Building and Consuming Web Services

This section is about learning how to build web services using the ASP.NET Core Web API, and then consuming web services using HTTP clients that could be any other type of .NET app, including a website, a Windows desktop app, or a mobile app.

This section assumes knowledge and skills that you learned in *Section 4*, *Working with Databases Using Entity Framework Core*, and Sections 6 to 8, about building websites using ASP.NET Core.

In this section, we will cover the following topics:

- Building web services using the ASP.NET Core Web API
- Documenting and testing web services
- Consuming services using HTTP clients
- Implementing advanced features
- Understanding other communication technologies

Building web services using the ASP.NET Core Web API

Before we build a modern web service, we need to cover some background to set the context for this section.

Understanding web service acronyms

Although HTTP was originally designed to request and respond with HTML and other resources for humans to look at, it is also good for building services.

Roy Fielding stated in his doctoral dissertation, describing the **Representational State Transfer** (**REST**) architectural style, that the HTTP standard would be great for building services because it defines the following:

- URIs to uniquely identify resources, like `https://localhost:5001/api/products/23`.
- Methods to perform common tasks on those resources, like `GET`, `POST`, `PUT`, and `DELETE`.
- The ability to negotiate the media type of content exchanged in requests and responses, such as XML and JSON. Content negotiation happens when the client specifies a request header like `Accept: application/xml,*/*;q=0.8`. The default response format used by the ASP.NET Core Web API is JSON, which means one of the response headers would be `Content-Type: application/json; charset=utf-8`.

More Information: You can read more about media types at the following link: http://en.wikipedia.org/wiki/Media_type

Web services are services that use the HTTP communication standard, so they are sometimes called HTTP or RESTful services. HTTP or RESTful services are what this section is about.

Web services can also mean **Simple Object Access Protocol** (**SOAP**) services that implement some of the WS-* standards.

More Information: You can read more about WS-* standards at the following link:
https://en.wikipedia.org/wiki/List_of_web_service_specifications

Microsoft .NET Framework 3.0 and later includes a **remote procedure call** (**RPC**) technology named **Windows Communication Foundation** (**WCF**), which makes it easy for developers to create services including SOAP services that implement WS-* standards, but Microsoft decided it is legacy and has not ported it to modern .NET platforms.

gRPC is a modern cross-platform open source RPC framework created by Google (the "g" in gRPC).

More Information: You can read about using gRPC as an alternative to WCF at the following link:
https://devblogs.microsoft.com/premier-developer/grpc-asp-net-core-as-a-migration-path-for-wcfs-in-net-core/

Creating an ASP.NET Core Web API project

We will build a web service that provides a way to work with data in the `Northwind` database using ASP.NET Core so that the data can be used by any client application on any platform that can make HTTP requests and receive HTTP responses:

1. In the folder named `PracticalApps`, create a folder named `NorthwindService`.
2. In Visual Studio Code, open the `PracticalApps` workspace and add the `NorthwindService` folder.
3. Navigate to **Terminal** | **New Terminal** and select `NorthwindService`.
4. In **TERMINAL**, use the `webapi` template to create a new ASP.NET Core Web API project, as shown in the following command:

```
dotnet new webapi
```

5. Set `NorthwindService` as the active project and add required assets when prompted.
6. In the `Controllers` folder, open `WeatherForecastController.cs`, as shown in the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
namespace NorthwindService.Controllers
{
  [ApiController]
  [Route("[controller]")]
  public class WeatherForecastController : ControllerBase
  {
    private static readonly string[] Summaries = new[]
    {
      "Freezing", "Bracing", "Chilly", "Cool", "Mild",
      "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
    };
    private readonly ILogger<WeatherForecastController> _logger;
    // The Web API will only accept tokens 1) for users, and
    // 2) having the access_as_user scope for this API
    static readonly string[] scopeRequiredByApi =
      new string[] { "access_as_user" };
    public WeatherForecastController(
      ILogger<WeatherForecastController> logger)
    {
      _logger = logger;
    }
    [HttpGet]
    public IEnumerable<WeatherForecast> Get()
    {
      var rng = new Random();
      return Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        {
          Date = DateTime.Now.AddDays(index),
          TemperatureC = rng.Next(-20, 55),
```

```
          Summary = Summaries[rng.Next(Summaries.Length)]
        })
        .ToArray();
    }
  }
}
```

While reviewing the preceding code, note the following:

- The `Controller` class inherits from `ControllerBase`. This is simpler than the `Controller` class used in MVC because it does not have methods like `View` to generate HTML responses using a Razor file.
- The `[Route]` attribute registers the `weatherforecast` relative URL for clients to use to make HTTP requests that will be handled by this controller. For example, an HTTP request for `http://localhost:5001/weatherforecast/` would be handled by this controller. Some developers like to prefix the controller name with `api/`, which is a convention to differentiate between MVC and Web API in mixed projects. If you use `[controller]` as shown, it uses the characters before `Controller` in the class name, in this case, `WeatherForecast`, or you can simply enter a different name without the square brackets, for example, `[Route("api/forecast")]`.
- The `[ApiController]` attribute was introduced with ASP.NET Core 2.1 and it enables REST-specific behavior for controllers, like automatic HTTP 400 responses for invalid models, as you will see later in this section.
- The `scopeRequiredByApi` field can be used to add authorization to ensure that your web API is only called by client apps and websites on behalf of users who have the right scopes.

  More Information: You can read more about verifying that the tokens used to call your web APIs are requested with the expected claims at the following link: https://docs.microsoft.com/en-us/azure/active-directory/develop/scenario-protected-web-api-verification-scope-app-roles

- The `[HttpGet]` attribute registers the `Get` method in the `Controller` class to respond to HTTP `GET` requests, and its implementation uses a `Random` object to return an array of `WeatherForecast` values with random temperatures and summaries like `Bracing` or `Balmy` for the next five days of weather.

7. Add a second `Get` method that allows the call to specify how many days ahead the forecast should be by implementing the following:
   - Add a comment above the original method to show the GET and URL path it responds to.
   - Add a new method with an integer parameter named `days`.
   - Cut and paste the original `Get` method implementation code statements into the new `Get` method.
   - Modify the new method to create an `IEnumerable` of `int` values up to the number of days requested, and modify the original `Get` method to call the new `Get` method and pass the value `5`.

Your methods should be as shown highlighted in the following code:

```
// GET /weatherforecast
[HttpGet]
public IEnumerable<WeatherForecast> Get() // original method
{
  return Get(5); // five day forecast
}
// GET /weatherforecast/7
[HttpGet("{days:int}")]
public IEnumerable<WeatherForecast> Get(int days) // new method
{
  var rng = new Random();
  return Enumerable.Range(1, days).Select(index =>
    new WeatherForecast
    {
      Date = DateTime.Now.AddDays(index),
      TemperatureC = rng.Next(-20, 55),
      Summary = Summaries[rng.Next(Summaries.Length)]
    })
    .ToArray();
}
```

In the `[HttpGet]` attribute, note the route format pattern that constrains the `days` parameter to `int` values.

More Information: You can read more about route constraints at the following link: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing#route-constraint-reference

Reviewing the web service's functionality

Now, we will test the web service's functionality:

1. In **TERMINAL**, start the website by entering `dotnet run`.
2. Start Chrome, navigate to `https://localhost:5001/`, and note you will get a 404 status code response because we have not enabled static files and there is not an `index.html`, nor is there an MVC controller with a route configured, either. Remember that this project is not designed for a human to view and interact with.
3. In Chrome, show the **Developer tools**, navigate to `https://localhost:5001/weatherforecast`, and note the Web API service should return a JSON document with five random weather forecast objects in an array, as shown in the following screenshot:

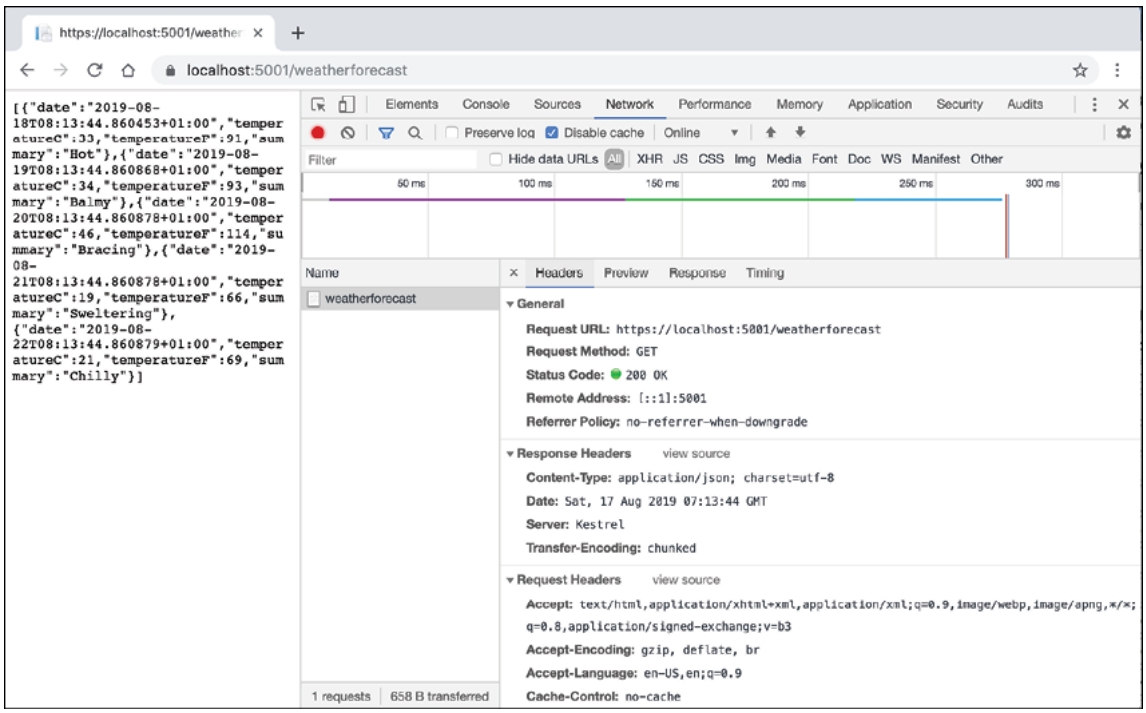


Figure 18.1: A request and response from a weather forecast web service

4. Close **Developer Tools**.
5. Navigate to `https://localhost:5001/weatherforecast/14`, and note the response when requesting a two-week weather forecast, as shown in the following screenshot:

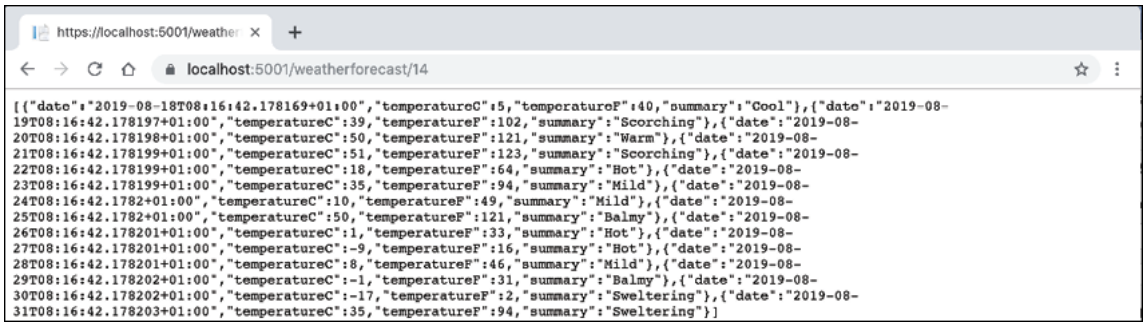


Figure 18.2: A two-week weather forecast as JSON

6. Close Chrome.
7. In **TERMINAL**, press `Ctrl + C` to stop the console application and shut down the Kestrel web server that is hosting your ASP.NET Core Web API service.

Creating a web service for the Northwind database

Unlike MVC controllers, Web API controllers do not call Razor views to return HTML responses for humans to see in browsers. Instead, they use **content negotiation** with the client application that made the HTTP request to return data in formats such as XML, JSON, or X-WWW-FORM-URLENCODED in their HTTP response.

The client application must then deserialize the data from the negotiated format. The most commonly used format for modern web services is **JavaScript Object Notation** (**JSON**) because it is compact and works natively with JavaScript in a browser when building **Single-Page Applications** (**SPAs**) with client-side technologies like Angular, React, and Vue.

We will reference the Entity Framework Core entity data model for the Northwind database that you created in *Section 6*, *Introducing Practical Applications of C# and .NET*:

1. In the `NorthwindService` project, open `NorthwindService.csproj`.
2. Add a project reference to `NorthwindContextLib`, as shown in the following markup:

```
<ItemGroup>
  <ProjectReference Include=
    "..\NorthwindContextLib\NorthwindContextLib.csproj" />
</ItemGroup>
```

3. In **TERMINAL**, enter the following command and ensure that the project builds:

```
dotnet build
```

4. Open `Startup.cs` and modify it to import the `System.IO`, `Microsoft.EntityFrameworkCore`, `Microsoft.AspNetCore.Mvc.Formatters`, and `Packt.Shared` namespaces, and statically import the `System.Console` class.
5. Add statements to the `ConfigureServices` method, before the call to `AddControllers`, to configure the Northwind data context, as shown in the following code:

```
string databasePath = Path.Combine("..", "Northwind.db");
services.AddDbContext<Northwind>(options =>
  options.UseSqlite($"Data Source={databasePath}"));
```

6. In the call to `AddControllers`, add statements to write the names and supported media types of the default output formatters to the console, and then add XML serializer formatters and set the compatibility to ASP.NET Core 3.0 after the method call to add controller support, as shown in the following code:

```
services.AddControllers(options =>
  {
    WriteLine("Default output formatters:");
    foreach(IOutputFormatter formatter in options.OutputFormatters)
    {
      var mediaFormatter = formatter as OutputFormatter;
      if (mediaFormatter == null)
      {
        WriteLine($" {formatter.GetType().Name}");
      }
      else // OutputFormatter class has SupportedMediaTypes
      {
        WriteLine(" {0}, Media types: {1}",
          arg0: mediaFormatter.GetType().Name,
          arg1: string.Join(", ",
            mediaFormatter.SupportedMediaTypes));
      }
    }
  })
  .AddXmlDataContractSerializerFormatters()
  .AddXmlSerializerFormatters()
  .SetCompatibilityVersion(CompatibilityVersion.Version_3_0);
```

More Information: You can read more about the benefits of setting version compatibility at the following link: https://docs.microsoft.com/en-us/aspnet/core/mvc/compatibility-version

7. Start the web service and note that there are four default output formatters, including ones that convert `null` values into `204 No Content` and ones to support responses that are plain text and JSON, as shown in the following output:

```
Default output formatters:
  HttpNoContentOutputFormatter
  StringOutputFormatter, Media types: text/plain
  StreamOutputFormatter
  SystemTextJsonOutputFormatter, Media types: application/json, text/json, application/*+json
```

8. Stop the web service.

Creating data repositories for entities

Defining and implementing a data repository to provide CRUD operations is good practice. The CRUD acronym includes the following operations:

- C for Create
- R for Retrieve (or Read)
- U for Update
- D for Delete

We will create a data repository for the `Customers` table in `Northwind`. There are only 91 customers in this table, so we will store a copy of the whole table in memory to improve scalability and performance when reading customer records. In a real web service, you should use a distributed cache like **Redis**, an open source data structure store that can be used as a high-performance, high-availability database, cache, or message broker.

More Information: You can read more about Redis at the following link: https://redis.io

We will follow modern good practice and make the repository API asynchronous. It will be instantiated by a `Controller` class using constructor parameter injection, so a new instance is created to handle every HTTP request:

1. In the `NorthwindService` project, create a `Repositories` folder.
2. Add two class files to the `Repositories` folder named `ICustomerRepository.cs` and `CustomerRepository.cs`.
3. The `ICustomerRepository` interface will define five methods, as shown in the following code:

```
using Packt.Shared;
using System.Collections.Generic;
using System.Threading.Tasks;
namespace NorthwindService.Repositories
{
  public interface ICustomerRepository
  {
    Task<Customer> CreateAsync(Customer c);
    Task<IEnumerable<Customer>> RetrieveAllAsync();
    Task<Customer> RetrieveAsync(string id);
    Task<Customer> UpdateAsync(string id, Customer c);
    Task<bool?> DeleteAsync(string id);
  }
}
```

4. The `CustomerRepository` class will implement the five methods, as shown in the following code:

```
using Microsoft.EntityFrameworkCore.ChangeTracking;
using Packt.Shared;
using System.Collections.Generic;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading.Tasks;
namespace NorthwindService.Repositories
{
  public class CustomerRepository : ICustomerRepository
  {
```

```
    // use a static thread-safe dictionary field to cache the customers
    private static ConcurrentDictionary
      <string, Customer> customersCache;
    // use an instance data context field because it should not be
    // cached due to their internal caching
    private Northwind db;
    public CustomerRepository(Northwind db)
    {
      this.db = db;
      // pre-load customers from database as a normal
      // Dictionary with CustomerID as the key,
      // then convert to a thread-safe ConcurrentDictionary
      if (customersCache == null)
      {
        customersCache = new ConcurrentDictionary<string, Customer>(
          db.Customers.ToDictionary(c => c.CustomerID));
      }
    }
    public async Task<Customer> CreateAsync(Customer c)
    {
      // normalize CustomerID into uppercase
      c.CustomerID = c.CustomerID.ToUpper();
      // add to database using EF Core
      EntityEntry<Customer> added = await db.Customers.AddAsync(c);
      int affected = await db.SaveChangesAsync();
      if (affected == 1)
      {
        // if the customer is new, add it to cache, else
        // call UpdateCache method
        return customersCache.AddOrUpdate(c.CustomerID, c, UpdateCache);
      }
      else
      {
        return null;
      }
    }
    public Task<IEnumerable<Customer>> RetrieveAllAsync()
    {
      // for performance, get from cache
      return Task.Run<IEnumerable<Customer>>(
        () => customersCache.Values);
    }
    public Task<Customer> RetrieveAsync(string id)
    {
      return Task.Run(() =>
      {
        // for performance, get from cache
        id = id.ToUpper();
        customersCache.TryGetValue(id, out Customer c);
        return c;
      });
    }
    private Customer UpdateCache(string id, Customer c)
    {
      Customer old;
      if (customersCache.TryGetValue(id, out old))
      {
        if (customersCache.TryUpdate(id, c, old))
        {
          return c;
        }
      }
      return null;
    }
    public async Task<Customer> UpdateAsync(string id, Customer c)
    {
      // normalize customer ID
      id = id.ToUpper();
      c.CustomerID = c.CustomerID.ToUpper();
      // update in database
      db.Customers.Update(c);
      int affected = await db.SaveChangesAsync();
      if (affected == 1)
      {
        // update in cache
        return UpdateCache(id, c);
      }
      return null;
    }
    public async Task<bool?> DeleteAsync(string id)
    {
      id = id.ToUpper();
      // remove from database
      Customer c = db.Customers.Find(id);
      db.Customers.Remove(c);
      int affected = await db.SaveChangesAsync();
      if (affected == 1)
      {
        // remove from cache
        return customersCache.TryRemove(id, out c);
      }
      else
      {
        return null;
      }
    }
  }
}
```

Implementing a Web API controller

There are some useful attributes and methods for implementing a controller that returns data instead of HTML.

With MVC controllers, a route like `/home/index/` tells us the `Controller` class name and the action method name, for example, the `HomeController` class and the `Index` action method.

With Web API controllers, a route like `/weatherforecast/` only tells us the `Controller` class name, for example, `WeatherForecastController`. To determine the action method name to execute, we must map HTTP methods like `GET` and `POST` to methods in the `Controller` class.

You should decorate `Controller` methods with the following attributes to indicate the HTTP method to respond to:

- `[HttpGet]`, `[HttpHead]`: These action methods respond to HTTP `GET` or `HEAD` requests to retrieve a resource and return either the resource and its response headers or just the headers.
- `[HttpPost]`: This action method responds to HTTP `POST` requests to create a new resource.
- `[HttpPut]`, `[HttpPatch]`: These action methods respond to HTTP `PUT` or `PATCH` requests to update an existing resource either by replacing it or updating some of its properties.
- `[HttpDelete]`: This action method responds to HTTP `DELETE` requests to remove a resource.
- `[HttpOptions]`: This action method responds to HTTP `OPTIONS` requests.

More Information: You can read more about the HTTP `OPTIONS` method and other HTTP methods at the following link: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/OPTIONS

An action method can return .NET types like a single `string` value, complex objects defined by a `class`, `record`, or `struct`, or collections of complex objects, and the ASP.NET Core Web API will automatically serialize them into the requested data format set in the HTTP request `Accept` header, for example, JSON, if a suitable serializer has been registered.

For more control over the response, there are helper methods that return an `ActionResult` wrapper around the .NET type.

Declare the action method's return type to be `IActionResult` if it could return different return types based on inputs or other variables. Declare the action method's return type to be `ActionResult<T>` if it will only return a single type but with different status codes.

Good Practice: Decorate action methods with the `[ProducesResponseType]` attribute to indicate all the known types and HTTP status codes that the client should expect in a response. This information can then be publicly exposed to document how a client should interact with your web service. Think of it as part of your formal documentation. Later in this section, you will learn how you can install a code analyzer to give you warnings when you do not decorate your action methods like this.

For example, an action method that gets a product based on an `id` parameter would be decorated with three attributes – one to indicate that it responds to `GET` requests and has an `id` parameter, and two to indicate what happens when it succeeds and when the client has supplied an invalid product ID, as shown in the following code:

```
[HttpGet("{id}")]
[ProducesResponseType(200, Type = typeof(Product))]
[ProducesResponseType(404)]
public IActionResult Get(string id)
```

The `ControllerBase` class has methods to make it easy to return different responses:

- `Ok`: Returns an HTTP 200 status code with a resource converted to the client's preferred format, like JSON or XML. Commonly used in response to an HTTP `GET` request.
- `CreatedAtRoute`: Returns an HTTP 201 status code with the path to the new resource. Commonly used in response to a `POST` request to create a resource that can be performed quickly.
- `Accepted`: Returns an HTTP 202 status code to indicate the request is being processed but has not completed. Commonly used in response to a request that triggers a background process that takes a long time to complete.
- `NoContentResult`: Returns an HTTP 204 status code. Commonly used in response to a `DELETE` request or a `PUT` request to update an existing resource when the response does not need to contain the updated resource.
- `BadRequest`: Returns an HTTP 400 status code with the optional message `string`.
- `NotFound`: Returns an HTTP 404 status code with an automatically populated `ProblemDetails` body (requires a compatibility version of 2.2 or later).

Configuring the customers repository and Web API controller

Now you will configure the repository so that it can be called from within a Web API controller.

You will register a scoped dependency service implementation for the repository when the web service starts up and then use constructor parameter injection to get it in a new Web API controller for working with customers.

More Information: You can read more about dependency injection at the following link: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection

To show an example of differentiating between MVC and Web API controllers using routes, we will use the common `/api` URL prefix convention for the customers controller:

1. Open `Startup.cs` and import the `NorthwindService.Repositories` namespace.
2. Add the following statement to the bottom of the `ConfigureServices` method, which will register the `CustomerRepository` for use at runtime, as shown in the following code:

```
services.AddScoped<ICustomerRepository, CustomerRepository>();
```

3. In the `Controllers` folder, add a new class named `CustomersController.cs`.
4. In the `CustomersController` class file, add statements to define a Web API `Controller` class to work with customers, as shown in the following code:

```
using Microsoft.AspNetCore.Mvc;
using Packt.Shared;
using NorthwindService.Repositories;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
namespace NorthwindService.Controllers
{
  // base address: api/customers
```

```
[Route("api/[controller]")]
[ApiController]
public class CustomersController : ControllerBase
{
  private ICustomerRepository repo;
  // constructor injects repository registered in Startup
  public CustomersController(ICustomerRepository repo)
  {
    this.repo = repo;
  }
  // GET: api/customers
  // GET: api/customers/?country=[country]
  // this will always return a list of customers even if its empty
  [HttpGet]
  [ProducesResponseType(200,
    Type = typeof(IEnumerable<Customer>))]
  public async Task<IEnumerable<Customer>> GetCustomers(
    string country)
  {
    if (string.IsNullOrWhiteSpace(country))
    {
      return await repo.RetrieveAllAsync();
    }
    else
    {
      return (await repo.RetrieveAllAsync())
        .Where(customer => customer.Country == country);
    }
  }
  // GET: api/customers/[id]
  [HttpGet("{id}", Name = nameof(GetCustomer))] // named route
  [ProducesResponseType(200, Type = typeof(Customer))]
  [ProducesResponseType(404)]
  public async Task<IActionResult> GetCustomer(string id)
  {
    Customer c = await repo.RetrieveAsync(id);
    if (c == null)
    {
      return NotFound(); // 404 Resource not found
    }
    return Ok(c); // 200 OK with customer in body
  }
  // POST: api/customers
  // BODY: Customer (JSON, XML)
  [HttpPost]
  [ProducesResponseType(201, Type = typeof(Customer))]
  [ProducesResponseType(400)]
  public async Task<IActionResult> Create([FromBody] Customer c)
  {
    if (c == null)
    {
      return BadRequest(); // 400 Bad request
    }
    if (!ModelState.IsValid)
    {
      return BadRequest(ModelState); // 400 Bad request
    }
    Customer added = await repo.CreateAsync(c);
    return CreatedAtRoute( // 201 Created
      routeName: nameof(GetCustomer),
      routeValues: new { id = added.CustomerID.ToLower() },
      value: added);
  }
  // PUT: api/customers/[id]
  // BODY: Customer (JSON, XML)
  [HttpPut("{id}")]
  [ProducesResponseType(204)]
  [ProducesResponseType(400)]
  [ProducesResponseType(404)]
  public async Task<IActionResult> Update(
    string id, [FromBody] Customer c)
  {
    id = id.ToUpper();
    c.CustomerID = c.CustomerID.ToUpper();
    if (c == null || c.CustomerID != id)
    {
      return BadRequest(); // 400 Bad request
    }
    if (!ModelState.IsValid)
    {
      return BadRequest(ModelState); // 400 Bad request
    }
    var existing = await repo.RetrieveAsync(id);
    if (existing == null)
    {
      return NotFound(); // 404 Resource not found
    }
    await repo.UpdateAsync(id, c);
    return new NoContentResult(); // 204 No content
  }
  // DELETE: api/customers/[id]
  [HttpDelete("{id}")]
  [ProducesResponseType(204)]
  [ProducesResponseType(400)]
  [ProducesResponseType(404)]
  public async Task<IActionResult> Delete(string id)
  {
    var existing = await repo.RetrieveAsync(id);
    if (existing == null)
    {
      return NotFound(); // 404 Resource not found
    }
    bool? deleted = await repo.DeleteAsync(id);
    if (deleted.HasValue && deleted.Value) // short circuit AND
```

```
      {
        return new NoContentResult(); // 204 No content
      }
      else
      {
        return BadRequest( // 400 Bad request
          $"Customer {id} was found but failed to delete.");
      }
    }
  }
}
```

While reviewing this Web API controller class, note the following:

- The `Controller` class registers a route that starts with `api/` and includes the name of the controller, that is, `api/customers`.
- The constructor uses dependency injection to get the registered repository for working with customers.
- There are five methods to perform CRUD operations on customers—two `GET` methods (all customers or one customer), `POST` (create), `PUT` (update), and `DELETE`.
- `GetCustomers` can have a `string` parameter passed with a country name. If it is missing, all customers are returned. If it is present, it is used to filter customers by country.
- `GetCustomer` has a route explicitly named `GetCustomer` so that it can be used to generate a URL after inserting a new customer.
- `Create` decorates the `customer` parameter with `[FromBody]` to tell the model binder to populate it with values from the body of the HTTP `POST` request.
- `Create` returns a response that uses the `GetCustomer` route so that the client knows how to get the newly created resource in the future. We are matching up two methods to create and then get a customer.
- `Create` and `Update` both check the model state of the customer passed in the body of the HTTP request and return a `400 Bad Request` containing details of the model validation errors if it is not valid.

When an HTTP request is received by the service, then it will create an instance of the `Controller` class, call the appropriate action method, return the response in the format preferred by the client, and release the resources used by the controller, including the repository and its data context.

Specifying problem details

A feature added in ASP.NET Core 2.1 and later is a n implementation of a web standard for specifying problem details.

More Information: You can read more about the proposed standard for Problem Details for HTTP APIs at the following link: https://tools.ietf.org/html/rfc7807

In Web API controllers decorated with `[ApiController]` in a project with ASP.NET Core 2.2 or later compatibility enabled, action methods that return `IActionResult` and return a client error status code, that is, `4xx`, will automatically include a serialized instance of the `ProblemDetails` class in the response body.

More Information: You can read more about implementing problem details at the following link: https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.problemdetails

If you want to take control, then you can create a `ProblemDetails` instance yourself and include additional information.

Let's simulate a bad request that needs custom data returned to the client:

1. At the top of the `CustomersController` class, import the `Microsoft.AspNetCore.Http` namespace.
2. At the top of the `Delete` method, add statements to check if the `id` matches the `string` value `"bad"`, and if so, then return a custom problem details object, as shown in the following example code:

```
// take control of problem details
if (id == "bad")
{
  var problemDetails = new ProblemDetails
  {
    Status = StatusCodes.Status400BadRequest,
    Type = "https://localhost:5001/customers/failed-to-delete",
    Title = $"Customer ID {id} found but failed to delete.",
    Detail = "More details like Company Name, Country and so on.",
    Instance = HttpContext.Request.Path
  };
  return BadRequest(problemDetails); // 400 Bad request
}
```

Controlling XML serialization

In the `Startup.cs` file, we added the `XmlSerializer` so that our Web API service can return XML as well as JSON if the client requests that.

However, the `XmlSerializer` cannot serialize interfaces, and our entity classes use `ICollection<T>` to define related child entities. This causes a warning at runtime, for example, for the `Customer` class and its `Orders` property, as shown in the following output:

```
warn: Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerOutputFormatter[1]
      An error occurred while trying to create an XmlSerializer for the type 'Packt.Shared.Customer'.
      System.InvalidOperationException: There was an error reflecting type 'Packt.Shared.Customer'.
       ---> System.InvalidOperationException: Cannot serialize member 'Packt.Shared.Customer.Orders' of type 'System.Collections.Generic.ICollection`1
```

We can prevent this warning by excluding the `Orders` property when serializing a `Customer` to XML:

1. In the `NorthwindEntitiesLib` project, open `Customers.cs`.
2. Import the `System.Xml.Serialization` namespace.
3. Decorate the `Orders` property with an attribute to ignore it when serializing, as shown highlighted in the following code:

```
[InverseProperty(nameof(Order.Customer))]
[XmlIgnore]
public virtual ICollection<Order> Orders { get; set; }
```

Documenting and testing web services

You can easily test a web service by making HTTP `GET` requests using a browser. To test other HTTP methods, we need a more advanced tool.

Testing GET requests using a browser

You will use Chrome to test the three implementations of a `GET` request – for all customers, for customers in a specified country, and for a single customer using their unique customer ID:

1. In **TERMINAL**, start the `NorthwindService` Web API web service by entering the command `dotnet run`.
2. In Chrome, navigate to `https://localhost:5001/api/customers` and note the JSON document returned, containing all 91 customers in the `Northwind` database (unsorted), as shown in the following screenshot:

Figure 18.3: Customers from the Northwind database as JSON

3. Navigate to `https://localhost:5001/api/customers/?country=Germany` and note the JSON document returned, containing only the customers in Germany, as shown in the following screenshot:

Figure 18.4: A list of customers from Germany as JSON

If you get an empty array returned, then make sure you have entered the country name using the correct casing because the database query is case-sensitive.

4. Navigate to `https://localhost:5001/api/customers/alfki` and note the JSON document returned containing only the customer named **Alfreds Futterkiste**, as shown in the following screenshot:

Figure 18.5: Specific customer information as JSON

We do not need to worry about casing for the customer `id` value because inside the `Controller` class, we normalized the `string` value to uppercase in code.

But how can we test the other HTTP methods, such as `POST`, `PUT`, and `DELETE`? And how can we document our web service so it's easy for anyone to understand how to interact with it?

To solve the first problem, we can install a Visual Studio Code extension named **REST Client**. To solve the second, we can enable **Swagger**, the world's most popular technology for documenting and testing HTTP APIs. But first, let's see what is possible with the Visual Studio Code extension.

Testing HTTP requests with the REST Client extension

REST Client is an extension that allows you to send any type of HTTP request and view the response in Visual Studio Code:

More Information: You can read more about how you can use REST Client at the following link: https://github.com/Huachao/vscode-restclient/blob/master/README.md

1. If you have not already installed REST Client by Huachao Mao (`humao.rest-client`), then install it now.
2. In Visual Studio Code, open the `NorthwindService` project.
3. If the web service is not already running, then start it by entering the following command in **TERMINAL**: `dotnet run`.
4. In the `NorthwindService` folder, create a `RestClientTests` folder.
5. In the `RestClientTests` folder, create a file named `get-customers.http`, and modify its contents to contain an HTTP `GET` request to retrieve all customers, as shown in the following code:

```
GET https://localhost:5001/api/customers/ HTTP/1.1
```

6. Navigate to **View** | **Command Palette**, enter `rest client`, select the command **Rest Client: Send Request**, and press `Enter`, as shown in the following screenshot:

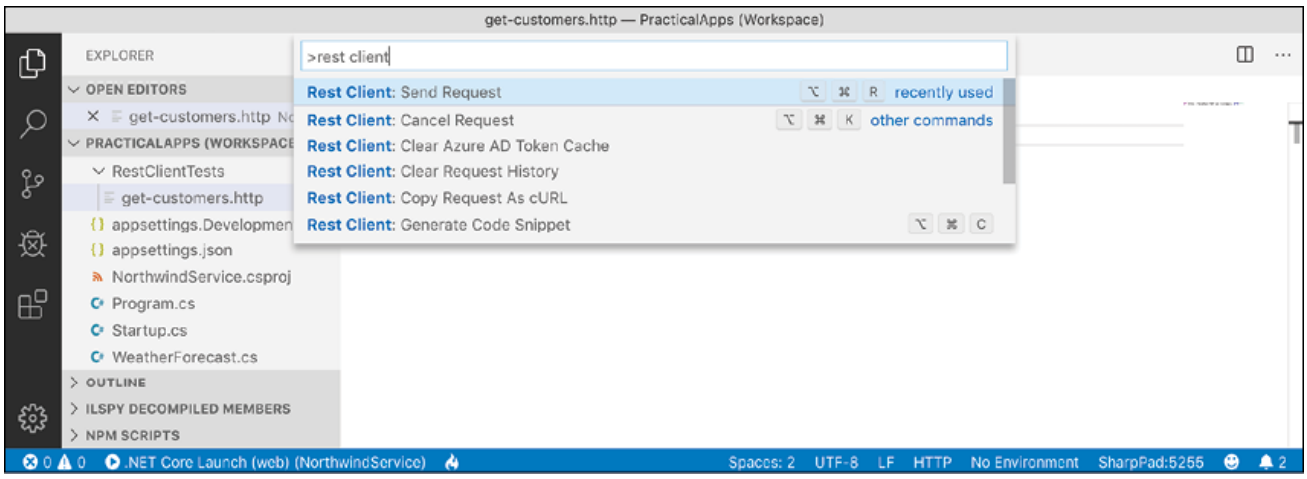


Figure 18.6: Testing HTTP requests with Rest Client

7. Note the **Response** is shown in a new tabbed window pane vertically and that you can rearrange the open tabs to a horizontal layout by dragging and dropping tabs.
8. Enter more HTTP `GET` requests, each separated by three hash symbols, to test getting customers in various countries and getting a single customer using their ID, as shown in the following code:

```
###
GET https://localhost:5001/api/customers/?country=Germany HTTP/1.1
###
GET https://localhost:5001/api/customers/?country=USA HTTP/1.1
Accept: application/xml
###
GET https://localhost:5001/api/customers/ALFKI HTTP/1.1
###
GET https://localhost:5001/api/customers/abcxy HTTP/1.1
```

9. Click inside each statement and press `Ctrl` or `Cmd` + `Alt` + `R` or click the **Send Request** link above each request to send it, as shown in the following screenshot:

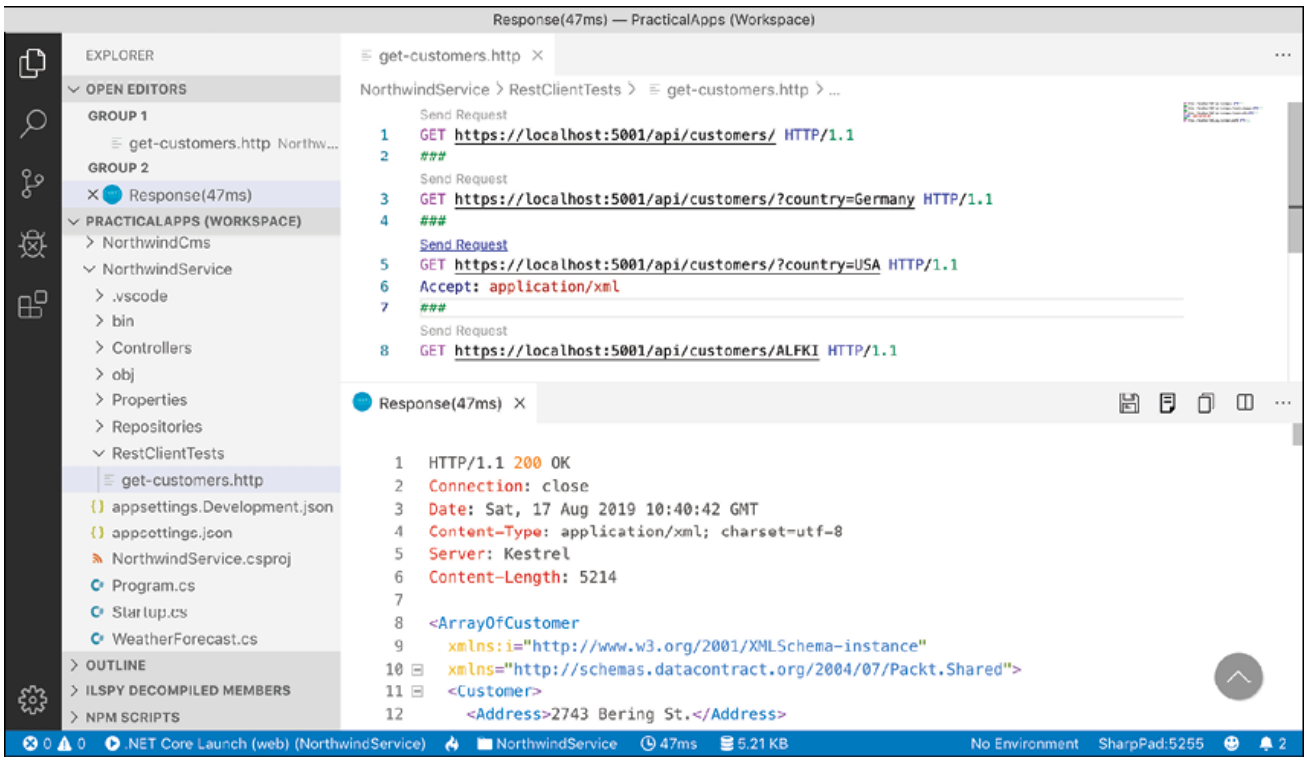


Figure 18.7: Sending a request and getting a response using Rest Client

10. In the `RestClientTests` folder, create a file named `create-customer.http` and modify its contents to define a `POST` request to create a new customer, as shown in the following code:

```
POST https://localhost:5001/api/customers/ HTTP/1.1
Content-Type: application/json
Content-Length: 287
{
  "customerID": "ABCXY",
  "companyName": "ABC Corp",
  "contactName": "John Smith",
  "contactTitle": "Sir",
  "address": "Main Street",
  "city": "New York",
  "region": "NY",
  "postalCode": "90210",
  "country": "USA",
  "phone": "(123) 555-1234",
  "fax": null,
  "orders": null
}
```

Note that REST Client will provide IntelliSense while you type common HTTP requests.

Due to different line endings in different operating systems, the value for the `Content-Length` header will be different on Windows and macOS or Linux. If the value is wrong, then the request will fail.

11. To discover the correct content length, select the body of the request and then look in the status bar for the number of characters, as shown in the following screenshot:

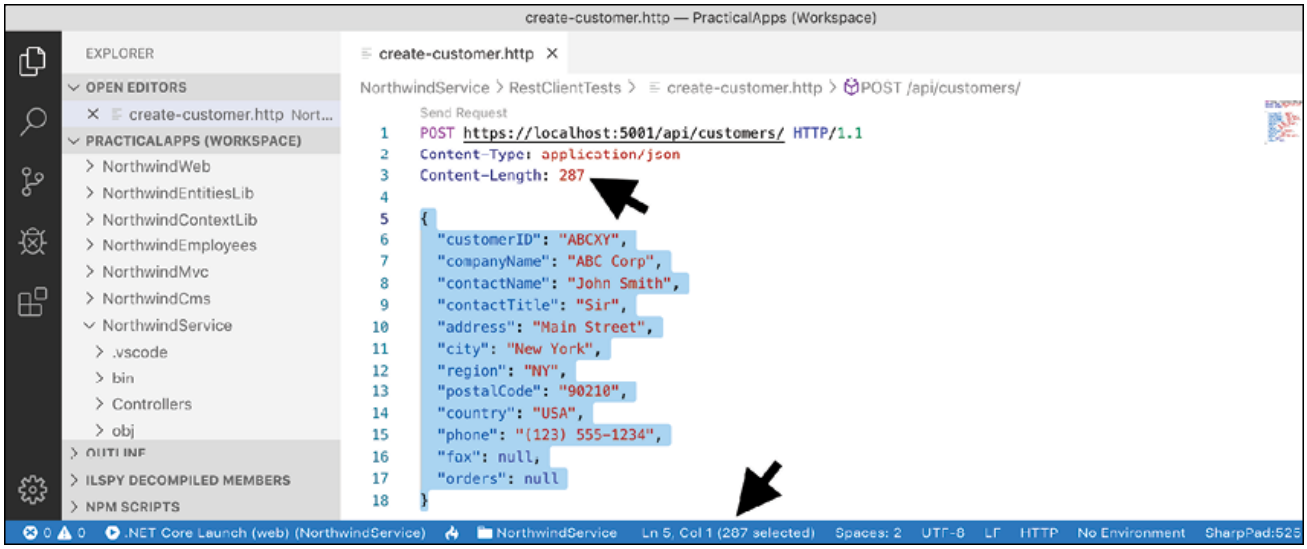


Figure 18.8: Checking the correct content length

12. Send the request and note the response is `201 Created`. Also note the location (that is, URL) of the newly created customer is `https://localhost:5001/api/Customers/abcxy`, and includes the newly created customer in the response body, as shown in the following screenshot:

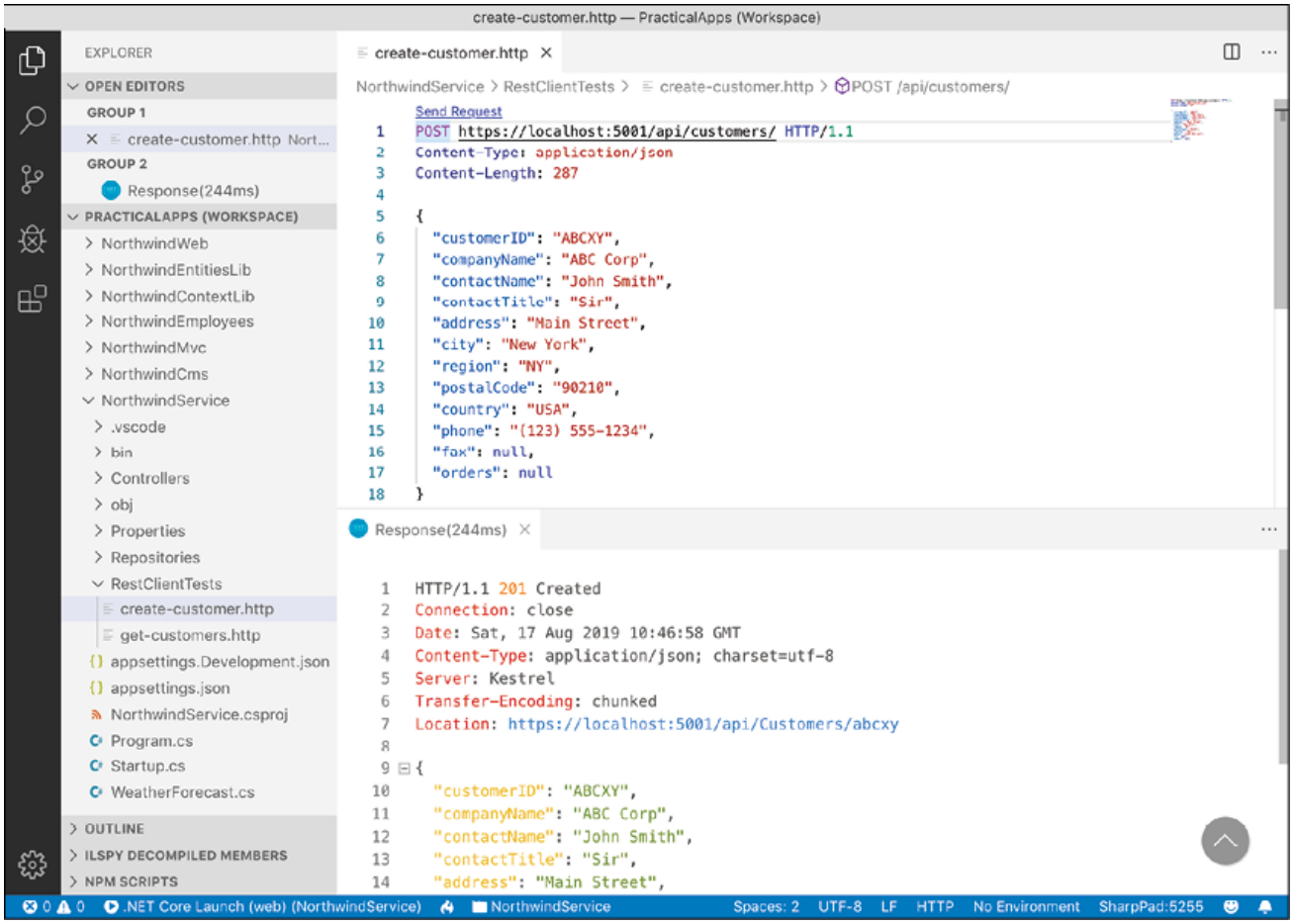


Figure 18.9: Adding a new customer

More Information: Learn more details about HTTP `POST` requests at the following link: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

Now that we've seen a quick and easy way to test our service, which also happens to be a great way to learn HTTP, what about external developers? We want it to be as easy as possible for them to learn and then call our service. For that purpose, we will use Swagger.

Understanding Swagger

The most important part of Swagger is the **OpenAPI Specification**, which defines a REST-style contract for your API, detailing all of its resources and operations in a human- and machine-readable format for easy development, discovery, and integration.

For us, another useful feature is **Swagger UI**, because it automatically generates documentation for your API with built-in visual testing capabilities.

More Information: You can read more about Swagger at the following link: https://swagger.io/

Let's review how Swagger is enabled for our web service using the `Swashbuckle` package:

1. If the web service is running, stop it by pressing `Ctrl` + `C` in **TERMINAL**.
2. Open `NorthwindService.csproj` and note the package reference for `Swashbuckle.AspNetCore`, as shown in the following markup:

```
<ItemGroup>
  <PackageReference Include="Swashbuckle.AspNetCore" Version="5.5.1" />
</ItemGroup>
```

3. Open `Startup.cs` and note the import for Microsoft's OpenAPI models namespace, and add statements to import Swashbuckle's `Swagger` and `SwaggerUI` namespaces, as shown in the following code:

```
using Swashbuckle.AspNetCore.Swagger;
using Swashbuckle.AspNetCore.SwaggerUI;
using Microsoft.OpenApi.Models;
```

4. At the bottom of the `ConfigureServices` method, note the statement to add Swagger support including documentation for the Northwind service, indicating that this is the first version of your service, and change the title, as shown in the following code:

```
// Register the Swagger generator and define a Swagger document
// for Northwind service
services.AddSwaggerGen(options =>
  {
    options.SwaggerDoc(name: "v1", info: new OpenApiInfo
      { Title = "Northwind Service API", Version = "v1" });
  });
```

More Information: You can read about how Swagger can support multiple versions of an API at the following link: https://stackoverflow.com/questions/30789045/leverage-multipleapiversions-in-swagger-with-attribute-versioning/30789944

5. In the `Configure` method, note the statements to use Swagger and Swagger UI, define an endpoint for the OpenAPI specification JSON document, and add code to list the HTTP methods supported by our web service, as shown highlighted in the following code:

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
  c.SwaggerEndpoint("/swagger/v1/swagger.json",
    "Northwind Service API Version 1");
  c.SupportedSubmitMethods(new[] {
    SubmitMethod.Get, SubmitMethod.Post,
    SubmitMethod.Put, SubmitMethod.Delete });
});
```

Testing requests with Swagger UI

You are now ready to test an HTTP request using Swagger:

1. Start the `NorthwindService` ASP.NET Web API service.
2. In Chrome, navigate to `https://localhost:5001/swagger/` and note that both the **Customers** and **WeatherForecast** Web API controllers have been discovered and documented, as well as **Schemas** used by the API.
3. Click `GET /api/Customers/{id}` to expand that endpoint and note the required parameter for the `id` of a customer, as shown in the following screenshot:

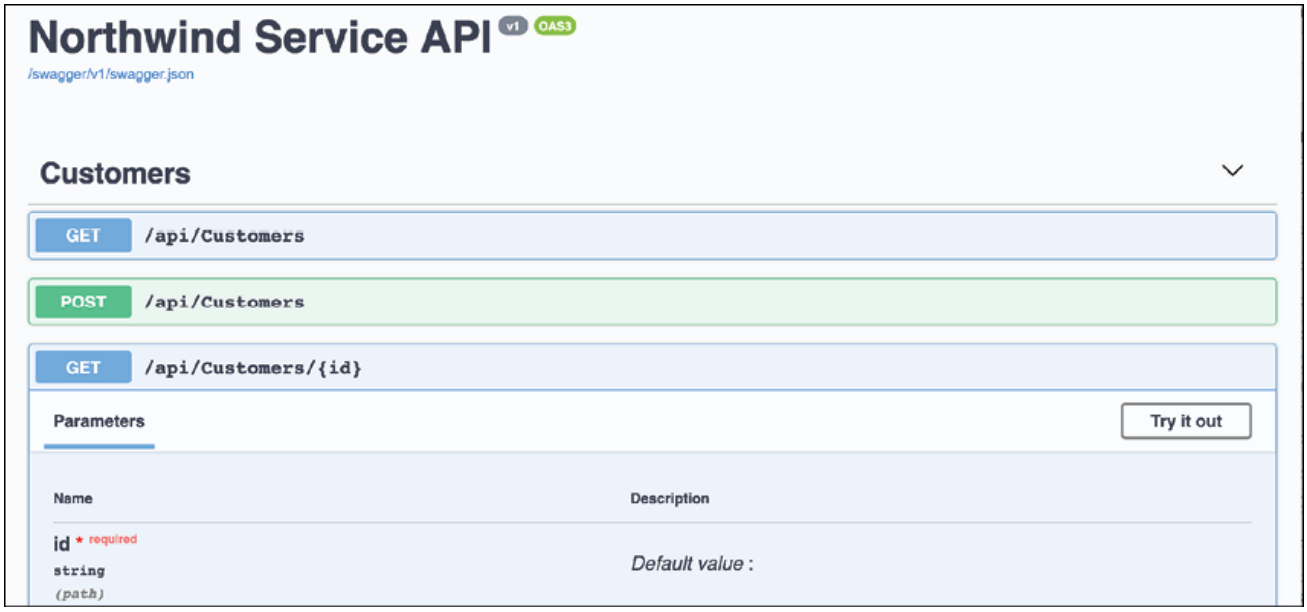


Figure 18.10: Checking the parameters for a GET request in Swagger

4. Click **Try it out**, enter an ID of `ALFKI`, and then click the wide blue **Execute** button, as shown in the following screenshot:

Figure 18.11: Inputting an id before execution

5. Scroll down and note the **Request URL**, **Server response** with **Code**, and **Details** including **Response body** and **Response headers**, as shown in the following screenshot:

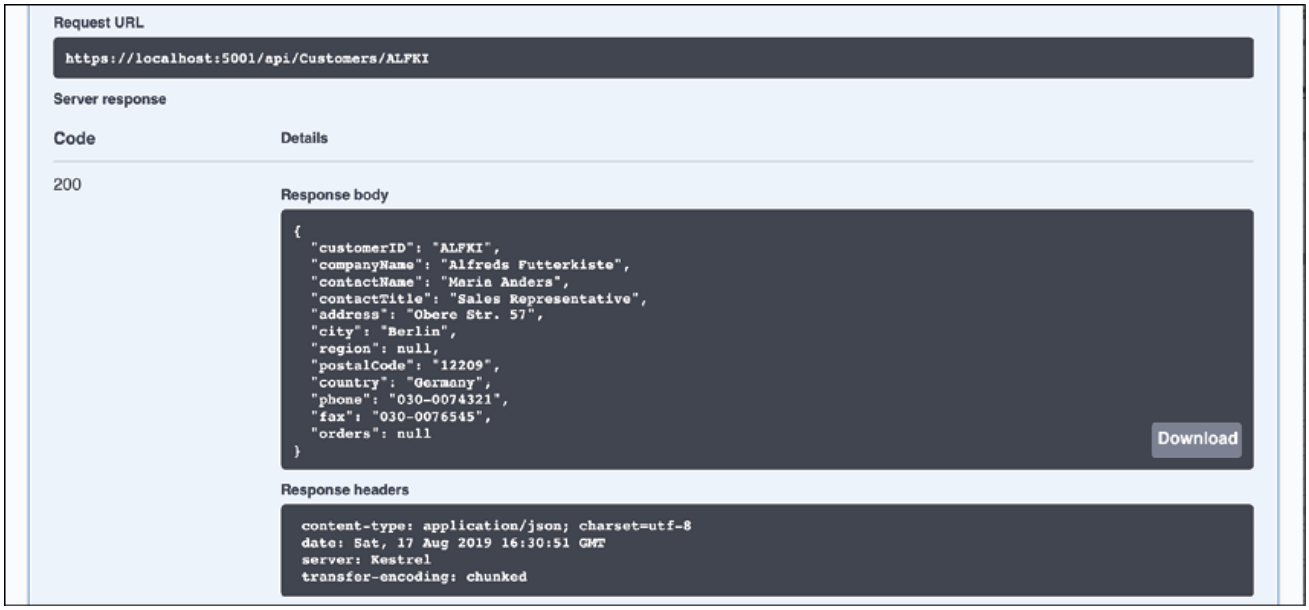


Figure 18.12: Information on ALFKI in a successful Swagger request

6. Scroll back up to the top of the page, click `POST /api/Customers` to expand that section, and then click **Try it out**.
7. Click inside the **Request body** box, and modify the JSON to define a new customer, as shown in the following JSON:

```
{
  "customerID": "SUPER",
  "companyName": "Super Company",
  "contactName": "Rasmus Ibensen",
  "contactTitle": "Sales Leader",
  "address": "Rotterslef 23",
  "city": "Billund",
  "region": null,
  "postalCode": "4371",
  "country": "Denmark",
  "phone": "31 21 43 21",
  "fax": "31 21 43 22",
  "orders": null
}
```

8. Click **Execute**, and note the **Request URL**, **Server response** with **Code**, and **Details** including **Response body** and **Response headers**, as shown in the following screenshot:

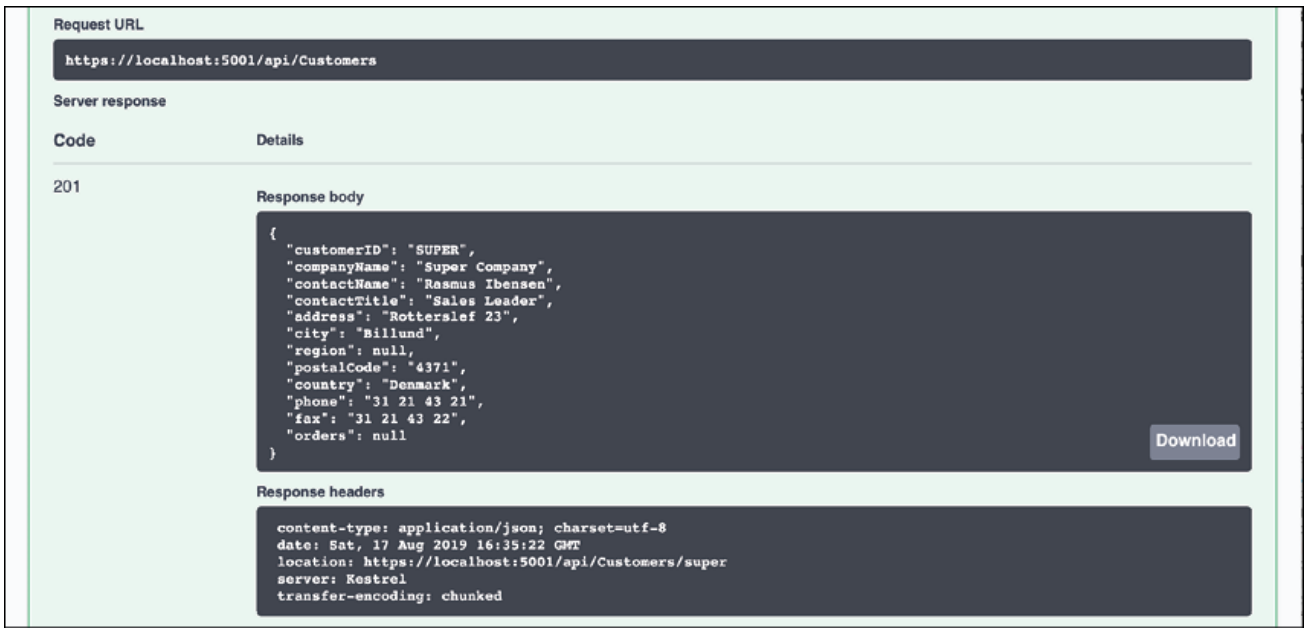


Figure 18.13: Successfully adding a new customer

A response code of `201` means the customer was successfully created.

9. Scroll back up to the top of the page, click `GET /api/Customers`, click **Try it out**, enter `Denmark` for the `country` parameter, and click **Execute**, to confirm that the new customer was added to the database, as shown in the following screenshot:

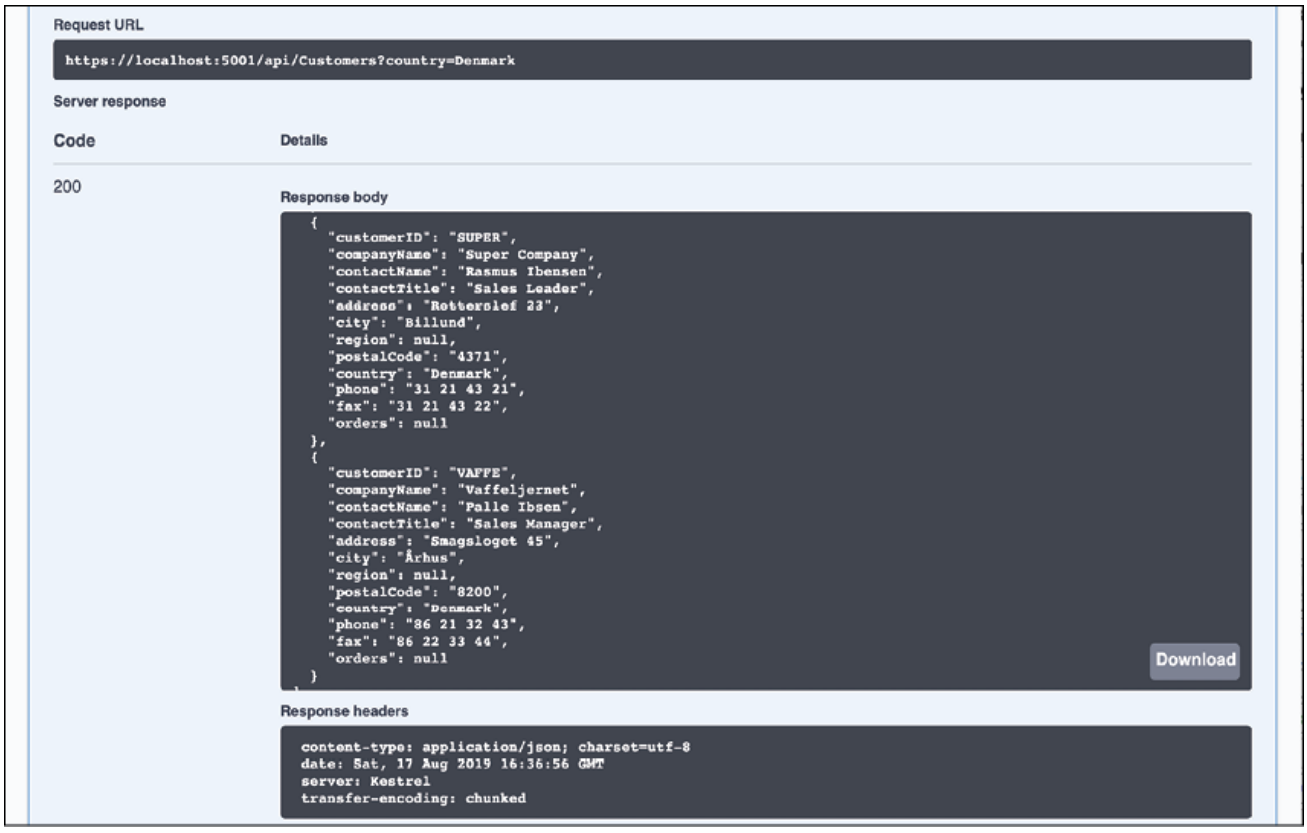


Figure 18.14: Successfully getting customers in Denmark

10. Click `DELETE /api/Customers/{id}`, click **Try it out**, enter `super` for the `id`, click **Execute**, and note that the **Server response Code** is `204`, indicating that it was successfully deleted, as shown in the following screenshot:

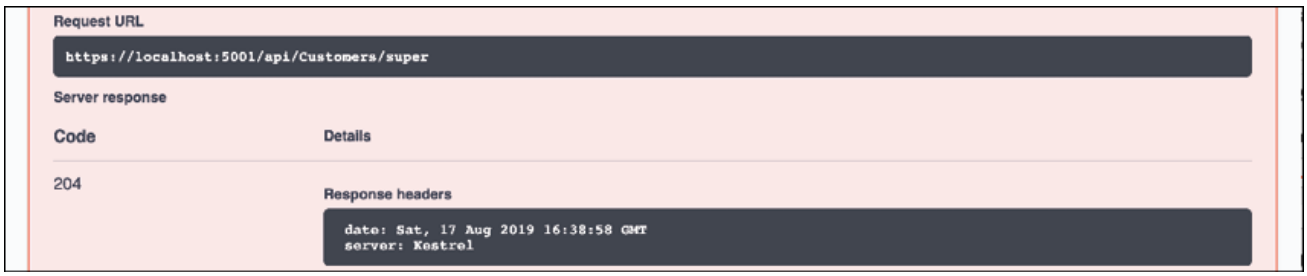


Figure 18.15: Successfully deleting a customer

11. Click **Execute** again, and note that the **Server response Code** is `404`, indicating that the customer does not exist anymore, and the **Response body** contains a problem details JSON document, as shown in the following screenshot:

Figure 18.16: The deleted customer does not exist anymore

12. Enter `bad`, click **Execute** again, and note that the **Server response Code** is `400`, indicating that the customer did exist but failed to delete (in this case, because the web service is simulating this error), and the **Response body** contains a custom problem details JSON document, as shown in the following screenshot:

Figure 18.17: The customer did exist but failed to delete

13. Use the `GET` methods to confirm that the new customer has been deleted from the database (there were originally only two customers in Denmark).

    I will leave testing updates to an existing customer by using `PUT` to the reader.

14. Close Chrome.
15. In **TERMINAL**, press `Ctrl` + `C` to stop the console application and shut down the Kestrel web server that is hosting your service.

More Information: You can read more about the importance of documenting services at the following link: https://idratherbewriting.com/learnapidoc/

You are now ready to build applications that consume your web service.

Consuming services using HTTP clients

Now that we have built and tested our Northwind service, we will learn how to call it from any .NET app using the `HttpClient` class and its new factory.

Understanding HttpClient

The easiest way to consume a web service is to use the `HttpClient` class. However, many people use it wrongly because it implements `IDisposable` and Microsoft's own documentation shows poor usage of it.

Usually, when a type implements `IDisposable`, you should create it inside a `using` statement to ensure that it is disposed of as soon as possible. `HttpClient` is different because it is shared, reentrant, and partially thread-safe.

More Information: It is the `BaseAddress` and `DefaultRequestHeaders` properties that you should treat with caution with multiple threads. You can read more details and recommendations at the following link: https://medium.com/@nuno.caneco/c-httpclient-should-not-be-disposed-or-should-it-45d2a8f568bc

The problem has to do with how the underlying network sockets have to be managed. The bottom line is that you should use a single instance of it for each HTTP endpoint that you consume during the life of your application.

This will allow each `HttpClient` instance to have defaults set that are appropriate for the endpoint it works with, while managing the underlying network sockets efficiently.

More Information: You're using HttpClient wrong and it is destabilizing your software: https://aspnetmonsters.com/2016/08/2016-08-27-httpclientwrong/

Configuring HTTP clients using HttpClientFactory

Microsoft is aware of the issue, and in .NET Core 2.1 they introduced `HttpClientFactory` to encourage best practice; that is the technique we will use.

More Information: You can read more about how to initiate HTTP requests at the following link: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/http-requests

In the following example, we will use the Northwind MVC website as a client to the Northwind Web API service. Since both need to be hosted on a web server simultaneously, we first need to configure them to use different port numbers, as shown in the following list:

- The Northwind Web API service will continue to listen on port `5001` using HTTPS.
- Northwind MVC will listen on ports `5000` using HTTP and `5002` using HTTPS.

Let's configure those ports:

1. In the `NorthwindMvc` project, open `Program.cs`.
2. In the `CreateHostBuilder` method, add an extension method call to `UseUrls` to specify port number `5000` for HTTP and port number `5002` for HTTPS, as shown highlighted in the following code:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
      webBuilder.UseStartup<Startup>();
      webBuilder.UseUrls(
        "http://localhost:5000",
        "https://localhost:5002"
      );
    });
```

3. Open `Startup.cs` and import the `System.Net.Http.Headers` namespace.
4. In the `ConfigureServices` method, add a statement to enable `HttpClientFactory` with a named client to make calls to the Northwind Web API service using HTTPS on port `5001` and request JSON as the default response format, as shown in the following code:

```
services.AddHttpClient(name: "NorthwindService",
  configureClient: options =>
  {
    options.BaseAddress = new Uri("https://localhost:5001/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
      "application/json", 1.0));
  });
```

Getting customers as JSON in the controller

We can now create an MVC controller action method that uses the factory to create an HTTP client, makes a `GET` request for customers, and deserializes the JSON response using convenience extension methods introduced with .NET 5 in the `System.Net.Http.Json` assembly and namespace.

More Information: You can read more about the `HttpClient` extension methods for easily working with JSON at the following link: https://github.com/dotnet/designs/blob/main/accepted/2020/json-http-extensions/json-http-extensions.md

1. Open `Controllers/HomeController.cs` and import the `System.Net.Http` and `System.Net.Http.Json` namespaces.
2. Declare a field to store the HTTP client factory, as shown in the following code:

```
private readonly IHttpClientFactory clientFactory;
```

3. Set the field in the constructor, as shown in the following code:

```
public HomeController(
  ILogger<HomeController> logger,
  Northwind injectedContext,
  IHttpClientFactory httpClientFactory)
{
  _logger = logger;
  db = injectedContext;
  clientFactory = httpClientFactory;
}
```

4. Create a new action method for calling the Northwind service, fetching all customers, and passing them to a view, as shown in the following code:

```
public async Task<IActionResult> Customers(string country)
{
  string uri;
  if (string.IsNullOrEmpty(country))
  {
    ViewData["Title"] = "All Customers Worldwide";
    uri = "api/customers/";
  }
  else
  {
    ViewData["Title"] = $"Customers in {country}";
    uri = $"api/customers/?country={country}";
  }
  var client = clientFactory.CreateClient(
    name: "NorthwindService");
  var request = new HttpRequestMessage(
    method: HttpMethod.Get, requestUri: uri);
  HttpResponseMessage response = await client.SendAsync(request);
  var model = await response.Content
    .ReadFromJsonAsync<IEnumerable<Customer>>();
  return View(model);
}
```

5. In the `Views/Home` folder, create a Razor file named `Customers.cshtml`.
6. Modify the Razor file to render the customers, as shown in the following markup:

```
@model IEnumerable<Packt.Shared.Customer>
<h2>@ViewData["Title"]</h2>
<table class="table">
  <thead>
    <tr>
      <th>Company Name</th>
      <th>Contact Name</th>
      <th>Address</th>
      <th>Phone</th>
    </tr>
  </thead>
  <tbody>
  @foreach (var item in Model)
  {
    <tr>
      <td>
        @Html.DisplayFor(modelItem => item.CompanyName)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.ContactName)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.Address)
        @Html.DisplayFor(modelItem => item.City)
        @Html.DisplayFor(modelItem => item.Region)
        @Html.DisplayFor(modelItem => item.Country)
        @Html.DisplayFor(modelItem => item.PostalCode)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.Phone)
      </td>
    </tr>
  }
```

```
  </tbody>
</table>
```

7. Open `Views/Home/Index.cshtml` and add a form after rendering the visitor count to allow visitors to enter a country and see the customers, as shown in the following markup:

```
<h3>Query customers from a service</h3>
<form asp-action="Customers" method="get">
  <input name="country" placeholder="Enter a country" />
  <input type="submit" />
</form>
```

Enabling Cross-Origin Resource Sharing

It would be useful to explicitly specify the port number for the `NorthwindService` so that it does not conflict with the defaults of `5000` for HTTP and `5002` for HTTPS used by websites like `NorthwindMvc`, and to enable **Cross-Origin Resource Sharing** (**CORS**):

More Information: The default browser same-origin policy prevents code downloaded from one origin from accessing resources downloaded from a different origin to improve security. CORS can be enabled to allow requests in ASP.NET Core at the following link: https://docs.microsoft.com/en-us/aspnet/core/security/cors

1. In the `NorthwindService` project, open `Program.cs`.
2. In the `CreateHostBuilder` method, add an extension method call to `UseUrls` and to specify port number `5001` for HTTPS, as shown highlighted in the following code:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
      webBuilder.UseStartup<Startup>();
      webBuilder.UseUrls("https://localhost:5001");
    });
```

3. Open `Startup.cs`, and add a statement to the top of the `ConfigureServices` method, to add support for CORS, as shown highlighted in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddCors();
```

4. Add a statement to the `Configure` method, before calling `UseEndpoints`, to use CORS and allow `HTTP GET`, `POST`, `PUT`, and `DELETE` requests from any website like Northwind MVC that has an origin of `https://localhost:5002`, as shown in the following code:

```
// must be after UseRouting and before UseEndpoints
app.UseCors(configurePolicy: options =>
{
  options.WithMethods("GET", "POST", "PUT", "DELETE");
  options.WithOrigins(
    "https://localhost:5002" // for MVC client
  );
});
```

5. Navigate to **Terminal** | **New Terminal** and select `NorthwindService`.
6. In **TERMINAL**, start the `NorthwindService` project by entering the command `dotnet run`. Confirm that the web service is listening only on port `5001`, as shown in the following output:

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
```

7. Navigate to **Terminal** | **New Terminal** and select `NorthwindMvc`.
8. In **TERMINAL**, start the `NorthwindMvc` project by entering the command `dotnet run`. Confirm that the website is listening on ports `5000` and `5002`, as shown in the following output:

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5002
```

9. Start Chrome, navigate to `http://localhost:5000/`, and note that it redirects to HTTPS on port `5002` and shows the home page of the Northwind MVC website.
10. In the customer form, enter a country like Germany, UK, or USA, click **Submit**, and note the list of customers, as shown in the following screenshot:

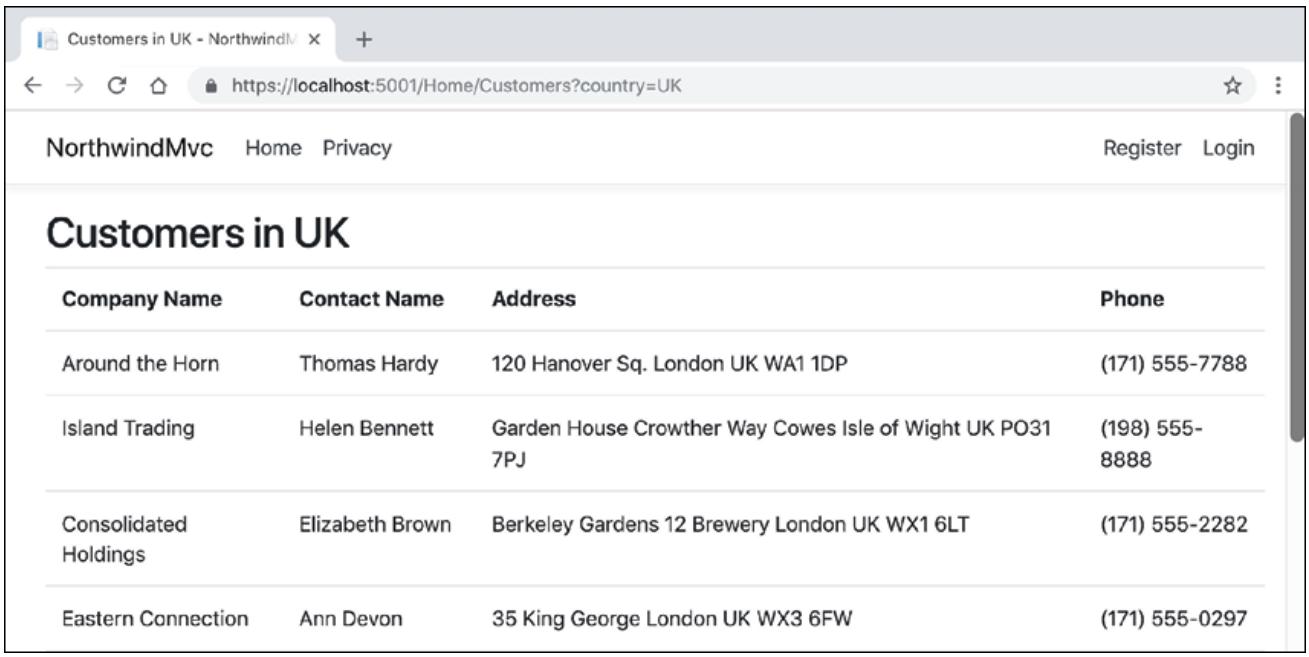


Figure 18.18: Customers in the UK

11. Click back into your browser, clear the country textbox, click **Submit**, and note the worldwide list of customers.

Implementing advanced features

Now that you have seen the fundamentals of building a web service and then calling it from a client, let us look at some more advanced features.

Implementing a Health Check API

There are many paid services that perform site availability tests that are basic pings, some with more advanced analysis of the HTTP response.

ASP.NET Core 2.2 and later makes it easy to implement more detailed website health checks. For example, your website might be live, but is it ready? Can it retrieve data from its database?

1. Open `NorthwindService.csproj`.
2. In the `<ItemGroup>` for the Swashbuckle package, add a project reference to enable Entity Framework Core database health checks, as shown in the following markup:

```
<PackageReference Include=
  "Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore"
  Version="5.0.0" />
```

3. In **TERMINAL**, restore packages and compile the website project, as shown in the following command:

```
dotnet build
```

4. Open `Startup.cs`.
5. At the bottom of the `ConfigureServices` method, add a statement to add health checks, including to the `Northwind` database context, as shown in the following code:

```
services.AddHealthChecks().AddDbContextCheck<Northwind>();
```

   By default, the database context check calls EF Core's `CanConnectAsync` method. You can customize what operation is run using the `AddDbContextCheck` method.

6. In the `Configure` method, before the call to `UseEndpoints`, add a statement to use basic health checks, as shown in the following code:

```
app.UseHealthChecks(path: "/howdoyoufeel");
```

7. Start the web service and navigate to `https://localhost:5001/howdoyoufeel`.
8. Note the website responds with plain text: `Healthy`.

More Information: You can extend the health check response as much as you want. Read more at the following link: https://blogs.msdn.microsoft.com/webdev/2018/08/22/asp-net-core-2-2-0-preview1-healthcheck/

Implementing Open API analyzers and conventions

In this section, you learned how to enable Swagger to document a web service by manually decorating a `Controller` class with attributes.

In ASP.NET Core 2.2 or later, there are API analyzers that reflect over `Controller` classes that have been annotated with the `[ApiController]` attribute to document it automatically. The analyzer assumes some API conventions.

To use it, your project must reference the NuGet package, as shown in the following markup:

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Api.Analyzers"
  Version="3.0.0" PrivateAssets="All" />
```

More Information: At the time of writing, the preceding package was version 3.0.0-preview3-19153-02, but after publishing, it should be a full version 3.0.0 release. You can check the latest version to use at the following link: http://www.nuget.org/packages/Microsoft.AspNetCore.Mvc.Api.Analyzers/

After installing, controllers that have not been properly decorated should have warnings (green squiggles) and warnings when you compile the source code using the `dotnet build` command. For example, the `WeatherForecastController` class.

Automatic code fixes can then add the appropriate `[Produces]` and `[ProducesResponseType]` attributes, although this only currently works in Visual Studio 2019. In Visual Studio Code, you will see warnings about where the analyzer thinks you should add attributes, but you must add them yourself.

Implementing transient fault handling

When a client app or website calls a web service, it could be from across the other side of the world. Network problems between the client and the server could cause issues that are nothing to do with your implementation code. If a client makes a call and it fails, the app should not just give up. If it tries again, the issue may now have resolved. We need a way to handle these temporary faults.

To handle these transient faults, Microsoft recommends that you use the third-party library Polly to implement automatic retries with exponential backoff. You define a policy and the library handles everything else.

More Information: You can read more about how Polly can make your web services more reliable at the following link: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/implement-http-call-retries-exponential-backoff-polly

Understanding endpoint routing

In earlier versions of ASP.NET Core, the routing system and the extendable middleware system did not always work easily together, for example, if you wanted to implement a policy like CORS in both middleware and MVC, so Microsoft has invested in improving routing with a system named **Endpoint Routing** introduced with ASP.NET Core 2.2.

Good Practice: Microsoft recommends every older ASP.NET Core project migrates to endpoint routing if possible.

Endpoint routing is designed to enable better interoperability between frameworks that need routing, like Razor Pages, MVC, or Web APIs, and middleware that need to understand how routing affects them, like localization, authorization, CORS, and so on.

More Information: You can read more about the design decisions around endpoint routing at the following link: https://devblogs.microsoft.com/aspnet/asp-net-core-2-2-0-preview1-endpoint-routing/

It gets its name because it represents the route table as a compiled tree of endpoints that can be walked efficiently by the routing system. One of the biggest improvements is the performance of routing and action method selection.

It is on by default with ASP.NET Core 2.2 or later if compatibility is set to 2.2 or later. Traditional routes registered using the `MapRoute` method or with attributes are mapped to the new system.

The new routing system includes a link generation service registered as a dependency service that does not need an `HttpContext`.

Configuring endpoint routing

Endpoint routing requires a pair of calls to `app.UseRouting()` and `app.UseEndpoints()`.

- `app.UseRouting()` marks the pipeline position where a routing decision is made.
- `app.UseEndpoints()` marks the pipeline position where the selected endpoint is executed.

Middleware like localization that runs in between these can see the selected endpoint and can switch to a different endpoint if necessary.

Endpoint routing uses the same route template syntax that has been used in ASP.NET MVC since 2010 and the `[Route]` attribute introduced with ASP.NET MVC 5 in 2013. Migration often only requires changes to the `Startup` configuration.

MVC controllers, Razor Pages, and frameworks like SignalR used to be enabled by a call to `UseMvc()` or similar methods, but they are now added inside `UseEndpoints()` because they are all integrated into the same routing system along with middleware.

Let's define some middleware that can output information about endpoints:

1. Open `Startup.cs` and import namespaces for working with endpoint routing, as shown in the following code:

```
using Microsoft.AspNetCore.Http;    // GetEndpoint() extension method
using Microsoft.AspNetCore.Routing; // RouteEndpoint
```

2. In the `Configure` method, add a statement before `UseEndpoints` to define a lambda statement to output information about the selected endpoint during every request, as shown in the following code:

```
app.Use(next => (context) =>
{
  var endpoint = context.GetEndpoint();
  if (endpoint != null)
  {
    WriteLine("*** Name: {0}; Route: {1}; Metadata: {2}",
      arg0: endpoint.DisplayName,
      arg1: (endpoint as RouteEndpoint)?.RoutePattern,
      arg2: string.Join(", ", endpoint.Metadata));
  }
  // pass context to next middleware in pipeline
  return next(context);
});
```

   While reviewing the preceding code, note the following:

   - The `Use` method requires an instance of `RequestDelegate` or a lambda statement equivalent.
   - `RequestDelegate` has a single `HttpContext` parameter that wraps all information about the current HTTP request (and its matching response).
   - Importing the `Microsoft.AspNetCore.Http` namespace adds the `GetEndpoint` extension method to the `HttpContext` instance.
3. Start the web service.
4. In Chrome, navigate to `https://localhost:5001/weatherforecast`.
5. In **TERMINAL**, note the result, as shown in the following output:

```
Request starting HTTP/1.1 GET https://localhost:5001/weatherforecast
*** Name: NorthwindService.Controllers.WeatherForecastController.Get (NorthwindService); Route: Microsoft.AspNetCore.Routing.Patterns.RoutePatte
```

6. Close Chrome and stop the web service.

More Information: You can read more about endpoint routing at the following link: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing

Endpoint routing replaces the `IRouter`-based routing used in ASP.NET Core 2.1 and earlier.

More Information: If you need to work with ASP.NET Core 2.1 or earlier, then you can read about the old routing system at the following link: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-2.1

Adding security HTTP headers

ASP.NET Core has built-in support for common security HTTP headers like HSTS. But there are many more HTTP headers that you should consider implementing.

The easiest way to add these headers is using a middleware class:

1. In the `NorthwindService` folder, create a file named `SecurityHeadersMiddleware.cs` and modify its statements, as shown in the following code:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Primitives;
namespace Packt.Shared
{
  public class SecurityHeaders
  {
    private readonly RequestDelegate next;
    public SecurityHeaders(RequestDelegate next)
    {
      this.next = next;
    }
    public Task Invoke(HttpContext context)
    {
      // add any HTTP response headers you want here
      context.Response.Headers.Add(
        "super-secure", new StringValues("enable"));
      return next(context);
    }
  }
}
```

2. Open the `Startup.cs` file and add a statement to register the middleware before the call to `UseEndpoints`, as shown in the following code:

```
app.UseMiddleware<SecurityHeaders>();
```

3. Start the web service.
4. Start Chrome and show **Developer Tools** and its **Network** tab to record requests and responses.
5. Navigate to `https://localhost:5001/weatherforecast`.
6. Note the custom HTTP response header that we added named `super-secure`, as shown in the following screenshot:

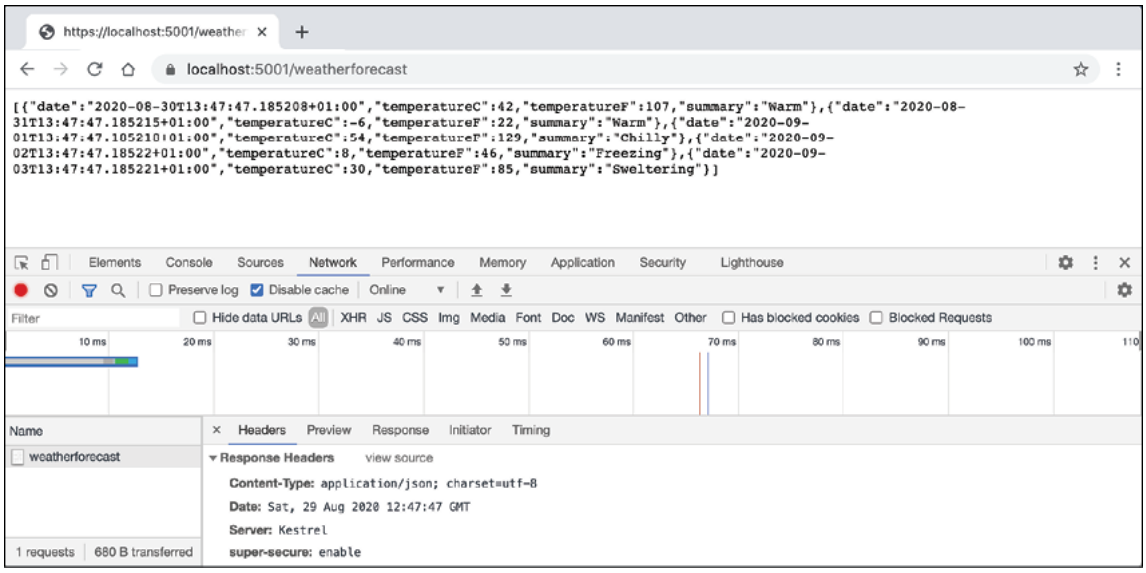


Figure 18.19: Adding a custom HTTP header named super-secure

More Information: You can read more about common HTTP security headers that you might want to add at the following link: https://www.meziantou.net/security-headers-in-asp-net-core.htm

Securing web services

An improvement for web services in .NET 5 includes a simple extension method for enabling anonymous HTTP calls when using endpoint routing, as shown highlighted in the following code:

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

Understanding other communication technologies

The ASP.NET Core Web API is not the only Microsoft technology for implementing services or communicating between components of a distributed application. Although we will not cover these technologies in detail, you should be aware of what they can do and when they should be used.

Understanding Windows Communication Foundation (WCF)

In 2006, Microsoft released .NET Framework 3.0 with some major frameworks, one of which was **Windows Communication Foundation** (**WCF**). It abstracted the business logic implementation of a service from the technology used to communicate with it. It heavily used XML configuration to declaratively define endpoints, including their address, binding, and contract (known as the ABCs of endpoints). Once you have understood how to do this, it is a powerful yet flexible technology.

Microsoft has decided not to officially port WCF to .NET Core, but there is a community-owned OSS project named **Core WCF** managed by the .NET Foundation. If you need to migrate an existing service from .NET Framework to .NET Core, or build a client to a WCF service, then you could use Core WCF. Be aware that it can never be a full port since parts of WCF are Windows-specific.

More Information: You can read more and download the Core WCF repository from the following link: https://github.com/CoreWCF/CoreWCF

Technologies like WCF allow for the building of distributed applications. A client application can make **remote procedure calls** (**RPC**) to a server application. Instead of using a port of WCF to do this, we could use an alternative RPC technology.

Understanding gRPC

gRPC is a modern open source high-performance RPC framework that can run in any environment.

More Information: You can read about gRPC at the following link: https://grpc.io

Like WCF, gRPC uses contract-first API development that supports language-agnostic implementations. You write the contracts using `.proto` files with their own language syntax and tools to convert them into various languages like C#. It minimizes network usage by using `Protobuf` binary serialization.

More Information: Microsoft officially supports gRPC with ASP.NET Core. You can learn how to use gRPC with ASP.NET Core at the following link: https://docs.microsoft.com/en-us/aspnet/core/grpc/aspnetcore

Explore topics

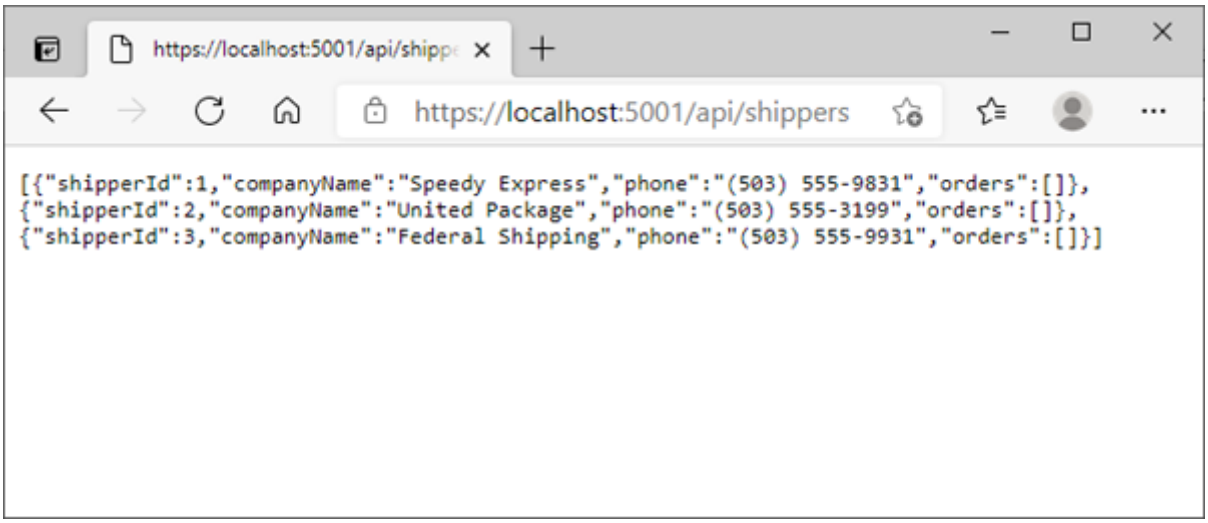
Use the following links to read more about this section's topics:

- **Create web APIs with ASP.NET Core**: https://docs.microsoft.com/en-us/aspnet/core/web-api/
- **Swagger Tools**: https://swagger.io/tools/
- **Swashbuckle for ASP.NET Core**: https://github.com/domaindrivendev/Swashbuckle.AspNetCore
- **Health checks in ASP.NET Core**: https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/health-checks
- **Use HttpClientFactory to implement resilient HTTP requests**: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/use-httpclientfactory-to-implement-resilient-http-requests

Exercises

Exercise 9.1.

- add a Shippers service to your NorthwindService application that provides five methods that perform CRUD operations on the Shippers table:
  - two GET methods (all shippers or one shipper)
  - POST (create)
  - PUT (update)
  - DELETE.
- Test your Shippers service using the REST Client extension and Swagger.



Exercise 9.2.

- add an MVC Page to your NorthwindMvc application that displays the contents of the Shippers table by consuming the above web service.