Exploring the Blazor Web Framework

In the previous module, we learned what Blazor is all about and also learned about the different hosting models that the framework offers. We started building the backend application using the ASP.NET Core web API, EF Core, and SignalR. In this module, we will build the remaining pieces to complete our goal.

Here is a list of the main topics that will be covered in this module:

- Learning about server-side and client-side Blazor
- Learning how to create Razor components
- Learning the basics of routing, state management, and data bindings
- Learning how to interact with the backend application to consume and pass data
- Building a tourist spot application using the two Blazor hosting models

By the end of this module, you will have learned how to build a tourist spot application to learn Blazor in conjunction with various technologies with the aid of hands-on practical examples.

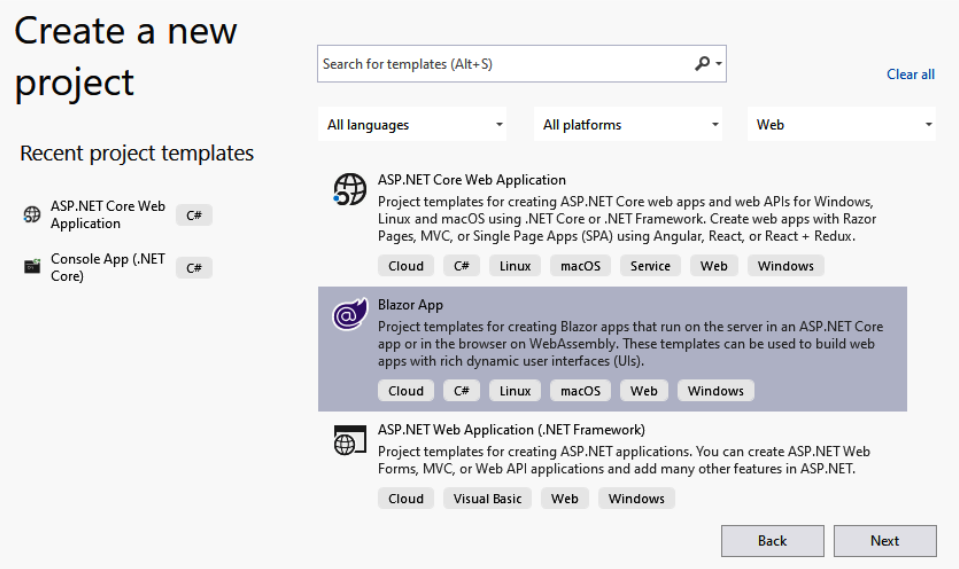
Technical requirements

This module follows on from the previous module, so before diving into this module, make sure that you've read Module 5, Getting Started with Blazor, and understand the goal of what we are going to achieve for building a sample application. It's also recommended to review Module 4, Razor View Engine, because Blazor uses the same markup engine for generating pages. Although not mandatory, a basic knowledge of HTML and CSS will be beneficial in helping you to easily understand how the page is constructed.

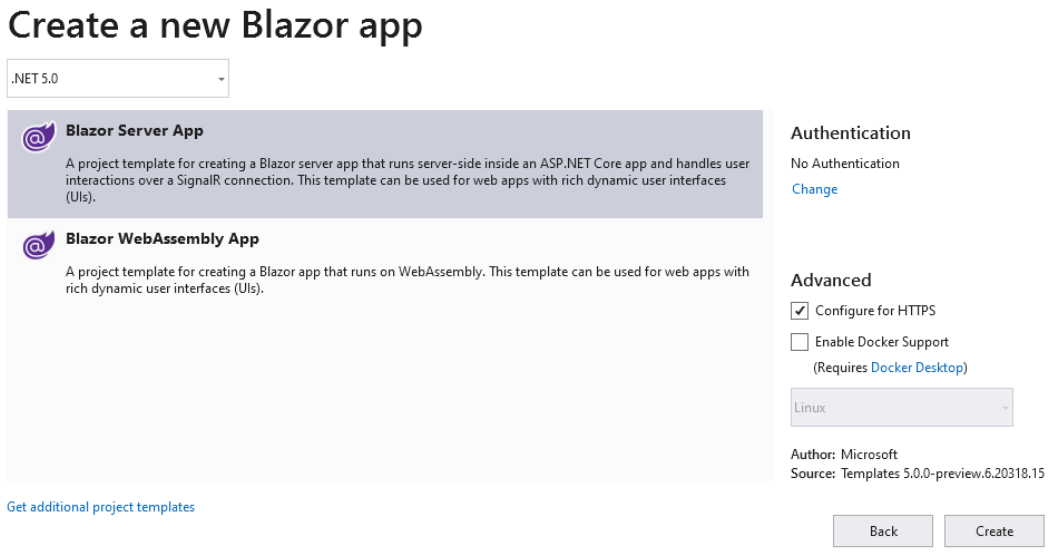
Creating the Blazor Server project

In this project, we will build the frontend web application for displaying the data from the web API.

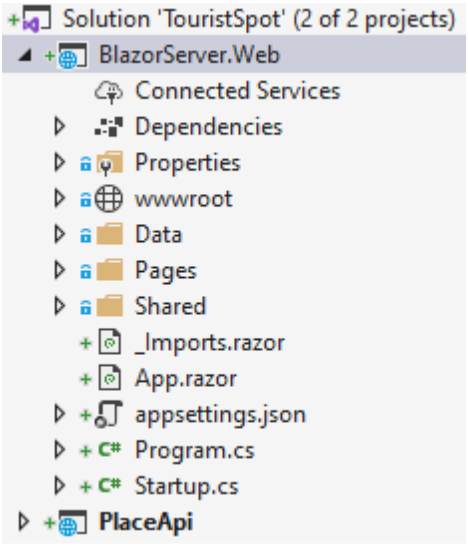
Let's go ahead and add a new Blazor Server project within the existing project solution. In the Visual Studio menu, select **File | New | Project**. Alternatively, you can also right-click on the solution to add a new project. In the **Create a new project** dialog field, select Blazor App, as shown in the following screenshot:



Click **Next**. In the next screen, you can configure the name and location path for your project. In this example, we will just name the project BlazorServer.Web. Click **Create** and you should be presented with the following dialog:



Select the **Blazor Server App** template, leave the default configuration as is, and then click **Create**. Visual Studio should scaffold the necessary files needed to build the Blazor Server app, as shown in the following screenshot:



If you've read previous module, **Razor View Engine**, you'll notice that the Blazor Server project structure is very similar to Razor Pages, except for the following:

- It uses the .razor file extension instead of .cshtml, the reason being that the Blazor application is mainly based on components. The .razor files are Razor components that enable you to build the UI using HTML and C#. It's basically the same as building UIs in a .cshtml file. In Blazor, components are pages themselves, or they could be a page with child components. Razor components can also be used in MVC or Razor Pages as they all use the same markup language, called **Razor View Engine**.

- Blazor applications contain an App.razor component. Just like any other SPA web framework, Blazor uses a main component to load the application UI. The App.razor component serves as the master component for the application and enables you to configure the routes for your components. Here is the default implementation of the App.razor file:

```
<Router AppAssembly="@typeof(Program).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
  </Found>
  <NotFound>
    <LayoutView Layout="@typeof(MainLayout)">
      <p>Sorry, there's nothing at this address.</p>
    </LayoutView>
  </NotFound>
</Router>
```

The preceding code defines a Router component and configures a default layout to be rendered in the browser when the application starts. In this case, the default layout will render the MainLayout.razor component. For more information about Blazor routing, refer to the following link: https://docs.microsoft.com/en-us/aspnet/core/blazor/fundamentals/routing.

The Blazor Server project also contains a Host.cshtml file that serves as the main entrypoint for the application. In a typical client-based SPA framework, the _Host.cshtml file represents the Index.html file, where the main App component is being referenced and bootstrapped. In this file, you can see that the App.razor component is being called within the <body> section of the HTML document, as shown in the following code block:

```
<body>
    <app>
        <component type="typeof(App)" render-mode="ServerPrerendered" />
    </app>
    @*Removed for brevity*@
</body>
```

The preceding code renders the App.razor component with ServerPrerendered as the default rendering mode. This mode tells the framework to render the component in static HTML first and then bootstrap the app when the browser starts.

Creating the model

The first thing that we are going to do in this project is to create a class that will contain some properties that match with what we expect from the web API response. Let's go ahead and create a new class called Place.cs under the Data folder. The class definition should look like the following:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace BlazorServer.Web.Data {
  public class Place {
    public int Id { get; set; }
    [Required] public string Name { get; set; }
    [Required] public string Location { get; set; }
    [Required] public string About { get; set; }
```

```
    public int Reviews { get; set; }
    public string ImageData { get; set; }
    public DateTime LastUpdated { get; set; }
  }

}
```

As you will observe, the preceding code is identical to the Place class that we've created in the web API project, except that we've used data annotation to decorate a few properties with the [Required] attribute. We are going to populate these properties with the result from the web API and use it in the Blazor components to display information. The required properties ensure that these fields will not be empty when updating the form. We are going to see how this is done later in this module.

Implementing a service for web API communication

Now that we have our Model in place, let's implement a service for invoking a couple of web API endpoints to fetch and update data. First, install the Microsoft.

AspNetCore.SignalR.Client NuGet package in order for us to be able to connect to Hub and listen to an event.

After installing the SignalR client package, create a new class called PlaceService.cs under the Data folder and copy the following code:

```
public class PlaceService {
  private readonly HttpClient _httpClient;
  private HubConnection _hubConnection;
  public PlaceService(HttpClient httpClient) {
    _httpClient = httpClient;
  }
  public string NewPlaceName { get; set; }
  public int NewPlaceId { get; set; }
  public event Action OnChange;

}
```

The preceding code defines a couple of private fields for HttpClient and HubConnection. We'll use these fields later to invoke methods. The PlaceService constructor takes an HttpClient object as a dependency to the class and assigns the _httpClient field. At runtime, the HttpClient object will be resolved by the DI container.

The NewPlaceName and NewPlaceId properties will be populated once the application receives the newly added record from Hub. The OnChange event is a special type of delegate in C# that allows you to subscribe to it when a certain action raises the event.

Now, let's implement the SignalR configuration for subscribing to Hub. Go ahead and append the following code within the PlaceService class:

```
public async Task InitializeSignalR() {
  _hubConnection = new HubConnectionBuilder()
  .WithUrl($"{ _httpClient.BaseAddress.AbsoluteUri}
  PlaceApiHub")
.Build();
```

```
_hubConnection.On<int, string>("NotifyNewPlaceAdded",
(placeId, placeName) => {
  UpdateUIState(placeId, placeName);
});
await _hubConnection.StartAsync();
}
public void UpdateUIState(int placeId, string placeName) {
  NewPlaceId = placeId;
  NewPlaceName = placeName;
  NotifyStateChanged();
}

private void NotifyStateChanged() => OnChange?.Invoke();
```

The InitializeSignalR() method is responsible for creating a connection to Hub by setting the HubConnection.WithUrl() method. We've used the value of _httpClient.BaseAddress.AbsoluteUri to avoid hardcoding the base URL of the web API endpoint. We'll configure the base URL later when we register the PlaceService class with the typed instance of HttpClient. The value of the WithUrl parameter is actually equivalent to https://localhost:44332/PlaceApiHub. If you recall, the /PlaceApiHub URL segment is the Hub route that we configured earlier when we created the API project. In the next line, we've used the On method of HubConnection to listen to the NotifyNewPlaceAdded event. When a server broadcasts data to this event, UpdateUIState() will be invoked, which sets the NewPlaceId and NewPlaceName properties and then ultimately invokes the NotifyStateChanged() method to trigger the OnChange event.

Next, let's implement the methods for connecting to the web API endpoints. Append the following code:

```
public async Task<IEnumerable<Place>> GetPlacesAsync() {
  var response = await _httpClient.GetAsync("/api/places");
  response.EnsureSuccessStatusCode();
  var json = await response.Content.ReadAsStringAsync();
  var jsonOption = new JsonSerializerOptions {
    PropertyNameCaseInsensitive = true
  };
  var data = JsonSerializer.
  Deserialize<IEnumerable<Place>>(json, jsonOption);
  return data;
}
public async Task UpdatePlaceAsync(Place place) {
  var response = await _httpClient.PutAsJsonAsync("/api/places", place);
  response.EnsureSuccessStatusCode();

}
```

The GetPlacesAsync() method calls the /api/places HTTP GET endpoint to fetch data. Notice that we are passing JsonSerializerOptions with PropertyNameCaseInsensitive set to true when deserializing the result to a Place model. This is to correctly map the properties in the Place model because the default JSON response from the API call is in camel case format. Without setting this option, you will not be able to populate the Place model properties with data because the format is in Pascal case.

The UpdatePlaceAsync() method is very straightforward. It takes a Place model as a parameter and then calls the API to save the changes to the database. The EnsureSuccessStatusCode() method call will throw an exception if the HTTP response was unsuccessful.

Next, add the following entry to the appSettings.json file:

```
"PlaceApiBaseUrl": "https://localhost:44332"
```

Defining common configuration values within appSettings.json is a good practice to avoid hardcoding any static values in your C# code.

Note: The ASP.NET Core project template will generate both appSettings.json and appSettings.Development.json files. If you are deploying your application in different environments, you can take advantage of the configuration and create specific configuration files targeting each environment. For local development, you can put all your local configuration values in the appSettings.Development.json file and the common configurations in the appSettings.json file. At runtime, and depending on which environment your application is running, the framework will automatically override whatever values you configured in the appSettings.json file with the values you configured in your environment-specific configuration file. For more information, check out the Further reading section of this module.

The final step for this to work is to register PlaceService in IServiceCollection. Go ahead and add the following code to the ConfigureServices() method of the Startup class:

```
      services.AddHttpClient<PlaceService>(client => {
        client.BaseAddress = new Uri(Configuration["PlaceApiBaseUrl"]);

      });
```

The preceding code registers a typed instance of HttpClientFactory in the DI container. Notice that the BaseAddress value is being pulled from appSettings.json via the Configuration object.

Implementing the application state

Blazor applications are made up of components and, in order to effectively communicate between the changes that are happening in dependent components, we need to implement some sort of state container to keep track of the changes. Create a new class called AppState.cs under the Data folder and copy the following code:

```
using System;
namespace BlazorServer.Web.Data {
  public class AppState {
    public Place Place { get; private set; }
    public event Action OnChange;
    public void SetAppState(Place place) {
      Place = place;
      NotifyStateChanged();
    }
    private void NotifyStateChanged() => OnChange?.Invoke();
  }
}
```

The preceding code consist of a property, an event, and methods. The Place property is used to hold the current Place model that has been modified. The OnChange event is used to trigger some logic when the application state has changed. The SetAppState() method handles the current state of the component.

This is where we set the properties to keep track of the change and call the NotifyStateChanged() method to invoke the OnChanged event.

The next step is to register the AppState class as a service so that we can inject it into any component. Go ahead and add the following code to the ConfigureServices() method of the Startup class:

```
public void ConfigureServices(IServiceCollection services) {
  services.AddScoped<AppState>();

  //removed other services for brevity

}
```

The preceding code registers the AppState class as a scoped service in the DI container because we wanted an instance of this service to be created for each web request.

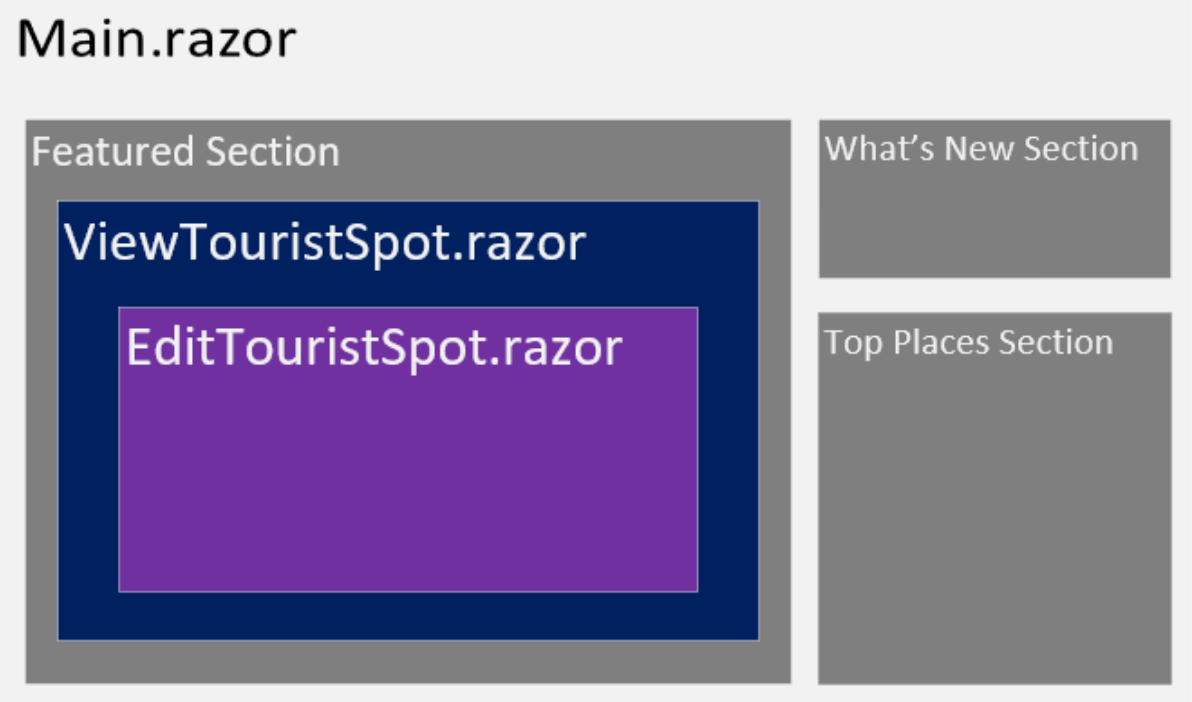
At this point, we now have what we need to build the UIs: a service to consume data and a service to keep track of the component state. Now, let's move on to the next step and start building the UIs for the application.

Creating Razor components

We are going to split the page implementation into components. With that said, we are now going to create the following Razor components:

- Main.razor
- ViewTouristSpot.razor
- EditTouristSpot.razor

The following diagram shows a graphical representation of how we are going to lay out our web page:



The Main.razor component will contain three main sections for displaying various data representations. These sections are just elements in the component. Under Featured Section, we will render the ViewTouristSpot.razor component as a child to the Main.razor component. ViewTouristSpot.razor will contain EditTouristSpot.razor as a child component.

Now that you already have an idea of how the page is going to look, let's start building the required components.

Composing the EditTouristSpot component

Let's start creating the inner child component. Create a new folder called **Spots** under the Pages folder. Right-click on the Spots folder and then select **Add | Razor Component**. A window dialog should appear for you to name the component. In this example, just set the name to EditTouristSpot.razor and then click **Add**. Delete the generated code because we are going to replace it with our code implementation.

A Razor component is typically divided into three main parts:

- The first part is for declaring class and service references that are required in order for us to invoke methods and members.
- The second part is for constructing the actual UI using Razor syntax by combining HTML, CSS, and C#.
- The third part is for handling any user interaction logic contained within the @code{} block.

Here's a quick summary of a typical component composition:

```
@*Routing, Namespace, Class and Service references goes here*@
@*HTML generation and UI construction goes here*@
@*UI logic and C# code block goes here*@
```

Let's start integrating the first part. Add the following code:

```
@using BlazorServer.Web.Data
@inject PlaceService _placeService
@inject AppState _appState
```

The preceding code uses the @using and @inject Razor directives to reference a server-side class and service within the Blazor component. This enables us to access members and methods that are available. For this specific example, declaring the @using BlazorServer.Web.Data reference allow us to access the Place class defined within that namespace. The same goes for the @inject directive. When injecting the AppState and PlaceService services, it allows us to access all the methods that they expose within the markup.

Now, let's integrate the second part. Append the following code:

```
@if (IsReadOnlyMode) {
<ViewTouristSpot Place="Place" /> } else {
<EditForm Model="@Place" OnValidSubmit="HandleValidSubmit">
  <div class="card">
    <div class="card-body">
      <DataAnnotationsValidator />
      <ValidationSummary />
```

```
      Name:
      <InputText class="form-control"
                 @bind-Value="Place.Name" />
      Location:
      <InputText class="form-control"
                 @bind-Value="Place.Location" />
      About:
      <InputTextArea class="form-control"
                     @bind-Value="Place.About" />

      <br />
      <button type="submit" class="btn btn-outline-primary">Save</button>
      <button type="button" class="btn btn-outline-primary"
@onclick="UndoChanges">Cancel</button>
    </div>
  </div>

</EditForm>}
```

The preceding code is referred to as a Razor code block. Razor code blocks normally start with the @ symbol and are enclosed by curly braces, {}. The if-else statement determines which HTML block to render in the browser based on the IsReadOnlyMode Boolean property defined within the @code section. By default, it's set to false, so the HTML block within the else part gets evaluated and displays the edit form. Otherwise, it renders the ViewTouristSpot.razor component to turn the display back into a readonly state.

In the read-only state, we've passed the Place object as a parameter to the ViewTouristSpot component so it can display the data without re-invoking the API. Keep in mind that the ViewTouristSpot component doesn't yet exist and we are going to create it in the next section. In the edit state, we've used the EditForm component to take advantage of its built-in features and form validations. The EditForm component takes a model to be validated. In this case, we've passed the Place object as the model and wired up the HandleValidSubmit() method to the OnValidSubmit event handler. We have also used various built-in components, such as DataAnnotationsValidator, ValidationSummary, InputText, and InputTextArea to handle input validations and model property bindings. In this example, we are using two-way data binding to bind the Place properties to input elements using the @bind-Value attribute. The EditForm component will render as an HTML <form> element in the browser and submit all form values when an HTML <input> of type="submit" is clicked. When the Save button is clicked, this triggers the DataAnnotationsValidator component and checks whether all validations are passed. If you recall, in the Creating the model section of this module, we only validated the Name, Location, and About properties to be required, and the HandleValidSubmit() method won't be triggered if any of those properties are left empty.

The form uses Boostrap 4 CSS classes to define the look and feel of the component. **Bootstrap** is part of the default template when creating any ASP.NET Core web frameworks and you can see that the CSS file sits under the wwwroot/css/bootstrap folder.

Now, let's integrate the last part of this component. Append the following code:

```
@code {
  [Parameter] public Place Place { get; set; }
  private Place PlaceCopy { get; set; }
  bool IsReadOnlyMode { get; set; } = false;
}
```

The preceding code is referred to as a **C# code block**. The @code directive is unique to .razor files, and allows you to add C# methods, properties, and fields to a component. You can think of the code block as a code-behind file (cshtml.cs) in Razor Pages or a Controller class in MVC, where you can implement C# code logic based on UI interactions.

The Place property is decorated with the [Parameter] attribute with a public access modifier to allow the parent component to set a value to this property. The PlaceCopy property is a holder property that contains the original values being passed from the parent component. In this case, the parent component is ViewTouristSpot.razor. The IsReadOnlyMode property is a Boolean flag used to determine which HTML block to render.

Let's continue by implementing the methods that are needed for this component. Append the following code within the @code{} block:

```
  protected override void OnInitialized() {

    PlaceCopy = new Place {
      Id = Place.Id,
      Name = Place.Name,
      Location = Place.Location,
      About = Place.About,
      Reviews = Place.Reviews,
      ImageData = Place.ImageData,
      LastUpdated = Place.LastUpdated
    };
  }
```

The OnInitialized() method is part of the Blazor framework, which allow us to override it to perform certain operations. This method is triggered during component initialization and is a perfect place to configure object initialization and assignments. As you will notice, this is where we assign the property values from the original Place model to a new Place object called PlaceCopy. The main reason why we keep the original state of the Place object is because we wanted to reset the data to its default state when cancelling the edit. We could have just set the IsReadOnlyMode flag to true for the cancel action. However, doing this alone would not reset the values to the original state when switching back to the read-only state. The reason for this is that we were using two-way data binding for our Place model, and any property changes made to the form will be kept.

The process of two-way data binding works like this:

- The input elements in the UI automatically reflect the changes when properties in the Place model get updated from the server.
- When UI elements get updated, the changes get propagated back to the Place model as well.

If you don't want to keep an original state of the Place model, you can inject the NavigationManager class and then simply redirect to the Main.razor component using the following code:

```
    NavigationManager.NavigateTo("/main", true);
```

The preceding code is the quickest and easiest way to switch to the read-only state. However, doing this would cause the page to reload and invoke the API again to fetch the data, which can be expensive.

Let's move on and append the following code within the @code{} block of EditTouristSpot.razor:

```
  private void NotifyStateChange(Place place) {
    _appState.SetAppState(place);
  }
```

The NotifyStateChange() method takes a Place model as an argument. This is where we invoke the SetAppState() method of AppState to notify the main component of the change. This way, when we modify the form or perform an update, the main component can perform certain actions to act on it; for example, refreshing the data or updating some UI in the main component.

Next, append the following code within the @code{} block:

```
  protected async Task HandleValidSubmit() {
    await _placeService.UpdatePlaceAsync(Place);
    IsReadOnlyMode = true;
    NotifyStateChange(Place);
  }
```

The HandleValidSubmit() method in the preceding code will be triggered when clicking the Save button and when no model validation error occurred. This method calls the UpdatePlaceAsync() method of PlaceService and invokes the API to update a Place record.

Finally, append the following code within the @code{} block:

```
  private void UndoChanges() {
    IsReadOnlyMode = true;
    if (Place.Name.Trim() != PlaceCopy.Name.Trim() ||
    Place.Location.Trim() != PlaceCopy.Location.Trim() ||
    Place.About.Trim() != PlaceCopy.About.Trim()) {
      Place = PlaceCopy;
      NotifyStateChange(PlaceCopy);
    }
  }
```

The UndoChanges() method in the preceding code will be triggered when clicking the Cancel button. This is where we revert back to the values from the PlaceCopy object when any of the Place properties have been modified.

Let's move on to the next step and create the ViewTouristSpot component for displaying a read-only state of data.

Composing the ViewTouristSpot component

Go ahead and create a new Razor component within the Spots folder and name it ViewTouristSpot.razor. Replace the code generated so that it will look like the following:

```
@using BlazorServer.Web.Data

@if (IsEdit) {
<EditTouristSpot Place="Place" /> } else {
<div class="card">
  <img class="card-img-top" src="@Place.ImageData" alt="Card image cap">
  <div class="card-body">
    <h5 class="card-title">@Place.Name</h5>
    <h6 class="card-subtitle mb-2 text-muted">
      Location: <b>@Place.Location</b>
```

```
      Reviews: @Place.Reviews
      Last Updated: @Place.LastUpdated.ToShortDateString()
    </h6>
    <p class="card-text">@Place.About</p>
    <button type="button" class="btn btn-outline-primary"
            @onclick="(() => IsEdit = true)">
      Edit
    </button>
  </div>
</div>}

@code { [Parameter] public Place Place { get; set; }
      bool IsEdit { get; set; } = false; }
```

There really isn't much going on in the preceding code. Since this component is meant to be a read-only view, there's really no complex logic here. Just like in the EditTouristSpot.razor file, we also implemented an if-else statement to determine which HTML block to render. In the @code section, we only have two properties; the Place property is used to pass the model to the EditTouristSpot component. The IsEdit Boolean property is used as a flag to render HTML. We only set this property to true when clicking the Edit button.

Composing the Main component

Now that we are already familiar with the components for editing and viewing data, the last thing that we need to do is to create the main component to contain them in a single page. Let's go ahead and create a new Razor component under the Pages folder and name it Main.razor. Now, replace the generated code with the following:

```
@page "/main"
@using BlazorServer.Web.Data
@using BlazorServer.Web.Pages.Spots
@inject PlaceService _placeService
@inject AppState _appState
@implements IDisposable
```

The preceding code defines a new route using the @page directive. At runtime, the /main route will be added to the route data collection, enabling you to navigate to this route and render its associated components. We've used the @using directive to reference a class from the server and used the @inject directive to reference a service. We also used the @implements directive to implement a disposable component. We'll see how this is used later.

Now, let's continue composing our main component. Append the following code:

```
@if (Places == null) {
<p><em>Loading...</em></p> } else {
<div class="container">
  <div class="row">
    <div class="col-8">
      <h3>Featured Tourist Spot</h3>
      <ViewTouristSpot Place="Place" />
    </div>
    <div class="col-4">
      <div class="row">
        <h3>What's New?</h3>
        <div class="card" style="width: 18rem;">
```

```
          <div class="card-body">
            <h5 class="card-title">@_placeService.NewPlaceName</h5>
          </div>
        </div>
      </div>
      <div class="row">
        <h3>Top Places</h3>
        <div class="card" style="width: 18rem;">
          <div class="card-body">
            <ul>
              @foreach (var place in Places) {
      <li>
        <a href="javascript:void(0)"
           @onclick="(() => ViewDetails(place.Id))">
          @place.Name
        </a>
      </li>}
            </ul>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>}
```

The preceding code is responsible for rendering HTML. Once again, we've used **Bootstrap** CSS to set up the layout. The layout is basically composed of two columns' elements. In the first column, we render the ViewTouristSpot component and pass the Place model as the parameter to the component. We are going to see how the model is populated in the next section. The second column renders two rows. The first row displays the NewPlaceName property from PlaceService, and the second column displays the list of places presented using the <ul> HTML element. Within the <ul> tag, we've used the @ symbol to start manipulating the data in C# code. The foreach keyword is one of the C# reserved keywords, which is used for iterating data in a collection. Within the foreach block, we have constructed the items to be displayed in the <li> tag. In this case, the Name property of the Place model is rendered using implicit expressions.

To complete the Main.razor component, let's implement the server-side logic to handle user interactions and application states. Go ahead and append the following code:

```
@code {
  private IEnumerable<Place> Places;
  public Place Place { get; set; }
}
```

The preceding code defines two properties for storing the list of places and the current place being viewed.

Next, append the following code within the @code{} block:

```
  protected override async Task OnInitializedAsync() {
    await _placeService.InitializeSignalR();

    Places = await _placeService.GetPlacesAsync();
    Place = Places.FirstOrDefault();

    _placeService.NewPlaceName = Place.Name;
    _placeService.NewPlaceId = Place.Id;
```

```
    _placeService.OnChange += HandleNewPlaceAdded;
    _appState.OnChange += HandleStateChange;
  }
```

In the OnInitializedAsync() method, we've invoked the InitializeSignalR() method of PlaceService to configure the SignalR and Hub connections. We've also populated each property in the component. The Places property contains the data from the GetPlacesAsync() method call. Under the hood, this method invokes an API call to fetch data. The Places property is used to display the list of places in the Top Places section. The Place property, on the other hand, contains the first result from the Places collection and is used for displaying the data in the ViewTouristSpot component. We also set the NewPlaceName and NewPlaceId properties of PlaceService so that we will have a default display for the What's new section. We've also wired up both OnChange events from the PlaceService and AppState services to each corresponding method.

Next, append the following code within the @code{} block:

```
  private async void HandleNewPlaceAdded() {
    Places = await _placeService.GetPlacesAsync();
    StateHasChanged();
  }
```

The HandleNewPlaceAdded() method will be invoked when a server sends the event to Hub. This process is done when a new record is added via an API POST request. This method is responsible for updating the data in the component to reflect the new record in real time.

Next, append the following code within the @code{} block:

```
  private async void HandleStateChange() {
    Places = await _placeService.GetPlacesAsync();
    Place = _appState.Place;

    if (_placeService.NewPlaceId == _appState.Place.Id) {
      _placeService.NewPlaceName = _appState.Place.Name;
    }

    StateHasChanged();
  }
```

The HandleStateChange() method in the preceding code is responsible for keeping the Models state up to date. You can see in this method that we are repopulating the Places, Place, and NewPlaceName properties when the state has been changed. Note that we are only updating the NewPlaceName value if NewPlaceId matches the Place records that are being modified. This is because we don't want to change this value when we are editing a record that is not new. The StateHasChanged() call is responsible for re-rendering the component with the new state.

Next, append the following code within the @code{} block:

```
  private void ViewDetails(int id) {
    Place = Places.FirstOrDefault(o => o.Id.Equals(id));
  }
```

The ViewDetails() method in the preceding code takes an integer as a parameter. This method is responsible for updating the current Place model based on Id.

Finally, append the following code within the @code{} block:

```
  public void Dispose() {
    _appState.OnChange -= StateHasChanged;
    _placeService.OnChange -= StateHasChanged;
  }
```

In the preceding code, we will unsubscribed to the OnChange event when the Dispose() method is invoked. The Dispose() method is automatically called when the component is removed from the UI. It is very important to always unhook the component's StateHasChanged method from the OnChange event to avoid potential memory leaks.

Updating the NavMenu component

Now, let's add the /main route to the existing navigation component. Go ahead and open the NavMenu.razor file, which resides under the Shared folder. Append the following code within the <ul> element:

```
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="main">
                <span class="oi oi-list-rich" aria-hidden="true"></span> Tourist Spots
            </NavLink>
        </li>
```

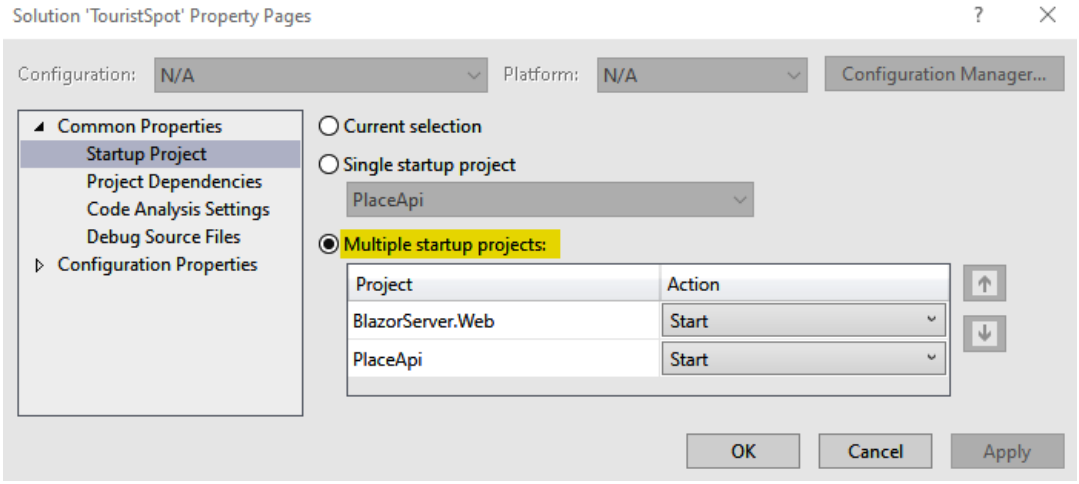
The preceding code adds a Tourist Spots link from the existing menu. This enables us to easily navigate to the main component page without having to manually type the route in the browser.

Running the application

One of the many great features built into Visual Studio is that it provides a capability for us to run multiple projects simultaneously in our local machine. Without this feature, we would have to deploy all applications in a web server where each of them can talk to one another. Otherwise, our Blazor web applications won't be able to connect to the web API.

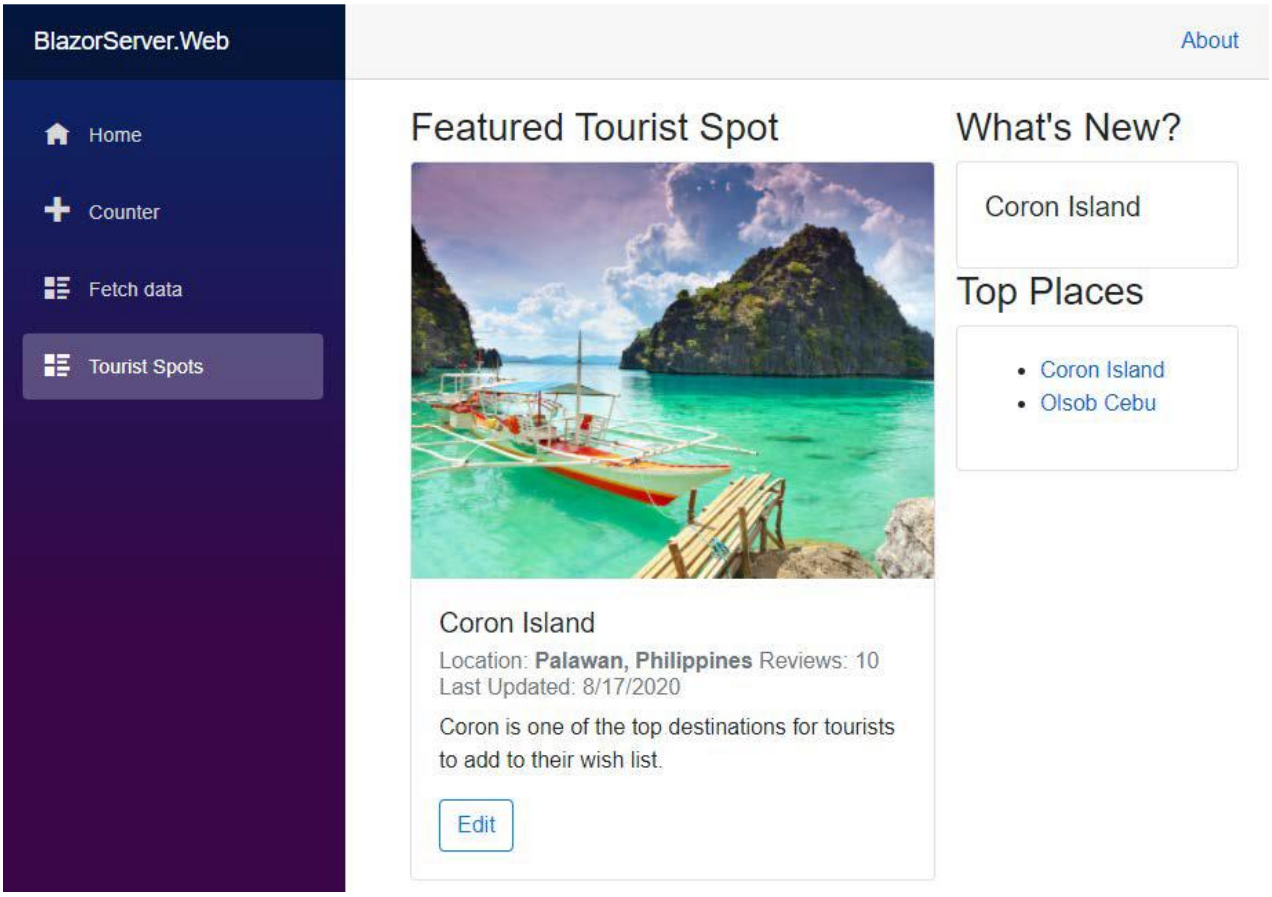
To run multiple projects at the same time in Visual Studio, perform the following steps:

1. Right-click on the Solution project and then select **Set startup projects**.
2. Select the **Multiple startup projects** radio button, as shown in the following screenshot:

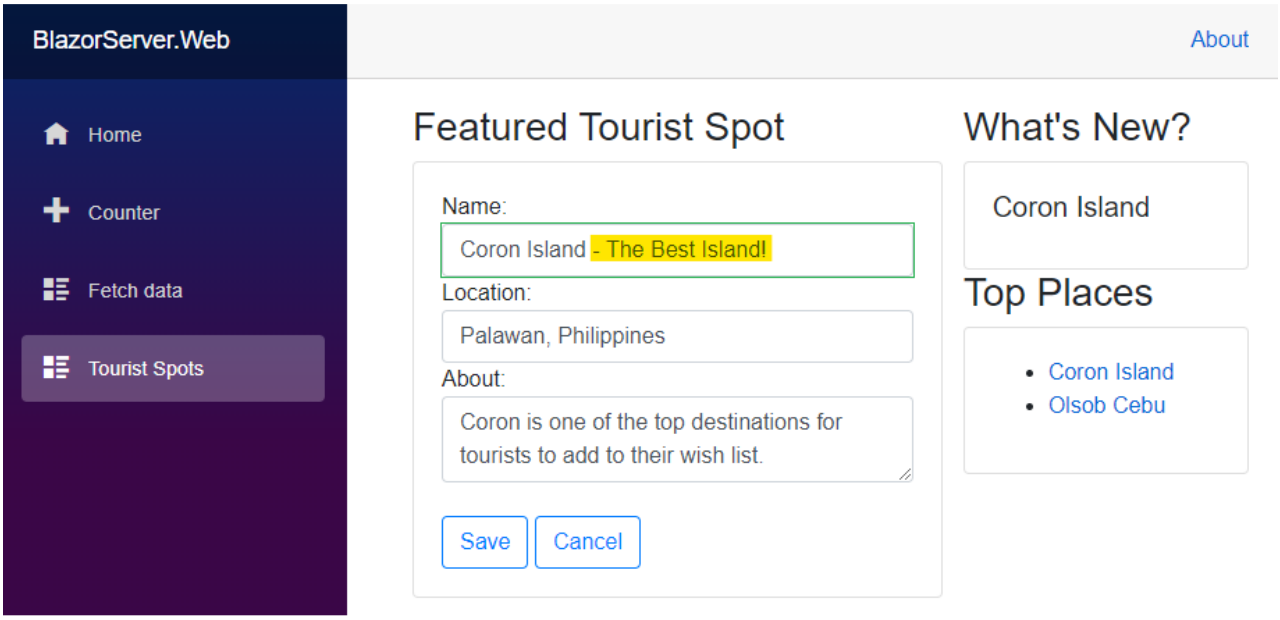


3. Select Start as the action for both projects.
4. Click **Apply** and then **OK**.

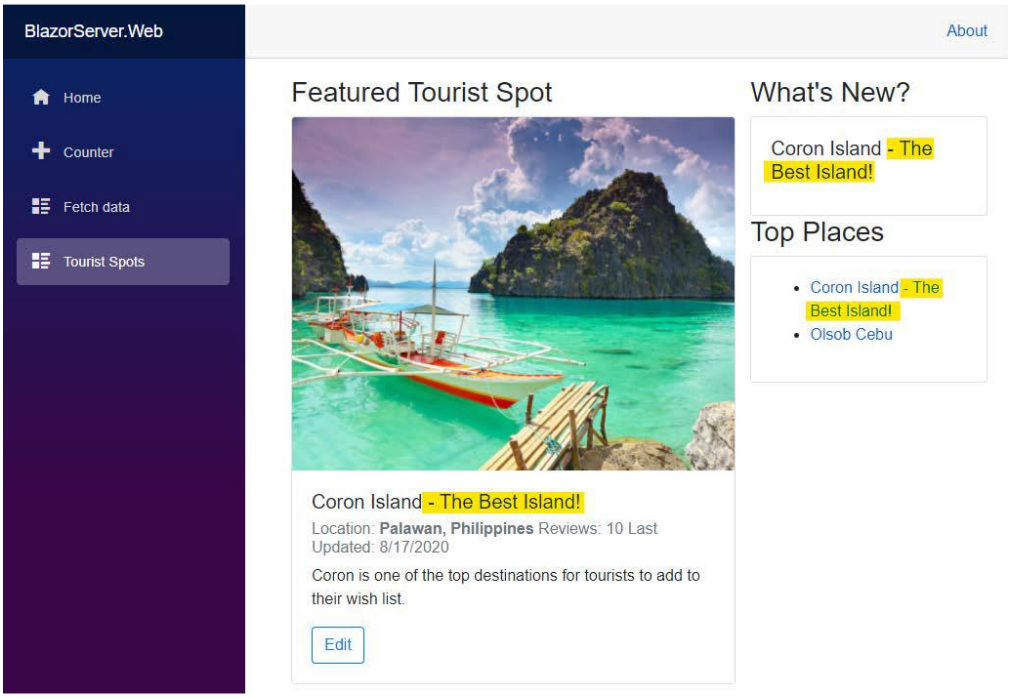
Now, build and run the application using **Ctrl + F5**. From the navigation sidebar menu, click the **Tourist Spots** link and the Main component page should display just like in the following screenshot:



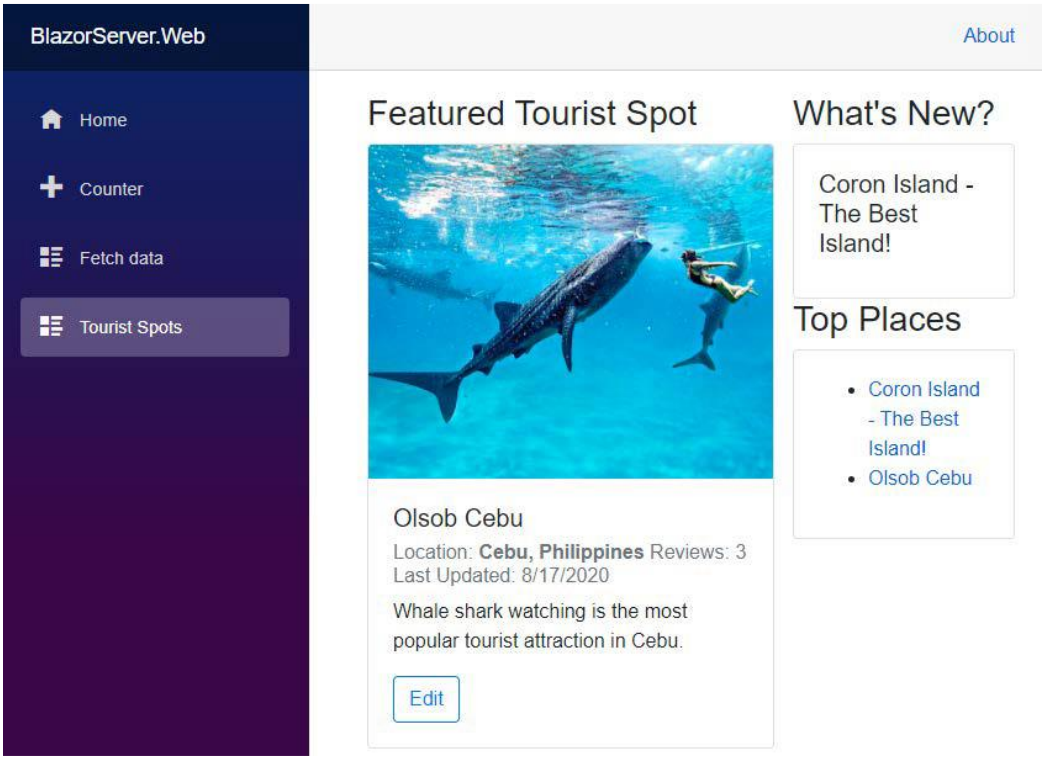
Clicking the Edit button will display the EditTouristSpot component, as shown in the following screenshot:



In the preceding screenshot, the **Name** property was modified. Clicking the Cancel button will discard the changes and bring you back to the default view. Clicking **Save** will update the record in our in-memory database, update the state, and reflect the changes to the **Main** component, as shown in the following screenshot:



You can also select any items from the **Top Places** section, and this should bring up the corresponding details on the page. For example, clicking on the **Oslob Cebu** item will update the page to the following:



Notice that all the details information has been updated except for the **What's New?** section. This was intentional because we only want to update it when there's a new record posted in the database. We are going to see how this section will be updated in the next section.

If you've made it this far, congratulations! You just had your first Blazor web application running with live data connected to an API! Now, let's continue the fun and create a Blazor WebAssembly WASM (app) where we can submit new tourist spot records and reflect the changes in the Blazor Server app in real time.