Lesson 07

API in ASP.NET Core MVC apps

ASP.NET Core

tongsreng TAL

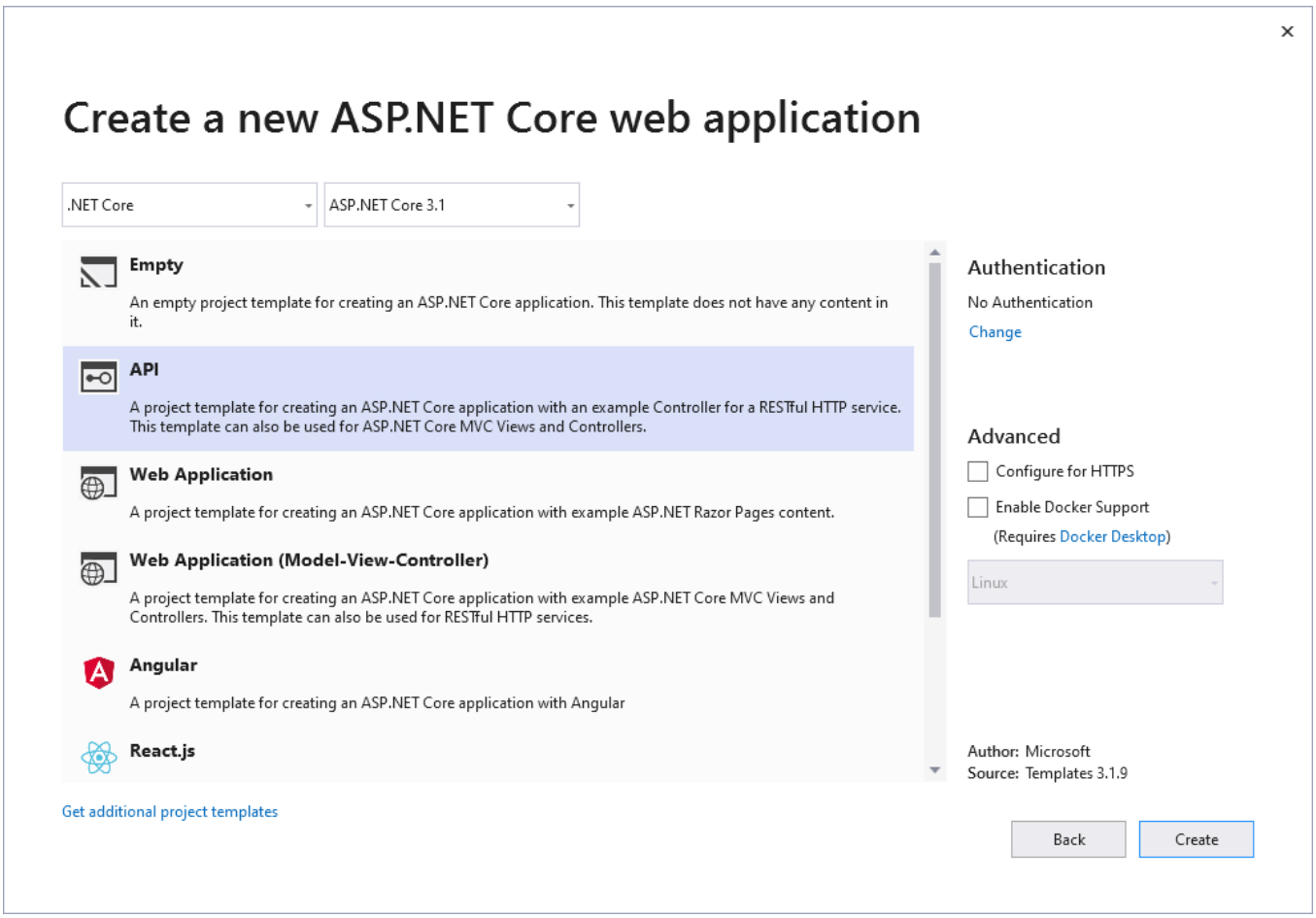
ASP.NET Core

Contents

In this lesson we will discuss about using API in View-Model-Controller Architecture in ASP.NET Core Web Application.

1 Prepare for this lesson

Create ASP.NET Core API project named “WebApp” with no HTTPS.



Open the WebApp.sln file in the WebApp folder. Select Project ➤ Platform Properties, navigate to the Debug page, and change the App URL field to http://localhost:5000, as shown in Figure 18-1. This changes the port that will be used to receive HTTP requests. Select File ➤ Save All to save the configuration changes.

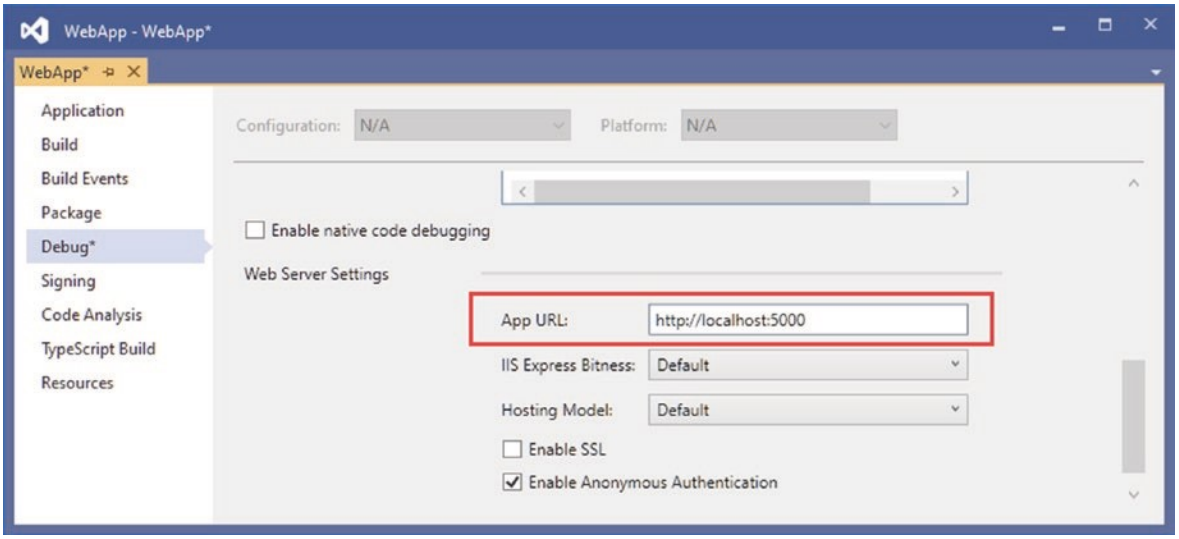


Figure 18-1. *Changing the HTTP port*

If you are using Visual Studio Code, open the WebApp folder. Click the Yes button when prompted to add the assets required for building and debugging the project, as shown in Figure 18-2.

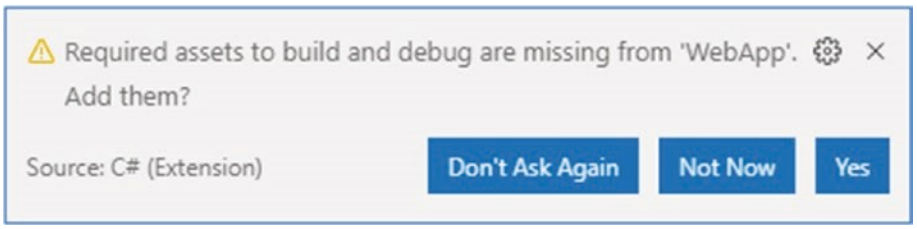


Figure 18-2. *Adding project assets*

1.1 Adding a Data Model

A data model helps demonstrate the different ways that web applications can be built using ASP.NET Core, showing how complex responses can be composed and how data can be submitted by the user. In the sections that follow, I create a simple data model and use it to create the database schema that will be used to store the application's data.

1.1.1 Adding NuGet Packages to the Project

The data model will use Entity Framework Core to store and query data in a SQL Server LocalDB database. To add the NuGet packages for Entity Framework Core, use a PowerShell command prompt to run the commands shown in Listing 18-2 in the WebApp project folder.

Listing 18-2. Adding Packages to the Project

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 3.1.9

dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 3.1.9
```

If you are using Visual Studio, you can add the packages by selecting Project ➤ Manage NuGet Packages. Take care to choose the correct version of the packages to add to the project.

1.1.2 Creating the Data Model

The data model for this part of the book will consist of three related classes: Product, Supplier, and Category. Create a new folder named Models and add to it a class file named Category.cs, with the contents shown in Listing 18-4.

Listing 18-4. The Contents of the Category.cs File in the Models Folder

```
using System.Collections.Generic;
namespace WebApp.Models {
    public class Category {
        public long CategoryId { get; set; }
        public string Name { get; set; }
        public IEnumerable<Product> Products { get; set; }
    }
}
```

Add a class called Supplier.cs to the Models folder and use it to define the class shown in Listing 18-5.

Listing 18-5. The Contents of the Supplier.cs File in the Models Folder

```
using System.Collections.Generic;
namespace WebApp.Models {
    public class Supplier {
        public long SupplierId { get; set; }
        public string Name { get; set; }
        public string City { get; set; }
        public IEnumerable<Product> Products { get; set; }
    }
}
```

Next, add a class named Product.cs to the Models folder and use it to define the class shown in Listing 18-6.

Listing 18-6. The Contents of the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
namespace WebApp.Models {
    public class Product {
        public long ProductId { get; set; }
        public string Name { get; set; }
```

```
        [Column(TypeName = "decimal(8, 2)")]
        public decimal Price { get; set; }
        public long CategoryId { get; set; }
        public Category Category { get; set; }
        public long SupplierId { get; set; }
        public Supplier Supplier { get; set; }
    }
}
```

Each of the three data model classes defines a key property whose value will be allocated by the database when new objects are stored. There are also navigation properties that will be used to query for related data so that it will be possible to query for all the products in a specific category, for example.

The Price property has been decorated with the Column attribute, which specifies the precision of the values that will be stored in the database. There isn't a one-to-one mapping between C# and SQL numeric types, and the Column attribute tells Entity Framework Core which SQL type should be used in the database to store Price values. In this case, the decimal(8, 2) type will allow a total of eight digits, including two following the decimal point.

To create the Entity Framework Core context class that will provide access to the database, add a file called DataContext.cs to the Models folder and add the code shown in Listing 18-7.

Listing 18-7. The Contents of the DataContext.cs File in the Models Folder

```
using Microsoft.EntityFrameworkCore;
namespace WebApp.Models {
    public class DataContext: DbContext {
        public DataContext(DbContextOptions<DataContext> opts)
            : base(opts) { }
        public DbSet<Product> Products { get; set; }
        public DbSet<Category> Categories { get; set; }
        public DbSet<Supplier> Suppliers { get; set; }
    }
}
```

The context class defines properties that will be used to query the database for Product, Category, and Supplier data.

1.1.3 Preparing the Seed Data

Add a class called SeedData.cs to the Models folder and add the code shown in Listing 18-8 to define the seed data that will be used to populate the database.

Listing 18-8. The Contents of the SeedData.cs File in the Models Folder

```
using Microsoft.EntityFrameworkCore; using System.Linq;
namespace WebApp.Models {
    public static class SeedData {
        public static void SeedDatabase(DataContext context) {
            context.Database.Migrate();
            if (context.Products.Count() == 0 && context.Suppliers.Count() == 0
                    && context.Categories.Count() == 0) {
```

```
                Supplier s1 = new Supplier
                    { Name = "Splash Dudes", City = "San Jose"};
                Supplier s2 = new Supplier
                    { Name = "Soccer Town", City = "Chicago"};
                Supplier s3 = new Supplier
                    { Name = "Chess Co", City = "New York"};
                Category c1 = new Category { Name = "Watersports" };
                Category c2 = new Category { Name = "Soccer" };
                Category c3 = new Category { Name = "Chess" };
                context.Products.AddRange(
                    new Product {  Name = "Kayak", Price = 275,
                        Category = c1, Supplier = s1},
                    new Product {  Name = "Lifejacket", Price = 48.95m,
                        Category = c1, Supplier = s1},
                    new Product {  Name = "Soccer Ball", Price = 19.50m,
                        Category = c2, Supplier = s2},
                    new Product {  Name = "Corner Flags", Price = 34.95m,
                        Category = c2, Supplier = s2},
                    new Product {  Name = "Stadium", Price = 79500,
                        Category = c2, Supplier = s2},
                    new Product {  Name = "Thinking Cap", Price = 16,
                        Category = c3, Supplier = s3},
                    new Product {  Name = "Unsteady Chair", Price = 29.95m,
                        Category = c3, Supplier = s3},
                    new Product {  Name = "Human Chess Board", Price = 75,
                        Category = c3, Supplier = s3},
                    new Product {  Name = "Bling-Bling King", Price = 1200,
                        Category = c3, Supplier = s3}
                );
                context.SaveChanges();
            }
        }
    }
}
```

The static SeedDatabase method ensures that all pending migrations have been applied to the database. If the database is empty, it is seeded with categories, suppliers, and products. Entity Framework Core will take care of mapping the objects into the tables in the database, and the key properties will be assigned automatically when the data is stored.

1.1.4 Configuring Entity Framework Core Services and Middleware

Make the changes to the Startup class shown in Listing 18-9, which configure Entity Framework Core and set up the DataContext services that will be used throughout this part of the book to access the database.

Listing 18-9. Preparing Services and Middleware in the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
```

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using WebApp.Models;

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

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });

            services.AddControllers();
        }

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

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

To define the connection string that will be used for the application's data, add the configuration settings shown in Listing 18-10 in the appsettings.json file. The connection string should be entered on a single line.

Listing 18-10. Defining a Connection String in the appsettings.json File in the WebApp Folder

```
{
```

```
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.EntityFrameworkCore": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "ProductConnection":
"Server=(localdb)\\MSSQLLocalDB;Database=Products;MultipleActiveResultSets=True"
  }
}
```

In addition to the connection string, Listing 18-10 increases the logging detail for Entity Framework Core so that the SQL queries sent to the database are logged.

1.1.5 Creating and Applying the Migration

To create the migration that will set up the database schema, use a PowerShell command prompt to run the command shown in Listing 18-11 in the WebApp project folder.

Listing 18-11. Creating an Entity Framework Core Migration

```
dotnet ef migrations add Initial
```

Once the migration has been created, apply it to the database using the command shown in Listing 18-12. ***Listing 18-12.*** Applying the Migration to the Database

```
dotnet ef database update
```

The logging messages displayed by the application will show the SQL commands that are sent to the database.

■ **Note** if you need to reset the database, then run the dotnet ef database drop --force command and then the command in listing 18-12.

1.2 Adding the CSS Framework

Later chapters will demonstrate the different ways that HTML responses can be generated. Run the commands shown in Listing 18-13 to remove any existing version of the LibMan package and install the latest version. ***Listing 18-13.*** Installing the LibMan Tool Package

```
dotnet tool uninstall --global Microsoft.Web.LibraryManager.Cli

dotnet tool install --global Microsoft.Web.LibraryManager.Cli
```

To add the Bootstrap CSS framework so that the HTML responses can be styled, run the commands shown in Listing 18-14 in the WebApp project folder.

Listing 18-14. Installing the Bootstrap CSS Framework

```
libman init -p cdnjs

libman install twitter-bootstrap@4.5.3 -d wwwroot/lib/twitter-bootstrap
```

2 RESTful Web Services

Web services accept HTTP requests and generate responses that contain data. In this chapter, I explain how the features provided by the MVC Framework, which is an integral part of ASP.NET Core, can be used to build on the capabilities described in Part 2 to create web services.

The nature of web services means that some of the examples in this chapter are tested using command-line tools provided by PowerShell, and it is important to enter the commands exactly as shown. Chapter 20 introduces more sophisticated tools for working with web services, but the command-line approach is better suited to following examples in a book chapter, even if they can feel a little awkward as you type them in. Table 19-1 puts RESTful web services in context.

Table 19-1. *Putting RESTful Web Services in Context*

Question	Answer
What are they?	Web services provide access to an application's data, typically expressed in the JSON format.
Why are they useful?	Web services are most often used to provide rich client-side applications with data.
How are they used?	The combination of the URL and an HTTP method describes an operation that is handled by an action method defined by an ASP.NET Core controller.
Are there any pitfalls or limitations?	

2.1 Understanding RESTful Web Services

Web services respond to HTTP requests with data that can be consumed by clients, such as JavaScript applications. There are no hard-and-fast rules for how web services should work, but the most common approach is to adopt the Representational State Transfer (REST) pattern. There is no authoritative specification for REST, and there is no consensus about what constitutes a RESTful web service, but there are some common themes that are widely used for web services. The lack of a detailed specification leads to endless disagreement about what REST means and how RESTful web services should be created, all of which can be safely ignored if the web services you create work for your projects.

2.1.1 Understanding Request URLs and Methods

The core premise of REST—and the only aspect for which there is broad agreement—is that a web service defines an API through a combination of URLs and HTTP methods such as GET and POST, which are also known as the HTTP *verbs*. The method specifies the type of operation, while the URL specifies the data object or objects that the operation applies to.

As an example, here is a URL that might identify a Product object in the example application:

```
/api/products/1
```

This URL may identify the Product object that has a value of 1 for its ProductId property. The URL identifies the Product, but it is the HTTP method that specifies what should be done with it. Table 19-3 lists the HTTP methods that are commonly used in web services and the operations they conventionally represent.

Table 19-3. *HTTP Methods and Operations*

HTTP Method	Description
GET	This method is used to retrieve one or more data objects.
POST	This method is used to create a new object.
PUT	This method is used to update an existing object.
PATCH	This method is used to update part of an existing object.
DELETE	This method is used to delete an object.

2.1.2 Understanding JSON

Most RESTful web services format the response data using the JavaScript Object Notation (JSON) format. JSON has become popular because it is simple and easily consumed by JavaScript clients. JSON is described in detail at www.json.org, but you don't need to understand every aspect of JSON to create web services because ASP.NET Core provides all the features required to create JSON responses.

UNDERSTANDING THE ALTERNATIVES TO RESTFUL WEB SERVICES

reSt isn't the only way to design web services, and there are some popular alternatives. *GraphQL* is most closely associated with the react JavaScript framework, but it can be used more widely. unlike reSt web services, which provide specific queries through individual combinations of a url and an http method, graphQl provides access to all an application's data and lets clients query for just the data they require in the format they require. graphQl can be complex to set up—and can require more sophisticated clients—but the result is a more flexible web service that puts the developers of the client in control of the data they consume. graphQl isn't supported directly by aSp.net Core, but there are .net implementations available. See https:// graphql.org for more detail.

a new alternative is grpC, a full remote procedure call framework that focuses on speed and efficiency. at the time of writing, grpC cannot be used in web browsers, such as by the angular or react framework, because browsers don't provide the finegrained access that grpC requires to formulate its http requests.

2.2 Creating a Web Service Using a Custom Endpoint

As you learn about the facilities that ASP.NET Core provides for web services, it can be easy to forget they are built on the features described in Part 2. To create a simple web service, add a file named WebServiceEndpoint.cs to the WebApp folder and use it to define the class shown in Listing 19-3.

Listing 19-3. The Contents of the WebServiceEndpoint.cs File in the WebApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using System.Collections.Generic;
using System.Text.Json;
using WebApp.Models;
using Microsoft.Extensions.DependencyInjection;
namespace WebApp
{
    public static class WebServiceEndpoint
    {
        private static string BASEURL = "api/products";
        public static void MapWebService(this IEndpointRouteBuilder app)
        {
            app.MapGet($"{BASEURL}/{{id}}", async context => {
                long key = long.Parse(context.Request.RouteValues["id"] as string);
                DataContext data = context.RequestServices.GetService<DataContext>();
                Product p = data.Products.Find(key);
                if (p == null)
                {
                    context.Response.StatusCode = StatusCodes.Status404NotFound;
                }
                else
                {
                    context.Response.ContentType = "application/json";
                    await context.Response
                        .WriteAsync(JsonSerializer.Serialize<Product>(p));
                }
            });
            app.MapGet(BASEURL, async context => {
                DataContext data = context.RequestServices.GetService<DataContext>();
                context.Response.ContentType = "application/json";
                await context.Response.WriteAsync(JsonSerializer
                    .Serialize<IEnumerable<Product>>(data.Products));
            });
            app.MapPost(BASEURL, async context => {
                DataContext data = context.RequestServices.GetService<DataContext>();
                Product p = await
                    JsonSerializer.DeserializeAsync<Product>(context.Request.Body);
                await data.AddAsync(p);
                await data.SaveChangesAsync();
                context.Response.StatusCode = StatusCodes.Status200OK;
            });
        }
    }
}
```

The MapWebService extension method creates three routes that form a basic web service using only the features that have been described in earlier chapters. The routes match URLs that start with /api, which is the conventional URL prefix for web services. The endpoint for the first route receives a value from a segment variable that is used to locate a single Product object in the database. The endpoint for the second route retrieves all the Product objects in the database. The third endpoint handles POST requests and reads the request body to get a JSON representation of a new object to add to the database.

There are better ASP.NET Core features for creating web services, which you will see shortly, but the code in Listing 19-3 shows how the HTTP method and the URL can be combined to describe an operation. Listing 19-4 uses the MapWebService extension method to add the endpoints to the example application's routing configuration.

Listing 19-4. Adding Routes in the Startup.cs File in the WebApp Folder

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

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
                endpoints.MapWebService();
            });
            SeedData.SeedDatabase(context);
        }

...
```

To test the web service, restart ASP.NET Core and request http://localhost:5000/api/products/1. The request will be matched by the first route defined in Listing 19-4 and will produce the response shown on the left of Figure 19-2. Next, request http://localhost:5000/api/products, which will be matched by the second route and produce the response shown on the right of Figure 19-2.

■ **Note** the responses shown in the figure contain null values for the Supplier and Category properties because the linQ queries do not include related data. See Chapter 20 for details.

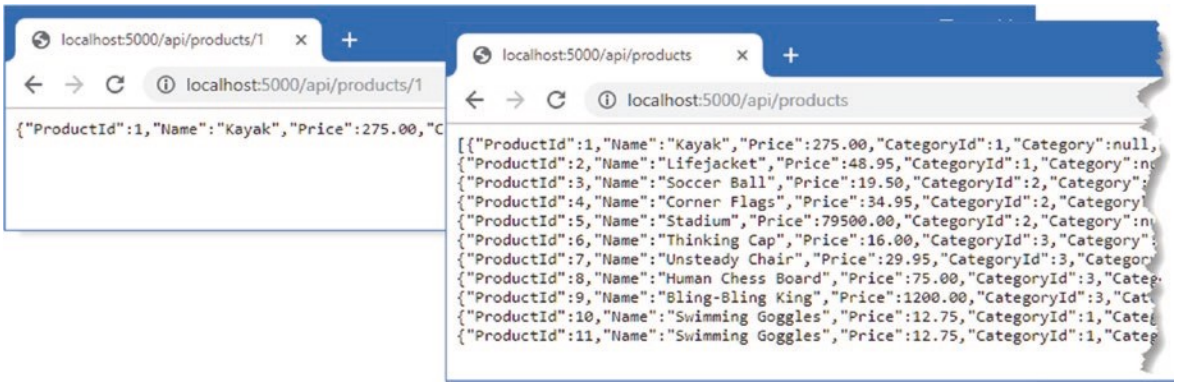


Figure 19-2. *Web service response*

Testing the third route requires a different approach because it isn't possible to send HTTP POST requests using the browser. Open a new PowerShell command prompt and run the command shown in Listing 19-5. It is important to enter the command exactly as shown because the Invoke-RestMethod command is fussy about the syntax of its arguments.

■ **Tip** You may receive an error when you use the Invoke-RestMethod or Invoke-WebRequest command to test the examples in this chapter if you have not performed the initial setup for Microsoft edge or internet explorer. the problem can be fixed by running ie and selecting the initial configurations you require.

Listing 19-5. Sending a POST Request

```
Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body (@{ Name="Swimming
Goggles"; Price=12.75; CategoryId=1; SupplierId=1} | ConvertTo-Json) -ContentType
"application/json"
```

The command sends an HTTP POST command that is matched by the third route defined in Listing 19-5. The body of the request is a JSON-formatted object that is parsed to create a Product, which is then stored in the database. The JSON object included in the request contains values for the Name, Price, CategoryId, and SupplierId properties. The unique key for the object, which

is associated with the ProductId property, is assigned by the database when the object is stored. Use the browser to request the http://localhost:5000/api/products URL again, and you will see that the JSON response contains the new object, as shown in Figure 19-3.

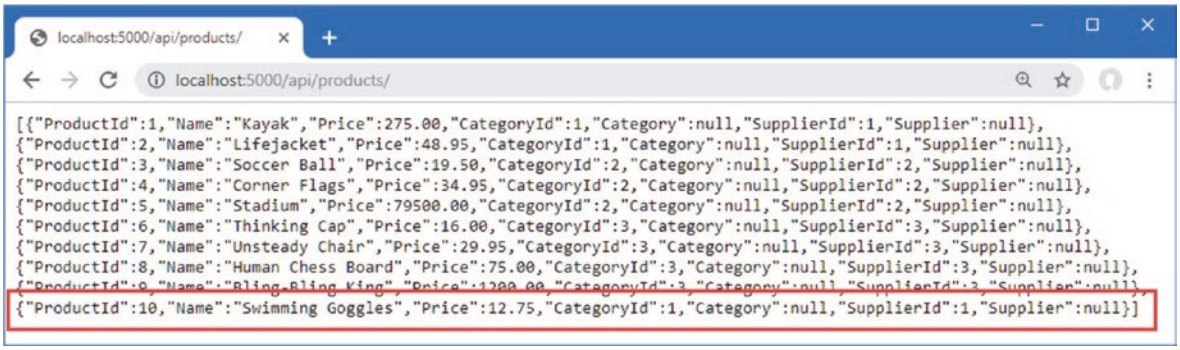


Figure 19-3. *Storing new data using the web service*

2.3 Creating a Web Service Using a Controller

The drawback of using endpoints to create a web service is that each endpoint has to duplicate a similar set of steps to produce a response: get the Entity Framework Core service so that it can query the database, set the Content-Type header for the response, serialize the objects into JSON, and so on. As a result, web services created with endpoints are difficult to understand and awkward to maintain.

A more elegant approach is to use a *controller*, which allows a web service to be defined in a single class. Controllers are part of the MVC Framework, which builds on the ASP.NET Core platform and takes care of handling data in the same way that endpoints take care of processing URLs.

THE RISE AND FALL OF THE MVC PATTERN IN ASP.NET CORE

the MvC framework is an implementation of the Model-view-Controller pattern, which describes one way to structure an application. the examples in this chapter use two of the three pillars of the pattern: a data model (the *M* in MvC) and controllers (the *C* in MvC). Chapter 21 provides the missing piece and explains how views can be used to create htMl responses using razor.

the MvC pattern was an important step in the evolution of aSp.net and allowed the platform to break away from the Web forms model that predated it. Web forms applications were easy to start but quickly became difficult to manage and hid details of http requests and responses from the developer. by contrast, the adherence to the MvC pattern provided a strong and scalable structure for applications written with the MvC framework and hid nothing from the developer. the MvC framework revitalized aSp.net and provided the foundation for what became aSp.net Core, which dropped support for Web forms and focused solely on using the MvC pattern.

as aSp.net Core evolved, other styles of web application have been embraced, and the MvC framework is only one of the ways that applications can be created. that doesn't undermine the utility of the MvC pattern, but it doesn't have the central role that it used to in aSp.net Core

development, and the features that used to be unique to the MvC framework can now be accessed through other approaches, such as razor pages and blazor.

a consequence of this evolution is that understanding the MvC pattern is no longer a prerequisite for effective aSp.net Core development. if you are interested in understanding the MvC pattern, then https://en.wikipedia.org/wiki/Model–view– controller is a good place to start. but for this book, understanding how the features provided by the MvC framework build on the aSp.net Core platform is all the context that is required.

2.3.1 Enabling the MVC Framework

The first step to creating a web service using a controller is to configure the MVC framework, which requires a service and an endpoint, as shown in Listing 19-6.

Listing 19-6. Enabling the MVC Framework in the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore; using WebApp.Models;
namespace WebApp {     public class Startup {
        public Startup(IConfiguration config) {
            Configuration = config;
        }         public IConfiguration Configuration { get; set; }
        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllers();
        }
        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<TestMiddleware>();
            app.UseEndpoints(endpoints => {
                //endpoints.MapWebService();
                endpoints.MapControllers();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

The AddControllers method defines the services that are required by the MVC framework, and the MapControllers method defines routes that will allow controllers to handle requests. You will see other methods used to configure the MVC framework used in later chapters, which provide access to different features, but the methods used in Listing 19-6 are the ones that configure the MVC framework for web services.

2.3.2 Creating a Controller

Controllers are classes whose methods, known as *actions*, can process HTTP requests. Controllers are discovered automatically when the application is started. The basic discovery process is simple: any public class whose name ends with Controller is a controller, and any public method a controller defines is an action. To demonstrate how simple a controller can be, create the WebApp/Controllers folder and add to it a file named ProductsController.cs with the code shown in Listing 19-7.

■ **Tip** Controllers are conventionally defined in the Controllers folder, but they can be defined anywhere in the project, and they will still be discovered.

Listing 19-7. The Contents of the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WebApp.Models;
namespace WebApp.Controllers {
    [Route("api/[controller]")]
    public class ProductsController: ControllerBase {
        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return new Product[] {
                new Product() { Name = "Product #1" },
                new Product() { Name = "Product #2" },
            };
        }
        [HttpGet("{id}")]
        public Product GetProduct() {
            return new Product() {
                ProductId = 1, Name = "Test Product"
            };
        }
    }
}
```

The ProductsController class meets the criteria that the MVC framework looks for in a controller. It defines public methods named GetProducts and GetProduct, which will be treated as actions.

2.3.2.1 Understanding the Base Class

Controllers are derived from the ControllerBase class, which provides access to features provided by the MVC Framework and the underlying ASP.NET Core platform. Table 19-4 describes the most useful properties provided by the ControllerBase class.

■ **Note** although controllers are typically derived from the ControllerBase or Controller classes (described in Chapter 21), this is just convention, and the MvC framework will accept any class whose name ends with Controller, that is derived from a class whose name ends with Controller, or that has been decorated with the Controller attribute. apply the NonController attribute to classes that meet these criteria but that should not receive http requests.

Table 19-4. *Useful ControllerBase Properties*

Name	Description
HttpContext	This property returns the HttpContext object for the current request.
ModelState	This property returns details of the data validation process, as demonstrated in the "Validating Data" section later in the chapter and described in detail in Chapter 29.
Request	This property returns the HttpRequest object for the current request.
Response	This property returns the HttpResponse object for the current response.
RouteData	This property returns the data extracted from the request URL by the routing middleware, as described in Chapter 13.
User	This property returns an object that describes the user associated with the current request, as described in Chapter 38.

A new instance of the controller class is created each time one of its actions is used to handle a request, which means the properties in Table 19-4 describe only the current request.

2.3.2.2 Understanding the Controller Attributes

The HTTP methods and URLs supported by the action methods are determined by the combination of attributes that are applied to the controller. The URL for the controller is specified by the Route attribute, which is applied to the class, like this:

```
...
[Route("api/[controller]")]
public class ProductsController: ControllerBase {
...
```

The [controller] part of the attribute argument is used to derive the URL from the name of the controller class. The

Controller part of the class name is dropped, which means that the attribute in Listing 19-7 sets the URL for the controller to /api/ products.

Each action is decorated with an attribute that specifies the HTTP method that it supports, like this:

```
...
[HttpGet] public Product[] GetProducts() {
...
```

The name given to action methods doesn't matter in controllers used for web services. There are other uses for controllers, described in Chapter 21, where the name does matter, but for web services, it is the HTTP method attributes and the route patterns that are important.

The HttpGet attribute tells the MVC framework that the GetProducts action method will handle HTTP GET requests. Table 19-5 describes the full set of attributes that can be applied to actions to specify HTTP methods.

Table 19-5. *The HTTP Method Attributes*

Name	Description
HttpGet	This attribute specifies that the action can be invoked only by HTTP requests that use the GET verb.
HttpPost	This attribute specifies that the action can be invoked only by HTTP requests that use the POST verb.
HttpDelete	This attribute specifies that the action can be invoked only by HTTP requests that use the DELETE verb.
HttpPut	This attribute specifies that the action can be invoked only by HTTP requests that use the PUT verb.
HttpPatch	This attribute specifies that the action can be invoked only by HTTP requests that use the PATCH verb.
HttpHead	This attribute specifies that the action can be invoked only by HTTP requests that use the HEAD verb.
AcceptVerbs	This attribute is used to specify multiple HTTP verbs.

The attributes applied to actions to specify HTTP methods can also be used to build on the controller's base URL.

```
...
[HttpGet("{id}")] public Product GetProduct() {
...
```

This attribute tells the MVC framework that the GetProduct action method handles GET requests for the URL pattern api/ products/{id}. During the discovery process, the attributes applied to the controller are used to build the set of URL patterns that the controller can handle, summarized in Table 19-6.

■ **Tip** When writing a controller, it is important to ensure that each combination of the http method and url pattern that the controller supports is mapped to only one action method. an exception will be thrown when a request can be handled by multiple actions because the MvC framework is unable to decide which to use.

Table 19-6. *The URL Patterns*

HTTP Method	URL Pattern	Action Method Name
GET	api/products	GetProducts
GET	api/products/{id}	GetProduct

You can see how the combination of attributes is equivalent to the MapGet methods I used for the same URL patterns when I used endpoints to create a web service earlier in the chapter.

GET AND POST: PICK THE RIGHT ONE

the rule of thumb is that get requests should be used for all read-only information retrieval, while pOSt requests should be used for any operation that changes the application state. in standards-compliance terms, get requests are for *safe* interactions (having no side effects besides information retrieval), and pOSt requests are for *unsafe* interactions (making a decision or changing something). these conventions are set by the World Wide Web Consortium (W3C), at www.w3.org/Protocols/ rfc2616/rfc2616-sec9.html.

get requests are *addressable*: all the information is contained in the url, so it's possible to bookmark and link to these addresses. Do not use get requests for operations that change state. Many web developers learned this the hard way in 2005 when google Web accelerator was released to the public. this application prefetched all the content linked from each page, which is legal within the http because get requests should be safe. unfortunately, many web developers had ignored the http conventions and placed simple links to "delete item" or "add to shopping cart" in their applications. Chaos ensued.

2.3.2.3 Understanding Action Method Results

One of the main benefits provided by controllers is that the MVC Framework takes care of setting the response headers and serializing the data objects that are sent to the client. You can see this in the results defined by the action methods, like this:

```
...
[HttpGet("{id}")]
public Product GetProduct() {
...
```

When I used an endpoint, I had to work directly with the JSON serializer to create a string that can be written to the response and set the Content-Type header to tell the client that the response contained JSON data. The action method returns a Product object, which is processed automatically.

To see how the results from the action methods are handled, restart ASP.NET Core and request http://localhost:5000/api/ products, which will produce the response shown on the left of Figure 19-4, which is produced by the GetProducts action method. Next, request http://localhost:5000/api/products/1, which will be handled by the GetProduct method and produce the result shown on the right side of Figure 19-4.

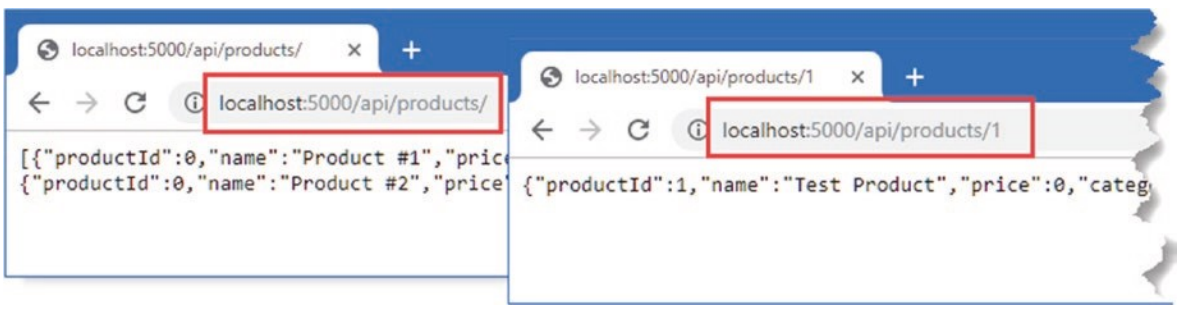


Figure 19-4. *Using a controller*

2.3.3 Using Dependency Injection in Controllers

A new instance of the controller class is created each time one of its actions is used to handle a request. The application's services are used to resolve any dependencies the controller declares through its constructor and any dependencies that the action method defines. This allows services that are required by all actions to be handled through the constructor while still allowing individual actions to declare their own dependencies, as shown in Listing 19-8.

Listing 19-8. Using Services in the ProductsController.cs File in the Controllers Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
```

```
using WebApp.Models;

namespace WebApp.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : ControllerBase
    {
        private DataContext context;
        public ProductsController(DataContext ctx)
        {
            context = ctx;
        }

        [HttpGet]
        public IEnumerable<Product> GetProducts()
        {
            return context.Products;
        }
        [HttpGet("{id}")]
        public Product GetProduct([FromServices]
                ILogger<ProductsController> logger)
        {
            logger.LogDebug("GetProduct Action Invoked");
            return context.Products.FirstOrDefault();
        }
    }
}
```

The constructor declares a dependency on the DataContext service, which provides access to the application's data. The services are resolved using the request scope, which means that a controller can request all services, without needing to understand their lifecycle.

THE ENTITY FRAMEWORK CORE CONTEXT SERVICE LIFECYCLE

a new entity framework Core context object is created for each controller. Some developers will try to refuse context objects as a perceived performance improvement, but this causes problems because data from one query can affect subsequent queries, as described in Chapter 20. behind the scenes, entity framework Core efficiently manages the connections to the database, and you should not try to store or reuse context objects outside of the controller for which they are created.

The GetProducts action method uses the DataContext to request all the Product objects in the database. The GetProduct method also uses the DataContext service, but it declares a dependency on ILogger<T>, which is the logging service described in Chapter 15. Dependencies that are declared by action methods must be decorated with the FromServices attribute, like this:

```
...
public Product GetProduct([FromServices] ILogger<ProductsController> logger) {
...
```

By default, the MVC Framework attempts to find values for action method parameters from the request URL, and the FromServices attribute overrides this behavior. To see the use of the services in the controller, restart ASP.NET Core and request http://localhost:5000/api/products/1, which will produce the response shown in Figure 19-5. You will also see the following logging message in the application's output:

```
...
dbug: WebApp.Controllers.ProductsController[0]
      GetProduct Action Invoked
...
```

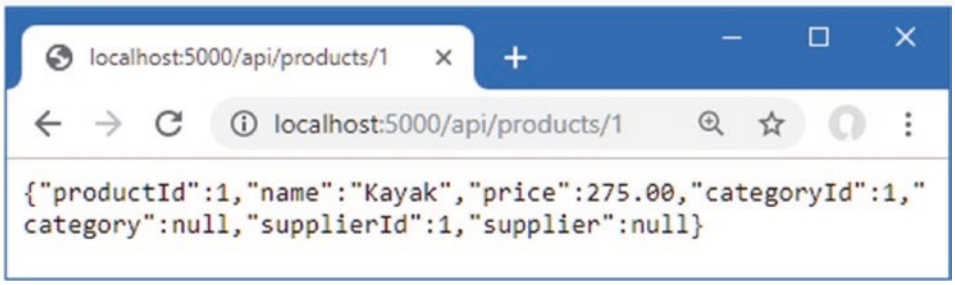


Figure 19-5. *Using services in a controller*

■ **Caution** One consequence of the controller lifecycle is that you can't rely on side effects caused by methods being called in a specific sequence. So, for example, i can't assign the ILogger<T> object received by the GetProduct method in listing 19-8 to a property that can be read by the GetProducts action in later requests. each controller object is used to handle one request, and only one action method will be invoked by the MvC framework for each object.

2.3.4 Using Model Binding to Access Route Data

In the previous section, I noted that the MVC Framework uses the request URL to find values for action method parameters, a process known as *model binding*. Model binding is described in detail in Chapter 28, but Listing 19-9 shows a simple example.

Listing 19-9. Using Model Binding in the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;
namespace WebApp.Controllers {
    [Route("api/[controller]")]
    public class ProductsController: ControllerBase {
        private DataContext context;
        public ProductsController(DataContext ctx) {
```

```
            context = ctx;
        }
        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }
        [HttpGet("{id}")]
        public Product GetProduct(long id,
                [FromServices] ILogger<ProductsController> logger) {
            logger.LogDebug("GetProduct Action Invoked");
            return context.Products.Find(id);
        }
    }
}
```

The listing adds a long parameter named id to the GetProduct method. When the action method is invoked, the MVC Framework injects the value with the same name from the routing data, automatically converting it to a long value, which is used by the action to query the database using the LINQ Find method. The result is that the action method responds to the URL, which you can see by restarting ASP.NET Core and requesting http://localhost:5000/api/products/5, which will produce the response shown in Figure 19-6.

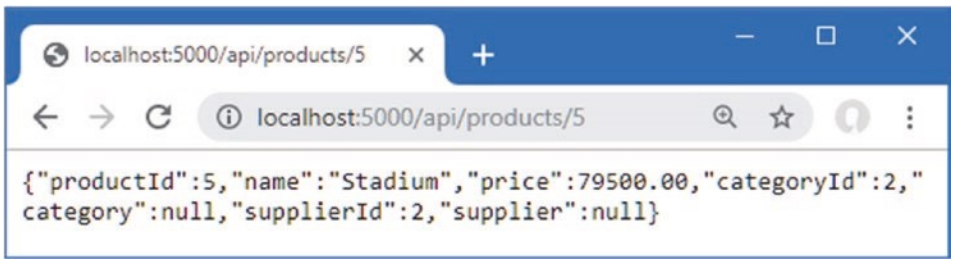


Figure 19-6. *Using model binding in an action*

2.3.5 Model Binding from the Request Body

The model binding feature can also be used on the data in the request body, which allows clients to send data that is easily received by an action method. Listing 19-10 adds a new action method that responds to POST requests and allows clients to provide a JSON representation of the Product object in the request body.

Listing 19-10. Adding an Action in the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;
namespace WebApp.Controllers {
    [Route("api/[controller]")]
    public class ProductsController: ControllerBase {
        private DataContext context;
        public ProductsController(DataContext ctx) {
            context = ctx;
        }
```

```
        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }
        [HttpGet("{id}")]
        public Product GetProduct(long id,
                [FromServices] ILogger<ProductsController> logger) {
            logger.LogDebug("GetProduct Action Invoked");
            return context.Products.Find(id);
        }
        [HttpPost]
        public void SaveProduct([FromBody]Product product) {
            context.Products.Add(product);
            context.SaveChanges();
        }
    }
}
```

The new action relies on two attributes. The HttpPost attribute is applied to the action method and tells the MVC Framework that the action can process POST requests. The FromBody attribute is applied to the action's parameter, and it specifies that the value for this parameter should be obtained by parsing the request body. When the action method is invoked, the MVC Framework will create a new Product object and populate its properties with the values in the request body. The model binding process can be complex and is usually combined with data validation, as described in Chapter 29, but for a simple demonstration, restart ASP.NET Core, open a new PowerShell command prompt, and run the command shown in Listing 19-11.

Listing 19-11. Sending a POST Request to the Example Application

```
Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body (@{ Name="Soccer
Boots"; Price=89.99; CategoryId=2; SupplierId=2} | ConvertTo-Json) -ContentType
"application/json"
```

Once the command has executed, use a web browser to request http://localhost:5000/api/products, and you will see the new object that has been stored in the database, as shown in Figure 19-7.

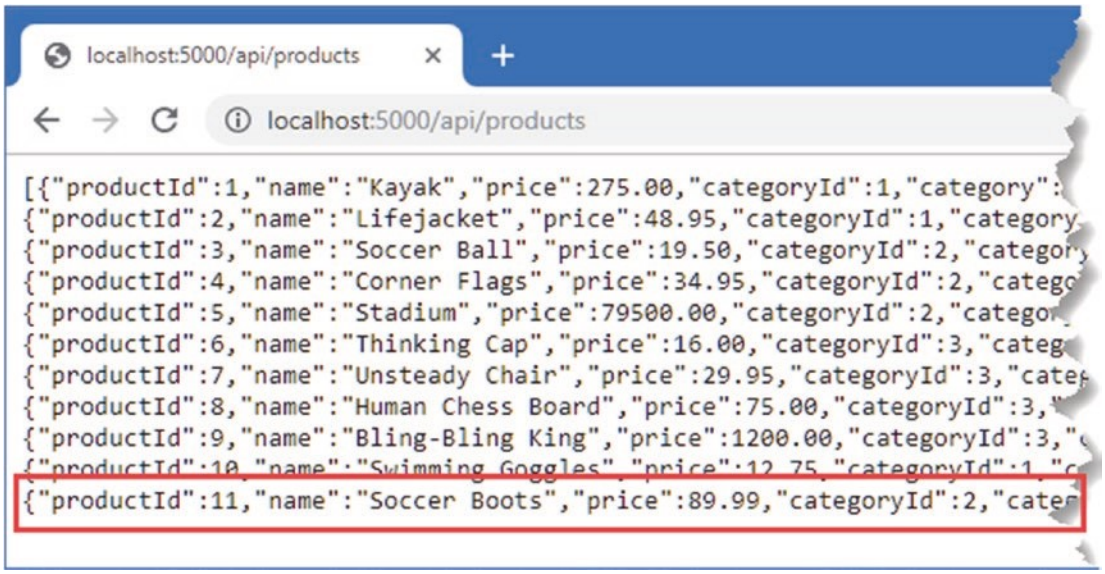


Figure 19-7. *Storing new data using a controller*

2.3.6 Adding Additional Actions

Now that the basic features are in place, I can add actions that allow clients to replace and delete Product objects using the HTTP PUT and DELETE methods, as shown in Listing 19-12.

Listing 19-12. Adding Actions in the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;
namespace WebApp.Controllers {
    [Route("api/[controller]")]
    public class ProductsController: ControllerBase {
        private DataContext context;
        public ProductsController(DataContext ctx) {
            context = ctx;
        }
        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }
        [HttpGet("{id}")]
        public Product GetProduct(long id,
                [FromServices] ILogger<ProductsController> logger) {
            logger.LogDebug("GetProduct Action Invoked");
            return context.Products.Find(id);
        }
        [HttpPost]
        public void SaveProduct([FromBody]Product product) {
            context.Products.Add(product);
            context.SaveChanges();
        }
        [HttpPut]
```

```
        public void UpdateProduct([FromBody]Product product) {
            context.Products.Update(product);
            context.SaveChanges();
        }
        [HttpDelete("{id}")]
        public void DeleteProduct(long id) {
            context.Products.Remove(new Product() { ProductId = id });
            context.SaveChanges();
        }
    }
}
```

The UpdateProduct action is similar to the SaveProduct action and uses model binding to receive a Product object from the request body. The DeleteProduct action receives a primary key value from the URL and uses it to create a Product that has a value only for the ProductId property, which is required because Entity Framework Core works only with objects, but web service clients typically expect to be able to delete objects using just a key value.

Restart ASP.NET Core and then use a different PowerShell command prompt to run the command shown in Listing 19-13, which tests the UpdateProduct action.

Listing 19-13. Updating an Object

```
Invoke-RestMethod http://localhost:5000/api/products -Method PUT -Body (@{ ProductId=1; Name="Green Kayak"; Price=275; CategoryId=1; SupplierId=1} | ConvertTo-Json) -ContentType "application/json"
```

The command sends an HTTP PUT request whose body contains a replacement object. The action method receives the object through the model binding feature and updates the database. Next, run the command shown in Listing 19-14 to test the DeleteProduct action.

Listing 19-14. Deleting an Object

```
Invoke-RestMethod http://localhost:5000/api/products/2 -Method DELETE
```

This command sends an HTTP DELETE request, which will delete the object whose ProductId property is 2. To see the effect of the changes, use the browser to request http://localhost:5000/api/products, which will send a GET request that is handled by the GetProducts action and produce the response shown in Figure 19-8.

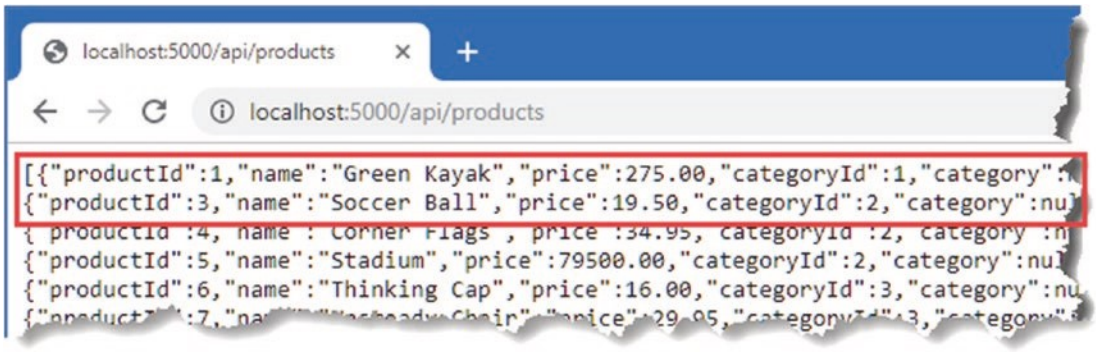


Figure 19-8. *Updating and deleting objects*

2.4 Improving the Web Service

The controller in Listing 19-14 re-creates all the functionality provided by the separate endpoints, but there are still improvements that can be made, as described in the following sections.

SUPPORTING CROSS-ORIGIN REQUESTS

if you are supporting third-party JavaScript clients, you may need to enable support for cross-origin requests (COrS). browsers protect users by only allowing JavaScript code to make http requests within the same origin, which means to urls that have the same scheme, host, and port as the url used to load the JavaScript code. COrS loosens this restriction by performing an initial http request to check that the server will allow requests originating from a specific url, helping prevent malicious code using your service without the user's consent.

aSp.net Core provides a built-in service that handles COrS, which is enabled by adding the following statement to the ConfigureServices method in the Startup class:

```
...
Services.AddCors();
...
```

the options pattern is used to configure COrS with the CorsOptions class defined in the Microsoft.AspNetCore. Cors.Infrastructure namespace. See https://docs.microsoft.com/en-gb/aspnet/core/security/ cors?view=aspnetcore-3.1 for details.

2.4.1 Using Asynchronous Actions

The ASP.NET Core platform processes each request by assigning a thread from a pool. The number of requests that can be processed concurrently is limited to the size of the pool, and a thread can't be used to process any other request while it is waiting for an action to produce a result.

Actions that depend on external resources can cause a request thread to wait for an extended period. A database server, for example, may have its own concurrency limits and may queue up

queries until they can be executed. The ASP.NET Core request thread is unavailable to process any other requests until the database produces a result for the action, which then produces a response that can be sent to the HTTP client.

This problem can be addressed by defining asynchronous actions, which allow ASP.NET Core threads to process other requests when they would otherwise be blocked, increasing the number of HTTP requests that the application can process simultaneously. Listing 19-15 revises the controller to use asynchronous actions.

■ **Note** asynchronous actions don't produce responses any quicker, and the benefit is only to increase the number of requests that can be processed concurrently.

Listing 19-15. Asynchronous Actions in the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;
using System.Threading.Tasks;
namespace WebApp.Controllers {
    [Route("api/[controller]")]
    public class ProductsController: ControllerBase {
        private DataContext context;
        public ProductsController(DataContext ctx) {
            context = ctx;
        }
        [HttpGet]
        public IAsyncEnumerable<Product> GetProducts() {
            return context.Products;
        }
        [HttpGet("{id}")]
        public async Task<Product> GetProduct(long id) {
            return await context.Products.FindAsync(id);
        }
        [HttpPost]
        public async Task SaveProduct([FromBody]Product product) {
            await context.Products.AddAsync(product);
            await context.SaveChangesAsync();
        }
        [HttpPut]
        public async Task UpdateProduct([FromBody]Product product) {
            context.Update(product);
            await context.SaveChangesAsync();
        }
        [HttpDelete("{id}")]
        public async Task DeleteProduct(long id) {
            context.Products.Remove(new Product() { ProductId = id });
            await context.SaveChangesAsync();
        }
    }
}
```

Entity Framework Core provides asynchronous versions of some methods, such as FindAsync, AdddAsync, and

SaveChangesAsync, and I have used these with the await keyword. Not all operations can be performed asynchronously, which is why the Update and Remove methods are unchanged.

For some operations—including LINQ queries to the database—the IAsyncEnumerable<T> interface can be used, which denotes a sequence of objects that should be enumerated asynchronously and prevents the ASP.NET Core request thread from waiting for each object to be produced by the database, as explained in Chapter 5.

There is no change to the responses produced by the controller, but the threads that ASP.NET Core assigns to process each request are not necessarily blocked by the action methods.

2.4.2 Preventing Over-Binding

Some of the action methods use the model binding feature to get data from the response body so that it can be used to perform database operations. There is a problem with the SaveProduct action, which can be seen by using a PowerShell prompt to run the command shown in Listing 19-16.

Listing 19-16. Saving a Product

```
Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body (@{ ProductId=100;
Name="Swim Buoy"; Price=19.99; CategoryId=1; SupplierId=1} | ConvertTo-Json) -ContentType
"application/json"
```

Unlike the command that was used to test the POST method in Listing 19-11, this command includes a value for the ProductId property. When Entity Framework Core sends the data to the database, the following exception is thrown:

```
...
Microsoft.Data.SqlClient.SqlException (0x80131904): Cannot insert explicit value for identity
column in table 'Products' when IDENTITY_INSERT is set to OFF.
...
```

By default, Entity Framework Core configures the database to assign primary key values when new objects are stored. This means the application doesn't have to worry about keeping track of which key values have already been assigned and allows multiple applications to share the same database without the need to coordinate key allocation. The Product data model class needs a ProductId property, but the model binding process doesn't understand the significance of the property and adds any values that the client provides to the objects it creates, which causes the exception in the SaveProduct action method.

This is known as *over-binding*, and it can cause serious problems when a client provides values that the developer wasn't expecting. At best, the application will behave unexpectedly, but this technique has been used to subvert application security and grant users more access than they should have.

The safest way to prevent over-binding is to create separate data model classes that are used only for receiving data through the model binding process. Add a class file named ProductBindingTarget.cs to the WebApp/Models folder and use it to define the class shown in Listing 19-17.

Listing 19-17. The Contents of the ProductBindingTarget.cs File in the WebApp/Models Folder

```
namespace WebApp.Models {
    public class ProductBindingTarget {
        public string Name { get; set; }
        public decimal Price { get; set; }
        public long CategoryId { get; set; }
        public long SupplierId { get; set; }
        public Product ToProduct() => new Product() {
            Name = this.Name, Price = this.Price,
            CategoryId = this.CategoryId, SupplierId = this.SupplierId
        };
    }
}
```

The ProductBindingTarget class defines only the properties that the application wants to receive from the client when storing a new object. The ToProduct method creates a Product that can be used with the rest of the application, ensuring that the client can only provide properties for the Name, Price, CategoryId, and SupplierId properties. Listing 19-18 uses the binding target class in the SaveProduct action to prevent over-binding.

Listing 19-18. Using a Binding Target in the ProductsController.cs File in the Controllers Folder

```
...
[HttpPost]
public async Task SaveProduct([FromBody]ProductBindingTarget target) {
    await context.Products.AddAsync(target.ToProduct());
    await context.SaveChangesAsync(); }
...
```

Restart ASP.NET Core and repeat the command from Listing 19-16, and you will see the response shown in Figure 19-9. The client has included the ProductId value, but it is ignored by the model binding process, which discards values for read-only properties. (You may see a different value for the ProductId property when you run this example depending on the changes you made to the database before running the command.)

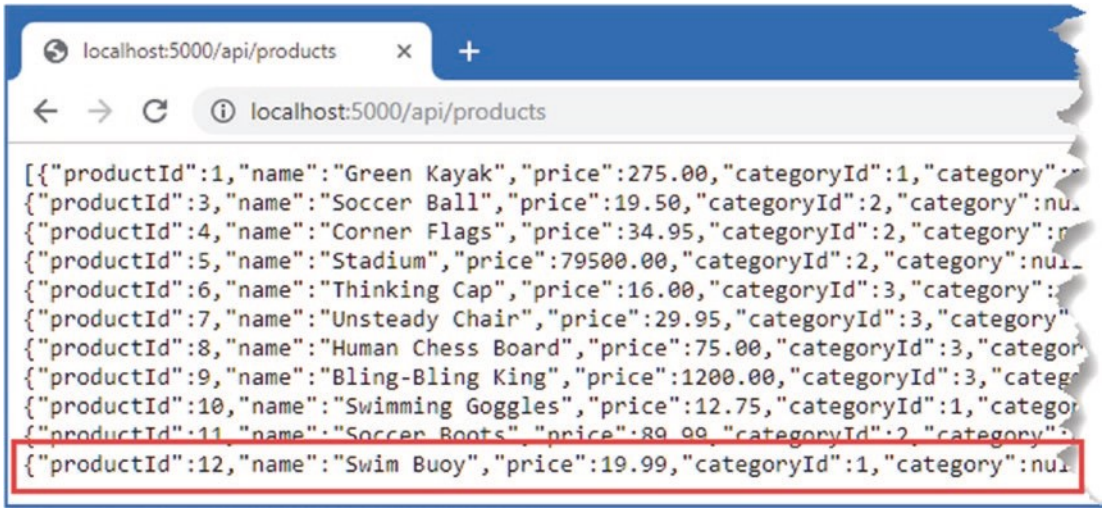


Figure 19-9. *Discarding unwanted data values*

2.4.3 Using Action Results

The MVC Framework sets the status code for responses automatically, but you won't always get the result you desire, in part because there are no firm rules for RESTful web services, and the assumptions that Microsoft makes may not match your expectations. To see an example, use a PowerShell command prompt to run the command shown in Listing 19-19, which sends a GET request to the web service.

Listing 19-19. Sending a GET Request

```
Invoke-WebRequest http://localhost:5000/api/products/1000 |Select-Object StatusCode
```

The Invoke-WebRequest command is similar to the Invoke-RestMethod command used in earlier examples but makes it easier to get the status code from the response. The URL requested in Listing 19-19 will be handled by the GetProduct action method, which will query the database for an object whose ProductId value is 1000, and the command produces the following output:

```
StatusCode
----------
       204
```

There is no matching object in the database, which means that the GetProduct action method returns null. When the MVC Framework receives null from an action method, it returns the 204 status code, which indicates a successful request that has produced no data. Not all web services behave this way, and a common alternative is to return a 404 response, indicating not found.

Similarly, the SaveProducts action will return a 200 response when it stores an object, but since the primary key isn't generated until the data is stored, the client doesn't know what key value was assigned.

■ **Note** there is no right or wrong when it comes to these kinds of web service implementation details, and you should pick the approaches that best suit your project and personal preferences. this section is an example of how to change the default behavior and not a direction to follow any specific style of web service.

Action methods can direct the MVC Framework to send a specific response by returning an object that implements the IActionResult interface, which is known as an *action result*. This allows the action method to specify the type of response that is required without having to produce it directly using the HttpResponse object.

The ControllerBase class provides a set of methods that are used to create action result objects, which can be returned from action methods. Table 19-7 describes the most useful action result methods.

Table 19-7. *Useful ControllerBase Action Result Methods*

Name	Description
Ok	The IActionResult returned by this method produces a 200 OK status code and sends an optional data object in the response body.
NoContent	The IActionResult returned by this method produces a 204 NO CONTENT status code.
BadRequest	The IActionResult returned by this method produces a 400 BAD REQUEST status code. The method accepts an optional model state object that describes the problem to the client, as demonstrated in the "Validating Data" section.

File	The IActionResult returned by this method produces a 200 OK response, sets the Content-Type header to the specified type, and sends the specified file to the client.
NotFound	The IActionResult returned by this method produces a 404 NOT FOUND status code.
RedirectRedirectPermanent	The IActionResult returned by these methods redirects the client to a specified URL.
RedirectToRoute RedirectToRoutePermanent	The IActionResult returned by these methods redirects the client to the specified URL that is created using the routing system, using convention routing, as described in the "Redirecting Using Route Values" sidebar.
LocalRedirectLocal RedirectPermanent	The IActionResult returned by these methods redirects the client to the specified URL that is local to the application.
RedirectToActionRedirect ToActionPermanent	The IActionResult returned by these methods redirects the client to an action method. The URL for the redirection is created using the URL routing system.
RedirectToPageRedirect ToPagePermanent	The IActionResult returned by these methods redirects the client to a Razor Page, described in Chapter 23.
StatusCode	The IActionResult returned by this method produces a response with a specific status code.

When an action method returns an object, it is equivalent to passing the object to the Ok method and returning the result. When an action returns null, it is equivalent to returning the result from the NoContent method. Listing 19-20 revises the behavior of the GetProduct and SaveProduct actions so they use the methods from Table 19-7 to override the default behavior for web service controllers.

Listing 19-20. Using Action Results in the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq; using System.Threading.Tasks;
namespace WebApp.Controllers {
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;
```

```
        public ProductsController(DataContext ctx) {
            context = ctx;
        }
        [HttpGet]
        public IAsyncEnumerable<Product> GetProducts() {
            return context.Products;         }
        [HttpGet("{id}")]
        public async Task<IActionResult> GetProduct(long id) {
            Product p = await context.Products.FindAsync(id);
            if (p == null) {
                return NotFound();
            }
            return Ok(p);
        }
        [HttpPost]
        public async Task<IActionResult>
                SaveProduct([FromBody]ProductBindingTarget target) {
            Product p = target.ToProduct();
            await context.Products.AddAsync(p);
            await context.SaveChangesAsync();
            return Ok(p);
        }
        [HttpPut]
        public async Task UpdateProduct([FromBody]Product product) {
            context.Update(product);
            await context.SaveChangesAsync();
        }
        [HttpDelete("{id}")]
        public async Task DeleteProduct(long id) {
            context.Products.Remove(new Product() { ProductId = id });
            await context.SaveChangesAsync();
        }
    }
}
```

Restart ASP.NET Core and repeat the command from Listing 19-19, and you will see an exception, which is how the InvokeWebRequest command responds to error status codes, such as the 404 Not Found returned by the GetProduct action method.

To see the effect of the change to the SaveProduct action method, use a PowerShell command prompt to run the command shown in Listing 19-21, which sends a POST request to the web service.

Listing 19-21. Sending a POST Request

```
Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body (@{Name="Boot Laces"; Price=19.99;

CategoryId=2; SupplierId=2} | ConvertTo-Json) -ContentType "application/json"
```

The command will produce the following output, showing the values that were parsed from the JSON data received from the web service:

```
productId  : 13
name       : Boot Laces
price      : 19.99
categoryId : 2
category   :
supplierId : 2
supplier   :
```

2.4.3.1 Performing Redirections

Many of the action result methods in Table 19-7 relate to redirections, which direct the client to another URL. The most basic way to perform a direction is to call the Redirect method, as shown in Listing 19-22.

■ **Tip** the LocalRedirect and LocalRedirectPermanent methods throw an exception if a controller tries to perform a redirection to any url that is not local. this is useful when you are redirecting to urls provided by users, where an *open redirection attack* is attempted to redirect another user to an untrusted site.

Listing 19-22. Redirecting in the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq; using System.Threading.Tasks;
namespace WebApp.Controllers {
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;
        public ProductsController(DataContext ctx) {
            context = ctx;
        }
        // ...other action methods omitted for brevity...
        [HttpGet("redirect")]
        public IActionResult Redirect() {
            return Redirect("/api/products/1");
        }
    }
}
```

The redirection URL is expressed as a string argument to the Redirect method, which produces a temporary redirection. Restart ASP.NET Core and use a PowerShell command prompt to run the command shown in Listing 19-23, which sends a GET request that will be handled by the new action method.

Listing 19-23. Testing Redirection

```
Invoke-RestMethod http://localhost:5000/api/products/redirect
```

The Invoke-RestMethod command will receive the redirection response from the web service and send a new request to the URL it is given, producing the following response:

```
productId  : 1
name       : GreenKayak
price      : 275.00
categoryId : 1
category   :
supplierId : 1
supplier   :
```

2.4.3.2 Redirecting to an Action Method

You can redirect to another action method using the RedirectToAction method (for temporary redirections) or the

RedirectToActionPermanent method (for permanent redirections). Listing 19-24 changes the Redirect action method so that the client will be redirected to another action method defined by the controller.

Listing 19-24. Redirecting to an Action the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;
using System.Threading.Tasks;
namespace WebApp.Controllers {
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;
        public ProductsController(DataContext ctx) {
            context = ctx;
        }
```

```
        // ...other action methods omitted for brevity...
        [HttpGet("redirect")]
        public IActionResult Redirect() {
            return RedirectToAction(nameof(GetProduct), new { Id = 1 });
        }
    }
}
```

The action method is specified as a string, although the nameof expression can be used to select an action method without the risk of a typo. Any additional values required to create the route are supplied using an anonymous object. Restart ASP.NET Core and use a PowerShell command prompt to repeat the command in Listing 19-23. The routing system will be used to create a URL that targets the specified action method, producing the following response:

```
productId  : 1
name       : Kayak
price      : 100.00
categoryId : 1
category   :
supplierId : 1
supplier   :
```

If you specify only an action method name, then the redirection will target the current controller. There is an overload of the RedirectToAction method that accepts action and controller names.

REDIRECTING USING ROUTE VALUES

the RedirectToRoute and RedirectToRoutePermanent methods redirect the client to a url that is created by providing the routing system with values for segment variables and allowing it to select a route to use. this can be useful for applications with complex routing configurations, and caution should be used because it is easy to create a redirection to the wrong url. here is an example of redirection with the RedirectToRoute method:

```
...
[HttpGet("redirect")] public IActionResult Redirect() {
    return RedirectToRoute(new {
        controller = "Products", action = "GetProduct", Id = 1
    });
}
...
```

the set of values in this redirection relies on convention routing to select the controller and action method. Convention routing is typically used with controllers that produce htMl responses, as described in Chapter 21.

2.4.4 Validating Data

When you accept data from clients, you must assume that a lot of the data will be invalid and be prepared to filter out values that the application can't use. The data validation features provided for MVC Framework controllers are described in detail in Chapter 29, but for this chapter, I am going to focus on only one problem: ensuring that the client provides values for the properties that are required to store data in the database. The first step in model binding is to apply attributes to the properties of the data model class, as shown in Listing 19-25.

Listing 19-25. Applying Attributes in the ProductBindingTarget.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations;
namespace WebApp.Models {
    public class ProductBindingTarget {
        [Required]
        public string Name { get; set; }
        [Range(1, 1000)]
        public decimal Price { get; set; }
        [Range(1, long.MaxValue)]
        public long CategoryId { get; set; }
        [Range(1, long.MaxValue)]
        public long SupplierId { get; set; }
        public Product ToProduct() => new Product() {
            Name = this.Name, Price = this.Price,
            CategoryId = this.CategoryId, SupplierId = this.SupplierId
        };
    }
}
```

The Required attribute denotes properties for which the client must provide a value and can be applied to properties that are assigned null when there is no value in the request. The Range attribute requires a value between upper and lower limits and is used for primitive types that will default to zero when there is no value in the request.

Listing 19-26 updates the SaveProduct action to perform validation before storing the object that is created by the model binding process, ensuring that only objects that contain values for all four properties are decorated with the validation attributes.

Listing 19-26. Applying Validation in the ProductsController.cs File in the Controllers Folder

```
...
[HttpPost]
public async Task<IActionResult> SaveProduct([FromBody]ProductBindingTarget target) {
    if (ModelState.IsValid) {
        Product p = target.ToProduct();
        await context.Products.AddAsync(p);
        await context.SaveChangesAsync();
        return Ok(p);
    }
    return BadRequest(ModelState); }
...
```

The ModelState property is inherited from the ControllerBase class, and the IsValid property returns true if the model binding process has produced data that meets the validation criteria. If the data received from the client is valid, then the action result from the Ok method is returned. If the data sent by the client fails the validation check, then the IsValid property will be false, and the action result from the BadRequest method is used instead. The BadRequest method accepts the object returned by the ModelState property, which is used to describe the validation errors to the client. (There is no standard way to describe validation errors, so the client may rely only on the 400 status code to determine that there is a problem.)

To test the validation, restart ASP.NET Core and use a new PowerShell command prompt to run the command shown in Listing 19-27.

Listing 19-27. Testing Validation

```
Invoke-WebRequest http://localhost:5000/api/products -Method POST -Body (@{Name="Boot Laces"} |

ConvertTo-Json) -ContentType "application/json"
```

The command will throw an exception that shows the web service has returned a 400 Bad Request response. Details of the validation errors are not shown because neither the Invoke-WebRequest command nor the Invoke-RestMethod command provides access to error response bodies. Although you can't see it, the body contains a JSON object that has properties for each data property that has failed validation, like this:

```
{
 "Price":["The field Price must be between 1 and 1000."],
 "CategoryId":["The field CategoryId must be between 1 and 9.223372036854776E+18."],
 "SupplierId":["The field SupplierId must be between 1 and 9.223372036854776E+18."]
}
```

You can see examples of working with validation messages in Chapter 29 where the validation feature is described in detail.

2.4.5 Applying the API Controller Attribute

The ApiController attribute can be applied to web service controller classes to change the behavior of the model binding and validation features. The use of the FromBody attribute to select data from the request body and explicitly checking the ModelState. IsValid property is not

required in controllers that have been decorated with the ApiController attribute. Getting data from the body and validating data are required so commonly in web services that they are applied automatically when the attribute is used, restoring the focus of the code in the controller's action to dealing with the application features, as shown in Listing 19-28.

Listing 19-28. Using ApiController in the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq; using System.Threading.Tasks;
namespace WebApp.Controllers {
    [ApiController]
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;
        public ProductsController(DataContext ctx) {
            context = ctx;
        }
        [HttpGet]
        public IAsyncEnumerable<Product> GetProducts() {
            return context.Products;
        }
        [HttpGet("{id}")]
        public async Task<IActionResult> GetProduct(long id) {
            Product p = await context.Products.FindAsync(id);
            if (p == null) {
                return NotFound();
            }
            return Ok(p);
        }
        [HttpPost]
        public async Task<IActionResult> SaveProduct(ProductBindingTarget target) {
            Product p = target.ToProduct();
            await context.Products.AddAsync(p);
            await context.SaveChangesAsync();
            return Ok(p);
        }
        [HttpPut]
        public async Task UpdateProduct(Product product) {
            context.Update(product);
            await context.SaveChangesAsync();
        }
        [HttpDelete("{id}")]
        public async Task DeleteProduct(long id) {
            context.Products.Remove(new Product() { ProductId = id });
            await context.SaveChangesAsync();
        }
        [HttpGet("redirect")]
        public IActionResult Redirect() {
            return RedirectToAction(nameof(GetProduct), new { Id = 1 });
        }
    }
```

```
}
```

Using the ApiController attribute is optional, but it helps produce concise web service controllers.

2.4.6 Omitting Null Properties

The final change I am going to make in this chapter is to remove the null values from the data returned by the web service. The data model classes contain navigation properties that are used by Entity Framework Core to associate related data in complex queries, as explained in Chapter 20. For the simple queries that are performed in this chapter, no values are assigned to these navigation properties, which means that the client receives properties for which values are never going to be available. To see the problem, use a PowerShell command prompt to run the command shown in Listing 19-29.

Listing 19-29. Sending a GET Request

```
Invoke-WebRequest http://localhost:5000/api/products/1 | Select-Object Content
```

The command sends a GET request and displays the body of the response from the web service, producing the following output:

```
Content
-------
{"productId":1,"name":"Green
Kayak","price":275.00,"categoryId":1,"category":null,"supplierId":1,"supplier":null}
```

The request was handled by the GetProduct action method, and the category and supplier values in the response will always be null because the action doesn't ask Entity Framework Core to populate these properties.

2.4.6.1 Projecting Selected Properties

The first approach is to return just the properties that the client requires. This gives you complete control over each response, but it can become difficult to manage and confusing for client developers if each action returns a different set of values. Listing 19-30 shows how the Product object obtained from the database can be projected so that the navigation properties are omitted.

Listing 19-30. Omitting Properties in the ProductsController.cs File in the Controllers Folder

```
...
[HttpGet("{id}")]
public async Task<IActionResult> GetProduct(long id) {
    Product p = await context.Products.FindAsync(id);
    if (p == null) {
        return NotFound();
    }
    return Ok(new {
        ProductId = p.ProductId, Name = p.Name,
        Price = p.Price, CategoryId = p.CategoryId,
        SupplierId = p.SupplierId
    });
} ...
```

The properties that the client requires are selected and added to an object that is passed to the Ok method. Restart ASP.NET Core and run the command from Listing 19-30, and you will receive a response that omits the navigation properties and their null values, like this:

Content

{"productId":1,"name":"Green Kayak","price":275.00,"categoryId":1,"supplierId":1}

2.4.6.2 Configuring the JSON Serializer

The JSON serializer can be configured to omit properties whose value is null when it serializes objects. The serializer is configured using the options pattern in the Startup class, as shown in Listing 19-31.

Listing 19-31. Configuring the JSON Serializer in the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq; using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting; using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models; using Microsoft.AspNetCore.Mvc;
namespace WebApp {
    public class Startup {
        public Startup(IConfiguration config) {
            Configuration = config;
        }
```

```
        public IConfiguration Configuration { get; set; }
        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllers();
            services.Configure<JsonOptions>(opts => {
                opts.JsonSerializerOptions.IgnoreNullValues = true;
            });
        }
        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<TestMiddleware>();
            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
                endpoints.MapControllers();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

The JSON serializer is configured using the JsonSerializerOptions property of the JsonOptions class, and null values are discarded when the IgnoreNullValues property is true.

This configuration change affects all JSON responses and should be used with caution, especially if any of your data model classes use null values to impart information to the client. To see the effect of the change, restart ASP.NET Core and use a browser to request http://localhost:5000/api/products, which will produce the response shown in Figure 19-10.

Creating reStful Web ServiCeS

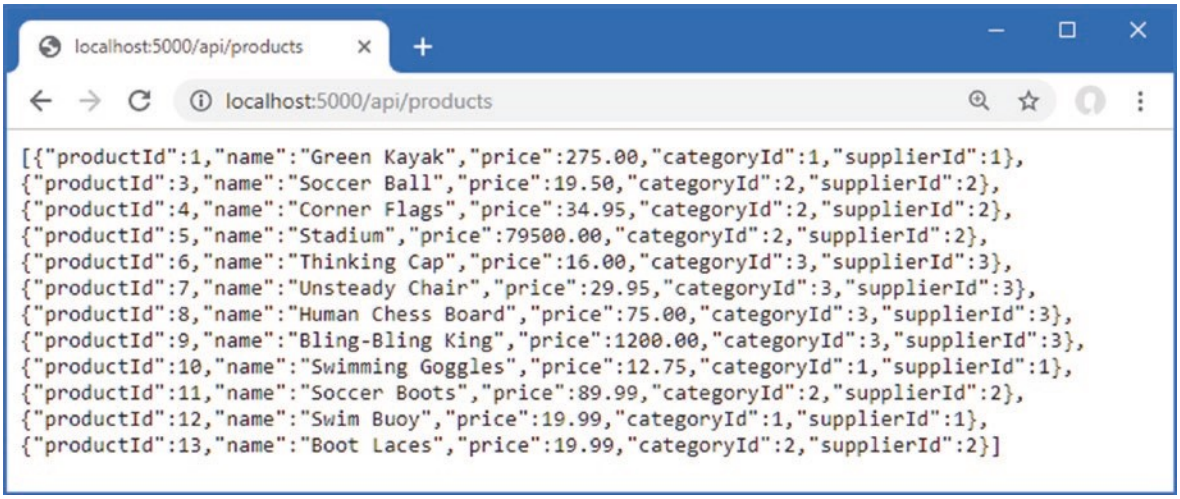


Figure 19-10. *Configuring the JSON serializer*

3 Advanced Web Service Features

In this section, I describe advanced features that can be used to create RESTful web services. I explain how to deal with related data in Entity Framework Core queries, how to add support for the HTTP PATCH method, how to use content negations, and how to use OpenAPI to describe your web services.

3.1 Preparing for This Section

This chapter uses the WebApp project created in Chapter 18 and modified in Chapter 19. To prepare for this chapter, add a file named SuppliersController.cs to the WebApp/Controllers folder with the content shown in Listing 20-1.

Listing 20-1. The Contents of the SuppliersController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Threading.Tasks;
namespace WebApp.Controllers {
    [ApiController]
    [Route("api/[controller]")]
    public class SuppliersController: ControllerBase {
        private DataContext context;
        public SuppliersController(DataContext ctx) {
            context = ctx;
        }
        [HttpGet("{id}")]
        public async Task<Supplier> GetSupplier(long id) {
            return await context.Suppliers.FindAsync(id);
        }
    }
}
```

The controller extends the ControllerBase class, declares a dependency on the DataContext service, and defines an action named GetSupplier that handles GET requests for the /api/[controller]/{id} URL pattern.

3.1.1 Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the WebApp.csproj file, and run the command shown in Listing 20-2 to drop the database.

Listing 20-2. Dropping the Database

```
dotnet ef database drop --force
```

3.1.2 Running the Example Application

Once the database has been dropped, select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 20-3. ***Listing 20-3.*** Running the Example Application

```
dotnet run
```

The database will be seeded as part of the application startup. Once ASP.NET Core is running, use a web browser to request http://localhost:5000/api/suppliers/1, which will produce the response shown in Figure 20-1

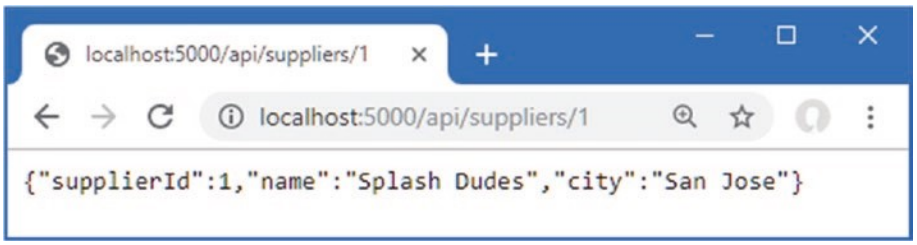


Figure 20-1. *Running the example application*

The response shows the Supplier object whose primary key matches the last segment of the request URL.

3.1.3 Dealing with Related Data

Entity Framework Core can populate by following relationships in the database using Include method as shown in Listing 20-4.

Listing 20-4. Requesting Related Data in the SuppliersController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models; using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
namespace WebApp.Controllers {
    [ApiController]
    [Route("api/[controller]")]
    public class SuppliersController: ControllerBase {
        private DataContext context;
        public SuppliersController(DataContext ctx) {
            context = ctx;         }
        [HttpGet("{id}")]
        public async Task<Supplier> GetSupplier(long id) {
           return await context.Suppliers
                .Include(s => s.Products)
                .FirstAsync(s => s.SupplierId == id);
```

```
        }
    }
}
```

The Include method tells Entity Framework Core to follow a relationship in the database and load the related data. In this case, the Include method selects the Products navigation property defined by the Supplier class, which causes Entity Framework Core to load the Product objects associated with the selected Supplier and assign them to the Products property.

Restart ASP.NET Core and use a browser to request http://localhost:5000/api/suppliers/1, which will target the GetSupplier action method. The request fails, and you will see the exception shown in Figure 20-2.

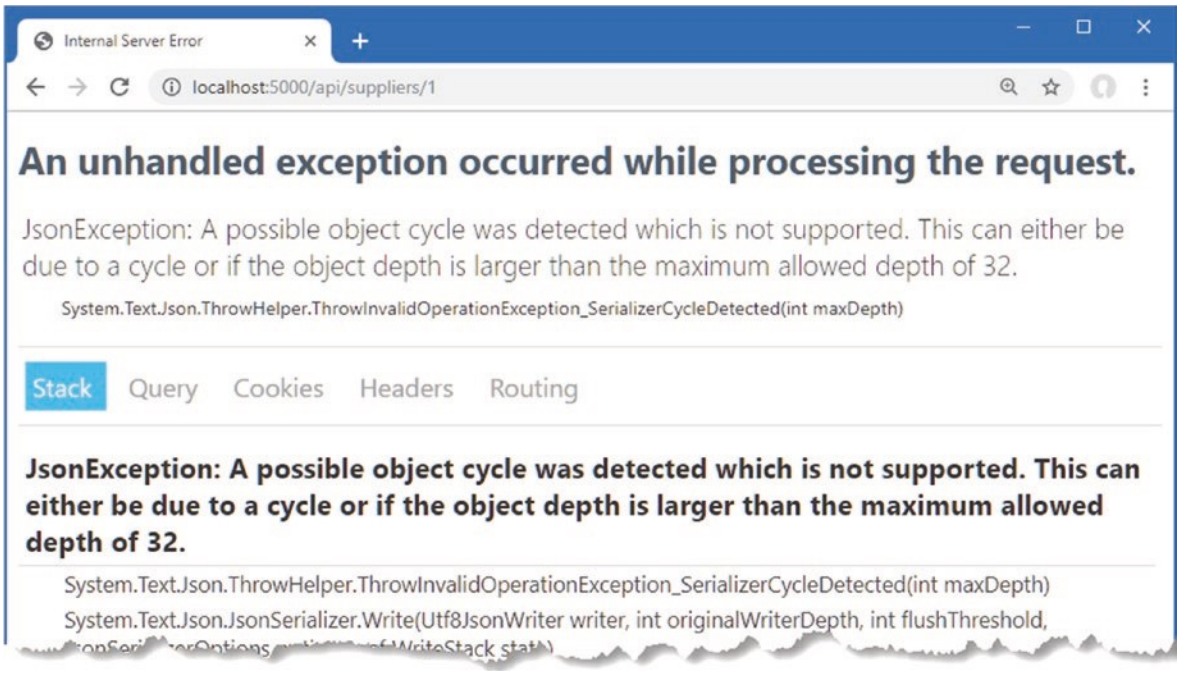


Figure 20-2. *An exception caused by querying for related data*

The JSON serializer has reported an "object cycle," which means there is a circular reference in the data that is being serialized for the response.

Looking at the code in Listing 20-4, you might struggle to see why using the Include method has created a circular reference. The problem is caused by an Entity Framework Core feature that attempts to minimize the amount of data read from the database but that causes problems in ASP.NET Core applications.

When Entity Framework Core creates objects, it populates navigation properties with objects that have already been created by the same database context. This can be a useful feature in some kinds of applications, such as desktop apps, where a database context object has a long life and

is used to make many requests over time. It isn't useful for ASP.NET Core applications, where a new context object is created for each HTTP request.

Entity Framework Core queries the database for the Product objects associated with the selected Supplier and assigns them to the Supplier.Products navigation property. The problem is that Entity Framework Core then looks at each Product object it has created and uses the query response to populate the Product.Supplier navigation property as well. For an ASP.NET Core application, this is an unhelpful step to take because it creates a circular reference between the navigation properties of the Supplier and Product objects, as shown in Figure 20-3.

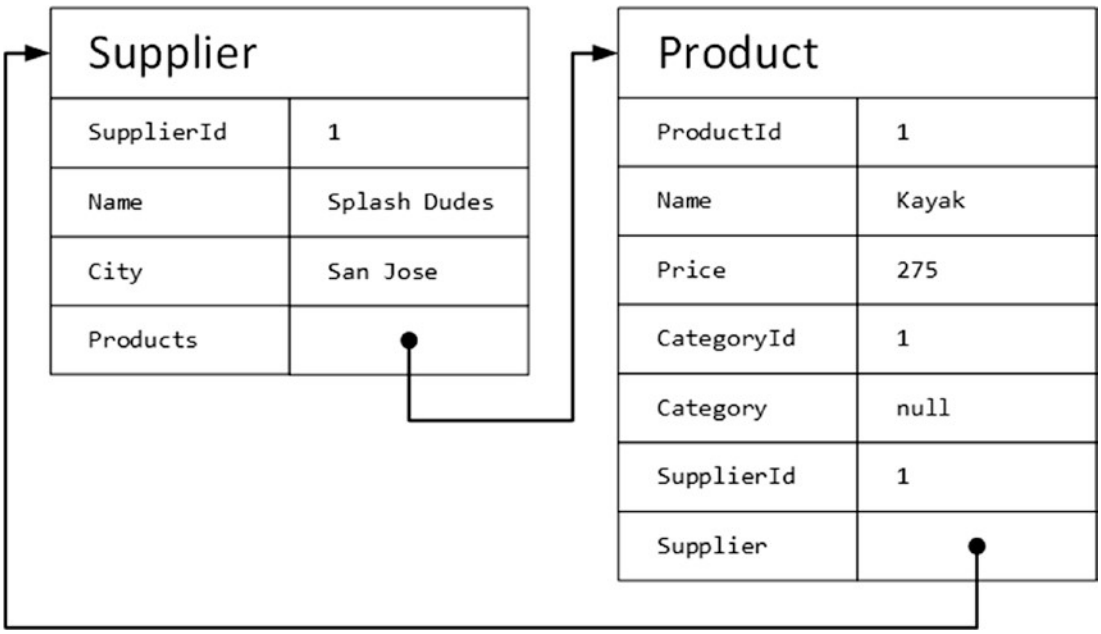


Figure 20-3. *Understanding how Entity Framework Core uses related data*

When the Supplier object is returned by the controller's action method, the JSON serializer works its way through the properties and follows the references to the Product objects, each of which has a reference back to the Supplier object, which it follows in a loop until the maximum depth is reached and the exception shown in Figure 20-2 is thrown.

3.1.4 Breaking Circular References in Related Data

There is no way to stop Entity Framework Core from creating circular references in the data it loads in the database. Preventing the exception means presenting the JSON serializer with data that doesn't contain circular references, which is most easily done by altering the objects after they have been created by Entity Framework Core and before they are serialized, as shown in Listing 20-5.

Listing 20-5. Breaking References in the SuppliersController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
namespace WebApp.Controllers {
    [ApiController]
    [Route("api/[controller]")]
    public class SuppliersController: ControllerBase {
        private DataContext context;
        public SuppliersController(DataContext ctx) {
            context = ctx;
        }
        [HttpGet("{id}")]
        public async Task<Supplier> GetSupplier(long id) {
            Supplier supplier = await context.Suppliers.Include(s => s.Products)
                .FirstAsync(s => s.SupplierId == id);
            foreach (Product p in supplier.Products) {
                p.Supplier = null;
            };
            return supplier;
        }
    }
}
```

The foreach loop sets the Supplier property of each Product object to null, which breaks the circular references. Restart ASP. NET Core and request http://localhost:5000/api/suppliers/1 to query for a supplier and its related products, which produces the response shown in Figure 20-4.

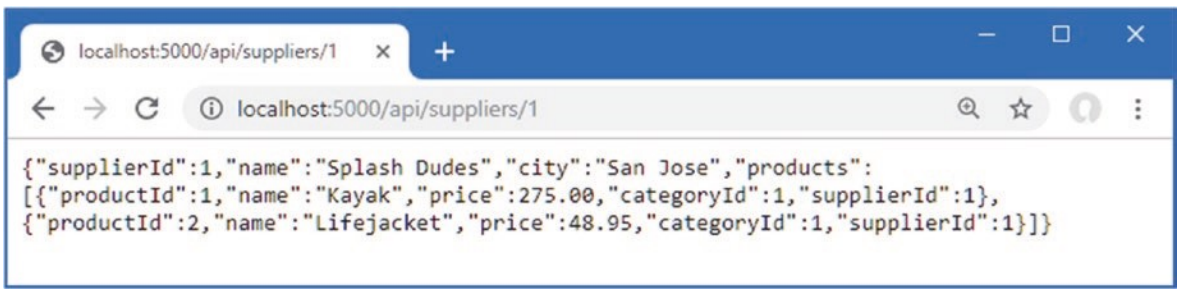


Figure 20-4. *Querying for related data*

3.1.5 Supporting the HTTP PATCH Method

For simple data types, edit operations can be handled by replacing the existing object using the PUT method, which is the approach I took in Chapter 19. Even if you only need to change a single property value in the Product class, for example, it isn't too much trouble to use a PUT method and include the values for all the other Product properties, too.

Not all data types are as easy to work with, either because they define too many properties or because the client has only received values for selected properties. The solution is to use a PATCH

request, which sends just the changes to the web service rather than a complete replacement object.

3.1.6 Understanding JSON Patch

ASP.NET Core has support for working with the JSON Patch standard, which allows changes to be specified in a uniform way. The JSON Patch standard allows for a complex set of changes to be described, but for this chapter, I am going to focus on just the ability to change the value of a property.

I am not going to go into the details of the JSON Patch standard, which you can read at https://tools.ietf.org/html/ rfc6902, but the client is going to send the web service JSON data like this in its HTTP PATCH requests:

```
[
 { "op": "replace", "path": "Name", "value": "Surf Co"},  { "op": "replace", "path": "City",
"value": “Los Angeles”},
]
```

A JSON Patch document is expressed as an array of operations. Each operation has an op property, which specifies the type of operation, and a path property, which specifies where the operation will be applied.

For the example application—and, in fact, for most applications—only the replace operation is required, which is used to change the value of a property. This JSON Patch document sets new values for the Name and City properties. The properties defined by the Supplier class not mentioned in the JSON Patch document will not be modified.

3.1.7 Installing and Configuring the JSON Patch Package

Support for JSON Patch isn’t installed when a project is created with the Empty template. To install the JSON Patch package, open a new PowerShell command prompt, navigate to the folder that contains the WebApp.csproj file, and run the command shown in Listing 20-6. If you are using Visual Studio, you can install the package by selecting Project ➤ Manage NuGet Packages.

Listing 20-6. Installing the JSON Patch Package

```
dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson --version 3.1.1
```

The Microsoft implementation of JSON Patch relies on the third-party Newtonsoft JSON.NET serializer that was used in ASP.NET Core 2.x but that has been replaced with a bespoke JSON serializer in ASP.NET Core 3.x. Add the statement shown in Listing 20-7 to the ConfigureServices method of the Startup class to enable the old serializer.

Listing 20-7. Enabling the JSON.NET Serializer in the Startup.cs File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllers().AddNewtonsoftJson();
    services.Configure<MvcNewtonsoftJsonOptions>(opts => {
        opts.SerializerSettings.NullValueHandling
            = Newtonsoft.Json.NullValueHandling.Ignore;
    });
    //services.Configure<JsonOptions>(opts => {
    //    opts.JsonSerializerOptions.IgnoreNullValues = true;
    //});
}
...
```

The AddNewtonsoftJson method enables the JSON.NET serializer, which replaces the standard ASP.NET Core serializer. The JSON.NET serializer has its own configuration class, MvcNewtonsoftJsonOptions, which is applied through the options pattern. Listing 20-7 sets the NullValueHandling value, which tells the serializer to discard properties with null values.

■ **Tip** See https://www.newtonsoft.com/json for details of the other configuration options available for the JSOn.net serializer.

3.1.8 Defining the Action Method

To add support for the PATCH method, add the action method shown in Listing 20-8 to the SuppliersController class. ***Listing 20-8.*** Adding an Action in the SuppliersController.cs File in the Controller Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.JsonPatch;
namespace WebApp.Controllers {
    [ApiController]
    [Route("api/[controller]")]
    public class SuppliersController : ControllerBase {
        private DataContext context;
        public SuppliersController(DataContext ctx) {
            context = ctx;
        }
        [HttpGet("{id}")]
```

```
        public async Task<Supplier> GetSupplier(long id) {
            Supplier supplier = await context.Suppliers.Include(s => s.Products)
                .FirstAsync(s => s.SupplierId == id);
            foreach (Product p in supplier.Products) {
                p.Supplier = null;
            };
            return supplier;
        }
        [HttpPatch("{id}")]
        public async Task<Supplier> PatchSupplier(long id,
                JsonPatchDocument<Supplier> patchDoc) {
            Supplier s = await context.Suppliers.FindAsync(id);
            if (s != null) {
                patchDoc.ApplyTo(s);
                await context.SaveChangesAsync();
            }
            return s;
        }
    }
}
```

The action method is decorated with the HttpPatch attribute, which denotes that it will handle HTTP requests. The model binding feature is used to process the JSON Patch document through a JsonPatchDocument<T> method parameter. The JsonPatchDocument<T> class defines an ApplyTo method, which applies each operation to an object. The action method in Listing 20-8 retrieves a Supplier object from the database, applies the JSON PATCH, and stores the modified object.

Restart ASP.NET Core and use a PowerShell command prompt to run the command shown in Listing 20-9, which sends an HTTP PATCH request with a JSON PATCH document that changes the value of the City property to Los Angeles.

Listing 20-9. Sending an HTTP PATCH Request

```
Invoke-RestMethod http://localhost:5000/api/suppliers/1 -Method PATCH -ContentType
"application/json"
-Body '[{"op":"replace","path":"City","value":"Los Angeles"}]'
```

The PatchSupplier action method returns the modified Supplier object as its result, which is serialized and sent to the client in the HTTP response. You can also see the effect of the change by using a web browser to request http://localhost:5000/ suppliers/1, which produces the response shown in Figure 20-5.

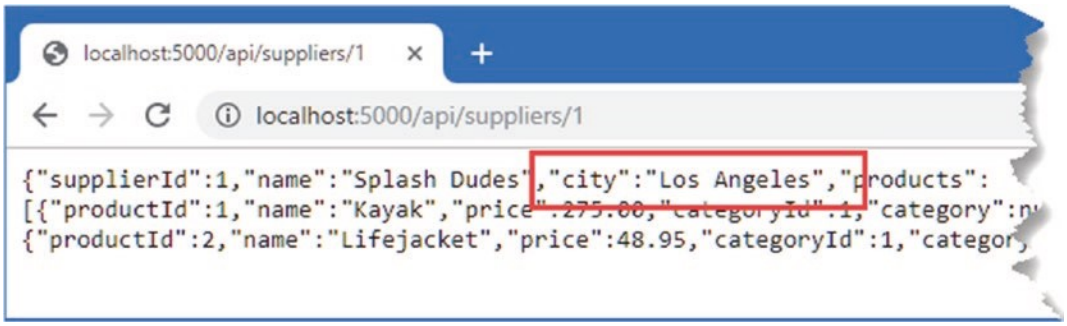


Figure 20-5. *Updating using a PATCH request*

3.2 Understanding Content Formatting

The web service examples so far have produced JSON results, but this is not the only data format that action methods can produce. The content format selected for an action result depends on four factors: the formats that the client will accept, the formats that the application can produce, the content policy specified by the action method, and the type returned by the action method. Figuring out how everything fits together can be daunting, but the good news is that the default policy works just fine for most applications, and you only need to understand what happens behind the scenes when you need to make a change or when you are not getting results in the format that you expect.

3.2.1 Understanding the Default Content Policy

The best way to get acquainted with content formatting is to understand what happens when neither the client nor the action method applies any restrictions to the formats that can be used. In this situation, the outcome is simple and predictable.

1. If the action method returns a string, the string is sent unmodified to the client, and the Content-Type header of the response is set to text/plain.
2. For all other data types, including other simple types such as int, the data is formatted as JSON, and the Content-Type header of the response is set to application/json.

Strings get special treatment because they cause problems when they are encoded as JSON. When you encode other simple types, such as the C# int value 2, then the result is a quoted string, such as "2". When you encode a string, you end up with two sets of quotes so that "Hello" becomes ""Hello"". Not all clients cope well with this double encoding, so it is more reliable to use the text/plain format and sidestep the issue entirely. This is rarely an issue because few applications send string values; it is more common to send objects in the JSON format. To see the default policy, add a class file named ContentController.cs to the WebApps/Controllers folder with the code shown in Listing 20-10.

Listing 20-10. The Contents of the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;
using WebApp.Models;
namespace WebApp.Controllers {
    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;
        public ContentController(DataContext dataContext) {
            context = dataContext;
        }
        [HttpGet("string")]
        public string GetString() => "This is a string response";
        [HttpGet("object")]
        public async Task<Product> GetObject() {
            return await context.Products.FirstAsync();
        }
    }
}
```

The controller defines actions that return string and object results. Restart ASP.NET Core and use a PowerShell prompt to run the command shown in Listing 20-11; this command sends a request that invokes the GetString action method, which returns a string.

Listing 20-11. Requesting a String Response

```
Invoke-WebRequest http://localhost:5000/api/content/string | select @{n='Content-Type';e={
$_.Headers."Content-Type" }}, Content
```

This command sends a GET request to the /api/content/string URL and processes the response to display the Content-Type header and the content from the response. The command produces the following output, which shows the Content-Type header for the response:

```
Content-Type                    Content
------------                    ------
text/plain; charset=utf-8       This is a string response
```

Next, run the command shown in Listing 20-12, which sends a request that will be handled by the GetObject action method.

Listing 20-12. Requesting an Object Response

```
Invoke-WebRequest http://localhost:5000/api/content/object | select @{n='Content-
Type';e={
$_.Headers."Content-Type" }}, Content
```

This command produces the following output, formatted for clarity, that shows that the response has been encoded as JSON:

```
Content-Type                     Content
------------                     -------
application/json; charset=utf-8 {"productId":1,"name":"Kayak",
                                 "price":275.00,"categoryId":1,"supplierId":1}
```

3.2.2 Understanding Content Negotiation

Most clients include an Accept header in a request, which specifies the set of formats that they are willing to receive in the response, expressed as a set of MIME types. Here is the Accept header that Google Chrome sends in requests:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,
        */*;q=0.8
```

This header indicates that Chrome can handle the HTML and XHTML formats (XHTML is an XML-compliant dialect of HTML), XML, and the WEBP image format (which is the animated PNG image format).

The q values in the header specify relative preference, where the value is 1.0 by default. Specifying a q value for 0.9 for application/xml tells the server that Chrome will accept XML data but prefers to deal with HTML or XHTML. The */* item tells the server that Chrome will accept any format, but its q value specifies that it is the lowest preference of the specified types. Putting this together means that the Accept header sent by Chrome provides the server with the following information:

1. Chrome prefers to receive HTML or XHTML data or WEBP and APNG images.
2. If those formats are not available, then the next most preferred format is XML.
3. If none of the preferred formats is available, then Chrome will accept any format.

You might assume from this that you can change the format produced by the ASP.NET Core application by setting the Accept header, but it doesn't work that way—or, rather, it doesn't work that way just yet because there is some preparation required.

To see what happens when the Accept header is changed, use a PowerShell prompt to run the command shown in Listing 20-13, which sets the Accept header to tell ASP.NET Core that the client is willing to receive only XML data.

Listing 20-13. Requesting XML Data

```
Invoke-WebRequest http://localhost:5000/api/content/object -Headers
@{Accept="application/xml"} | select
@{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

Here are the results, which show that the application has sent an application/json response:

```
Content-Type                     Content
------------                     -------
application/json; charset=utf-8 {"productId":1,"name":"Kayak",
                                 "price":275.00,"categoryId":1,"supplierId":1}
```

Including the Accept header has no effect on the format, even though the ASP.NET Core application sent the client a format that it hasn't specified. The problem is that, by default, the MVC Framework is configured to only use JSON. Rather than return an error, the MVC Framework sends JSON data in the hope that the client can process it, even though it was not one of the formats specified by the request Accept header.

3.2.2.1 Enabling XML Formatting

For content negotiation to work, the application must be configured so there is some choice in the formats that can be used. Although JSON has become the default format for web applications, the MVC Framework can also support encoding data as XML, as shown in Listing 20-14.

■ **Tip** You can create your own content format by deriving from the Microsoft.AspNetCore.Mvc.Formatters.OutputFormatter class. this is rarely used because creating a custom data format isn't a useful way of exposing the data in your application, and the most common formats—JSOn and XML—are already implemented.

Listing 20-14. Enabling XML Formatting in the Startup.cs File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllers().AddNewtonsoftJson().AddXmlSerializerFormatters();
services.Configure<MvcNewtonsoftJsonOptions>(opts => {
        opts.SerializerSettings.NullValueHandling
            = Newtonsoft.Json.NullValueHandling.Ignore;
    });
}
...
```

The XML Serializer has some limitations, including the inability to deal with Entity Framework Core navigation properties because they are defined through an interface. To create an object that can be serialized, Listing 20-15 uses ProductBindingTarget defined in Chapter 19.

Listing 20-15. Creating a Serializable Object in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks; using WebApp.Models;
namespace WebApp.Controllers {
    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;
        public ContentController(DataContext dataContext) {
            context = dataContext;
        }
        [HttpGet("string")]
        public string GetString() => "This is a string response";
        [HttpGet("object")]
        public async Task<ProductBindingTarget> GetObject() {
            Product p = await context.Products.FirstAsync();
            return new ProductBindingTarget() {
                Name = p.Name, Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            };
        }
    }
}
```

When the MVC Framework had only the JSON format available, it had no choice but to encode responses as JSON. Now that there is a choice, you can see the content negotiation process working more fully. Restart ASP.NET Core MVC and run the command in Listing 20-13 again to

request XML data, and you will see the following output (from which I have omitted he namespace attributes for brevity):

```
Content-Type                    Content
------------                    -------
application/xml; charset=utf-8 <ProductBindingTarget>
                                 <Name>Kayak</Name>
                                 <Price>275.00</Price>
                                 <CategoryId>1</CategoryId>
                                 <SupplierId>1</SupplierId>
                               </ProductBindingTarget>
```

3.2.2.2 Fully Respecting Accept Headers

The MVC Framework will always use the JSON format if the Accept header contains */*, indicating any format, even if there are other supported formats with a higher preference. This is an odd feature that is intended to deal with requests from browsers consistently, although it can be a source of confusion. Run the command shown in Listing 20-16 to send a request with an Accept header that requests XML but will accept any other format if XML isn't available.

Listing 20-16. Requesting an XML Response with a Fallback

```
Invoke-WebRequest http://localhost:5000/api/content/object -Headers
@{Accept="application/xml,*/*;q=0.8"} | select @{n='Content-Type';e={ $_.Headers."Content-
Type" }}, Content
```

Even though the Accept header tells the MVC Framework that the client prefers XML, the presence of the */* fallback means that a JSON response is sent. A related problem is that a JSON response will be sent when the client requests a format that the MVC Framework hasn't been configured to produce, which you can see by running the command shown in Listing 20-17.

Listing 20-17. Requesting a PNG Response

```
Invoke-WebRequest http://localhost:5000/api/content/object -Headers @{Accept="img/png"} |
select @{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

The commands in Listing 20-16 and Listing 20-17 both produce this response:

```
Content-Type                     Content
------------                     -------
application/json; charset=utf-8 {"name":"Kayak","price":275.00,
                                 "categoryId":1,"supplierId":1}
```

In both cases, the MVC Framework returns JSON data, which may not be what the client is expecting. Two configuration settings are used to tell the MVC Framework to respect the Accept setting sent by the client and not send JSON data by default. To change the configuration, add the statements shown in Listing 20-18 to the Startup class.

Listing 20-18. Configuring Content Negotiation in the Startup.cs File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllers().AddNewtonsoftJson().AddXmlSerializerFormatters();
    services.Configure<MvcNewtonsoftJsonOptions>(opts => {
        opts.SerializerSettings.NullValueHandling
            = Newtonsoft.Json.NullValueHandling.Ignore;
    });
    services.Configure<MvcOptions>(opts => {
        opts.RespectBrowserAcceptHeader = true;
        opts.ReturnHttpNotAcceptable = true;
    });
}
...
```

The options pattern is used to set the properties of a MvcOptions object. Setting RespectBrowserAcceptHeader to true disables the fallback to JSON when the Accept header contains */*. Setting ReturnHttpNotAcceptable to true disables the fallback to JSON when the client requests an unsupported data format.

Restart ASP.NET Core and repeat the command from Listing 20-16. Instead of a JSON response, the format preferences specified by the Accept header will be respected, and an XML response will be sent. Repeat the command from Listing 20-17, and you will receive a response with the 406 status code.

```
...
Invoke-WebRequest : The remote server returned an error: (406) Not Acceptable.
...
```

Sending a 406 code indicates there is no overlap between the formats the client can handle and the formats that the MVC Framework can produce, ensuring that the client doesn't receive a data format it cannot process.

3.2.3 Specifying an Action Result Format

The data formats that the MVC Framework can use for an action method result can be constrained using the Produces attribute, as shown in Listing 20-19.

■ **Tip** the Produces attribute is an example of a filter, which allows attributes to alter requests and responses. See Chapter 30 for more details.

Listing 20-19. Specifying a Data Format in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks; using WebApp.Models;
namespace WebApp.Controllers {
    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;
        public ContentController(DataContext dataContext) {
            context = dataContext;
        }
        [HttpGet("string")]
        public string GetString() => "This is a string response";
        [HttpGet("object")]
        [Produces("application/json")]
        public async Task<ProductBindingTarget> GetObject() {
            Product p = await context.Products.FirstAsync();
            return new ProductBindingTarget() {
                Name = p.Name, Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            };
        }
    }
}
```

The argument for the attribute specifies the format that will be used for the result from the action, and more than one type can be specified. The Produces attribute restricts the types that the MVC Framework will consider when processing an Accept header. To see the effect of the Produces attribute, use a PowerShell prompt to run the command shown in Listing 20-20.

Listing 20-20. Requesting Data

```
Invoke-WebRequest http://localhost:5000/api/content/object -Headers
@{Accept="application/xml,application/ json;q=0.8"} | select @{n='Content-Type';e={
$_.Headers."Content-Type" }}, Content
```

The Accept header tells the MVC Framework that the client prefers XML data but will accept JSON. The Produces attribute means that XML data isn't available as the data format for the GetObject action method and so the JSON serializer is selected, which produces the following response:

```
Content-Type                     Content
------------                     -------
application/json; charset=utf-8 {"name":"Kayak","price":275.00,
                                 "categoryId":1,"supplierId":1}
```

3.2.4 Requesting a Format in the URL

The Accept header isn't always under the control of the programmer who is writing the client. In such situations, it can be helpful to allow the data format for the response to be requested using the URL. This feature is enabled by decorating an action method with the FormatFilter attribute and ensuring there is a format segment variable in the action method's route, as shown in Listing 20-21.

Listing 20-21. Enabling Formatting in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks; using WebApp.Models;
namespace WebApp.Controllers {
    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;
        public ContentController(DataContext dataContext) {
            context = dataContext;
        }
        [HttpGet("string")]
        public string GetString() => "This is a string response";
        [HttpGet("object/{format?}")]
        [FormatFilter]
        [Produces("application/json", "application/xml")]
        public async Task<ProductBindingTarget> GetObject() {
            Product p = await context.Products.FirstAsync();
            return new ProductBindingTarget() {
                Name = p.Name, Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            };
        }
    }
}
```

The FormatFilter attribute is an example of a filter, which is an attribute that can modify requests and responses, as described in Chapter 30. This filter gets the value of the format segment

variable from the route that matched the request and uses it to override the Accept header sent by the client. I have also expanded the range of types specified by the Produces attribute so that the action method can return both JSON and XML responses.

Each data format supported by the application has a shorthand: xml for XML data and json for JSON data. When the action method is targeted by a URL that contains one of these shorthand names, the Accept header is ignored, and the specified format is used. To see the effect, restart ASP.NET Core and use the browser to request http://localhost:5000/api/content/object/json and http://localhost:5000/api/content/object/xml, which produce the responses shown in Figure 20-6.

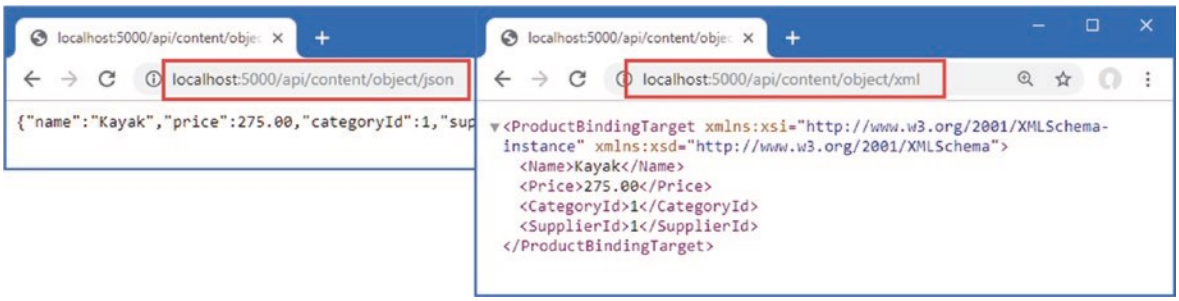


Figure 20-6. *Requesting data formats in the URL*

3.2.5 Restricting the Formats Received by an Action Method

Most content formatting decisions focus on the data formats the ASP.NET Core application sends to the client, but the same serializers that deal with results are used to deserialize the data sent by clients in request bodies. The deserialization process happens automatically, and most applications will be happy to accept data in all the formats they are configured to send. The example application is configured to send JSON and XML data, which means that clients can send JSON and XML data in requests. The Consumes attribute can be applied to action methods to restrict the data types it will handle, as shown in Listing 20-22.

Listing 20-22. Adding Action Methods in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks; using WebApp.Models;
namespace WebApp.Controllers {
    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;
        public ContentController(DataContext dataContext) {
            context = dataContext;
        }
        [HttpGet("string")]
        public string GetString() => "This is a string response";
```

```
        [HttpGet("object/{format?}")]
        [FormatFilter]
        [Produces("application/json", "application/xml")]
        public async Task<ProductBindingTarget> GetObject() {
            Product p = await context.Products.FirstAsync();
            return new ProductBindingTarget() {
                Name = p.Name, Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            };
        }
        [HttpPost]
        [Consumes("application/json")]
        public string SaveProductJson(ProductBindingTarget product) {
            return $"JSON: {product.Name}";
        }
        [HttpPost]
        [Consumes("application/xml")]
        public string SaveProductXml(ProductBindingTarget product) {
            return $"XML: {product.Name}";
        }
    }
}
```

The new action methods are decorated with the Consumes attribute, restricting the data types that each can handle. The combination of attributes means that HTTP POST attributes whose Content-Type header is application/json will be handled by the SaveProductJson action method. HTTP POST requests whose Content-Type header is application/xml will be handled by the SaveProductXml action method. Restart ASP.NET Core and use a PowerShell command prompt to run the command shown in Listing 20-23 to send JSON data to the example application.

Listing 20-23. Sending JSON Data

```
Invoke-RestMethod http://localhost:5000/api/content -Method POST -Body  (@{ Name="Swimming
Goggles";
Price=12.75; CategoryId=1; SupplierId=1} | ConvertTo-Json) -ContentType "application/json"
```

The request is automatically routed to the correct action method, which produces the following response:

```
JSON: Swimming Goggles
```

Run the command shown in Listing 20-24 to send XML data to the example application.

Listing 20-24. Sending XML Data

```
Invoke-RestMethod http://localhost:5000/api/content -Method POST -Body
"<ProductBindingTarget><Name>Kayak
</Name><Price>275.00</Price><CategoryId>1</CategoryId><SupplierId>1</SupplierId></ProductBi
ndingTarget>"
-ContentType "application/xml"
```

The request is routed to the SaveProductXml action method and produces the following response:

```
XML: Kayak
```

The MVC Framework will send a 415 - Unsupported Media Type response if a request is sent with a Content-Type header that doesn't match the data types that the application supports.

3.3 Documenting and Exploring Web Services

When you are responsible for developing both the web service and its client, the purpose of each action and its results are obvious and are usually written at the same time. If you are responsible for a web service that is consumed by third-party developers, then you may need to provide documentation that describes how the web service works. The OpenAPI specification, which is also known as Swagger, describes web services in a way that can be understood by other programmers and consumed programmatically. In this section, I demonstrate how to use OpenAPI to describe a web service and show you how to fine-tune that description.

3.3.1 Resolving Action Conflicts

The OpenAPI discovery process requires a unique combination of the HTTP method and URL pattern for each action method. The process doesn't support the Consumes attribute, so a change is required to the ContentController to remove the separate actions for receiving XML and JSON data, as shown in Listing 20-25.

Listing 20-25. Removing an Action in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks; using WebApp.Models;
namespace WebApp.Controllers {
    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;
        public ContentController(DataContext dataContext) {
            context = dataContext;
```

```
        }
        [HttpGet("string")]
        public string GetString() => "This is a string response";
        [HttpGet("object/{format?}")]
        [FormatFilter]
        [Produces("application/json", "application/xml")]
        public async Task<ProductBindingTarget> GetObject() {
            Product p = await context.Products.FirstAsync();
            return new ProductBindingTarget() {
                Name = p.Name, Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            };
        }
        [HttpPost]
        [Consumes("application/json")]
        public string SaveProductJson(ProductBindingTarget product) {
            return $"JSON: {product.Name}";
        }
        //[HttpPost]
        //[Consumes("application/xml")]
        //public string SaveProductXml(ProductBindingTarget product) {
        //    return $"XML: {product.Name}";
        //}
    }
}
```

Commenting out one of the action methods ensures that each remaining action has a unique combination of HTTP method and URL.

3.3.2 Installing and Configuring the Swashbuckle Package

The Swashbuckle package is the most popular ASP.NET Core implementation of the OpenAPI specification and will automatically generate a description for the web services in an ASP.NET Core application. The package also includes tools that consume that description to allow the web service to be inspected and tested.

Open a new PowerShell command prompt, navigate to the folder that contains the WebApp.csproj file, and run the commands shown in Listing 20-26 to install the NuGet package. If you are using Visual Studio, you can select Project ➤ Manage Nuget Packages and install the package through the Visual Studio package user interface. ***Listing 20-26.*** Adding a Package to the Project

```
dotnet add package Swashbuckle.AspNetCore --version 5.0.0-rc2
```

Add the statements shown in Listing 20-27 to the Startup class to add the services and middleware provided by the Swashbuckle package.

Listing 20-27. Configuring Swashbuckle in the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq; using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models; using Microsoft.AspNetCore.Mvc;
using Microsoft.OpenApi.Models;
namespace WebApp {
    public class Startup {
        public Startup(IConfiguration config) {
            Configuration = config;
        }         public IConfiguration Configuration { get; set; }
        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllers()
                .AddNewtonsoftJson().AddXmlSerializerFormatters();
            services.Configure<MvcNewtonsoftJsonOptions>(opts => {
                opts.SerializerSettings.NullValueHandling
                    = Newtonsoft.Json.NullValueHandling.Ignore;
            });
            services.Configure<MvcOptions>(opts => {
                opts.RespectBrowserAcceptHeader = true;
                opts.ReturnHttpNotAcceptable = true;             });
            services.AddSwaggerGen(options => {
                options.SwaggerDoc("v1",
                    new OpenApiInfo { Title = "WebApp", Version = "v1" });
            });
        }
        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<TestMiddleware>();
            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });;
                endpoints.MapControllers();
            });
            app.UseSwagger();
            app.UseSwaggerUI(options => {
                options.SwaggerEndpoint("/swagger/v1/swagger.json", "WebApp");
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

There are two features set up by the statements in Listing 20-27. The feature generates an OpenAPI description of the web services that the application contains. You can see the description by restarting ASP.NET Core and using the browser to request the URL http://localhost:5000/swagger/v1/swagger.json, which produces the response shown in Figure 20-7. The OpenAPI format is verbose, but you can see each URL that the web service controllers support, along with details of the data each expects to receive and the range of responses that it will generate.

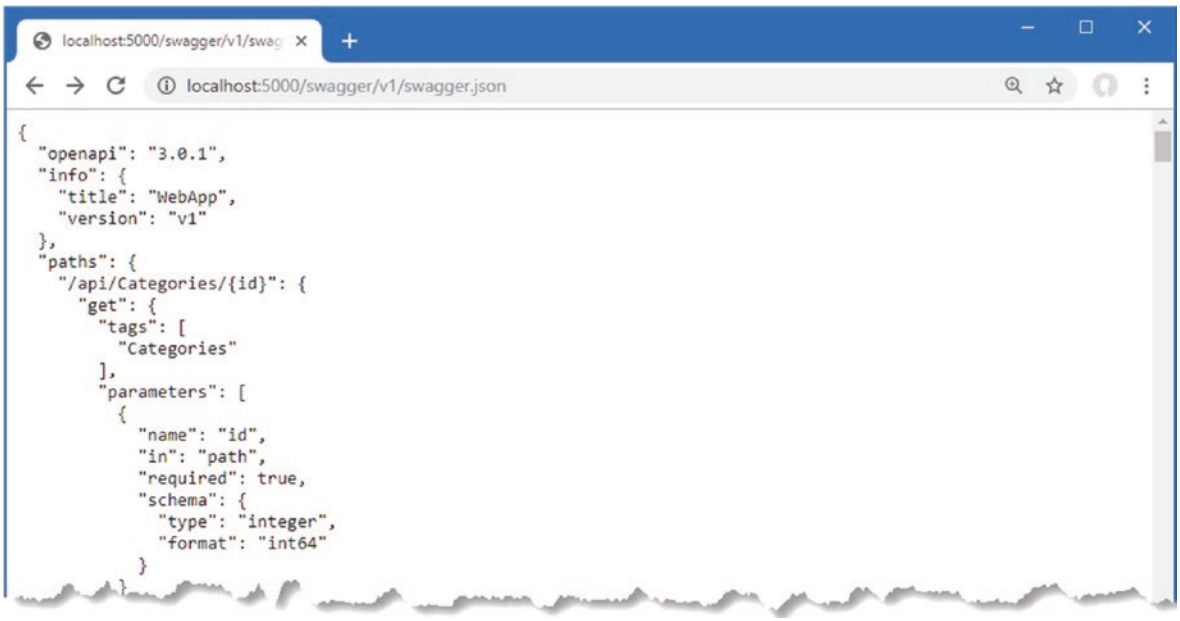


Figure 20-7. *The OpenAPI description of the web service*

The second feature is a UI that consumes the OpenAPI description of the web service and presents the information in a more easily understood way, along with support for testing each action. Use the browser to request http://localhost:5000/swagger, and you will see the interface shown in Figure 20-8. You can expand each action to see details, including the data that is expected in the request and the different responses that the client can expect.

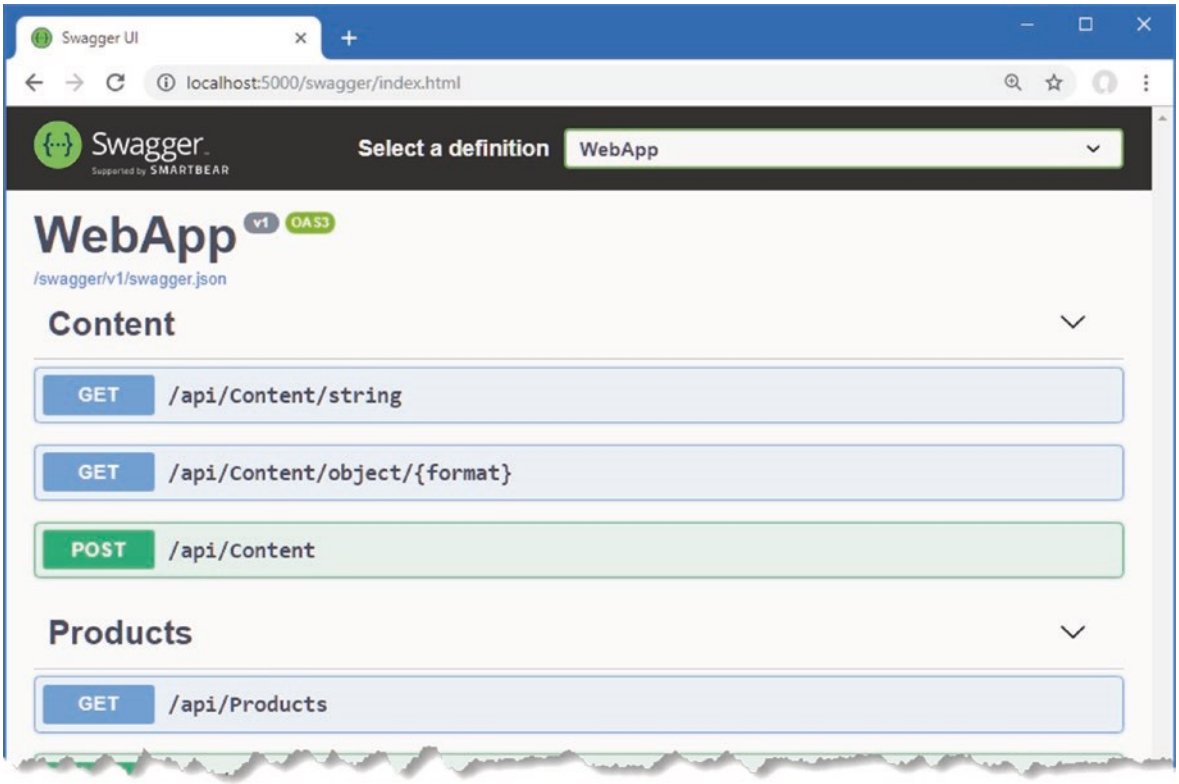


Figure 20-8. *The OpenAPI explorer interface*

3.3.3 Fine-Tuning the API Description

Relying on the API discovery process can produce a result that doesn't truly capture the web service. You can see this by examining the entry in the Products section that describes GET requests matched by the /api/Product/{id} URL pattern. Expand this item and examine the response section, and you will see there is only one status code response that will be returned, as shown in Figure 20-9.

Figure 20-9. *The data formats listed in the OpenAPI web service description*

The API discovery process makes assumptions about the responses produced by an action method and doesn't always reflect what can really happen. In this case, the GetProduct action method in the ProductController class can return another response that the discovery process hasn't detected.

```
...
[HttpGet("{id}")]
public async Task<IActionResult> GetProduct(long id) {
    Product p = await context.Products.FindAsync(id);
    if (p == null) {
        return NotFound();
    }
    return Ok(new {
        ProductId = p.ProductId, Name = p.Name,
        Price = p.Price, CategoryId = p.CategoryId,
        SupplierId = p.SupplierId
    });
} ...
```

If a third-party developer attempts to implement a client for the web service using the OpenAPI data, they won't be expecting the 404 - Not Found response that the action sends when it can't find an object in the database.

3.3.3.1 Running the API Analyzer

ASP.NET Core includes an analyzer that inspects web service controllers and highlights problems like the one described in the previous section. To enable the analyzer, add the elements shown in Listing 20-28 to the WebApp.cspoj file. (If you are using Visual

Studio, right-click the WebApp project item in the Solution Explorer and select Edit Project File from the popup menu.)

Listing 20-28. Enabling the Analyzer in the WebApp.csproj File in the WebApp Folder

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson"
        Version="3.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.1">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;
          buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
         Version="3.1.1" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="5.0.0-rc2" />
  </ItemGroup>
  <PropertyGroup>
    <IncludeOpenAPIAnalyzers>true</IncludeOpenAPIAnalyzers>
```

```
  </PropertyGroup>
</Project>
```

If you are using Visual Studio, you will see any problems detected by the API analyzer shown in the controller class file, as shown in Figure 20-10.

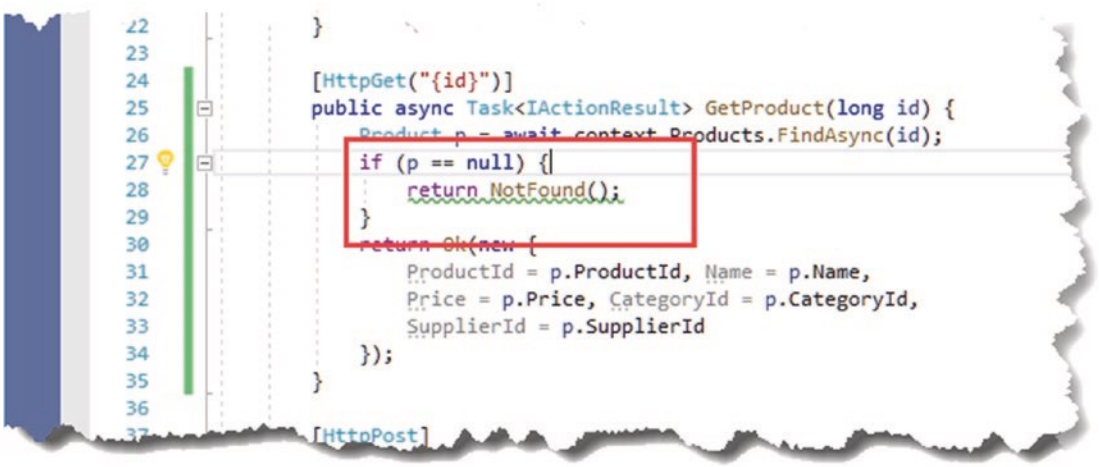


Figure 20-10. *A problem detected by the API analyzer*

If you are using Visual Studio Code, you will see warning messages when the project is compiled, either using the dotnet build command or when it is executed using the dotnet run command. When the project is compiled, you will see this message that describes the issue in the ProductController class:

```
Controllers\ProductsController.cs(28,9): warning API1000: Action method returns undeclared
status code '404'.
[C:\WebApp\WebApp.csproj]
    1 Warning(s)
    0 Error(s)
```

3.3.3.2 Declaring the Action Method Result Type

To fix the problem detected by the analyzer, the ProducesResponseType attribute can be used to declare each of the response types that the action method can produce, as shown in Listing 20-29.

Listing 20-29. Declaring the Result in the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic; using Microsoft.Extensions.Logging;
using System.Linq; using System.Threading.Tasks; using Microsoft.AspNetCore.Http; namespace
WebApp.Controllers {
    [ApiController]
    [Route("api/[controller]")]
```

```
    public class ProductsController : ControllerBase {
        private DataContext context;
        public ProductsController(DataContext ctx) {
            context = ctx;
        }
        [HttpGet]
        public IAsyncEnumerable<Product> GetProducts() {
            return context.Products;
        }
        [HttpGet("{id}")]
        [ProducesResponseType(StatusCodes.Status200OK)]
        [ProducesResponseType(StatusCodes.Status404NotFound)]
        public async Task<IActionResult> GetProduct(long id) {
            Product p = await context.Products.FindAsync(id);
            if (p == null) {
                return NotFound();
            }
            return Ok(new {
                ProductId = p.ProductId, Name = p.Name,
                Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            });
        }
        // ...action methods omitted for brevity...
    }
}
```

Restart ASP.NET Core and use a browser to request http://localhost:5000/swagger, and you will see the description for the action method has been updated to reflect the 404 response, as shown in Figure 20-11.

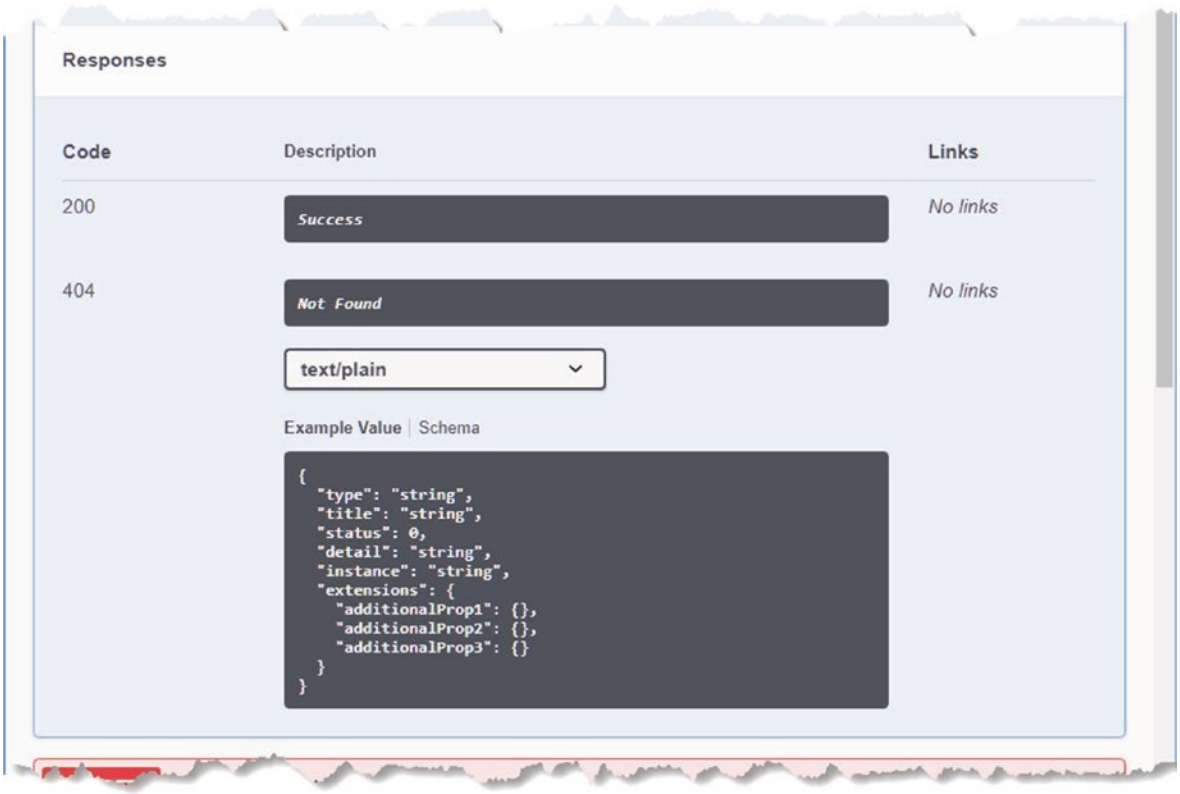


Figure 20-11. *Reflecting all the status codes produced by an action method*