Bonus Content

We finish this book with three online chapters on building and consuming specialized services, using .NET MAUI, and protecting your data and applications. The book closes with the answers to the Test your knowledge sections from the end of each chapter. This PDF contains all three online chapters and the appendix.

Chapter 18, *Building and Consuming Specialized Services*, introduces you to building services using gRPC, implementing real-time communications between server and client using SignalR, exposing an EF Core model using OData, and hosting functions in the cloud that respond to triggers using Azure Functions.

Chapter 19, *Building Mobile and Desktop Apps Using .NET MAUI*, introduces you to building cross-platform mobile and desktop apps for Android, iOS, macOS, and Windows. You will learn the basics of XAML, which can be used to define the user interface for a graphical app.

Chapter 20, *Protecting Your Data and Applications*, is about protecting your data from being viewed by malicious users using encryption and from being manipulated or corrupted using hashing and signing. You will also learn about authentication and authorization to protect applications from unauthorized users.

Appendix, *Answers to the Test Your Knowledge Questions*, has the answers to the test questions at the end of each chapter.

Table of Contents

18

Building and Consuming Specialized Services

In this chapter, you will be introduced to the fundamentals of several service technologies that are used in more specialized scenarios than a general-purpose web service. Once you understand the concepts and benefits of each, then you can dig deeper into the ones that interest you most.

This chapter will cover the following topics:

- Understanding specialized service technologies
- Exposing data as a web service using OData
- Exposing data as a service using GraphQL
- Implementing services using gRPC
- Implementing real-time communication using SignalR
- Implementing serverless services using Azure Functions
- Understanding identity services
- Summary of choices for specialized services

Understanding specialized service technologies

ASP.NET Core Web API is often the best choice for implementing a general-purpose web service, but it is not the only 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 new frameworks, one of which was **Windows Communication Foundation** (**WCF**). It abstracted the business logic implementation of a service from the communication technology infrastructure so that you could easily switch to an alternative in the future or even have multiple mechanisms to communicate with the service.

WCF heavily uses XML configuration to declaratively define endpoints, including their address, binding, and contract. This is known as the ABC of WCF endpoints. Once you understand how to do this, WCF is a powerful yet flexible technology.

Microsoft decided not to officially port WCF to modern .NET, 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 modern .NET 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.

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

Exposing data as a web service using OData

One of the most common uses of a web service is to expose a database to clients that do not understand how to work with a native database. Another common use is to provide a simplified or abstracted API that exposes only an authenticated interface to a subset of the data.

In *Chapter 10*, *Working with Data Using Entity Framework Core*, you learned how to create an EF Core model to expose a database to any .NET app or website. But what about non-.NET apps and websites? I know it's crazy to imagine, but not every developer uses .NET!

Luckily, all development platforms support HTTP so that they can call web services, and ASP.NET Core has a package for making that easy and powerful using a standard named OData.

Understanding OData

OData (**Open Data Protocol**) is an ISO/IEC approved OASIS standard that defines a set of best practices for building and consuming RESTful APIs. Microsoft created it in 2007 and released versions 1.0, 2.0, and 3.0 under their Microsoft Open Specification Promise. Version 4.0 was then standardized at OASIS and released in 2014.

ASP.NET Core OData implements OData version 4.0.

OData is based on HTTP and has multiple endpoints to support multiple versions and entity sets.

Unlike traditional Web APIs where the service defines all the methods and what gets returned, OData uses query strings for its queries that enable the client to have more control over what is returned and minimizes round trips. For example, a client might only need two fields of data, `ProductName` and `Cost`, and the related `Supplier` object, and only for products where `ProductName` contains the word `burger` and the cost is less than `4.95`, as shown in the following request:

```
GET https://example.com/v1/products?$filter=contains(ProductName, 'burger') and
Cost lt 4.95&$orderby=Country,Cost&$select=ProductName,Cost&$expand=Supplier
```

Building a web service that supports OData

There is no `dotnet new` project template for ASP.NET Core OData, and it uses controller classes, so we will use the Web API project template and then add package references for OData support:

1. Use your preferred code editor to add a new project, as defined in the following list:
    1. Project template: **ASP.NET Core Web API** / `webapi`
    2. Workspace/solution file and folder: `PracticalApps`
    3. Project file and folder: `Northwind.OData`
    4. Other Visual Studio options: **Authentication Type**: None, **Configure for HTTPS**: selected, **Enable Docker**: cleared, **Enable OpenAPI support**: selected. In Visual Studio Code, select `Northwind.OData` as the active OmniSharp project.
2. Add a package reference for ASP.NET Core OData, as shown highlighted in the following markup:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.OData"
    Version="8.0.1" />
  <PackageReference Include="Swashbuckle.AspNetCore"
    Version="6.1.4" />
</ItemGroup>
```

Good Practice: The version numbers of the NuGet packages above are likely to increase after the book is published. As a general guide, you will want to use the latest package version.

3. Add a project reference to the Northwind database context project for either SQLite or SQL Server, as shown in the following markup:

```
<ItemGroup>
  <!-- change Sqlite to SqlServer if you prefer -->
```

```
  <ProjectReference Include=
"..\Northwind.Common.DataContext.Sqlite\Northwind.Common.DataContext.
Sqlite.csproj" />
</ItemGroup>
```

4. In the `Northwind.OData` folder, delete `WeatherForecast.cs`.
5. In the `Controllers` folder, delete `WeatherForecastController.cs`.
6. In `Program.cs`, configure `UseUrls` to specify port `5004` for `HTTPS`, as shown highlighted in the following code:

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

builder.WebHost.UseUrls("https://localhost:5004");
```

7. Build the `Northwind.OData` project.

Defining OData models for EF Core models

The first task is to define what we want to expose as OData models in the web service. You have complete control, so if you have an existing EF Core model, as we do for Northwind, you do not have to expose all of it. You do not even have to use EF Core models. The data source can be anything, although in this book we will only look at using it with EF Core because that is the most common use for .NET developers.

Let's define two OData models: one to expose the Northwind product catalog, i.e. the categories and products tables; and one to expose the customers, their orders, and related tables:

1. In `Program.cs`, import namespaces for working with OData and our EF Core model for the Northwind database, as shown in the following code:

```
using Microsoft.AspNetCore.OData; // AddOData extension method
using Microsoft.OData.Edm; // IEdmModel
using Microsoft.OData.ModelBuilder; // ODataConventionModelBuilder
using Packt.Shared; // NorthwindContext and entity models
```

2. At the bottom of `Program.cs`, add a method to define and return an OData model for the Northwind catalog that will only expose the entity sets, i.e. tables for `Categories`, `Products`, and `Suppliers`, as shown in the following code:

```
IEdmModel GetEdmModelForCatalog()
{
  ODataConventionModelBuilder builder = new();
  builder.EntitySet<Category>("Categories");
  builder.EntitySet<Product>("Products");
  builder.EntitySet<Supplier>("Suppliers");
  return builder.GetEdmModel();
}
```

3. Add a method to define an OData model for the Northwind customer orders, and note that the same entity set can appear in multiple OData models like `Products` does, as shown in the following code:

```
IEdmModel GetEdmModelForOrderSystem()
{
  ODataConventionModelBuilder builder = new();
  builder.EntitySet<Customer>("Customers");
  builder.EntitySet<Order>("Orders");
  builder.EntitySet<Employee>("Employees");
  builder.EntitySet<Product>("Products");
  builder.EntitySet<Shipper>("Shippers");
  return builder.GetEdmModel();
}
```

4. In the services configuration section, after the call to `AddControllers`, chain a call to the `AddOData` extension method to define two OData models and enable features like projection, filtering, and sorting, as shown in the following code:

```
builder.Services.AddControllers()
  .AddOData(options => options
    // register OData models including multiple versions
    .AddRouteComponents(routePrefix: "catalog",
      model: GetEdmModelForCatalog())
    .AddRouteComponents(routePrefix: "ordersystem",
      model: GetEdmModelForOrderSystem())

    // enable query options
    .Select() // enable $select for projection
    .Expand() // enable $expand to navigate to related entities
    .Filter() // enable $filter
    .OrderBy() // enable $orderby
    .SetMaxTop(100) // enable $top
    .Count() // enable $count
  );
```

5. Add statements before the call to `AddControllers`, to register the `Northwind` database context, as shown in the following code:

```
builder.Services.AddNorthwindContext();
```

6. In the `Properties` folder, open `launchSettings.json`.
7. In the `Northwind.OData` profile, modify the `applicationUrl` to use port `5004`, as shown in the following markup:

```
"applicationUrl": "https://localhost:5004",
```

Testing the OData models

Now we can check that the OData models have been defined correctly:

1. Start the `Northwind.OData` web service.
2. Start Chrome.
3. Navigate to `https://localhost:5004/swagger` and note the `Northwind.OData v1` service is documented.
4. In the **Metadata** section, click **GET /catalog**, click **Try it out**, click **Execute**, and note the response body that shows the names and URLs of the three entity sets in the catalog OData model, as shown in the following output:

```
{
  "@odata.context": "https://localhost:5004/catalog/$metadata",
  "value": [
    {
      "name": "Categories",
      "kind": "EntitySet",
      "url": "Categories"
    },
    {
      "name": "Products",
      "kind": "EntitySet",
      "url": "Products"
    },
    {
      "name": "Suppliers",
      "kind": "EntitySet",
      "url": "Suppliers"
    }
  ]
}
```

5. Click **GET /catalog** to collapse that section.
6. Click **GET /catalog/$metadata**, click **Try it out**, click **Execute**, and note the model describes entities like `Category` in detail with properties and keys, including navigation properties for the products in each category, as shown in *Figure 18.1*:

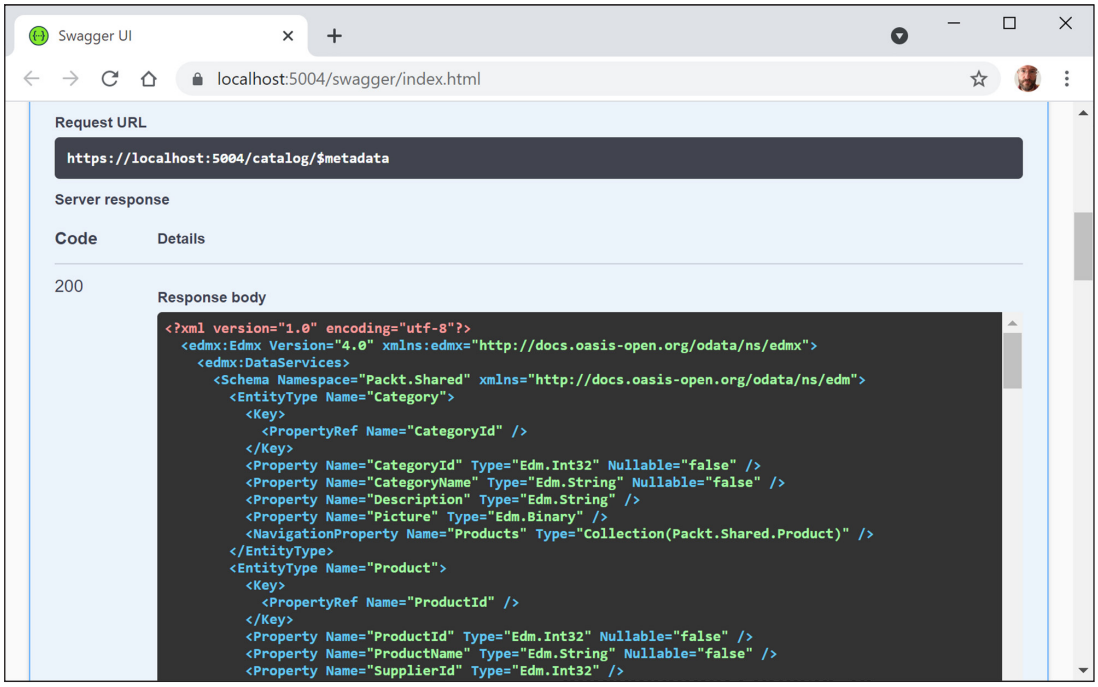


Figure 18.1: OData model metadata for the Northwind catalog

7. Click **GET /catalog/$metadata** to collapse that section.
8. Close Chrome and shut down the web server.

Creating and testing OData controllers

Next, we must create OData controllers, one for each type of entity, to retrieve data:

1. In the `Controllers` folder, add an empty controller class named `CategoriesController`.
2. Modify its contents to inherit from `ODataController`, get an instance of the Northwind database context using constructor parameter injection, and define two `Get` methods to retrieve all categories or one category using a unique key, as shown in the following code:

```
using Microsoft.AspNetCore.Mvc; // IActionResult
using Microsoft.AspNetCore.OData.Query; // [EnableQuery]
using Microsoft.AspNetCore.OData.Routing.Controllers; // ODataController
using Packt.Shared; // NorthwindContext

namespace Northwind.OData.Controllers;
```

```
public class CategoriesController : ODataController
{
  private readonly NorthwindContext db;

  public CategoriesController(NorthwindContext db)
  {
    this.db = db;
  }

  [EnableQuery]
  public IActionResult Get()
  {
    return Ok(db.Categories);
  }

  [EnableQuery]
  public IActionResult Get(int key)
  {
    return Ok(db.Categories.Find(key));
  }
}
```

3. Repeat the above step for `Products` and `Suppliers`. (I will leave it to you to do the same for the other entities to enable the order system OData model if you choose. Note that `CustomerId` is a `string` instead of an integer.)
4. Start the `Northwind.OData` web service.
5. Start Chrome.
6. Navigate to `https://localhost:5004/swagger` and note the **Categories**, **Products**, and **Suppliers** entity sets are now usable because you created OData controllers for them.
7. Click **GET /catalog/Categories**, click **Try it out**, click **Execute**, and note the response body that shows a JSON document containing all categories in the entity set, as partially shown in the following output:

```
{
  "@odata.context": "https://localhost:5004/catalog/$metadata#Categories",
  "value": [
    {
      "CategoryId": 1,
      "CategoryName": "Beverages",
      "Description": "Soft drinks, coffees, teas, beers, and ales",
      "Picture": null
    },
    {
      "CategoryId": 2,
```

```
      "CategoryName": "Condiments",
      "Description": "Sweet and savory sauces, relishes, spreads, and
seasonings",
      "Picture": null
    },
    ...
  ]
}
```

8. Close Chrome and shut down the web server.

Testing OData controllers using REST Client

Using the Swagger user interface to test OData controllers can quickly get clumsy. A better tool is the Visual Studio Code extension named REST Client:

1. If you have not already installed REST Client by Huachao Mao (`humao.rest-client`), then install it in Visual Studio Code now.
2. In your preferred code editor, start the `Northwind.OData` project web service.
3. In Visual Studio Code, in the `PracticalApps` folder, if it does not already exist, create a `RestClientTests` folder, and then open the folder.
4. In the `RestClientTests` folder, create a file named `odata-catalog.http` and modify its contents to contain a request to get all categories, as shown in the following code:

```
GET https://localhost:5004/catalog/categories/ HTTP/1.1
```

5. Click **Send Request**, and note the response is the same as what was returned by Swagger, a JSON document containing all categories.
6. In `odata-catalog.http`, add more requests separated by `###`, as shown in the following table:

Request	Response
`https://localhost:5004/catalog/categories(3)`	`{ "@odata.context": "https://localhost:5004/catalog/$metadata#Categories/$entity", "CategoryId":3, "CategoryName":"Confections", "Description":"Desserts, candies, and sweet breads", "Picture":null }`
`https://localhost:5004/catalog/categories/3`	Same as above.
`https://localhost:5004/catalog/categories/$count`	`8`

`https://localhost:5004/catalog/products`	JSON document containing all products.
`https://localhost:5004/catalog/products/$count`	`77`

`https://localhost:5004/catalog/products(2)`	{
	"@odata.context": "https://localhost:5004/catalog/$metadata#Products/$entity",
	"ProductId": 2,
	"ProductName": "Chang",
	"SupplierId": 1,
	"CategoryId": 1,
	"QuantityPerUnit": "24 - 12 oz bottles",
	"UnitPrice": 19,
	"UnitsInStock": 17,
	"UnitsOnOrder": 40,
	"ReorderLevel": 25,
	"Discontinued": false
	}

`https://localhost:5004/catalog/suppliers`	JSON document containing all suppliers.
`https://localhost:5004/catalog/suppliers/$count`	`29`

Querying OData models

To execute arbitrary queries against an OData model, we earlier enabled selecting, filtering, and ordering. One of the benefits of OData is that it defines standard query options, as shown in the following table:

Option	Description	Example
`$select`	Selects properties for each entity.	`$select=CategoryId,CategoryName`
`$expand`	Selects related entities via navigation properties.	`$expand=Products`
`$filter`	The expression is evaluated for each resource, and only entities where the expression is true are included in the response.	`$filter=startswith(ProductName,'ch') or (UnitPrice gt 50)`
`$orderby`	Sorts the entities by the properties listed in ascending (default) or descending order.	`$orderby=UnitPrice desc,ProductName`
`$skip` `$top`	Skips the specified number of items. Takes the specified number of items.	`$skip=40&$take=10`

For performance reasons, batching with `$skip` and `$top` is disabled by default.

Understanding OData operators

OData has operators for use with the `$filter` option, as shown in the following table:

Operator	Description
`eq`	Equals.
`ne`	Not equals.
`lt`	Less than.
`gt`	Greater than.
`le`	Less than or equal to.
`ge`	Greater than or equal to.
`and`	And.
`or`	Or.
`not`	Not.
`add`	Arithmetic addition for numbers and date/time values.
`sub`	Arithmetic subtraction for numbers and date/time values.
`mul`	Arithmetic multiplication for numbers.
`div`	Arithmetic division for numbers.
`mod`	Arithmetic modulus division for numbers.

Understanding OData functions

OData has functions for use with the `$filter` option, as shown in the following table:

Operator	Description
`startswith(property, 'value')`	Text values that start with the specified value.
`endswith(property, 'value')`	Text values that end with the specified value.
`concat(property, 'value')`	Concatenate two text values.
`contains(property, 'value')`	Text values that contain the specified value.
`indexof(property, 'value')`	Returns the position of a text value.
`length(property)`	Returns the length of a text value.
`substring`	Extracts a substring from a text value.
`tolower`	Converts to lowercase.
`toupper`	Converts to uppercase.
`trim`	Trims whitespace before and after text value.
`now`	The current date and time.
`date, day, month, year`	Extracts date components.
`time, hour, minute, second`	Extracts time components.

Exploring OData queries

Let's experiment with some OData queries:

1. In the `RestClientTests` folder, create a file named `odata-catalog-queries.http`, and modify its contents to contain a request to get all categories, as shown in the following code:

```
GET https://localhost:5004/catalog/categories/
  ?$select=CategoryId,CategoryName
```

2. Click **Send Request** and note the response – a JSON document containing all categories but only the `CategoryId` and `CategoryName` properties.
3. In `odata-catalog-queries.http`, add a request to get products with names that start with "Ch," like Chai and Chef Anton's Gumbo Mix, or have a unit price of more than 50, like Mishi Kobe Niku or Sir Rodney's Marmalade, as shown in the following request:

```
GET https://localhost:5004/catalog/products/
  ?$filter=startswith(ProductName,'Ch') or (UnitPrice gt 50)
```

4. In `odata-catalog-queries.http`, add a request to get products sorted by price, with the most expensive at the top, and then sorted by product name, and only include the ID, name, and price properties, as shown in the following request:

```
GET https://localhost:5004/catalog/products/
  ?$orderby=UnitPrice desc,ProductName
  &$select=ProductId,ProductName,UnitPrice
```

5. In `odata-catalog-queries.http`, add a request to get categories and their related products, as shown in the following request:

```
GET https://localhost:5004/catalog/categories/
  ?$select=CategoryId,CategoryName
  &$expand=Products
```

Logging OData requests

How does OData querying work? Let's find out by adding logging to the Northwind database context to see the actual SQL statements that are executed:

1. In the `Northwind.Common.DataContext.Sqlite` (and `SqlServer`) project, add a file named `ConsoleLogger.cs`.
2. Modify the file to define three classes, one to implement `ILoggerFactory`, one to implement `ILoggerProvider`, and one to implement `ILogger`, as shown in the following code:

```
using Microsoft.Extensions.Logging;

using static System.Console;
```

```
namespace Packt.Shared;

public class ConsoleLoggerFactory : ILoggerFactory
{
  public void AddProvider(ILoggerProvider provider) { }

  public ILogger CreateLogger(string categoryName)
  {
    return new ConsoleLogger();
  }

  public void Dispose() { }
}

public class ConsoleLogger : ILogger
{
  public IDisposable BeginScope<TState>(TState state)
  {
    return null;
  }

  public bool IsEnabled(LogLevel logLevel)
  {
    switch(logLevel)
    {
      case LogLevel.Trace:
      case LogLevel.Information:
      case LogLevel.None:
        return false;
      case LogLevel.Debug:
      case LogLevel.Warning:
      case LogLevel.Error:
      case LogLevel.Critical:
      default:
        return true;
    };
  }

  public void Log<TState>(LogLevel logLevel,
    EventId eventId, TState state, Exception exception,
    Func<TState, Exception, string> formatter)
  {
    if (eventId.Id == 20100) // execute SQL statement
    {
```

```
        Write($"Level: {logLevel}, Event Id: {eventId.Id}");

        // only output the state or exception if it exists
        if (state != null)
        {
          Write($", State: {state}");
        }

        if (exception != null)
        {
          Write($", Exception: {exception.Message}");
        }
        WriteLine();
      }
    }
  }
```

3. In `NorthwindContextExtensions.cs`, in the `AddNorthwindContext` method, after the call to `UseSqlServer` or `UseSqlite`, call `UseLoggerFactory` to register your custom console logger, as shown highlighted in the following code:

```
services.AddDbContext<NorthwindContext>(options =>
  options.UseSqlServer(connectionString) // or UseSqlite(...)
  .UseLoggerFactory(new ConsoleLoggerFactory())
);
```

4. Start the `Northwind.OData` web service.
5. Start Chrome.
6. Navigate to `https://localhost:5004/catalog/products/?$filter=startswith(ProductName,'Ch') or (UnitPrice gt 50)&$select=ProductId,ProductName,UnitPrice`.
7. In Chrome, note the result, as shown in the following output:

```
{"@odata.context":"https://localhost:5004/catalog/$metadata#Products(Prod
uctId,ProductName,UnitPrice)","value":[{"ProductId":1,"ProductName":"Chai
","UnitPrice":18.0000},{"ProductId":2,"ProductName":"Chang","UnitPrice":1
9.0000},{"ProductId":4,"ProductName":"Chef Anton's Cajun Seasoning","Uni
tPrice":22.0000},{"ProductId":5,"ProductName":"Chef Anton's Gumbo Mix","
UnitPrice":21.3500},{"ProductId":9,"ProductName":"Mishi Kobe Niku","Unit
Price":97.0000},{"ProductId":18,"ProductName":"Carnarvon Tigers","UnitPr
ice":62.5000},{"ProductId":20,"ProductName":"Sir Rodney's Marmalade","Un
itPrice":81.0000},{"ProductId":29,"ProductName":"Th\u00fcringer Rostbrat
wurst","UnitPrice":123.7900},{"ProductId":38,"ProductName":"C\u00f4te de
Blaye","UnitPrice":263.5000},{"ProductId":39,"ProductName":"Chartreuse ver
te","UnitPrice":18.0000},{"ProductId":48,"ProductName":"Chocolade","UnitPr
ice":12.7500},{"ProductId":51,"ProductName":"Manjimup Dried Apples","UnitP
rice":53.0000},{"ProductId":59,"ProductName":"Raclette Courdavault","UnitP
rice":55.0000}]}
```

8. At the command prompt or terminal, note the logged SQL statement, for example, if using the SQL Server database provider, as shown in the following output:

```
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[@__
TypedProperty_0='?' (Size = 4000), @__TypedProperty_1='?' (DbType =
Decimal)], CommandType='Text', CommandTimeout='30']
SELECT [p].[ProductId], [p].[ProductName], [p].[UnitPrice]
FROM [Products] AS [p]
WHERE ((@__TypedProperty_0 = N'') OR (LEFT([p].[ProductName], LEN(@__
TypedProperty_0)) = @__TypedProperty_0)) OR ([p].[UnitPrice] > @__
TypedProperty_1)
```

It might look like the `Get` action method on the `ProductsController` returns the entire `Products` table, but it actually returns an `IQueryable<Products>` object. In other words, it returns a LINQ query, not yet the results. We decorated the `Get` action method with the `[EnableQuery]` attribute. This enables OData to extend the LINQ query with filters, projections, sorting, and so on, and only then does it execute the query, serialize the results, and return them to the client. This makes OData services as flexible *and* efficient as possible.

Versioning OData controllers

It is good practice to plan for future versions of your OData models that might have different schemas and behavior.

To maintain backwards compatibility, you can use OData URL prefixes to specify a version number:

1. In the `Northwind.OData` project, in `Program.cs`, in the services configuration section, after adding the two OData models for `catalog` and `ordersystem`, add a third OData model that has a version number, as shown highlighted in the following code:

```
.AddRouteComponents(routePrefix: "catalog",
  model: GetEdmModelForCatalog())
.AddRouteComponents(routePrefix: "ordersystem",
  model: GetEdmModelForOrderSystem())
.AddRouteComponents(routePrefix: "v{version}",
  model: GetEdmModelForCatalog())
```

2. In `ProductsController.cs`, statically import `Console` and then modify the `Get` methods to add a `string` parameter named `version`, and use it to change the behavior of the methods if version 2 is specified in a request, as shown highlighted in the following code:

```
[EnableQuery]
public IActionResult Get(string version = "1")
{
```

```
  WriteLine($"ProductsController version {version}.");
  return Ok(db.Products);
}

[EnableQuery]
public IActionResult Get(int key, string version = "1")
{
  WriteLine($"ProductsController version {version}.");
  Product? p = db.Products.Find(key);
  if (p is null)
  {
    return NotFound($"Product with id {key} not found.");
  }
  if (version == "2")
  {
    p.ProductName += " version 2.0";
  }
  return Ok(p);
}
```

3. In your preferred code editor, start the `Northwind.OData` project web service.
4. In Visual Studio Code, in `odata-catalog-queries.http`, add a request to get the product with ID `50` using the v2 OData model, as shown in the following code:

```
GET https://localhost:5004/v2/products(50)
```

5. Click **Send Request**, and note the response is the product with its name appended with **version 2.0**, as shown highlighted in the following output:

```
{
  "@odata.context": "https://localhost:5004/
v2/$metadata#Products/$entity",
  "ProductId": 50,
  "ProductName": "Valkoinen suklaa version 2.0",
  "SupplierId": 23,
  "CategoryId": 3,
  "QuantityPerUnit": "12 - 100 g bars",
  "UnitPrice": 16.2500,
  "UnitsInStock": 65,
  "UnitsOnOrder": 0,
  "ReorderLevel": 30,
  "Discontinued": false
}
```

Enabling entity inserts using POST

The most common use for OData is to provide a Web API that supports custom queries. You might also want to support CRUD operations like inserts.

1. In `ProductsController.cs`, add an action method to respond to `POST` requests, as shown in the following code:

```
public IActionResult Post([FromBody] Product product)
{
  db.Products.Add(product);
  db.SaveChanges();
  return Created(product);
}
```

2. Start the web service.
3. Create a file named `odata-catalog-insert-product.http`, as shown in the following HTTP request:

```
POST https://localhost:5004/catalog/products
Content-Type: application/json
Content-Length: 234

{
  "ProductName": "Impossible Burger",
  "SupplierId": 7,
  "CategoryId": 6,
  "QuantityPerUnit": "Pack of 4",
  "UnitPrice": 40.25,
  "UnitsInStock": 50,
  "UnitsOnOrder": 0,
  "ReorderLevel": 30,
  "Discontinued": false
}
```

4. Click **Send Request**.
5. Note the successful response, as shown in the following markup:

```
HTTP/1.1 201 Created
Connection: close
Content-Type: application/json; odata.metadata=minimal; odata.
streaming=true
Date: Sat, 17 Jul 2021 12:01:57 GMT
Server: Kestrel
Location: https://localhost:5004/catalog/Products(80)
Transfer-Encoding: chunked
```

```
OData-Version: 4.0

{
  "@odata.context": "https://localhost:5004/catalog/$metadata#Products/$e
ntity",
  "ProductId": 78,
  "ProductName": "Impossible Burger",
  "SupplierId": 7,
  "CategoryId": 6,
  "QuantityPerUnit": "Pack of 4",
  "UnitPrice": 40.25,
  "UnitsInStock": 50,
  "UnitsOnOrder": 0,
  "ReorderLevel": 30,
  "Discontinued": false
}
```

Building a client for OData

In this section, let's see how a client might call the OData web service.

If we want to query the OData service for products that start with the letters `Cha`, then we would need to send a `GET` request with a relative URL path similar to the following:

```
catalog/products/?$filter=startswith(ProductName, 'Cha')&$select=ProductId,Produ
ctName,UnitPrice
```

OData returns data in a JSON document with a property named `value` that contains the resulting products as an array, as shown in the following JSON document:

```
{
  "@odata.context": "https://localhost:5004/catalog/$metadata#Products",
  "value": [
    {
      "ProductId": 1,
      "ProductName": "Chai",
      "SupplierId": 1,
      "CategoryId": 1,
      "QuantityPerUnit": "10 boxes x 20 bags",
      "UnitPrice": 18,
      "UnitsInStock": 39,
      "UnitsOnOrder": 0,
      "ReorderLevel": 10,
      "Discontinued": false
    },
```

We will create a model class to make it easy to deserialize the response:

1. In the `Northwind.Mvc` project, in the `Models` folder, add a new class file named `ODataProducts.cs`, as shown in the following code:

```
using Packt.Shared; // Product

namespace Northwind.Mvc.Models;

public class ODataProducts
{
  public Product[]? Value { get; set; }
}
```

2. In `Program.cs`, add statements to register an HTTP client for the OData service, as shown in the following code:

```
builder.Services.AddHttpClient(name: "Northwind.OData",
  configureClient: options =>
  {
    options.BaseAddress = new Uri("https://localhost:5004/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
      "application/json", 1.0));
  });
```

Adding a services page to the Northwind MVC website

Next, we will create a services page:

1. In the `Controllers` folder, open `HomeController.cs`, and add a new action method for services that calls the OData service to get products that start with `Cha` and stores the result in the `ViewData` dictionary, as shown in the following code:

```
public async Task<IActionResult> Services()
{
  try
  {
    HttpClient client = clientFactory.CreateClient(
      name: "Northwind.OData");

    HttpRequestMessage request = new(
      method: HttpMethod.Get, requestUri:
      "catalog/products/?$filter=startswith(ProductName, 'Cha')&$select=Pr
oductId,ProductName,UnitPrice");

    HttpResponseMessage response = await client.SendAsync(request);
```

```
    ViewData["productsCha"] = (await response.Content
      .ReadFromJsonAsync<ODataProducts>())?.Value;
  }
  catch (Exception ex)
  {
    _logger.LogWarning($"Northwind.OData service exception: {ex.
Message}");
  }

  return View();
}
```

2. In `Views/Shared`, in `_Layout.cshtml`, add a new nav item after the `Privacy` nav item that goes to a services page, as shown in the following markup:

```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area=""
       asp-controller="Home" asp-action="Services">Services</a>
</li>
```

3. In `Views/Home`, add a new empty view named `Services.cshtml` and modify its contents to render the products, as shown in the following markup:

```
@using Packt.Shared
@using Northwind.Common
@{
  ViewData["Title"] = "Services";
  Product[]? products = ViewData["productsCha"] as Product[];
}
<div class="text-center">
  <h1 class="display-4">@ViewData["Title"]</h1>
  @if (ViewData["productsCha"] != null)
  {
    <h2>Products that start with Cha using OData</h2>
    <p>
      @if (products is null)
      {
        <span class="badge badge-info">No products found.</span>
      }
      else
      {
        @foreach (Product p in products)
        {
          <span class="badge badge-info">
            @p.ProductId
            @p.ProductName
```

```
            @(p.UnitPrice is null ? "" : p.UnitPrice.Value.ToString("c"))
          </span>
        }
      }
    </p>
  }
</div>
```

4. Optionally, start the `Minimal.Web` project without debugging. (If you are using Visual Studio 2022, select the project in **Solution Explorer** to make it the current selection and then navigate to **Debug | Start Without Debugging**.)
5. Start the `Northwind.OData` project without debugging.
6. Start the `Northwind.Mvc` project without debugging.
7. Start Chrome.
8. Navigate to the **Services** page by clicking the menu or navigating to the following link: `https://localhost:5001/home/services`.
9. Note three products are returned from the OData service, as shown in *Figure 18.2*:

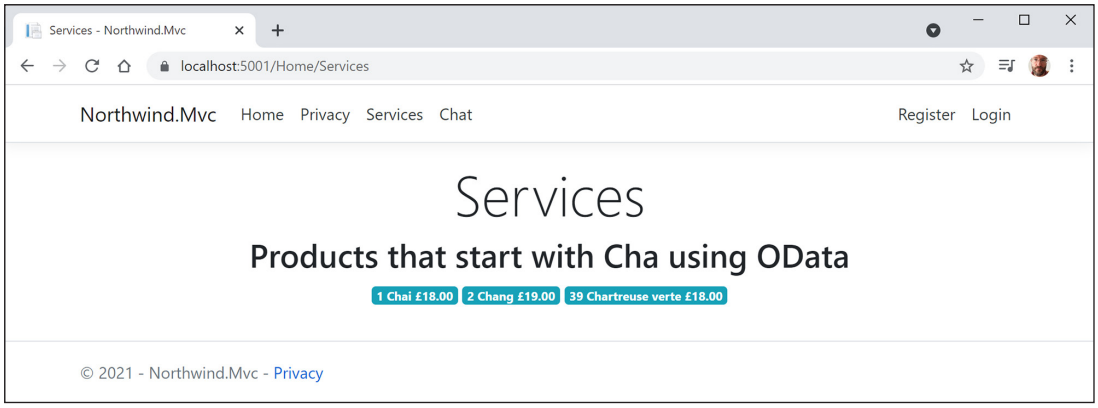


Figure 18.2: Three product names start with Cha returned from the OData service

10. Close Chrome and shut down all the web servers.

Exposing data as a service using GraphQL

If you would prefer to use more modern technology for exposing your data as a service, then an alternative to OData is GraphQL.

Understanding GraphQL

Like OData, GraphQL is a standard for describing your data and then querying it that gives the client control over exactly what they need. It was developed internally by Facebook in 2012 before being open sourced in 2015 and is now managed by the GraphQL Foundation.

Some benefits of GraphQL over OData are that it does not require HTTP because it is transport-agnostic, so you could use alternative transport protocols like WebSockets, and GraphQL has a single endpoint, usually simply `/graphql`.

GraphQL uses its own document format for its queries, which is a bit like JSON, but GraphQL queries do not require commas between field names, as shown in the following query:

```
{
  product (productId: 23) {
    productId
    productName
    cost
    supplier {
      companyName
      country
    }
  }
}
```

The official media type for GraphQL query documents is `application/graphql`.

Building a service that supports GraphQL

There is no `dotnet new` project template for GraphQL so we will use the Web API project template (even though GraphQL does not have to be hosted in a web service) and then add package references for GraphQL support:

1. Use your preferred code editor to add a new project, as defined in the following list:
    1. Project template: **ASP.NET Core Web API** / `webapi`
    2. Workspace/solution file and folder: `PracticalApps`
    3. Project file and folder: `Northwind.GraphQL`
    4. Other Visual Studio options: **Authentication Type**: None, **Configure for HTTPS**: selected, **Enable Docker**: cleared, **Enable OpenAPI support**: cleared.

Good Practice: Since GraphQL is not a traditional Web API service, Swagger should not be enabled. If you are using the command line, then use the following switch: `dotnet new webapi --no-openapi`.

2. In Visual Studio Code, select `Northwind.GraphQL` as the active OmniSharp project.

3. Add package references for the core GraphQL server-side components and the GraphQL playground user interface, as shown in the following markup:

```
<ItemGroup>
  <PackageReference
    Include="GraphQL.Server.Transports.AspNetCore"
    Version="5.0.2" />
  <PackageReference
    Include="GraphQL.Server.Transports.AspNetCore.SystemTextJson"
    Version="5.0.2" />
  <PackageReference
    Include="GraphQL.Server.Ui.Playground"
    Version="5.0.2" />
</ItemGroup>
```

Good Practice: Remove the GraphQL playground user interface package before deploying the service into production. Although you could only enable the playground in development mode, any unused packages increase the potential attack surface of your project.

4. Add a project reference to the Northwind database context project for either SQLite or SQL Server, as shown in the following markup:

```
<ItemGroup>
  <!-- change Sqlite to SqlServer if you prefer -->
  <ProjectReference Include=
"..\Northwind.Common.DataContext.Sqlite\Northwind.Common.DataContext.
Sqlite.csproj" />
</ItemGroup>
```

5. In the `Northwind.GraphQL` folder, delete `WeatherForecast.cs`.
6. In the `Controllers` folder, delete `WeatherForecastController.cs`.
7. Delete the `Controllers` folder.
8. In `Program.cs`, add an extension method call to `UseUrls` to specify port `5005` for `HTTPS`, as shown highlighted in the following code:

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

builder.WebHost.UseUrls("https://localhost:5005/");
```

9. In the `Properties` folder, open `launchSettings.json` and modify the `launchUrl` and `applicationUrl` settings, as shown highlighted in the following markup:

```
"profiles": {
  "Northwind.GraphQL": {
    "commandName": "Project",
    "dotnetRunMessages": "true",
    "launchBrowser": true,
```

```
    "launchUrl": "ui/playground",
    "applicationUrl": "https://localhost:5005",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  },
```

10. Build the `Northwind.GraphQL` project.

Defining GraphQL schema for Hello World

The first task is to define what we want to expose as GraphQL models in the web service.

Let's define a GraphQL model for the most basic Hello World example:

1. In the `Northwind.GraphQL` project/folder, add a class file named `GreetQuery.cs`.
2. Modify the class to define an object graph type named `greet` that responds with the plain text `"Hello, World!"`, as shown in the following code:

```
using GraphQL.Types; // ObjectGraphType

namespace Northwind.GraphQL;

public class GreetQuery : ObjectGraphType
{
  public GreetQuery()
  {
    Field<StringGraphType>(name: "greet",
      description: "A query type that greets the world.",
      resolve: context => "Hello, World!");
  }
}
```

3. In the `Northwind.GraphQL` project/folder, add a class file named `NorthwindSchema.cs`.
4. Modify the class to define an object graph schema that registers the `GreetQuery` class as the only type of query, as shown in the following code:

```
using GraphQL.Types; // Schema

namespace Northwind.GraphQL;

public class NorthwindSchema : Schema
{
  public NorthwindSchema(IServiceProvider provider) : base(provider)
  {
    Query = new GreetQuery();
  }
}
```

Good Practice: Use constructor parameter injection to get the `IServiceProvider` that will be used to get registered dependency services.

5. In `Program.cs`, import the namespace for working with GraphQL and the GraphQL query and schema that you just defined, as shown in the following code:

```
using GraphQL.Server; // GraphQLOptions
using Northwind.GraphQL; // GreetQuery, NorthwindSchema
```

6. In the section for configuring services, after the call to `AddControllers`, add statements to register the schema class as a scoped dependency service and to add GraphQL support, as shown in the following code:

```
builder.Services.AddScoped<NorthwindSchema>();

builder.Services.AddGraphQL()
  .AddGraphTypes(typeof(NorthwindSchema), ServiceLifetime.Scoped)
  .AddDataLoader()
  .AddSystemTextJson(); // serialize responses as JSON
```

Good Practice: Later, the Northwind database context will be registered as a scoped dependency service so any services that use it must also be registered as scoped rather than singleton.

7. In the section for configuring the HTTP pipeline, add statements to use GraphQL with the Northwind schema and use the playground user interface if in development mode, as shown highlighted in the following code:

```
if (builder.Environment.IsDevelopment())
{
  app.UseGraphQLPlayground(); // default path is /ui/playground
}

app.UseGraphQL<NorthwindSchema>(); // default path is /graphql

app.UseHttpsRedirection();
```

8. Start the `Northwind.GraphQL` service project.
9. If you are using Visual Studio Code, then start Chrome and navigate to `https://localhost:5005/ui/playground`.

10. On the left-hand side, write an unnamed query to request `greet`, as shown in the following markup:

```
{
  greet
}
```

11. Click the circular gray play button, and note the endpoint URL used by the playground, `https://localhost:5005/graphql`, and the response, as shown in the following output and *Figure 18.3*:

```
{
  "data": {
    "greet": "Hello, World!"
  },
  "extensions": {}
}
```

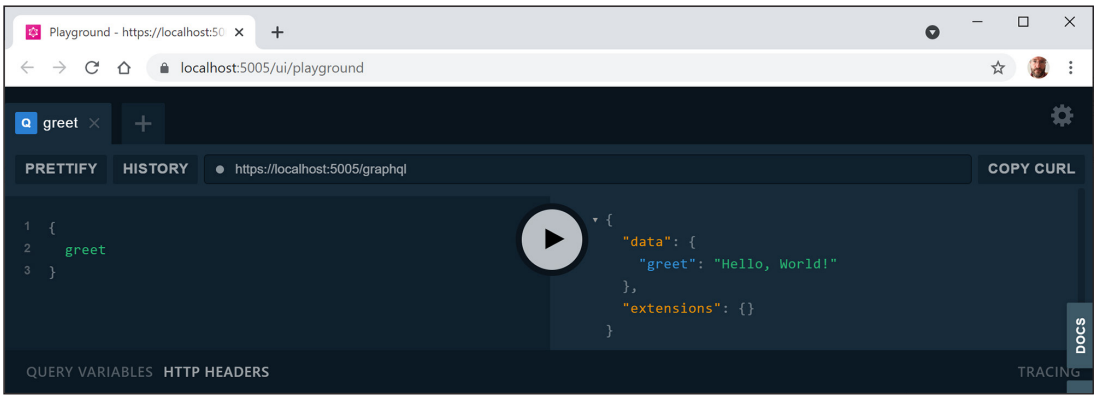


Figure 18.3: Using the GraphQL playground to execute a greet query

12. Close Chrome and shut down the web server.

We could also have created it as a named query, as shown in the following code:

```
query QueryNameGoesHere {
  greet
}
```

Named queries allow clients to identify queries and responses for telemetry purposes, for example, when hosting in Microsoft Azure cloud services and monitoring using Application Insights.

Defining GraphQL schema for EF Core models

Now that we have a basic GraphQL service operating successfully, let's add types to enable querying the Northwind database:

1. In `Program.cs`, import the namespace for working with our EF Core model for the Northwind database, as shown in the following code:

```
using Packt.Shared; // AddNorthwindContext extension method
```

2. Add a statement to the top of the section for configuring services to register the `Northwind` database context class, as shown in the following code:

```
builder.Services.AddNorthwindContext();
```

3. In the `Northwind.GraphQL` project/folder, add a class file named `CategoryType.cs`. This is used to describe the `Category` entity class to the GraphQL system.
4. Modify the `CategoryType` class to define an object graph type that matches the structure of the `Category` entity model, as shown in the following code:

```
using GraphQL.Types; // ObjectGraphType<T>, ListGraphType<T>
using Packt.Shared; // Category

namespace Northwind.GraphQL;

public class CategoryType : ObjectGraphType<Category>
{
  public CategoryType()
  {
    Name = "Category";
    Field(c => c.CategoryId).Description("Id of the category.");
    Field(c => c.CategoryName).Description("Name of the category.");
    Field(c => c.Description).Description("Description of the category.");
    Field(c => c.Products, type: typeof(ListGraphType<ProductType>))
      .Description("Products in the category.");
  }
}
```

The `ProductType` class will generate a temporary error because we have not created it yet.

5. In the `Northwind.GraphQL` project/folder, add a class file named `ProductType.cs`.
6. Modify the class to define an object graph type that matches the structure of the `Product` entity model, as shown in the following code:

```
using GraphQL.Types; // ObjectGraphType<T>, IntGraphType, DecimalGraphType
using Packt.Shared; // Category, Product

namespace Northwind.GraphQL;

public class ProductType : ObjectGraphType<Product>
```

```
{
  public ProductType()
  {
    Name = "Product";
    Field(p => p.ProductId).Description("Id of the product.");
    Field(p => p.ProductName).Description("Name of the product.");
    Field(p => p.CategoryId, type: typeof(IntGraphType))
      .Description("CategoryId of the product.");
    Field(p => p.Category, type: typeof(CategoryType))
      .Description("Category of the product.");
    Field(p => p.UnitPrice, type: typeof(DecimalGraphType))
      .Description("Unit price of the product.");
    Field(p => p.UnitsInStock, type: typeof(IntGraphType))
      .Description("Units in stock of the product.");
    Field(p => p.UnitsOnOrder, type: typeof(IntGraphType))
      .Description("Units on order of the product.");
  }
}
```

Good Practice: Simple types like `int` and `string` are automatically recognized by GraphQL. Nullable types like `int?` and `decimal?` used by the `CategoryId` and `UnitPrice` properties, complex types like `Category` used by the `Category` property, and small integers like `short` need to be explicitly specified, otherwise GraphQL will throw exceptions at runtime.

7. In the `Northwind.GraphQL` project/folder, add a class file named `NorthwindQuery.cs`.
8. Modify the class to define an object graph type that has three types of query to return a list of categories, a single category, and products for a category, as shown in the following code:

```
using GraphQL; // GetArgument extension method
using GraphQL.Types; // ObjectGraphType, QueryArguments, QueryArgument<T>
using Microsoft.EntityFrameworkCore; // Include extension method
using Packt.Shared; // NorthwindContext

namespace Northwind.GraphQL;

public class NorthwindQuery : ObjectGraphType
{
  public NorthwindQuery(NorthwindContext db)
  {
    Field<ListGraphType<CategoryType>>(
```

```
      name: "categories",
      description: "A query type that returns a list of all categories.",
      resolve: context => db.Categories.Include(c => c.Products)
    );

    Field<CategoryType>(
      name: "category",
      description: "A query type that returns a category using its Id.",
      arguments: new QueryArguments(
        new QueryArgument<IntGraphType> { Name = "categoryId" }),
      resolve: context =>
      {
        Category? category = db.Categories.Find(
          context.GetArgument<int>("categoryId"));
        db.Entry(category).Collection(c => c.Products).Load();
        return category;
      }
    );

    Field<ListGraphType<ProductType>>(
      name: "products",
      arguments: new QueryArguments(
        new QueryArgument<IntGraphType> { Name = "categoryId" }),
      resolve: context =>
      {
        Category? category = db.Categories.Find(
          context.GetArgument<int>("categoryId"));
        db.Entry(category).Collection(c => c.Products).Load();
        return category.Products;
      }
    );
  }
}
```

9. In `NorthwindSchema.cs`, import the namespace for getting a service with dependency injection and for the Northwind database context, as shown in the following code:

```
using Packt.Shared; // NorthwindContext
using Microsoft.Extensions.DependencyInjection; // GetRequiredService
```

10. In the constructor, comment out the statement that sets `Query` to use `GreetQuery` and add a statement that sets it to use `NorthwindQuery`, getting and passing the required Northwind database context, as shown in the following code:

```
// Query = new GreetQuery();
Query = new NorthwindQuery(provider.GetRequiredService<NorthwindConte
xt>());
```

Exploring GraphQL queries with Northwind

Now we can test writing GraphQL queries for the Northwind database:

1. Start the `Northwind.GraphQL` service project.
2. If you are using Visual Studio Code, then start Chrome and navigate to `https://localhost:5005/ui/playground`.
3. In the playground, click the **Schema** tab on the right-hand side, and note the schema, query, and type definitions, and note the IntelliSense help provided as you type a query, as shown in *Figure 18.4*:

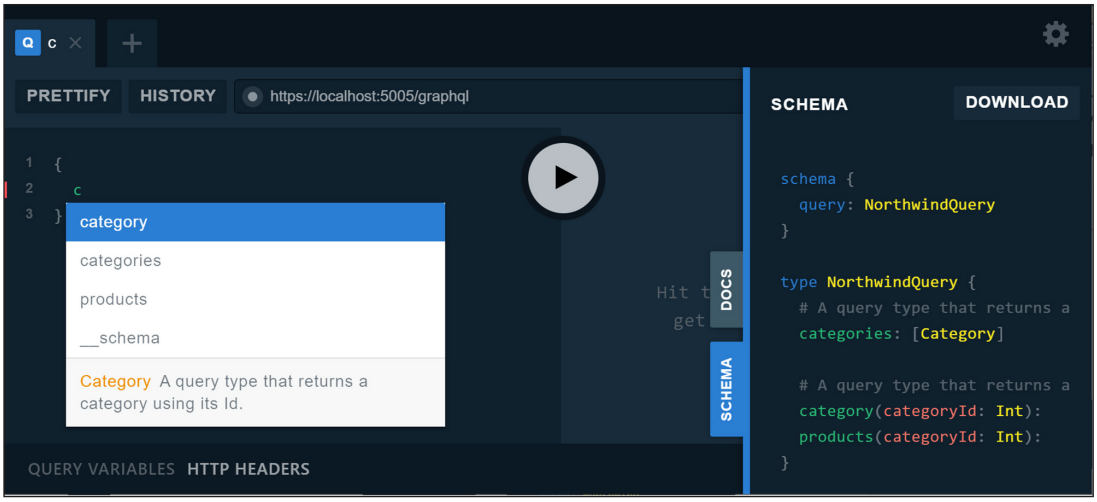


Figure 18.4: Schema for querying the Northwind categories and products using GraphQL

4. Click the **Docs** tab and then click **categories**, **category(...)**, and **products(...)** to review the documentation.
5. Click the **Docs** tab again to collapse the pane.
6. On the left-hand side, write a named query to request all categories, as shown in the following markup:

```
query AllCategories {
  categories {
    categoryId
    categoryName
    description
  }
}
```

7. Click the play button, and note the response, as shown in the following partial output:

```
{
  "data": {
```

```
    "categories": [
      {
        "categoryId": 1,
        "categoryName": "Beverages",
        "description": "Soft drinks, coffees, teas, beers, and ales"
      },
      {
        "categoryId": 2,
        "categoryName": "Condiments",
        "description": "Sweet and savory sauces, relishes, spreads, and
seasonings"
      },
      ...
```

8. Click the **+** tab to open a new tab, write a query to request the category with Id `2`, including the Id, name, and price of its products, as shown in the following markup:

```
query Condiments {
  category (categoryId: 2) {
    categoryId
    categoryName
    products {
      productId
      productName
      unitPrice
    }
  }
}
```

Make sure that the `I` in `categoryId` is uppercase.

9. Click the play button, and note the response, as shown in the following partial output:

```
{
  "data": {
    "category": {
      "categoryId": 2,
      "categoryName": "Condiments",
      "products": [
        {
          "productId": 3,
          "productName": "Aniseed Syrup",
```

```
          "unitPrice": 10
        },
        {
          "productId": 4,
          "productName": "Chef Anton's Cajun Seasoning",
          "unitPrice": 22
        },
        ...
```

10. Click the **+** tab to open a new tab, write a query to request the Id, name, and units in stock of the products in the category with Id `1`, as shown in the following markup:

```
query BeverageProducts {
  products (categoryId: 1) {
    productId
    productName
    unitsInStock
  }
}
```

11. Click the play button, and note the response, as shown in the following partial output:

```
{
  "data": {
    "products": [
      {
        "productId": 1,
        "productName": "Chai",
        "unitsInStock": 39
      },
      {
        "productId": 2,
        "productName": "Chang",
        "unitsInStock": 17
      },
      ...
```

12. Close Chrome and shut down the web server.

Understanding GraphQL mutations and subscriptions

As well as queries, other standard GraphQL features are mutations and subscriptions.

- **Mutations** enable creating, updating, and deleting resources.

- **Subscriptions** enable a client to get notified when resources change. They work best with alternative communication technologies like WebSockets.

If you would like me to add coverage of these in the next edition, please get in touch and let me know.

Building a client for GraphQL

Finally in this section, let's see how a client could call the GraphQL service:

We will create a model class to make it easy to deserialize the response:

1. In the `Northwind.Mvc` project, in the `Models` folder, add a new class file named `GraphQLProducts.cs`, as shown in the following code:

```
using Packt.Shared; // Product

namespace Northwind.Mvc.Models;

public class GraphQLProducts
{
  public class DataProducts
  {
    public Product[]? Products { get; set; }
  }

  public DataProducts? Data { get; set; }
}
```

2. In `Program.cs`, add statements to register an HTTP client for the GraphQL service, as shown in the following code:

```
builder.Services.AddHttpClient(name: "Northwind.GraphQL",
  configureClient: options =>
  {
    options.BaseAddress = new Uri("https://localhost:5005/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
      "application/json", 1.0));
  });
```

3. In the `Controllers` folder, in `HomeController.cs`, import the namespace for working with text encodings, as shown in the following code:

```
using System.Text; // Encoding
```

4. In the `Services` action method, add statements to call the GraphQL service, and note the HTTP request is a `POST` request, the media type is for GraphQL, and the query requests all products in category 8 (which is Seafood), as shown in the following code:

```
try
{
  HttpClient client = clientFactory.CreateClient(
    name: "Northwind.GraphQL");

  HttpRequestMessage request = new(
    method: HttpMethod.Post, requestUri: "graphql");

  request.Content = new StringContent(content: @"
    {
      products (categoryId: 8) {
        productId
        productName
        unitsInStock
      }
    }",
    encoding: Encoding.UTF8,
    mediaType: "application/graphql");

  HttpResponseMessage response = await client.SendAsync(request);

  if (response.IsSuccessStatusCode)
  {
    ViewData["seafoodProducts"] = (await response.Content
      .ReadFromJsonAsync<GraphQLProducts>())?.Data?.Products;
  }
  else
  {
    ViewData["seafoodProducts"] = Enumerable.Empty<Product>().ToArray();
  }
}
catch (Exception ex)
{
  _logger.LogWarning($"Northwind.GraphQL service exception: {ex.
Message}");
}
```

5. In the `Views/Home` folder, in `Services.cshtml`, add a new section inside the outmost `<div>` after the `@if` section that renders products that start with `Cha` from the OData service, and to render the Seafood products, as shown highlighted in the following markup:

```
@{
  ViewData["Title"] = "Services";
  Product[]? products = ViewData["productsCha"] as Product[];
```

```
    Product[]? seafoodProducts = ViewData["seafoodProducts"] as Product[];
}
...
@if (seafoodProducts is not null)
{
  <h2>Seafood products using GraphQL</h2>
  <p>
    @foreach (Product p in seafoodProducts)
    {
      <span class="badge badge-success">
        @p.ProductId
        @p.ProductName
        -
        @(p.UnitsInStock is null ? "0" : p.UnitsInStock.Value) in stock
      </span>
    }
  </p>
}
```

6. Optionally, start the `Minimal.Web` project without debugging.
7. Optionally, start the `Northwind.OData` project without debugging.
8. Start the `Northwind.GraphQL` project without debugging.
9. Start the `Northwind.Mvc` project.
10. Navigate to the **Services** page: `https://localhost:5001/home/services`.
11. Note Seafood products are successfully retrieved using GraphQL, as shown in *Figure 18.5*:

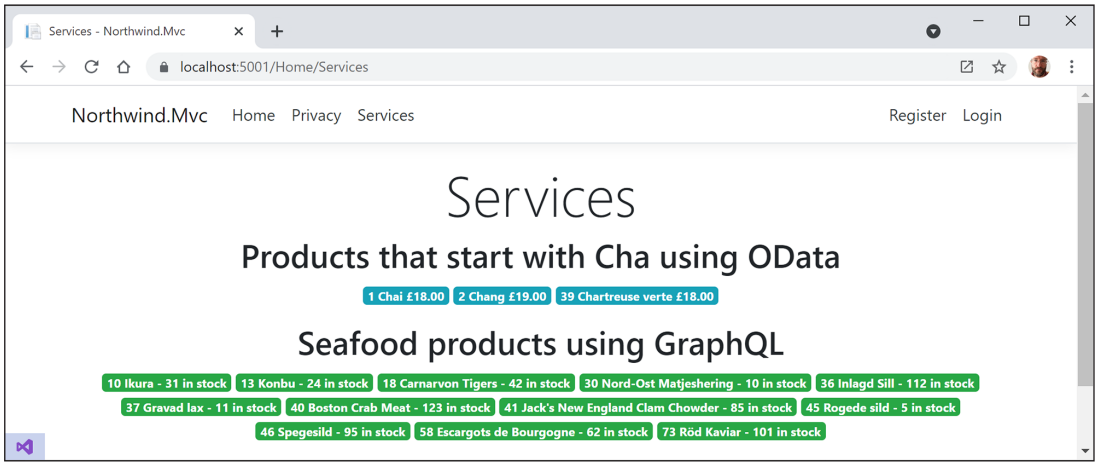


Figure 18.5: Products in the Seafood category from the GraphQL service

12. Close Chrome and shut down all the web servers.

Implementing services using gRPC

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

Understanding gRPC

A gRPC client can call methods in a gRPC service on a different server as if it was a local object. The developer defines a service interface with methods that can be called remotely including their parameters and return types. The server implements this interface and runs a gRPC server to handle client calls. On the client, a strongly typed gRPC client provides the same methods as on the server.

Like WCF, gRPC uses contract-first API development that supports language-agnostic implementations. You write the contracts using `.proto` files that have their own language syntax and then use tools to convert them into various languages like C#. The `.proto` files are used by both the server and client to exchange messages in the correct format.

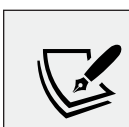
gRPC minimizes network usage by using Protobuf binary serialization that is not human readable, unlike JSON or XML used by web services. gRPC requires HTTP/2 that provides significant performance benefits over earlier versions like binary framing and compression, and multiplexing of HTTP/2 calls over a single connection.

The main limitation of gRPC is that it cannot be used in web browsers because no browser provides the level of control required to support a gRPC client. For example, browsers do not allow a caller to require that HTTP/2 be used. There is an initiative called **gRPC-Web** that adds an extra proxy layer and the proxy forwards requests to the gRPC server.

Building a gRPC service

Let's see an example service for managing suppliers in the Northwind database:

1. Use your preferred code editor to add a new project, as defined in the following list:
    1. Project template: **ASP.NET Core gRPC Service** / `grpc`
    2. Workspace/solution file and folder: `PracticalApps`
    3. Project file and folder: `Northwind.gRPC`
2. In Visual Studio Code, select `Northwind.gRPC` as the active OmniSharp project.



For working with `.proto` files in Visual Studio Code, you can install the extension vscode-proto3 (`zxh404.vscode-proto3`).

3. In the `Protos` folder, open `greet.proto`, and note that it defines a service named `Greeter` with a method named `SayHello` that exchanges messages named `HelloRequest` and `HelloReply`, as shown in the following code:

```
syntax = "proto3";

option csharp_namespace = "Northwind.gRPC";

package greet;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply);
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings.
message HelloReply {
  string message = 1;
}
```

4. Open the `Northwind.gRPC.csproj` file, and note the package reference for implementing a gRPC service hosted in ASP.NET Core and the `.proto` file is registered for use on the server-side, as shown in the following markup:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
</ItemGroup>

<ItemGroup>
  <PackageReference Include="Grpc.AspNetCore" Version="2.32.0" />
</ItemGroup>
```

5. In the `Services` folder, open `GreeterService.cs`, and note that it implements the `Greeter` service contract, as shown in the following code:

```
using Grpc.Core; // ServerCallContext
using Northwind.gRPC;

namespace Northwind.gRPC.Services;
public class GreeterService : Greeter.GreeterBase
{
  private readonly ILogger<GreeterService> _logger;
```

```
  public GreeterService(ILogger<GreeterService> logger)
  {
    _logger = logger;
  }

  public override Task<HelloReply> SayHello(
    HelloRequest request, ServerCallContext context)
  {
    return Task.FromResult(new HelloReply
    {
      Message = "Hello " + request.Name
    });
  }
}
```

6. In `Program.cs`, in the section that configures services, note the call to add gRPC to the services collection, as shown in the following code:

```
builder.Services.AddGrpc();
```

7. In `Program.cs`, in the section for configuring the HTTP pipeline, note the call to map the `Greeter` service, as shown in the following code:

```
app.MapGrpcService<GreeterService>();
```

8. In `Program.cs`, add an extension method call to `UseUrls` to specify port `5006` for `HTTPS`, as shown highlighted in the following code:

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

builder.WebHost.UseUrls("https://localhost:5006/");
```

9. In the `Properties` folder, open `launchSettings.json` and modify the `applicationUrl` setting to use port `5006`, as shown highlighted in the following markup:

```
{
  "profiles": {
    "Northwind.gRPC": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": false,
      "applicationUrl": "https://localhost:5006",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

10. Build the `Northwind.gRPC` project.

Building a gRPC client

We will add gRPC client packages to the Northwind MVC website project to enable it to call the gRPC service:

1. In the `Northwind.Mvc` project, add package references for Google's Protobuf format, the gRPC client, and tools, as shown in the following markup:

```
<PackageReference Include="Google.Protobuf" Version="3.17.3" />
<PackageReference Include="Grpc.Net.Client" Version="2.38.0" />
<PackageReference Include="Grpc.Tools" Version="2.38.1">
  <PrivateAssets>all</PrivateAssets>
  <IncludeAssets>runtime; build; native; contentfiles;
    analyzers; buildtransitive</IncludeAssets>
</PackageReference>
```

Good Practice: The `Grpc.Tools` package is only used during development, so it is marked as `PrivateAssets=all` to ensure that the tools are not published with the production website.

2. Copy the `Protos` folder from the `Northwind.gRPC` project/folder to the `Northwind.Mvc` project/folder. (In Visual Studio 2022, you can drag and drop to copy. In Visual Studio Code, drag and drop while holding the *Ctrl* or *Cmd* key.)
3. In the `Northwind.Mvc` project, in `greet.proto`, modify the namespace to match the namespace for the current project so that the automatically generated classes will be in the same namespace, as shown in the following code:

```
option csharp_namespace = "Northwind.Mvc";
```

4. In the `Northwind.Mvc` project file, add an item group to register the `.proto` file as being used on the client side, as shown in the following markup:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>
```

Visual Studio will have created the item group for you but it will set the `GrpcServices` to `Server` by default so you must manually change that to `Client`.

5. Build the `Northwind.Mvc` project to ensure that the automatically generated classes are created.

6. In `HomeController.cs`, import namespaces to work with gRPC as a client, as shown in the following code:

```
using Grpc.Net.Client; // GrpcChannel
```

7. In the `Services` action method, add statements to create a gRPC client and call the `Greet` method, as shown in the following code:

```
try
{
  using (GrpcChannel channel =
    GrpcChannel.ForAddress("https://localhost:5006"))
  {
    Greeter.GreeterClient greeter = new(channel);
    HelloReply reply = await greeter.SayHelloAsync(
      new HelloRequest { Name = "Henrietta" });
    ViewData["greeting"] = "Greeting from gRPC service: " + reply.Message;
  }
}
catch (Exception)
{
  _logger.LogWarning($"Northwind.gRPC service is not responding.");
}
```

8. In `Views/Home`, in `Services.cshtml`, add code to render the greeting directly below the title, before the products are rendered, as shown in the following markup:

```
@if (ViewData["greeting"] != null)
{
  <p class="alert alert-primary">@ViewData["greeting"]</p>
}
```

Good Practice: If you clean a gRPC project then you will lose the automatically generated types and see compile errors. To recreate them, simply make any change to a `.proto` file or close and reopen the project/solution.

Testing a gRPC client to the gRPC service

Now we can start the gRPC service and see if the Northwind MVC website can call it successfully:

1. Optionally, start the `Minimal.WebApi` project without debugging.
2. Optionally, start the `Northwind.OData` project without debugging.
3. Optionally, start the `Northwind.GraphQL` project without debugging.
4. Start the `Northwind.gRPC` service project without debugging.

5. Start the `Northwind.Mvc` project.
6. Navigate to the **Services** page: `https://localhost:5001/home/services`.
7. Note the greeting on the services page, as shown in *Figure 18.6*:

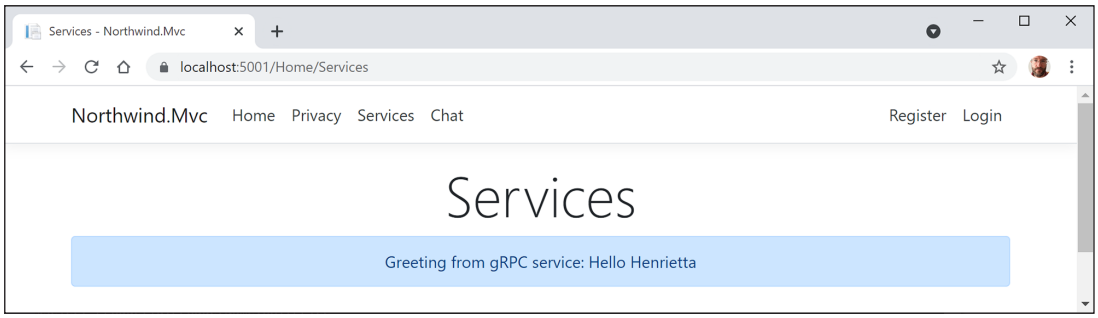


Figure 18.6: Services page after calling the gRPC service to get a greeting

8. View the command prompt or terminal for the gRPC service and note the info messages that indicate an `HTTP/2 POST` was processed by the `greet.Greeter/SayHello` endpoint in about 41ms, as shown in the following output:

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:5006/greet.Greeter/
SayHello application/grpc -
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/2 POST https://localhost:5006/greet.Greeter/
SayHello application/grpc - - 200 - application/grpc 41.3434ms
```

9. Close Chrome and shut down the web servers.

Implementing a gRPC service for an EF Core model

Now we will add support for working with the Northwind database to the gRPC service:

1. In the `Northwind.gRPC` project, add a project reference to the Northwind database context project for either SQLite or SQL Server, as shown in the following markup:

```
<ItemGroup>
  <!-- change Sqlite to SqlServer if you prefer -->
  <ProjectReference Include=
"..\Northwind.Common.DataContext.Sqlite\Northwind.Common.DataContext.
Sqlite.csproj" />
</ItemGroup>
```

2. In the `Northwind.gRPC` project, in the `Protos` folder, add a new file (the item template is named **Protocol Buffer File** in Visual Studio) named `shipper.proto`, as shown in the following code:

```
syntax = "proto3";

option csharp_namespace = "Northwind.gRPC";

package shipr;

service Shipr {
  rpc GetShipper (ShipperRequest) returns (ShipperReply);
}

message ShipperRequest {
  int32 shipperId = 1;
}

message ShipperReply {
  int32 shipperId = 1;
  string companyName = 2;
  string phone = 3;
}
```

3. Open the project file and add an entry to include the `shipper.proto` file, as shown highlighted in the following markup:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
  <Protobuf Include="Protos\shipper.proto" GrpcServices="Server" />
</ItemGroup>
```

4. Build the `Northwind.gRPC` project.
5. In the `Services` folder, add a new class file named `ShipperService.cs`, and modify its contents to define a shipper service that uses the Northwind database context to return shippers, as shown in the following code:

```
using Grpc.Core; // ServerCallContext
using Packt.Shared; // NorthwindContext, Shipper

namespace Northwind.gRPC.Services;

public class ShipperService : Shipr.ShiprBase
{
  private readonly ILogger<ShipperService> _logger;
  private readonly NorthwindContext db;

  public ShipperService(ILogger<ShipperService> logger,
```

```
    NorthwindContext db)
  {
    _logger = logger;
    this.db = db;
  }

  public override async Task<ShipperReply> GetShipper(
    ShipperRequest request, ServerCallContext context)
  {
    return ToShipperReply(
      await db.Shippers.FindAsync(request.ShipperId));
  }

  private ShipperReply ToShipperReply(Shipper? shipper)
  {
    return new ShipperReply
    {
      ShipperId = shipper?.ShipperId ?? 0,
      CompanyName = shipper?.CompanyName ?? string.Empty,
      Phone = shipper?.Phone ?? string.Empty
    };
  }
}
```

6. In `Program.cs`, import the namespace for the Northwind database context, as shown in the following code:

```
using Packt.Shared; // AddNorthwindContext extension method
```

7. In the section that configures services, add a call to register the Northwind database context, as shown in the following code:

```
builder.Services.AddNorthwindContext();
```

8. In the section that configures the HTTP pipeline, after the call to register the Greeter service, add a statement to register the shipper service, as shown in the following code:

```
app.MapGrpcService<ShipperService>();
```

Implementing a gRPC client for an EF Core model

Now we can add client capabilities to the Northwind MVC website:

1. Copy the `shipper.proto` file from the `Protos` folder in the `Northwind.gRPC` project to the `Protos` folder in the `Northwind.Mvc` project. (Hold down *Ctrl* or *Cmd* while dragging and dropping if you use Visual Studio Code.)

2. In the `Northwind.Mvc` project, in `shipper.proto`, modify the namespace to match the namespace for the current project so that the automatically generated classes will be in the same namespace, as shown in the following code:

```
option csharp_namespace = "Northwind.Mvc";
```

3. In the `Northwind.Mvc` project file, add an entry to register the `.proto` file as being used on the client side, as shown highlighted in the following markup:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
  <Protobuf Include="Protos\shipper.proto" GrpcServices="Client" />
</ItemGroup>
```

4. In the `Controllers` folder, in `HomeController.cs`, in the `Services` action method, add statements to call the Shipper gRPC service, as shown in the following code:

```
try
{
  using (GrpcChannel channel =
    GrpcChannel.ForAddress("https://localhost:5006"))
  {
    Shipr.ShiprClient shipr = new(channel);

    ShipperReply reply = await shipr.GetShipperAsync(
      new ShipperRequest { ShipperId = 3 });

    ViewData["shipr"] = new Shipper
    {
      ShipperId = reply.ShipperId,
      CompanyName = reply.CompanyName,
      Phone = reply.Phone
    };
  }
}
catch (Exception)
{
  _logger.LogWarning($"Northwind.gRPC service is not responding.");
}
```

5. In `Views/Home`, in `Services.cshtml`, add code to render the shipper details after the greeting, as shown in the following markup:

```
@if (ViewData["shipr"] != null)
{
  Shipper? shipper = ViewData["shipr"] as Shipper;
  <p class="alert alert-danger">
    ShipperId: @shipper?.ShipperId, CompanyName: @shipper?.CompanyName,
Phone: @shipper?.Phone
```

```
    </p>
}
```

6. Optionally, start the `Minimal.WebApi` project without debugging.
7. Optionally, start the `Northwind.OData` project without debugging.
8. Optionally, start the `Northwind.GraphQL` project without debugging.
9. Start the `Northwind.gRPC` service project without debugging.
10. Start the `Northwind.Mvc` project.
11. Navigate to the **Services** page: `https://localhost:5001/home/services`.
12. Note the shipper information on the services page, as shown in *Figure 18.7*:

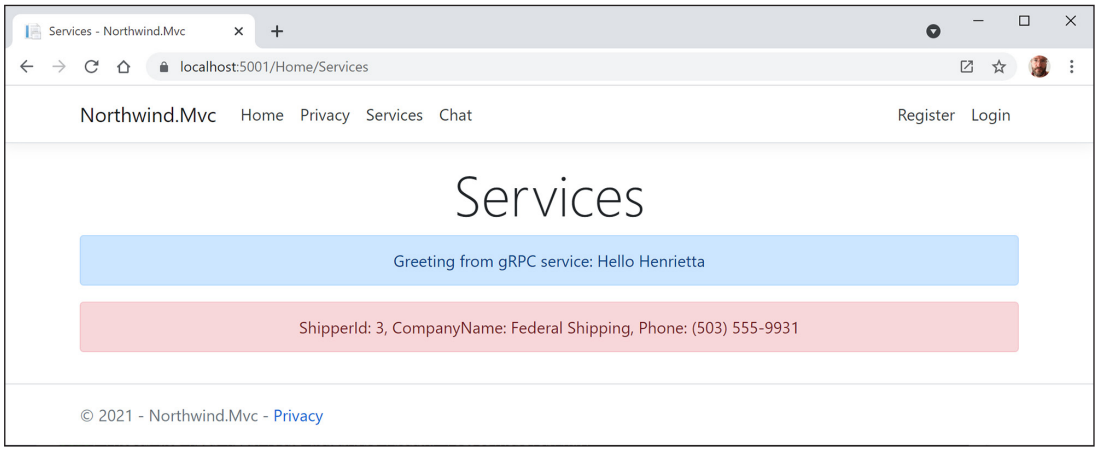


Figure 18.7: Services page after calling the gRPC service to get a shipper

13. Close Chrome and shut down the web servers.

Implementing real-time communication using SignalR

The web is great for building general-purpose websites and services, but it was not designed for specialized scenarios that require a web page to be instantaneously updated with new information as it becomes available.

Understanding the history of real-time communication on the web

To understand the benefits of SignalR, it helps to know the history of HTTP and how organizations worked to make it better for real-time communication between clients and servers.

In the early days of the Web in the 1990s, browsers had to make a full-page HTTP `GET` request to the web server to get fresh information to show to the visitor.

Understanding XMLHttpRequest

In late 1999, Microsoft released Internet Explorer 5.0 with a component named **XMLHttpRequest** that could make asynchronous HTTP calls in the background. This, alongside **dynamic HTML** (**DHTML**), allowed parts of the web page to be updated with fresh data smoothly.

The benefits of this technique were obvious and soon all browsers added the same component.

Understanding AJAX

Google took maximum advantage of this capability to build clever web applications such as Google Maps and Gmail. A few years later, the technique became popularly known as **Asynchronous JavaScript and XML** (**AJAX**).

AJAX still uses HTTP to communicate, however, and that has limitations:

- First, HTTP is a request-response communication protocol, meaning that the server cannot push data to the client. It must wait for the client to make a request.
- Second, HTTP request and response messages have headers with lots of potentially unnecessary overhead.
- Third, HTTP typically requires a new underlying TCP connection to be created on each request.

Understanding WebSocket

WebSocket is full duplex, meaning that either the client or server can initiate communicating new data. WebSocket uses the same TCP connection for the lifecycle of the connection. It is also more efficient in the message sizes that it sends because they are minimally framed with 2 bytes.

WebSocket works over HTTP ports `80` and `443` so it is compatible with the HTTP protocol and the WebSocket handshake uses the HTTP `Upgrade` header to switch from the HTTP protocol to the WebSocket protocol.

Modern web apps are expected to deliver up-to-date information. Live chat is a canonical example, but there are lots of potential applications, from stock prices to games.

Whenever you need the server to push updates to the web page, you need a web-compatible, real-time communication technology. WebSocket could be used but it is not supported by all clients.

Introducing SignalR

ASP.NET Core SignalR is an open source library that simplifies adding real-time web functionality to apps by being an abstraction over multiple underlying communication technologies, which allows you to add real-time communication capabilities using C# code.

The developer does not need to understand or implement the underlying technology used, and SignalR will automatically switch between underlying technologies depending on what the visitor's web browser supports. For example, SignalR will use WebSocket when it's available, and gracefully falls back on other technologies such as AJAX long polling when it isn't, while your application code stays the same.

SignalR is an API for server-to-client **remote procedure calls** (**RPCs**). The RPCs call JavaScript functions on clients from server-side .NET code. SignalR has hubs to define the pipeline and handles the message dispatching automatically using two built-in hub protocols: JSON and a binary one based on MessagePack.

On the server side, SignalR runs everywhere that ASP.NET Core runs: Windows, macOS, or Linux servers. SignalR supports the following client platforms:

- JavaScript clients for current browsers including Chrome, Firefox, Safari, Edge, and Internet Explorer 11.
- .NET clients including Blazor and Xamarin for Android and iOS mobile apps.
- Java 8 and later.

Designing method signatures

When designing the method signatures for a SignalR service, it is best to define methods with a single object parameter rather than multiple simple type parameters. For example, define a class with multiple properties to use as the type of a single parameter instead of passing multiple `string` values, as shown in the following code:

```
// bad practice
public void SendMessage(string to, string body)

// better practice
public class Message
{
  public string To { get; set; }
  public string Body { get; set; }
}

public void SendMessage(Message message)
```

The reason is that it allows future changes like adding a message title. For the bad practice example, a third `string` parameter named `title` would need to be added, and existing clients would get errors because they are not sending the extra `string` value. But using the good practice example will not break the method signature so existing clients can continue to call it as before the change. On the server side, the extra `title` property will just have a `null` value that can be checked for and perhaps set to a default value.

Building a live communication service using SignalR

The SignalR server library is included in ASP.NET Core. But the JavaScript client library is not automatically included in the project. We will use the **Library Manager CLI** to get the client library from **unpkg**, a content delivery network (CDN) that can deliver anything found in Node.js package manager.

Let's add a SignalR server-side hub and client-side JavaScript to the Northwind MVC project to implement a chat feature to allows visitors to send messages to everyone currently using the website, to dynamically defined groups, or to a single specified user.

Good Practice: In a production solution it would be better to host the SignalR hub in a separate web project so that it can be hosted and scaled independently from the rest of the website. Live communication can often put excessive load on a website.

Defining some shared models

First, we will define two shared models that can be used on both the server-side and client-side .NET projects that will work with our chat service:

1. In the `Northwind.Common` project, add a class file named `RegisterModel.cs`, and modify its contents to define a model for registering a username and the groups that they want to belong to, as shown in the following code:

```
namespace Northwind.Chat.Models;

public class RegisterModel
{
  public string? Username { get; set; }
  public string? Groups { get; set; }
}
```

2. In the `Northwind.Common` project, add a class file named `MessageModel.cs`, and modify its contents to define a message model with properties for who the message is sent to and their type (user, group, or everyone) and who the message was sent from, and the message body, as shown in the following code:

```
namespace Northwind.Chat.Models;

public class MessageModel
{
  public string? To { get; set; }
  public string? ToType { get; set; }
  public string? From { get; set; }
  public string? Body { get; set; }
}
```

Enabling a server-side SignalR hub

Next, we will enable a SignalR hub on the server side of the Northwind MVC project:

1. In the `Northwind.Mvc` project, add a reference to the `Northwind.Common` project, if you did not add the project reference earlier.
2. In the `Northwind.Mvc` project, add a `Hubs` folder.
3. In the `Hubs` folder, add a class file named `ChatHub.cs`, and modify its contents to inherit from the `Hub` class, and implement two methods that can be called by a client, as shown in the following code:

```
using Microsoft.AspNetCore.SignalR; // Hub
using Northwind.Chat.Models; // RegisterModel, MessageModel

namespace Northwind.Mvc.Hubs;

public class ChatHub : Hub
{
  // a new instance of ChatHub is created to process each method so
  // we must store usernames and their connectionids in a static field
  private static Dictionary<string, string> users = new();

  public async Task Register(RegisterModel model)
  {
    // add to or update dictionary with username and its connectionId
    users[model.Username] = Context.ConnectionId;

    foreach (string group in model.Groups.Split(','))
    {
      await Groups.AddToGroupAsync(Context.ConnectionId, group);
    }
  }

  public async Task SendMessage(MessageModel command)
  {
    MessageModel reply = new()
    {
```

```
        From = command.From,
        Body = command.Body
      };

      IClientProxy proxy;

      switch (command.ToType)
      {
        case "User":
          string connectionId = users[command.To];
          reply.To = $"{command.To} [{connectionId}]";
          proxy = Clients.Client(connectionId);
          break;

        case "Group":
          reply.To = $"Group: {command.To}";
          proxy = Clients.Group(command.To);
          break;

        default:
          reply.To = "Everyone";
          proxy = Clients.All;
          break;
      }

      await proxy.SendAsync(
        method: "ReceiveMessage", arg1: reply);
    }
  }
```

Note the following:

- `ChatHub` has two methods that a client can call: `Register` and `SendMessage`.
- `Register` has a single parameter of type `RegisterModel`. The username and its connection Id are stored in the static dictionary so that the username can be used to look up the connection Id later and send messages directly to that one user.
- `SendMessage` has a single parameter of type `MessageModel`. The method creates an instance of the `MessageModel` class that will be the message it sends to one or more clients. Then it switches based on the type of recipient. For a user, it looks up the connection Id using the username and then calls the `Client` method to get a proxy that will communicate just with that one client. For a group, it calls the `Group` method to get a proxy that will communicate with just the members of that group. In all other cases, it calls the `All` method to get a proxy that will communicate with every client. Finally, it sends the message asynchronously using the proxy.

4. In `Program.cs`, import the namespace for your SignalR hub, as shown in the following code:

```
using Northwind.Mvc.Hubs; // ChatHub
```

5. In the section that configures services, add a statement to add support for SignalR to the services collection, as shown in the following code:

```
builder.Services.AddSignalR();
```

6. In the section that configures the HTTP pipeline, after the call to map Razor Pages, add a statement to map the relative URL path `/chat` to your SignalR hub, as shown in the following code:

```
app.MapHub<ChatHub>("/chat");
```

Adding the SignalR client-side JavaScript library

Next, we will add the SignalR client-side JavaScript library so that we can use it on a web page:

1. Open a command prompt or terminal for the `Northwind.Mvc` project.
2. Install the Library Manager CLI tool, as shown in the following command:

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

This tool might already be installed. To update it to the latest version, repeat the command but replace `install` with `update`.

3. Note the success message, as shown in the following output:

```
You can invoke the tool using the following command: libman
Tool 'microsoft.web.librarymanager.cli' (version '2.1.113') was
successfully installed.
```

4. Add the `signalr.js` and `signalr.min.js` libraries to the project from the `unpkg` source, as shown in the following command:

```
libman install @microsoft/signalr@latest -p unpkg -d wwwroot/js/signalr
--files dist/browser/signalr.js --files dist/browser/signalr.min.js
```

5. Note the success message, as shown in the following output:

```
Downloading file https://unpkg.com/@microsoft/signalr@latest/dist/browser/
signalr.js...
Downloading file https://unpkg.com/@microsoft/signalr@latest/dist/browser/
signalr.min.js...
wwwroot/js/signalr/dist/browser/signalr.js written to disk
wwwroot/js/signalr/dist/browser/signalr.min.js written to disk
Installed library "@microsoft/signalr@latest" to "wwwroot/js/signalr"
```

Visual Studio has a GUI for adding client-side JavaScript libraries. To use it, right-click a web project and then navigate to **Add** | **Client Side Libraries**.

Adding a chat page to the Northwind MVC website

Next, we will create a chat page:

1. In the `Controllers` folder, open `HomeController.cs`, and add a new action method for chat, as shown in the following code:

```
public IActionResult Chat()
{
  return View();
}
```

2. In `Views/Shared`, in `_Layout.cshtml`, add a new nav item after the `Services` nav item that goes to a chat page, as shown in the following markup:

```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area=""
       asp-controller="Home" asp-action="Chat">Chat</a>
</li>
```

3. In `Views/Home`, add a new empty view named `Chat.cshtml`, and modify its contents, as shown in the following markup:

```
@{
  ViewData["Title"] = "Chat";
}
<div class="container">
  <h1>@ViewData["Title"]</h1>
  <div class="row">
    <div class="col-12">
      <h2>Register</h2>
    </div>
  </div>
  <div class="row">
    <div class="col-4">My name</div>
    <div class="col-8"><input type="text" id="from" /></div>
  </div>
  <div class="row">
    <div class="col-4">My groups</div>
    <div class="col-8"><input type="text" id="groups" value="Sales,IT"
/></div>
  </div>
  <div class="row">
    <div class="col-12">
      <input type="button" id="registerButton" value="Register" />
```

```
    </div>
  </div>
  <div class="row">
    <div class="col-12">
      <h2>Message</h2>
    </div>
  </div>
  <div class="row">
    <div class="col-4">To type</div>
    <div class="col-8">
      <select id="toType">
        <option selected>Everyone</option>
        <option>Group</option>
        <option>User</option>
      </select>
    </div>
  </div>
  <div class="row">
    <div class="col-4">To</div>
    <div class="col-8"><input type="text" id="to" /></div>
  </div>
  <div class="row">
    <div class="col-4">Body</div>
    <div class="col-8"><input type="text" id="body" /></div>
  </div>
  <div class="row">
    <div class="col-12">
      <input type="button" id="sendButton" value="Send" />
    </div>
  </div>
  <div class="row">
    <div class="col-12">
      <hr />
    </div>
  </div>
  <div class="row">
    <div class="col-12">
      <h2>Messages received</h2>
    </div>
  </div>
  <div class="row">
    <div class="col-12">
      <ul id="messages"></ul>
    </div>
  </div>
</div>
<script src="~/js/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

Note the following:

- There are three sections on the page: **Register**, **Message**, and **Messages received**.
- The **Register** section has two inputs for the visitor's name and a comma-separated list of the groups that they want to be a member of, and a button to click to register.
- The **Message** section has three inputs for the type of recipient, the recipient's name, and the body of the message, and a button to click to send the message.
- The **Messages received** section has an unordered list element that will be dynamically populated with list items when a message is received.
- The two script elements for the SignalR client-side library are followed by the implementation of the chat client.

4. In `wwwroot/js`, add a new JavaScript file named `chat.js`, and modify its contents, as shown in the following code:

```
"use strict";

var connection = new signalR.HubConnectionBuilder()
  .withUrl("/chat").build();

document.getElementById("registerButton").disabled = true;
document.getElementById("sendButton").disabled = true;

connection.start().then(function () {
  document.getElementById("registerButton").disabled = false;
  document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
  return console.error(err.toString());
});

connection.on("ReceiveMessage", function (received) {
  var li = document.createElement("li");
  document.getElementById("messages").appendChild(li);
  // note the use of backtick ` to enable a formatted string
  li.textContent =
    `${received.from} says ${received.body} (sent to ${received.to})`;
});

document.getElementById("registerButton").addEventListener("click",
  function (event) {
  var registermodel = {
    username: document.getElementById("from").value,
    groups: document.getElementById("groups").value
  };
  connection.invoke("Register", registermodel).catch(function (err) {
```

```
    return console.error(err.toString());
  });
  event.preventDefault();
});

document.getElementById("sendButton").addEventListener("click",
  function (event) {
  var messageToSend = {
    to: document.getElementById("to").value,
    toType: document.getElementById("toType").value,
    from: document.getElementById("from").value,
    body: document.getElementById("body").value
  };
  connection.invoke("SendMessage", messageToSend).catch(function (err) {
    return console.error(err.toString());
  });
  event.preventDefault();
});
```

Note the following:

- The script creates a SignalR hub connection builder specifying the relative URL path to the chat hub on the server/chat.
- The script disables the **Register** and **Send** buttons until the connection is successfully established to the server-side hub.
- When the connection gets a `ReceiveMessage` call from the server-side hub, it adds a list item element to the `messages` unordered list. The content of the list item contains details of the message like `from`, `to`, and `body`. Note JavaScript uses camelCasing.
- A click event handler is added to the **Register** button that creates a register model with the username and their groups and then invokes the `Register` method on the server side.
- A click event handler is added to the **Send** button that creates a message model with the from, to, type, and message body, and then invokes the `SendMessage` method on the server side.

Testing the chat feature

Now we are ready to try sending chat messages between multiple website visitors:

1. Start the `Northwind.Mvc` project website.
2. Start Chrome.
3. Navigate to `https://localhost:5001/home/chat`.
4. Enter `Alice` for the name, `Sales,IT` for the groups, and then click **Register**.
5. Open a new browser window or start another browser like Firefox or Edge.

6. Navigate to `https://localhost:5001/home/chat`.
7. Enter `Bob` for the name, `Sales` for the groups, and then click **Register**.
8. Open a new browser window or start another browser like Firefox or Edge.
9. Navigate to `https://localhost:5001/home/chat`.
10. Enter `Charlie` for the name, `IT` for the groups, and then click **Register**.
11. Arrange the browser windows so that you can see all three simultaneously.
12. In Alice's browser, select **Group**, enter `Sales`, enter `Sell more!` and then click **Send**.
13. Note that Alice and Bob receive the message, as shown in *Figure 18.8*:

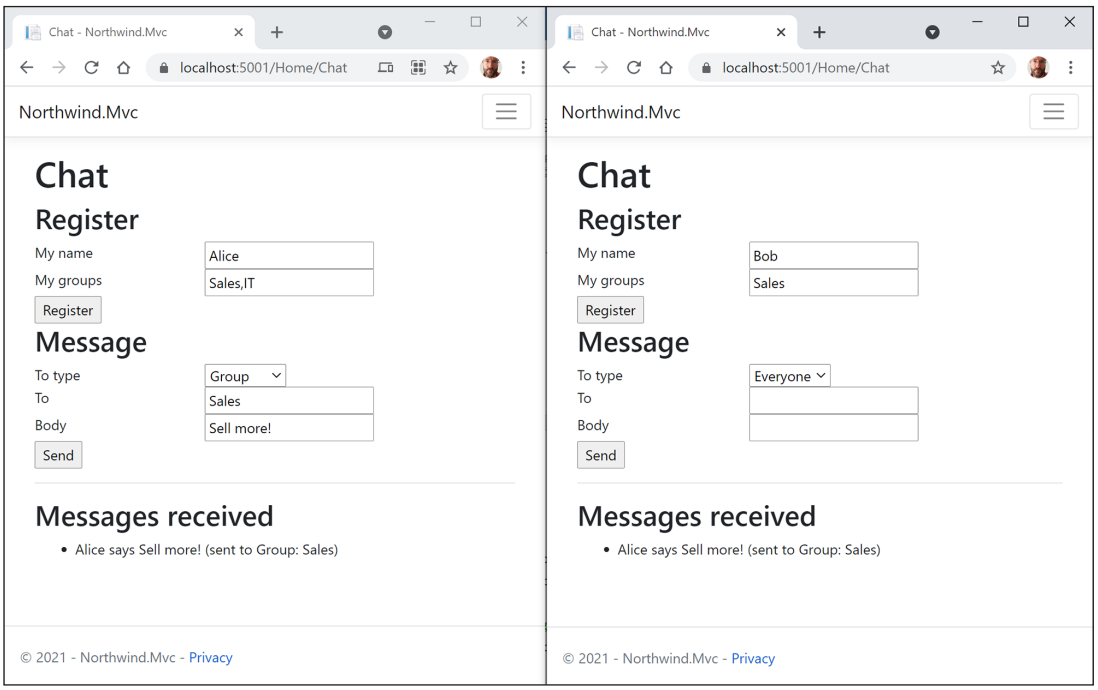


Figure 18.8: Alice sends a message to the Sales group

14. In Bob's browser, select **Group**, enter `IT`, enter `Fix more bugs!` and then click **Send**.
15. Note that Alice and Charlie receive the message, as partially shown in *Figure 18.9*:

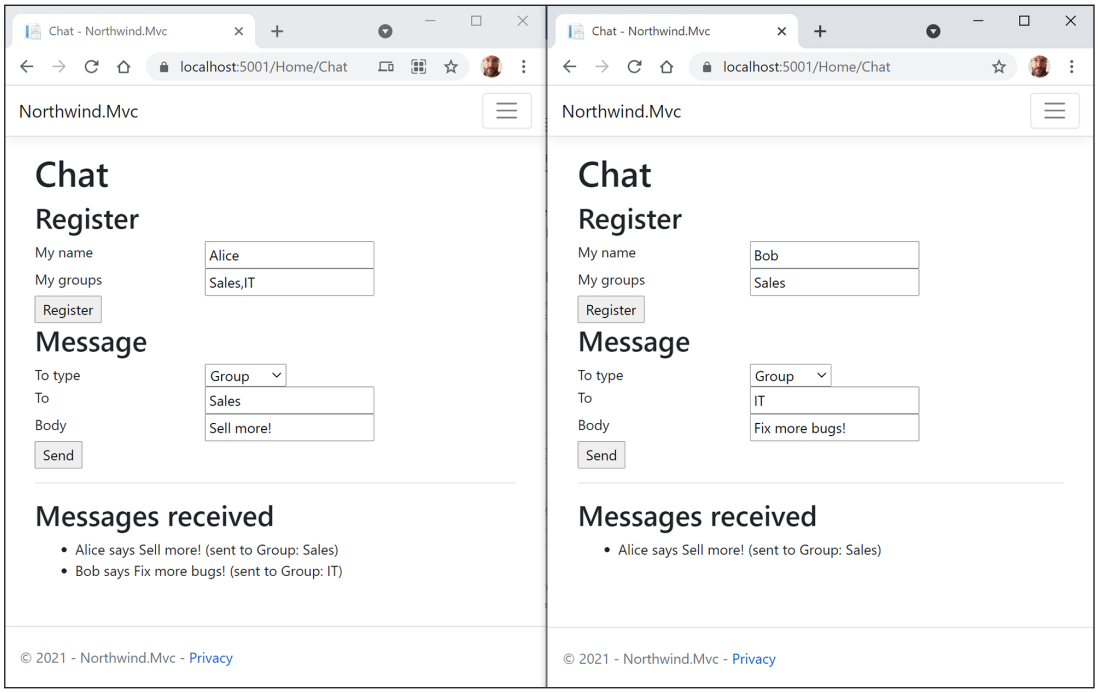


Figure 18.9: Bob sends a message to the IT group

16. In Alice's browser, select **User**, enter `Bob`, enter `Bonjour Bob!` and then click **Send**.

17. Note that only Bob receives the message, as shown in *Figure 18.10*:

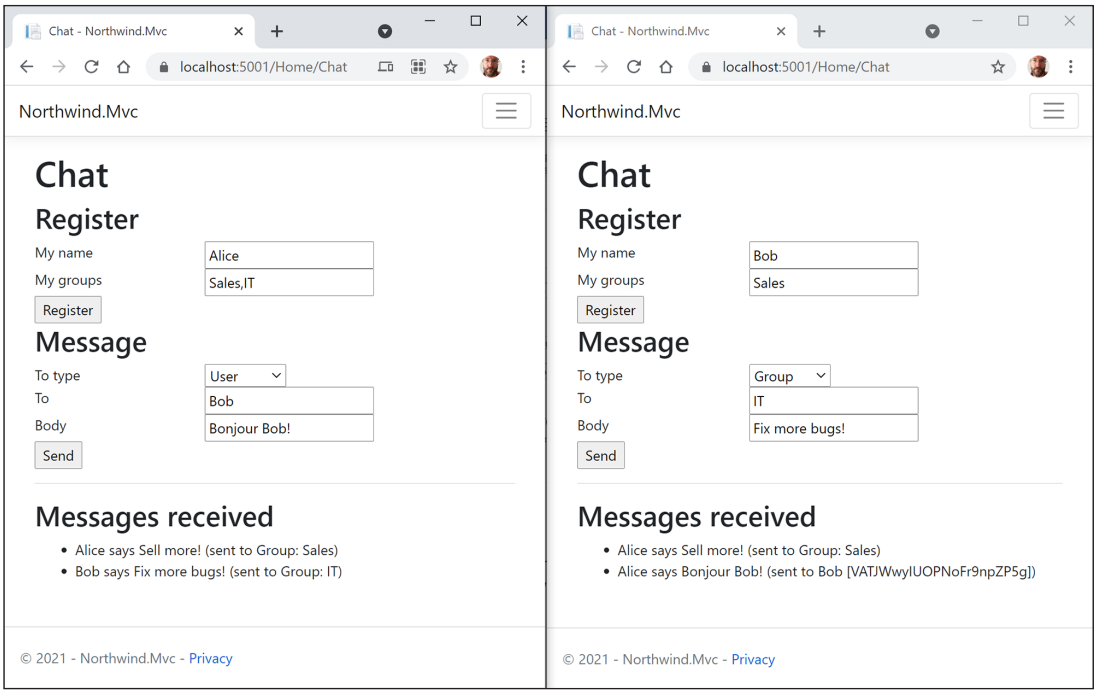


Figure 18.10: Alice sends a message to Bob

18. In Charlie's browser, leave **To type** set to **Everyone**, leave **To** blank, enter any message and then click **Send**, and note that everyone receives the message.
19. Close the browsers and shut down the web server.

Building a console app chat client

Now, let's create a .NET client for SignalR. We will use a console app, although any .NET project type would need the same package reference and implementation code:

1. Use your preferred code editor to add a new project, as defined in the following list:
    1. Project template: **Console Application** / `console`
    2. Workspace/solution file and folder: `PracticalApps`
    3. Project file and folder: `Northwind.SignalR.ConsoleClient`
2. Add a package reference for the ASP.NET Core SignalR client and a project reference for `Northwind.Common`, as shown in the following markup:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.SignalR.Client"
                    Version="6.0.0" />
</ItemGroup>
<ItemGroup>
```

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

3. In `Program.cs`, import namespaces for working with SignalR as a client and the chat models, and then add statements to create a hub connection, prompt the user to enter a username and groups to register with, and then listen for received messages, as shown in the following code:

```
using Microsoft.AspNetCore.SignalR.Client; // HubConnection
using Northwind.Chat.Models; // RegisterModel, MessageModel

using static System.Console;

Write("Enter a username: ");
string? username = ReadLine();

Write("Enter your groups: ");
string? groups = ReadLine();

HubConnection hubConnection = new HubConnectionBuilder()
  .WithUrl("https://localhost:5001/chat")
  .Build();

hubConnection.On<MessageModel>("ReceiveMessage", message =>
{
  WriteLine($"{message.From} says {message.Body} (sent to {message.To})");
});

await hubConnection.StartAsync();

WriteLine("Successfully started.");

RegisterModel registration = new()
{
  Username = username,
  Groups = groups
};

await hubConnection.InvokeAsync("Register", registration);

WriteLine("Successfully registered.");
WriteLine("Listening... (press ENTER to stop.)");
ReadLine();
```

4. Start the `Northwind.Mvc` project website without debugging.

5. Start Chrome.
6. Navigate to `https://localhost:5001/home/chat`.
7. Enter `Alice` for the name, `Sales,IT` for the groups, and then click **Register**.
8. Start the `Northwind.SignalR.ConsoleClient` project, and then enter your name and the groups `Sales,Admins`.
9. Arrange the browser and console app windows so that you can see both simultaneously.
10. In Alice's browser, in the **Message** section, select **Group**, enter `Sales`, enter `Go team!`, click **Send**, and note that Alice and you receive the message.
11. Try sending messages only to yourself, only to members of the Admins group, and to everyone, as shown in *Figure 18.11*:

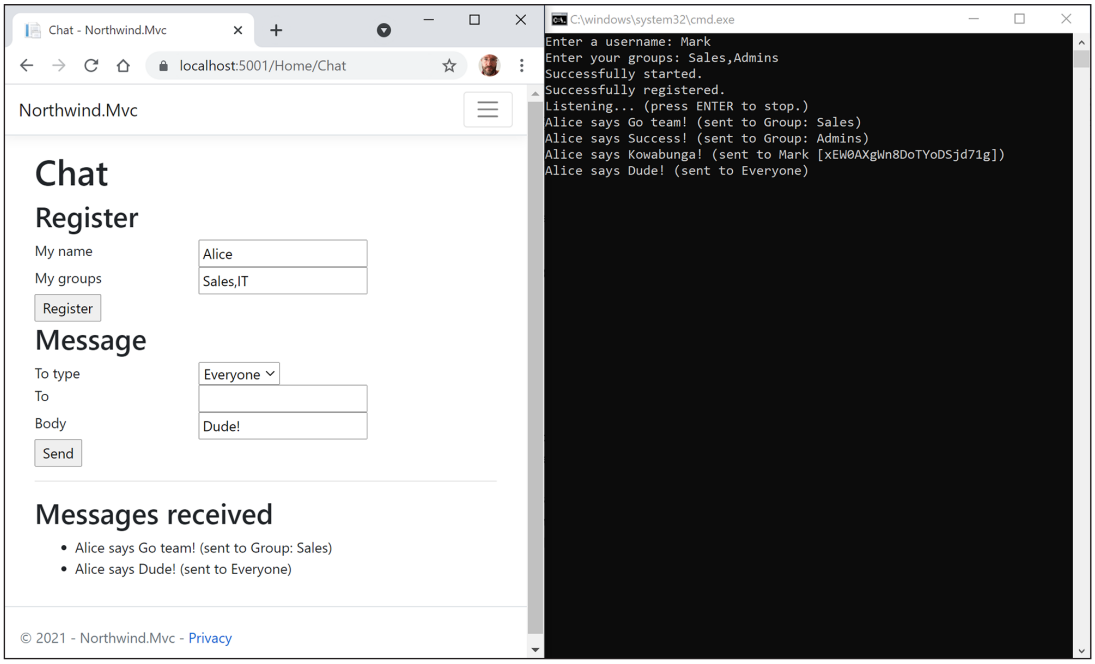


Figure 18.11: Alice sends messages to different types of recipient

12. In the console app, press *Enter* to stop it.
13. Close Chrome and shut down the web server.

Implementing serverless services using Azure Functions

Azure Functions is an event-driven serverless compute platform. You can build and debug locally and later deploy to the Microsoft Azure cloud. Azure Functions can be implemented in many languages, not just C# and .NET. It has extensions for Visual Studio and Visual Studio Code.

Why would you need to create a service without a server? Serverless does not literally mean without a server. What serverless means is without a permanently running server, usually for most of the time.

For example, organizations often have business functions that only need to run once per month, or on an ad hoc basis. Perhaps the organization prints checks (cheques) to pay its employees at the end of the month. Those checks might need the salary amounts converted to words to print on the check. A function to convert numbers to words could be implemented as a serverless service.

Azure Functions can be much more than just a single function. They support complex, stateful workflows and event-driven solutions using **Durable Functions**. We will not cover these in this book so if you are interested then you can learn more about them at the following link: `https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp`

Understanding Azure Functions

Azure Functions has a programming model based on triggers and bindings that enable your serverless applications to respond to events and connect to other services like data stores.

Understanding Azure Functions triggers and bindings

Triggers and bindings are key concepts of Azure Functions. Triggers are what make a function execute. Each function must have one and only one trigger. The most common triggers are shown in the following list:

- **HTTP**: this trigger responds to an incoming HTTP request.
- **Queue**: this trigger responds to a message arriving in a queue ready for processing.
- **Timer**: this trigger responds to a time occurring.
- **Event Grid**: this trigger responds when a predefined event occurs.

Bindings allow functions to have inputs and outputs. Each function can have zero, one, or more bindings. Some common bindings are shown in the following list:

- **Blob storage**: read or write to any file stored as a **binary large object** (**BLOB**).
- **Cosmos DB**: read or write documents to a cloud-scale data store.
- **SignalR**: receive or make remote method calls.
- **Queue**: write a message to a queue.
- **SendGrid**: send an email message.
- **Twilio**: send an SMS message.
- **IoT Hub**: write to an internet-connected device.

You can see the full list of supported bindings at the following link: `https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings?tabs=csharp#supported-bindings`

Triggers and bindings are configured differently for different languages. For C# and Java, you decorate methods and parameters with attributes. For the other languages, you configure a file named `function.json`.

Understanding Azure Functions versions and languages

Azure Functions supports four versions of the runtime host and multiple languages, as shown in the following table:

Language	v1	v2	v3	v4
C#, F#	.NET Framework 4.8	.NET Core 2.1	.NET Core 3.1, .NET 5.0[2]	.NET 6.0[2]
JavaScript[1]	Node 6	Node 8, 10	Node 10, 12, 14	
Java	-	Java 8	Java 8, 11	
PowerShell	-	PowerShell Core 6	PowerShell Core 6, 7	
Python	-	Python 3.6, 3.7	Python 3.6, 3.7, 3.8, 3.9	

[1] Azure Functions supports the TypeScript language via transpiling (transforming/compiling) to JavaScript.

[2] .NET 5.0 is only supported in the isolated hosting model because it is a Current release. .NET 6.0 supports both isolated and in-process because it is a Long Term Support release.

In this book, we will only look at implementing Azure Functions using C# and .NET.

Understanding Azure Functions hosting models

Azure Functions has two hosting models: in-process and isolated.

- **In-process**: Your function is implemented in a class library that runs in the same process as the host. Your functions are required to run on the same version of .NET as the Azure Functions runtime.
- **Isolated**: Your function is implemented in a console app that runs in its own process. Your function can therefore execute on Current releases like .NET 5.0 that are not supported by the Azure Functions runtime, which only allows LTS releases in-process.

Azure Functions only natively supports one LTS version of .NET. For example, for Azure Functions v3, your function must use .NET Core 3.1 in-process. For Azure Functions v4, your function must use .NET 6.0 in-process. If you create an isolated function, then you can choose any .NET version.

Setting up a local development environment for Azure Functions

First, you will need to install the latest version of **Azure Functions Core Tools**, which at the time of writing is v4, from the following link:

`https://www.npmjs.com/package/azure-functions-core-tools`

Azure Functions Core Tools provide the core runtime and templates for creating functions, which enable local development on Windows, macOS, and Linux using any code editor.

Azure Functions Core Tools is included in the **Azure development** workload of Visual Studio 2022, so you might already have it installed.

Building an Azure Functions project for running locally

Now, we can create an Azure Functions project. Although they can be created in the cloud using the Azure portal, developers will have a better experience creating and running them locally. You can then deploy to the cloud once you have tested your function on your own computer.

Each code editor has a slightly different experience to get started with an Azure Functions project.

Using Visual Studio 2022

If you prefer to use Visual Studio, here are the steps to create an Azure Functions project:

1. Use your preferred code editor to add a new project, as defined in the following list:
    1. Project template: **Azure Functions**
    2. Workspace/solution file and folder: `PracticalApps`
    3. Project file and folder: `Northwind.AzureFuncs`

2. In Visual Studio, choose **.NET 6 (Isolated)**, **Http trigger**, **Storage emulator**, and for **Authorization level**, choose **Anonymous**, and then click **Create**, as partially shown in *Figure 18.12*:

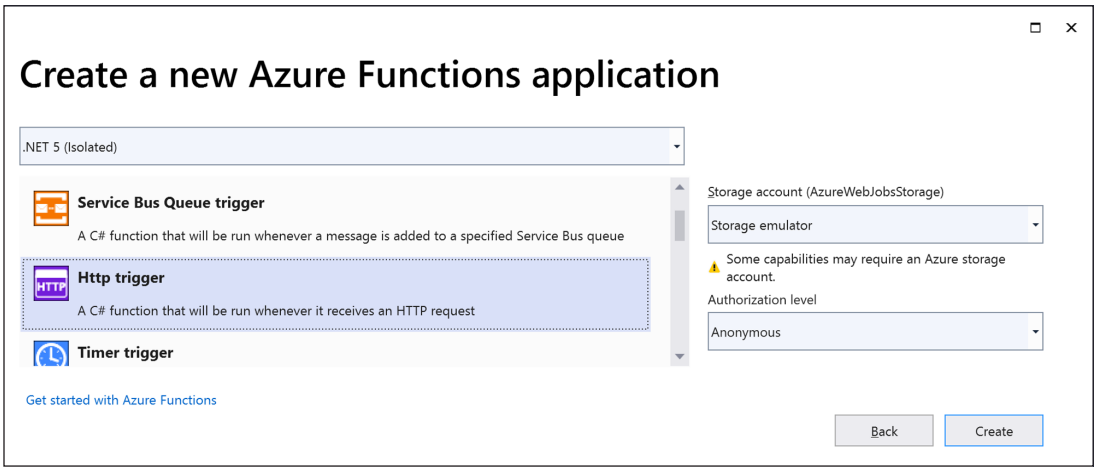


Figure 18.12: Choosing options for your Azure Functions project in Visual Studio 2022

Using Visual Studio Code

If you prefer to use Visual Studio Code, here are the steps to create an Azure Functions project:

1. In Visual Studio Code, navigate to **Extensions** and search for Azure Functions (`ms-azuretools.vscode-azurefunctions`). It has dependencies on two other extensions: Azure Account (`ms-vscode.azure-account`) and Azure Resources (`ms-azuretools.vscode-azureresourcegroups`), so those will be installed too.
2. In the `PracticalApps` folder, create a new folder named `Northwind.AzureFuncs` and add it to the `PracticalApps` workspace.
3. Close the `PracticalApps` workspace and then open the `Northwind.AzureFuncs` folder. (The following steps only work outside a workspace.)
4. In the **Azure** extension, in the **FUNCTIONS** section, click the **Create new project** button, and then select the `Northwind.AzureFuncs` folder, as shown in *Figure 18.13*:

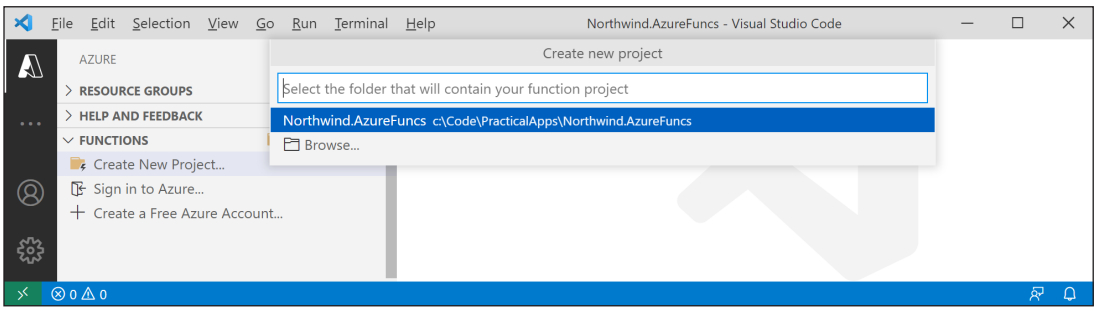


Figure 18.13: Selecting the folder for your Azure Functions project

5. At the prompts, select the following:
    1. Select a language for your function project: **C#**.
    2. Select **.NET 6 LTS** as the .NET runtime, unfortunately not shown in *Figure 18.14* because it was not released at the time of writing:

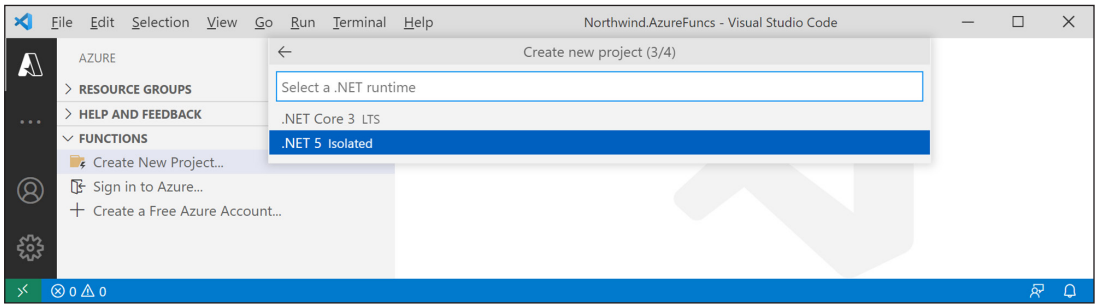


Figure 18.14: Selecting the target .NET runtime for your Azure Functions project

    3. Select a template for your project's first function: **HTTP trigger**.
    4. Provide a function name: `NumbersToWordsFunction`.
    5. Provide a namespace: `Northwind.AzureFuncs`.
    6. Select the authorization level: **Anonymous**.
6. In the Visual Studio Code **File** menu, close the folder.
7. Open the `PracticalApps` workspace.

Using the func CLI

If you prefer to use the command-line and some other code editor, here are the steps to create an Azure Functions project:

1. In the `PracticalApps` folder, create a new folder named `Northwind.AzureFuncs` and add it to the `PracticalApps` workspace.
2. In command prompt or terminal, in the `Northwind.AzureFuncs` folder, create a new Azure Functions project using C#, as shown in the following command:

```
func init --csharp
```

3. In command prompt or terminal, in the `Northwind.AzureFuncs` folder, create a new Azure Functions function using `HTTP trigger`, which can be called anonymously, as shown in the following command:

```
func new --name NumbersToWordsFunction --template "HTTP trigger"
--authlevel "anonymous"
```

4. Optionally, you can start the function locally, as shown in the following command:

```
func start
```

Reviewing the project

Before we write a function, let's review what makes an Azure Functions project:

1. Open the project file, and note the Azure Functions version and the package references needed to implement an Azure Function that responds to HTTP requests, as shown in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <AzureFunctionsVersion>v4</AzureFunctionsVersion>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Sdk.Functions"
      Version="3.0.13" />
  </ItemGroup>
  <ItemGroup>
    <None Update="host.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
    <None Update="local.settings.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
      <CopyToPublishDirectory>Never</CopyToPublishDirectory>
    </None>
  </ItemGroup>
</Project>
```

2. Open the `local.settings.json` file, and note that during local development your project will use local development storage and an isolated process, as shown in the following markup:

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet"
  }
}
```

Implementing the function

Now, we can implement the function to convert numbers into words:

1. If you completed the exercise in *Chapter 8*, *Working with Common .NET Types*, to write a function that converts numbers to words, then use your implementation. If not, then use the class at the following link: `https://github.com/markjprice/cs10dotnet6/blob/master/vscode/PracticalApps/Northwind.AzureFuncs/NumbersToWords.cs`.

2. If you are using Visual Studio, in the `Northwind.AzureFuncs` project, right-click `Function1.cs` and rename it to `NumbersToWordsFunction.cs`.
3. Open `NumbersToWordsFunction.cs` and modify the contents to implement an Azure Function to convert an amount as a number into words, as shown in the following code:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs; // [FunctionName], [HttpTrigger]
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using System.Numerics; // BigInteger
using Packt.Shared; // ToWords extension method
using System.Threading.Tasks; // Task

namespace Northwind.AzureFuncs;

public static class NumbersToWordsFunction
{
  [FunctionName(nameof(NumbersToWordsFunction))]
  public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post")]
    HttpRequest req, ILogger log)
  {
    log.LogInformation($"C# HTTP trigger function processed a request.");

    string amount = req.Query["amount"];

    if (BigInteger.TryParse(amount, out BigInteger number))
    {
      return new OkObjectResult(number.ToWords());
    }
    else
    {
      return new BadRequestObjectResult($"Failed to parse: {amount}");
    }
  }
}
```

Testing the function

Now we can test the function:

1. Start the `Northwind.AzureFuncs` project. If you are using Visual Studio Code, you will need to navigate to the **Run and Debug** pane, make sure that **Attach to .NET Functions** is selected, and then click the **Run** button.

2. Note that **Azure Storage emulator** starts.
3. On Windows, if you see a **Windows Security Alert** from **Windows Defender Firewall**, then click **Allow access**.
4. Note Azure Functions Core Tools hosts your function, usually on port `7071`, as shown in the following output:

```
Azure Functions Core Tools
Core Tools Version:       4.0.3743 Commit hash:
44e84987044afc45f0390191bd5d70680a1c544e  (64-bit)
Function Runtime Version: 4.0.16281

Functions:
        NumbersToWordsFunction: [GET,POST] http://localhost:7071/api/
NumbersToWordsFunction

For detailed output, run func with --verbose flag.
[2021-09-12T18:44:47.499Z] Worker process started and initialized.
[2021-09-12T18:44:51.038Z] Host lock lease acquired by instance ID '000000
00000000000000000011150C3D'.
```

5. Select the URL for your function and copy it to the clipboard.
6. Start Chrome.
7. Paste the URL into the address box, append the query string: `?amount=123456`, and note the successful response, as shown in *Figure 18.15*:

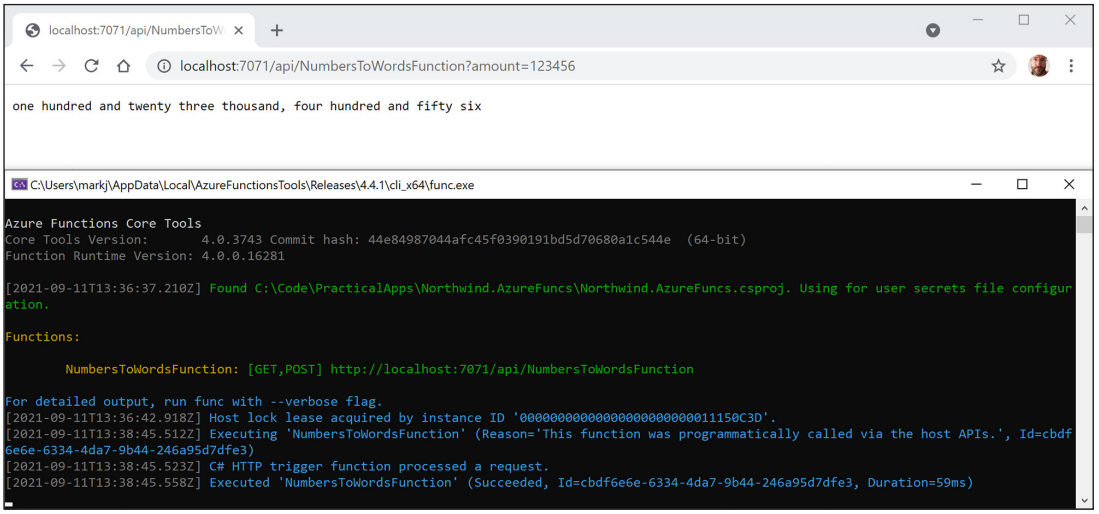


Figure 18.15: A successful call to the Azure Function running locally

8. In the command prompt or terminal, note the function was called successfully, as shown in the following output:

```
[2021-09-14T05:58:27.357Z] Executing 'Functions.NumbersToWordsFunction'
(Reason='This function was programmatically called via the host APIs.',
Id=c2c98c67-bf9f-4121-8f7b-701dbc9c0bad)
[2021-09-14T05:58:27.417Z] C# HTTP trigger function processed a request.
[2021-09-14T05:58:27.461Z] Executed 'Functions.NumbersToWordsFunction'
(Succeeded, Id=c2c98c67-bf9f-4121-8f7b-701dbc9c0bad, Duration=111ms)
```

9. Try calling the function without an amount in the query string, or a non-integer value for the amount, and note the function returns a `400` status code indicating a bad request, as shown in *Figure 18.16*:

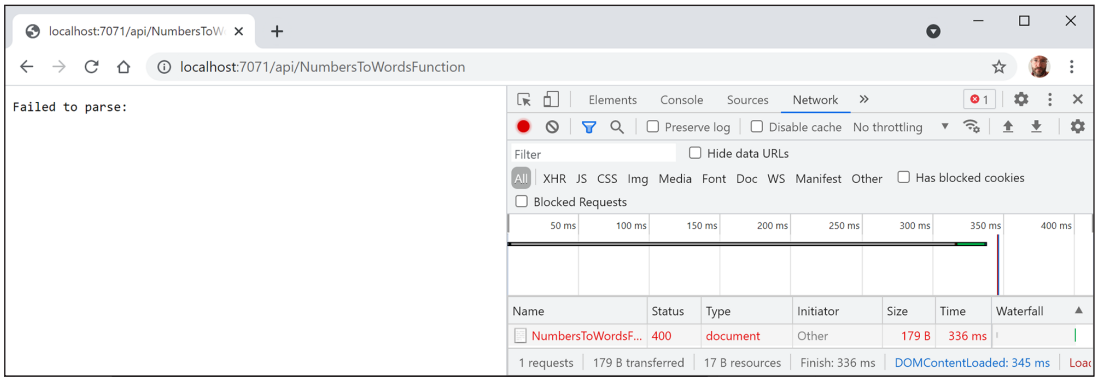


Figure 18.16: A bad request to the Azure Function running locally

10. Close Chrome and shut down the web server (or in Visual Studio Code stop debugging).

Publishing an Azure Functions project to the cloud

Now, let's create a function app and related resources in an Azure subscription, then deploy your function to the cloud and run it there.

If you do not already have an Azure account, then you can sign up for a free one at the following link: `https://azure.microsoft.com/en-us/free/`

Using Visual Studio 2022

Visual Studio has a GUI to publish to Azure:

1. In **Solution Explorer**, right-click the `Northwind.AzureFuncs` project and select **Publish**.
2. Select **Azure** and then click **Next**.
3. Select **Azure Function App (Windows)** and click **Next**.
4. Sign in and enter your credentials.
5. Select your subscription.

6. In the **Function Instance** section, click the **+** button that has a tooltip that says **Create a new Azure Function…**
7. Complete the dialog box, as shown in the following screenshot in *Figure 18.17*:
    1. **Name**: This must be globally unique.
    2. **Subscription name**: Your subscription.
    3. **Resource group**: Create a new resource group to make it easier to delete everything later. I entered `cs10dotnet6projects`.
    4. **Plan Type**: **Consumption** (pay for only what you use).
    5. **Location**: A data center nearest to you. I chose **UK South**.
    6. **Azure Storage**: Create a new account named `cs10dotnet6projects` (or something else that is globally unique—try appending your initials) in a data center nearest to you and choose **Standard – Locally Redundant Storage** for the account type.

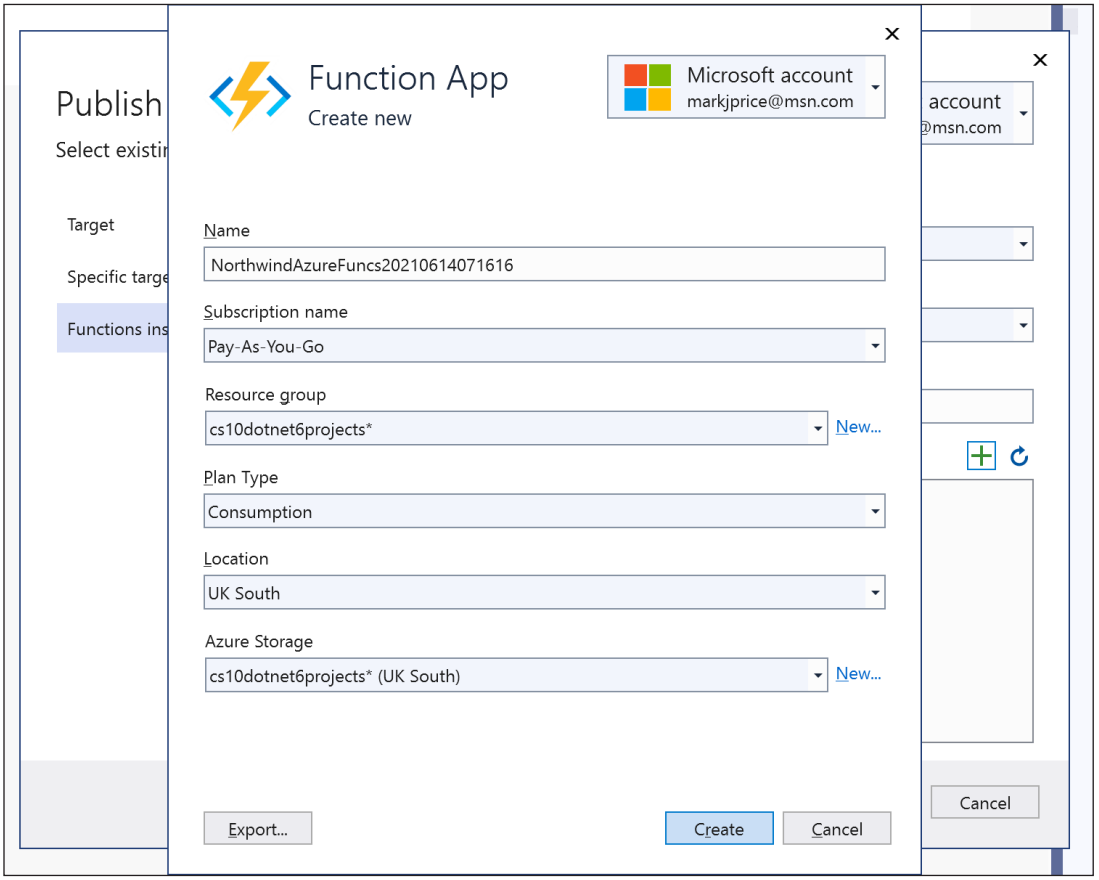


Figure 18.17: Creating a new Azure Function app

8. Click **Create**. This process can take a minute or more.
9. In the **Publish** dialog, click **Finish**.

10. In the **Publish** window, click the **Publish** button, as shown in *Figure 18.18*:

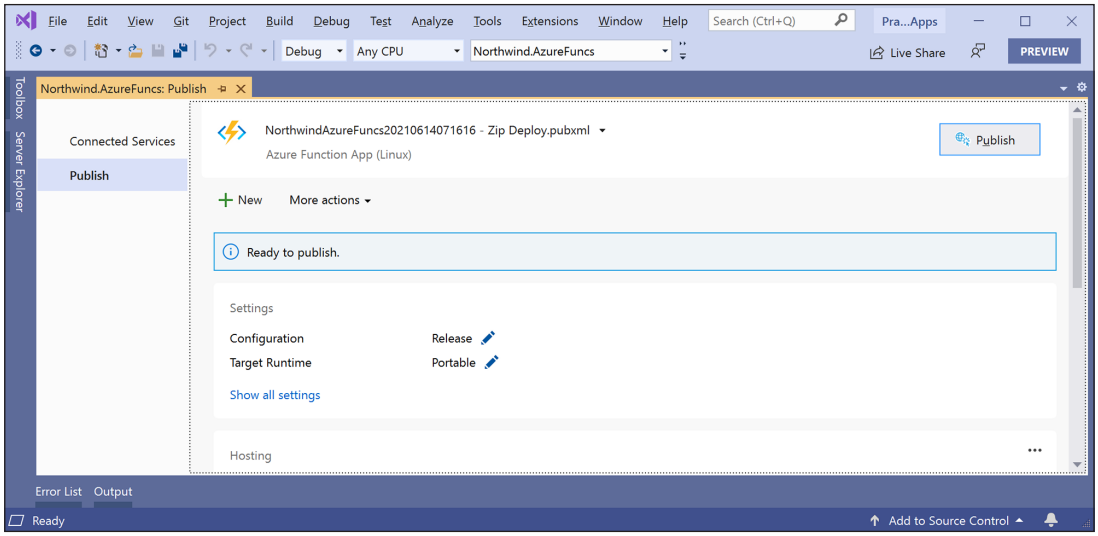


Figure 18.18: An Azure Function app ready to publish

11. Review the output window, as shown in the following publishing output:

```
Build started...
2>------ Publish started: Project: Northwind.AzureFuncs, Configuration:
Release Any CPU ------
2>Northwind.AzureFuncs -> C:\Code\PracticalApps\Northwind.AzureFuncs\bin\
Release\net6.0\Northwind.AzureFuncs.dll
2>Northwind.AzureFuncs -> C:\Code\PracticalApps\Northwind.AzureFuncs\obj\
Release\net6.0\PubTmp\Out\
2>Publishing C:\Code\PracticalApps\Northwind.AzureFuncs\obj\Release\
net6.0\PubTmp\Northwind.AzureFuncs - 20210911153432123.zip to https://
northwindazurefuncs20210911151522.scm.azurewebsites.net/api/zipdeploy...
2>Zip Deployment succeeded.
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
==========
========== Publish: 1 succeeded, 0 failed, 0 skipped ==========
Waiting for function app ready....
Finished waiting for function app to be ready
```

12. Test the function in your browser, as shown in *Figure 18.19*:

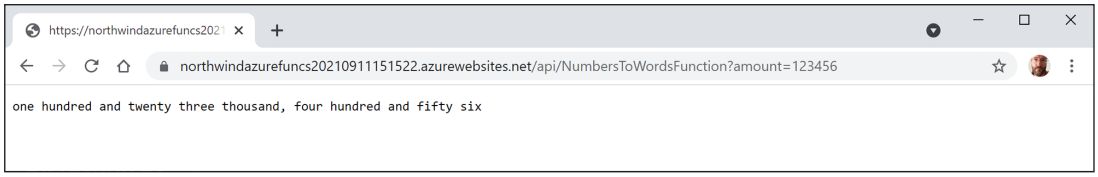


Figure 18.19: Calling the Azure Function in the cloud

Cleaning up Azure resources

You can use the following steps to delete the function app and its related resources to avoid incurring any further costs:

1. In Visual Studio Code, navigate to **View** | **Command Palette**.
2. Search for and select **Azure Functions: Open in portal**.
3. Select your function app.
4. In the Azure portal, in your function app **Overview** blade, select the **Resource Group**.
5. Confirm that it contains only resources that you want to delete.
6. Click **Delete resource group** and accept any other confirmations.

Understanding identity services

Identity services are used to authenticate and authorize users. It is important for these services to implement open standards so that you can integrate disparate systems. Common standards include **OpenID Connect** and **OAuth 2.0**.

A popular free open source implementation of these identity standards is **IdentityServer4**. It enables developers to integrate token-based authentication, single-sign-on, and API access control in websites, services, and applications.

Microsoft has no plans to officially support IdentityServer4 because, "creating and sustaining an authentication server is a full-time endeavor, and Microsoft already has a team and a product in that area, Azure Active Directory, which allows 500,000 objects for free."

You can read the documentation for IdentityServer4 at the following link: `https://identityserver4.readthedocs.io/`.

Summary of choices for specialized services

Use the recommendations for various scenarios as guidance, as shown in the following table:

Scenario	Recommendation
Public services	REST aka HTTP-based services are best for services that need to be publicly accessible, especially if they need to be called from a browser or even a mobile device.
Public data services	OData and GraphQL are both good choices for exposing complex hierarchical data sets that could come from different data stores. OData is designed and supported by Microsoft via official .NET packages. GraphQL is designed by Facebook and supported by third-party packages.
Service-to-services	gRPC is designed for low latency and high throughput communication. gRPC is great for lightweight internal microservices where efficiency is critical.
Point-to-point real-time communication	gRPC has excellent support for bidirectional streaming. gRPC services can push messages in real time without polling. SignalR is also an option for real-time communication of many kinds although it is less efficient than gRPC.
Broadcast real-time communication	SignalR has great support for broadcast real-time communication to many clients.
Polyglot environments	gRPC tooling supports all popular development languages, making gRPC a good choice for multi-language and platform environments.
Network bandwidth-constrained environments	gRPC messages are serialized with Protobuf, a lightweight message format. A gRPC message is always smaller than an equivalent JSON message.
Nanoservices	Azure Functions do not need to be hosted 24/7 so they are a good choice for nanoservices that usually do not need to be running constantly.

Practicing and exploring

Test your knowledge and understanding by answering some questions, get some hands-on practice, and explore this chapter's topics with deeper research.

Exercise 18.1 – Test your knowledge

Answer the following questions:

1. You have an app that communicates with a service built using the legacy Windows Communication Foundation service. What are two possible options for migrating the service and client to modern .NET?
2. What transport protocol does an OData service use?
3. Why is an OData web service more flexible than a traditional ASP.NET Core Web API web service?

4. What must you do to an action method in an OData controller to enable query strings to customize what it returns?
5. What transport protocol does a GraphQL service use?
6. How are contracts defined in gRPC?
7. What are three benefits of gRPC that make it a good choice for implementing services?
8. What transports does SignalR use, and which is the default?
9. What is the difference between the in-process and isolated hosting models for Azure Functions?
10. What is a good practice for RPC method signature design?

Exercise 18.2 – Explore topics

Use the links on the following page to learn more detail about the topics covered in this chapter:

`https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-18---building-and-consuming-other-services`

Summary

In this chapter, you learned how to build more specialized types of service using various technologies including gRPC, SignalR, OData, GraphQL, and Azure Functions.

In the next chapter, you will learn how to build cross-platform mobile and desktop apps using .NET MAUI.

19

Building Mobile and Desktop Apps Using .NET MAUI

This chapter is about learning how to make **graphical user interface** (**GUI**) apps by building a cross-platform mobile and desktop app for iOS and Android, macOS Catalyst, and Windows using **.NET MAUI** (**Multi-platform App User Interface**).

Warning! This chapter was tested using .NET Release Candidate 2, .NET MAUI Preview 9, and Visual Studio 2022 Preview 5. Future previews are likely to fix some things that are not working, but also break some things that were working, until the GA release in Q2 2022. For the latest updates, please see the updated chapter online at the following link: `https://github.com/markjprice/cs10dotnet6/tree/main/docs/chapter19`

You will see how **eXtensible Application Markup Language** (**XAML**) makes it easy to define the user interface for a graphical app.

Cross-platform GUI development cannot be learned in a single chapter, but like web development, it is so important that I want to introduce you to some of what is possible. Think of this chapter as an introduction that will give you a taste to inspire you, and then you can learn more from a book dedicated to mobile or desktop development.

The app will allow the listing and management of customers in the Northwind database. The mobile app that you create will call the Northwind service that you built using the ASP.NET Core Web API in *Chapter 16*, *Building and Consuming Web Services*. If you have not built the Northwind service, please go back and build it now or download it from the GitHub repository for this book at the following link: `https://github.com/markjprice/cs10dotnet6`.

After .NET MAUI has its general availability release expected in May 2022, either a Windows computer with Visual Studio or a macOS computer with Visual Studio for Mac can be used to create a .NET MAUI project. But you will need a computer with Windows to compile WinUI 3 apps and you will need a computer with macOS and Xcode to compile for macOS Catalyst and iOS.

Although you can create a .NET MAUI project at the command line and then edit it using Visual Studio Code, there is no official tooling to help you yet. That is expected to come with .NET 7.0 in late 2022.

In this chapter, we will cover the following topics:

- Understanding the .NET MAUI delay
- Understanding XAML
- Understanding .NET MAUI
- Building mobile and desktop apps using .NET MAUI
- Consuming a web service from a .NET MAUI app

Understanding the .NET MAUI delay

On September 14, 2021, Microsoft announced that .NET MAUI would be delayed. "Unfortunately, .NET MAUI will not be ready for production with .NET 6 GA in November." – Scott Hunter, Director of Program Management, .NET. You can read more from Scott's announcement at the following link:

`https://devblogs.microsoft.com/dotnet/update-on-dotnet-maui/`

The following seems a likely timeline of preview and release candidate releases leading to the general availability release of .NET MAUI in Q2 2022:

- October 12, 2021: .NET MAUI Preview 9 and .NET 6 Release Candidate 2 that were used for this chapter published in the printed and eBook editions of this book
- November 9, 2021: .NET MAUI Preview 10 and .NET 6 GA
- December 2021: .NET MAUI Preview 11
- January 2022: .NET MAUI Preview 12
- February 2022: .NET MAUI Preview 13
- March 2022: .NET MAUI Release Candidate 1
- April 2022: .NET MAUI Release Candidate 2
- May 2022: .NET MAUI General Availability at Microsoft Build
- November 2022: .NET MAUI included with .NET 7

My publisher, Packt, and I wanted to include this chapter in the published book even though parts are likely to change after publishing. To keep the chapter up to date as .NET MAUI previews continue to be released, I plan to update this chapter in the GitHub repository for this book up until the GA release. You can find the online version of this chapter at the following link:

`https://github.com/markjprice/cs10dotnet6/tree/main/docs/chapter19`

Let's start by looking at the markup language used by .NET MAUI.

Understanding XAML

In 2006, Microsoft released **Windows Presentation Foundation** (**WPF**), which was the first technology to use **XAML** (**eXtensible Application Markup Language**). Silverlight, for web and mobile apps, quickly followed, but it is no longer supported by Microsoft. WPF is still used today to create Windows desktop applications; for example, Visual Studio for Windows is partially built using WPF.

XAML can be used to build parts of the following apps:

- **.NET MAUI apps** for mobile and desktop devices, including Android, iOS, Windows, and macOS. It is an evolution of a technology named **Xamarin.Forms**.
- **WinUI 3 apps** for Windows 10 and 11.
- **Universal Windows Platform** (**UWP**) **apps** for Windows 10 and 11, Xbox, and Mixed Reality headsets.
- **WPF apps** for Windows desktop, including Windows 7 and later.
- **Avalonia** and **Uno Platform apps** using cross-platform, third-party technologies.

Simplifying code using XAML

XAML simplifies C# code, especially when building a user interface.

Imagine that you need two or more buttons laid out horizontally to create a toolbar.

In C#, you might write this code:

```
StackPanel toolbar = new();
toolbar.Orientation = Orientation.Horizontal;

Button newButton = new();
newButton.Content = "New";
newButton.Background = new SolidColorBrush(Colors.Pink);
toolbar.Children.Add(newButton);

Button openButton = new();
openButton.Content = "Open";
openButton.Background = new SolidColorBrush(Colors.Pink);
toolbar.Children.Add(openButton);
```

In XAML, this could be simplified to the following lines of code. When this XAML is processed, the equivalent properties are set, and methods are called to achieve the same goal as the preceding C# code:

```
<StackPanel Name="toolbar" Orientation="Horizontal">
  <Button Name="newButton" Background="Pink">New</Button>
  <Button Name="openButton" Background="Pink">Open</Button>
</StackPanel>
```

You can think of XAML as an alternative and easier way of declaring and instantiating .NET types, especially when defining a user interface and the resources that it uses.

XAML allows resources such as brushes, styles, and themes to be declared at different levels, like a UI element, a page, or globally for the application to enable resource sharing.

XAML allows data binding between UI elements or between UI elements and objects and collections.

Choosing common controls

There are lots of predefined controls that you can choose from for common user interface scenarios. Almost all dialects of XAML support these controls:

Controls	**Description**
`Button`, `ImageButton`, `Menu`, `Toolbar`	Executing actions
`CheckBox`, `RadioButton`	Choosing options
`Calendar`, `DatePicker`	Choosing dates
`ComboBox`, `ListBox`, `ListView`, `TreeView`	Choosing items from lists and hierarchical trees
`Canvas`, `DockPanel`, `Grid`, `StackPanel`, `WrapPanel`	Layout containers that affect their children in different ways
`Label`, `TextBlock`	Displaying read-only text
`RichTextBox`, `TextBox`	Editing text
`Image`, `MediaElement`	Embedding images, videos, and audio files
`DataGrid`	Viewing and editing data as quickly and easily as possible
`Scrollbar`, `Slider`, `StatusBar`	Miscellaneous user interface elements

Understanding markup extensions

To support some advanced features, XAML uses markup extensions. Some of the most important enable element and data binding and the reuse of resources, as shown in the following list:

- `{Binding}` links an element to a value from another element or a data source
- `{StaticResource}` links an element to a shared resource
- `{ThemeResource}` links an element to a shared resource defined in a theme

You will see some practical examples of markup extensions throughout this chapter.

Understanding .NET MAUI

To create a mobile app that only needs to run on iPhones, you might choose to build it with either the Objective-C or Swift language and the UIKit libraries using the Xcode development tool.

To create a mobile app that only needs to run on Android phones, you might choose to build it with either the Java or Kotlin language and the Android SDK libraries using the Android Studio development tool.

But what if you need to create a mobile app that can run on iPhones *and* Android phones? And what if you only want to create that mobile app once using a programming language and development platform that you are already familiar with? And what if you realized that with a bit more coding effort to adapt the user interface to desktop size devices, you could target macOS and Windows desktops too?

.NET MAUI enables developers to build cross-platform mobile apps for Apple iOS (iPhone), iPadOS, macOS using Catalyst, Windows using WinUI 3, and Google Android using C# and .NET, which are then compiled to native APIs and executed on native phone and desktop platforms.

Business logic layer code can be written once and shared between all platforms. User interface interactions and APIs are different on various mobile and desktop platforms, so the user interface layer is sometimes custom for each platform.

Like WPF and UWP apps, .NET MAUI uses XAML to define the user interface once for all platforms using abstractions of platform-specific user interface components. Applications built with .NET MAUI draw the user interface using native platform widgets, meaning the app's look and feel fit naturally with the target mobile platform.

A user experience built using .NET MAUI will not perfectly fit a specific platform in a way that one custom built with native tools for that platform would, but for mobile and desktop apps that will not have millions of users, it is good enough.

Development tools for mobile first, cloud first

Mobile apps are often supported by services in the cloud.

Satya Nadella, CEO of Microsoft, famously said the following:

> "*To me, when we say mobile first, it's not the mobility of the device, it's actually the mobility of the individual experience. [...] The only way you are going to be able to orchestrate the mobility of these applications and data is through the cloud.*"

As you have seen earlier in this book, to create an ASP.NET Core Web API service to support a mobile app, we can use Visual Studio Code. To create .NET MAUI apps, developers can use either Visual Studio 2022 for Windows or Visual Studio 2022 for Mac.

When installing Visual Studio 2022, you must select the **.NET MAUI (Preview)** checkbox that is part of the **Mobile development with .NET** workload, as shown in *Figure 19.1*:

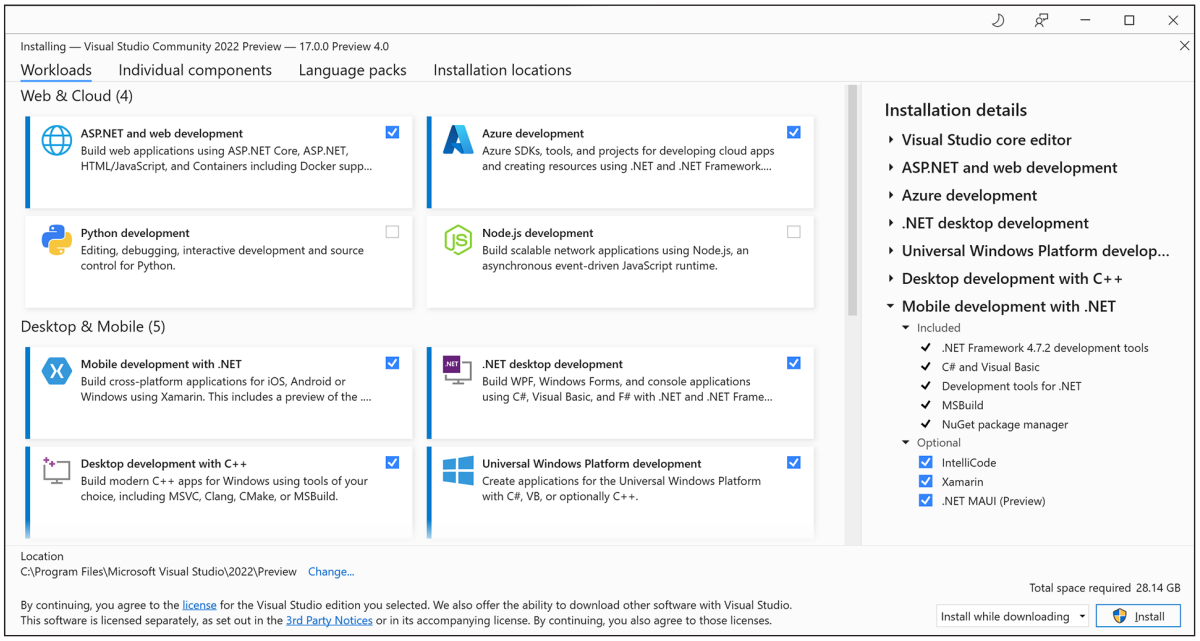


Figure 19.1: Selecting the .NET MAUI workload for Visual Studio 2022

Using Windows to create iOS and macOS apps

If you want to use Visual Studio 2022 for Windows to create an iOS mobile app or a macOS Catalyst desktop app, then you can connect over a network to a Mac build host. Instructions can be found at the following link:

```
https://docs.microsoft.com/en-us/xamarin/ios/get-started/installation/windows/
connecting-to-mac/
```

Understanding additional functionality

We will build a cross-platform mobile and desktop app that uses a lot of the skills and knowledge that you learned in previous chapters. We will also use some functionality that you have not seen before.

Understanding MVVM

Model-View-ViewModel (**MVVM**) is a design pattern like MVC. The letters in the acronym stand for:

- **Model**: an entity class that represents a data object in a store, like a relational database.
- **View**: a way to represent data in a graphical user interface, including fields to show and edit data fields and buttons and other elements to interact with the data.

- **ViewModel**: a class that represents the data fields, actions, and events that can then be bound to elements like textboxes and buttons in a view.

In MVC, models passed to a view are read-only because they are only passed one way into the view. That is why immutable records are good for MVC models. But ViewModels are different. They need to support two-way interactions and if the original data changes during the lifetime of the object, the view needs to dynamically update.

Understanding the INotificationPropertyChanged interface

The `INotifyPropertyChanged` interface enables a model class to support two-way data binding. It works by forcing the class to have an event named `PropertyChanged`, with a parameter of type `PropertyChangedEventArgs`, as shown in the following code:

```
namespace System.ComponentModel
{
  public class PropertyChangedEventArgs : EventArgs
  {
    public PropertyChangedEventArgs(string? propertyName);
    public virtual string? PropertyName { get; }
  }

  public delegate void PropertyChangedEventHandler(
    object? sender, PropertyChangedEventArgs e);

  public interface INotifyPropertyChanged
  {
    event PropertyChangedEventHandler PropertyChanged;
  }
}
```

Inside each property in the class, when setting a new value, you must raise the event (if it is not `null`) with an instance of `PropertyChangedEventArgs` containing the name of the property as a `string` value, as shown in the following code:

```
private string companyName;

public string CompanyName
{
  get => companyName;
  set
  {
    companyName = value; // store the new value being set
    PropertyChanged?.Invoke(this,
      new PropertyChangedEventArgs(nameof(CompanyName)));
  }
}
```

When a user interface control is data-bound to the property, it will automatically update to show the new value when it changes.

To simplify the implementation, we can use a compiler feature to get the name of the property by decorating a `string` parameter with the `[CallerMemberName]` attribute, as shown in the following code:

```
private void NotifyPropertyChanged(
  [CallerMemberName] string propertyName = "")
{
  // if an event handler has been set then invoke
  // the delegate and pass the name of the property
  PropertyChanged?.Invoke(this,
    new PropertyChangedEventArgs(propertyName));
}

public string CompanyName
{
  get => companyName;
  set
  {
    companyName = value; // store the new value being set
    NotifyPropertyChanged(); // caller member name is "CompanyName"
  }
}
```

Understanding ObservableCollection

Related to `INotifyPropertyChanged` is the `INotifyCollectionChanged` interface, which is implemented by the `ObservableCollection<T>` class. This gives notifications when items get added, removed, or when the collection is refreshed. When bound to controls like `ListView` or `TreeView`, the user interface will update dynamically to reflect changes.

Understanding dependency services

Mobile platforms such as iOS and Android, and desktop platforms like Windows and macOS, implement common features in different ways, so we need a way to get a platform-native implementation of common features. We can do that using dependency services. It works like this:

- Define an interface for the common feature, for example, `IDialer` for a phone number dialer on a phone device, or `INotificationManager` for a pop-up local notification on desktop and mobile devices.
- Implement the interface for all the platforms that you need to support, for example, iOS and Android for a phone dialer, and register the implementations with an attribute, as shown in the following code:

```
[assembly: Dependency(typeof(PhoneDialer))]
```

```
namespace Northwind.Maui.iOS
{
  public class PhoneDialer : IDialer
```

- Get the platform-native implementation of an interface by using the dependency service, as shown in the following code:

```
IDialer dialer = DependencyService.Get<IDialer>();
```

.NET MAUI Essentials includes a `PhoneDialer` component, so we will use that in our project rather than have to define our own phone dialer dependency service.

Understanding .NET MAUI user interface components

.NET MAUI includes some common controls for building user interfaces. They are divided into four categories:

- **Pages**: represent cross-platform application screens, for example, `ContentPage`, `NavigationPage`, `FlyoutPage`, and `TabbedPage`.
- **Layouts**: represent the structure of a combination of other user interface components, for example, `Grid`, `StackLayout`, and `FlexLayout`.
- **Views**: represent a single user interface component, for example, `CarouselView`, `CollectionView`, `Label`, `Entry`, `Editor`, and `Button`.
- **Cells**: represent a single item in a list or table view, for example, `TextCell`, `ImageCell`, `SwitchCell`, and `EntryCell`.

You can track the status of the migration progress of .NET MAUI components at the following link: `https://github.com/dotnet/maui/wiki/Status`

Understanding the ContentPage view

The `ContentPage` view is for simple user interfaces. It has a `ToolbarItems` property that shows the actions the user can perform in a platform-native way. Each `ToolbarItem` can have an icon and text:

```
<ContentPage.ToolbarItems>
  <ToolbarItem Text="Add" Activated="Add_Activated"
    Order="Primary" Priority="0" />
  ...
</ContentPage.ToolbarItems>
```

Understanding the ListView control

The `ListView` control is used for long lists of data-bound values of the same type. It can have headers and footers and its list items can be grouped.

It has cells to contain each list item. There are two built-in cell types: text and image. Developers can define custom cell types.

Cells can have context actions that appear when the cell is swiped on iPhone or long pressed on Android. A context action that is destructive can be shown in red, as shown in the following markup:

```
<TextCell Text="{Binding CompanyName}" Detail="{Binding Location}">
  <TextCell.ContextActions>
    <MenuItem Clicked="Customer_Phoned" Text="Phone" />
    <MenuItem Clicked="Customer_Deleted" Text="Delete" IsDestructive="True" />
  </TextCell.ContextActions>
</TextCell>
```

Understanding the Entry and Editor controls

The `Entry` and `Editor` controls are used for editing text values and are often data-bound to an entity model property, as shown in the following markup:

```
<Editor Text="{Binding CompanyName, Mode=TwoWay}" />
```

Use `Entry` for a single line of text. Use `Editor` for multiple lines of text.

Understanding .NET MAUI handlers

In .NET MAUI, XAML controls are defined in the `Microsoft.Maui.Controls` namespace. Components called **handlers** map these common controls to native controls on each platform. On iOS, a handler will map a .NET MAUI `Button` to an iOS-native `UIButton` defined by UIKit. On macOS, `Button` is mapped to `NSButton` defined by AppKit. On Android, `Button` is mapped to an Android-native `AppCompatButton`.

Handlers have a `NativeView` property that exposes the underlying native control. This allows you to work with platform-specific features like properties, methods, and events and customize all instances of a native control.

Writing platform-specific code

If you need to write code statements that only execute for a specific platform like Android, then you can use compiler directives.

For example, by default, `Entry` controls on Android show an underline character.

If you want to hide the underline, you could write some Android-specific code to get the handler for the `Entry` control, use its `NativeView` property to access the underlying native control, and then set the property that controls that feature to `false`, as shown in the following code:

```
#if __ANDROID__
  Handlers.EntryHandler.EntryMapper[nameof(IEntry.BackgroundColor)] = (h, v) =>
  {
    (h.NativeView as global::Android.Views.Entry).UnderlineVisible = false;
  };
#endif
```

Predefined compiler constants include the following:

- `__ANDROID__`
- `__IOS__`
- `WINDOWS`

The compiler `#if` statement syntax is slightly different from the C# `if` statement syntax, as shown in the following code:

```
#if __IOS__
  // iOS-specific statements
#elif __ANDROID__
  // Android-specific statements
#elif WINDOWS
  // Windows-specific statements
#endif
```

Building mobile and desktop apps using .NET MAUI

We will build a mobile and desktop app for managing customers in Northwind.

Good Practice: If you have never run Xcode, run it now until you see the Start window to ensure that all its required components are installed and registered. If you do not run Xcode, then you might get errors with your projects later in Visual Studio for Mac.

Creating a virtual Android device for local app testing

To target Android, you must install at least one Android SDK. A default installation of Visual Studio with the mobile development workload already includes one Android SDK, but it is often an older version to support as many Android devices as possible.

To use the latest features of .NET MAUI, you must install a more recent Android SDK:

1. In Windows, start **Visual Studio 2022**.
2. Navigate to **Tools** | **Android** | **Android Device Manager**.
3. In **Android Device Manager**, click the **+ New** button to create a new device.
4. In the **New Device** dialog, make the following choices:
    1. **Base Device**: **Pixel 2 (+ Store)**
    2. **Processor**: **x86**
    3. **OS**: **Pie 9.0 – API 28**
5. Click **Create**.
6. Accept any license agreements.
7. Wait for any required downloads.
8. In **Android Device Manager**, in the list of devices, in the row for the device that you just created, click **Start**.
9. When the Android device has finished starting, click the browser and test that it has access to the network by navigating to `https://www.bbc.co.uk/news`.
10. Close the emulator.
11. Restart Visual Studio 2022 to ensure that it is aware of the new emulator.

Creating a .NET MAUI solution

We will now create a project for a cross-platform mobile and desktop app:

1. In Visual Studio for Windows, add a new project, as defined in the following list:
    1. Project template: **.NET MAUI App (Preview)** / `maui`
    2. Workspace/solution file and folder: `PracticalApps`
    3. Project file and folder: `Northwind.Maui.Customers`
2. Open the project file, and uncomment the element to enable Windows targeting, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <TargetFrameworks>net6.0-ios;net6.0-android;net6.0-maccatalyst</
TargetFrameworks>
        <TargetFrameworks Condition="$([MSBuild]::IsOSPlatform('windo
ws')) and '$(MSBuildRuntimeType)' == 'Full'">$(TargetFrameworks);net6.0-
windows10.0.19041</TargetFrameworks>
        <OutputType>Exe</OutputType>
        <RootNamespace>Northwind.Maui.Customers</RootNamespace>
```

```
        <UseMaui>true</UseMaui>
        <SingleProject>true</SingleProject>
```

3. To the right of the **Run** button in the toolbar, set the **Framework** to **net6.0-android**, and select the **Pixel 2 - API 28 (Android 9.0 - API 28)** emulator image that you previously created, as shown in *Figure 19.2*:

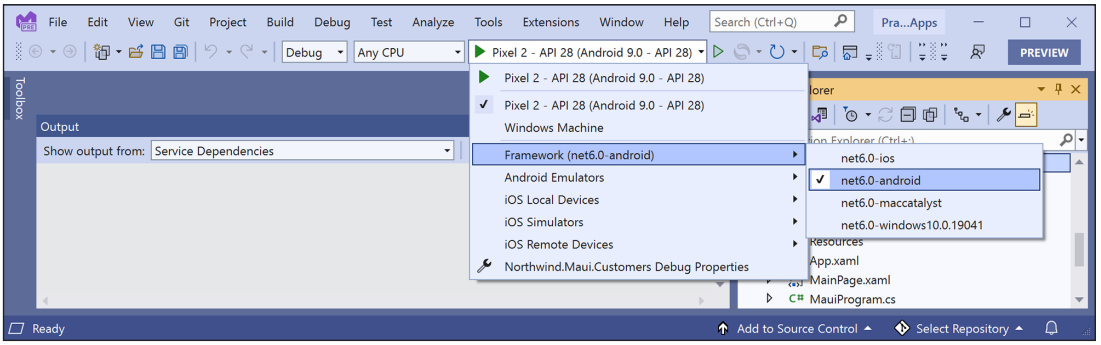


Figure 19.2: Selecting Android as the target for startup

4. Click the **Run** button in the toolbar and wait for the device emulator to start the Android operating system and launch your mobile app.
5. In the .NET MAUI app, click the **Click me** button to increment the counter three times, as shown in *Figure 19.3*:

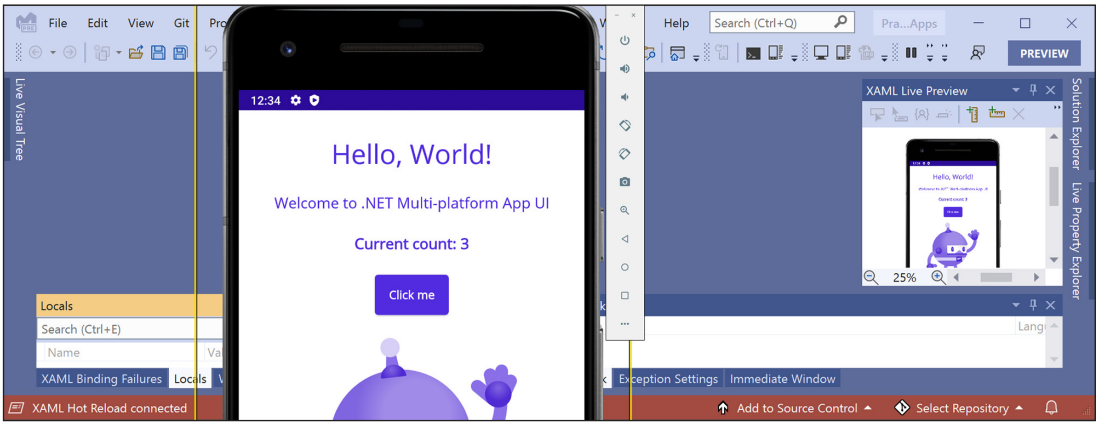


Figure 19.3: Incrementing the counter in the Android .NET MAUI app

6. Note the **XAML Live Preview** window in Visual Studio and that **XAML Hot Reload** is **connected** so that you could make changes to the XAML and see them reflected in the app without restarting. For example, try changing the text of the Hello World label to something else, save the XAML file, and click the **Hot Reload** button in the toolbar.
7. Close the Android device emulator.
8. Navigate to **Build | Configuration Manager**.

9. In the row for the **Northwind.Maui.Customers** project, select the checkbox in the **Deploy** column, as shown in *Figure 19.4*:

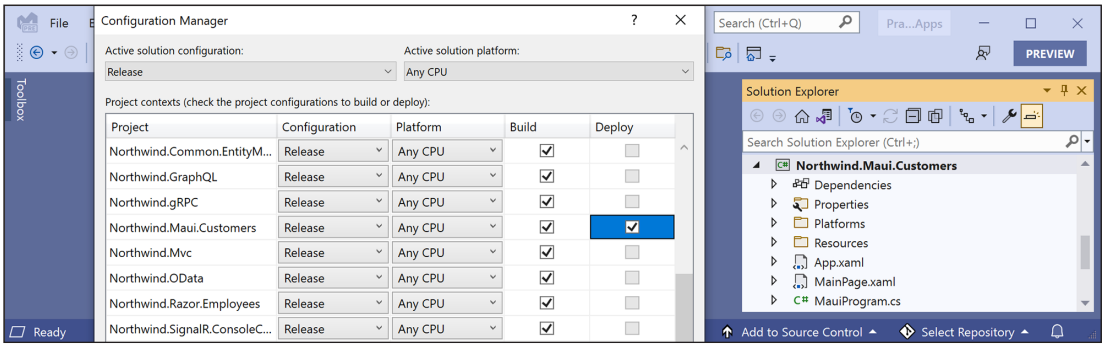


Figure 19.4: Enabling the Windows app to deploy to the Windows machine

10. To the right of the **Run** button in the toolbar, set the **Framework** to **net6.0-windows**, and then select **Windows Machine**.
11. Make sure that the **Debug** configuration is selected and then click the green triangle start button labeled **Windows Machine**.
12. After a few moments, note that the Windows app displays with the same **Click me** button and counter functionality, as shown in *Figure 19.5*:

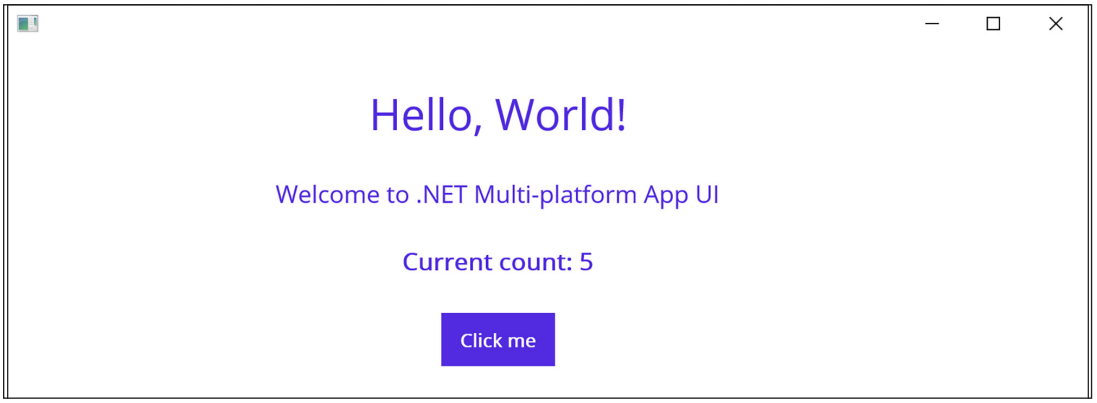


Figure 19.5: Incrementing the counter in the Windows .NET MAUI app

13. Close the Windows app.

Creating a view model with two-way data binding

We need to create a view model that will allow us to show and modify a customer entity so the class should implement two-way data binding:

1. In the `Northwind.Maui.Customers` project folder, create two classes, one named `CustomerDetailViewModel.cs` to show the details of a single customer, and one named `CustomersListViewModel.cs` to show a list of customers.

2. In `CustomerDetailViewModel.cs`, modify the statements to define a class that implements the `INotifyPropertyChanged` interface and has six properties, as shown in the following code:

```
using System.ComponentModel; // INotifyPropertyChanged
using System.Runtime.CompilerServices; // [CallerMemberName]

namespace Northwind.Maui.Customers;

public class CustomerDetailViewModel : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;

  private string customerId;
  private string companyName;
  private string contactName;
  private string city;
  private string country;
  private string phone;

  // this attribute sets the propertyName parameter
  // using the context in which this method is called
  private void NotifyPropertyChanged(
    [CallerMemberName] string propertyName = "")
  {
    // if an event handler has been set then invoke
    // the delegate and pass the name of the property
    PropertyChanged?.Invoke(this,
      new PropertyChangedEventArgs(propertyName));
  }

  public string CustomerId
  {
    get => customerId;
    set
    {
      customerId = value;
      NotifyPropertyChanged();
    }
  }

  public string CompanyName
  {
    get => companyName;
    set
    {
```

```
        companyName = value;
        NotifyPropertyChanged();
      }
    }

    public string ContactName
    {
      get => contactName;
      set
      {
        contactName = value;
        NotifyPropertyChanged();
      }
    }

    public string City
    {
      get => city;
      set
      {
        city = value;
        NotifyPropertyChanged();
        NotifyPropertyChanged(nameof(Location));
      }
    }

    public string Country
    {
      get => country;
      set
      {
        country = value;
        NotifyPropertyChanged();
        NotifyPropertyChanged(nameof(Location));
      }
    }

    public string Phone
    {
      get => phone;
      set
      {
        phone = value;
        NotifyPropertyChanged();
      }
    }
```

```
  public string Location
  {
    get => $"{City}, {Country}";
  }
}
```

Note the following:

- The class implements `INotifyPropertyChanged`, so a two-way bound control like `Editor` will update the property and vice versa. There is a `PropertyChanged` event that is raised whenever one of the properties is modified using a `NotifyPropertyChanged` private method to simplify the implementation.
- In addition to properties for storing values retrieved from the HTTP service, the class defines a read-only `Location` property. This will be bound to a summary list of customers to show the location of each one. Whenever the `City` or `Country` property changes, we also need to notify that the `Location` has changed, or any views bound to `Location` would not update correctly.

3. In `CustomersListViewModel.cs`, modify the statements to define a class that inherits from `ObservableCollection<T>` and has a method to populate sample data, as shown in the following code:

```
using System.Collections.ObjectModel; // ObservableCollection<T>

namespace Northwind.Maui.Customers;

public class CustomersListViewModel :
  ObservableCollection<CustomerDetailViewModel>
{
  // for testing before calling web service
  public void AddSampleData(bool clearList = true)
  {
    if (clearList) Clear();

    Add(new CustomerDetailViewModel
    {
      CustomerId = "ALFKI",
      CompanyName = "Alfreds Futterkiste",
      ContactName = "Maria Anders",
      City = "Berlin",
      Country = "Germany",
      Phone = "030-0074321"
    });

    Add(new CustomerDetailViewModel
    {
```

```
        CustomerId = "FRANK",
        CompanyName = "Frankenversand",
        ContactName = "Peter Franken",
        City = "München",
        Country = "Germany",
        Phone = "089-0877310"
      });

      Add(new CustomerDetailViewModel
      {
        CustomerId = "SEVES",
        CompanyName = "Seven Seas Imports",
        ContactName = "Hari Kumar",
        City = "London",
        Country = "UK",
        Phone = "(171) 555-1717"
      });
    }
  }
```

Note the following:

- After loading from the service, which will be implemented later in this chapter, the customers are cached locally using `ObservableCollection<T>`. This supports notifications to any bound user interface components, such as `ListView`, so that the user interface can redraw itself when the underlying data adds or removes items from the collection.
- For testing purposes, when the HTTP service is not available, there is a static method to populate three sample customers.

Creating views for the list of customers and customer details

You will now replace the existing `MainPage` with a view to show a list of customers and a view to show the details for a customer:

1. In the `Northwind.Maui.Customers` project, delete **MainPage.xaml**.
2. Open `App.xaml` and add a style to apply the same background color and font family to `Entry` controls as are being applied to `Label` controls, as shown in the following markup:

```
<Style TargetType="Entry">
  <Setter Property="TextColor" Value="{DynamicResource PrimaryTextColor}"
/>
  <Setter Property="FontFamily" Value="OpenSansRegular" />
  <Setter Property="HorizontalOptions" Value="StartAndExpand" />
```

```
    <Setter Property="WidthRequest" Value="300" />
</Style>
```

Implementing the customer list view

First, we will create two views for a list of customers and show details of one customer, and then we will implement the list of customers:

1. Right-click the `Northwind.Maui.Customers` project folder, choose **Add | New Item...**, select **Content Page**, enter the name `CustomersListPage`, and click **Add**, as shown in *Figure 19.6*:

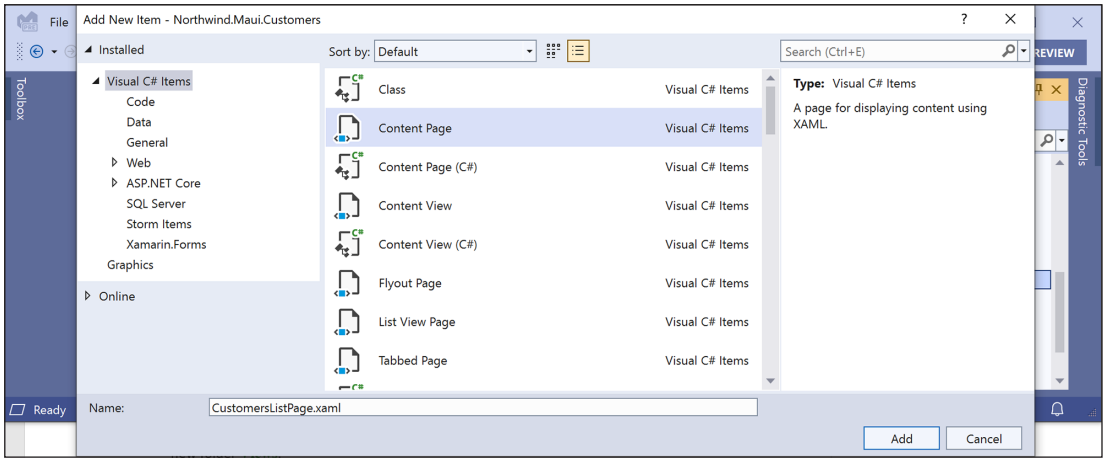


Figure 19.6: Adding a new XAML Content Page item

2. Right-click the **Views** folder, choose **Add | New Item...**, select **Content Page**, enter the name `CustomerDetailPage`, and click **Add**.

At the time of writing, Visual Studio 2022 does not have project item templates for .NET MAUI. The `ContentPage` project item template is for the older Xamarin.Forms. In the next step, we will replace almost all the markup and code anyway, so it is not an issue. By May 2022, I expect Visual Studio 2022 to have project item templates for common .NET MAUI file types.

3. Open `CustomersListPage.xaml` and modify its contents, as shown in the following markup:

```
<ContentPage
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="Northwind.Maui.Customers.CustomersListPage"
  BackgroundColor="{DynamicResource PageBackgroundColor}"
  Title="List">
```

```
    <ContentPage.Content>
      <ListView ItemsSource="{Binding .}"
                VerticalOptions="Center"
                HorizontalOptions="Center"
                IsPullToRefreshEnabled="True"
                ItemTapped="Customer_Tapped"
                Refreshing="Customers_Refreshing">
        <ListView.Header>
          <StackLayout Orientation="Horizontal">
            <Label Text="Northwind Customers"
                   FontSize="Subtitle" Margin="10" />
            <Button Text="Add" Clicked="Add_Clicked" />
          </StackLayout>
        </ListView.Header>
        <ListView.ItemTemplate>
          <DataTemplate>
            <TextCell Text="{Binding CompanyName}"
                      Detail="{Binding Location}"
                      TextColor="{DynamicResource PrimaryTextColor}"
                      DetailColor="{DynamicResource PrimaryTextColor}" >
              <TextCell.ContextActions>
                <MenuItem Clicked="Customer_Phoned" Text="Phone" />
                <MenuItem Clicked="Customer_Deleted" Text="Delete"
                          IsDestructive="True" />
              </TextCell.ContextActions>
            </TextCell>
          </DataTemplate>
        </ListView.ItemTemplate>
      </ListView>
    </ContentPage.Content>
  </ContentPage>
```

Note the following:

- `ContentPage` has had its `Title` attribute set to `List`.
- `ListView` has its `IsPullToRefreshEnabled` attribute set to `true`.
- Handlers have been written for the following events:
- `Customer_Tapped`: A customer being tapped to show their details.
- `Customers_Refreshing`: The list being pulled down to refresh its items.
- `Customer_Phoned`: A cell being swiped left on iPhone or long pressed on Android and then tapping **Phone**.
- `Customer_Deleted`: A cell being swiped left on iPhone or long pressed on Android and then tapping **Delete**.
- `Add_Clicked`: The **Add** button being clicked.

- A data template defines how to display each customer: larger text for the company name and smaller text for the location underneath.
- An **Add** button is in the list view header so that users can navigate to a detail view to add a new customer.

4. Open `CustomersListPage.xaml.cs` and modify the contents, as shown in the following code:

```
using Microsoft.Maui.Controls; // ContentPage, ListView
using Microsoft.Maui.Essentials; // PhoneDialer
using System;
using System.Threading.Tasks;

namespace Northwind.Maui.Customers;

public partial class CustomersListPage : ContentPage
{
  public CustomersListPage()
  {
    InitializeComponent();

    CustomersListViewModel viewModel = new();
    viewModel.AddSampleData();
    BindingContext = viewModel;
  }

  async void Customer_Tapped(object sender, ItemTappedEventArgs e)
  {
    if (e.Item is not CustomerDetailViewModel c) return;

    // navigate to the detail view and show the tapped customer
    await Navigation.PushAsync(new CustomerDetailPage(
      BindingContext as CustomersListViewModel, c));
  }

  async void Customers_Refreshing(object sender, EventArgs e)
  {
    if (sender is not ListView listView) return;

    listView.IsRefreshing = true;

    // simulate a refresh
    await Task.Delay(1500);

    listView.IsRefreshing = false;
  }
```

```
    void Customer_Deleted(object sender, EventArgs e)
    {
      MenuItem menuItem = sender as MenuItem;
      if (menuItem.BindingContext is not CustomerDetailViewModel c) return;
      (BindingContext as CustomersListViewModel).Remove(c);
    }

    async void Customer_Phoned(object sender, EventArgs e)
    {
      MenuItem menuItem = sender as MenuItem;
      if (menuItem.BindingContext is not CustomerDetailViewModel c) return;

      if (await DisplayAlert("Dial a Number",
        "Would you like to call " + c.Phone + "?",
        "Yes", "No"))
      {
        PhoneDialer.Open(c.Phone);
      }
    }

    async void Add_Clicked(object sender, EventArgs e)
    {
      await Navigation.PushAsync(new CustomerDetailPage(
        BindingContext as CustomersListViewModel));
    }
  }
```

Note the following:

- `BindingContext` is set to an instance of `CustomersViewModel`, which is populated with sample data in the constructor of the page.
- When a customer in the list view is tapped, the user is taken to a details view (which you will implement in the next step).
- When the list view is pulled down, it triggers a simulated refresh that takes 1.5 seconds.
- When a customer is deleted in the list view, they are removed from the bound customers view model.
- When a customer in the list view is swiped, and the **Phone** button is tapped, a dialog prompts the user as to whether they want to dial the number, and if so, the platform-native implementation will be retrieved using the dependency resolver and then used to dial the number.
- When the **Add** button is tapped, the user is taken to the customer detail page to enter details for a new customer.

Implementing the customer detail view

Next, we will implement the customer detail view:

1. Open `CustomerDetailPage.xaml` and modify its contents, as shown in the following markup, and note the following:
    - `Title` of the content page has been set to `Edit`.
    - A customer `Grid` with two columns and six rows is used for the layout.
    - `Entry` views are two-way data bound to properties of the `CustomerViewModel` class.
    - `InsertButton` has an event handler to execute code to add a new customer.

```
<ContentPage
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="Northwind.Maui.Customers.Views.CustomerDetailPage"
  BackgroundColor="{DynamicResource PageBackgroundColor}"
  Title="Edit">

  <ContentPage.Content>
    <StackLayout VerticalOptions="Fill" HorizontalOptions="Fill">
      <Grid ColumnDefinitions="Auto,Auto"
            RowDefinitions="Auto,Auto,Auto,Auto,Auto,Auto">
        <Label Text="Customer Id" VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding CustomerId, Mode=TwoWay}" Grid.Column="1"
               MaxLength="5" TextTransform="Uppercase" />
        <Label Text="Company Name" Grid.Row="1"
               VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding CompanyName, Mode=TwoWay}"
               Grid.Column="1" Grid.Row="1" />
        <Label Text="Contact Name" Grid.Row="2"
             VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding ContactName, Mode=TwoWay}"
               Grid.Column="1" Grid.Row="2" />
        <Label Text="City" Grid.Row="3"
               VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding City, Mode=TwoWay}"
               Grid.Column="1" Grid.Row="3" />
        <Label Text="Country" Grid.Row="4"
               VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding Country, Mode=TwoWay}"
               Grid.Column="1" Grid.Row="4" />
        <Label Text="Phone" Grid.Row="5"
               VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding Phone, Mode=TwoWay}"
```

```
                Grid.Column="1" Grid.Row="5" />
        </Grid>
        <Button x:Name="InsertButton" Text="Insert Customer"
                Clicked="InsertButton_Clicked" />
      </StackLayout>
    </ContentPage.Content>
  </ContentPage>
```

2. Open `CustomerDetailPage.xaml.cs` and modify its contents, as shown in the following code:

```
using Microsoft.Maui.Controls;
using System;
using System.Threading.Tasks;

namespace Northwind.Maui.Customers;

public partial class CustomerDetailPage : ContentPage
{
  private CustomersListViewModel customers;

  public CustomerDetailPage(CustomersListViewModel customers)
  {
    InitializeComponent();

    this.customers = customers;
    BindingContext = new CustomerDetailViewModel();
    Title = "Add Customer";
  }

  public CustomerDetailPage(CustomersListViewModel customers,
    CustomerDetailViewModel customer)
  {
    InitializeComponent();

    this.customers = customers;
    BindingContext = customer;
    InsertButton.IsVisible = false;
  }

  async void InsertButton_Clicked(object sender, EventArgs e)
```

```
    {
      customers.Add((CustomerDetailViewModel)BindingContext);
      await Navigation.PopAsync(animated: true);
    }
  }
```

Note the following:

- The default constructor sets the binding context to a new `customer` instance and the view title is changed to **Add Customer**.
- The constructor with a `customer` parameter sets the binding context to that instance and hides the **Insert** button because it is not needed when editing an existing customer due to two-way data binding.
- When the **Insert** button is tapped, the new customer is added to the customers view model and the navigation is moved back to the previous view asynchronously.

Setting the main page for the mobile app

Finally, we need to modify the mobile app to use our customer list wrapped in a navigation page as the main page instead of the old one that we deleted, which was created by the project template:

1. Open `App.xaml.cs`.
2. In the `App` constructor, modify the statement that creates a `MainPage` to instead create an instance of `CustomersListPage` wrapped in an instance of `NavigationPage`, as shown highlighted in the following code:

```
public App()
{
  InitializeComponent();

  MainPage = new NavigationPage(new CustomersListPage());
}
```

Testing the mobile app

We will now test the mobile app using the Android device emulator:

1. In Visual Studio, to the right of the **Run** button in the toolbar, set the target **Framework** to **net6.0-android** and select the Android emulator.

2. Start with project debugging. The project will build, and then after a few moments, the Android device emulator will appear with your running .NET MAUI app, as shown in *Figure 19.7*:

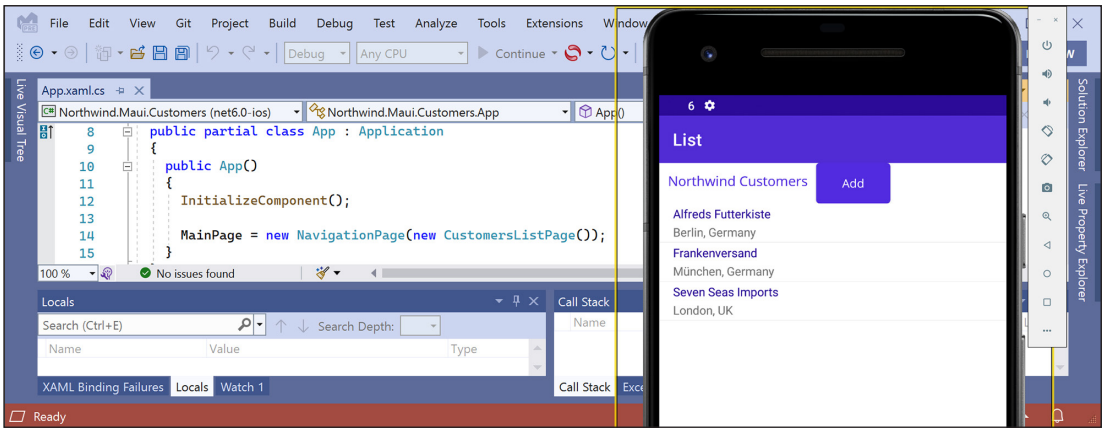


Figure 19.7: The Android device emulator running the Northwind Customers .NET MAUI app

3. Click **Seven Seas Imports** and modify **Company Name** to **Seven Oceans Imports**, as shown in the following screenshot of the customer detail page in *Figure 19.8*:

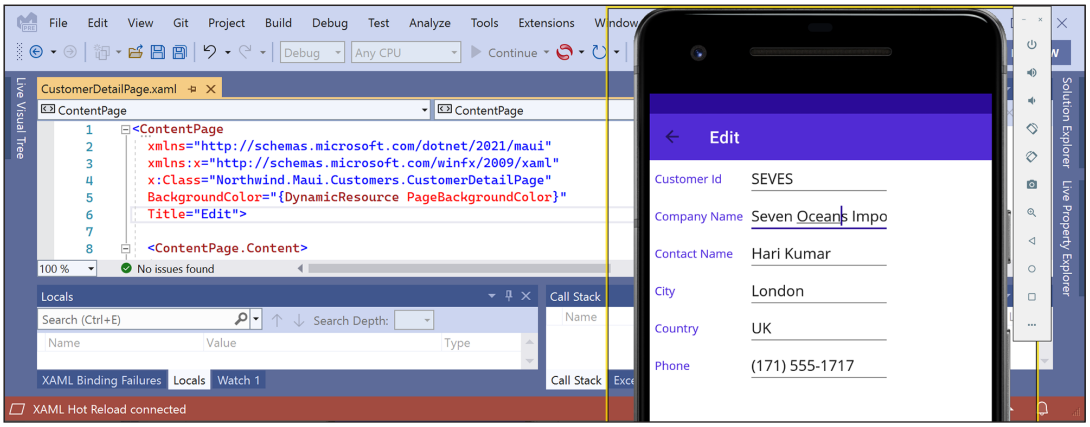


Figure 19.8: Editing a company name on the customer detail page

4. Click the back button to return to the list of customers and note that the company name has been updated due to the two-way data binding.
5. Click **Add**, and then fill in the fields for a new customer.

By default, in the Android device emulator, the virtual keyboard is shown when typing on a physical keyboard. To hide the virtual keyboard, click the keyboard icon to the right of the square Android soft button and then toggle **Show virtual keyboard**.

6. On the customer detail page, click **Insert Customer**, and after being returned to the list of customers, note that the new customer has been added to the bottom of the list. (At the time of writing using .NET MAUI Preview 9, there is a bug that means the list view does not update properly. Click, hold, and drag down on the list view and then release to refresh it.)
7. Click and hold on one of the customers to reveal two action buttons, **Phone** and **Delete**, as shown in *Figure 19.9*:

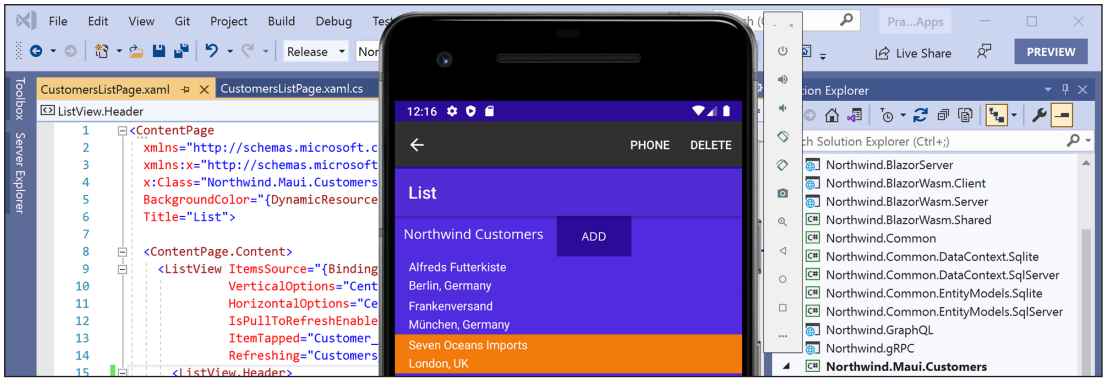


Figure 19.9: Extra commands for a selected customer

8. Click **Phone** and note the pop-up prompt to the user to dial the number of that customer with **Yes** and **No** buttons.
9. Click **No**.
10. Click and hold on one of the customers to reveal two action buttons, **Phone** and **Delete**, and then click on **Delete** and note that the customer is removed.
11. Click, hold, and drag the list down and then release, and note the animation effect for refreshing the list, but remember that this feature is simulated, so the list does not change.
12. Close the Android device emulator.

We will now make the app call the `Northwind.WebApi` service to get the list of customers.

Consuming a web service from a mobile app

Apple's **App Transport Security** (**ATS**) forces developers to use good practice, including secure connections between an app and a web service. ATS is enabled by default and your mobile apps will throw an exception if they do not connect securely.

If you need to call a web service that is secured with a self-signed certificate like our `Northwind.WebApi` service is, it is possible but complicated. For simplicity, we will allow insecure connections to the web service and disable the security checks in the mobile app.

Configuring the web service to allow insecure requests

First, we will enable the web service to handle insecure connections at a new URL:

1. In the `Northwind.WebApi` project, in `Program.cs`, in the section that configures the HTTP pipeline, comment out the HTTPS redirection, as shown in the following code:

```
// commented out for the .NET MAUI app project to use
// app.UseHttpsRedirection();
```

2. In `Program.cs`, in the `UseUrls` method, add the insecure URL, as shown highlighted in the following code:

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

builder.WebHost.UseUrls(
    "https://localhost:5002"
    , "http://localhost:5008" // for .NET MAUI client
  );
```

3. Start the `Northwind.WebApi` web service project without debugging.
4. Start Chrome and test that the web service is returning customers as JSON by navigating to the following URL: `http://localhost:5008/api/customers/`.
5. Close Chrome but leave the web service running.

Configuring the iOS app to allow insecure connections

Now you will configure the `Northwind.Maui.Customers` project to disable ATS to allow insecure HTTP requests to the web service:

1. In the `Northwind.Maui.Customers` project, in the `Platforms/iOS` folder, open the `Info.plist` file by right-clicking and open it with the **XML (Text) Editor**.
2. At the bottom of the dictionary, add a new key named **NSAppTransportSecurity**, which is a dictionary, and in it, add a key named **NSAllowsArbitraryLoads** that has a value of `true`, as shown highlighted in the following partial markup:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
          "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>LSRequiresIPhoneOS</key>
  <true/>
  ...
  <key>NSAppTransportSecurity</key>
  <dict>
```

```
    <key>NSAllowsArbitraryLoads</key>
    <true/>
  </dict>
</dict>
</plist>
```

3. Save and close `Info.plist`.

Configuring the Android app to allow insecure connections

In a similar way to Apple and ATS, with Android 9 (API level 28) cleartext (that is, non-HTTPS), support is disabled by default.

Now you will configure the project to enable cleartext to allow insecure HTTP requests to the web service:

1. In the `Platforms/Android` folder, in the `Properties` folder, open `MainApplication.cs`.
2. In the `Application` attribute, enable cleartext, as shown highlighted in the following code:

```
namespace Northwind.Maui.Customers
{
  [Application(UsesCleartextTraffic = true)]
  public class MainApplication : MauiApplication
```

Getting customers from the web service

Now, we can modify the customers list page to get its list of customers from the web service instead of using sample data:

1. In the `Northwind.Maui.Customers` project, open `CustomersListPage.xaml.cs`.
2. Import the following additional namespaces:

```
using System.Collections.Generic; // IEnumerable<T>
using System.Linq; // OrderBy
using System.Net.Http; // HttpClient
using System.Net.Http.Headers; // MediaTypeWithQualityHeaderValue
using System.Net.Http.Json; // ReadFromJsonAsync<T>
```

3. Modify the `CustomersListPage` constructor to load the list of customers using the service proxy and only call the `AddSampleData` method if an exception occurs, as shown in the following code:

```
public CustomersListPage()
{
  InitializeComponent();
```

```
    CustomersListViewModel viewModel = new();

    try
    {
      HttpClient client = new()
      {
        BaseAddress = new Uri("http://localhost:5008/")
      };

      client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json"));

      HttpResponseMessage response = client
        .GetAsync("api/customers").Result;

      response.EnsureSuccessStatusCode();

      IEnumerable<CustomerDetailViewModel> customersFromService =
        response.Content.ReadFromJsonAsync
        <IEnumerable<CustomerDetailViewModel>>().Result;

      foreach (CustomerDetailViewModel c in customersFromService
        .OrderBy(customer => customer.CompanyName))
      {
        viewModel.Add(c);
      }
    }
    catch (Exception ex)
    {
      DisplayAlert(title: "Exception",
        message: $"App will use sample data due to: {ex.Message}",
        cancel: "OK");

      viewModel.AddSampleData();
    }

    BindingContext = viewModel;
  }
```

4. Navigate to **Build | Clean Northwind.Maui.Customers** because changes to `Info.plist`, such as allowing insecure connections, sometimes require a clean build.
5. Navigate to **Build | Build Northwind.Maui.Customers**.
6. Run the `Northwind.Maui.Customers` project in the Android emulator and note that 91 customers are loaded from the web service.
7. Close the Android emulator.

Practicing and exploring

Test your knowledge and understanding by answering some questions, get some hands-on practice, and explore this chapter's topics with more in-depth research.

Exercise 19.1 – Test your knowledge

Answer the following questions:

1. What are the four categories of .NET MAUI user interface components, and what do they represent?
2. List four types of cell.
3. How can you enable a user to perform an action on a cell in a list view?
4. When would you use an `Entry` instead of an `Editor`?
5. What is the effect of setting `IsDestructive` to `true` for a menu item in a cell's context actions?
6. When would you call the methods `PushAsync` and `PopAsync` in a .NET MAUI app?
7. What is the difference between `Margin` and `Padding` for an element like a `Button`?
8. How are event handlers attached to an object using XAML?
9. What do XAML styles do?
10. Where can you define resources?

Exercise 19.2 – Explore topics

Use the links on the following page to learn more about the topics covered in this chapter:

`https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-19---building-mobile-and-desktop-apps-using-net-maui`

Summary

In this chapter, you learned how to build a cross-platform mobile and desktop app using .NET MAUI, which consumed data from a web service.

In the next chapter, you will learn to protect data and files using hashing, signing encryption, authentication, and authorization.

20

Protecting Your Data and Applications

This chapter is about protecting your data from being viewed by malicious users using encryption and from being manipulated or corrupted using hashing and signing.

In .NET Core 2.1, Microsoft introduced `Span<T>`-based cryptography APIs for hashing, random number generation, asymmetric signature generation and processing, and **Rivest-Shamir-Adleman** (**RSA**) encryption.

Cryptographic operations are performed by operating system implementations so that when an OS has a security vulnerability fixed, then .NET apps benefit immediately. But this means that those .NET apps can only use features that an OS supports. You can read about which features are supported by which OS at the following link:

`https://docs.microsoft.com/en-us/dotnet/standard/security/cross-platform-cryptography`

This chapter covers the following topics:

- Understanding the vocabulary of protection
- Encrypting and decrypting data
- Hashing data
- Signing data
- Generating random numbers
- Authenticating and authorizing users

Warning! The code in this chapter shows security primitives for basic educational purposes only. You must not use any of the code in this chapter for production libraries and apps. Only use professionally written security libraries that are built using these security primitives and that have been hardened for real-world use following the latest best security practices.

Understanding the vocabulary of protection

There are many techniques for protecting your data; below we'll briefly introduce some of the most popular ones and you will see more detailed explanations and practical implementations throughout this chapter:

- **Encryption and decryption**: These are a two-way process for converting your data from cleartext into ciphertext and back again.
- **Hashes**: This is a one-way process for generating a hash value to securely store passwords or can be used to detect malicious changes or corruption of your data. Simple hashes should not be used for passwords. You should use PBKDF2, bcrypt, or scrypt because these guarantee that there cannot be two inputs that generate the same hash.
- **Signatures**: This technique is used to ensure that data has come from a claimed source by validating a signature that has been applied to some data against someone's public key.
- **Authentication**: This technique is used to identify someone by checking their credentials.
- **Authorization**: This technique is used to ensure that someone has permission to perform an action or work with some data by checking the roles or groups they belong to.

Good Practice: If security is important to you (and it should be!), then hire an experienced security expert for guidance rather than relying on advice found online. It is very easy to make small mistakes and leave your applications and data vulnerable without realizing until it is too late!

Keys and key sizes

Protection algorithms often use a **key**. Keys are represented by byte arrays of varying sizes. Keys are used for various purposes, as shown in the following list:

- Encryption and decryption: AES, 3DES, RC2, Rijndael, RSA.
- Signing and verifying: RSA, ECDSA, DSA.
- Message authentication and validation: HMAC.
- Key agreement: Diffie-Hellman, Elliptical Curve Diffie-Hellman.

Good Practice: Choose a bigger key size for stronger protection. This is an oversimplification because some RSA implementations support up to 16,384-bit keys that can take days to generate and would be overkill in most scenarios. A 2048-bit key should be sufficient until the year 2030, at which point you should upgrade to 3192-bit keys.

Keys for encryption and decryption can be **symmetric** (also known as **shared** or **secret** because the same key is used to encrypt and decrypt and therefore must be kept safe) or **asymmetric** (a public-private key pair where the public key is used to encrypt and only the private key can be used to decrypt).

Good Practice: Symmetric key encryption algorithms are fast and can encrypt large amounts of data using a stream. Asymmetric key encryption algorithms are slow and can only encrypt small byte arrays. The most common use of asymmetric keys is signature creation and validation.

In the real world, get the best of both worlds by using a symmetric key to encrypt your data, and an asymmetric key to share the symmetric key. This is how **Secure Sockets Layer** (**SSL**) 2.0 encryption on the internet worked in 1995. Today, what is still often called SSL is actually **Transport Layer Security** (**TLS**), which uses key agreement rather than RSA-encrypted session keys.

Keys come in various byte array sizes.

IVs and block sizes

When encrypting large amounts of data, there are likely to be repeating sequences. For example, in an English document, in the sequence of characters, `the` would appear frequently, and each time it might get encrypted as `hQ2`. A good cracker would use this knowledge to make it easier to crack the encryption, as shown in the following output:

```
When the wind blew hard the umbrella broke.
5:s4&hQ2aj#D f9d1d£8fh"&hQ2s0)an DF8SFd#][1
```

We can avoid repeating sequences by dividing data into **blocks**. After encrypting a block, a byte array value is generated from that block, and this value is fed into the next block to adjust the algorithm. The next block is encrypted so the output is different even for the same input as the preceding block. To encrypt the first block, we need a byte array to feed in. This is called the **initialization vector** (**IV**).

An IV should:

- Be generated randomly along with every encrypted message.
- Be transmitted along with the encrypted message.
- Not itself be a secret.

Salts

A **salt** is a random byte array that is used as an additional input to a one-way hash function. If you do not use a salt when generating hashes, then when many of your users register with 123456 as their password (about 8% of users still did this in 2016!), they will all have the same hashed value, and their accounts will be vulnerable to a dictionary attack.

When a user registers, the salt should be randomly generated and concatenated with their chosen password before being hashed. The output (but not the original password) is stored with the salt in the database.

Then, when the user logs in next and enters their password, you look up their salt, concatenate it with the entered password, regenerate a hash, and then compare its value with the hash stored in the database. If they are the same, you know they entered the correct password.

Even salting passwords is not enough for truly secure storage. You should do a lot more work, such as PBKDF2, bcrypt, or scrypt, but such work is beyond the scope of this book.

Generating keys and IVs

Keys and IVs are byte arrays. Both parties that want to exchange encrypted data need the key and IV values, but byte arrays can be difficult to exchange reliably.

You can reliably generate a key or IV using a **password-based key derivation function** (**PBKDF2**). A good one is the `Rfc2898DeriveBytes` class, which takes a password, a salt, an iteration count, and a hash algorithm (the default is SHA-1, which is no longer recommended). It then generates keys and IVs by making calls to its `GetBytes` method. The iteration count is the number of times that the password is hashed during the process. The more iterations, the harder it will be to crack.

Although the `Rfc2898DeriveBytes` class can be used to generate the IV as well as the key, the IV should be randomly generated each time and transmitted with the encrypted message as plaintext because it does not need to be secret.

Good Practice: The salt size should be 8 bytes or larger, and the iteration count should be a value that takes about 100ms to generate a key and IV for the encryption algorithm on the target machine. This value will increase over time as CPUs improve. In the example code written below, we use 150,000, but that value will already be too low for some computers by the time you read this.

Encrypting and decrypting data

In .NET, there are multiple encryption algorithms you can choose from.

In legacy .NET Framework, some algorithms are implemented by the operating system and their names are suffixed with `CryptoServiceProvider` or `Cng`. Some algorithms are implemented in the .NET BCL and their names are suffixed with `Managed`.

In modern .NET, all algorithms are implemented by the operating system. If the OS algorithms are certified by the **Federal Information Processing Standards** (**FIPS**), then .NET uses FIPS-certified algorithms.

Generally, you will always use an abstract class like `Aes` and its `Create` factory method to get an instance of an algorithm, so you will not need to know if you are using `CryptoServiceProvider` or `Managed` anyway.

Some algorithms use symmetric keys, and some use asymmetric keys. The main asymmetric encryption algorithm is RSA. Ron Rivest, Adi Shamir, and Leonard Adleman described the algorithm in 1977. A similar algorithm was designed in 1973 by Clifford Cocks, an English mathematician working for GCHQ, the British intelligence agency, but it was not declassified until 1997 so Rivest, Shamir, and Adleman got the credit and had their names immortalized in the RSA acronym.

Symmetric encryption algorithms use `CryptoStream` to encrypt or decrypt large amounts of bytes efficiently. Asymmetric algorithms can only handle small amounts of bytes; stored in a byte array instead of a stream.

The most common symmetric encryption algorithms derive from the abstract class named `SymmetricAlgorithm` and are shown in the following list:

- AES
- DESCryptoServiceProvider
- TripleDES
- RC2CryptoServiceProvider
- RijndaelManaged

If you need to write code to decrypt some data sent by an external system, then you will have to use whatever algorithm the external system used to encrypt the data. Or, if you need to send encrypted data to a system that can only decrypt using a specific algorithm, then again, you will not have a choice of algorithm.

If your code will both encrypt and decrypt, then you can choose the algorithm that best suits your requirements for strength, performance, and so on.

Good Practice: Choose the **Advanced Encryption Standard** (**AES**), which is based on the Rijndael algorithm, for symmetric encryption. Choose RSA for asymmetric encryption. Do not confuse RSA with DSA. **Digital Signature Algorithm** (**DSA**) cannot encrypt data. It can only generate and verify signatures.

Encrypting symmetrically with AES

To make it easier to reuse your protection code in multiple projects, we will create a static class named `Protector` in its own class library and then reference it in a console app.

Let's go!

1. Use your preferred code editor to create a new solution/workspace named `Chapter20`.
2. Add a console app project, as defined in the following list:
    1. Project template: **Console Application** / `console`
    2. Workspace/solution file and folder: `Chapter20`
    3. Project file and folder: `EncryptionApp`

3. Add a new class library named `CryptographyLib` to the `Chapter20` solution/workspace.
    1. In Visual Studio, set the startup project for the solution to the current selection.
    2. In Visual Studio Code, select `EncryptionApp` as the active OmniSharp project.
4. In the `CryptographyLib` project, rename the `Class1.cs` file to `Protector.cs`.
5. In the `EncryptionApp` project, add a project reference to the `CryptographyLib` library, as shown in the following markup:

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

6. Build the `EncryptionApp` project and make sure there are no compile errors.
7. Open the `Protector.cs` file and change its contents to define a static class named `Protector` with fields for storing a salt byte array and a large number of iterations, and methods to `Encrypt` and `Decrypt`, as shown in the following code:

```
using System.Diagnostics;
using System.Security.Cryptography;
using System.Security.Principal;
using System.Text;
using System.Xml.Linq;

using static System.Console;
using static System.Convert;

namespace Packt.Shared
{
  public static class Protector
  {
    // salt size must be at least 8 bytes, we will use 16 bytes
    private static readonly byte[] salt =
      Encoding.Unicode.GetBytes("7BANANAS");

    // iterations should be high enough to take at least 100ms to
    // generate a Key and IV on the target machine. 150,000 iterations
    // takes 139ms on my 11th Gen Intel Core i7-1165G7 @ 2.80GHz.
    private static readonly int iterations = 150_000;

    public static string Encrypt(
      string plainText, string password)
    {
      byte[] encryptedBytes;
      byte[] plainBytes = Encoding.Unicode.GetBytes(plainText);
```

```
      using (Aes aes = Aes.Create()) // abstract class factory method
      {
        // record how long it takes to generate the Key and IV
        Stopwatch timer = Stopwatch.StartNew();

        using (Rfc2898DeriveBytes pbkdf2 = new(
          password, salt, iterations))
        {
          aes.Key = pbkdf2.GetBytes(32); // set a 256-bit key
          aes.IV = pbkdf2.GetBytes(16); // set a 128-bit IV
        }

        timer.Stop();

        WriteLine("{0:N0} milliseconds to generate Key and IV using {1:N0}
iterations.",
          arg0: timer.ElapsedMilliseconds,
          arg1: iterations);

        using (MemoryStream ms = new())
        {
          using (ICryptoTransform transformer = aes.CreateEncryptor())
          {
            using (CryptoStream cs = new(
              ms, transformer, CryptoStreamMode.Write))
            {
              cs.Write(plainBytes, 0, plainBytes.Length);
            }
          }
          encryptedBytes = ms.ToArray();
        }
      }

      return ToBase64String(encryptedBytes);
    }

    public static string Decrypt(
      string cipherText, string password)
    {
      byte[] plainBytes;
      byte[] cryptoBytes = FromBase64String(cipherText);

      using (Aes aes = Aes.Create())
      {
```

```
        using (Rfc2898DeriveBytes pbkdf2 = new(
          password, salt, iterations))
        {
          aes.Key = pbkdf2.GetBytes(32);
          aes.IV = pbkdf2.GetBytes(16);
        }

        using (MemoryStream ms = new())
        {
          using (ICryptoTransform transformer = aes.CreateDecryptor())
          {
            using (CryptoStream cs = new(
              ms, transformer, CryptoStreamMode.Write))
            {
              cs.Write(cryptoBytes, 0, cryptoBytes.Length);
            }
          }
          plainBytes = ms.ToArray();
        }
      }

      return Encoding.Unicode.GetString(plainBytes);
    }
  }
}
```

Note the following points about the preceding code:

- Although the salt and iteration count can be hardcoded (but preferably stored in the message itself), the password must be passed as a parameter at runtime when calling the `Encrypt` and `Decrypt` methods.
- We use a temporary `MemoryStream` type to store the results of encrypting and decrypting, and then call `ToArray` to turn the stream into a byte array.
- We convert the encrypted byte arrays to and from a `Base64` encoding to make them easier to read for humans.

Good Practice: Never hardcode a password in your source code because, even after compilation, the password can be read in the assembly by using disassembler tools.

8. In the `EncryptionApp` project, open the `Program.cs` file and then import the namespace for the `Protector` class, the namespace for the `CryptographicException` class, and statically import the `Console` class, as shown in the following code:

```
using System.Security.Cryptography; // CryptographicException
using Packt.Shared; // Protector

using static System.Console;
```

9. In `Program.cs`, add statements to prompt the user for a message and a password, and then encrypt and decrypt, as shown in the following code:

```
Write("Enter a message that you want to encrypt: ");
string? message = ReadLine();

Write("Enter a password: ");
string? password = ReadLine();

if ((password is null) || (message is null))
{
  WriteLine("Message or password cannot be null.");
  return;
}

string cipherText = Protector.Encrypt(message, password);

WriteLine($"Encrypted text: {cipherText}");

Write("Enter the password: ");
string? password2Decrypt = ReadLine();

if (password2Decrypt is null)
{
 WriteLine("Password to decrypt cannot be null.");
 return;
}

try
{
  string clearText = Protector.Decrypt(cipherText, password2Decrypt);
  WriteLine($"Decrypted text: {clearText}");
}
catch (CryptographicException ex)
{
  WriteLine("{0}\nMore details: {1}",
    arg0: "You entered the wrong password!",
    arg1: ex.Message);
}
catch (Exception ex)
```

```
{
  WriteLine("Non-cryptographic exception: {0}, {1}",
    arg0: ex.GetType().Name,
    arg1: ex.Message);
}
```

10. Run the code, try entering a message and password to encrypt, enter the same password to decrypt, and view the result, as shown in the following output:

```
Enter a message that you want to encrypt: Hello Bob
Enter a password: secret
139 milliseconds to generate Key and IV using 150,000 iterations.
Encrypted text: eWt8sgL7aSt5DC9g74ONEPO7mjd55lXB/MmCZpUsFE0=
Enter the password: secret
Decrypted text: Hello Bob
```

If your output shows the number of milliseconds less than 100, then increase the number of iterations until the number of milliseconds is 100 or higher. Note that a different number of iterations will affect the hashed value so it will look different from the above output.

11. Rerun the code and try entering a message and password to encrypt, but this time enter the password incorrectly to decrypt and view the result, as shown in the following output:

```
Enter a message that you want to encrypt: Hello Bob
Enter a password: secret
134 milliseconds to generate Key and IV using 150,000 iterations.
Encrypted text: eWt8sgL7aSt5DC9g74ONEPO7mjd55lXB/MmCZpUsFE0=
Enter the password: 123456
You entered the wrong password!
More details: Padding is invalid and cannot be removed.
```

Good Practice: To support future encryption upgrades, record information about what choices you made, for example, AES-256, CBC mode with PKCS#7 padding, and PBKDF2 and its hash algorithm and iteration count. This is known as cryptographic agility.

Hashing data

In .NET, there are multiple hash algorithms you can choose from. Some do not use any key, some use symmetric keys, and some use asymmetric keys.

There are two important factors to consider when choosing a hash algorithm:

- **Collision resistance**: How rare is it to find two inputs that share the same hash?
- **Preimage resistance**: For a hash, how difficult would it be to find another input that shares the same hash?

Some common non-keyed hashing algorithms are shown in the following table:

Algorithm	Hash size	Description
MD5	16 bytes	This is commonly used because it is fast, but it is not collision-resistant.
SHA1	20 bytes	The use of SHA1 on the internet has been deprecated since 2011.
SHA256, SHA384, SHA512	32 bytes, 48 bytes, 64 bytes	These are the **Secure Hashing Algorithm 2nd generation** (**SHA2**) algorithms with different hash sizes.

Good Practice: Avoid MD5 and SHA1 because they have known weaknesses. Choose a larger hash size to reduce the possibility of repeated hashes. The first publicly known MD5 collision happened in 2010. The first publicly known SHA1 collision happened in 2017. You can read more at the following link: `https://arstechnica.co.uk/information-technology/2017/02/at-deaths-door-for-years-widely-used-sha1-function-is-now-dead/`

Hashing with the commonly used SHA256

We will now add a class to represent a user stored in memory, a file, or a database. We will use a dictionary to store multiple users in memory:

1. In the `CryptographyLib` class library project, add a new class file named `User.cs`, and give it three properties for storing a user's name, a random salt value, and their salted and hashed password, as shown in the following code:

```
namespace Packt.Shared;

public class User
{
  public string Name { get; set; }
  public string Salt { get; set; }
  public string SaltedHashedPassword { get; set; }

  public User(string name, string salt,
    string saltedHashedPassword)
  {
    Name = name;
    Salt = salt;
    SaltedHashedPassword = saltedHashedPassword;
  }
}
```

2. In the `Protector` class, add statements to declare a dictionary to store users and define two methods, one to register a new user and one to validate their password when they subsequently log in, as shown in the following code:

```
private static Dictionary<string, User> Users = new();

public static User Register(
  string username, string password)
{
  // generate a random salt
  RandomNumberGenerator rng = RandomNumberGenerator.Create();
  byte[] saltBytes = new byte[16];
  rng.GetBytes(saltBytes);
  string saltText = ToBase64String(saltBytes);

  // generate the salted and hashed password
  string saltedhashedPassword = SaltAndHashPassword(password, saltText);

  User user = new(username, saltText, saltedhashedPassword);

  Users.Add(user.Name, user);

  return user;
}

// check a user's password that is stored
// in the private static dictionary Users
public static bool CheckPassword(string username, string password)
{
  if (!Users.ContainsKey(username))
  {
    return false;
  }

  User u = Users[username];

  return CheckPassword(password,
    u.Salt, u.SaltedHashedPassword);
}

// check a user's password using salt and hashed password
public static bool CheckPassword(string password,
  string salt, string hashedPassword)
{
  // re-generate the salted and hashed password
  string saltedhashedPassword = SaltAndHashPassword(
```

```
    password, salt);

  return (saltedhashedPassword == hashedPassword);
}

private static string SaltAndHashPassword(string password, string salt)
{
  using (SHA256 sha = SHA256.Create())
  {
    string saltedPassword = password + salt;
    return ToBase64String(sha.ComputeHash(
      Encoding.Unicode.GetBytes(saltedPassword)));
  }
}
```

3. Use your preferred code editor to add a new console app named `HashingApp` to the `Chapter20` solution/workspace.
4. In Visual Studio Code, select `HashingApp` as the active OmniSharp project.
5. In the `HashingApp` project, add a project reference to `CryptographyLib`.
6. Build the `HashingApp` project and make sure there are no compile errors.
7. In `Program.cs`, import the `Packt.Shared` namespace.
8. In `Program.cs`, add statements to register a user and prompt to register a second user, and then prompt to log in as one of those users and validate the password, as shown in the following code:

```
WriteLine("Registering Alice with Pa$$w0rd:");
User alice = Protector.Register("Alice", "Pa$$w0rd");

WriteLine($"  Name: {alice.Name}");
WriteLine($"  Salt: {alice.Salt}");
WriteLine("  Password (salted and hashed): {0}",
  arg0: alice.SaltedHashedPassword);
WriteLine();

Write("Enter a new user to register: ");
string? username = ReadLine();

Write($"Enter a password for {username}: ");
string? password = ReadLine();

if ((username is null) || (password is null))
{
  WriteLine("Username or password cannot be null.");
  return;
}
```

```
WriteLine("Registering a new user:");
User newUser = Protector.Register(username, password);
WriteLine($"  Name: {newUser.Name}");
WriteLine($"  Salt: {newUser.Salt}");
WriteLine("  Password (salted and hashed): {0}",
  arg0: newUser.SaltedHashedPassword);
WriteLine();

bool correctPassword = false;

while (!correctPassword)
{
  Write("Enter a username to log in: ");
  string? loginUsername = ReadLine();

  Write("Enter a password to log in: ");
  string? loginPassword = ReadLine();

  if ((loginUsername is null) || (loginPassword is null))
  {
    WriteLine("Login username or password cannot be null.");
    return;
  }

  correctPassword = Protector.CheckPassword(
    loginUsername, loginPassword);

  if (correctPassword)
  {
    WriteLine($"Correct! {loginUsername} has been logged in.");
  }
  else
  {
    WriteLine("Invalid username or password. Try again.");
  }
}
```

9. Run the code, register a new user with the same password as `Alice`, and view the result, as shown in the following output:

```
Registering Alice with Pa$$w0rd:
  Name: Alice
  Salt: I1I1dzIjkd7EYDf/6jaf4w==
  Password (salted and hashed): pIoadjE4W/XaRFkqS3br3UuAuPv/3LVQ8kzj6mvcz
+s=
```

```
Enter a new user to register: Bob
Enter a password for Bob: Pa$$w0rd
Registering a new user:
  Name: Bob
  Salt: 1X7ym/UjxTiuEWBC/vIHpw==
  Password (salted and hashed):
DoBFtDhKeN0aaaLVdErtrZ3mpZSvpWDQ9TXDosTq0sQ=

Enter a username to log in: Alice
Enter a password to log in: secret
Invalid username or password. Try again.
Enter a username to log in: Bob
Enter a password to log in: secret
Invalid username or password. Try again.
Enter a username to log in: Bob
Enter a password to log in: Pa$$w0rd
Correct! Bob has been logged in.
```

Even if two users register with the same password, they have randomly generated salts so that their salted and hashed passwords are different.

Signing data

To prove that some data has come from someone we trust, it can be signed. You do not sign the data itself; instead, you sign a hash of the data, because all the signature algorithms first hash the data as an implementation step. They also allow you to shortcut this step and provide the data already hashed.

We will be using the SHA256 algorithm to generate the hash, combined with the RSA algorithm for signing the hash.

We could use DSA for both hashing and signing. DSA is faster than RSA for generating a signature, but it is slower than RSA for validating a signature. Since a signature is generated once but validated many times, it is best to have faster validation than generation.

Good Practice: DSA is rarely used today. The improved equivalent is Elliptic Curve DSA. Although ECDSA is slower than RSA, it generates a shorter signature with the same level of security.

Signing with SHA256 and RSA

Let's explore signing data and checking the signature with a public key:

1. In the `Protector` class, add statements to declare a field to store a public key as a `string` value, and two methods to generate and validate a signature, as shown in the following code:

```
public static string? PublicKey;

public static string GenerateSignature(string data)
{
  byte[] dataBytes = Encoding.Unicode.GetBytes(data);
  SHA256 sha = SHA256.Create();
  byte[] hashedData = sha.ComputeHash(dataBytes);
  RSA rsa = RSA.Create();

  PublicKey = rsa.ToXmlString(false); // exclude private key

  return ToBase64String(rsa.SignHash(hashedData,
    HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1));
}

public static bool ValidateSignature(
  string data, string signature)
{
  if (PublicKey is null) return false;
  byte[] dataBytes = Encoding.Unicode.GetBytes(data);
  SHA256 sha = SHA256.Create();
  byte[] hashedData = sha.ComputeHash(dataBytes);
  byte[] signatureBytes = FromBase64String(signature);
  RSA rsa = RSA.Create();
  rsa.FromXmlString(PublicKey);
  return rsa.VerifyHash(hashedData, signatureBytes,
    HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
}
```

Note the following from the preceding code:

- Only the public part of the public-private key pair needs to be made available to the code that is checking the signature so that we can pass `false` when we call the `ToXmlString` method. The private part is required to sign data and must be kept secret because anyone with the private part can sign data as if they are you!
- The hash algorithm used to generate the hash from the data by calling the `SignHash` method must match the hash algorithm set when calling the `VerifyHash` method. In the preceding code, we used `SHA256`.

Now we can test signing some data and checking its signature.

2. Use your preferred code editor to add a new console app named `SigningApp` to the `Chapter20` solution/workspace.
3. In Visual Studio Code, select `SigningApp` as the active OmniSharp project.
4. In the `SigningApp` project, add a project reference to `CryptographyLib`.
5. Build the `SigningApp` project and make sure there are no compile errors.
6. In `Program.cs`, import the `Packt.Shared` namespace.
7. Add statements to prompt the user to enter some text, sign it, check its signature, then modify the signature, and check the signature again to deliberately cause a mismatch, as shown in the following code:

```
Write("Enter some text to sign: ");
string? data = ReadLine();

string signature = Protector.GenerateSignature(data);

WriteLine($"Signature: {signature}");
WriteLine("Public key used to check signature:");
WriteLine(Protector.PublicKey);

if (Protector.ValidateSignature(data, signature))
{
  WriteLine("Correct! Signature is valid.");
}
else
{
  WriteLine("Invalid signature.");
}

// simulate a fake signature by replacing the
// first character with an X or Y
string fakeSignature = signature.Replace(signature[0], 'X');
if (fakeSignature == signature)
{
  fakeSignature = signature.Replace(signature[0], 'Y');
}

if (Protector.ValidateSignature(data, fakeSignature))
{
  WriteLine("Correct! Signature is valid.");
}
else
{
  WriteLine($"Invalid signature: {fakeSignature}");
}
```

8. Run the code and enter some text, as shown in the following output (edited for length):

```
Enter some text to sign: The cat sat on the mat.
Signature: BXSTdM...4Wrg==
Public key used to check signature:
<RSAKeyValue><Modulus>nHtwl3...mw3w==</Modulus><Exponent>AQAB</Exponent></
RSAKeyValue>
Correct! Signature is valid.
Invalid signature: XXSTdM...4Wrg==
```

Generating random numbers

Sometimes, you need to generate random numbers, perhaps in a game that simulates rolls of a die, or for use with cryptography in encryption or signing. There are a couple of classes that can generate random numbers in .NET.

Generating random numbers for games and similar apps

In scenarios that don't need truly random numbers like games, you can create an instance of the `Random` class, as shown in the following code example:

```
Random r = new();
```

`Random` has a constructor with a parameter for specifying a seed value used to initialize its pseudo-random number generator, as shown in the following code:

```
Random r = new(Seed: 46378);
```

As you learned in *Chapter 2*, *Speaking C#*, parameter names should use camel case. The developer who defined the constructor for the `Random` class broke this convention! The parameter name should be `seed`, not `Seed`.

Good Practice: Shared seed values act as a secret key, so if you use the same random number generation algorithm with the same seed value in two applications, then they can generate the same "random" sequences of numbers. Sometimes this is necessary, for example, when synchronizing a GPS receiver with a satellite, or when a game needs to randomly generate the same level. But usually, you want to keep your seed secret.

Once you have a `Random` object, you can call its methods to generate random numbers, as shown in the following code examples:

```
// minValue is an inclusive lower bound i.e. 1 is a possible value
// maxValue is an exclusive upper bound i.e. 7 is not a possible value
```

```
int dieRoll = r.Next(minValue: 1, maxValue: 7); // returns 1 to 6

double randomReal = r.NextDouble(); // returns 0.0 to less than 1.0

byte[] arrayOfBytes = new byte[256];
r.NextBytes(arrayOfBytes); // 256 random bytes in an array
```

The `Next` method takes two parameters: `minValue` and `maxValue`. But `maxValue` is not the maximum value that the method returns! It is an *exclusive upper bound*, meaning it is one more than the maximum value. In a similar way, the value returned by the `NextDouble` method is greater than or equal to 0.0, and less than 1.0.

Generating random numbers for cryptography

The `Random` class generates cryptographically weak **pseudo-random** numbers. This is not good enough for cryptography. If the random numbers are not truly random, then they are predictable, and then a cracker can break your protection.

For cryptographically strong pseudo-random numbers, you must use a `RandomNumberGenerator`-derived type, such as `RNGCryptoServiceProvider`.

We will now create a method to generate a truly random byte array that can be used in algorithms like encryption for key and IV values:

1. In the `Protector` class, add statements to define a method to get a random key or IV for use in encryption, as shown in the following code:

```
public static byte[] GetRandomKeyOrIV(int size)
{
  RandomNumberGenerator r = RandomNumberGenerator.Create();
  byte[] data = new byte[size];
  r.GetBytes(data);

  // data is an array now filled with
  // cryptographically strong random bytes
  return data;
}
```

   Now we can test the random bytes generated for a truly random encryption key or IV.

2. Use your preferred code editor to add a new console app named `RandomizingApp` to the `Chapter20` solution/workspace.
3. In Visual Studio Code, select `RandomizingApp` as the active OmniSharp project.
4. In the `RandomizingApp` project, add a project reference to `CryptographyLib`.
5. Build the `RandomizingApp` project and make sure there are no compile errors.
6. In `Program.cs`, import the `Packt.Shared` namespace.

7. Add statements to prompt the user to enter a size of byte array and then generate random byte values and write them to the console, as shown in the following code:

```
Write("How big do you want the key (in bytes): ");
string? size = ReadLine();

byte[] key = Protector.GetRandomKeyOrIV(int.Parse(size));

WriteLine($"Key as byte array:");
for (int b = 0; b < key.Length; b++)
{
  Write($"{key[b]:x2} ");
  if (((b + 1) % 16) == 0) WriteLine();
}
WriteLine();
```

8. Run the code, enter a typical size for the key, such as 256, and view the randomly generated key, as shown in the following output:

```
How big do you want the key (in bytes): 256
Key as byte array:
f1 57 3f 44 80 e7 93 dc 8e 55 04 6c 76 6f 51 b9
e8 84 59 e5 8d eb 08 d5 e6 59 65 20 b1 56 fa 68
...
```

Authenticating and authorizing users

Authentication is the process of verifying the identity of a user by validating their credentials against some authority. Credentials include a username and password combination, or a fingerprint or face scan. Once authenticated, the authority can make claims about the user, for example, what their email address is, and what groups or roles they belong to.

Authorization is the process of verifying the membership of groups or roles before allowing access to resources such as application functions and data. Although authorization can be based on individual identity, it is good security practice to authorize based on group or role membership (that can be indicated via claims) even when there is only one user in the role or group. This is because that allows the user's membership to change in the future without reassigning the user's individual access rights.

For example, instead of assigning access rights to Buckingham Palace to *Elizabeth Alexandra Mary Windsor* (a user), you would assign access rights to the *Monarch of the United Kingdom of Great Britain and Northern Ireland and other realms and territories* (a role) and then add Elizabeth as the only member of that role. Then, at some point in the future, you do not need to change any access rights for the *Monarch* role; you just remove Elizabeth and add the next person in the line of succession. And of course, you would implement the line of succession as a queue.

Authentication and authorization mechanisms

There are multiple authentication and authorization mechanisms to choose from. They all implement a pair of interfaces in the `System.Security.Principal` namespace: `IIdentity` and `IPrincipal`.

Identifying a user

`IIdentity` represents a user, so it has a `Name` property and an `IsAuthenticated` property to indicate whether they are anonymous or whether they have been successfully authenticated from their credentials, as shown in the following code:

```
namespace System.Security.Principal
{
  public interface IIdentity
  {
    string? AuthenticationType { get; }
    bool IsAuthenticated { get; }
    string? Name { get; }
  }
}
```

A common class that implements this interface is `GenericIdentity`, which inherits from `ClaimsIdentity`, as shown in the following code:

```
namespace System.Security.Principal
{
  public class GenericIdentity : ClaimsIdentity
  {
    public GenericIdentity(string name);
    public GenericIdentity(string name, string type);
    protected GenericIdentity(GenericIdentity identity);
    public override string AuthenticationType { get; }
    public override IEnumerable<Claim> Claims { get; }
    public override bool IsAuthenticated { get; }
    public override string Name { get; }
    public override ClaimsIdentity Clone();
  }
}
```

The `Claim` objects have a `Type` property that indicates if the claim is for their name, their membership of a role or group, their date of birth, and so on, as shown in the following code:

```
namespace System.Security.Claims
{
  public class Claim
  {
```

```
    // various constructors

    public string Type { get; }
    public ClaimsIdentity? Subject { get; }
    public IDictionary<string, string> Properties { get; }
    public string OriginalIssuer { get; }
    public string Issuer { get; }
    public string ValueType { get; }
    public string Value { get; }
    protected virtual byte[]? CustomSerializationData { get; }
    public virtual Claim Clone();
    public virtual Claim Clone(ClaimsIdentity? identity);
    public override string ToString();
    public virtual void WriteTo(BinaryWriter writer);
    protected virtual void WriteTo(BinaryWriter writer, byte[]? userData);
  }

  public static class ClaimTypes
  {
    public const string Actor = "http://schemas.xmlsoap.org/ws/2009/09/identity/
claims/actor";
    public const string NameIdentifier = "http://schemas.xmlsoap.org/ws/2005/05/
identity/claims/nameidentifier";
    public const string Name = "http://schemas.xmlsoap.org/ws/2005/05/identity/
claims/name";
    public const string PostalCode = "http://schemas.xmlsoap.org/ws/2005/05/
identity/claims/postalcode";

    // ...many other string constants

    public const string MobilePhone = "http://schemas.xmlsoap.org/ws/2005/05/
identity/claims/mobilephone";
    public const string Role = "http://schemas.microsoft.com/ws/2008/06/identity/
claims/role";
    public const string Webpage = "http://schemas.xmlsoap.org/ws/2005/05/
identity/claims/webpage";
  }
}
```

User membership

`IPrincipal` is used to associate an identity with the roles and groups that they are members of, so it can be used for authorization purposes, as shown in the following code:

```
namespace System.Security.Principal
{
  public interface IPrincipal
```

```
  {
    IIdentity? Identity { get; }
    bool IsInRole(string role);
  }
}
```

The current thread executing your code has a `CurrentPrincipal` property that can be set to any object that implements `IPrincipal`, and it will be checked when permission is needed to perform a secure action.

The most common class that implements this interface is `GenericPrincipal`, which inherits from `ClaimsPrincipal`, as shown in the following code:

```
namespace System.Security.Principal
{
  public class GenericPrincipal : ClaimsPrincipal
  {
    public GenericPrincipal(IIdentity identity, string[]? roles);
    public override IIdentity Identity { get; }
    public override bool IsInRole([NotNullWhen(true)] string? role);
  }
}
```

Implementing authentication and authorization

Let's explore authentication and authorization by implementing a custom authentication and authorization mechanism:

1. In the `CryptographyLib` project, add a property to the `User` class to store an array of roles, as shown in the following code:

```
public string[]? Roles { get; set; }
```

2. In `User.cs`, add a parameter to set the `Roles` in the constructor.
3. Modify the `Register` method in the `Protector` class to allow an array of roles to be passed as an optional parameter, as shown highlighted in the following code:

```
public static User Register(
  string username, string password,
  string[]? roles = null)
```

4. In the `Register` method, add a parameter to set the array of roles in the new `User` object, as shown highlighted in the following code:

```
User user = new(username, saltText,
  saltedhashedPassword, roles);
```

5. In the `CryptographyLib` project, add statements to the `Protector` class to define a `LogIn` method to log in a user, and if the username and password are valid, then create a generic identity and principal and assign them to the current thread, indicating that the type of authentication was a custom one named `PacktAuth`, as shown in the following code:

```
public static void LogIn(string username, string password)
{
  if (CheckPassword(username, password))
  {
    GenericIdentity gi = new(
      name: username, type: "PacktAuth");

    GenericPrincipal gp = new(
      identity: gi, roles: Users[username].Roles);

    // set the principal on the current thread so that
    // it will be used for authorization by default
    Thread.CurrentPrincipal = gp;
  }
}
```

6. Use your preferred code editor to add a new console app named `SecureApp` to the `Chapter20` solution/workspace.
7. In Visual Studio Code, select `SecureApp` as the active OmniSharp project.
8. In the `SecureApp` project, add a project reference to `CryptographyLib`.
9. Build the `SecureApp` project and make sure there are no compile errors.
10. In `Program.cs`, import the required namespaces for working with authentication and authorization, as shown in the following code:

```
using Packt.Shared; // Protector
using System.Security; // SecurityException
using System.Security.Principal; // IPrincipal
using System.Security.Claims; // ClaimsPrincipal, Claim

using static System.Console;
```

11. Write statements to register three users, named `Alice`, `Bob`, and `Eve`, in various roles, prompt the user to log in, and then output information about them, as shown in the following code:

```
Protector.Register("Alice", "Pa$$w0rd",
  roles: new[] { "Admins" });

Protector.Register("Bob", "Pa$$w0rd",
  roles: new[] { "Sales", "TeamLeads" });

// Eve is not a member of any roles
Protector.Register("Eve", "Pa$$w0rd");
```

```
// prompt user to enter username and password to login
// as one of these three users

Write($"Enter your user name: ");
string? username = ReadLine();

Write($"Enter your password: ");
string? password = ReadLine();

if ((username == null) || (password == null))
{
  WriteLine("Username or password is null. Cannot login.");
  return;
}

Protector.LogIn(username, password);

if (Thread.CurrentPrincipal == null)
{
  WriteLine("Log in failed.");
  return;
}

IPrincipal p = Thread.CurrentPrincipal;

WriteLine(
  $"IsAuthenticated: {p.Identity?.IsAuthenticated}");
WriteLine(
  $"AuthenticationType: {p.Identity?.AuthenticationType}");
WriteLine($"Name: {p.Identity?.Name}");
WriteLine($"IsInRole(\"Admins\"): {p.IsInRole("Admins")}");
WriteLine($"IsInRole(\"Sales\"): {p.IsInRole("Sales")}");

if (p is ClaimsPrincipal)
{
  WriteLine(
    $"{p.Identity?.Name} has the following claims:");

  IEnumerable<Claim>? claims = (p as ClaimsPrincipal)?.Claims;

  if (claims is not null)
  {
    foreach (Claim claim in claims)
    {
      WriteLine($"{claim.Type}: {claim.Value}");
    }
  }
}
```

12. Run the code, log in as `Alice` with `Pa$$w0rd`, and view the results, as shown in the following output:

```
Enter your user name: Alice
Enter your password: Pa$$w0rd
IsAuthenticated: True
AuthenticationType: PacktAuth
Name: Alice
IsInRole("Admins"): True
IsInRole("Sales"): False
Alice has the following claims:
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: Alice
http://schemas.microsoft.com/ws/2008/06/identity/claims/role: Admins
```

13. Run the code, log in as `Alice` with `secret`, and view the results, as shown in the following output:

```
Enter your user name: Alice
Enter your password: secret
Log in failed.
```

14. Run the code, log in as `Bob` with `Pa$$w0rd`, and view the results, as shown in the following output:

```
Enter your user name: Bob
Enter your password: Pa$$w0rd
IsAuthenticated: True
AuthenticationType: PacktAuth
Name: Bob
IsInRole("Admins"): False
IsInRole("Sales"): True
Bob has the following claims:
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: Bob
http://schemas.microsoft.com/ws/2008/06/identity/claims/role: Sales
http://schemas.microsoft.com/ws/2008/06/identity/claims/role: TeamLeads
```

Protecting application functionality

Now let's explore how we can use authorization to prevent some users from accessing some features of an application:

1. At the bottom of `Program.cs`, add a method that is secured by checking for permission inside the method, and throw appropriate exceptions if the user is anonymous or not a member of the `Admins` role, as shown in the following code:

```
static void SecureFeature()
{
  if (Thread.CurrentPrincipal == null)
```

```
  {
    throw new SecurityException(
      "A user must be logged in to access this feature.");
  }

  if (!Thread.CurrentPrincipal.IsInRole("Admins"))
  {
    throw new SecurityException(
      "User must be a member of Admins to access this feature.");
  }

  WriteLine("You have access to this secure feature.");
}
```

2. Above the `SecureFeature` method, add statements to call the `SecureFeature` method in a `try` statement, as shown in the following code:

```
try
{
  SecureFeature();
}
catch (Exception ex)
{
  WriteLine($"{ex.GetType()}: {ex.Message}");
}
```

3. Run the code, log in as `Alice` with `Pa$$word`, and view the result, as shown in the following output:

```
You have access to this secure feature.
```

4. Run the code, log in as `Bob` with `Pa$$word`, and view the result, as shown in the following output:

```
System.Security.SecurityException: User must be a member of Admins to
access this feature.
```

Real-world authentication and authorization

Although it is valuable to see some examples of how authentication and authorization can work, in the real world, you should not build your own security systems because it is too likely that you might introduce flaws.

Instead, you should look at commercial or open-source implementations. These usually implement standards such as OAuth 2.0 and OpenID Connect. A popular open-source one is **IdentityServer4**, but it will only be maintained until November 2022. A semi-commercial option is Duende IdentityServer.

Microsoft's official position is that "Microsoft already has a team and a product in that area, Azure Active Directory, which allows 500,000 objects for free." You can read more at the following link:

`https://devblogs.microsoft.com/aspnet/asp-net-core-6-and-authentication-servers/`

Practicing and exploring

Test your knowledge and understanding by answering some questions, get some hands-on practice, and explore the topics covered in this chapter with more in-depth research.

Exercise 20.1 – Test your knowledge

Answer the following questions:

1. Of the encryption algorithms provided by .NET, which is the best choice for symmetric encryption?
2. Of the encryption algorithms provided by .NET, which is the best choice for asymmetric encryption?
3. What is a rainbow attack?
4. For encryption algorithms, is it better to have a larger or smaller block size?
5. What is a cryptographic hash?
6. What is a cryptographic signature?
7. What is the difference between symmetric and asymmetric encryption?
8. What does RSA stand for?
9. Why should passwords be salted before being stored?
10. SHA1 is a hashing algorithm designed by the United States National Security Agency. Why should you never use it?

Exercise 20.2 – Practice protecting data with encryption and hashing

In the `Chapter10` solution/workspace, add a console application named `Exercise02` that protects sensitive data like a credit card number or password stored in an XML file, such as the following example:

```
<?xml version="1.0" encoding="utf-8" ?>
<customers>
  <customer>
    <name>Bob Smith</name>
    <creditcard>1234-5678-9012-3456</creditcard>
```

```
    <password>Pa$$w0rd</password>
  </customer>
  ...
</customers>
```

The customer's credit card number and password are currently stored in cleartext. The credit card number must be encrypted so that it can be decrypted and used later, and the password must be salted and hashed.

Good Practice: You should not store credit card numbers in your applications. This is just an example of a secret that you might want to protect. If you have to store credit card numbers, then there is a lot more you must do to be Payment Card Industry (PCI) compliant.

Exercise 20.3 – Practice protecting data with decryption

In the `Chapter20` solution/workspace, add a console application named `Exercise03` that opens the XML file that you protected in the preceding code and decrypts the credit card number.

Exercise 20.4 – Explore topics

Use the links on the following webpage to learn more about the topics covered in this chapter:

`https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-20---protecting-your-data-and-applications`

Summary

In this chapter, you learned how to encrypt and decrypt using symmetric encryption, how to generate a salted hash, how to sign data and check the signature on the data, how to generate truly random numbers, and how to use authentication and authorization to protect features of your applications.

Appendix

Answers to the Test Your Knowledge Questions

Chapter 18 – Building and Consuming Specialized Services

1. You have an app that communicates with a service built using the legacy Windows Communication Foundation service. What are two possible options for migrating the service and client to modern .NET?

   Answer: Two options for migrating a WCF service and client are (1) use the Core WCF open source project, or (2) re-implement the service and client using gRPC.

2. What transport protocol does an OData service use?

   Answer: OData uses HyperText Transport Protocol (HTTP).

3. Why is an OData web service more flexible than a traditional ASP.NET Core Web API web service?

   Answer: OData uses query strings for its queries that enable the client to control what is returned and minimizes round trips. A traditional Web API defines all the methods and what gets returned.

4. What must you do to an action method in an OData controller to enable query strings to customize what it returns?

   Answer: You must decorate an action method in an OData controller with the `[EnableQuery]` attribute to enable query strings to customize what it returns.

5. What transport protocol does a GraphQL service use?

   Answer: GraphQL can use HTTP or others like WebSocket.

6. How are contracts defined in gRPC?

   Answer: In gRPC, contracts are defined using `.proto` files.

7. What are three benefits of gRPC that make it a good choice for implementing services?

   Answer: Three benefits of gRPC that make it a good choice for implementing services are (1) its Protobuf binary serialization that minimizes network usage, (2) its requirement of HTTP/2 that provides significant performance benefits, and (3) its support by almost all languages and platforms.

8. What transports does SignalR use, and which is the default?

   Answer: SignalR prefers to use WebSockets as its transport, then it will fall back to server-side events, and finally, it will use long polling if neither of the others is supported by the client and server.

9. What is the difference between the in-process and isolated hosting models for Azure Functions?

   Answer: The in-process hosting model requires your Azure Function to be loaded alongside other code and to target a predefined version of an LTS release like .NET Core 3.1 or .NET 6. The isolated hosting model allows your Azure Function to load in its own process and use any version of .NET that you choose.

10. What is good practice for RPC method signature design?

    Answer: Good practice for RPC method signature design is to define a single parameter using a complex type. This allows additional properties to be added to the type in the future without breaking the contract between the client and service.

Chapter 19 – Building Mobile and Desktop Apps Using .NET MAUI

1. What are the four categories of .NET MAUI user interface components and what do they represent?

   Answer: The four categories of .NET MAUI user interface components are:

- Pages: This represents mobile application screens.
- Layouts: This represents the structure of a combination of the user interface components.
- Views: This represents a single user interface component.
- Cells: This represents a single item in a list or table view.

2. List four types of cell.

   Answer: Four types of cell are `TextCell`, `SwitchCell`, `EntryCell`, and `ImageCell`.

3. How can you enable a user to perform an action on a cell in a list view?

   Answer: To enable a user to perform an action on a cell in a list view, you can set some context actions that are menu items that raise an event, as shown in the following markup:

```
<TextCell Text="{Binding CompanyName}"
          Detail="{Binding Location}"
          TextColor="{DynamicResource PrimaryTextColor}"
          DetailColor="{DynamicResource PrimaryTextColor}" >
  <TextCell.ContextActions>
    <MenuItem Clicked="Customer_Phoned" Text="Phone" />
    <MenuItem Clicked="Customer_Deleted" Text="Delete"
              IsDestructive="True" />
  </TextCell.ContextActions>
</TextCell>
```

4. When would you use an `Entry` instead of an `Editor`?

   Answer: Use an `Entry` for a single line of text, and use an `Editor` for multiple lines of text.

5. What is the effect of setting `IsDestructive` to `true` for a menu item in a cell's context actions?

   Answer: The menu item is colored red as a warning to the user.

6. When would you call the methods `PushAsync` and `PopAsync` in a .NET MAUI app?

   Answer: To provide navigation between screens with built-in support to go back to the previous screen, wrap the first screen in a `NavigationPage` when the app first starts, as shown in the following code:

```
MainPage = new NavigationPage(new CustomersListPage());
```

   To go to the next screen, push the next page onto the `Navigation` object, as shown in the following code:

```
await Navigation.PushAsync(new CustomerDetailPage(c));
```

To return to the previous screen, pop the page from the `Navigation` object, as shown in the following code:

```
await Navigation.PopAsync();
```

7. What is the difference between `Margin` and `Padding` for an element like a `Button`?

   Answer: The difference between `Margin` and `Padding` for an element like a `Button` is that `Margin` is outside the `Border`, while `Padding` is inside the `Border`.

8. How are event handlers attached to an object using XAML?

   Answer: Event handlers are attached to an object using XAML by setting an attribute for the event name to the name of a method in the code-behind class, as shown in the following markup:

```
<Button Clicked="SaveButton_Clicked">
```

9. What do XAML styles do?

   Answer: XAML styles enable the setting of one or more properties.

10. Where can you define resources?

    Answer: You can define resources in any element depending on where you want to share those resources:

    - To share resources throughout an app, define resources in the `<Application.Resources>` element.
    - To share resources only within a page, define resources in its `<Page.Resources>` element.
    - To share resources only in a single element like a button, define resources in its `<Button.Resources>` element.

Chapter 20 – Protecting Your Data and Applications

1. Of the encryption algorithms provided by .NET, which is the best choice for symmetric encryption?

   Answer: The AES algorithm is the best choice for symmetric encryption.

2. Of the encryption algorithms provided by .NET, which is the best choice for asymmetric encryption?

   Answer: The RSA algorithm is the best choice for asymmetric encryption.

3. What is a rainbow attack?

   Answer: A rainbow attack uses a table of precalculated hashes of passwords. When a database of password hashes is stolen, the attacker can compare against the rainbow table hashes quickly and determine the original passwords.

4. For encryption algorithms, is it better to have a larger or smaller block size?

   Answer: For encryption algorithms, it is better to have a smaller block size.

5. What is a cryptographic hash?

   Answer: A cryptographic hash is a fixed-size output that results from an input of arbitrary size being processed by a hash function. Hash functions are one-way, which means that the only way to recreate the original input is to brute-force all possible inputs and compare the results.

6. What is a cryptographic signature?

   Answer: A cryptographic signature is a value appended to a digital document to prove its authenticity. A valid signature tells the recipient that the document was created by a known sender and has not been modified.

7. What is the difference between symmetric and asymmetric encryption?

   Answer: Symmetric encryption uses a secret shared key to both encrypt and decrypt. Asymmetric encryption uses a public key to encrypt and a private key to decrypt.

8. What does RSA stand for?

   Answer: Rivest-Shamir-Adleman, the surnames of the three men who publicly described the algorithm in 1978.

9. Why should passwords be salted before being stored?

   Answer: To slow down rainbow dictionary attacks.

10. SHA-1 is a hashing algorithm designed by the United States National Security Agency. Why should you never use it?

    Answer: SHA-1 is no longer secure. All modern browsers have stopped accepting SHA-1 SSL certificates.