Getting Started with Blazor

In the previous module, we learned about the fundamentals of Razor View Engine and understood how it powers different web frameworks to render web UIs. We covered hands-on coding exercises to get a feel for both the MVC and Razor Pages web frameworks that ship with ASP.NET Core for building powerful web applications. In this module, we are going to look at the latest addition to the ASP.NET Core web framework –Blazor.

The Blazor web framework is a huge topic; this book splits the topic into two modules for you to easily grasp the core concepts and fundamentals needed for you to get started with the framework. By the time you've finished both modules, you will know how Blazor applications can be used in concert with various technologies to build powerful and dynamic web applications.

Here are the topics that we'll cover in this module:

- Understanding the Blazor web framework and its different flavors
- Understanding the goal of what we are going to build using various technologies
- Creating a simple Web API
- Learning how to use in-memory databases with Entity Framework Core
- Learning how to perform real-time updates with SignalR
- Implementing the backend application for the tourist spot application

This module is mainly targeted at beginner- and intermediate-level .NET developers with prior C# experience, who want to jump into Blazor and get their hands dirty with practical examples. It will help you learn the basics of the Blazor programming model, for you to build your first web application from scratch.

Technical requirements

This module uses Visual Studio 2019 to build the project.

Before diving into this module, make sure that you have a basic understanding of ASP.NET Core and C# in general, because we're not going to cover their fundamentals in this module.

Understanding the Blazor web framework

Blazor was introduced as an experimental project in early 2018. It's the latest addition to the Single-Page Application (SPA)-based ASP.NET Core web frameworks. You can think of it as similar to React, Angular, Vue, and other SPA-based frameworks, but it is powered by C# and the Razor markup language, enabling you to create web applications without having to write JavaScript. Yes, you heard that right – without JavaScript! Though Blazor doesn't require you to use JavaScript, it offers a feature called JavaScript interoperability (JS interop), which allows you to invoke JavaScript code from your C# code and vice versa. Pretty neat!

Regardless of whether you are coming from a Windows, Xamarin, Web Forms, or traditional ASP.NET MVC development background, or are completely new to ASP.NET Core and want to take your skills to the next level, Blazor is definitely a great choice for you since it enables you to use your existing C# skills to write web UIs. Learning the framework itself is easy, as long as you know basic HTML and CSS. It was designed

to enable C# developers to take advantage of their skills to easily transition to the web paradigm for building SPA-based web applications.

Reviewing the different flavors of Blazor

Before we talk about the different flavors of Blazor, let's have a quick overview of Razor components. Razor components are the building blocks for Blazor applications. They are selfcontained chunks of UI that are composed of HTML, CSS, and C# code using Razor markup. These components can be a whole page, a section in a page, a form, or a dialog box. Components are very flexible, lightweight, and easy to reuse, nest, or even share across different applications, such as Razor Pages or MVC apps. Any changes that happen in a component, such as a button click that affects the state of an app, will render a graph and a UI diff is calculated. This diff contains a set of DOM edits that are required to update the UI and is applied by the browser.

Blazor has gained a lot of popularity, even if the framework is still pretty much new to the market. In fact, big UI providers, such as Telerik, Syncfusion, and DevExpress, already offer a bunch of Razor components that you can integrate into your application. There are also other open-source projects that provide ready-made components that you can use for free, such as MatBlazor and RadZen.

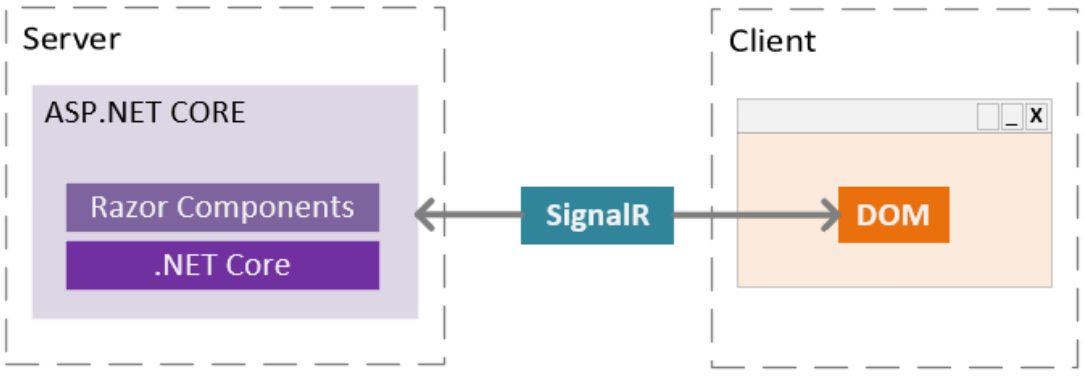
Blazor comes with two main hosting models:

- Blazor Server
- Blazor WebAssembly (WASM)

Let's do a quick rundown of each.

Blazor Server

Blazor Server, often referred to as server-side Blazor, is a type of Blazor application that runs on a server. It was the first Blazor model to be officially shipped in .NET Core and is ready for production use. Figure below shows how Blazor Server works under the hood.



In the preceding diagram, we can see that the server-based Blazor application is wrapped within the ASP.NET Core application, allowing it to run and be executed on the server. It mainly uses SignalR to manage and drive real-time server updates to the UI and vice versa. This means that maintaining the application state, DOM interactions, and rendering of the components happens in the server, and SignalR will notify the UI via a hub with a diff to update the DOM when the application state changes.

In the preceding diagram, we can see that the server-based Blazor application is wrapped within the ASP.NET Core application, allowing it to run and be executed on the server. It mainly uses SignalR to manage and drive real-time server updates to the UI and vice versa. This means that maintaining the application state, DOM interactions, and rendering of the components happens in the server, and SignalR will notify the UI via a hub with a diff to update the DOM when the application state changes.

The pros of this are as follows:

- No need for you to write JavaScript to run the app.
- Your application code stays on the server.
- Since the application runs on the server, you can take advantage of ASP.NET Core features, such as hosting a Web API in a shared project, integrating other middleware, and connecting to a database and other external dependencies via DI.
- Enables fast load times and small download sizes, since the server takes care of heavy workloads.
- Runs on any browser.
- Great debugging capability.

The cons are as follows:

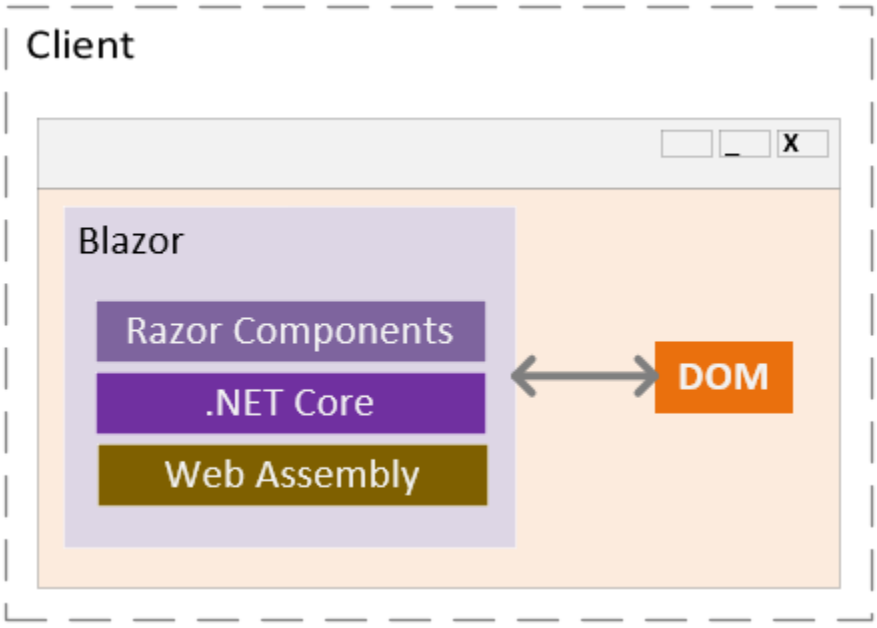
- It requires a server to bootstrap the application.
- No offline support. SignalR requires an open connection to the server. The moment the server goes down, so does your application.
- There is higher network latency, since every UI interaction needs to call the server to re-render the component state. This can be resolved if you have a geo-replicated server that hosts your app in various regions.
- Maintaining and scaling can be costly and difficult. This is because every time you open an instance of a page, a separate SignalR connection is created, which can be hard to manage. This can be resolved when using the Azure SignalR service when deploying your app to Azure. For non-Azure cloud providers, you may have to relyon your traffic manager to get around this challenge.

Blazor WebAssembly

WASM, in simple terms, is an abstraction that enables high-level programming languages, such as C#, to run in the browser. This process is done by downloading all the required WASM-based .NET assemblies and application DLLs in the browser, so that the application can run independently in the client browser. Most major browsers nowadays, such as Google Chrome, Microsoft Edge, Mozilla Firefox, and Apple's Safari and WebKit, support WASM technology.

Blazor WASM has recently been integrated into Blazor. Under the hood, Blazor WASM uses WASM-based .NET runtimes to execute an application's .NET assemblies and DLLs. This type of application can run on a browser that supports WASM web standards with no plugins required. That said, Blazor WASM is not a new form of Silverlight.

Figure below shows you how a Blazor WASM application works under the hood.



In the preceding illustration, we can see that the Blazor WASM application doesn't depend on ASP.NET Core; the application is directly executed on the client. Client-side Blazor is running using WASM technology. By default, a Blazor WASM application runs purely on the client; however, there's an option for you to turn it into an ASP.NET-hosted app to get all the benefits of Blazor and full-stack .NET web development.

The pros of this are as follows:

- No need for you to write JavaScript to run the app.
- No server-side dependency, which means no latency or scalability issues since the app runs on the client machine.
- Enables offline support, since the app is offloaded to the client as a self-contained app. This means you can still run the application while being disconnected from the server where your application is hosted.
- Support for Progressive Web Applications (PWAs). PWAs are web applications that use modern browser APIs and capabilities to behave like native ones.

These are the cons:

- The initial loading of a page is slow, and the download size is huge because all the required dependencies need to be pulled upfront to offload your application to the client's browser. This can be optimized in the future, when caching is implemented to reduce the size of downloads and the amount of time that subsequent requests take to process.
- Since DLLs are downloaded to the client, your application code is exposed. So, you must be very careful about what you put there.
- Requires a browser that supports WASM. Note that most major browsers now support WASM.
- It's a less mature runtime as it's new.
- Debugging might be harder and limited, compared to Blazor Server.

For more information about Blazor hosting models, see https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models.

Mobile Blazor Bindings

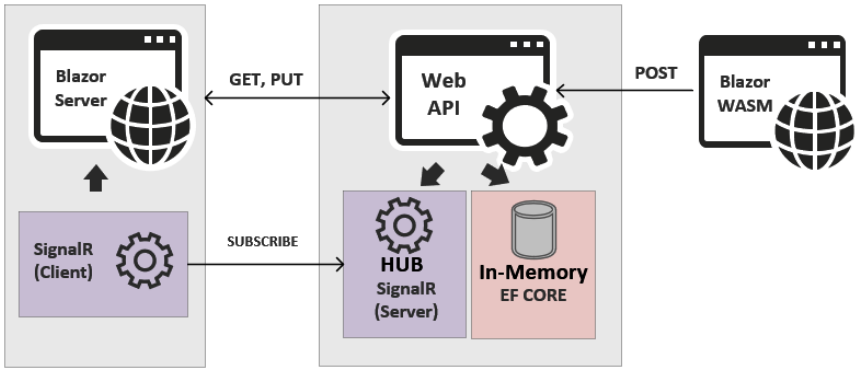
Blazor also provides a framework for building native and hybrid mobile applications for Android, iOS, Windows, and macOS, using C# and .NET. Mobile Blazor Bindings uses the same markup engine for building UI components. This means that you can use Razor syntax to define UI components and their behaviors. Under the hood, the UI components are still based on Xamarin.Forms, as it uses the same XAML-based structure to build components. What makes this framework stand out over Xamarin.Forms is that it allows you to mix in HTML, giving developers the choice to write apps using the markup they prefer. With hybrid apps, you can mix in HTML to build components just as you would build web UI components. This makes it a great stepping stone for ASP.NET developers looking to get into cross-platform native mobile application development using their existing skills. With that being said, Mobile Blazor Bindings is still in its experimental stage and there is no guarantee about anything until it is officially released.

We won't be covering Mobile Blazor Bindings development in this module. If you want tolearn more about it, you can refer to the official documentation here: https://docs.microsoft.com/en-us/mobile-blazor-bindings.

Five players, one goal

As we've learned from the previous section, Blazor is only a framework for building UIs. To make learning Blazor fun and interesting, we are going to use various technologies to build a whole web application to fulfill a goal. That goal is to build a simple data-driven web application with real-time capability using cutting-edge technologies: Blazor Server, Blazor WASM, ASP.NET Core Web API, SignalR, and Entity Framework Core.

Figure below illustrates the high-level process of how each technology connects.



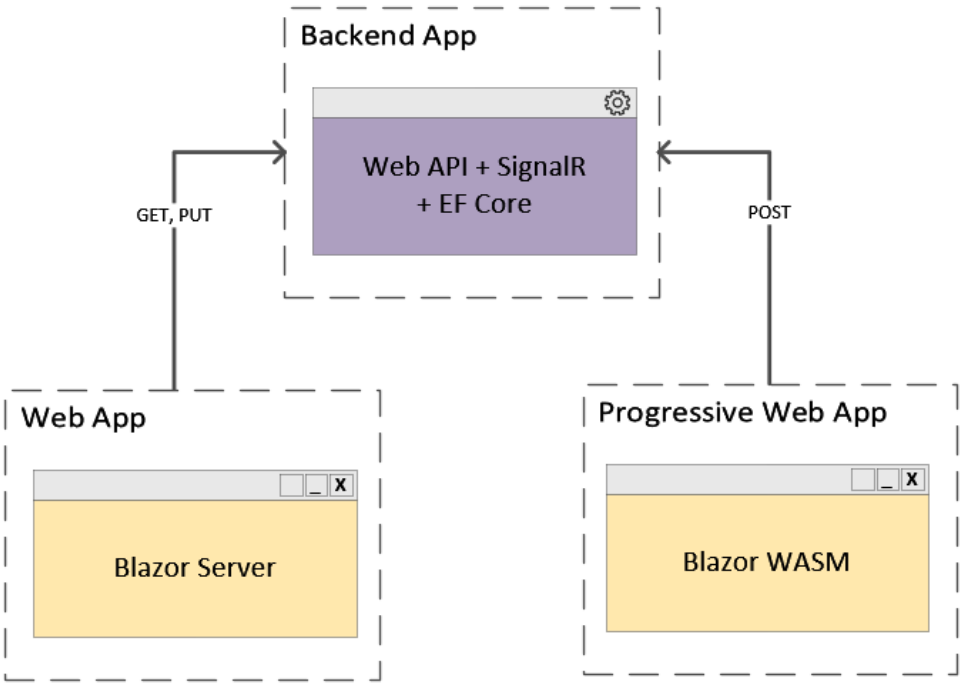
Based on the preceding diagram, we are going to need to build the following applications:

- A web app that displays and updates information on the page via API calls. This application will also implement a SignalR subscription that acts as the client to perform real-time data updates to the UI.
- A Web API app that exposes GET, PUT, and POST public-facing API endpoints. This application will also configure an in-memory data store to persist data and implement SignalR to broadcast a message to the hub where clients can subscribe and get data in real time.
- A PWA that submits a new record via an API call.

Building a tourist spot application

In order to cover real-world scenarios in a typical data-driven web application, we will build a simple tourist spot application that composes various applications to perform different tasks. You can think of this application as a wiki for tourist destinations, where users can view and edit information about places. Users can also see the top places, based on reviews, and they also see new places submitted by other similar applications in real time. By real time, we mean without the user having to refresh the page to see new data.

Figure below describes the applications needed and the high-level flow of the process for our tourist spot application example



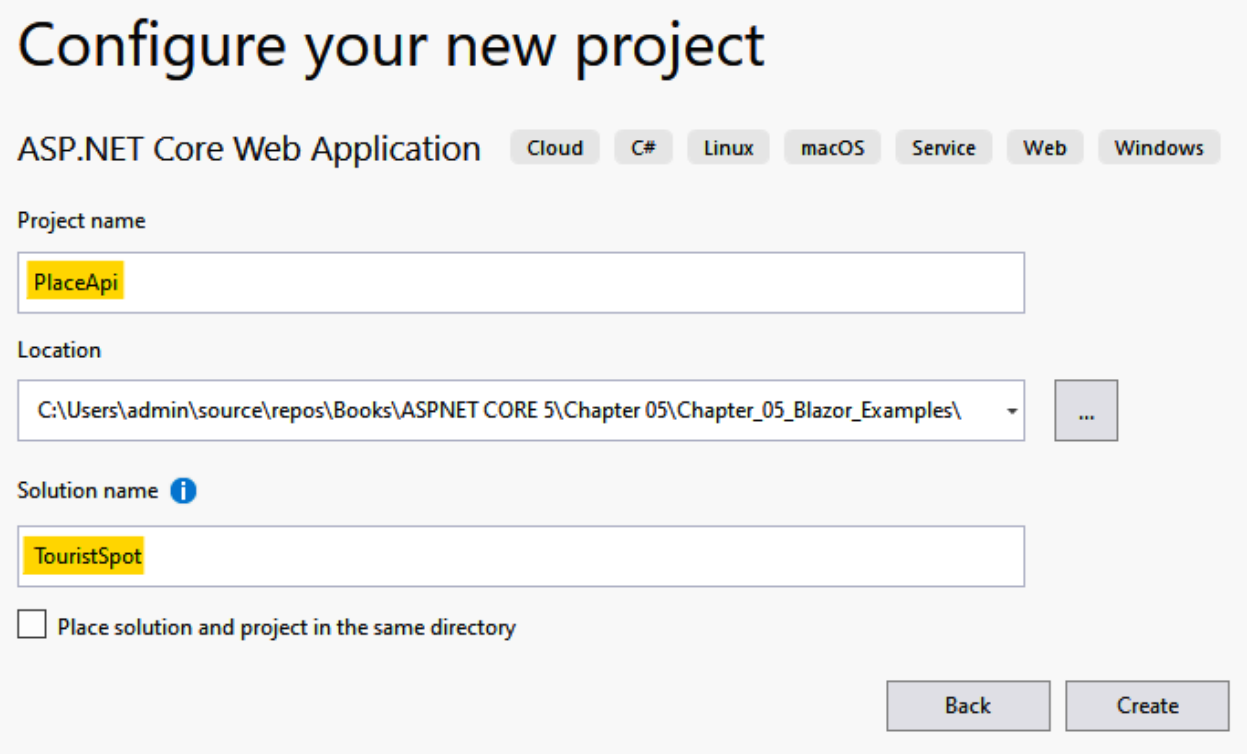
If you're ready, then let's get cracking. We'll start by building the backend application, which exposes the API endpoints to serve data so that other applications can consume it.

Creating the backend application

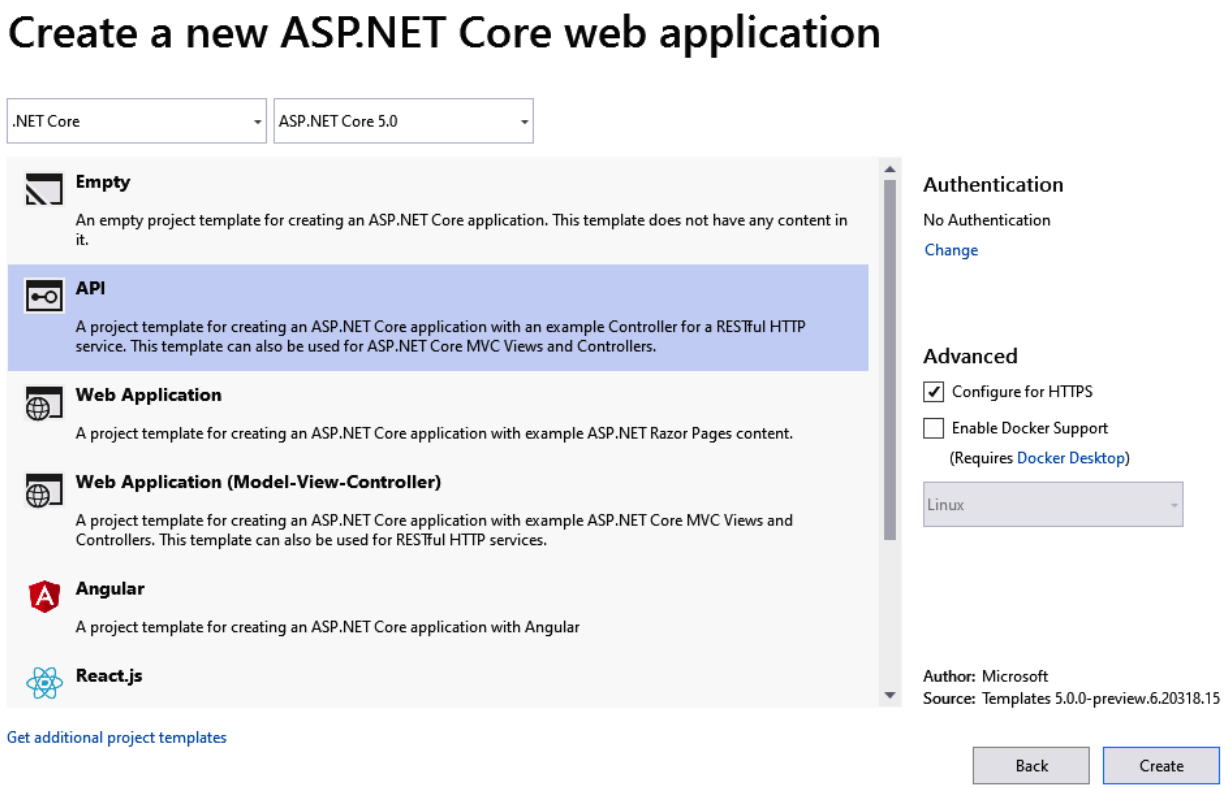
For the tourist spot application project, we are going to use ASP.NET Core Web API as our backend application.

Let's go ahead and fire up Visual Studio 2019 and then select the **Create a new project** option. On the next screen, select **ASP.NET Core Web Application** and then click **Next**.

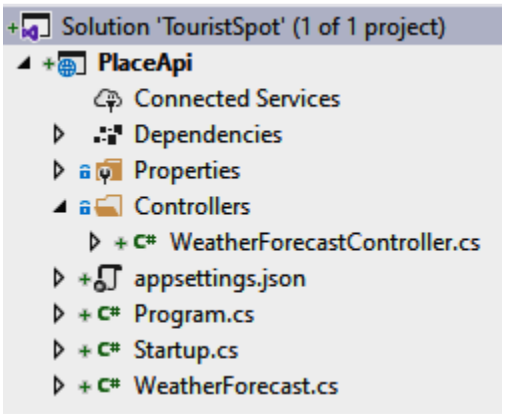
The Configure your new project dialog should appear as it does in Figure below:



This dialog allows you to configure your project and solution name, as well as the location path to where you want the project to be created. For this particular example, we'll just name the project PlaceApi and set the solution name to TouristSpot. Now, click Create and you should see the dialog shown in figure below:



This dialog allows you to choose the type of web framework that you want to create. For this project, just select API and then click Create to let Visual Studio generate the necessary files for you. The default files generated should look something like it does in figure below:



The preceding screenshot shows the default structure of an ASP.NET Core Web API application. Please note that we won't dig into the details about Web API in this module, but to give you a quick overview, Web API works the same way as the traditional ASP.NET MVC, except that it was designed for building RESTful APIs that can be consumed over HTTP. In other words, Web API doesn't have Razor View Engine and it wasn't meant to generate pages. We'll deep dive into the details of Web API in the module **APIs and Data Access**.

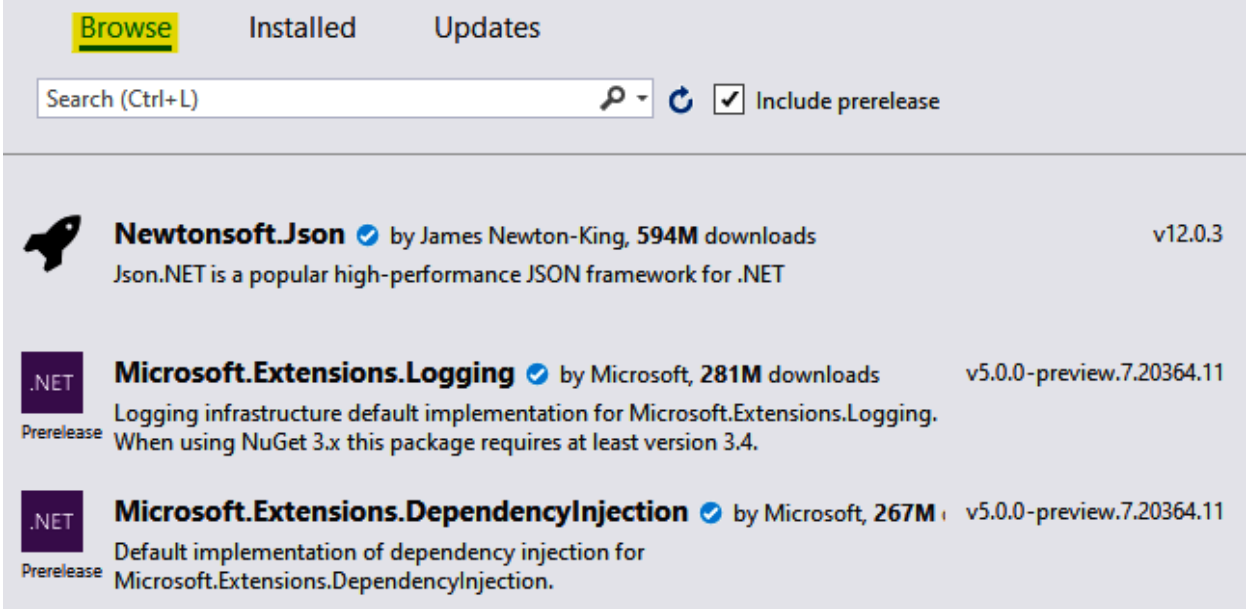
Now, let's move on to the next step.

Configuring an in-memory database

In the previous module, we learned how to use an in-memory database with Entity Framework Core. If you've made it this far, you should now be familiar with how to configure an in-memory data store. For this demonstration, we will be using the technique you're now familiar with to easily create a data-driven app, without the need to spin up a real database server to store data. Working with a real database in Entity Framework Core will be covered in module **APIs and Data Access**; for now, let's just make use of an in-memory database, for the simplicity of this exercise.

Installing Entity Framework Core

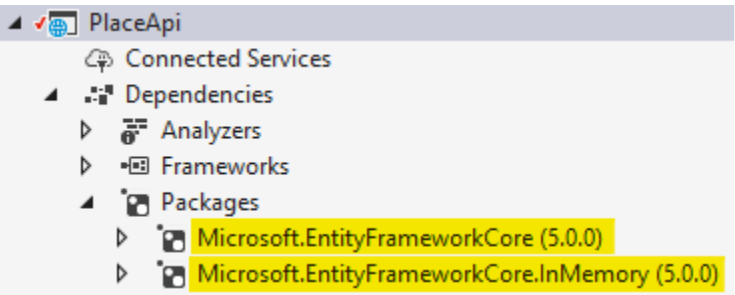
Entity Framework Core was implemented as a separate NuGet package to allow developers to easily integrate it when needed. There are many ways to integrate NuGet package dependencies in your application. We could either install it via the command line (CLI) or via the NuGet package management interface (the UI) integrated into Visual Studio. To install dependencies using the UI, simply right-click on the Dependencies folder of the project and then select the Manage NuGet Packages… option. Figure below shows you how the UI should come up.



In the Browse tab, type in the package names listed here and install them:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.InMemory

After successfully installing both packages, make sure to check your project's Dependencies folder and verify that they were added (as shown in figure below).



> **Note:**
>
> The latest official version of Microsoft.EntityFrameworkCore at the time of writing is 5.0.0. Future versions may change and could impact the sample code used in this module. So, make sure to always check for any breaking changes when deciding to upgrade to newer versions.

Now that we have Entity Framework Core in place, let's move on to the next step and configure some test data.

Implementing the data access layer

Create a new folder called Db in the project root and then create a sub-folder called Models. Right-click on the Models folder and select **Add > Class.** Name the class Places.cs, click **Add**, and then paste the following code:

```
using System;

namespace PlaceApi.Db.Models {
  public class Place {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Location { get; set; }
    public string About { get; set; }
    public int Reviews { get; set; }
    public string ImageData { get; set; }
    public DateTime LastUpdated { get; set; }
  }
}
```

The preceding code is just a plain class that houses some properties. We will use this class later to populate each property with test data.

Now, create a new class called PlaceDbContext.cs in the Db folder and copy the following code:

```
using Microsoft.EntityFrameworkCore;
using PlaceApi.Db.Models;

namespace PlaceApi.Db {
  public class PlaceDbContext : DbContext {
    public PlaceDbContext(DbContextOptions<PlaceDbContext> options)
    : base(options) { }
    public DbSet<Place> Places { get; set; }
  }
}
```

The preceding code defines a DbContext instance and a single entity that exposes a Places property (entity) as a DbSet instance. DbSet<Place> represents a collection of data in memory and is the gateway to performing database operations. For example, any changes to DbSet<Place> will be committed to the database, right after invoking the SaveChanges() method of DbContext.

Let's continue by adding another new class called PlaceDbSeeder.cs in the Db folder. Now, paste the following method into the class:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using PlaceApi.Db.Models;
using System;
using System.IO;
using System.Linq;

namespace PlaceApi.Db {
  public class PlaceDbSeeder {
    public static void Seed(IServiceProvider serviceProvider) {
      using var context = new
PlaceDbContext(serviceProvider.GetRequiredService<DbContextOptions<PlaceDbContext>>());

      if (context.Places.Any()) {
        return;
      }

      context.Places.AddRange(
        new Place {
          Id = 1,
          Name = "Coron Island",
          Location = "Palawan, Philippines",
          About = "Coron is one of the top destinations for tourists to add to their wish
list.",
          Reviews = 10,
          ImageData = GetImage("coron_island.jpg", "image/jpeg"),
          LastUpdated = DateTime.Now
        },
        new Place {
          Id = 2,
          Name = "Olsob Cebu",
          Location = "Cebu, Philippines",
          About = "Whale shark watching is the most popular tourist attraction in Cebu.",
          Reviews = 3,
          ImageData = GetImage("oslob_whalesharks.png", "image/png"),
          LastUpdated = DateTime.Now
        }
      );
      context.SaveChanges();
    }

    private static string GetImage(string fileName, string fileType) {
      var path = Path.Combine(Environment.CurrentDirectory, "Db/Images", fileName);
      var imageBytes = File.ReadAllBytes(path);
      return $"data:{fileType};base64,{Convert.ToBase64String(imageBytes)}";
    }
  }
}
```

The GetImage() method, in the preceding code, gets the image files stored within the Db/Images folder and converts the image to a byte array. It then converts the bytes to the base64 string format and returns the formatted data as an image. We are going to reference this method in the next step.

The Seed() method in the preceding code will initialize a couple of Place data sets when the application starts. This is done by adding the data into the Places entity of PlaceDbContext. You can see that we set the value of the ImageData property by calling the GetImage() method created earlier.

Now that we have implemented our seeder class, the next thing we need to do is to create a new class that will house a couple of extension methods for registering our in-memory database and using our seeder class as a middleware. Within the Db folder, go ahead and add a new class called PlaceDbServiceExtension.cs and paste in the following code:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace PlaceApi.Db {
  public static class PlaceDbServiceExtension {
    public static void AddInMemoryDatabaseService(this IServiceCollection services,
string dbName)
          => services.AddDbContext<PlaceDbContext>(options =>
options.UseInMemoryDatabase(dbName));

    public static void InitializeSeededData(this IApplicationBuilder app) {
      using var serviceScope =
app.ApplicationServices.GetRequiredService<IServiceScopeFactory>().CreateScope();
      var service = serviceScope.ServiceProvider;
      PlaceDbSeeder.Seed(service);
    }
  }
}
```

The preceding code defines two main static methods. AddInMemoryDatabaseService() is an IServiceCollection extension method that registers PlaceDbContext as a service in the **dependency injection (DI)** container. Notice that we are configuring the UseInMemoryDatabase () extension method as a parameter to the AddDbContext() method call. This tells the framework to spin up an in-memory database with a given database name. The InitializeSeededData() extension method is responsible for generating test data when the application runs. It uses the GetRequiredService() method of the ApplicationServices class to reference the service provider used to resolved dependencies from the scope. It then calls the PlaceDbSeeder.Seed() method that we created earlier and passes the service provider to initialize the test data.

The this keyword, before the object type in each method's parameters, denotes that a method is an extension method. Extension methods enable you to add a method to an existing type. For this particular example, we are adding the AddInMemoryDatabaseService() method to an object of type IServiceCollection and adding the InitializeSeededData() method to an object of type IApplicationBuilder. For more information about extension methods, see https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods.

At this point, we now have a DbContext instance that enables us to access our Places DbSet, a helper class that will generate some data, and a couple of extension methods to register our in-memory service. What we need to do next is to wire them into Startup.cs to populate our data when the application starts.

Modifying the Startup class

Let's update the ConfigureServices() method of the Startup class to the following code:

```
public void ConfigureServices(IServiceCollection services) {
  services.AddInMemoryDatabaseService("PlacedDb");
  services.AddControllers();
}
```

In the preceding code, we've invoked the AddInMemoryDatabaseService() extension method that we created earlier. Again, this process registers PlaceDbContext in IServiceCollection and defines an in-memory database called PlacedDb. Registering DbContext as a service into the DI container enables us to reference an instance of this service in any class within the application via DI.

Now, the final step that we need to do is to call the InitializeSeededData() extension method in the Configure() method as follows:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
  app.InitializeSeededData();

  //removed other middlewares for brevity

}
```

At this point, our test data should now be loaded into our in-memory database when the application starts and should be ready for use in our application.

Implementing real-time functionality with SignalR

Adding real-time functionality to any ASP.NET Core server application is pretty easy nowadays, because SignalR is fully integrated into the framework. This means that there's no need to download or reference a separate NuGet package just to be able to implement real-time capability.

ASP.NET SignalR is a technology that offers a clean set of APIs that enables real-time behavior for your web application, where the server pushes data to the client, as opposed to the traditional way of having the client continuously pull data from the server to get updated.

To start working with ASP.NET Core SignalR, we need to create a hub first. Hub is a special class in SignalR that enables us to call methods on connected clients from the server. The server in this example is our Web API, for which we will define a method for clients to invoke. The client in this example is the Blazor Server application.

Let's create a new class called PlaceApiHub under the root of the application and then paste in the following code:

```
using Microsoft.AspNetCore.SignalR;
namespace PlaceApi {
  public class PlaceApiHub : Hub {
  }
}
```

The preceding code is just a class that inherits from the Hub class. We'll leave the Hub class empty, as we are not invoking any methods from the client. Instead, the API will send the events over the hub.

Next, we are going to register SignalR and the ResponseCompression service in the DI container. Add the following code within the ConfigureServices() method of the Startup class:

```
public void ConfigureServices(IServiceCollection services) {
  services.AddSignalR();
  services.AddResponseCompression(opts =>
  {
    opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.
    Concat(
    new[] { "application / octet - stream" });
  });

  // Removed other services for brevity

}
```

Next, we need to add the ResponseCompression middleware in the pipeline and map our Hub. Add the following code within the Configure() method:

```
public void Configure(IApplicationBuilder app,IWebHostEnvironment env) {

  // Removed other code for brevity

  app.UseResponseCompression();
  app.UseEndpoints(endpoints =>
  {
    endpoints.MapControllers();
    endpoints.MapHub<PlaceApiHub>("/PlaceApiHub");
  });
}
```

The preceding code defines a route for the SignalR hub by mapping the PlaceApiHub class. This enables the client application to connect to the hub and listen to events being sent from the server.

That was simple. We will implement sending an event in the next section when creating the Web API endpoints.

Creating the API endpoints

Now that our in-memory database is all set and we've configured SignalR for our realtime capability, it's time for us to create the API controller and expose some endpoints to serve data to the client. In this particular example, we are going to need the following API endpoints to handle fetching, creating, and updating data:

- GET: api/places
- POST: api/places
- PUT: api/places

Go ahead and right-click on the Controllers folder and then select **Add > Controller > API Controller Empty**, and then click **Add**.

Name the class PlacesController.cs and then click **Add**. Now, replace the default generated code so that what you have looks like the following code:

```
using Microsoft.AspNetCore.Mvc;
using PlaceApi.Db;
using PlaceApi.Db.Models;
using System;
using System.Linq;

namespace PlaceApi.Controllers {
  [ApiController]
  [Route("api/[controller]")]
  public class PlacesController : ControllerBase {
    private readonly PlaceDbContext _dbContext;
    private readonly IHubContext<PlaceApiHub> _hubContext;
    public PlacesController(PlaceDbContext dbContext,
    IHubContext<PlaceApiHub> hubContext) {
      _dbContext = dbContext;
      _hubContext = hubContext;
    }
    [HttpGet]
    public IActionResult GetTopPlaces() {
      var places = _dbContext.Places.OrderByDescending(o
      => o.Reviews).Take(10);
      return Ok(places);
    }
  }
}
```

The preceding code shows the typical structure of an API Controller class. An API should implement the ControllerBase abstract class to utilize the existing functionalities built into the framework for building RESTful APIs. We'll talk in more depth about APIs in the next module. For the time being, let's just walk through what we did in the preceding code. The first two lines of the PlacesController class define private and read-only fields for PlaceDbContext and IHubContext<PlaceApiHub>. The next line defines the class constructor and injects PlaceDbContext and IHubContext<PlaceApiHub> as dependencies to the class. In this case, any methods within the PlacesController class will be able to access the instance of PlaceDbContext and IHubContext, allowing us to invoke all its available methods and properties.

Currently, we have only defined one method in our PlaceController. The GetTopPlaces() method is responsible for returning the top 10 rows of data from our in-memory datastore. We've used the LINQ OrderByDescending() and Take() extension methods, of the Enumerable type, to get the top rows based on the Reviews value. You can see that the method has been decorated with the [HttpGet] attribute, which signifies that the method can only be invoked by an HTTP GET request.

Now, let's add another method for handling new record creation. Append the following code within the class:

```
[HttpPost]
public IActionResult CreateNewPlace([FromBody] Place place) {
  var newId = _dbContext.Places.Select(x => x.Id).Max() + 1;
  place.Id = newId;
  place.LastUpdated = DateTime.Now;
  _dbContext.Places.Add(place);
  int rowsAffected = _dbContext.SaveChanges();
```

```
  if (rowsAffected > 0) {
    _hubContext.Clients.All.SendAsync("NotifyNewPlaceAdded",
    place.Id, place.Name);
  }
  return Ok("New place has been added successfully.");
}
```

The preceding code is responsible for creating a new Place record in our in-memory database and at the same time broadcasting an event to the hub. In this case, we are invoking the Clients.All.SendAsync() method of the Hub class and passing place.Id and place.Name to the NotifyNewPlaceAdded event. Note that you can also pass an object to the SendAsync() method instead of passing individual parameters, just like what we did in this example. You can see that the CreateNewPlace() method has been decorated with the [HttpPost] attribute, which signifies that the method can be invoked only by HTTP POST requests. Keep in mind that we are generating Id manually by incrementing the existing maximum ID from our data store. In a real application using a real database, you may not need to do this as you can let the database auto-generate Id for you.

Let's create the last endpoint that we need for our application. Add the following code block to the class:

```
[HttpPut]
public IActionResult UpdatePlace([FromBody] Place place) {
  var placeUpdate = _dbContext.Places.Find(place.Id);
  if (placeUpdate == null) {
    return NotFound();
  }
  placeUpdate.Name = place.Name;
  placeUpdate.Location = place.Location;
  placeUpdate.About = place.About;
  placeUpdate.Reviews = place.Reviews;
  placeUpdate.ImageDataUrl = place.ImageDataUrl;
  placeUpdate.LastUpdated = DateTime.Now;
  _dbContext.Update(placeUpdate);
  _dbContext.SaveChanges();
  return Ok("Place has been updated successfully.");
}
```

The preceding code is responsible for updating an existing Place record in our in-memory database. The UpdatePlace() method takes a Place object as a parameter. It first checks whether the record exists based on the ID. If the record isn't in the database, we return a NotFound() response. Otherwise, we update the record in the database and then return an OK() response with a message. Notice that the method in this case is decorated with the [HttpPut] attribute, which denotes that this method can only be invoked by an HTTP PUT request.

Enabling CORS

Now that we have our API ready, the next step that we are going to take is to enable **Cross-Origin Resource Sharing (CORS)**. We need to configure this so that other client applications that are hosted in different domains/ports can access the API endpoints. To enable CORS in ASP.NET Core Web API, add the following code in the ConfigureServices() method of the Startup class:

```
      services.AddCors(options => {
        options.AddPolicy("AllowAll",
        builder => {
          builder.AllowAnyOrigin()
                         .AllowAnyHeader()
                         .AllowAnyMethod();
        });
      });
```

The preceding code adds a CORS policy to allow any client applications access to our API. In this case, we've set up a CORS policy with the AllowAnyOrigin(), AllowAnyHeader(), and AllowAnyMethod() configurations. Bear in mind, though, that you should consider setting the allowable origins, methods, headers, and credentials before exposing your APIs publicly in real-world applications. For details about CORS, see the official documentation here: https://docs.microsoft.com/en-us/aspnet/core/security/cors.

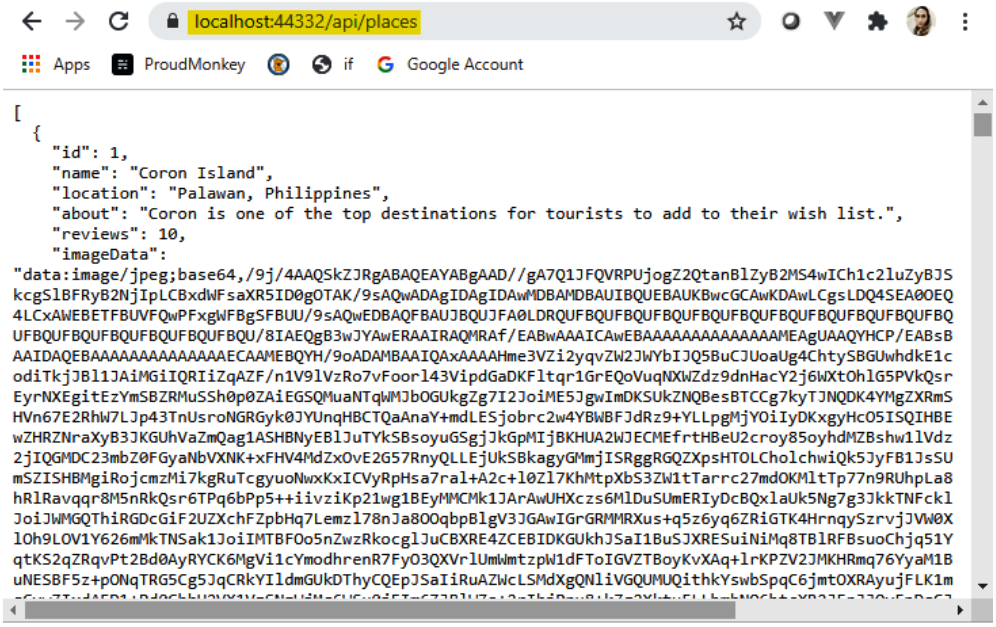
Now, add the following code in the Configure() method after the UseRouting() middleware:

```
app.UseCors("AllowAll");
```

That's it.

Testing the endpoints

Now that we have implemented the required API endpoints for our application, let's do a quick test to ensure that our API endpoints are working. Press Ctrl + F5 to launch the application in the browser and then navigate to the https://localhost:44332/api/places endpoint. You should be presented with the output shown in figure below:



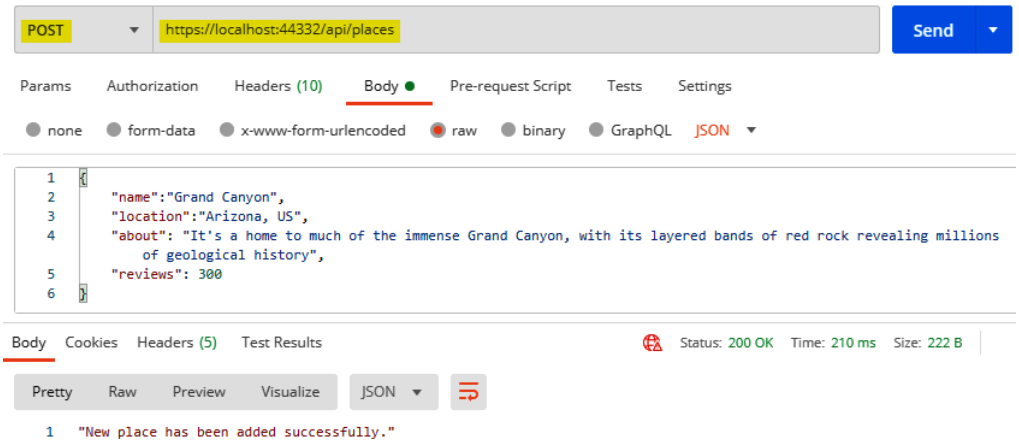
The preceding screenshot shows the result of our GetTopPlaces() GET endpoint in JSON format. Keep note of the localhost port value on which our API is currently running, as we are going to use the exact same port number when invoking the endpoints in our Blazor applications. In this case, our API is running on port 44332 locally in IIS Express. You can see how this was defined by looking at the launchSettings.json file within the Properties folder, as shown in the following code:

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:60766",
      "sslPort": 44332
    }
  },
  //Removed other configuration for brevity
}
```

The preceding code shows the profile configurations when running the application locally, including IIS Express. You can update the configuration and add new profiles to run the application on different environments. In this example, we'll just leave the default configuration as is for simplicity's sake. The default IIS Express configuration sets the applicationUrl port to 60766 when running in http and sets the port to 44332 when running in https. By default, the application uses the UseHttpsRedirection() middleware in the Configure() method of the Startup class. This means that when you try to use the http://localhost:60766 URL, the application will automatically redirect you to a secured port, which in this case is port 44332.

Using the browser only allows us to test HTTP GET endpoints. To test the remaining endpoints, such as POST and PUT, you may have to install a browser app extension. In Chrome, you can install the Advanced REST client extension. You can also download Postman to test out the API endpoints that we created earlier. Postman is a really handy tool for testing APIs without having to create a UI, and it's absolutely free. You can get it here: https://www.getpostman.com/.

Figure below shows you a sample screenshot of the API tested in Postman.



At this point, we have working API endpoints that we can use to present data on our page. Learning the basics of creating a Web API is very important for the overall implementation of our project.

Summary

In this module, we've learned about the concepts behind the different types of Blazor hosting model. We've identified the goal of the application that we are going to build while learning about Blazor, and we've identified the various technologies needed to reach it. We started creating the backend application using the ASP.NET Core API, and we saw how we can easily configure test data, without having to set up a real database, using Entity Framework Core's in-memory provider feature. This enables us to easily spin up data-driven applications when doing proof-of-concept (POC) projects. We also learned how to create simple REST Web APIs to serve data and learned how to configure SignalR to perform real-time updates. Understanding the basic concepts of the technologies and frameworks used in this module is very important to successfully working with real applications.

We've learned that both of the Blazor models we saw in this module are great choices, despite their cons. The programming behind Blazor allows C# developers, who want to avoid JavaScript hurdles, to build SPAs without having to learn a new programming language. Despite being fairly new, it's clear that Blazor is going to be an incredible hit and a great contender among other well-known SPA frameworks, such as Angular, React, and Vue, and that's because of how WASM essentially supersedes JavaScript. Sure, JavaScript and its frameworks aren't going anywhere, but being able to use an existing C# skillset to build a web application that produces the same output as a JavaScript web application is a great advantage, in terms of avoiding having to learn a new programming language just to build web UIs. On top of that, we've learned that Blazor isn't limited to web applications only; Mobile Blazor Bindings is in the works to provide a framework for developers to write cross-platform native mobile applications.

In the next module, we are going to continue exploring Blazor and build the remaining pieces to complete our tourist spot application.