

12

Building Efficient Microservices Using gRPC

In this chapter, you will be introduced to gRPC, which enables a developer to build highly efficient services across most platforms. However, web browsers do not have full support for programmatic access to all features of HTTP/2, which is required by gRPC. This makes gRPC most useful for implementing intermediate tier-to-tier services and microservices because they must perform a lot of communication between multiple microservices to achieve a complete task.

Improving the efficiency of that communication is vital to success for the scalability and performance of microservices. A monolithic, two-tier, client-to-service style service could get away with being less efficient because there is only one layer of communication. The more tiers and therefore the more layers of communication there are between microservices, the more important efficient communication between those layers becomes, as shown in *Figure 12.1*:

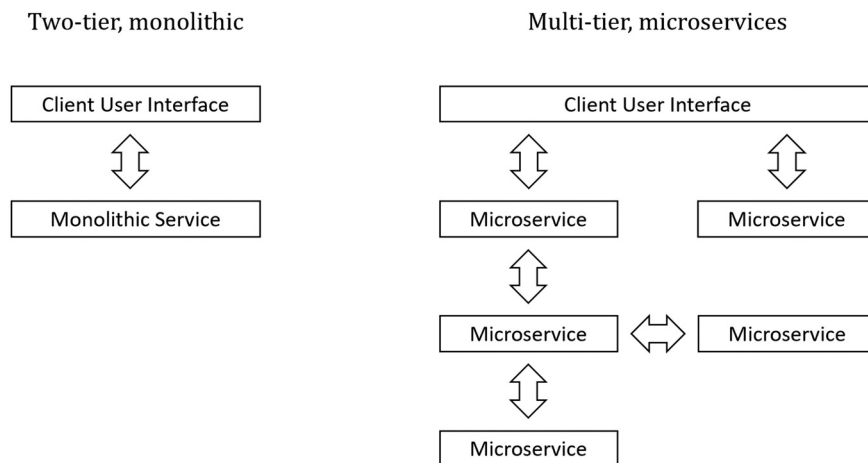


Figure 12.1: Comparing a two-tier monolithic service with multi-tier microservices

This chapter will cover the following topics:

- Understanding gRPC
- Building a gRPC service and client
- Implementing gRPC for an EF Core model
- Implementing gRPC JSON transcoding

Understanding gRPC

gRPC is a modern open-source high-performance **Remote Procedure Call (RPC)** framework that can run in any environment. An RPC is when one computer calls a procedure in another process or on another computer over a network as if it were calling a local procedure. It is an example of a client-server architecture.



You can learn more about the RPCs at the following link: https://en.wikipedia.org/wiki/Remote_procedure_call.

How gRPC works

A gRPC service developer defines a service interface for the methods that can be called remotely, including defining the method parameters and return types. The service 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.

Defining gRPC contracts with .proto files

gRPC uses contract-first API development that supports language-agnostic implementations. A **contract** in this case is an agreement that a service will expose a defined list of methods with specified parameters and return types that implement a prescribed behavior. A client that wishes to call the service can be certain that the service will continue to conform to the contract over time. For example, although new methods might be added, existing ones will never change or be removed.

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 benefits

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, which provides significant performance benefits over earlier versions, like binary framing and compression, and multiplexing of HTTP/2 calls over a single connection.

Binary framing means how the HTTP messages are transferred between the client and server. HTTP/1.x uses newline delimited plaintext. HTTP/2 splits communication into smaller messages (frames) that are encoded in binary format. Multiplexing means combining multiple messages from different sources into a single message to more efficiently use a shared resource like a network transport.

gRPC limitations

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.

Another limitation for developers is that due to the binary format of the messages, it is harder to diagnose and monitor issues. Many tools do not understand the format and cannot show the messages in a human-readable format.

There is an initiative called **gRPC-Web** that adds an extra proxy layer, and the proxy forwards requests to the gRPC server. However, it only supports a subset of gRPC due to the listed limitations.

Types of gRPC methods

gRPC has four types of method:

- **Unary** methods have structured request and response messages. A unary method completes when the response message is returned. Unary methods should be chosen in all scenarios that do not require a stream.
- **Streaming** methods are used when a large amount of data must be exchanged, and they do so using a stream of bytes. They have the `stream` keyword prefix for either an input parameter, an output parameter, or both:
 - **Server streaming** methods receive a request message from the client and return a stream. Multiple messages can be returned over the stream. A server streaming call ends when the server-side method returns, but the server-side method could run until it receives a cancellation token from the client.
 - **Client streaming** methods only receive a stream from the client without any message. The server-side method processes the stream until it is ready to return a response message. Once the server-side method returns its message, the client streaming call is done.
 - **Bi-directional streaming** methods only receive a stream from the client without any message and only return data via a second stream. The call is done when the server-side method returns. Once a bi-directional streaming method is called, the client and service can send messages to each other at any time.

In this book, we will only look at the details of unary methods. If you would like the next edition to cover streaming methods, please let me know.

Microsoft's gRPC packages

Microsoft has invested in building a set of packages for .NET to work with gRPC and, since May 2021, it is Microsoft's recommended implementation of gRPC for .NET.

Microsoft's gRPC for .NET includes:

- `Grpc.AspNetCore` for hosting a gRPC service in ASP.NET Core.
- `Grpc.Net.Client` for adding gRPC client support to any .NET project by building on `HttpClient`.
- `Grpc.Net.ClientFactory` for adding gRPC client support to any .NET code base by building on `HttpClientFactory`.



You can learn more at the following link: <https://github.com/grpc/grpc-dotnet>.

Building a gRPC service and client

Let's see an example service and client for sending and receiving simple messages.

Building a Hello World gRPC service

We will start by building the gRPC service using one of the project templates provided as standard:

1. Use your preferred code editor to create a new project, as defined in the following list:
 - Project template: **ASP.NET Core gRPC Service/grpc**
 - Workspace/solution file and folder: **Chapter12**
 - Project file and folder: **Northwind.Grpc.Service**



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

2. In the `Protos` folder, in `greet.proto`, 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.Service";

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;
}
```

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

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

<ItemGroup>
  <PackageReference Include="Grpc.AspNetCore" Version="2.48.0" />
</ItemGroup>
```

4. In the `Services` folder, in `GreeterService.cs`, note that it inherits from a class named `GreeterBase` and it asynchronously implements the `Greeter` service contract by having a `SayHello` method that accepts a `HelloRequest` input parameter and returns a `HelloReply`, as shown in the following code:

```
using Grpc.Core;
using Northwind.Grpc.Service;

namespace Northwind.Grpc.Service.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
            });
        }
    }
}
```

```

    }
  }
}

```

5. If you are using Visual Studio 2022, in **Solution Explorer**, click **Show All Files**.
6. In the `obj\Debug\net7.0\Protos` folder, note the two class files named `Greet.cs` and `GreetGrpc.cs` that are automatically generated from the `greet.proto` file, as shown in *Figure 12.2*:

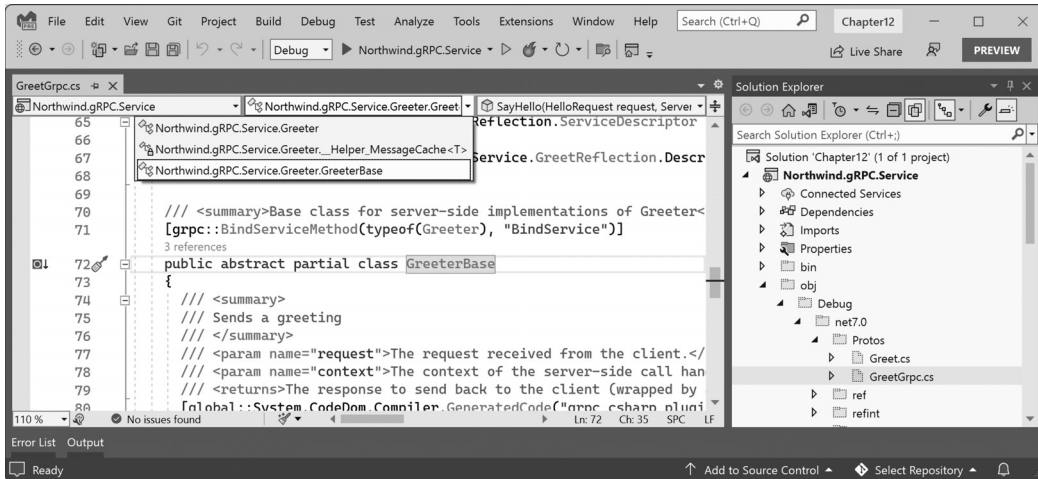


Figure 12.2: The autogenerated class files from a .proto file for a gRPC service

7. In `GreetGrpc.cs`, note the `Greeter.GreeterBase` class that the `GreeterService` class inherited from. You do not need to understand how this base class is implemented, but you should know it is what handles all the details of gRPC's efficient communication.
8. If you are using Visual Studio 2022, in **Solution Explorer**, expand **Dependencies**, expand **Packages**, expand **Grpc.AspNetCore**, and note that it has dependencies on Google's `Google.Protobuf` package, and Microsoft's `Grpc.AspNetCore.Server.ClientFactory` and `Grpc.Tools` packages, as shown in *Figure 12.3*:

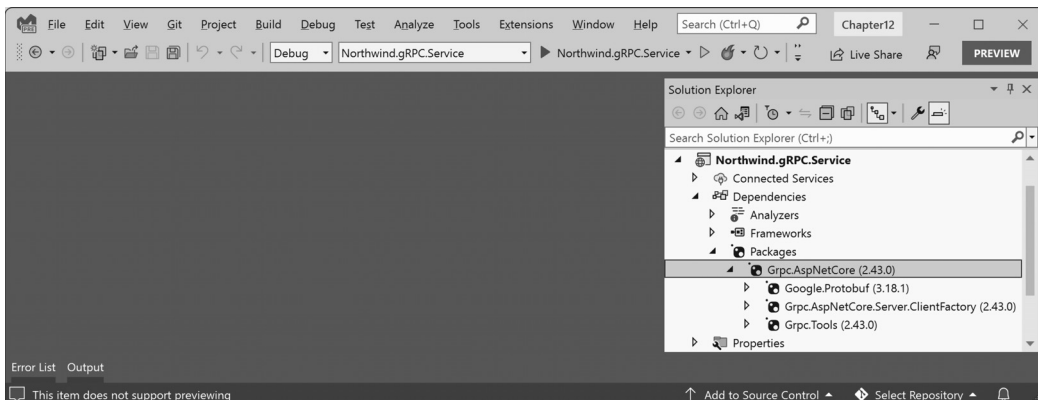


Figure 12.3: The `Grpc.AspNetCore` package references the `Grpc.Tools` and `Google.Protobuf` packages



The `Grpc.Tools` package generates the C# class files from the registered `.proto` files, and those class files use types defined in Google's package to implement the serialization to the Protobuf serialization format. The `Grpc.AspNetCore.Server.ClientFactory` package includes both server-side and client-side support for gRPC in a .NET project.

9. 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();
```

10. 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>();
```

11. In the `Properties` folder, open `launchSettings.json` and modify the `applicationUrl` setting to use port 5121, as shown highlighted in the following markup:

```
{
  "profiles": {
    "Northwind.Grpc.Service": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "applicationUrl": "https://localhost:5121",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

12. Build the `Northwind.Grpc.Service` project.

Building a Hello World gRPC client

We will add an ASP.NET Core MVC website project and then add the gRPC client packages to enable it to call the gRPC service:

1. Use your preferred code editor to add a new project, as defined in the following list:
 - Project template: **ASP.NET Core Web App (Model-View-Controller)/mvc**
 - Workspace/solution file and folder: `Chapter12`
 - Project file and folder: `Northwind.Grpc.Client.Mvc`
 - If you are using Visual Studio 2022, set the startup project to the current selection.

2. In the `Northwind.Grpc.Client.Mvc` project, add package references for Microsoft's gRPC client factory and tools, and Google's .NET library for Protocol Buffers, as shown in the following markup:

```
<ItemGroup>
  <PackageReference Include="Grpc.Net.ClientFactory" Version="2.48.0" />
  <PackageReference Include="Grpc.Tools" Version="2.48.0">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles;
      analyzers; buildtransitive</IncludeAssets>
  </PackageReference>
  <PackageReference Include="Google.Protobuf" Version="3.21.5" />
</ItemGroup>
```



Good Practice: The `Grpc.Net.ClientFactory` package references the `Grpc.Net.Client` package that implements client-side support for gRPC in a .NET project, but it does not reference other packages like `Grpc.Tools` or `Google.Protobuf`. We must reference those packages explicitly. 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.

3. In the Properties folder, open `launchSettings.json` and modify the `applicationUrl` setting to use port 5122, as shown highlighted in the following markup:

```
{
  "profiles": {
    "Northwind.Grpc.Client.Mvc": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "applicationUrl": "https://localhost:5122",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

4. Copy the `Protos` folder from the `Northwind.Grpc.Service` project/folder to the `Northwind.Grpc.Client.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.

5. In the `Northwind.Grpc.Client.Mvc` project, in the `Protos` folder, 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.Grpc.Client.Mvc";
```

6. In the `Northwind.Grpc.Client.Mvc` project file, add or modify the item group that registers the `.proto` file to indicate that it is being used on the client side, as shown highlighted in the following markup:

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



Visual Studio 2022 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`.

7. Build the `Northwind.Grpc.Client.Mvc` project to ensure that the automatically generated classes are created.
8. In the `obj\Debug\net7.0\Protos` folder, in `GreetGrpc.cs`, note the `Greeter.GreeterClient` class, as partially shown in the following code:

```
public static partial class Greeter
{
    ...
    public partial class GreeterClient : grpc::ClientBase<GreeterClient>
    {
```

9. In `Program.cs`, import the namespace for `Greeter.GreeterClient`, as shown in the following code:

```
using Northwind.Grpc.Client.Mvc; // Greeter.GreeterClient
```

10. In `Program.cs`, in the section of configuring services, add a statement to add the `GreeterClient` as a named gRPC client that will be communicating with a service that is listening on port 5121, as shown in the following code:

```
builder.Services.AddGrpcClient<Greeter.GreeterClient>("Greeter",
    options =>
    {
        options.Address = new Uri("https://localhost:5121");
    });
```

11. In the `Controllers` folder, in `HomeController.cs`, import the namespaces to work with gRPC channels and the gRPC client factory, as shown in the following code:

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

```
using Grpc.Net.ClientFactory; // GrpcClientFactory
```

12. In the controller class, declare a field to store a greeter client instance and set it by using the client factory in the constructor, as shown highlighted in the following code:

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    protected readonly Greeter.GreeterClient greeterClient;

    public HomeController(ILogger<HomeController> logger,
        GrpcClientFactory factory)
    {
        _logger = logger;
        greeterClient = factory.CreateClient<Greeter.
GreeterClient>("Greeter");
    }
}
```

13. In the Index action method, make the method asynchronous, add a string parameter named name with a default value of Henrietta, and then add statements to use the gRPC client to call the SayHelloAsync method, passing a HelloRequest object and storing the HelloReply response in ViewData, while catching any exceptions, as shown highlighted in the following code:

```
public async Task<IActionResult> Index(string name = "Henrietta")
{
    try
    {
        HelloReply reply = await greeterClient.SayHelloAsync(
            new HelloRequest { Name = name });

        ViewData["greeting"] = "Greeting from gRPC service: " + reply.
Message;
    }
    catch (Exception ex)
    {
        _logger.LogWarning($"Northwind.Grpc.Service is not responding.");
        ViewData["exception"] = ex.Message;
    }

    return View();
}
```

14. In Views/Home, in Index.cshtml, after the Welcome heading, remove the existing <p> element and then add markup to render a form for the visitor to enter their name, and then if they submit and the gRPC service responds, to output the greeting, as shown in the following markup:

```
<div class="alert alert-secondary">
    <form>
```

```

        <input name="name" placeholder="Enter your name" />
        <input type="submit" />
    </form>
</div>
@if (ViewData["greeting"] is not null)
{
    <p class="alert alert-primary">@ViewData["greeting"]</p>
}
@if (ViewData["exception"] is not null)
{
    <p class="alert alert-danger">@ViewData["exception"]</p>
}

```



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 service and client

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

1. Start the `Northwind.Grpc.Service` project without debugging.
2. Start the `Northwind.Grpc.Client.Mvc` project.
3. If necessary, start a browser and navigate to the home page: `https://localhost:5122/`.
4. Note the greeting on the home page, as shown in *Figure 12.4*:

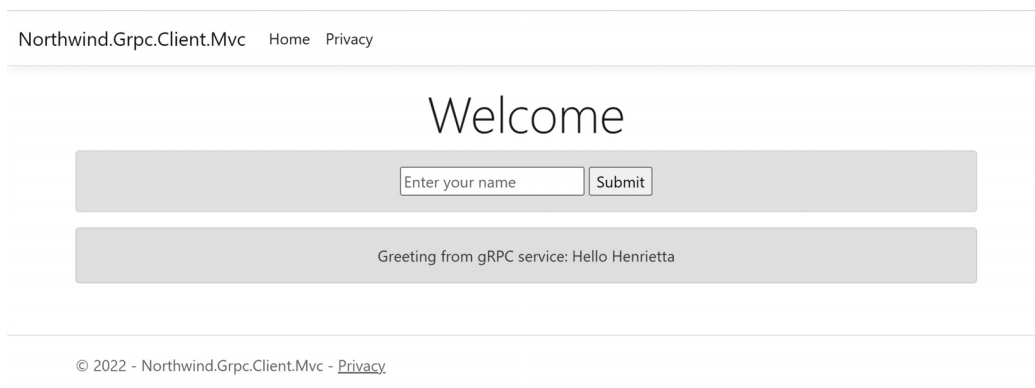


Figure 12.4: Home page after calling the gRPC service to get a greeting

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

```
info: System.Net.Http.HttpClient.Greeter.LogicalHandler[100]
```

```

    Start processing HTTP request POST https://localhost:5121/greet.
Greeter/SayHello
info: System.Net.Http.HttpClient.Greeter.ClientHandler[100]
    Sending HTTP request POST https://localhost:5121/greet.Greeter/
SayHello
info: System.Net.Http.HttpClient.Greeter.ClientHandler[101]
    Received HTTP response headers after 60.5352ms - 200
info: System.Net.Http.HttpClient.Greeter.LogicalHandler[101]
    End processing HTTP request after 69.1623ms - 200

```

6. Enter and submit another name in the page.
7. Close the browser and shut down the web servers.

Implementing gRPC for an EF Core model

Now we will add a service for working with the Northwind database to the gRPC project.

Implementing the gRPC service

We will reference the EF Core model that you created in *Chapter 2, Managing Relational Data Using SQL Server*, then define a contract for the gRPC service using a .proto file, and finally implement the service:

1. In the Northwind.Grpc.Service project, add a project reference to the Northwind database context project, as shown in the following markup:

```

<ItemGroup>
  <ProjectReference Include="..\..\Chapter02\Northwind.Common.DataContext
.SqlServer\Northwind.Common.DataContext.SqlServer.csproj" />
</ItemGroup>

```



The Include path must not have a line break.

2. At the command line or terminal, build the Northwind.Grpc.Service project.
3. In the Northwind.Grpc.Service project, in the Protos folder, add a new file (the item template is named **Protocol Buffer File** in Visual Studio 2022) named shipper.proto, as shown in the following code:

```

syntax = "proto3";

option csharp_namespace = "Northwind.Grpc.Service";

package shipper;

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

```

```

}

message ShipperRequest {
    int32 shipperId = 1;
}

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

```

4. 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>

```

5. Build the `Northwind.Grpc.Service` project.
6. 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
using ShipperEntity = Packt.Shared.Shipper;

namespace Northwind.Grpc.Service.Services;

public class ShipperService : Shipper.ShipperBase
{
    protected readonly ILogger<ShipperService> _logger;
    protected 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)
    {
        ShipperEntity? shipper = await db.Shippers.FindAsync(request.
            ShipperId);
    }
}

```

```

        if (shipper == null)
        {
            return null;
        }
        else
        {
            return ToShipperReply(shipper);
        }
    }

    private ShipperReply ToShipperReply(ShipperEntity shipper)
    {
        return new ShipperReply
        {
            ShipperId = shipper.ShipperId,
            CompanyName = shipper.CompanyName,
            Phone = shipper.Phone
        };
    }
}

```



The .proto file generates classes that represent the messages sent to and from a gRPC service. We therefore cannot use the entity classes defined for the EF Core model. We need a helper method like `ToShipperReply` that can map an instance of an entity class to an instance of the .proto-generated classes like `ShipperReply`. This could be a good use for `AutoMapper`, although in this case the mapping is simple enough to hand-code.

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

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

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

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

9. In the section that configures the HTTP pipeline, after the call to register `GreeterService`, add a statement to register `ShipperService`, as shown in the following code:

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

Implementing the gRPC client

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.Service` project to the Protos folder in the `Northwind.Grpc.Client.Mvc` project.
2. In the `Northwind.Grpc.Client.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.Grpc.Client.Mvc";
```

3. In the `Northwind.Grpc.Client.Mvc` project file, modify or add the 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 `Northwind.Grpc.Client.Mvc` project file, in `Program.cs`, add a statement to register the `ShipperClient` class to connect to the gRPC service listening on port 5121, as shown in the following code:

```
builder.Services.AddGrpcClient<Shipper.ShipperClient>("Shipper",
    options =>
    {
        options.Address = new Uri("https://localhost:5121");
    });
```

5. In the `Controllers` folder, in `HomeController.cs`, declare a field to store a shipper client instance and set it by using the client factory in the constructor, as shown highlighted in the following code:

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    protected readonly Greeter.GreeterClient greeterClient;
    protected readonly Shipper.ShipperClient shipperClient;

    public HomeController(ILogger<HomeController> logger,
        GrpcClientFactory factory)
    {
        _logger = logger;
        greeterClient = factory.CreateClient<Greeter.GreeterClient>("Greeter");
        shipperClient = factory.CreateClient<Shipper.ShipperClient>("Shipper");
    }
}
```

6. In `HomeController.cs`, in the `Index` action method, add a parameter named `id` and statements to call the `Shipper gRPC` service to get a shipper with the matching `ShipperId`, as shown highlighted in the following code:

```
public async Task<IActionResult> Index(
    string name = "Henrietta", int id = 1)
{
    try
    {
        HelloReply reply = await greeterClient.SayHelloAsync(
            new HelloRequest { Name = name });

        ViewData["greeting"] = "Greeting from gRPC service: " + reply.
        Message;

        ShipperReply shipperReply = await shipperClient.GetShipperAsync(
            new ShipperRequest { ShipperId = id });

        ViewData["shipper"] = "Shipper from gRPC service: " +
            $"ID: {shipperReply.ShipperId}, Name: {shipperReply.CompanyName}, "
            + $" Phone: {shipperReply.Phone}.";
    }
    catch (Exception ex)
    {
        _logger.LogWarning($"Northwind.Grpc.Service is not responding.");
        ViewData["exception"] = ex.Message;
    }

    return View();
}
```

7. In `Views/Home`, in `Index.cshtml`, add code to render a form for the visitor to enter a shipper ID, and render the shipper details after the greeting, as shown highlighted in the following markup:

```
@{
    ViewData["Title"] = "Home Page";
}
<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <div class="alert alert-secondary">
        <form>
            <input name="name" placeholder="Enter your name" />
            <input type="submit" />
        </form>
        <form>
            <input name="id" placeholder="Enter a shipper id" />
            <input type="submit" />
        </form>
    </div>
</div>
```



```

</form>
</div>
@if (ViewData["greeting"] is not null)
{
    <p class="alert alert-primary">@ViewData["greeting"]</p>
}
@if (ViewData["exception"] is not null)
{
    <p class="alert alert-danger">@ViewData["exception"]</p>
}
@if (ViewData["shipper"] is not null)
{
    <p class="alert alert-primary">@ViewData["shipper"]</p>
}
</div>

```

8. Start the Northwind.Grpc.Service project without debugging.
9. Start the Northwind.Grpc.Client.Mvc project.
10. If necessary, start a browser and navigate to the home page: <https://localhost:5122/>.
11. Note the shipper information on the services page, as shown in *Figure 12.5*:

Northwind.Grpc.Client.Mvc Home Privacy

Welcome

Greeting from gRPC service: Hello Henrietta

Shipper from gRPC service: ID: 1, Name: Speedy Express, Phone: (503) 555-9831.

© 2022 - Northwind.Grpc.Client.Mvc - [Privacy](#)

Figure 12.5: Home page after calling the gRPC service to get a shipper

12. There are three shippers in the Northwind database with IDs of 1, 2, and 3. Try entering their IDs to ensure they can all be retrieved, and try entering an ID that does not exist, like 4.
13. Close the browser and shut down the web servers.

Getting request and response metadata

Formally defined request and response messages as part of a contract are not the only mechanism to pass data between client and service. You can also use metadata sent as headers and trailers. Both are simple dictionaries that are passed along with the messages.

Let's see how you can get metadata about a gRPC call:

1. In `HomeController.cs`, import the namespace to use the `AsyncUnaryCall<T>` class, as shown in the following code:

```
using Grpc.Core; // AsyncUnaryCall<T>
```

2. In the `Index` method, comment out the statement that makes the call to the gRPC shipper service. Add statements that get the underlying `AsyncUnaryCall<T>` object, then use it to get the headers, output them to the log, and then get the response, as shown highlighted in the following code:

```
// ShipperReply shipperReply = await shipperClient.GetShipperAsync(
//     new ShipperRequest { ShipperId = id });

// the same call as above but not awaited
AsyncUnaryCall<ShipperReply> shipperCall = shipperClient.GetShipperAsync(
    new ShipperRequest { ShipperId = id });

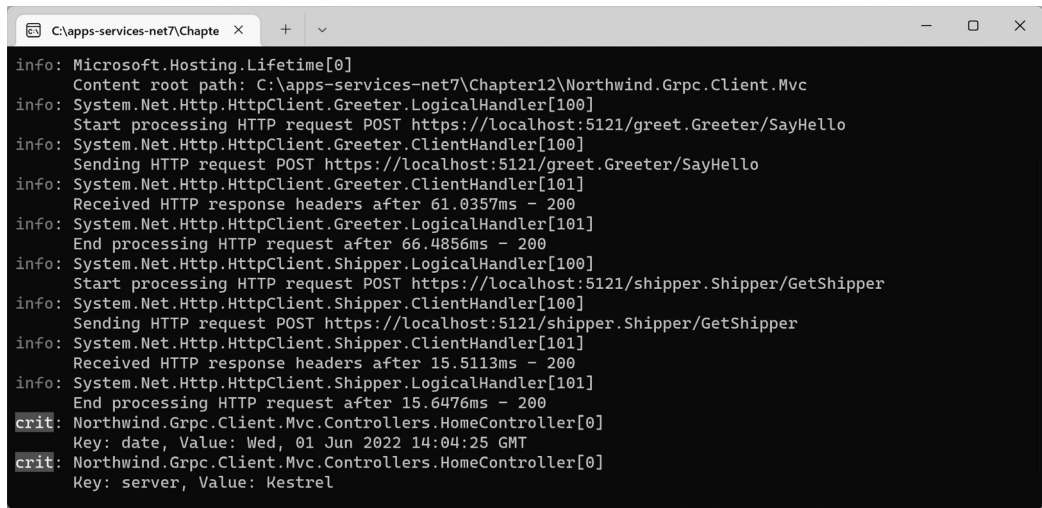
Metadata metadata = await shipperCall.ResponseHeadersAsync;

foreach (Metadata.Entry entry in metadata)
{
    // not really critical, just doing this to make it easier to see
    _logger.LogCritical($"Key: {entry.Key}, Value: {entry.Value}");
}

ShipperReply shipperReply = await shipperCall.ResponseAsync;

ViewData["shipper"] = "Shipper from gRPC service: " +
    $"ID: {shipperReply.ShipperId}, Name: {shipperReply.CompanyName},"
    + $" Phone: {shipperReply.Phone}.";
```

3. Start the `Northwind.Grpc.Service` project without debugging.
4. Start the `Northwind.Grpc.Client.Mvc` project.
5. If necessary, start a browser and navigate to the home page: <https://localhost:5122/>.
6. Note the client successfully POSTing to the gRPC Greeter and Shipper services and the red critical messages outputting the two entries in the gRPC metadata for the call to `GetShipper`, with keys of `date` and `server`, as shown in *Figure 12.6*:



```

C:\apps-services-net7\Chapte x + -
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\apps-services-net7\Chapter12\Northwind.Grpc.Client.Mvc
info: System.Net.Http.HttpClient.Greeter.LogicalHandler[100]
      Start processing HTTP request POST https://localhost:5121/greet.Greeter/SayHello
info: System.Net.Http.HttpClient.Greeter.ClientHandler[100]
      Sending HTTP request POST https://localhost:5121/greet.Greeter/SayHello
info: System.Net.Http.HttpClient.Greeter.ClientHandler[101]
      Received HTTP response headers after 61.0357ms - 200
info: System.Net.Http.HttpClient.Greeter.LogicalHandler[101]
      End processing HTTP request after 66.4856ms - 200
info: System.Net.Http.HttpClient.Shipper.LogicalHandler[100]
      Start processing HTTP request POST https://localhost:5121/shipper.Shipper/GetShipper
info: System.Net.Http.HttpClient.Shipper.ClientHandler[100]
      Sending HTTP request POST https://localhost:5121/shipper.Shipper/GetShipper
info: System.Net.Http.HttpClient.Shipper.ClientHandler[101]
      Received HTTP response headers after 15.5113ms - 200
info: System.Net.Http.HttpClient.Shipper.LogicalHandler[101]
      End processing HTTP request after 15.6476ms - 200
crit: Northwind.Grpc.Client.Mvc.Controllers.HomeController[0]
      Key: date, Value: Wed, 01 Jun 2022 14:04:25 GMT
crit: Northwind.Grpc.Client.Mvc.Controllers.HomeController[0]
      Key: server, Value: Kestrel
  
```

Figure 12.6: Logging metadata from a gRPC call

7. Close the browser and shut down the web servers.



The trailers equivalent of the `ResponseHeadersAsync` property is the `GetTrailers` method. It has a return value of `Metadata` that contains the dictionary of trailers. Trailers are accessible at the end of a call.

Adding a deadline for higher reliability

Setting a deadline for a gRPC call is recommended practice because it controls the upper limit on how long a gRPC call can run for. It prevents gRPC services from potentially consuming too many server resources.

The deadline information is sent to the service, so the service has an opportunity to give up its work once the deadline has passed instead of continuing forever. Even if the server completes its work within the deadline, the client may give up before the response arrives at the client because the deadline has passed due to the overhead of communication.

Let's see an example:

1. In the `Northwind.Grpc.Service` project, in the `Services` folder, in `ShipperService.cs`, in the `GetShipper` method, add a statement to pause for 5 seconds, as shown highlighted in the following code:

```

public override async Task<ShipperReply?> GetShipper(
    ShipperRequest request, ServerCallContext context)
{
    _logger.LogCritical(
        "This request has a deadline of {0:T}. It is now {1:T}.",
  
```

```

        context.Deadline, DateTime.UtcNow);

        await Task.Delay(TimeSpan.FromSeconds(5));

        return ToShipperReply(
            await db.Shippers.FindAsync(request.ShipperId));
    }

```

2. In `HomeController.cs`, in the `Index` method, set a deadline of 3 seconds when calling the `GetShipperAsync` method, as shown highlighted in the following code:

```

AsyncUnaryCall<ShipperReply> shipperCall = shipperClient.GetShipperAsync(
    new ShipperRequest { ShipperId = id },
    deadline: DateTime.UtcNow.AddSeconds(3)); // must be a UTC DateTime

```

3. In `HomeController.cs`, in the `Index` method, before the existing catch block, add a catch block for an `RpcException` when the exception's status code matches the code for deadline exceeded, as shown highlighted in the following code:

```

catch (RpcException rpcex) when (rpcex.StatusCode ==
    global::Grpc.Core.StatusCode.DeadlineExceeded)
{
    _logger.LogWarning("Northwind.Grpc.Service deadline exceeded.");
    ViewData["exception"] = rpcex.Message;
}
catch (Exception ex)
{
    _logger.LogWarning($"Northwind.Grpc.Service is not responding.");
    ViewData["exception"] = ex.Message;
}

```

4. In the `Northwind.Grpc.Service` project, in `appsettings.Development.json`, modify the logging level for ASP.NET Core from the default of `Warning` to `Information`, as shown highlighted in the following configuration:

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Information"
    }
  }
}

```

5. In the `Northwind.Grpc.Client.Mvc` project, in `appsettings.Development.json`, modify the logging level for ASP.NET Core from the default of `Warning` to `Information`, as shown highlighted in the following configuration:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Information"
    }
  }
}
```

6. Start the `Northwind.Grpc.Service` project without debugging.
7. Start the `Northwind.Grpc.Client.Mvc` project.
8. If necessary, start a browser and navigate to the home page: `https://localhost:5122/`.
9. At the command prompt or terminal for the gRPC service, note the request has a three second deadline, as shown in the following output:

```
crit: Northwind.Grpc.Service.Services.ShipperService[0]
      This request has a deadline of 14:56:30. It is now 14:56:27.
```

10. In the browser, note that after three seconds the home page shows a deadline exceeded exception, as shown in *Figure 12.7*:

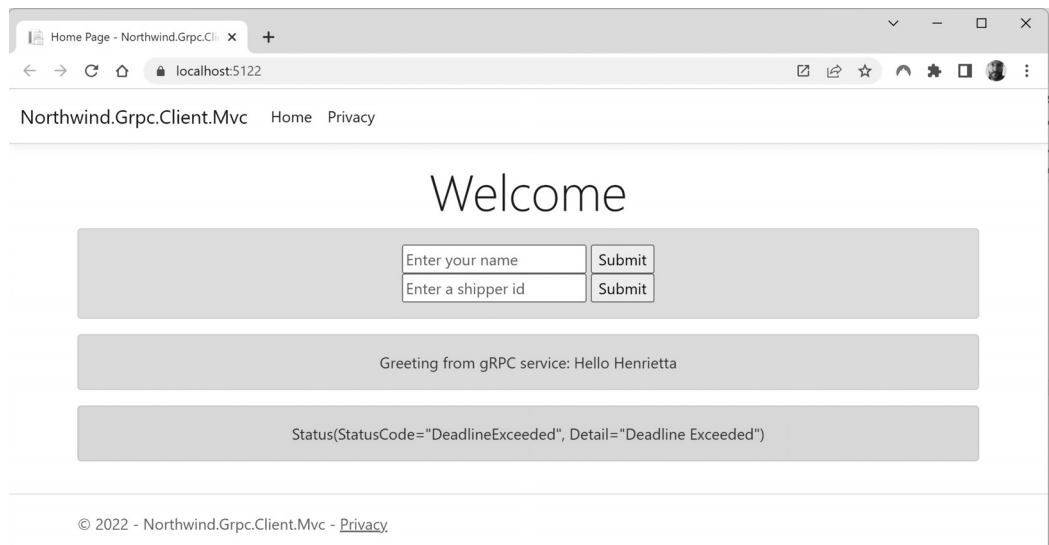


Figure 12.7: A deadline has passed

11. At the command prompt or terminal for the ASP.NET Core MVC client, note the logs that start at the point where a request is made to the `GetShipper` method on the gRPC service, but the deadline is exceeded, as shown in the following output:

```
info: System.Net.Http.HttpClient.Shipper.LogicalHandler[100]
      Start processing HTTP request POST https://localhost:5121/shipper.
      Shipper/GetShipper
```

```

info: System.Net.Http.HttpClient.Shipper.ClientHandler[100]
      Sending HTTP request POST https://localhost:5121/shipper.Shipper/
GetShipper
warn: Grpc.Net.Client.Internal.GrpcCall[7]
      gRPC call deadline exceeded.
info: System.Net.Http.HttpClient.Shipper.ClientHandler[101]
      Received HTTP response headers after 3031.4299ms - 200
info: System.Net.Http.HttpClient.Shipper.LogicalHandler[101]
      End processing HTTP request after 3031.5469ms - 200
crit: Northwind.Grpc.Client.Mvc.Controllers.HomeController[0]
      Key: date, Value: Thu, 18 Aug 2022 15:14:17 GMT
crit: Northwind.Grpc.Client.Mvc.Controllers.HomeController[0]
      Key: server, Value: Kestrel
info: Grpc.Net.Client.Internal.GrpcCall[3]
      Call failed with gRPC error status. Status code:
'DeadlineExceeded', Message: 'Deadline Exceeded'.
warn: Northwind.Grpc.Client.Mvc.Controllers.HomeController[0]
      Northwind.Grpc.Service deadline exceeded.

```

12. Close the browser and shut down the web servers.



Good Practice: The default is no deadline. Always set a deadline in the client call. In your service implementation, get the deadline and use it to automatically abandon the work if it is exceeded. Pass the cancellation token to any asynchronous calls so that work completes quickly on the server and frees up resources.

Implementing gRPC JSON transcoding

JSON is the most popular format for services that return data to a browser or mobile device. It would be great if we could create a gRPC service and magically make it callable via non-HTTP/2 using JSON. Thankfully, there is a solution.

Microsoft has a new technology they have named **gRPC JSON transcoding**, which is an ASP.NET Core extension that creates HTTP endpoints with JSON for gRPC services, based on Google's `HttpRule` class for their gRPC Transcoding. You can read about that at the following link: <https://cloud.google.com/dotnet/docs/reference/Google.Api.CommonProtos/latest/Google.Api.HttpRule>.

Enabling gRPC JSON transcoding

Let's see how to enable gRPC JSON transcoding in our gRPC service:

1. In the `Northwind.Grpc.Service` project, add a package reference for gRPC JSON transcoding, as shown highlighted in the following markup:

```

<ItemGroup>
  <PackageReference Include="Grpc.AspNetCore" Version="2.48.0" />
  <PackageReference Include="Microsoft.AspNetCore.Grpc.JsonTranscoding"

```

```
Version="7.0.0" />
</ItemGroup>
```

2. In `Program.cs`, add a call to add JSON transcoding after the call to add gRPC, as shown highlighted in the following code:

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

3. In the `Northwind.Grpc.Service` project/folder, add a folder named `google`.
4. In the `google` folder, add a folder named `api`.
5. In the `api` folder, add two `.proto` files named `http.proto` and `annotations.proto`.
6. Copy and paste the raw contents for the two files from the files found at the following link: <https://github.com/dotnet/aspnetcore/tree/main/src/Grpc/JsonTranscoding/test/testassets/Sandbox/google/api>.
7. In the `Protos` folder, in `greet.proto`, import the `annotations.proto` file and use it to add an option to expose an endpoint to make an HTTP request to the `SayHello` method, as shown highlighted in the following code:

```
syntax = "proto3";

import "google/api/annotations.proto";

option csharp_namespace = "Northwind.Grpc.Service";

package greet;

// The greeting service definition.
service Greeter {
    // Sends a greeting
    rpc SayHello (HelloRequest) returns (HelloReply) {
        option (google.api.http) = {
            get: "/v1/greeter/{name}"
        };
    }
}

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

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

8. In the Protos folder, in `shipper.proto`, import the `annotations.proto` file and use it to add an option to expose an endpoint to make an HTTP request to the `GetShipper` method, as shown in the following code:

```
syntax = "proto3";

import "google/api/annotations.proto";

option csharp_namespace = "Northwind.Grpc.Service";

package shipper;

service Shipper {
  rpc GetShipper (ShipperRequest) returns (ShipperReply) {
    option (google.api.http) = {
      get: "/v1/shipper/{shipperId}"
    };
  }
}

message ShipperRequest {
  int32 shipperId = 1;
}

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

Testing gRPC JSON transcoding

Now we can start the `gRPC` service and call it directly from any browser:

1. Start the `Northwind.Grpc.Service` project.
2. Start any browser, show the developer tools, and click the **Network** tab to start recording network traffic.
3. Navigate to a URL to make a GET request that will call the `SayHello` method: `https://localhost:5121/v1/greeter/Bob`, and note the JSON response returned by the `gRPC` service, as shown in *Figure 12.8*:

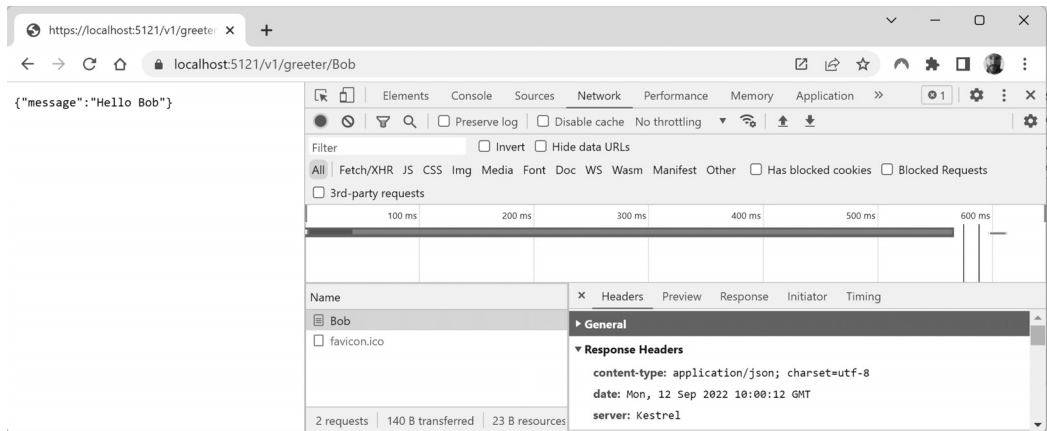


Figure 12.8: Making an HTTP 1.1 GET request to a gRPC service and receiving a response in JSON

4. Navigate to a URL to make a GET request to call the `GetShipper` method: `https://localhost:5121/v1/shipper/2`, and note the JSON response returned by the gRPC service, as shown in Figure 12.9:

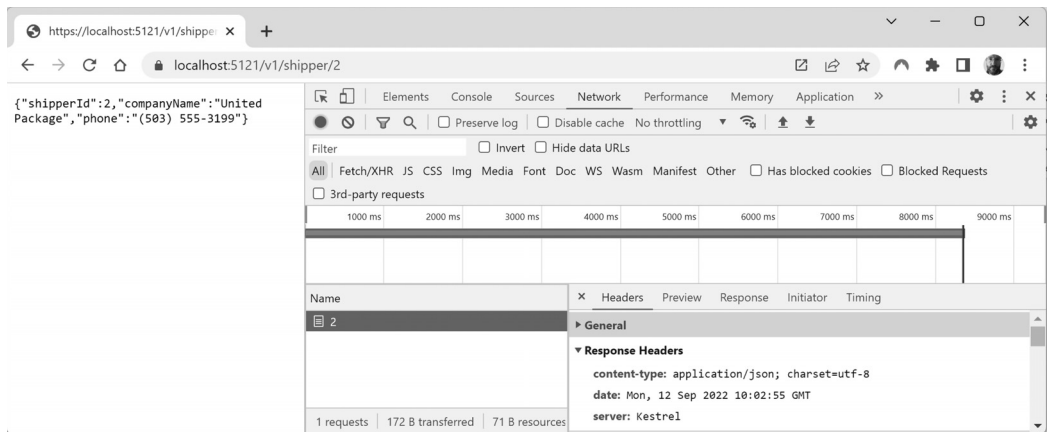


Figure 12.9: Making an HTTP 1.1 GET request to a gRPC service and receiving a response in JSON

5. Close the browser and shut down the web server.

Comparing with gRPC-Web

gRPC-Web is an alternative to gRPC JSON transcoding to allow gRPC services to be called from a browser. gRPC-Web achieves this by executing a gRPC-Web client inside the browser. This has the advantage that the communications between browser and gRPC service use Protobuf and therefore get all the performance and scalability benefits of true gRPC communication.

As you have seen, gRPC JSON transcoding allows browsers to call gRPC services as if they were HTTP APIs with JSON. The browser needs to know nothing about gRPC. The gRPC service is responsible for converting those HTTP API calls into calls to the actual gRPC service implementation.

To simplify and summarize:

- gRPC JSON transcoding happens on the server side.
- gRPC-Web happens on the client side.



Good Practice: Add gRPC JSON transcoding support to all your gRPC services hosted in ASP.NET Core. This provides the best of both worlds. Clients that cannot use gRPC natively can call the Web API. Clients that can use gRPC natively can call it directly.

Practicing and exploring

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

Exercise 12.1 – Test your knowledge

Answer the following questions:

1. What are three benefits of gRPC that make it a good choice for implementing services?
2. How are contracts defined in gRPC?
3. Which of the following .NET types require extensions to be imported: `int`, `double`, and `DateTime`?
4. Why should you set a deadline when calling a gRPC method?
5. What are the benefits of enabling gRPC JSON transcoding to a gRPC service hosted in ASP.NET Core?

Exercise 12.2 – Explore topics

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

<https://github.com/markjprice/apps-services-net7/blob/main/book-links.md#chapter-12--building-efficient-microservices-using-grpc>

Summary

In this chapter, you:

- Learned some concepts around gRPC services, how they work, and their benefits.
- Implemented a simple gRPC service.
- Implemented a gRPC service that uses an EF Core model.
- Learned how to set deadlines and read metadata sent as headers and trailers.

- Extended a gRPC service with support for being called as an HTTP service with JSON, to support clients that cannot work with gRPC natively.

In the next chapter, you will learn about SignalR, a technology for performing real-time communication between client and server.

7

Handling Dates, Times, and Internationalization

This chapter is about some common types that are included with .NET. These include types for manipulating dates and times and implementing internationalization, which includes globalization and localization.

When writing code to handle times, it is especially important to consider time zones. Bugs are often introduced because two times are compared in different time zones without taking that into account. It is important to understand the concept of **Coordinated Universal Time (UTC)** and to convert time values into UTC before performing time manipulation. You should also be aware of any **Daylight Saving Time (DST)** adjustments that might be needed.

This chapter covers the following topics:

- Working with dates and times
- Working with time zones
- Working with cultures

Working with dates and times

After numbers and text, the next most popular types of data to work with are dates and times. The two main types are as follows:

- **DateTime**: Represents a combined date and time value for a fixed point in time.
- **TimeSpan**: Represents a duration of time.

These two types are often used together. For example, if you subtract one **DateTime** value from another, the result is a **TimeSpan**. If you add a **TimeSpan** to a **DateTime**, then the result is a **DateTime** value.

Specifying date and time values

A common way to create a date and time value is to specify individual values for the date and time components like day and hour, as described in the following table:

Date/time parameter	Value range
year	1 to 9,999
month	1 to 12
day	1 to the number of days in that month
hour	0 to 23
minute	0 to 59
second	0 to 59
millisecond	0 to 999
microsecond	0 to 999

An alternative is to provide the value as a string to be parsed, but this can be misinterpreted depending on the default culture of the thread. For example, in the UK, dates are specified as day/month/year, compared to the US, where dates are specified as month/day/year.

Let's see what you might want to do with dates and times:

1. Use your preferred code editor to create a new project, as defined in the following list:
 - Project template: **Console App/console**
 - Project file and folder: `WorkingWithTime`
 - Workspace/solution file and folder: `Chapter07`

In Visual Studio Code, select `WorkingWithTime` as the active OmniSharp project.

2. In the project file, add an element to statically and globally import the `System.Console` class.
3. Add a new class file named `Program.Helpers.cs` and modify its contents, as shown in the following code:

```
partial class Program
{
    static void SectionTitle(string title)
    {
        ConsoleColor previousColor = ForegroundColor;
        ForegroundColor = ConsoleColor.DarkYellow;
        WriteLine("*");
        WriteLine($"* {title}");
        WriteLine("*");
        ForegroundColor = previousColor;
    }
}
```

4. In `Program.cs`, delete the existing statements and then add statements to initialize some special date/time values, as shown in the following code:

```
SectionTitle("Specifying date and time values");

WriteLine($"DateTime.MinValue: {DateTime.MinValue}");
WriteLine($"DateTime.MaxValue: {DateTime.MaxValue}");
WriteLine($"DateTime.UnixEpoch: {DateTime.UnixEpoch}");
WriteLine($"DateTime.Now: {DateTime.Now}");
WriteLine($"DateTime.Today: {DateTime.Today}");
```

5. Run the code and note the results, as shown in the following output:

```
DateTime.MinValue: 01/01/0001 00:00:00
DateTime.MaxValue: 31/12/9999 23:59:59
DateTime.UnixEpoch: 01/01/1970 00:00:00
DateTime.Now: 23/03/2022 13:50:30
DateTime.Today: 23/03/2022 00:00:00
```



The date and time formats output are determined by the culture settings of your console app; for example, mine uses *English (Great Britain)* culture. Optionally, to see the same output as mine, add a statement to the top of your `Program.cs`, as shown in the following code:

```
Thread.CurrentThread.CurrentCulture =
    System.Globalization.CultureInfo.GetCultureInfo("en-GB");
```

Formatting date and time values

You have just seen that dates and times have default formats based on the current culture. You can take control of date and time formatting using custom format codes:

1. Add statements to define Christmas Day in 2024 and display it in various ways, as shown in the following code:

```
DateTime xmas = new(year: 2024, month: 12, day: 25);
WriteLine($"Christmas (default format): {xmas}");
WriteLine($"Christmas (custom format): {xmas:dddd, dd MMMM yyyy}");
WriteLine($"Christmas is in month {xmas.Month} of the year.");
WriteLine($"Christmas is day {xmas.DayOfYear} of the year 2024.");
WriteLine($"Christmas {xmas.Year} is on a {xmas.DayOfWeek}.");
```

2. Run the code and note the results, as shown in the following output:

```
Christmas (default format): 25/12/2024 00:00:00
Christmas (custom format): Wednesday, 25 December 2024
Christmas is in month 12 of the year.
Christmas is day 360 of the year 2024.
Christmas 2024 is on a Wednesday.
```

Date and time calculations

Now, let's try performing simple calculations on date and time values:

1. Add statements to perform addition and subtraction with Christmas 2024, as shown in the following code:

```
SectionTitle("Date and time calculations");

DateTime beforeXmas = xmas.Subtract(TimeSpan.FromDays(12));
DateTime afterXmas = xmas.AddDays(12);

// :d means format as short date only without time
WriteLine($"12 days before Christmas: {beforeXmas:d}");
WriteLine($"12 days after Christmas: {afterXmas:d}");

TimeSpan untilXmas = xmas - DateTime.Now;

WriteLine($"Now: {DateTime.Now}");
WriteLine("There are {0} days and {1} hours until Christmas 2024.",
    arg0: untilXmas.Days, arg1: untilXmas.Hours);

WriteLine("There are {0:N0} hours until Christmas 2024.",
    arg0: untilXmas.TotalHours);
```

2. Run the code and note the results, as shown in the following output:

```
12 days before Christmas: 13/12/2024
12 days after Christmas: 06/01/2025
Now: 23/03/2022 16:16:02
There are 1007 days and 7 hours until Christmas 2024.
There are 24,176 hours until Christmas 2024.
```

3. Add statements to define the time on Christmas Day that your children (or dog? Or cat? Or iguana?) might wake up to open presents, and display it in various ways, as shown in the following code:

```
DateTime kidsWakeUp = new(
    year: 2024, month: 12, day: 25,
    hour: 6, minute: 30, second: 0);

WriteLine($"Kids wake up: {kidsWakeUp}");

WriteLine("The kids woke me up at {0}",
    arg0: kidsWakeUp.ToShortTimeString());
```

4. Run the code and note the results, as shown in the following output:

```
Kids wake up: 25/12/2024 06:30:00
```

```
The kids woke me up at 06:30
```

Microseconds and nanoseconds

In earlier versions of .NET, the smallest unit of time measurement was a tick. A tick is 100 nanoseconds, so developers used to have to do the calculations themselves. .NET 7 introduces milli- and microsecond parameters to constructors, and micro- and nanosecond properties to the `DateTime`, `DateTimeOffset`, `TimeSpan`, and `TimeOnly` types.

Let's see some examples:

1. Add statements to construct a date and time value with more precision than was possible and to display its value, as shown in the following code:

```
SectionTitle("Milli-, micro-, and nanoseconds");

DateTime preciseTime = new(
    year: 2022, month: 11, day: 8,
    hour: 12, minute: 0, second: 0,
    millisecond: 6, microsecond: 999);

Writeline("Millisecond: {0}, Microsecond: {1}, Nanosecond: {2}",
    preciseTime.Millisecond, preciseTime.Microsecond, preciseTime.Nanosecond);

preciseTime = DateTime.UtcNow;

// Nanosecond value will be 0 to 900 in 100 nanosecond increments.
Writeline("Millisecond: {0}, Microsecond: {1}, Nanosecond: {2}",
    preciseTime.Millisecond, preciseTime.Microsecond, preciseTime.Nanosecond);
```

2. Run the code and note the results, as shown in the following output:

```
*
* Milli-, micro-, and nanoseconds
*
Millisecond: 6, Microsecond: 999, Nanosecond: 0
Millisecond: 243, Microsecond: 958, Nanosecond: 400
```

Globalization with dates and times

The current culture controls how dates and times are formatted and parsed:

1. At the top of `Program.cs`, import the namespace for working with globalization, as shown in the following code:

```
using System.Globalization; // CultureInfo
```


2. Add statements to show the current culture that is used to display date and time values, and then parse the United States' Independence Day and display it in various ways, as shown in the following code:

```

SectionTitle("Globalization with dates and times");

// same as Thread.CurrentThread.CurrentCulture
WriteLine($"Current culture is: {CultureInfo.CurrentCulture.Name}");

string textDate = "4 July 2024";
DateTime independenceDay = DateTime.Parse(textDate);

WriteLine($"Text: {textDate}, DateTime: {independenceDay:d MMMM}");

textDate = "7/4/2024";
independenceDay = DateTime.Parse(textDate);

WriteLine($"Text: {textDate}, DateTime: {independenceDay:d MMMM}");

independenceDay = DateTime.Parse(textDate,
    provider: CultureInfo.GetCultureInfo("en-US"));

WriteLine($"Text: {textDate}, DateTime: {independenceDay:d MMMM}");

```



Good Practice: Although you can create a `CultureInfo` instance using its constructor, unless you need to make changes to it, you should get a read-only shared instance by calling the `GetCultureInfo` method.

3. Run the code and note the results, as shown in the following output:

```

Current culture is: en-GB
Text: 4 July 2024, DateTime: 4 July
Text: 7/4/2024, DateTime: 7 April
Text: 7/4/2024, DateTime: 4 July

```



On my computer, the current culture is *English (Great Britain)*. If a date is given as 4 July 2021, then it is correctly parsed regardless of whether the current culture is British or American. But if the date is given as 7/4/2024, then it is wrongly parsed as 7 April. You can override the current culture by specifying the correct culture as a provider when parsing, as shown in the third example above.

4. Add statements to loop from the year 2022 to 2028, displaying if the year is a leap year and how many days there are in February, and then show if Christmas and Independence Day are during DST, as shown in the following code:

```

for (int year = 2022; year <= 2028; year++)
{
    Write($"{year} is a leap year: {DateTime.IsLeapYear(year)}. ");
    WriteLine("There are {0} days in February {1}.",
        arg0: DateTime.DaysInMonth(year: year, month: 2), arg1: year);
}

Writeline("Is Christmas daylight saving time? {0}",
    arg0: xmas.IsDaylightSavingTime());

Writeline("Is July 4th daylight saving time? {0}",
    arg0: independenceDay.IsDaylightSavingTime());

```

5. Run the code and note the results, as shown in the following output:

```

2022 is a leap year: False. There are 28 days in February 2022.
2023 is a leap year: False. There are 28 days in February 2023.
2024 is a leap year: True. There are 29 days in February 2024.
2025 is a leap year: False. There are 28 days in February 2025.
2026 is a leap year: False. There are 28 days in February 2026.
2027 is a leap year: False. There are 28 days in February 2027.
2028 is a leap year: True. There are 29 days in February 2028.
Is Christmas daylight saving time? False
Is July 4th daylight saving time? True

```



DST is not used in all countries; it is also determined by hemisphere, and politics plays a role. For example, the United States is currently debating if they should make DST permanent. They might decide to leave the decision up to individual states. It could all get extra confusing for Americans over the next few years.

Localizing the DayOfWeek enum

DayOfWeek is an enum so it cannot be localized as you might expect. Its string values are hardcoded in English, as shown in the following code:

```

namespace System
{
    public enum DayOfWeek
    {
        Sunday = 0,
        Monday = 1,
        Tuesday = 2,
        Wednesday = 3,
        Thursday = 4,
        Friday = 5,
        Saturday = 6
    }
}

```

There are two solutions to this problem. First, you could apply the `dddd` date format code to a whole date value. For example:

```
WriteLine($"The day of the week is {0:dddd}.", DateTime.Now);
```

Second, you can use a helper method of the `DateTimeFormatInfo` class to convert a `DayOfWeek` value into a localized string for output as text.

Let's see an example of the problem and solution:

1. Add statements to explicitly set the current culture to Danish and then output the current day of the week in that culture, as shown in the following code:

```
SectionTitle("Localizing the DayOfWeek enum");

CultureInfo previousCulture = Thread.CurrentThread.CurrentCulture;

// explicitly set culture to Danish (Denmark)
Thread.CurrentThread.CurrentCulture =
    CultureInfo.GetCultureInfo("da-DK");

WriteLine("Culture: {0}, DayOfWeek: {1}",
    Thread.CurrentThread.CurrentCulture.NativeName,
    DateTime.Now.DayOfWeek);

WriteLine("Culture: {0}, DayOfWeek: {1:dddd}",
    Thread.CurrentThread.CurrentCulture.NativeName,
    DateTime.Now);

WriteLine("Culture: {0}, DayOfWeek: {1}",
    Thread.CurrentThread.CurrentCulture.NativeName,
    DateTimeFormatInfo.CurrentInfo.GetDayName(DateTime.Now.DayOfWeek));

Thread.CurrentThread.CurrentCulture = previousCulture;
```

2. Run the code and note the results, as shown in the following output:

```
*
* Localizing the DayOfWeek enum
*
Culture: dansk (Danmark), DayOfWeek: Thursday
Culture: dansk (Danmark), DayOfWeek: torsdag
Culture: dansk (Danmark), DayOfWeek: torsdag
```

Working with only a date or a time

.NET 6 introduced some new types for working with only a date value or only a time value, named `DateOnly` and `TimeOnly`.

These are better than using a `DateTime` value with a zero time to store a date-only value because it is type-safe and avoids misuse. `DateOnly` also maps better to database column types, for example, a date column in SQL Server. `TimeOnly` is good for setting alarms and scheduling regular meetings or the opening hours for an organization, and it maps to a time column in SQL Server.

Let's use them to plan the coronation of the new King of England, Charles III, probably during the spring of 2023:

1. Add statements to define the King's coronation, and a time for it to start, and then combine the two values to make a calendar entry so that we don't miss it, as shown in the following code:

```
SectionTitle("Working with only a date or a time");

DateOnly coronation = new(year: 2023, month: 5, day: 6);
WriteLine($"The King's Coronation is on {coronation.ToLongDateString()}");

TimeOnly starts = new(hour: 11, minute: 30);
WriteLine($"The King's Coronation starts at {starts}");

DateTime calendarEntry = coronation.ToDateTime(starts);
WriteLine($"Add to your calendar: {calendarEntry}");
```

2. Run the code and note the results, as shown in the following output:

```
The King's Coronation is on Saturday, 6 May 2023.
The King's Coronation starts at 11:30.
Add to your calendar: 06/05/2023 11:30:00.
```

Working with time zones

In the code example about the King's coronation, using a `TimeOnly` was not actually a good idea because the time-only value does not include information about time zone. It is only useful if you are in the correct time zone. `TimeOnly` is therefore a poor choice for an event. For events, we need to understand and handle time zones.

Understanding `DateTime` and `TimeZoneInfo`

The `DateTime` class has many useful members related to time zones, as shown in the following table:

Member	Description
Now property	A <code>DateTime</code> value that represents the current date and time in the local time zone
UtcNow property	A <code>DateTime</code> value that represents the current date and time in the UTC time zone
Kind property	A <code>DateTimeKind</code> value that indicates if the <code>DateTime</code> value is <code>Unspecified</code> , <code>Utc</code> , or <code>Local</code>

IsDaylightSavingTime method	A bool that indicates if the DateTime value is during DST
ToLocalTime method	Converts a UTC DateTime value to the equivalent local time
ToUniversalTime method	Converts a local DateTime value to the equivalent UTC time

The TimeZoneInfo class has many useful members, as shown in the following table:

Member	Description
Id property	A string that uniquely identifies the time zone.
Local property	A TimeZoneInfo value that represents the current local time zone. Varies depending on where the code executes.
Utc property	A TimeZoneInfo value that represents the UTC time zone.
StandardName property	A string for the name of the time zone when Daylight Saving is not active.
DaylightName property	A string for the name of the time zone when Daylight Saving is active.
DisplayName property	A string for the general name of the time zone.
BaseUtcOffset property	A TimeSpan that represents the difference between this time zone and the UTC time zone, ignoring any potential Daylight Saving adjustments.
SupportsDaylightSavingTime property	A bool that indicates if this time zone has Daylight Saving adjustments.
ConvertTime method	Converts a DateTime value to another DateTime value in a different time zone. You can specify the source and destination time zones.
ConvertTimeFromUtc method	Converts a DateTime value in the UTC time zone to a DateTime value in a specified time zone.
ConvertTimeToUtc method	Converts a DateTime value in a specified time zone to a DateTime value in the UTC time zone.
IsDaylightSavingTime method	Returns a bool indicating if the DateTime value is in Daylight Saving.
GetSystemTimeZones method	Returns a collection of time zones registered with the operating system.

Exploring DateTime and TimeZoneInfo

Use the TimeZoneInfo class to work with time zones:

1. Use your preferred code editor to add a new console app named `WorkingWithTimeZones` to the `Chapter07` solution/workspace:
 - In Visual Studio 2022, set the **Startup Project** to **Current selection**.
 - In Visual Studio Code, select `WorkingWithTimeZones` as the active OmniSharp project.
2. Statically and globally import the `System.Console` class.
3. Add a new class file named `Program.Helpers.cs`.
4. Modify its contents to define some helper methods to output a section title in a visually different way, output a list of all time zones in the current system, and output details about a `DateTime` or `TimeZoneInfo` object, as shown in the following code:

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

partial class Program
{
    static void SectionTitle(string title)
    {
        ConsoleColor previousColor = ForegroundColor;
        ForegroundColor = ConsoleColor.DarkYellow;
        WriteLine("");
        WriteLine($"* {title}");
        WriteLine("");
        ForegroundColor = previousColor;
    }

    static void OutputTimeZones()
    {
        // get the time zones registered with the OS
        ReadOnlyCollection<TimeZoneInfo> zones =
            TimeZoneInfo.GetSystemTimeZones();

        WriteLine("");
        WriteLine($"* {zones.Count} time zones:");
        WriteLine("");

        // order the time zones by Id instead of DisplayName
        foreach (TimeZoneInfo zone in zones.OrderBy(z => z.Id))
```

```

    {
        WriteLine($"{zone.Id}");
    }
}

static void OutputDateTime(DateTime dateTime, string title)
{
    SectionTitle(title);
    WriteLine($"Value: {dateTime}");
    WriteLine($"Kind: {dateTime.Kind}");
    WriteLine($"IsDaylightSavingTime: {dateTime.IsDaylightSavingTime()}");
    WriteLine($"ToLocalTime(): {dateTime.ToLocalTime()}");
    WriteLine($"ToUniversalTime(): {dateTime.ToUniversalTime()}");
}

static void OutputTimeZone(TimeZoneInfo zone, string title)
{
    SectionTitle(title);
    WriteLine($"Id: {zone.Id}");
    WriteLine($"IsDaylightSavingTime(DateTime.Now): {0}",
        zone.IsDaylightSavingTime(DateTime.Now));
    WriteLine($"StandardName: {zone.StandardName}");
    WriteLine($"DaylightName: {zone.DaylightName}");
    WriteLine($"BaseUtcOffset: {zone.BaseUtcOffset}");
}

static string GetCurrentZoneName(TimeZoneInfo zone, DateTime when)
{
    // time zone names change if Daylight Saving time is active
    // e.g. GMT Standard Time becomes GMT Summer Time
    return zone.IsDaylightSavingTime(when) ?
        zone.DaylightName : zone.StandardName;
}
}

```

5. In Program.cs, delete the existing statements. Add statements to output the current date and time in the local and UTC time zones, and then output details about the local and UTC time zones, as shown in the following code:

```

OutputTimeZones();

OutputDateTime(DateTime.Now, "DateTime.Now");
OutputDateTime(DateTime.UtcNow, "DateTime.UtcNow");

OutputTimeZone(TimeZoneInfo.Local, "TimeZoneInfo.Local");
OutputTimeZone(TimeZoneInfo.Utc, "TimeZoneInfo.Utc");

```

6. Run the console app and note the results, including the time zones registered on your operating system (there are 141 on my Windows 11 laptop), and that it is currently 4:17pm on 31 May 2022 in England, meaning I am in the GMT Standard Time zone. However, because DST is active, it is currently known as GMT Summer Time, which is one hour ahead of UTC, as shown in the following output:

```
*
* 141 time zones:
*
Afghanistan Standard Time
Alaskan Standard Time
...
West Pacific Standard Time
Yakutsk Standard Time
Yukon Standard Time
*
* DateTime.Now
*
Value: 31/05/2022 16:17:03
Kind: Local
IsDaylightSavingTime: True
ToLocalTime(): 31/05/2022 16:17:03
ToUniversalTime(): 31/05/2022 15:17:03
*
* DateTime.UtcNow
*
Value: 31/05/2022 15:17:03
Kind: Utc
IsDaylightSavingTime: False
ToLocalTime(): 31/05/2022 16:17:03
ToUniversalTime(): 31/05/2022 15:17:03
*
* TimeZoneInfo.Local
*
Id: GMT Standard Time
IsDaylightSavingTime(DateTime.Now): True
StandardName: GMT Standard Time
DaylightName: GMT Summer Time
BaseUtcOffset: 00:00:00
*
* TimeZoneInfo.Utc
*
Id: UTC
IsDaylightSavingTime(DateTime.Now): False
StandardName: Coordinated Universal Time
```



```
DaylightName: Coordinated Universal Time
BaseUtcOffset: 00:00:00
```



The **BaseUtcOffset** of the **GMT Standard Time** zone is zero because normally Daylight Saving is not active. That is why it is prefixed Base.

7. In `Program.cs`, add statements to prompt the user to enter a time zone (using Eastern Standard Time as a default), get that time zone, output details about it, and then compare a time entered by the user with the equivalent time in the other time zone, and catch potential exceptions, as shown in the following code:

```
Write("Enter a time zone or press Enter for US East Coast: ");
string zoneId = ReadLine()!;

if (string.IsNullOrEmpty(zoneId))
{
    zoneId = "Eastern Standard Time";
}

try
{
    TimeZoneInfo otherZone = TimeZoneInfo.FindSystemTimeZoneById(zoneId);
    OutputTimeZone(otherZone,
        $"TimeZoneInfo.FindSystemTimeZoneById(\"{zoneId}\")");

    SectionTitle($"What's the time in {zoneId}?");

    Write("Enter a local time or press Enter for now: ");
    string? timeText = ReadLine();
    DateTime localTime;
    if ((string.IsNullOrEmpty(timeText)) ||
        (!DateTime.TryParse(timeText, out localTime)))
    {
        localTime = DateTime.Now;
    }

    DateTime otherZoneTime = TimeZoneInfo.ConvertTime(
        dateTime: localTime, sourceTimeZone: TimeZoneInfo.Local,
        destinationTimeZone: otherZone);

    WriteLine("{0} {1} is {2} {3}.",
        localTime, GetCurrentZoneName(TimeZoneInfo.Local, localTime),
        otherZoneTime, GetCurrentZoneName(otherZone, otherZoneTime));
}
```

```

catch (TimeZoneNotFoundException)
{
    WriteLine($"The {zoneId} zone cannot be found on the local system.");
}
catch (InvalidTimeZoneException)
{
    WriteLine($"The {zoneId} zone contains invalid or missing data.");
}
catch (System.Security.SecurityException)
{
    WriteLine("The application does not have permission to read time zone
information.");
}
catch (OutOfMemoryException)
{
    WriteLine($"Not enough memory is available to load information on the
{zoneId} zone.");
}

```

8. Run the console app, press *Enter* for US East Coast, and then enter 12:30pm for the local time, and note the results, as shown in the following output:

```

Enter a time zone or press Enter for US East Coast:
*
* TimeZoneInfo.FindSystemTimeZoneById("Eastern Standard Time")
*
Id: Eastern Standard Time
IsDaylightSavingTime(DateTime.Now): True
StandardName: Eastern Standard Time
DaylightName: Eastern Summer Time
BaseUtcOffset: -05:00:00
*
* What's the time in Eastern Standard Time?
*
Enter a local time or press Enter for now: 12:30pm
31/05/2022 12:30:00 GMT Summer Time is 31/05/2022 07:30:00 Eastern Summer
Time.

```



My local time zone is GMT Standard Time so there is currently a five-hour time difference between me and the US East Coast. Your local time zone will be different.

9. Run the console app, copy one of the time zones to the clipboard and paste it at the prompt, and then press *Enter* for the local time. Note the results, as shown in the following output:

```

Enter a time zone or press Enter for US East Coast: AUS Eastern Standard
Time

```

```

*
* TimeZoneInfo.FindSystemTimeZoneById("AUS Eastern Standard Time")
*
Id: AUS Eastern Standard Time
IsDaylightSavingTime(DateTime.Now): False
StandardName: AUS Eastern Standard Time
DaylightName: AUS Eastern Summer Time
BaseUtcOffset: 10:00:00
*
* What's the time in AUS Eastern Standard Time?
*
Enter a local time or press Enter for now:
31/05/2022 17:00:04 GMT Summer Time is 01/06/2022 02:00:04 AUS Eastern
Standard Time.

```



Sydney, Australia, is currently nine hours ahead, so at 5pm for me, it is 2am on the following day for them.

Working with cultures

Internationalization is the process of enabling your code to correctly run all over the world. It has two parts, **globalization** and **localization**, and both of them are about working with cultures.

Globalization is about writing your code to accommodate multiple languages and region combinations. The combination of a language and a region is known as a culture. It is important for your code to know both the language and region because, for example, the date and currency formats are different in Quebec and Paris, despite them both using the French language.

There are **International Organization for Standardization (ISO)** codes for all culture combinations. For example, in the code `da-DK`, `da` indicates the Danish language and `DK` indicates the Denmark region, and in the code `fr-CA`, `fr` indicates the French language and `CA` indicates the Canada region.



ISO is not an acronym. ISO is a reference to the Greek word *isos* (which means equal). You can see a list of ISO culture codes at the following link: <https://lonewolfonline.net/list-net-culture-country-codes/>.

Localization is about customizing the user interface to support a language, for example, changing the label of a button to be `Close (en)` or `Fermer (fr)`. Since localization is more about the language, it doesn't always need to know about the region, although ironically enough, `standardization (en-US)` and `standardisation (en-GB)` suggest otherwise.



Good Practice: I am not a professional translator of software user interfaces, so take all examples in this chapter as general guidance. My research into French user interface labeling common practice led me to the following links, but it would be best to hire a professional if you are not a native language speaker: <https://french.stackexchange.com/questions/12969/translation-of-it-terms-like-close-next-search-etc> and <https://www.linguee.com/english-french/translation/close+button.html>.

Detecting and changing the current culture

Internationalization is a huge topic on which thousand-page books have been written. In this section, you will get a brief introduction to the basics using the `CultureInfo` and `RegionInfo` types in the `System.Globalization` namespace.

Let's write some code:

1. Use your preferred code editor to add a new console app named `WorkingWithCultures` to the `Chapter07` solution/workspace.
 - In Visual Studio Code, select `WorkingWithCultures` as the active OmniSharp project.
2. In the project file, statically and globally import the `System.Console` class and globally import the `System.Globalization` namespace so that we can use the `CultureInfo` class, as shown in the following markup:

```
<ItemGroup>
  <Using Include="System.Console" Static="true" />
  <Using Include="System.Globalization" />
</ItemGroup>
```

3. Add a new class file named `Program.Helpers.cs`, and modify its contents to add a method to the partial `Program` class that will output information about the cultures used for globalization and localization, as shown in the following code:

```
partial class Program
{
    static void OutputCultures(string title)
    {
        ConsoleColor previousColor = ForegroundColor;
        ForegroundColor = ConsoleColor.DarkYellow;

        WriteLine("*");
        WriteLine($"* {title}");
        WriteLine("*");

        // get the cultures from the current thread
        CultureInfo globalization = CultureInfo.CurrentCulture;
```

```

CultureInfo localization = CultureInfo.CurrentUICulture;

Writeline("The current globalization culture is {0}: {1}",
    globalization.Name, globalization.DisplayName);

Writeline("The current localization culture is {0}: {1}",
    localization.Name, localization.DisplayName);

Writeline("Days of the week: {0}",
    string.Join(", ", globalization.DateTimeFormat.DayNames));

Writeline("Months of the year: {0}",
    string.Join(", ", globalization.DateTimeFormat.MonthNames
    // some calendars have 13 months; most have 12 and the last is empty
    .TakeWhile(month => !string.IsNullOrEmpty(month))));

Writeline("1st day of this year: {0}",
    new DateTime(year: DateTime.Today.Year, month: 1, day: 1)
    .ToString("D", globalization));

Writeline("Number group separator: {0}",
    globalization.NumberFormat.NumberGroupSeparator);

Writeline("Number decimal separator: {0}",
    globalization.NumberFormat.NumberDecimalSeparator);

RegionInfo region = new RegionInfo(globalization.LCID);

Writeline("Currency symbol: {0}", region.CurrencySymbol);

Writeline("Currency name: {0} ({1})",
    region.CurrencyNativeName, region.CurrencyEnglishName);

Writeline("IsMetric: {0}", region.IsMetric);

Writeline();

ForegroundColor = previousColor;
}
}

```

4. In Program.cs, delete the existing statements and add statements to set the output encoding of the Console to support Unicode. Then, output information about the globalization and localization cultures. Then, prompt the user to enter a new culture code and show how that affects the formatting of common values such as dates and currency, as shown in the following code:

```
// to enable special characters like €
OutputEncoding = System.Text.Encoding.Unicode;

OutputCultures("Current culture");

Writeline("Example ISO culture codes:");

string[] cultureCodes = new[] {
    "da-DK", "en-GB", "en-US", "fa-IR",
    "fr-CA", "fr-FR", "he-IL", "pl-PL", "sl-SI" };

foreach (string code in cultureCodes)
{
    CultureInfo culture = CultureInfo.GetCultureInfo(code);
    Writeline("  {0}: {1} / {2}",
        culture.Name, culture.EnglishName, culture.NativeName);
}

Writeline();

Write("Enter an ISO culture code: ");
string? cultureCode = ReadLine();

if (string.IsNullOrEmpty(cultureCode))
{
    cultureCode = "en-US";
}

CultureInfo ci;

try
{
    ci = CultureInfo.GetCultureInfo(cultureCode);
}
catch (CultureNotFoundException)
{
    Writeline($"Culture code not found: {cultureCode}");
    Writeline("Exiting the app.");
    return;
}

// change the current cultures on the thread
CultureInfo.CurrentCulture = ci;
CultureInfo.CurrentUICulture = ci;

OutputCultures("After changing the current culture");
```

```

Write("Enter your name: ");
string? name = ReadLine();
if (string.IsNullOrEmpty(name))
{
    name = "Bob";
}

Write("Enter your date of birth: ");
string? dobText = ReadLine();

if (string.IsNullOrEmpty(dobText))
{
    // if they do not enter a DOB then use
    // sensible defaults for their culture
    dobText = ci.Name switch
    {
        "en-US" or "fr-CA" => "1/27/1990",
        "da-DK" or "fr-FR" or "pl-PL" => "27/1/1990",
        "fa-IR" => "1990/1/27",
        _ => "1/27/1990"
    };
}

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

if (string.IsNullOrEmpty(salaryText))
{
    salaryText = "34500";
}

DateTime dob = DateTime.Parse(dobText);
int minutes = (int)DateTime.Today.Subtract(dob).TotalMinutes;
decimal salary = decimal.Parse(salaryText);

WriteLine(
    "{0} was born on a {1:dddd}. {0} is {2:N0} minutes old. {0} earns {3:C}.",
    name, dob, minutes, salary);

```

When you run an application, it automatically sets its thread to use the culture of the operating system. I am running my code in London, UK, so the thread is set to English (Great Britain).

The code prompts the user to enter an alternative ISO code. This allows your applications to replace the default culture at runtime.

The application then uses standard format codes to output the day of the week using format code `dddd`, the number of minutes with thousand separators using format code `N0`, and the salary with the currency symbol. These adapt automatically, based on the thread's culture.

5. Run the code and enter `en-US` for the ISO code (or press *Enter*) and then enter some sample data including a date in a format valid for US English, as shown in the following output:

```
*
* Current culture
*

The current globalization culture is en-GB: English (United Kingdom)
The current localization culture is en-GB: English (United Kingdom)
Days of the week: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday
Months of the year: January, February, March, April, May, June, July,
August, September, October, November, December
1st day of this year: 01 January 2022
Number group separator: ,
Number decimal separator: .
Currency symbol: £
Currency name: British Pound (British Pound)
IsMetric: True

Example ISO culture codes:
da-DK: Danish (Denmark) / dansk (Danmark)
en-GB: English (United Kingdom) / English (United Kingdom)
en-US: English (United States) / English (United States)
fa-IR: Persian (Iran) / (فارسی)
fr-CA: French (Canada) / français (Canada)
fr-FR: French (France) / français (France)
he-IL: Hebrew (Israel) / (עברית)
pl-PL: Polish (Poland) / polski (Polska)
sl-SI: Slovenian (Slovenia) / slovenščina (Slovenija)

Enter an ISO culture code: en-US
*
* After changing the current culture
*

The current globalization culture is en-US: English (United States)
The current localization culture is en-US: English (United States)
Days of the week: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday
Months of the year: January, February, March, April, May, June, July,
August, September, October, November, December
1st day of this year: Saturday, January 1, 2022
Number group separator: ,
Number decimal separator: .
```



```

Currency symbol: $
Currency name: US Dollar (US Dollar)
IsMetric: False

Enter your name: Alice
Enter your date of birth: 3/30/1967
Enter your salary: 34500
Alice was born on a Thursday. Alice is 28,938,240 minutes old. Alice
earns $34,500.00

```

6. Run the code again and try Danish in Denmark (da-DK), as shown in the following output:

```

Enter an ISO culture code: da-DK
*
* After changing the current culture
*

The current globalization culture is da-DK: dansk (Danmark)
The current localization culture is da-DK: dansk (Danmark)
Days of the week: søndag, mandag, tirsdag, onsdag, torsdag, fredag,
lørdag
Months of the year: januar, februar, marts, april, maj, juni, juli,
august, september, oktober, november, december
1st day of this year: lørdag den 1. januar 2022
Number group separator: .
Number decimal separator: ,
Currency symbol: kr.
Currency name: dansk krone (Danish Krone)
IsMetric: True

Enter your name: Mikkell
Enter your date of birth: 16/3/1980
Enter your salary: 65000
Mikkell was born on a søndag. Mikkell is 22.119.840 minutes old. Mikkell
earns 65.000,00 kr.

```

In this example, only the date and salary are globalized into Danish. The rest of the text is hardcoded as English. Later, we will translate that English text into other languages. For now, let's see some other differences between cultures:

7. Run the code again and try Polish in Poland (pl-PL), and note the grammar rules in Polish make the day number possessive for the month name, so the month styczeń becomes stycznia, as shown in the following output:

```

The current globalization culture is pl-PL: polski (Polska)
...
Months of the year: styczeń, luty, marzec, kwiecień, maj, czerwiec,
lipiec, sierpień, wrzesień, październik, listopad, grudzień
1st day of this year: sobota, 1 stycznia 2022
...

```

```
Enter your name: Bob
Enter your date of birth: 1972/4/16
Enter your salary: 50000
Bob was born on a niedziela. Bob is 26 398 080 minutes old. Bob earns 50
000,00 zł.
```

8. Run the code again and try Persian in Iran (fa-IR), and note that dates in Iran must be specified as year/month/day, and that this year (2022) is the year 1400 in the Persian calendar, as shown in the following output:

```
The current globalization culture is fa-IR: (فارسی)
The current localization culture is fa-IR: (فارسی)
Days of the week: سه‌شنبه, چهارشنبه, پنجشنبه, شنبه, یکشنبه, دوشنبه, سه‌شنبه
Months of the year: فروردین, اردیبهشت, خرداد, تیر, مرداد, شهریور, مهر, آبان, آذر, دی, بهمن, اسفند
1st day of this year: 11 دی 1400
Number group separator: ,
Number decimal separator: .
Currency symbol: ریال
Currency name: (Iranian Rial) ریال فارسی
IsMetric: True

Enter your name: Cyrus
Enter your date of birth: 1372/4/16
Enter your salary: 50000
Cyrus was born on a دوشنبه. Cyrus is 15,242,400 minutes old. Cyrus earns
50,000.
```



Although I tried to confirm with a Persian reader if this example is correct, due to factors like right-to-left languages being tricky to work with in console apps and copying and pasting from a console window into a word processor, I apologize in advance to my Persian readers if this example is all messed up!

Temporarily using the invariant culture

Sometimes you might need to temporarily use a different culture without actually switching the current thread to that culture. For example, when automatically generating documents, queries, and commands that include data values, you might need to ignore your current culture and use a standard culture. For this purpose, you can use the invariant culture that is based on US English.

For example, you might need to generate a JSON document with a decimal number value and format the number with two decimal places, as shown in the following code:

```
decimal price = 54321.99M;
string document = $$$"
{
  "price": "{{price:N2}}"
}
$$$;
```

If you were to execute this on a Slovenian computer, you would get the following output:

```
{
  "price": "54.321,99"
}
```

If you then tried to insert this JSON document into a cloud database, then it would fail because it would not understand the number format that uses commas for decimals and dots for groups.

So, you can override the current culture and specify the invariant culture when outputting the number as a string value, as shown in the following code:

```
decimal price = 54321.99M;
string document = $$"""
{
  "price": "{{price.ToString("N2", CultureInfo.InvariantCulture)}}"
}
""";
```

If you were to execute this on a Slovenian (or any other culture) computer, you would now get the following output that would be successfully recognized by a cloud database and not throw exceptions:

```
{
  "price": "54,321.99"
}
```

Now let's see how to translate text from one language to another so that the label prompts are in the correct language for the current culture.

Localizing your user interface

A localized application is divided into two parts:

- An assembly containing code that is the same for all locales and contains resources for when no other resource file is found.
- One or more assemblies that contain the user interface resources that are different for different locales. These are known as **satellite assemblies**.

This model allows the initial application to be deployed with default invariant resources and, over time, additional satellite assemblies can be deployed as the resources are translated.

User interface resources include any text for messages, logs, dialog boxes, buttons, labels, or even file-names of images, videos, and so on. Resource files are XML files with the `.resx` extension. The filename includes a culture code, for example, `PacktResources.en-GB.resx` or `PacktResources.da-DK.resx`.

The automatic culture fallback search path for resources goes from specific culture (language and region) to neutral culture (language only) to invariant culture (supposed to be independent but basically US English).

If the current thread culture is en-AU (Australian English), then it will search for the resource file in the following order:

1. Australian English: `PacktResources.en-AU.resx`
2. Neutral English: `PacktResources.en.resx`
3. Invariant: `PacktResources.resx`

Defining and loading resources

To load resources from these satellite assemblies, we use some standard .NET types named `IStringLocalizer<T>` and `IStringLocalizerFactory`. Implementations of these are loaded from the .NET generic host as dependency services:

1. In the `WorkingWithCultures` project, add package references to Microsoft extensions for working with generic hosting and localization, as shown in the following markup:

```
<ItemGroup>
  <PackageReference Include="Microsoft.Extensions.Hosting"
                    Version="7.0.0" />
  <PackageReference Include="Microsoft.Extensions.Localization"
                    Version="7.0.0" />
</ItemGroup>
```

2. Build the `WorkingWithCultures` project to restore packages.
3. In the project folder, create a new folder named `Resources`.
4. In the `Resources` folder, add a new XML file named `PacktResources.resx`, and modify the contents to contain default invariant language resources (usually equivalent to US English), as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Enter your date of birth: </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
    <value>Enter your name: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Enter your salary: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} was born on a {1:dddd}. {0} is {2:N0} minutes old. {0}
    earns {3:C}.</value>
  </data>
</root>
```

5. In the `WorkingWithCultures` project folder, add a new class file named `PacktResources.cs` that will load text resources for the user interface, as shown in the following code:

```
using Microsoft.Extensions.Localization; // IStringLocalizer,
LocalizedString

public class PacktResources
{
    private readonly IStringLocalizer<PacktResources> localizer = null!;

    public PacktResources(IStringLocalizer<PacktResources> localizer)
    {
        this.localizer = localizer;
    }

    public string? GetEnterYourNamePrompt()
    {
        string resourceStringName = "EnterYourName";

        // 1. get the LocalizedString object
        LocalizedString localizedString = localizer[resourceStringName];

        // 2. check if the resource string was found
        if (localizedString.ResourceNotFound)
        {
            ConsoleColor previousColor = ConsoleColor;
            ConsoleColor = ConsoleColor.Red;
            WriteLine($"Error: resource string \"{resourceStringName}\" not
found."
                + Environment.NewLine
                + $"Search path: {localizedString.SearchedLocation}");
            ConsoleColor = previousColor;

            return $"{localizedString}: ";
        }
        // 3. return the found resource string
        return localizedString;
    }

    public string? GetEnterYourDobPrompt()
    {
        // LocalizedString has an implicit cast to string
        // that falls back to the key if the resource string is not found
        return localizer["EnterYourDob"];
    }
}
```

```

public string? GetEnterYourSalaryPrompt()
{
    return localizer["EnterYourSalary"];
}

public string? GetPersonDetails(
    string name, DateTime dob, int minutes, decimal salary)
{
    return localizer["PersonDetails", name, dob, minutes, salary];
}
}

```



For the `GetEnterYourNamePrompt` method, I broke the implementation down into steps to get useful information like checking if the resource string is found and showing the search path if not. The other method implementations use a simplified fallback to the key name for the resource string if they are not found.

6. In `Program.cs`, at the top, import the namespaces for working with hosting and dependency injection, and then configure a host that enables localization and the `PacktResources` service, as shown in the following code:

```

using Microsoft.Extensions.Hosting; // IHost, Host

// AddLocalization, AddTransient<T>
using Microsoft.Extensions.DependencyInjection;

using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddLocalization(options =>
        {
            options.ResourcesPath = "Resources";
        });

        services.AddTransient<PacktResources>();
    })
    .Build();

```



Good Practice: By default, `ResourcesPath` is an empty string, meaning it looks for `.resx` files in the current directory. We are going to make the project tidier by putting resources into a subfolder.

- After changing the current culture, add a statement to get the `PacktResources` service and use it to output localized prompts for the user to enter their name, date of birth, and salary, and then output their details, as highlighted in the following code:

```
OutputCultures("After changing the current culture");

PacktResources resources =
    host.Services.GetRequiredService<PacktResources>();

Write(resources.GetEnterYourNamePrompt());
string? name = ReadLine();
if (string.IsNullOrEmpty(name))
{
    name = "Bob";
}

Write(resources.GetEnterYourDobPrompt());
string? dobText = ReadLine();

if (string.IsNullOrEmpty(dobText))
{
    // if they do not enter a DOB then use
    // sensible defaults for their culture
    dobText = ci.Name switch
    {
        "en-US" or "fr-CA" => "1/27/1990",
        "da-DK" or "fr-FR" or "pl-PL" => "27/1/1990",
        "fa-IR" => "1990/1/27",
        _ => "1/27/1990"
    };
}

Write(resources.GetEnterYourSalaryPrompt());
string? salaryText = ReadLine();

if (string.IsNullOrEmpty(salaryText))
{
    salaryText = "34500";
}

DateTime dob = DateTime.Parse(dobText);
int minutes = (int)DateTime.Today.Subtract(dob).TotalMinutes;
decimal salary = decimal.Parse(salaryText);

WriteLine(resources.GetPersonDetails(name, dob, minutes, salary));
```

Testing globalization and localization

Now we can run the console app and see the resources being loaded:

1. Run the console app and enter da-DK for the ISO code. Note that the prompts are in US English because we currently only have invariant culture resources.



To save time and to make sure you have the correct structure, you can copy, paste, and rename the .resx files instead of creating empty new ones.

2. In the Resources folder, add a new XML file named `PacktResources.da.resx`, and modify the contents to contain non-region-specific Danish language resources, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Indtast din fødselsdato: </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
    <value>Indtast dit navn: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Indtast din løn: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} blev født på en {1:dddd}. {0} er {2:N0} minutter gammel.
    {0} tjener {3:C}.</value>
  </data>
</root>
```

3. In the Resources folder, add a new XML file named `PacktResources.fr.resx`, and modify the contents to contain non-region-specific French language resources, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Entrez votre date de naissance: </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
    <value>Entrez votre nom: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Entrez votre salaire: </value>
  </data>
```



```
<data name="PersonDetails" xml:space="preserve">
  <value>{0} est né un {1:dddd}. {0} a {2:N0} minutes. {0} gagne
{3:C}.</value>
</data>
</root>
```

4. In the Resources folder, add a new XML file named `PacktResources.fr-CA.resx`, and modify the contents to contain French language in Canada region resources, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Entrez votre date de naissance / Enter your date of birth:
    </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
    <value>Entrez votre nom / Enter your name: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Entrez votre salaire / Enter your salary: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} est né un {1:dddd}. {0} a {2:N0} minutes. {0} gagne
{3:C}.</value>
  </data>
</root>
```

5. In the Resources folder, add a new XML file named `PacktResources.pl-PL.resx`, and modify the contents to contain Polish language in Poland region resources, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Wpisz swoją datę urodzenia: </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
    <value>Wpisz swoje imię i nazwisko: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Wpisz swoje wynagrodzenie: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} urodził się na {1:dddd}. {0} ma {2:N0} minut. {0} zarabia
{3:C}.</value>
  </data>
</root>
```

6. In the Resources folder, add a new XML file named `PacktResources.fa-IR.resx`, and modify the contents to contain Farsi language in Iranian region resources, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>دینک دراو ار دوخ دلوت خیرات / Enter your date of birth:
    </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
    <value>نک دراو ار تم سا / Enter your name: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>دینک دراو ار دوخ قوقح / Enter your salary: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} تس ا هقیقد {2:N0} دم آ ای ند هب {1:dddd} رد {3:C}.</value>
  </data>
</root>
```

7. Run the code and enter `da-DK` for the ISO code. Note that the prompts are in Danish, as shown in the following output:

```
The current localization culture is da-DK: dansk (Danmark)
...
Indtast dit navn: Bob
Indtast din fødselsdato: 3/4/1987
Indtast din løn: 45449
Bob blev født på en fredag. Bob er 18.413.280 minutter gammel. Bob tjener
45.449,00 kr.
```

8. Run the code and enter `fr-FR` for the ISO code. Note that the prompts are in French only, as shown in the following output:

```
The current localization culture is fr-FR: français (France)
...
Entrez votre nom: Monique
Entrez votre date de naissance: 2/12/1990
Entrez votre salaire: 45000
Monique est né un Dimanche. Monique a 16 485 120 minutes. Monique gagne
45 000,00 €.
```

9. Run the code and enter `fr-CA` for the ISO code. Note that the prompts are in French and English because Canada might have a requirement to support both as official languages, as shown in the following output:

```
The current localization culture is fr-CA: français (Canada)
```

```
...
Entrez votre nom / Enter your name: Sophie
Entrez votre date de naissance / Enter your date of birth: 4/5/2001
Entrez votre salaire / Enter your salary: 65000
Sophie est né un jeudi. Sophie a 11 046 240 minutes. Sophie gagne 65
000,00 $ CA.
```

10. Run the code and enter fa-IR for the ISO code. Note that the prompts are in Persian/Farsi and English, and there is the additional complication of a right-to-left language, as shown in the following output:

```
The current localization culture is fa-IR: (ناری ا) یراف
...
نک دراو ار تمسا / Enter your name: Hoshyar
دینک دراو ار دوخ دلوت خیرات / Enter your date of birth: 1370/3/6
دینک دراو ار دوخ قوقج / Enter your salary: 90000
Hoshyar تساهقیقد 11,190,240 دمآیند هبه بنشراهچ رد Hoshyar 90,000. لای
```



If you need to work with Persian dates then there are NuGet packages with open-source GitHub repositories that you can try, although I cannot vouch for their correctness, like <https://github.com/VahidN/DNTPersianUtils.Core> and <https://github.com/imanabidi/PersianDate.NET>.

11. In the Resources folder, in `PacktResources.da.resx`, modify the contents to deliberately change the key for the prompt to enter your name by appending `Wrong`, as shown highlighted in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Indtast din fødselsdato: </value>
  </data>
  <data name="EnterYourNameWrong" xml:space="preserve">
    <value>Indtast dit navn: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Indtast din løn: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} blev født på en {1:dddd}. {0} er {2:N0} minutter gammel.
    {0} tjener {3:C}.</value>
  </data>
</root>
```

12. Run the code and enter da-DK for the ISO code. Note that the prompts are in Danish, except for the enter your name prompt in English, due to it falling back to the default resource file, as shown in the following output:

```
The current localization culture is da-DK: dansk (Danmark)
...
Enter your name: Bob
Indtast din fødselsdato: 3/4/1987
Indtast din løn: 45449
Bob blev født på en fredag. Bob er 18.413.280 minutter gammel. Bob tjener
45.449,00 kr.
```

13. In the Resources folder, in PacktResources.resx, modify the contents to deliberately change the key for the prompt to enter your name by appending Wrong.
14. Run the code and enter da-DK for the ISO code. Note that the prompts are in Danish, except for the enter your name prompt, which shows an error and uses the key name as a last resort fallback, as shown in the following output:

```
The current localization culture is da-DK: dansk (Danmark)
...
Error: resource string "EnterYourName" not found.
Search path: WorkingWithCultures.Resources.PacktResources
EnterYourName: Bob
Indtast din fødselsdato: 3/4/1987
Indtast din løn: 45449
Bob blev født på en fredag. Bob er 18.413.280 minutter gammel. Bob tjener
45.449,00 kr.
```

15. Remove the Wrong suffix in both resource files.
16. In **File Explorer**, open the WorkingWithCultures project folder, and select the bin/Debug/net7.0/da folder, as shown in *Figure 7.1*:

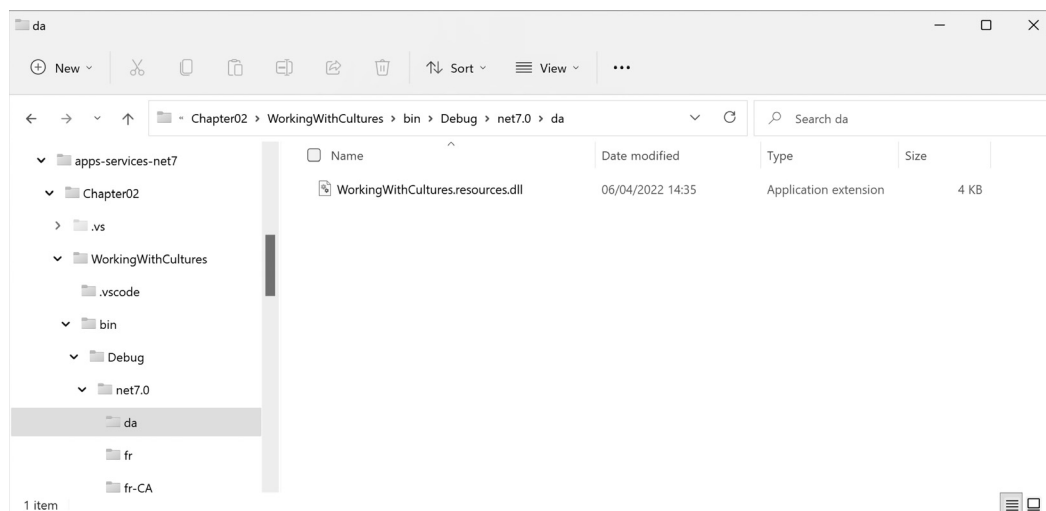


Figure 7.1: The satellite assembly folders for culture resources

17. Note the satellite assembly named `WorkingWithCultures.resources.dll` for the neutral Danish resources.

Any other culture resource assemblies are named the same but stored in folders that match the appropriate culture code. You can use tools like ResX Resource Manager, found at <https://dotnetfoundation.org/projects/resx-resource-manager>, to create many more .resx files, compile them into satellite assemblies, and then deploy them to users without needing to recompile the original console app.



Good Practice: Consider whether your application needs to be internationalized and plan for that before you start coding! Think about all the data that will need to be globalized (date formats, number formats, and sorting text behavior). Write down all the pieces of text in the user interface that will need to be localized.

Microsoft has an online tool (found at the following link: <https://www.microsoft.com/en-us/Language/>) that can help you translate text in your user interfaces:

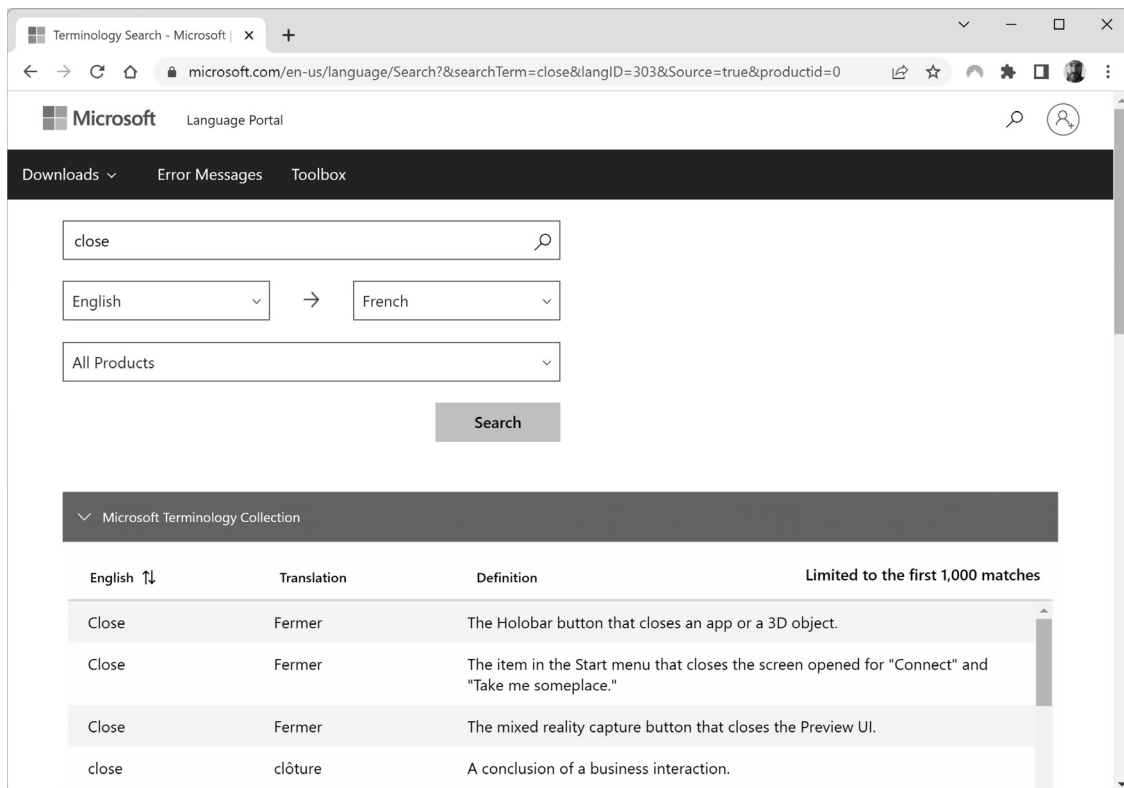


Figure 7.2: Microsoft user interface online text translation tool

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring the topics in this chapter with deeper research.

Exercise 7.1 – Test your knowledge

Use the web to answer the following questions:

1. What is the difference between localization, globalization, and internationalization?
2. What is the smallest measurement of time available in .NET?
3. How long is a “tick” in .NET?
4. In what scenario might you use a `DateOnly` value instead of a `DateTime` value?
5. For a time zone, what does its `BaseUtcOffset` property tell you?
6. How can you get information about the local time zone in which your code is executing?
7. For a `DateTime` value, what does its `Kind` property tell you?
8. How can you control the current culture for your executing code?
9. What is the ISO culture code for Welsh?
10. How do localization resource file fallbacks work?

Exercise 7.2 – Explore topics

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

<https://github.com/markjprice/apps-services-net7/blob/main/book-links.md#chapter-7---handling-dates-times-and-internationalization>

Exercise 7.3 – Learn from expert Jon Skeet

Jon Skeet is a world-renowned expert on internationalization. Watch him present *Working with Time is Easy* at the following link: <https://www.youtube.com/watch?v=saeKBuPewcU>.

Read about `NodaTime`, Jon’s library that provides a better date and time API for .NET: <https://nodatime.org/>.

Summary

In this chapter, you:

- Explored dates and times.
- Learned how to handle time zones.
- Learned how to internationalize your code using globalization and localization.

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