Introducing Practical Applications of C# and .NET

The third part of this module is about practical applications of C# and .NET. You will learn how to build cross-platform projects such as websites, web services, Windows desktop apps, and cross-platform mobile apps, as well as how to add intelligence to them with machine learning.

Microsoft calls platforms for building applications **App Models**.

I recommend that you work through this and subsequent sections sequentially because later sections will reference projects in earlier sections, and you will build up sufficient knowledge and skills to tackle the trickier problems in later sections.

In this section, we will cover the following topics:

- Understanding app models for C# and .NET
- Understanding SignalR
- Understanding Blazor
- Building an entity data model for Northwind

Understanding app models for C# and .NET

Since this module is about C# 9 and .NET 5, we will learn about app models that use them to build the practical applications that we will encounter in the remaining sections of this module.

More Information: Microsoft has extensive guidance for implementing App Models such as ASP.NET web applications, Xamarin mobile apps, and UWP apps in its .NET Application Architecture Guidance documentation, which you can read at the following link: https://www.microsoft.com/net/learn/architecture

Building websites using ASP.NET Core

Websites are made up of multiple web pages loaded statically from the filesystem or generated dynamically by a server-side technology such as ASP.NET Core. A web browser makes GET requests using URLs that identify each page and can manipulate data stored on the server using POST, PUT, and DELETE requests.

With many websites, the web browser is treated as a presentation layer, with almost all of the processing performed on the server side. A small amount of JavaScript might be used on the client side to implement some presentation features, such as carousels.

ASP.NET Core provides multiple technologies for building websites:

- **ASP.NET Core Razor Pages** and **Razor class libraries** are ways to dynamically generate HTML for simple websites.
- **ASP.NET Core MVC** is an implementation of the Model-View-Controller design pattern that is popular for developing complex websites.
- **Blazor** lets you build server-side or client-side components and user interfaces using C# instead of JavaScript.

Building websites using a web content management system

Most websites have a lot of content, and if developers had to be involved every time some content needed to be changed, that would not scale well. A web **Content Management System** (**CMS**) enables developers to define content structure and templates to provide consistency and good design while making it easy for a non-technical content owner to manage the actual content. They can create new pages or blocks of content, and update existing content, knowing it will look great for visitors with minimal effort.

There is a multitude of CMSs available for all web platforms, like WordPress for PHP or django CMS for Python. Enterprise-level CMSs for .NET Framework include Episerver and Sitecore, but neither is available yet for .NET Core or .NET 5 and later. CMSs that support .NET Core include Piranha CMS, Squidex, and Orchard Core.

The key benefit of using a CMS is that it provides a friendly content management user interface. Content owners log in to the website and manage the content themselves. The content is then rendered and returned to visitors using ASP.NET Core MVC controllers and views.

In summary, C# and .NET can be used on both the server side and the client side to build websites, as shown in the following diagram:

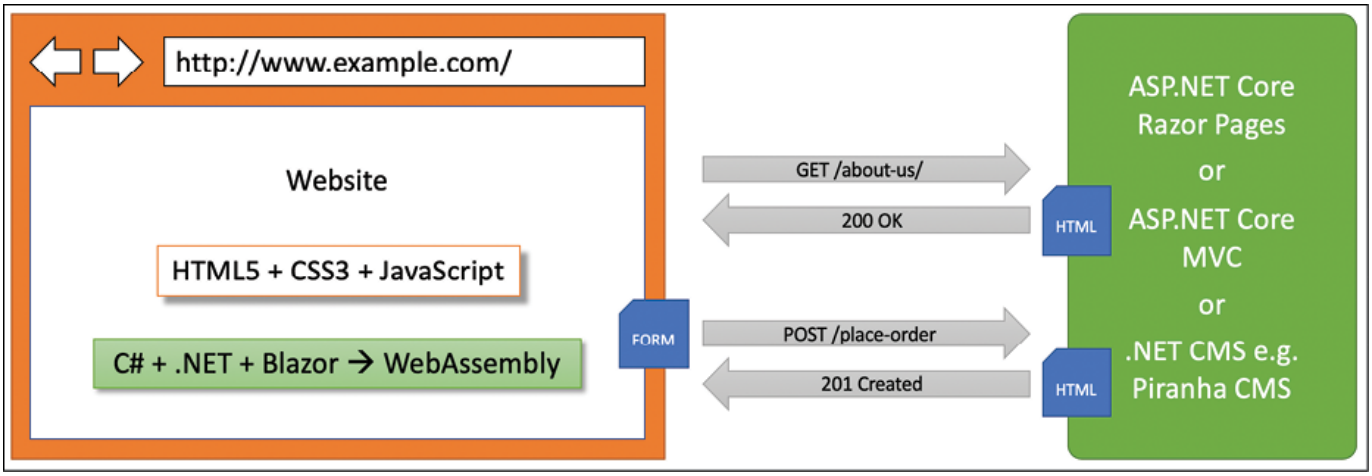

Figure 6.1: The use of C# and .NET to build websites

Understanding web applications

Web applications, also known as **Single-Page Applications** (**SPAs**), are made up of a single web page built with a frontend technology such as Angular, React, Vue, or a proprietary JavaScript library that can make requests to a backend web service for getting more data when needed and posting updated data, using common serialization formats, such as XML and JSON. The canonical examples are Google web apps like Gmail, Maps, and Docs.

With a web application, the client side uses JavaScript libraries to implement sophisticated user interactions, but most of the important processing and data access still happens on the server side, because the web browser has limited access to local system resources.

More Information: JavaScript is loosely typed and is not designed for complex projects so most JavaScript libraries these days use Microsoft TypeScript, which adds strong typing to JavaScript and is designed with many modern language features for handling complex implementations. TypeScript 4.0 was released in August 2020. You can read more about TypeScript at the following link: https://www.typescriptlang.org

.NET has project templates for JavaScript and TypeScript-based SPAs, but we will not spend any time learning how to build JavaScript and TypeScript-based SPAs in this module, even though these are commonly used with ASP.NET Core as the backend.

Building and consuming web services

Although we will not learn about JavaScript and TypeScript-based SPAs, we will learn how to build a web service using the **ASP.NET Core Web API**, and then call that web service from the server-side code in our ASP.NET Core websites, and then later, we will call that web service from Blazor web components, Windows desktop, and cross-platform mobile apps.

Building intelligent apps

In a traditional app, the algorithms it uses to process its data are designed and implemented by a human. Humans are good at many things, but writing complex algorithms is not one of them, especially algorithms for spotting meaningful patterns in vast quantities of data.

Machine learning algorithms that work with custom trained models like those provided by Microsoft's **ML.NET** can add intelligence to your apps. We will use ML.NET algorithms with custom trained models to process the tracked behavior of visitors to a product website and then make recommendations for other product pages that they might be interested in. It will work rather like how Netflix recommends films and TV shows you might like based on your previous behavior and the behavior of people who have expressed similar interests to you.

Understanding SignalR

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

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

The benefits of this technique were obvious and soon all browsers added the same component. Google took maximum advantage of this capability to build clever web applications such as Google Maps and Gmail. A few years later, the technique became popularly known as **Asynchronous JavaScript and XML** (**AJAX**).

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

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

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

More Information: You can read more about WebSocket at the following link: https://en.wikipedia.org/wiki/WebSocket

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

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

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

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

SignalR is an API for server-to-client **remote procedure calls** (**RPCs**). The RPCs call JavaScript functions on clients from server-side .NET Core code. SignalR has hubs to

define the pipeline and handles the message dispatching automatically using two built-in hub protocols: JSON and a binary one based on MessagePack.

More Information: You can read more about MessagePack at the following link: https://msgpack.org

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

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

More Information: You can read more about SignalR at the following link: https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction

Understanding Blazor

Blazor lets you build shared components and interactive web user interfaces using C# instead of JavaScript. In April 2019, Microsoft announced that Blazor "is no longer experimental and we are committing to ship it as a supported web UI framework including support for running client-side in the browser on WebAssembly."

JavaScript and friends

Traditionally, any code that needs to execute in a web browser is written using the JavaScript programming language or a higher-level technology that **transpiles** (transforms or compiles) into JavaScript. This is because all browsers have supported JavaScript for about two decades, so it has become the lowest common denominator for implementing business logic on the client side.

JavaScript does have some issues, however. First, although it has superficial similarities to C-style languages like C# and Java, it is actually very different once you dig beneath the surface. Second, it is a dynamically typed pseudo-functional language that uses prototypes instead of class inheritance for object reuse. It might look human, but you will get a surprise when it's revealed to actually be a Skrull.

Wouldn't it be great if we could use the same language and libraries in a web browser as we do on the server side?

Silverlight – C# and .NET using a plugin

Microsoft made a previous attempt at achieving this goal with a technology named **Silverlight**. When Silverlight 2.0 was released in 2008, a C# and .NET developer could use their skills to build libraries and visual components that were executed in the web browser by the Silverlight plugin.

By 2011 and Silverlight 5.0, Apple's success with the iPhone and Steve Jobs' hatred of browser plugins like Flash eventually led to Microsoft abandoning Silverlight since, like Flash, Silverlight is banned from iPhones and iPads.

WebAssembly – a target for Blazor

A recent development in browsers has given Microsoft the opportunity to make another attempt. In 2017, the WebAssembly Consensus was completed and all major browsers now support it: Chromium (Chrome, Edge, Opera, Brave), Firefox, and WebKit (Safari). It is not supported by Microsoft's Internet Explorer because it is a legacy web browser.

WebAssembly (**Wasm**) is a binary instruction format for a virtual machine that provides a way to run code written in multiple languages on the web at near-native speed. Wasm is designed as a portable target for the compilation of high-level languages like C#.

More Information: You can learn more about WebAssembly at the following link: https://webassembly.org

Blazor on the server side or client side

Blazor is a single programming or **app** model with two **hosting** models:

- Server-side Blazor runs on the server side using SignalR to communicate with the client side, and it shipped as part of .NET Core 3.0.
- Client-side Blazor runs on the client side using WebAssembly, and it shipped as an extension to .NET Core 3.1, although it is only a Current release, which is why it was versioned as 3.2.

This means that a web developer can write Blazor components once, and then run them either on the server side or client side (with careful planning).

More Information: You can read the official documentation for Blazor at the following link: https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor

Building an entity data model for Northwind

Practical applications usually need to work with data in a relational database or another data store. In this section, we will define an entity data model for the `Northwind` database stored in SQLite. It will be used in most of the apps that we create in subsequent sections.

Although macOS includes an installation of SQLite by default, if you are using Windows or a variety of Linux then you might need to download, install, and configure SQLite for your operating system. Instructions to do so can be found in *Section 4*, *Working with Databases Using Entity Framework Core*. In that section, you will also find instructions for installing the `dotnet-ef` tool, which you will use to scaffold an entity model from an existing database.

Good Practice: You should create a separate class library project for your entity data models. This allows easier sharing between backend web servers and frontend desktop, mobile, and Blazor clients.

Creating a class library for Northwind entity models

You will now define entity data models in a .NET 5 class library so that they can be reused in other types of projects including client-side app models.

Generating entity models using dotnet-ef

First, we will automatically generate some entity models using the EF Core command-line tool:

1. In your existing `Code` folder, create a folder named `PracticalApps`.
2. In Visual Studio Code, open the `PracticalApps` folder.
3. Create the `Northwind.db` file by copying the `Northwind.sql` file into the `PracticalApps` folder, and then enter the following command in **TERMINAL**:

```
sqlite3 Northwind.db -init Northwind.sql
```

4. Be patient because this command might take a while to create the database structure, as shown in the following output:

```
-- Loading resources from Northwind.sql
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
sqlite>
```

5. Press `Ctrl` + `D` on macOS or `Ctrl` + `C` on Windows to exit SQLite command mode.
6. In the **File** menu, close the `PracticalApps` folder.
7. In Visual Studio Code, navigate to **File | Save Workspace As**..., enter the name `PracticalApps`, change to the `PracticalApps` folder, and click **Save**.
8. In the `PracticalApps` folder, create a folder named `NorthwindEntitiesLib`.
9. Add the `NorthwindEntitiesLib` folder to the workspace.
10. Navigate to **Terminal | New Terminal** and select **NorthwindEntitiesLib**.
11. In **TERMINAL**, enter the command `dotnet new classlib`.
12. Open the `NorthwindEntitiesLib.csproj` file, and add two package references for EF Core for SQLite and design-time support, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Sqlite"
      Version="5.0.0" />
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Design"
      Version="5.0.0" />
  </ItemGroup>
</Project>
```

13. In **TERMINAL**, download the referenced package and compile the current project, as shown in the following command:

```
dotnet build
```

14. Delete the `Class1.cs` file.
15. In **TERMINAL**, generate entity class models for all tables, as shown in the following command:

```
dotnet ef dbcontext scaffold "Filename=../Northwind.db" Microsoft.EntityFrameworkCore.Sqlite --namespace Packt.Shared --data-annotations --conte
```

    Note the following:

    - The command to perform: `dbcontext scaffold`
    - The connection string: `"Filename=../Northwind.db"`
    - The database provider: `Microsoft.EntityFrameworkCore.Sqlite`
    - The namespace: `--namespace Packt.Shared`
    - To use data annotations as well as the Fluent API: `--data-annotations`
    - To rename the context from `[database_name]Context`: `--context Northwind`

16. Note the build messages and warnings, as shown in the following output:

```
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string, you should move it out of source code. You can avoid scaffolding the con
```

Manually improving the class-to-table mapping

Second, we will make some small changes to improve the model mapping.

We will make the changes to these entity classes: `Category.cs`, `Customer.cs`, `Employee.cs`, `Order.cs`, `OrderDetail.cs`, `Product.cs`, `Shipper.cs`, `Supplier.cs`, and `Territory.cs`.

The changes we will make to each entity class are the following:

1. Change all primary or foreign key properties to use standard naming, for example, change `CategoryId` to `CategoryID`. This will allow you to also remove the attribute `[Column]` attribute that maps the property to the column in the table, as shown in the following code:

```
// before
[Key]
[Column("CategoryID")]
public long CategoryId { get; set; }
// after
[Key]
public long CategoryID { get; set; }
```

2. Decorate all `string` properties with the `[StringLength]` attribute to limit the maximum number of characters allowed for the matching field using the information in the `[Column]` attribute, as shown in the following code:

```
// before
[Required]
[Column(TypeName = "nvarchar (15)")]
public string CategoryName { get; set; }
// after
[Required]
[Column(TypeName = "nvarchar (15)")]
[StringLength(15)]
public string CategoryName { get; set; }
```

3. Open the `Customer.cs` file and add a regular expression to validate its primary key value to only allow uppercase Western characters, as shown in the following code:

```
[Key]
[Column(TypeName = "nchar (5)")]
[StringLength(5)]
[RegularExpression("[A-Z]{5}")]
public string CustomerID { get; set; }
```

4. Change any date/time properties, for example, in `Employee.cs`, to use a nullable `DateTime` instead of an array of bytes, as shown in the following code:

```
// before
[Column(TypeName = "datetime")]
public byte[] BirthDate { get; set; }
// after
[Column(TypeName = "datetime")]
public DateTime? BirthDate { get; set; }
```

5. Change any `money` properties, for example, in `Order.cs`, to use a nullable `decimal` instead of an array of bytes, as shown in the following code:

```
// before
[Column(TypeName = "money")]
public byte[] Freight { get; set; }
// after
[Column(TypeName = "money")]
public decimal? Freight { get; set; }
```

6. Change any `bit` properties, for example, in `Product.cs`, to use a `bool` instead of an array of bytes, as shown in the following code:

```
// before
[Required]
[Column(TypeName = "bit")]
public byte[] Discontinued { get; set; }
// after
[Required]
[Column(TypeName = "bit")]
public bool Discontinued { get; set; }
```

Now that we have a class library for the entity classes, we can create a class library for the database context.

Creating a class library for a Northwind database context

You will now define a database context class library:

1. In the `PracticalApps` folder, create a folder named `NorthwindContextLib`.
2. Add the `NorthwindContextLib` folder to the workspace.
3. Navigate to **Terminal** | **New Terminal** and select `NorthwindContextLib`.
4. In **TERMINAL**, enter the command `dotnet new classlib`.
5. Navigate to **View** | **Command Palette**, enter and select **OmniSharp: Select Project**, and select the `NorthwindContextLib` project.
6. Modify `NorthwindContextLib.csproj` to add a reference to the `NorthwindEntitiesLib` project and the Entity Framework Core package for SQLite, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include=
      "..\NorthwindEntitiesLib\NorthwindEntitiesLib.csproj" />
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.SQLite"
      Version="5.0.0" />
  </ItemGroup>
</Project>
```

7. In the `NorthwindContextLib` project, delete the `Class1.cs` class file.
8. Drag and drop the `Northwind.cs` file from the `NorthwindEntitiesLib` folder to the `NorthwindContextLib` folder.
9. In `Northwind.cs`, modify the statements to remove the compiler warning about the connection string. Then remove the Fluent API statements for validation and defining keys and relationships except for `OrderDetail` because multi-field primary keys can only be defined using the Fluent API. This is because they are duplicating work done by the attributes in the entity model classes, as shown in the following code:

```
using Microsoft.EntityFrameworkCore;
#nullable disable
namespace Packt.Shared
{
  public partial class Northwind : DbContext
  {
    public Northwind()
    {
    }
    public Northwind(DbContextOptions<Northwind> options)
      : base(options)
    {
    }
    public virtual DbSet<Category> Categories { get; set; }
    public virtual DbSet<Customer> Customers { get; set; }
    public virtual DbSet<Employee> Employees { get; set; }
    public virtual DbSet<EmployeeTerritory> EmployeeTerritories
      { get; set; }
    public virtual DbSet<Order> Orders { get; set; }
    public virtual DbSet<OrderDetail> OrderDetails { get; set; }
    public virtual DbSet<Product> Products { get; set; }
    public virtual DbSet<Shipper> Shippers { get; set; }
    public virtual DbSet<Supplier> Suppliers { get; set; }
    public virtual DbSet<Territory> Territories { get; set; }
    protected override void OnConfiguring(
      DbContextOptionsBuilder optionsBuilder)
    {
      if (!optionsBuilder.IsConfigured)
      {
        optionsBuilder.UseSqlite("Filename=../Northwind.db");
      }
    }
    protected override void OnModelCreating(
      ModelBuilder modelBuilder)
    {
      modelBuilder.Entity<OrderDetail>(entity =>
      {
        entity.HasKey(x => new { x.OrderID, x.ProductID });
        entity.HasOne(d => d.Order)
          .WithMany(p => p.OrderDetails)
          .HasForeignKey(x => x.OrderID)
          .OnDelete(DeleteBehavior.ClientSetNull);
        entity.HasOne(d => d.Product)
                      .WithMany(p => p.OrderDetails)
                      .HasForeignKey(x => x.ProductID)
                      .OnDelete(DeleteBehavior.ClientSetNull);
      });

      modelBuilder.Entity<Product>()
        .Property(product => product.UnitPrice)
        .HasConversion<double>();
      OnModelCreatingPartial(modelBuilder);
    }
    partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
  }
}
```

10. In **TERMINAL**, restore packages and compile the class libraries to check for errors by entering the command `dotnet build`.

We will override the default database connection string in any projects such as websites that need to work with the `Northwind` database, so the class derived from `DbContext` must have a constructor with a `DbContextOptions` parameter for this to work.