                    = Configuration["ConnectionStrings:CacheConnection"];
                opts.SchemaName = "dbo";
                opts.TableName = "DataCache";
            });
        }

        public void Configure(IApplicationBuilder app) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {

                endpoints.MapEndpoint<SumEndpoint>("/sum/{count:int=1000000000}");

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

The IConfiguration service is used to access the connection string from the application's configuration data. The cache service is created using the AddDistributedSqlServerCache method and is configured using an instance of the SqlServerCacheOptions class, whose most useful properties are described in Table 17-7.

Table 17-7. *Useful SqlServerCacheOptions Properties*

Name	Description
ConnectionString	This property specifies the connection string, which is conventionally stored in the JSON configuration file and accessed through the IConfguration service.
SchemaName	This property specifies the schema name for the cache table.
TableName	This property specifies the name of the cache table.
ExpiredItemsDeletionInterval	This property specifies how often the table is scanned for expired items. The default is 30 minutes.
DefaultSlidingExpiration	This property specifies how long an item remains unread in the cache before it expires. The default is 20 minutes.

The listing uses the ConnectionString, SchemaName, and TableName properties to configure the cache middleware to use the database table. Restart ASP.NET Core and use a browser to request the http://localhost:5000/sum URL. There is no change in the response produced by the application, which is shown in Figure 17-4, but you will find that the cached responses are persistent and will be used even when you restart ASP.NET Core.

CACHING SESSION-SPECIFIC DATA VALUES

When you use the IDistributedCache service, the data values are shared between all requests. If you want to cache different data values for each user, then you can use the session middleware described in Chapter 16. The session middleware relies on the IDistributedCache service to store its data, which means that session data will be stored persistently and be available to a distributed application when the AddDistributedSqlServerCache method is used.

Caching Responses

An alternative to caching individual data items is to cache entire responses, which can be a useful approach if a response is expensive to compose and is likely to be repeated. Caching responses requires the addition of a service and a middleware component, as shown in Listing 17-15.

Listing 17-15. Configuring Response Caching in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Platform.Services;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration {get; set;}
        public void ConfigureServices(IServiceCollection services) {
            services.AddDistributedSqlServerCache(opts => {
                opts.ConnectionString
                    = Configuration["ConnectionStrings:CacheConnection"];
                opts.SchemaName = "dbo";
                opts.TableName = "DataCache";
            });
            services.AddResponseCaching();
            services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();
        }

        public void Configure(IApplicationBuilder app) {
            app.UseDeveloperExceptionPage();
            app.UseResponseCaching();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {

                endpoints.MapEndpoint<SumEndpoint>("/sum/{count:int=1000000000}");

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

The AddResponseCaching method is used in the ConfigureServices method to set up the service used by the cache. The middleware component is added with the UseResponseCaching method, which should be called before any endpoint or middleware that needs its responses cached.

I have also defined the IResponseFormatter service, which I used to explain how dependency injection works in Chapter 14. Response caching is used only in certain circumstances, and, as I explain shortly, demonstrating the feature requires an HTML response.

■ **Note** The response caching feature does not use the IDistributedCache service. Responses are cached in memory and are not distributed.

In Listing 17-16, I have updated the SumEndpoint class so that it requests response caching instead of caching just a data value.

Listing 17-16. Using Response Caching in the SumEndpoint.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Caching.Distributed;
using Microsoft.AspNetCore.Routing;
using Platform.Services;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context, IDistributedCache cache,
                IResponseFormatter formatter, LinkGenerator generator) {
```

```
            int count = int.Parse((string)context.Request.RouteValues["count"]);
            long total = 0;
            for (int i = 1; i <= count; i++) {
                total += i;
            }
            string totalString = $"({ DateTime.Now.ToLongTimeString() }) {total}";

            context.Response.Headers["Cache-Control"] = "public, max-age=120";

            string url = generator.GetPathByRouteValues(context, null,
                new { count = count });

            await formatter.Format(context,
                $"<div>({DateTime.Now.ToLongTimeString()}) Total for {count}"
                + $" values:</div><div>{totalString}</div>"
                + $"<a href={url}>Reload</a>");
        }
    }
}
```

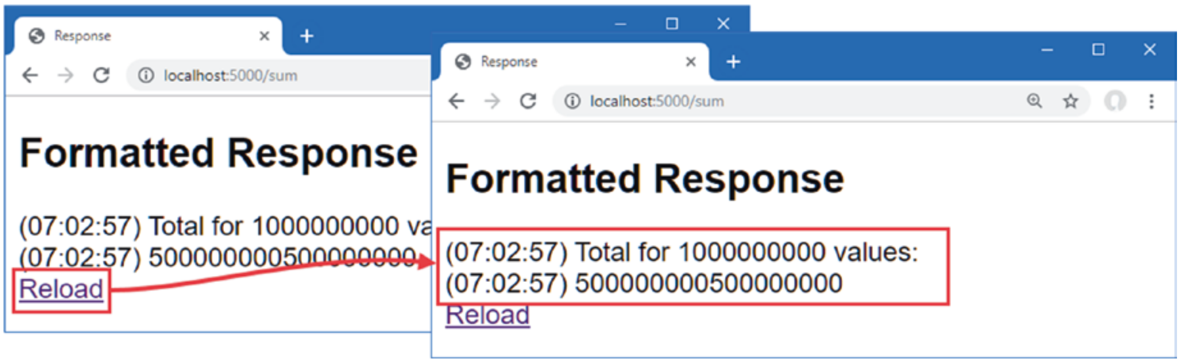
Some of the changes to the endpoint enable response caching, but others are just to demonstrate that it is working. For enabling response caching, the important statement is the one that adds a header to the response, like this:

```
...
context.Response.Headers["Cache-Control"] = "public, max-age=120";
...
```

The `Cache-Control` header is used to control response caching. The middleware will only cache responses that have a `Cache-Control` header that contains the `public` directive. The `max-age` directive is used to specify the period that the response can be cached for, expressed in seconds. The `Cache-Control` header used in Listing 17-16 enables caching and specifies that responses can be cached for two minutes.

Enabling response caching is simple, but checking that it is working requires care. When you reload the browser window or press Return in the URL bar, browsers will include a `Cache-Control` header in the request that sets the `max-age` directive to zero, which bypasses the response cache and causes a new response to be generated by the endpoint. The only reliable way to request a URL without the `Cache-Control` header is to navigate using an HTML anchor element, which is why the endpoint in Listing 17-16 uses the `IResponseFormatter` service to generate an HTML response and uses the `LinkGenerator` service to create a URL that can be used in the anchor element's `href` attribute.

To check the response cache, restart ASP.NET Core and use the browser to request `http://localhost:5000/sum`. Once the response has been generated, click the Reload link to request the same URL. You will see that neither of the timestamps in the response change, indicating that the entire response has been cached, as shown in Figure 17-5.



```
(07:02:57) Total for 1000000000 values:
(07:02:57) 500000000500000000
```

Figure 17-5. *Caching responses*

The Cache-Control header can be combined with the Vary header to provide fine-grained control over which requests are cached. See https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control and https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Vary for details of the features provided by both headers.

COMPRESSING RESPONSES

ASP.NET Core includes middleware that will compress responses for browsers that have indicated they can handle compressed data. The middleware is added to the pipeline with the UseResponseCompression method. Compression is a trade-off between the server resources required for compression and the bandwidth required to deliver content to the client, and it should not be switched on without testing to determine the performance impact.

Using Entity Framework Core

Not all data values are produced directly by the application, and most projects will need to access data in a database. Entity Framework Core is well-integrated into the ASP.NET Core platform, with good support for creating a database from C# classes and for creating C# classes to represent an existing database. In the sections that follow, I demonstrate the process for creating a simple data model, using it to create a database, and querying that database in an endpoint.

WORKING WITH ENTITY FRAMEWORK CORE

The most common complaint about Entity Framework Core is poor performance. When I review projects that have Entity Framework Core performance issues, the problem is almost always because the development team has treated Entity Framework Core as a black box and not paid attention to the SQL queries that are sent to the database. Not all LINQ features can be translated into SQL, and the most common problem is a query that retrieves large amounts of data from the database, which is then discarded after it has been reduced to produce a single value.

Using Entity Framework Core requires a good understanding of SQL and ensuring that the LINQ queries made by the application are translated into efficient SQL queries. There are rare applications that have high-performance data requirements that cannot be met by Entity Framework Core, but that isn't the case for most typical web applications.

That is not to say that Entity Framework Core is perfect. It has its quirks and requires an investment in time to become proficient. If you don't like the way that Entity Framework Core works, then you may prefer to use an alternative, such as Dapper (https://github.com/StackExchange/Dapper). But if your issue is that queries are not being performed fast enough, then you should spend some time exploring how those queries are being processed, which you can do using the techniques described in the remainder of the chapter.

Installing Entity Framework Core

Entity Framework Core requires a global tool package that is used to manage databases from the command line and to manage packages for the project that provide data access. To install the tools package, open a new PowerShell command prompt and run the commands shown in Listing 17-17.

Listing 17-17. Installing the Entity Framework Core Global Tool Package

```
dotnet tool uninstall --global dotnet-ef
dotnet tool install --global dotnet-ef --version 3.1.1
```

The first command removes any existing version of the dotnet-ef package, and the second command installs the version required for the examples in this book. This package provides the dotnet ef commands that you will see in later examples. To ensure the package is working as expected, run the command shown in Listing 17-18.

Listing 17-18. Testing the Entity Framework Core Global Tool

```
dotnet ef --help
```

This command shows the help message for the global tool and produces the following output:

```
Entity Framework Core .NET Command-line Tools 3.1.1
Usage: dotnet ef [options] [command]
Options:
  --version        Show version information
  -h|--help        Show help information
  -v|--verbose     Show verbose output.
  --no-color       Don't colorize output.
  --prefix-output  Prefix output with level.
Commands:
  database    Commands to manage the database.
  dbcontext   Commands to manage DbContext types.
  migrations  Commands to manage migrations.
Use "dotnet ef [command] --help" for more information about a command.
```

Entity Framework Core also requires packages to be added to the project. If you are using Visual Studio Code or prefer working from the command line, navigate to the `Platform` project folder (the folder that contains the `Platform.csproj` file) and run the commands shown in Listing 17-19.

Listing 17-19. Adding Entity Framework Core Packages to the Project

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 3.1.1
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 3.1.1
```

Creating the Data Model

For this chapter, I am going to define the data model using C# classes and use Entity Framework Core to create the database and schema. Create the `Platform/Models` folder and add to it a class file called `Calculation.cs` with the contents shown in Listing 17-20.

Listing 17-20. The Contents of the Calculation.cs File in the Platform/Models Folder

```
namespace Platform.Models {

    public class Calculaton {
        public long Id { get; set; }
        public int Count { get; set; }
        public long Result { get; set; }
    }
}
```

You can see more complex data models in other chapters, but for this example, I am going to keep with the theme of this chapter and model the calculation performed in earlier examples. The `Id` property will be used to create a unique key for each object stored in the database, and the `Count` and `Result` properties will describe a calculation and its result.

Entity Framework Core uses a context class that provides access to the database. Add a file called `CalculationContext.cs` to the `Platform/Models` folder with the content shown in Listing 17-21.

Listing 17-21. The Contents of the CalculationContext.cs File in the Platform/Models Folder

```
using Microsoft.EntityFrameworkCore;

namespace Platform.Models {

    public class CalculationContext: DbContext {

        public CalculationContext(DbContextOptions<CalculationContext> opts)
            : base(opts) {}

        public DbSet<Calculaton> Calculations { get; set; }
    }
}
```

The `CalculationContext` class defines a constructor that is used to receive an options object that is passed on to the base constructor. The `Calculations` property provides access to the `Calculation` objects that Entity Framework Core will retrieve from the database.

Configuring the Database Service

Access to the database is provided through a service that is configured in the `Startup` class, as shown in Listing 17-22.

Listing 17-22. Configuring the Data Service in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Platform.Services;
using Microsoft.EntityFrameworkCore;
using Platform.Models;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration {get; set;}

        public void ConfigureServices(IServiceCollection services) {
            services.AddDistributedSqlServerCache(opts => {
                opts.ConnectionString
                    = Configuration["ConnectionStrings:CacheConnection"];
                opts.SchemaName = "dbo";
                opts.TableName = "DataCache";
            });
            services.AddResponseCaching();
            services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();

            services.AddDbContext<CalculationContext>(opts => {
                opts.UseSqlServer(Configuration["ConnectionStrings:CalcConnection"]);
            });
        }
        public void Configure(IApplicationBuilder app) {
            app.UseDeveloperExceptionPage();
            app.UseResponseCaching();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {

                endpoints.MapEndpoint<SumEndpoint>("/sum/{count:int=1000000000}");

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

The `AddDbContext` method creates a service for an Entity Framework Core context class. The method receives an options object that is used to select the database provider, which is done with the `UseSqlServer` method. The `IConfiguration` service is used to get the connection string for the database, which is defined in Listing 17-23.

Listing 17-23. Defining a Connection String in the appsettings.json File in the Platform Folder

```
{
    "Location": {
        "CityName": "Buffalo"
    },
    "Logging": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft": "Warning",
        "Microsoft.Hosting.Lifetime": "Information",
        "Microsoft.EntityFrameworkCore": "Information"
      }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": {
        "CacheConnection": "Server=(localdb)\\MSSQLLocalDB;Database=CacheDb",
        "CalcConnection": "Server=(localdb)\\MSSQLLocalDB;Database=CalcDb"
      }
  }
```

The listing also sets the logging level for the `Microsoft.EntityFrameworkCore` category, which will show the SQL statements that are used by Entity Framework Core to query the database.

■ **Tip** Set the `MultipleActiveResultSets` option to `True` for connection strings that will be used to make queries with multiple result sets. You can see an example of this option set in the connection strings for the SportsStore project in Chapter 7.

Creating and Applying the Database Migration

Entity Framework Core manages the relationship between data model classes and the database using a feature called *migrations*. When changes are made to the model classes, a new migration is created that modifies the database to match those changes. To create the initial migration, which will create a new database and prepare it to store `Calculation` objects, open a new PowerShell command prompt, navigate to the folder that contains the `Platform.csproj` file, and run the command shown in Listing 17-24.

Listing 17-24. Creating a Migration

```
dotnet ef migrations add Initial
```

The dotnet ef commands relate to Entity Framework Core. The command in Listing 17-24 creates a new migration named Initial, which is the name conventionally given to the first migration for a project. You will see that a Migrations folder has been added to the project and that it contains class files whose statements prepare the database so that it can store the objects in the data model. To apply the migration, run the command shown in Listing 17-25 in the Platform project folder.

Listing 17-25. Applying a Migration

```
dotnet ef database update
```

This command executes the commands in the migration created in Listing 17-24 and uses them to prepare the database, which you can see in the SQL statements written to the command prompt.

Seeding the Database

Most applications require some seed data, especially during development. Entity Framework Core does provide a database seeding feature, but it is of limited use for most projects because it doesn't allow data to be seeded where the database allocates unique keys to the objects it stores. This is an important feature in most data models because it means the application doesn't have to worry about allocating unique key values.

A more flexible approach is to use the regular Entity Framework Core features to add seed data to the database. Create a file called SeedData.cs in the Platform/Models folder with the code shown in Listing 17-26.

Listing 17-26. The Contents of the SeedData.cs File in the Platform/Models Folder

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System.Collections.Generic;
using System.Linq;

namespace Platform.Models {
    public class SeedData {
        private CalculationContext context;
        private ILogger<SeedData> logger;

        private static Dictionary<int, long> data
            = new Dictionary<int, long>() {
                {1, 1}, {2, 3}, {3, 6}, {4, 10}, {5, 15},
                {6, 21}, {7, 28}, {8, 36}, {9, 45}, {10, 55}
            };

        public SeedData(CalculationContext dataContext, ILogger<SeedData> log) {
            context = dataContext;
            logger = log;
        }

        public void SeedDatabase() {
            context.Database.Migrate();
            if (context.Calculations.Count() == 0) {
                logger.LogInformation("Preparing to seed database");
                context.Calculations!.AddRange(data.Select(kvp => new Calculaton() {
                    Count = kvp.Key, Result = kvp.Value
                }));
                context.SaveChanges();
                logger.LogInformation("Database seeded");
            } else {
                logger.LogInformation("Database not seeded");
            }
        }
    }
}
```

The SeedData class declares constructor dependencies on the CalculationContext and ILogger<T> types, which are used in the SeedDatabase method to prepare the database. The context's Database.Migrate method is used to apply any pending migrations to the database, and the Calculations property is used to store new data using the AddRange method, which accepts a sequence of Calculation objects.

The new objects are stored in the database using the SaveChanges method. To use the SeedData class, make the changes shown in Listing 17-27 to the Startup class.

Listing 17-27. Enabling Database Seeding in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Platform.Services;
using Microsoft.EntityFrameworkCore;
using Platform.Models;
using Microsoft.Extensions.Hosting;
using Microsoft.AspNetCore.Hosting;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration {get; set;}

        public void ConfigureServices(IServiceCollection services) {
            services.AddDistributedSqlServerCache(opts => {
                opts.ConnectionString
                    = Configuration["ConnectionStrings:CacheConnection"];
                opts.SchemaName = "dbo";
                opts.TableName = "DataCache";
            });
            services.AddResponseCaching();
            services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();

            services.AddDbContext<CalculationContext>(opts => {
                opts.UseSqlServer(Configuration["ConnectionStrings:CalcConnection"]);
            });
            services.AddTransient<SeedData>();
        }

        public void Configure(IApplicationBuilder app,
                IHostApplicationLifetime lifetime, IWebHostEnvironment env,
                SeedData seedData) {
            app.UseDeveloperExceptionPage();
            app.UseResponseCaching();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {

                endpoints.MapEndpoint<SumEndpoint>("/sum/{count:int=1000000000}");

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

            bool cmdLineInit = (Configuration["INITDB"] ?? "false") == "true";
            if (env.IsDevelopment() || cmdLineInit) {
                seedData.SeedDatabase();
                if (cmdLineInit) {
                    lifetime.StopApplication();
                }
            }
        }
    }
}
```

The statement in the ConfigureServices method creates a SeedData service. Although services are generally defined using interfaces, as described in Chapter 14, there are versions of the AddSingleton, AddScoped, and AddTransient methods that create a service using a single type argument. This makes it easy to instantiate a class that has constructor dependencies and allows the use of the service lifecycles. The statement in Listing 17-27 creates a transient service, which means that a new SeedData object will be created to resolve each dependency on the service.

The additions to the Configure method allow the database to be seeded in two situations. If the hosting environment is Development, the database will be seeded automatically as the application starts. It can also be useful to seed the database explicitly, especially when setting up the application for staging or production testing. This statement checks for a configuration setting named INITDB:

```
...
bool cmdLineInit = (Configuration["INITDB"] ?? "false") == "true";
...
```

This setting can be supplied on the command line to seed the database, after which the application is terminated using the IHostApplicationLifetime service. This is one of the less-used services, but it provides events that are triggered when the application is started and stopped, and it provides the StopApplication method, which terminates the application. To see the database, open a new PowerShell command prompt, navigate to the project folder, and run the command shown in Listing 17-28.

Listing 17-28. Seeding the Database

```
dotnet run INITDB=true
```

The application will start, and the database will be seeded with the results for the ten calculations defined by the SeedData class, after which the application will be terminated. During the seeding process, you will see the SQL statements that are sent to the database, which check to see whether there are any pending migrations, count the number of rows in the table used to store Calculation data, and, if the table is empty, add the seed data.

If you prefer not to use the command line, then you can seed the database by selecting Debug ➤ Start Without Debugging. As long as the environment is Development, the database will be seeded as part of the normal ASP.NET Core start sequence.

■ **Note** If you need to reset the database, you can use the `dotnet ef database drop --force` command. You can then use `dotnet run INITDB=true` to re-create and seed the database again.

Using Data in an Endpoint

Endpoints and middleware components access Entity Framework Core data by declaring a dependency on the context class and using its DbSet<T> properties to perform LINQ queries. The LINQ queries are translated into SQL and sent to the database. The tabular data received from the database is used to create data model objects that are used to produce responses. Listing 17-29 updates the SumEndpoint class to use Entity Framework Core.

Listing 17-29. Using a Database in the SumEndpoint.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Caching.Distributed;
using Microsoft.AspNetCore.Routing;
using Platform.Services;
using Platform.Models;
using System.Linq;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context,
                CalculationContext dataContext) {
            int count = int.Parse((string)context.Request.RouteValues["count"]);
            long total = dataContext.Calculations
                .FirstOrDefault(c => c.Count == count)?.Result ?? 0;
            if (total == 0) {
                for (int i = 1; i <= count; i++) {
                    total += i;
                }
                dataContext.Calculations!
                    .Add(new Calculaton() { Count = count, Result = total});
                await dataContext.SaveChangesAsync();
            }
            string totalString = $"({ DateTime.Now.ToLongTimeString() }) {total}";
            await context.Response.WriteAsync(
                $"({DateTime.Now.ToLongTimeString()}) Total for {count}"
                + $" values:\n{totalString}\n");
        }
    }
}
```

The endpoint uses the LINQ FirstOrDefault to search for a stored Calculation object for the calculation that has been requested like this:

```
...
dataContext.Calculations.FirstOrDefault(c => c.Count == count)?.Result ?? 0;
...
```

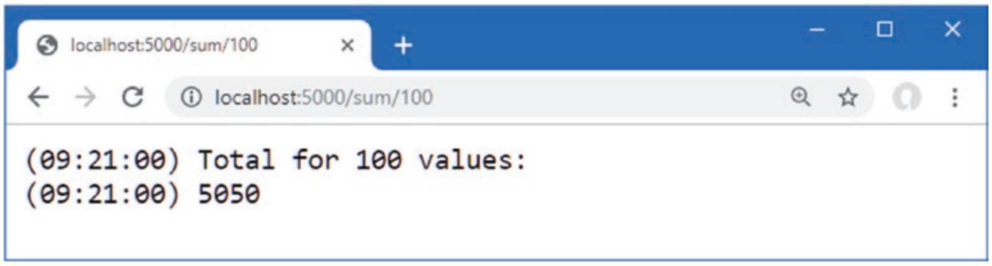
If an object has been stored, it is used to prepare the response. If not, then the calculation is performed, and a new Calculation object is stored by these statements:

```
...
dataContext.Calculations!.Add(new Calculaton() { Count = count, Result = total});
await dataContext.SaveChangesAsync();
...
```

The Add method is used to tell Entity Framework Core that the object should be stored, but the update isn't performed until the SaveChangesAsync method is called. To see the effect of the changes, restart ASP.NET Core MVC (without the INITDB argument if you are using the command line) and request the http://localhost:5000/sum/10 URL. This is one of the calculations with which the database has been seeded, and you will be able to see the query sent to the database in the logging messages produced by the application.

```
...
Executing DbCommand [Parameters=[@__count_0='?' (DbType = Int32)],
   CommandType='Text', CommandTimeout='30']
SELECT TOP(1) [c].[Id], [c].[Count], [c].[Result]
FROM [Calculations] AS [c]
WHERE ([c].[Count] = @__count_0) AND @__count_0 IS NOT NULL
...
```

If you request http://localhost:5000/sum/100, the database will be queried, but no result will be found. The endpoint performs the calculation and stores the result in the database before producing the result shown in Figure 17-6.



```
(09:21:00) Total for 100 values:
(09:21:00) 5050
```

Figure 17-6. Performing a calculation

Once a result has been stored in the database, subsequent requests for the same URL will be satisfied using the stored data. You can see the SQL statement used to store the data in the logging output produced by Entity Framework Core.

```
...
Executing DbCommand [Parameters=[@p0='?' (DbType = Int32), @p1='?' (DbType = Int64)],
    CommandType='Text', CommandTimeout='30']
SET NOCOUNT ON;
INSERT INTO [Calculations] ([Count], [Result])
VALUES (@p0, @p1);
SELECT [Id]
FROM [Calculations]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
...
```

■ **Note** Notice that the data retrieved from the database is not cached and that each request leads to a new SQL query. Depending on the frequency and complexity of the queries you require, you may want to cache data values or responses using the techniques described earlier in the chapter.

Enabling Sensitive Data Logging

Entity Framework Core doesn't include parameter values in the logging messages it produces, which is why the logging output contains question marks, like this:

```
...
Executing DbCommand [Parameters=[@__count_0='?' (DbType = Int32)], CommandType='Text', CommandTimeout='30']
...
```

The data is omitted as a precaution to prevent sensitive data from being stored in logs. If you are having problems with queries and need to see the values sent to the database, then you can use the EnableSensitiveDataLogging method when configuring the database context, as shown in Listing 17-30.

Listing 17-30. Enabling Sensitive Data Logging in the Startup.cs File in the Platform Folder

```
...
services.AddDbContext<CalculationContext>(opts => {
    opts.UseSqlServer(Configuration["ConnectionStrings:CalcConnection"]);
    opts.EnableSensitiveDataLogging(true);
});
...
```

Restart ASP.NET Core MVC and request the http://localhost:5000/sum/100 URL again. When the request is handled, Entity Framework Core will include parameter values in the logging message it creates to show the SQL query, like this:

```
...
Executing DbCommand [Parameters=[@__count_0='100'], CommandType='Text',
    CommandTimeout='30']
SELECT TOP(1) [c].[Id], [c].[Count], [c].[Result]
FROM [Calculations] AS [c]
WHERE ([c].[Count] = @__count_0) AND @__count_0 IS NOT NULL
...
```

This is a feature that should be used with caution because logs are often accessible by people who would not usually have access to the sensitive data that applications handle, such as credit card numbers and account details.

Summary

In this chapter, I demonstrated the ASP.NET Core platform features that are useful for working with data. I showed you how to cache individual data values, both locally and in a shared database. I also showed you how to cache responses and how to read and write data in a database using Entity Framework Core. In Part 3 of this book, I explain how to build on the ASP.NET Core platform to create web applications.