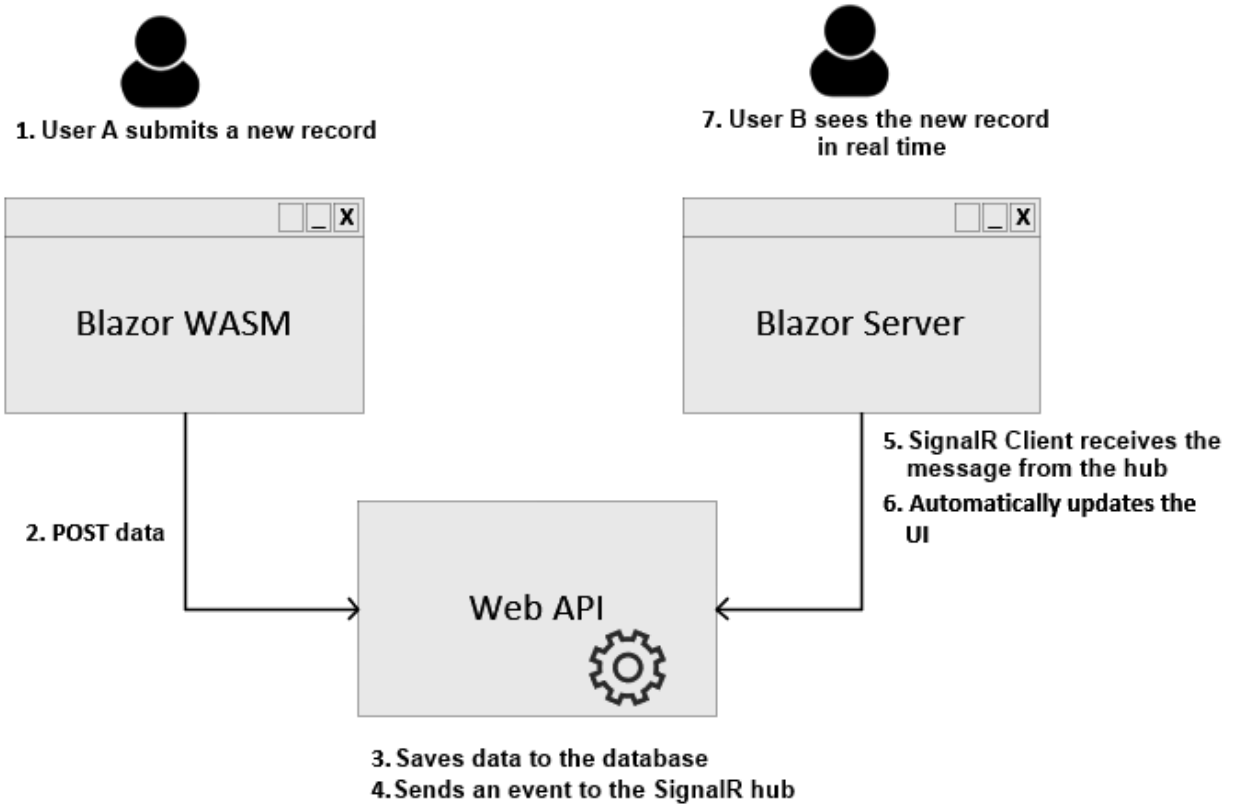
Exploring the Blazor Web Framework

Creating the Blazor Web Assembly project

In the previous project, we learned how to create a web app with basic functionalities such as fetching and updating records via a web API call. In this project, we will build the frontend Progressive Web Application (PWA) to create a new record. This process is executed by invoking an API endpoint to post data and sends an event to Hub to automatically update the Blazor Server UI in real time when a new record is submitted.

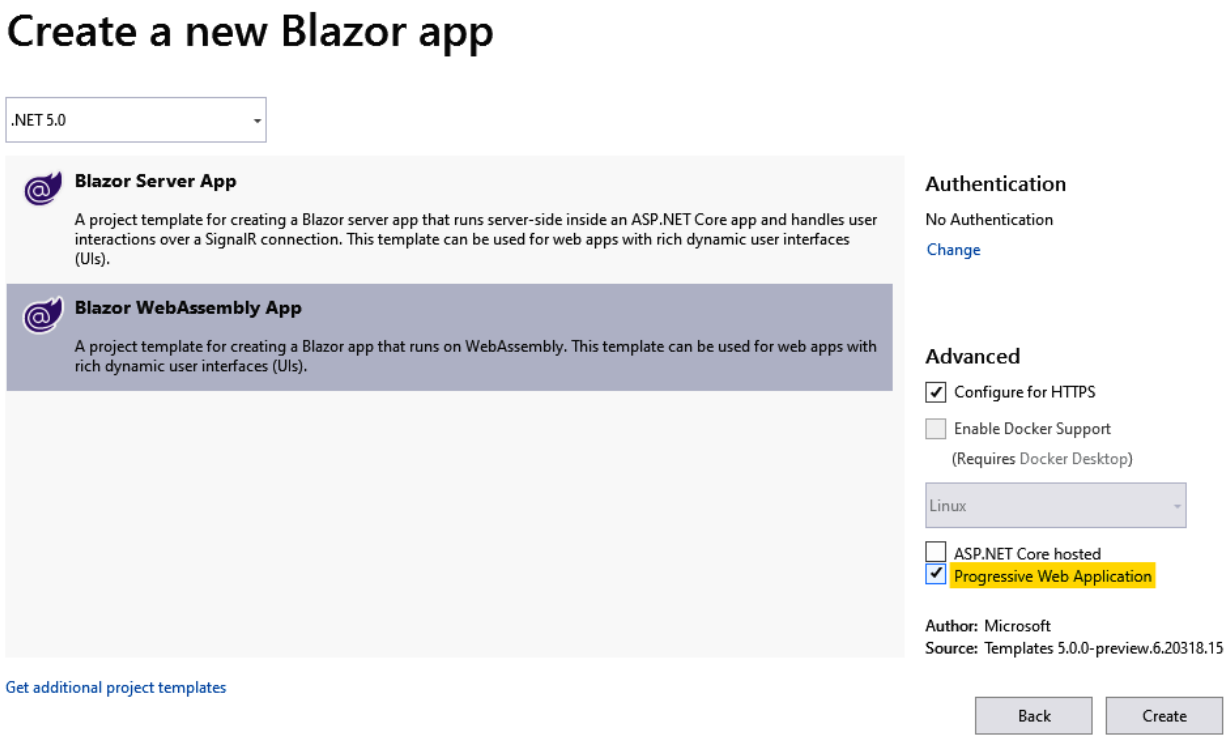
Here's an attempt showing how the process works:



The preceding diagram shows the high-level process of how the real-time functionality works. The steps are pretty much self-explanatory, and it should give you a better understanding of how each application connects to one another. Without further ado, let's start building the last project to complete the whole application.

Go ahead and add a new Blazor WebAssembly project within the existing project solution. To do this, just right-click on **Solution** and then select **Add | New Project**. In the window dialog, select Blazor App and then click **Next**. Set the name of the project to BlazorWasm.PWA and then click **Create**.

In the next dialog, select **Blazor WebAssembly App** and then check **the Progressive Web Application** checkbox, as shown in the following screenshot:



Click **Create** to let Visual Studio generate the default template.

The project structure of the Blazor WebAssembly project is somewhat similar to Blazor Server except for the following:

- It doesn't have a Startup.cs file. This is because a Blazor WASM project is configured differently and uses its own host to run the application.
- The Progam.cs file now contains the following code:

```
    public static async Task Main(string[] args) {
      var builder = WebAssemblyHostBuilder.CreateDefault(args);
      builder.RootComponents.Add<App>("app");

      builder.Services.AddTransient(sp => new HttpClient {
       BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });

      await builder.Build().RunAsync();
    }
```

  In the preceding code, we can see that it uses WebAssemblyHostBuilder instead of using the typical ASP.NET Core IHostBuilder to configure a web Host. It also configures HttpClient with BaseAddress set to HostEnvironment.BaseAddress, which is the host address where the application itself is running, for example, localhost:<port>.
- It doesn't have the _Host.chtml file in the **Pages** folder. If you recall, in the Blazor Server project, the _Host.chtml file is the main entry point for the application where it bootstraps the App.razor

component. In Blazor WASM, App.razor is added to the application start instead, as you can see in the Program.cs file.

- It doesn't have the **Data** folder where it configures sample data for the default Weatherforecast service. The sample data is now moved to the weather.json file under the wwwroot/sample-data folder.
- A few other new files have been added to wwwroot as well, such as index. html, manifest.json, and service-worker.js. index.html is the one that actually replaces the _Host.chtml file, which contains the main HTML document for the application. You can see that this file contains the <head> and <body> tags, as well as rendering the <app> component, CSS, and the JavaScript framework. The manifest.json and service-worker.js files enable the Blazor WASM app to turn into a PWA.

I am pretty sure that there are many other differences between Blazor Server and WebAssembly, but the items highlighted in the list are the key differences.

Creating the model

Now, let's start adding the feature we need for this project. Create a new folder called Dto in the project root. Within the Dto folder, add a new class called CreatePlaceRequest.cs and copy the following code:

```
using System.ComponentModel.DataAnnotations;

namespace BlazorWasm.PWA.Dto {
  public class CreatePlaceRequest {
    [Required]
    public string Name { get; set; }
    [Required]
    public string Location { get; set; }
    [Required]
    public string About { get; set; }
    [Required]
    public int Reviews { get; set; }
    public string ImageData { get; set; }
  }
}
```

The preceding code defines a class that houses some properties. Notice that the class resembles the Place class from the web API, except that we've used data annotations by decorating a few properties with the [Required] attribute. This attribute ensures that the properties will not be posted to the database if they are left empty.

Let's move on to the next step and create the component for adding new records to the database.

Composing the Index component

Now, navigate to the Index.razor component. Delete the existing code within it and add the following code:

```
@page "/"

@using Dto
@inject HttpClient client
```

The preceding code sets the route to the root using the @page directive. The next line declares a reference to the C# namespace using the @using directive. We are going to use the Dto namespace to access a class and populate the component with values from the properties in the class. The last line injects an HttpClient object for us to communicate with the web API.

Next, append the following code block:

```
<h1>Submit a new Tourist Destination Spot</h1>
<EditForm Model="@NewPlace" OnValidSubmit="HandleValidSubmit">
  <div class="card" style="width: 30rem;">
    <div class="card-body">
      <DataAnnotationsValidator />
      <ValidationSummary />
      Browse Image:
      <InputFile OnChange="@HandleSelection" />
      <p class="alert-danger">@errorMessage</p>
      <p>@status</p>
      <p>
        <img src="@imageData" style="width:300px; height:200px;">
      </p>

      Name:
      <InputText class="form-control" id="name" @bind-Value="NewPlace.Name" />
      Location:
      <InputText class="form-control" id="location" @bind-Value="NewPlace.Location" />
      About:
      <InputTextArea class="form-control" id="about" @bind-Value="NewPlace.About" />
      Review:
      <InputNumber class="form-control" id="review" @bind-Value="NewPlace.Reviews" />

      <br />
      <button type="submit" class="btn btn-outline-primary oi-align-right">Post</button>
    </div>
  </div>
</EditForm>
```

The preceding code is the HTML code that renders the form with input elements and a button to upload an image. It also uses an EditForm component to handle form submission and model validations. We're not going to elaborate on how the code works because we've already covered this in the previous section when we built the components for the Blazor Server project.

In this example, we are using the InputFile Blazor component to upload an image and configure the OnChange event that is wired to the HandleSelection method.
By default, the InputFile component only allows single-file selection. To supportmultiple-file selection and uploading, set the multiple attribute just like in the following code snippet:

```
  <InputFile OnChange="HandleSelection" multiple />
```

For more information about the InputFile component, check out the Further reading section of this module.

Let's continue by implementing the server-side code logic. Append the following code:

```
@code { string status;
      string imageData;
      string errorMessage;
}
```

The preceding code defines a few private fields that are required in the component UI. The status field is a variable for storing the uploaded status text. imageData is for storing the encodedimage data, and errorMessage is for storing the error text.

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

```
async Task HandleSelection(InputFileChangeEventArgs e) {
  errorMessage = string.Empty;
  int maxFileSize = 2 * 1024 * 1024;
  var acceptedFileTypes = new List<string>() { "image/png", "image/jpeg", "image/gif" };
  var file = e.File;

  if (e.File != null) {
    if (!acceptedFileTypes.Contains(file.ContentType)) {
      errorMessage = "File is invalid.";
      return;
    }

    if (file.Size > maxFileSize) {
      errorMessage = "File size exceeds 2MB";
      return;
    }

    var buffer = new byte[file.Size];
    await file.OpenReadStream().ReadAsync(buffer);
    status = $"Finished loading {file.Size} bytes from {file.Name}";
    imageData = $"data:{file.ContentType};base64,{Convert.ToBase64String(buffer)}";
  }
}
```

The HandleSelection() method in the preceding code takes InputFileChangeEventArgs as the parameter. In this method, we only allow a single file to be uploaded instead of multiple files by reading the e.File property. If you accept multiple files, then use the e.GetMultipleFiles() method instead. We also defined a couple of pre-validation values for the maximum file size and file types. In this example, we only allow 2 MB as the maximum file size and only accept .PNG, .JPEG, and .GIF file types to be uploaded. We then perform some validation checks and display an error if any condition is not met. If all conditions are met, we copy the file being uploaded in a stream and convert the resulting bytes into Base64String so we can set the image data to an <img> HTML element.

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

```
private CreatePlaceRequest NewPlace = new CreatePlaceRequest();

async Task HandleValidSubmit() {
  NewPlace.ImageData = imageData;
  var result = await client.PostAsJsonAsync(
       "https://localhost:44332/api/places", NewPlace);

}
```

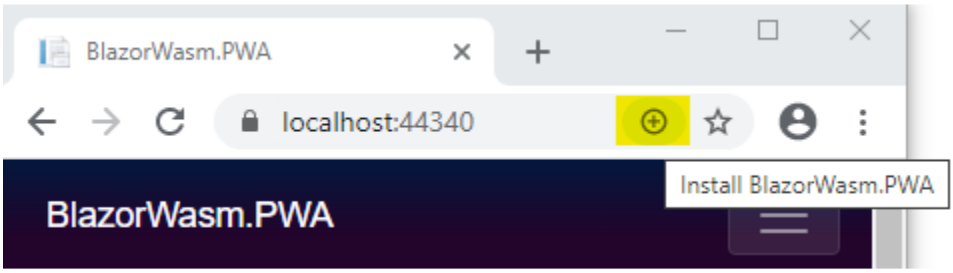
The HandleValidSubmit() method in the preceding code will be invoked when clicking the Post button and if no model validation errors occurred. This method takes the NewPlace object and passes it the API call to perform HTTP POST.

That's it! Now, let's try to run the application.

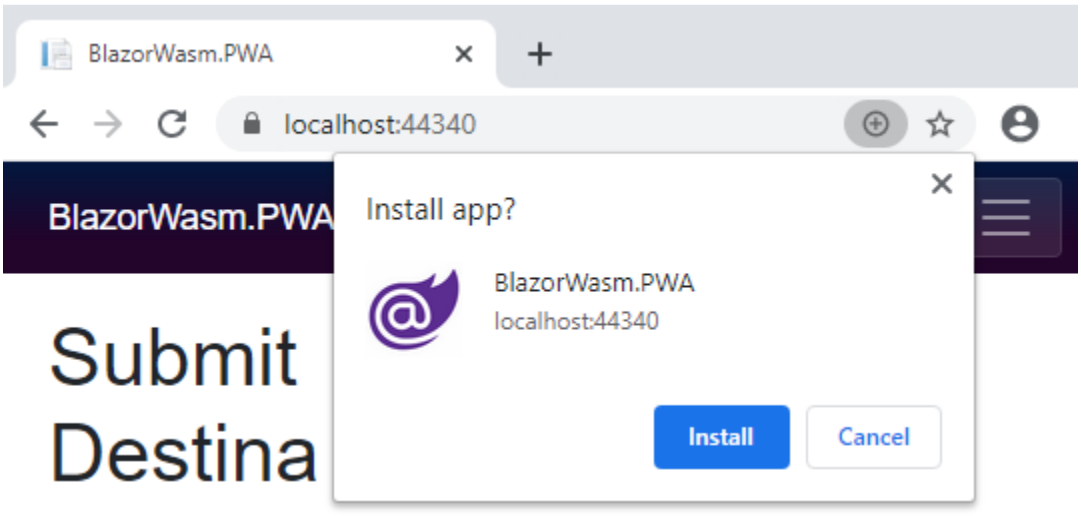
Running the application

Now, include the Blazor WASM project as a startup project and then click **Ctrl + F5** to run the application. You should see three browser tabs running each application. You can minimize the tab that runs the web API because we don't need to do anything with it. Now, look for the Blazor WASM tab.

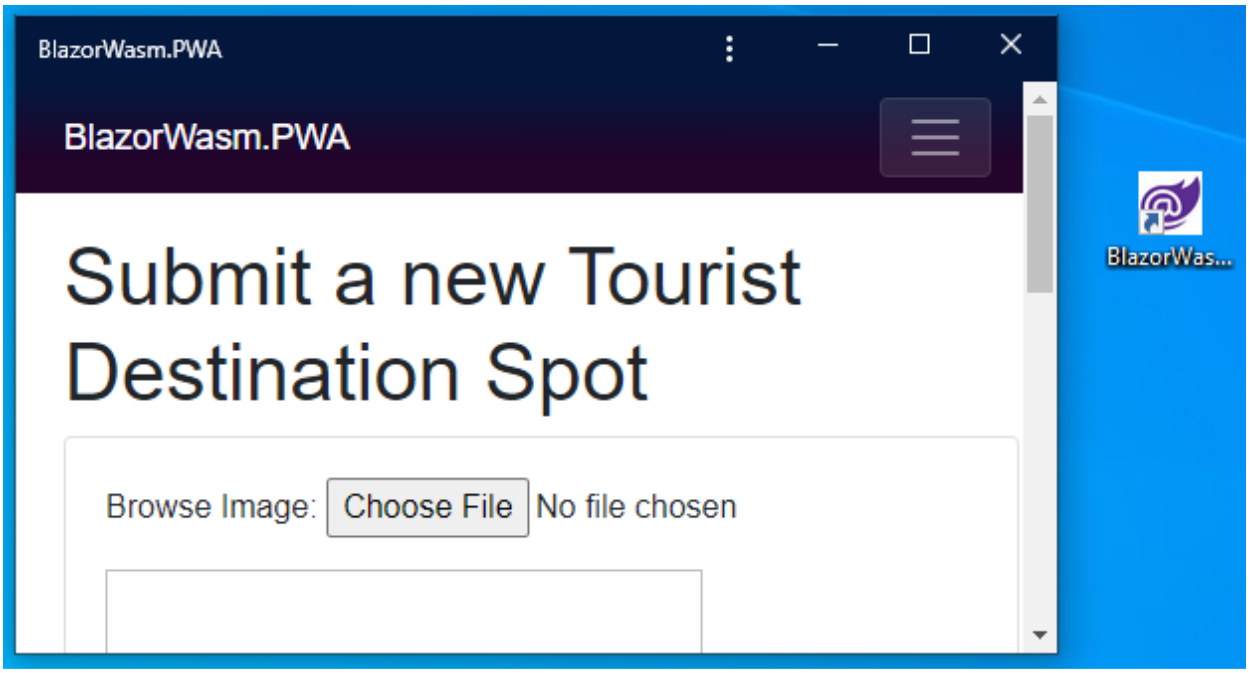
To turn the Blazor WebAssembly page into a PWA, you simply click the + sign located in the browser navigation bar, as shown in the following screenshot:



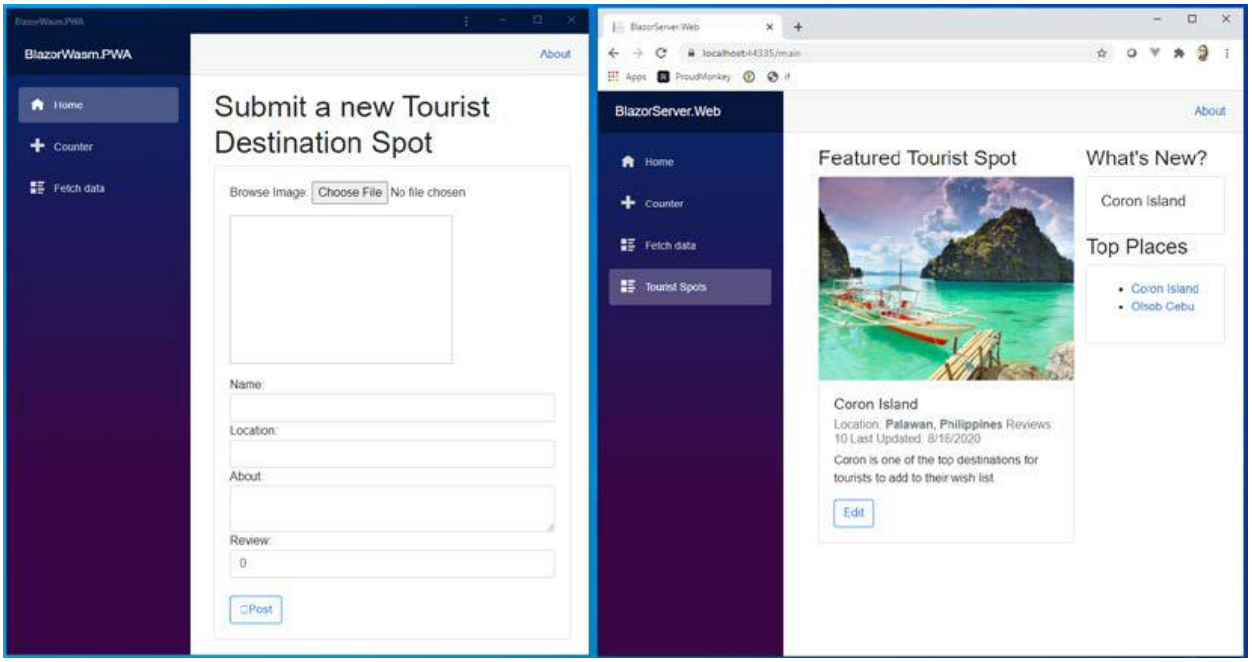
Clicking the + sign will prompt a dialog asking you whether you want to install Blazor as a standalone app on your desktop or mobile device, as shown in the following screenshot:



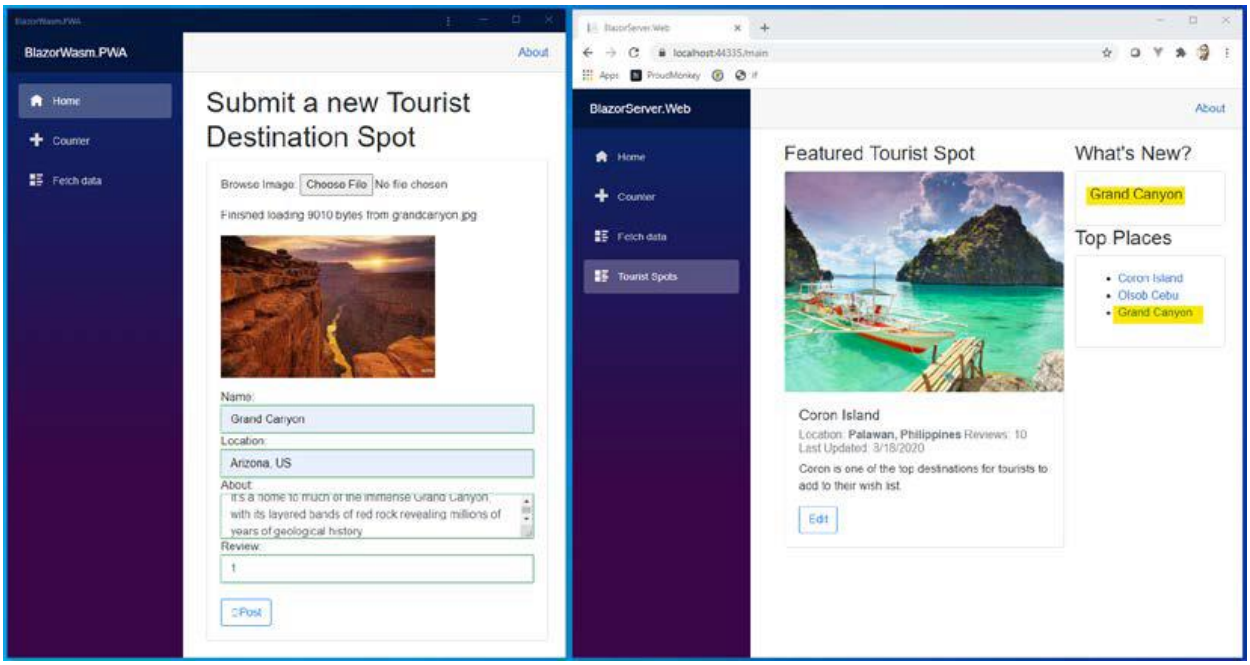
Clicking Install will create an icon on your desktop or mobile device as if it's a regular native app that has been installed and turns the web page into a window without the URL bar, like this:



Now, open both the Blazor Server app and Blazor PWA app side by side so you'll see how a real-time update works:



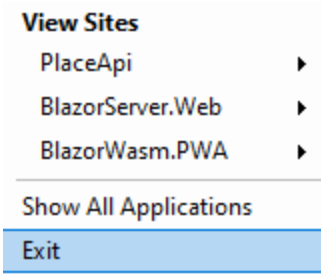
Now, browse an image and enter the required fields to submit a new Place record. When you click Submit, you'll notice in the Blazor Server app (the right-hand window in the preceding screenshot) that the What's New? and Top Places sections are automatically updated with the newly added Place name without you having to refresh the page. Here's an example of what it looks like:



In the preceding screenshot, the Grand Canyon name automatically appears in the Blazor Server web UI in real time right after clicking the **Post** button.

Uninstalling the PWA app

To completely uninstall the PWA app from your local machine or device, make sure to exit all the apps that are running in IIS Express. You can access the IIS Express manager in the bottom-right corner of your Windows machine task bar, as shown here:



After exiting all the apps, you can uninstall the PWA app just like you would normally uninstall an application on your machine.

Summary

In this module, we learned about the different flavors of the Blazor web framework by doing some hands-on coding. We learned how we can easily build a powerful web application in Blazor in concert with other ASP.NET Core technology stacks by just applying our C# skills and without the need to write JavaScript. We saw how we can easily integrate features and capabilities that are already available in .NET, such as realtime functionality. We also learned how to perform basic form data bindings, state management, routing, and how to interact with the backend REST APIs to consume and pass data. Having to learn these basic concepts and fundamentals is crucial when you will be building real-world applications.