AZURE MAPS USING BLAZOR

SUCCINCTLY

BY **MICHAEL WASHINGTON**

SUCCINCTLY EBOOK SERIES

Syncfusion®

Azure Maps Using Blazor Succinctly

By

Michael Washington

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

Copyright © 2022 by Syncfusion, Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.
ISBN: 978-1-64200-226-3

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books 6

About the Author 8

Introduction 9

Chapter 1 Getting Started with Azure Maps 11

Controlling pricing and tracking usage 12

Authentication and security 13

Shared Key authentication 13

Azure Active Directory (AAD) authentication 14

Additional information and help 14

Chapter 2 Displaying Azure Maps in Blazor 15

Install .NET Core and Visual Studio 15

Create the application 16

Render image using a shared key 18

Render image using AAD token 24

Create the service principal 25

Assign the service principal to the Azure Maps account 28

Store and retrieve the service principal settings 32

Display the tile 37

Retrieve your current location 40

Render map using AzureMapsControl.Components 43

Display an Azure map 47

Chapter 3 The Store Finder Application 51

Chapter 4 Create the Application 52

Install SQL Server 53

Add Syncfusion 53

Install and configure Syncfusion 54

Create the database...55

Chapter 5 Creating Store Administration 60

Create the data layer...60

Store location service 63

Geocode store locations..66

Add and edit store locations 71

Chapter 6 Create the Store Search Page...78

Reverse-geocode a location...78

Update store location service 81

Search for store locations...84

Re-center map on selected store...90

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, CEO
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn’t everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors’ tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

About the Author

Michael Washington is a Microsoft MVP, an ASP.NET C# programmer, and the founder of BlazorHelpWebsite.com. He is the author of over 15 books, including *An Introduction to Building Applications with Blazor*. He has extensive knowledge in process improvement, billing systems, and student information systems. He has a son, Zachary, and resides in Los Angeles with his wife Valerie.

You can follow him on Twitter at @ADefWebserver.

Introduction

You can use Microsoft Azure Maps to create advanced mapping applications.

Microsoft Azure Maps is part of Microsoft Azure Cloud Services and provides a wide range of powerful geospatial capabilities and a rich set of REST APIs. It has SDKs for both web and mobile applications.

Azure Maps provides a full range of services, including: maps, traffic, weather, search, routing, geolocation, and geofencing. It also offers a map creator to create indoor maps.

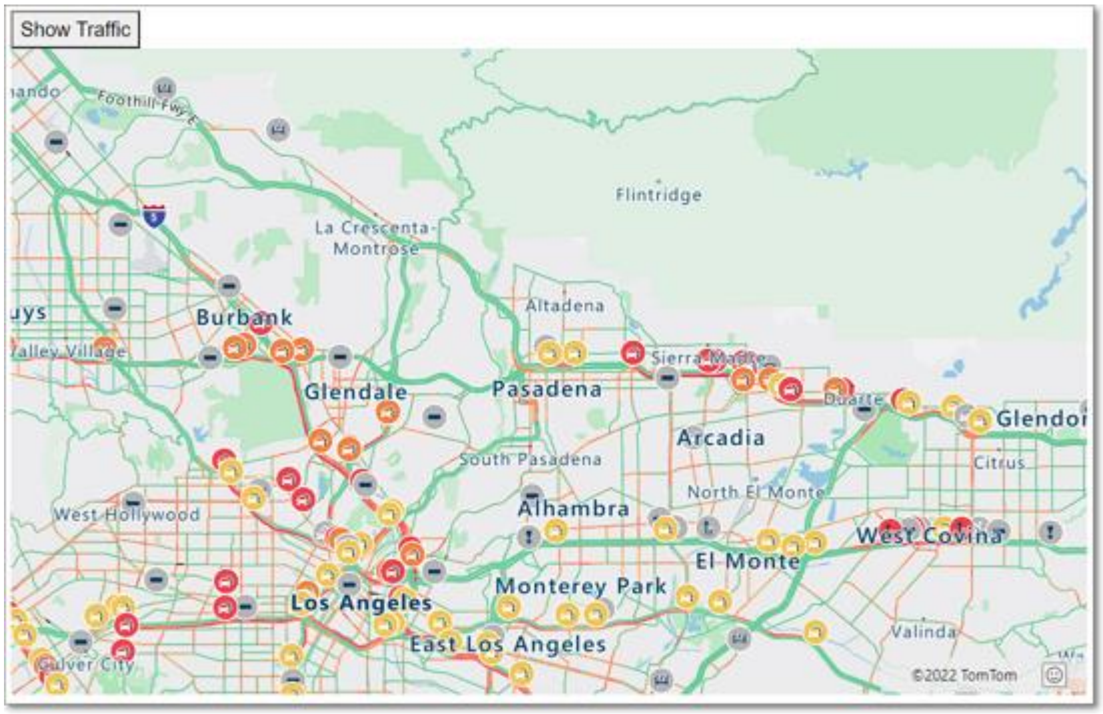


Figure 1: Azure Map Showing Live Traffic

While Azure Maps offers a number of services, the examples in this book will cover the following popular features:

- **Render service**: Azure Maps provides REST APIs to render vector and raster maps. Various styles and satellite imagery can be combined with custom markers to produce the final image.
- **Map control**: Azure Maps features a web-based JavaScript control capable of displaying vector map tiles from the Azure Maps API that also allows user interaction. In the examples presented in this book, we will use AzureMapsControl.Components, a Microsoft Blazor component that wraps the Azure Maps control in an easy-to-use API that allows programming using C# rather than JavaScript.

- **Search service**: The search service allows you to easily geocode addresses and locations to obtain latitude and longitude values needed to perform functionality such as determining nearby stores to the user’s current location. The search service can also reverse geocode to provide street addresses when provided latitudes and longitudes.
- **Traffic**: Real-time traffic flow and incident views can be obtained and displayed on the map control.

See the Azure Maps website for live examples of what Azure Maps can do.

The code for this e-book is available on GitHub. The step-by-step instructions use Visual Studio 2022 Community edition, which is available for free at this link.

Chapter 1 Getting Started with Azure Maps

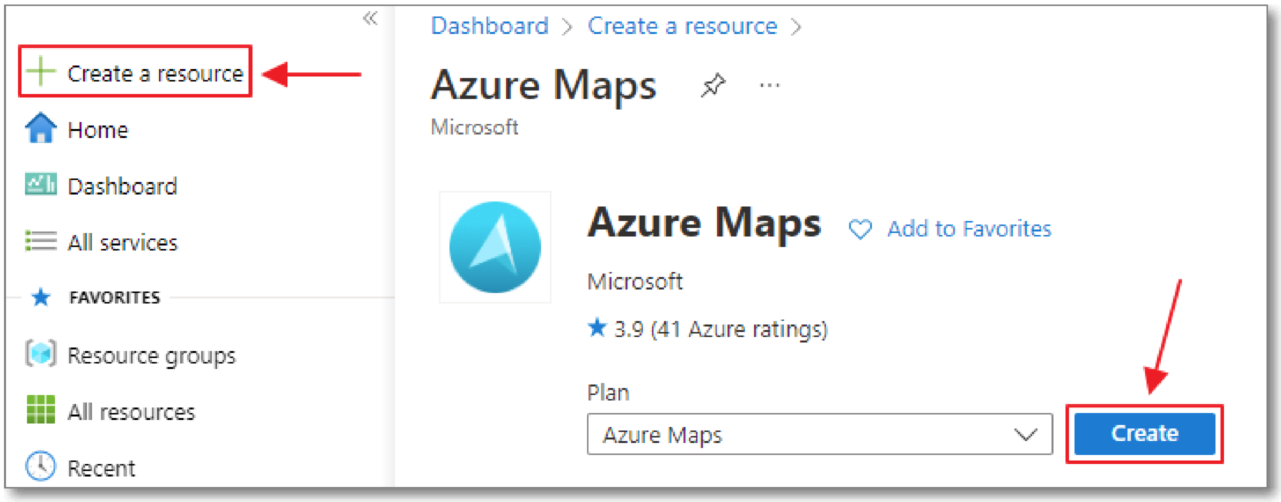


Figure 2: Creating Azure Maps Account

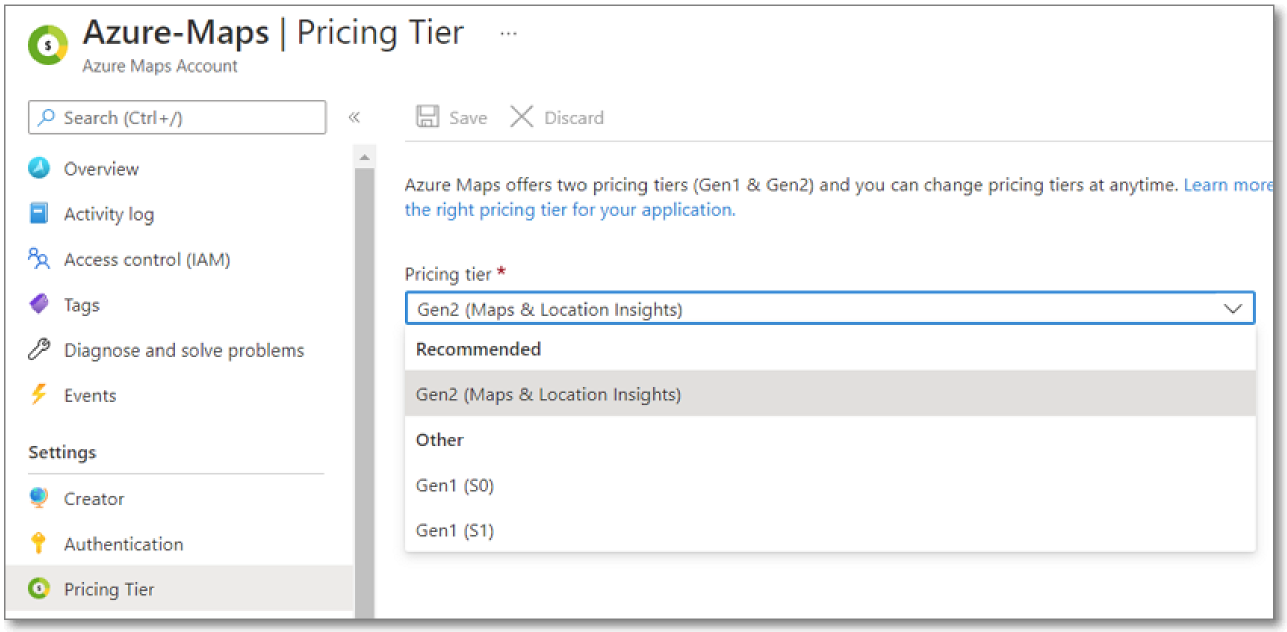
Getting started with Azure Maps is easy. In your web browser, navigate to https://portal.azure.com/ and log in using an existing account or create a new one.

Next, select **Create a resource** > **Azure Maps** > **Create**.

This will display forms that, at minimum, require you to select the subscription, resource group, region, and pricing tier (see the following section for more information on pricing tiers).

Finally click **Create**.

Controlling pricing and tracking usage



Azure Maps offers two pricing tiers (Gen1 & Gen2) and you can change pricing tiers at anytime. Learn more the right pricing tier for your application.

Pricing tier *

Gen2 (Maps & Location Insights)

Recommended

Gen2 (Maps & Location Insights)

Other

Gen1 (S0)

Gen1 (S1)

Figure 3: Configuring Pricing Tier

You configure your Azure Maps pricing on the Pricing Tier tab.

Azure Maps bills by the number of API transactions, such as performing a geocoding transaction or retrieving a map tile. At the time of writing, Azure Maps provides two pricing options, Gen1 and Gen2:

- **Gen1** pricing consists of two tiers, **Standard S0** and **Standard S1**. **Standard S0** is less expensive per transaction, but it does not support the full set of Azure Map services and limits the maximum number of queries per second (QPS) allowed. **Standard S1** supports the full range of Azure Maps services and higher QPS.
- **Gen2** pricing is the newest offering, and it is the pricing option Microsoft recommends. It has several pricing tiers, including a free tier. All tiers support the full range of Azure Maps services. With the **Gen2** option, you do not preselect the tier, as you do with the **Gen1** offering; the cost, per 1,000 transactions, is reduced based on the number of transactions.

Note: You can find detailed information on the queries per second (QPS) limitations for each pricing plan here and detailed pricing information here.

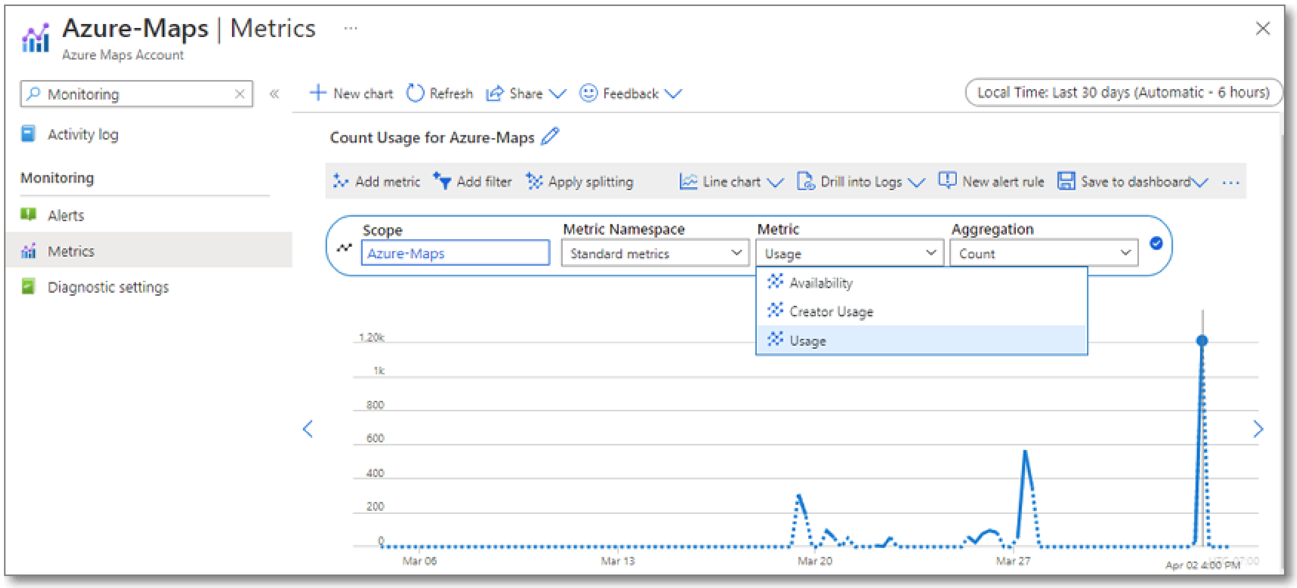


Figure 4: Track Azure Maps Usage

You can track the API usage on the metrics tab of the Azure Maps account. Simply select **Azure-Maps** as the scope and **Usage** as the metric.

Authentication and security

When making calls to the Azure Maps API, authentication is required. Azure Maps supports three methods to authenticate these requests: Shared Key, Azure Active Directory (AAD), and Shared Access Signature (SAS) Token.

Shared Key and AAD authentication will be covered in this book. At the time of this writing, SAS Token authentication is only a preview feature and is not approved for production applications.

Shared Key authentication

To use a shared key for a request to Azure Maps services, you simply add the key as a parameter to the URL. For example:

```
https://atlas.microsoft.com/map/static/png?subscription-key={SubscriptionKey}
```

However, the Shared Key authentication is not recommended by Microsoft for use in web applications. Microsoft only recommends using them for daemon apps where the key is not exposed and potentially compromised.

Azure Active Directory (AAD) authentication

AAD authentication requires your code to obtain an authorization token and to pass that token in the request header. For example:

```
client.DefaultRequestHeaders.Authorization = new
System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", AccessToken)
```

This is the recommended method for authentication because the tokens are short-lived and can be issued with fine control.

Note: See this article for more information on security options and recommendations.

Additional information and help

Check the following resources for additional information and assistance:

- Azure Maps documentation
- Azure Maps Q&A forum
- Best practices

Chapter 2 Displaying Azure Maps in Blazor

We will now call the Azure Maps API and return a map image. We will perform the task using two different security authentication methods; Shared Key and AAD. To do this, we will create an application using Microsoft Blazor and display map tiles and interactive maps.

Initially, we will only retrieve and display a single map tile. We will conclude this chapter by demonstrating how you can retrieve a full interactive map and even display live traffic.

Note: See this website for more information about Microsoft Blazor.

Note: Azure Map controls can be placed in any kind of web application, including simple HTML pages, ASP.NET applications, and Blazor WebAssembly applications.

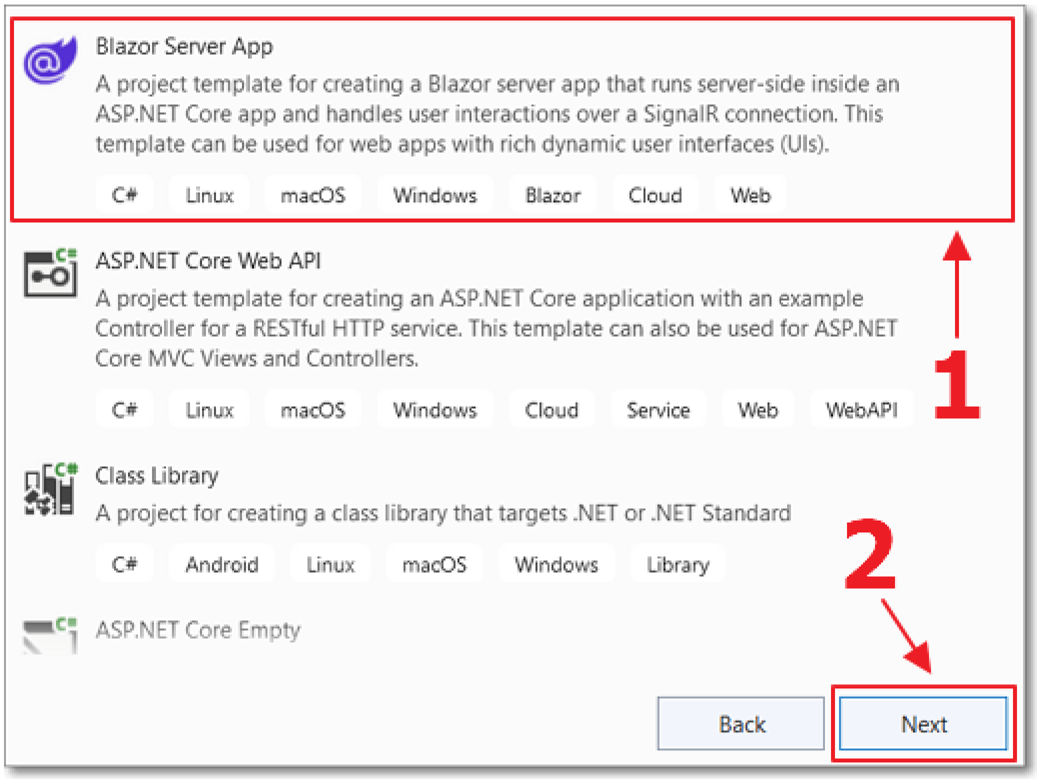
Install .NET Core and Visual Studio

If you do not already have the following software installed, these steps are required to create the application:

1. Install the .NET 6 SDK from this site.
2. Install Visual Studio 2022 version 17.1 (or later), with the ASP.NET, web development, and NET Core workload here.

Note: The requirements to create applications using Blazor are constantly evolving. For the latest requirements, see this article.

Create the application


Blazor Server App

A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

C# Linux macOS Windows Blazor Cloud Web

ASP.NET Core Web API

A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

C# Linux macOS Windows Cloud Service Web WebAPI

Class Library

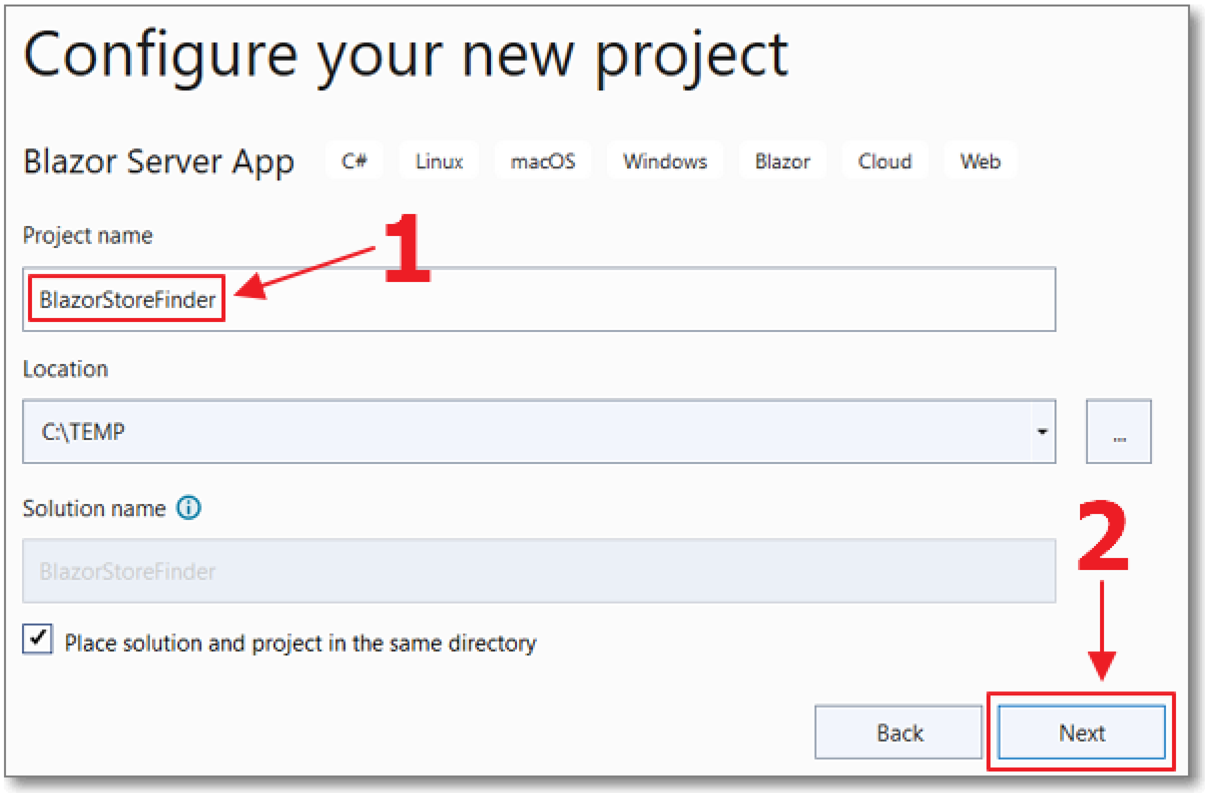
A project for creating a class library that targets .NET or .NET Standard

C# Android Linux macOS Windows Library

ASP.NET Core Empty

Figure 5: Create Blazor Server Application

To create the application, open **Visual Studio** and select **Create a new project** > **Blazor Server App** > **Next**.


Configure your new project

Blazor Server App

Project name

BlazorStoreFinder

Location

C:\TEMP

Solution name

BlazorStoreFinder

✓ Place solution and project in the same directory

Figure 6: Create BlazorStoreFinder Project

Set the **Project name** to **BlazorStoreFinder** and click **Next**.

On the next screen, select **.NET 6.0** as the **Framework** and click **Create**.

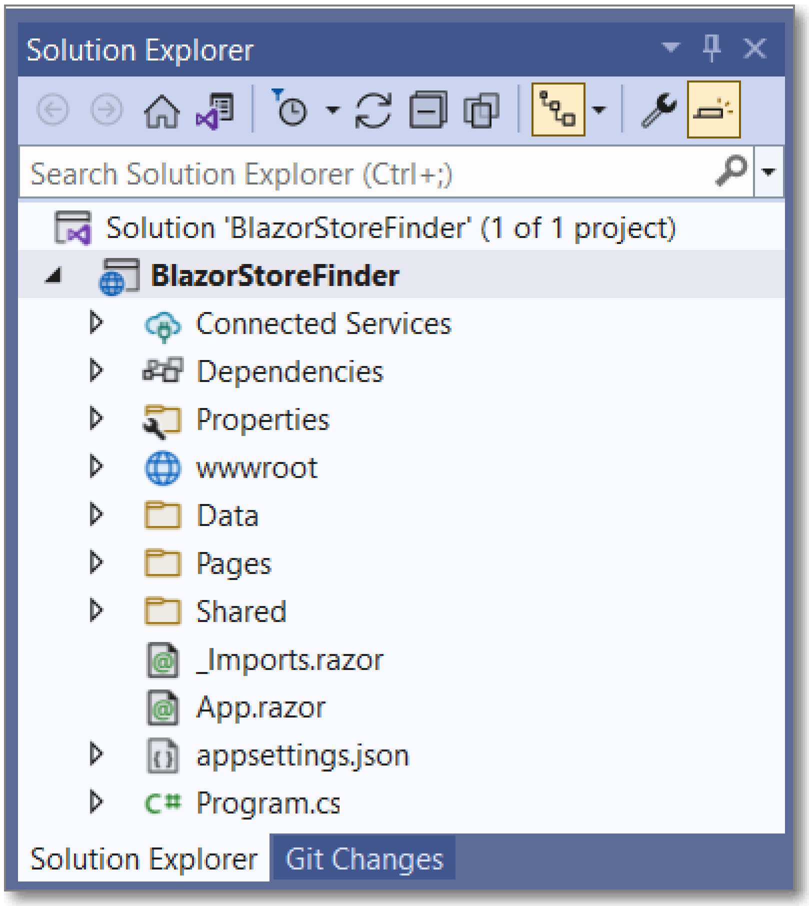


Figure 7: BlazorStoreFinder Project

The project will be created.

Render image using a shared key

In the following steps, we will retrieve a map image using a shared key.

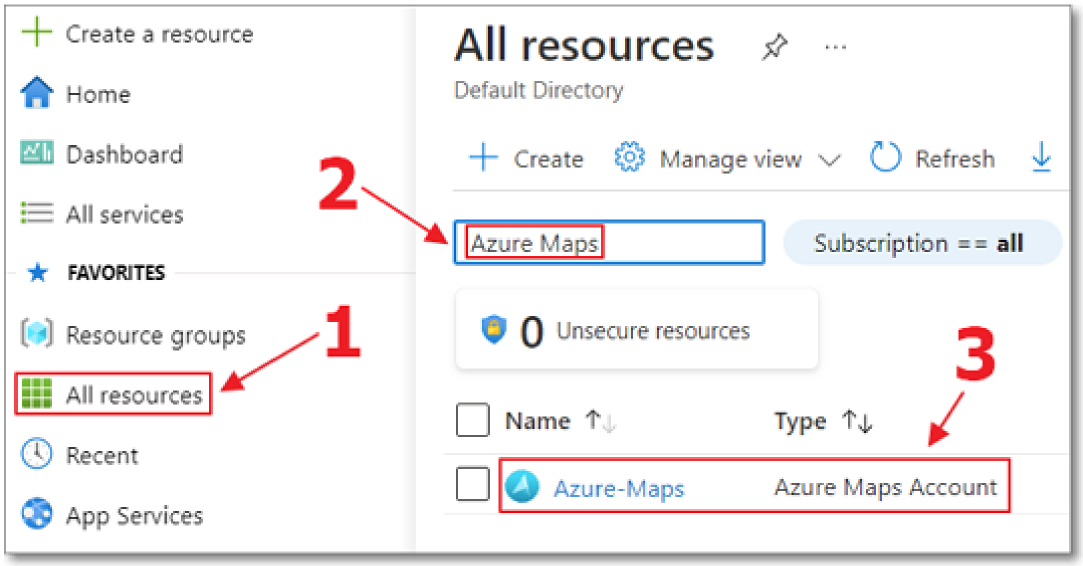


Figure 8: Search and Open Azure Maps Account

Navigate to the Azure Portal (https://portal.azure.com/), log in, click **All resources**, and search for the Azure Maps account you created earlier.

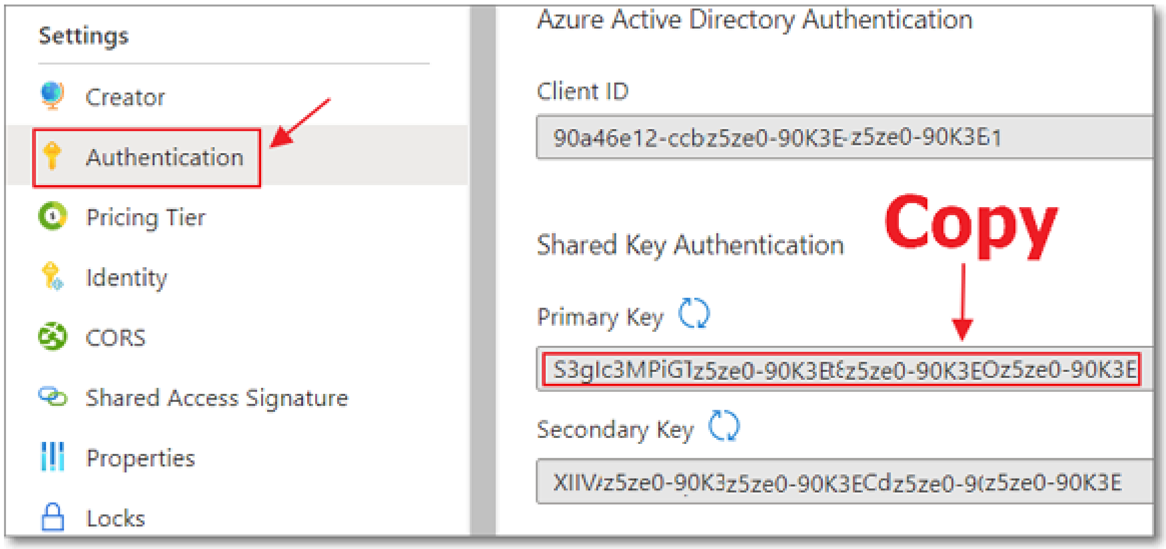

Settings
Creator
Authentication
Pricing Tier
Identity
CORS
Shared Access Signature
Properties
Locks
Azure Active Directory Authentication
Client ID
90a46e12-ccbz5ze0-90K3E-z5ze0-90K3E1
Shared Key Authentication
Copy
Primary Key
S3glc3MPiG1z5ze0-90K3E€z5ze0-90K3EOz5ze0-90K3E
Secondary Key
XIIV/z5ze0-90K3z5ze0-90K3ECdz5ze0-9(z5ze0-90K3E

Figure 9: Copy Shared Key

Click the **Authentication** section and copy the text from the **Primary Key** field.

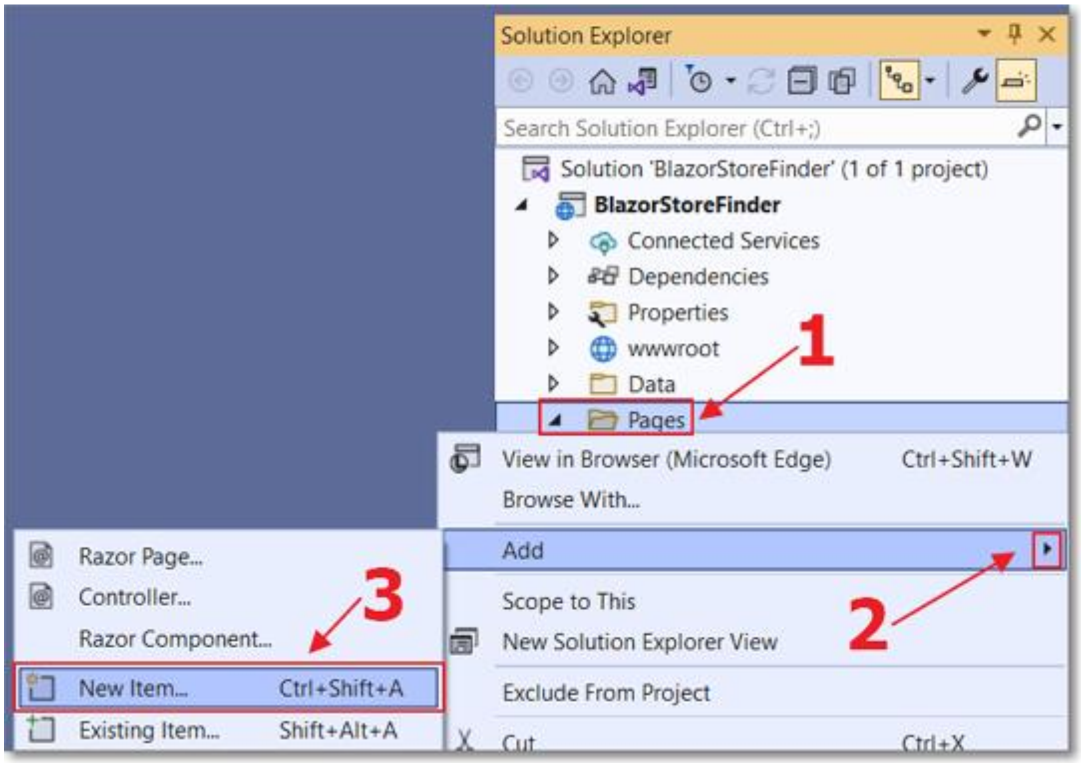


Figure 10: Add New Item

In Visual Studio, right-click the **Pages** folder and select **New Item**.

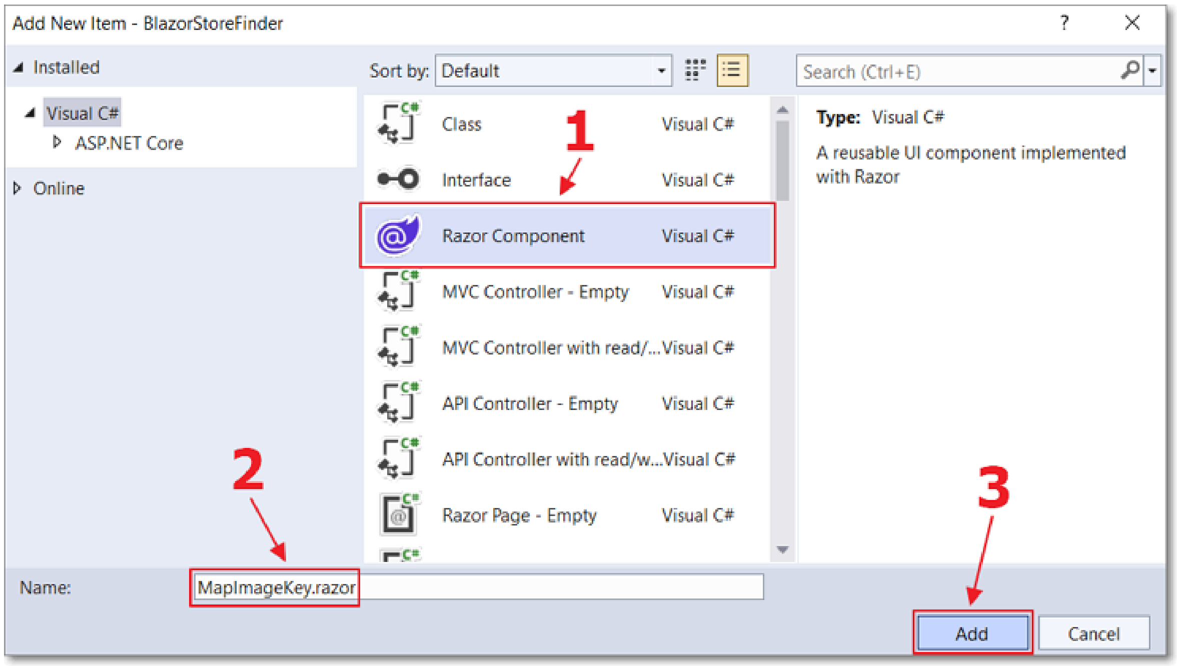

Add New Item - BlazorStoreFinder
Installed
Visual C#
ASP.NET Core
Online
Sort by: Default
Search (Ctrl+E)
Class Visual C#
Interface Visual C#
Razor Component Visual C#
MVC Controller - Empty Visual C#
MVC Controller with read/... Visual C#
API Controller - Empty Visual C#
API Controller with read/w... Visual C#
Razor Page - Empty Visual C#
Type: Visual C#
A reusable UI component implemented with Razor
Name: MapImageKey.razor
Add
Cancel
1
2
3

Figure 11: Create Razor Component

Create a Razor component named **MapImagekey.razor**.

Replace all the code with the following.

Code Listing 1: MapImageKey Header

```
@page "/mapimagekey"
@using System.Text;
```

This instructs the control to be loaded when a user navigates to the **`mapimagekey`** URL and adds the required using statements for the **`StringBuilder`** code that will be added later.

Next, add the following user interface code.

Code Listing 2: MapImageKey UI

```
<input type="text" placeholder="Latitude" @bind="Latitude" />
<input type="text" placeholder="Longitude" @bind="Longitude" />
<input type="text" placeholder="SubscriptionKey" @bind="SubscriptionKey" />
<br />
<br />
<button class="btn btn-success" @onclick="GetTile">Get Tile</button>
<br />
<br />
@if (isLoading)
{
    <div class="spinner-border text-primary" role="status"></div>
}
<br />
@if (PngImage != "")
{
    <div class="row">
        <div style="width: 300px; height: 300px;">
            <img src="@PngImage" height="300" width="300">
        </div>
    </div>
}
```

This creates text boxes for the user to enter a latitude, longitude, an Azure Maps shared key, and a **Get Tile** button to click to retrieve a map image.

If an image is retrieved (**`PngImage`**), it will be displayed in an **`img`** control.

Add the following additional code.

Code Listing 3: MapImageKey Variable Declarations

```
@code {
    string PngImage = string.Empty;
    bool isLoading = false;
    string SubscriptionKey = "";

    // Location of Los Angeles
    string Latitude = "34.0522";
    string Longitude = "-118.2437";
```

This sets the default latitude and longitude to Los Angeles.

Finally, add the following code to call the Azure Maps API and retrieve the tile image.

Code Listing 4: MapImageKey GetTile Method

```
    public async Task GetTile()
    {
        PngImage = "";
        isLoading = true;

        // Create an HTTP Client to make the REST call.
        using (var client = new System.Net.Http.HttpClient())
        {
            // Build the URL.
            StringBuilder sb = new StringBuilder();

            // Request a PNG image.
            sb.Append("https://atlas.microsoft.com/map/static/png?");
            // Pass the Subscription Key.
            sb.Append($"subscription-key={SubscriptionKey}&");
            // Specify the API version, layer type, and zoom level.
            sb.Append("api-version=1.0&layer=basic&style=main&zoom=10&");
            // Pass the Latitude and Longitude
            sb.Append($"&center={Longitude},%20{Latitude}");
            // Request that a pin be placed at the Latitude and
 Longitude.
            sb.Append($"&pins=default%7C%7C{Longitude}+{Latitude}");

            // Set the URL
            var url = new Uri(sb.ToString());

            // Call Azure Maps and get the response.
            var Response = await client.GetAsync(url);

            // Read the response.
            var responseContent =
            await Response.Content.ReadAsByteArrayAsync();

            // Convert the response to an image.
            PngImage =
            $"data:image/png;base64,{Convert.ToBase64String(responseContent)}";
```

```
                isLoading = false;
                StateHasChanged();
            }
        }
}
```

This passes the Azure Maps shared key (in the **`SubscriptionKey`** variable) to the Azure Maps API, along with the latitude and longitude, and requests a map tile for the location. It also requests that a map pin marker be placed at the latitude and longitude location.

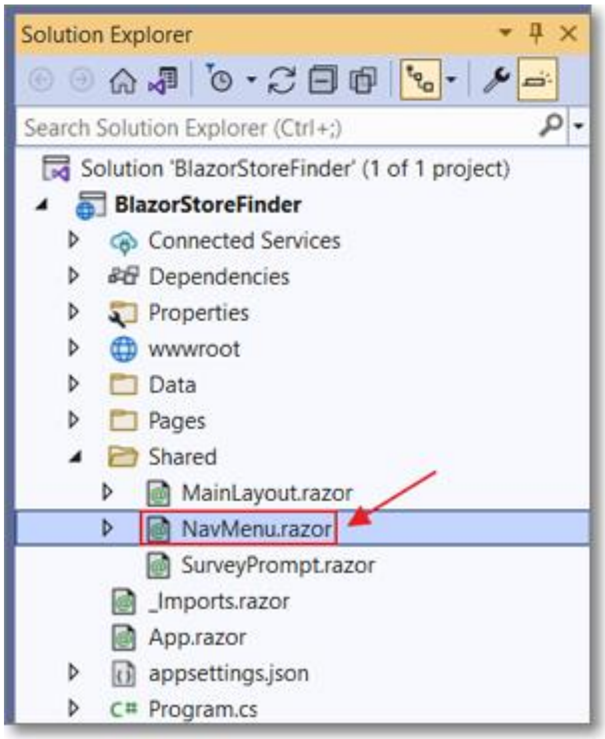


Figure 12: NavMenu.razor

We will now add a link to the **`MapImageKey`** control on the navigation menu. Open the **NavMenu.razor** control and add the following code under the existing code for the **`Home`** link.

Code Listing 5: MapImageKey NavLink

```
    <div class="nav-item px-3">
        <NavLink class="nav-link" href="mapimagekey">
            <span class="oi oi-map" aria-hidden="true"></span> Map (Key)
        </NavLink>
    </div>
```

On the Visual Studio toolbar, select **Debug** > **Start Debugging** (**F5**) to run the project.

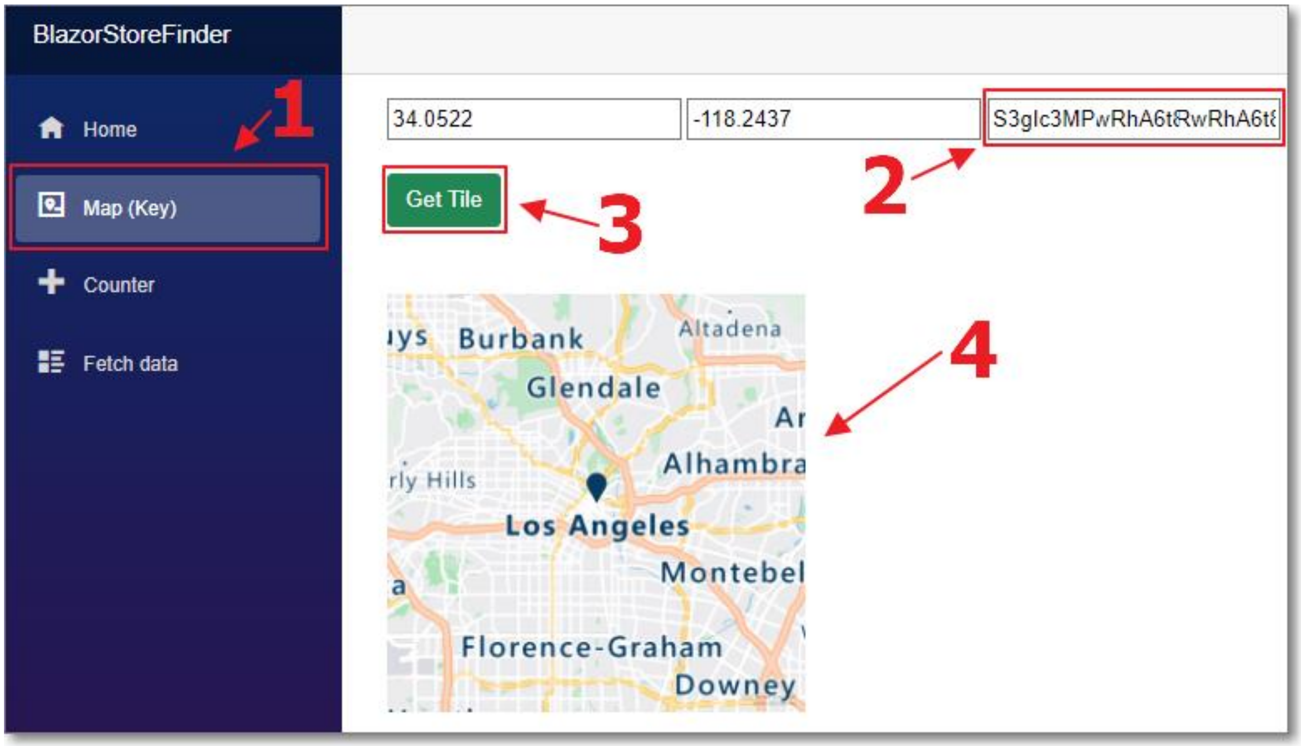


Figure 13: Display Image Using Shared Key

When the project opens up in your web browser, click the **Map (Key)** link to navigate to the **MapImageKey.razor** control.

Enter the shared key (that you copied from the portal.azure.com page earlier) in the box labeled **SubscriptionKey** and click **Get Tile**.

The map tile will be displayed. You can enter different latitude and longitude values in the first two text boxes and click **Get Tile** to retrieve different map tile images.

Render image using AAD token

We will now retrieve the same image using AAD authentication. The setup is more involved; however, we will be able to leverage the authentication code for the remaining examples in this book.

We will create an Azure service principal that will be assigned to the Azure Maps Data Reader role in the Azure Maps account we created earlier. The Azure documentation states that "the security principal defines the access policy and permissions for the user/application in the Azure AD tenant." Later, in the Blazor application, we will request an authentication token for the Azure service principal. We will then use this authentication token to make calls to the Azure Maps API.

Create the service principal

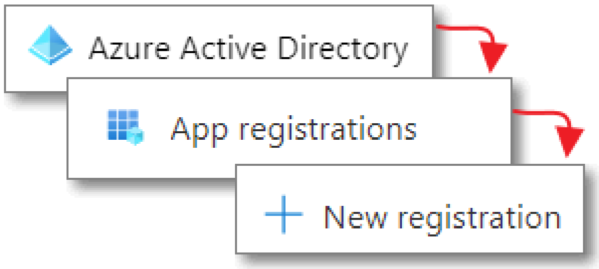
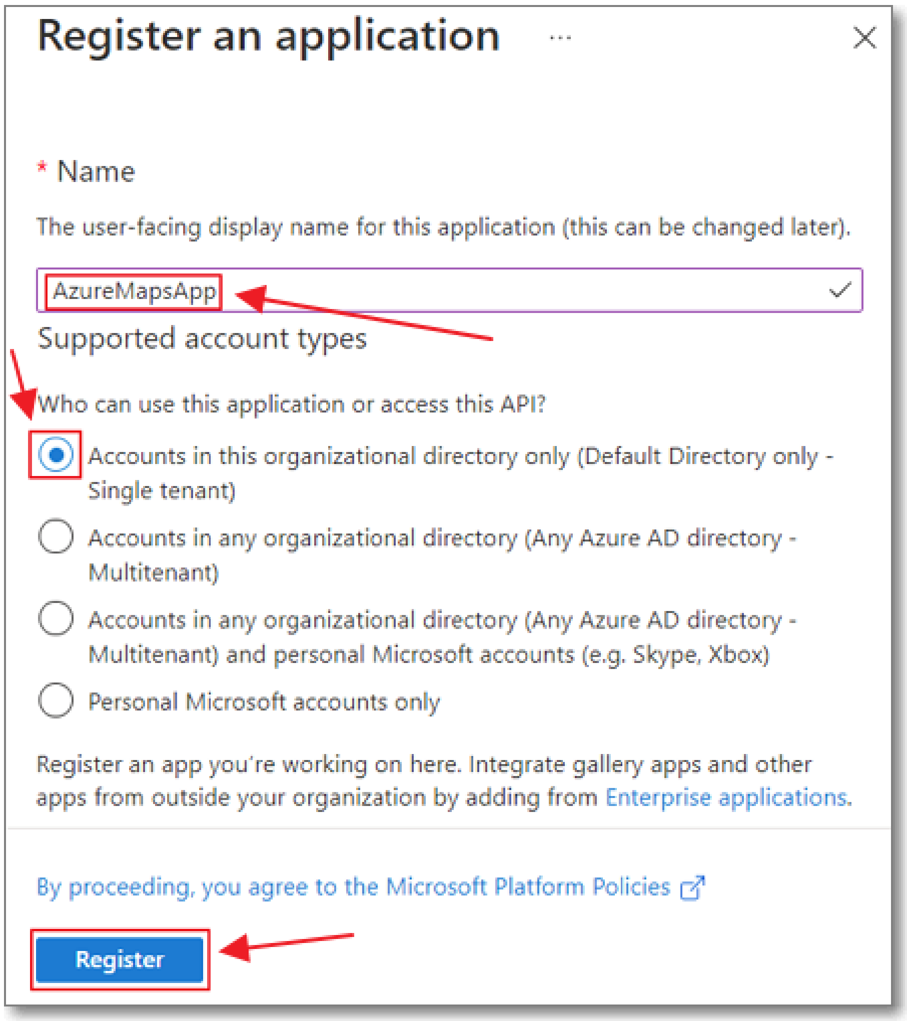


Figure 14: New Registration

In the Azure portal, navigate to the **Azure Active Directory** section and select **App registrations** > **New registration**.


Register an application

* Name

The user-facing display name for this application (this can be changed later).

AzureMapsApp

Supported account types

Who can use this application or access this API?

[✓ Accounts in this organizational directory only (Default Directory only - Single tenant)]

Accounts in any organizational directory (Any Azure AD directory - Multitenant)

Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)

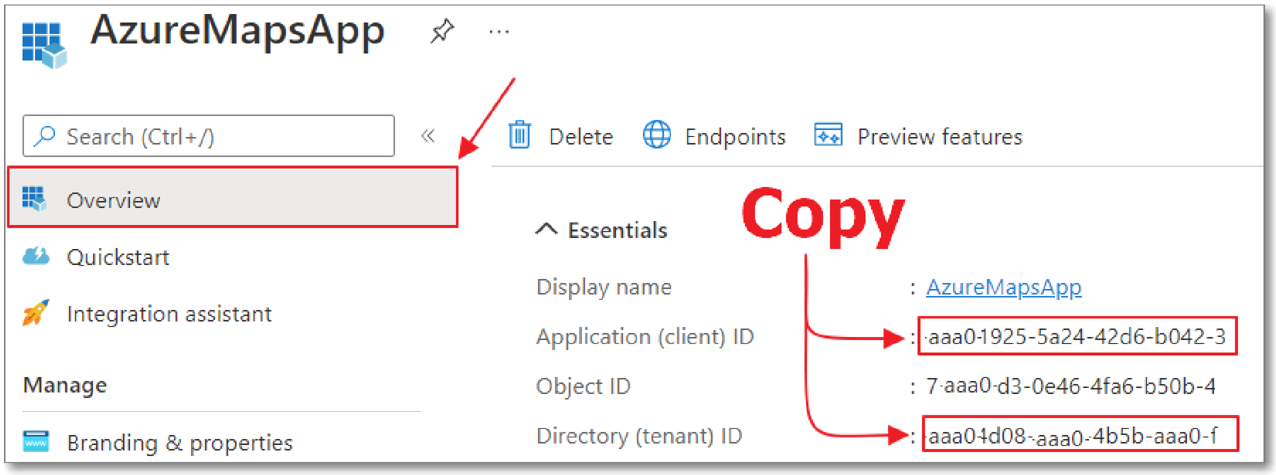
Personal Microsoft accounts only

Register an app you're working on here. Integrate gallery apps and other apps from outside your organization by adding from Enterprise applications.

By proceeding, you agree to the Microsoft Platform Policies

Figure 15: Register AzureMapsApp

Name it **AzureMapsApp**, select **Accounts in this organizational directory only**, and click **Register**.



Display name : AzureMapsApp
Application (client) ID : aaa0-1925-5a24-42d6-b042-3
Object ID : 7-aaa0-d3-0e46-4fa6-b50b-4
Directory (tenant) ID : aaa0-1d08-aaa0-4b5b-aaa0-f

Figure 16: Copy Client ID and Tenant ID

Copy and save the **Client ID** and **Tenant ID**, because you will need them in a later step.

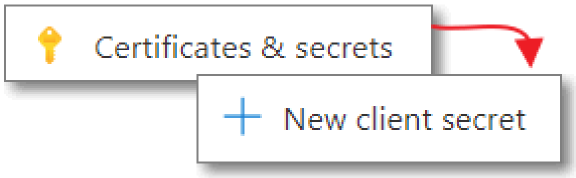
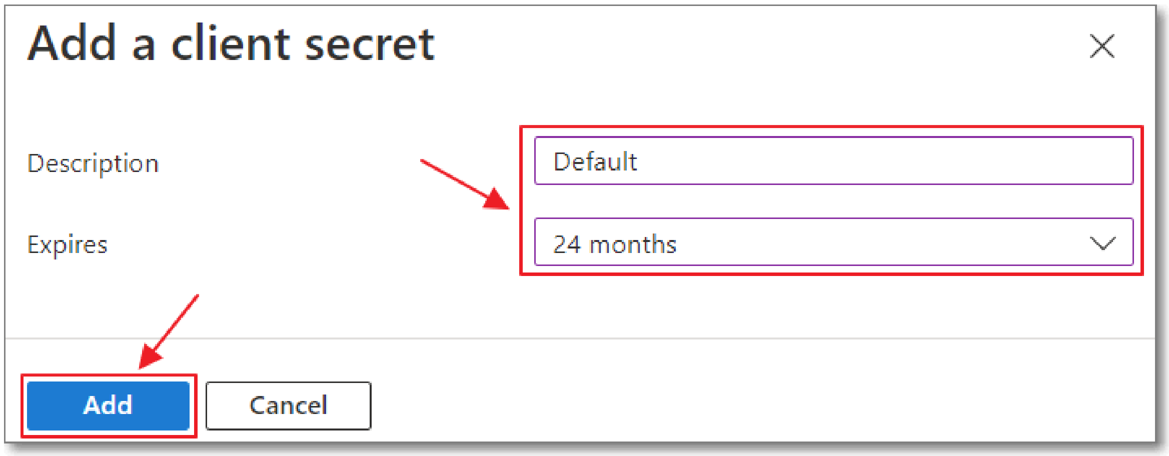


Figure 17: Certificates & Secrets

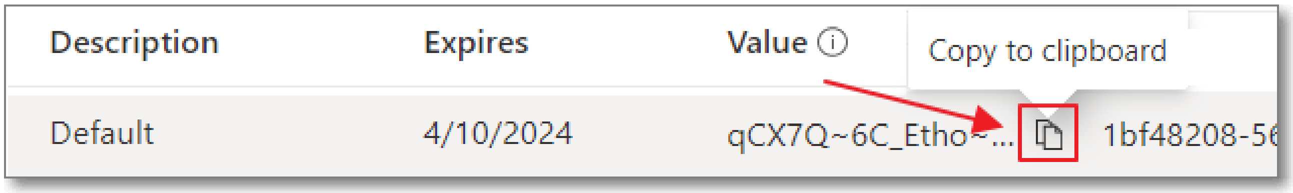
Click **Certificates & secrets** to navigate to that section. Then, click **New client secret**.



Description: Default
Expires: 24 months

Figure 18: Create Client Secret

Give it a name and an expiration date. Click **Add** to create the secret.



Description	Expires	Value	Copy to clipboard
Default	4/10/2024	qCX7Q~6C_Etho~...	1bf48208-56

Figure 19: Copy Secret Value

After the **Client Secret** is created, click **Copy to clipboard** next to the value and save it (you will need it in a later step).

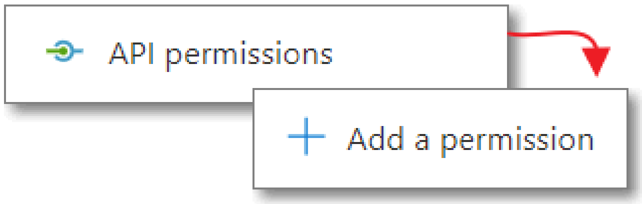
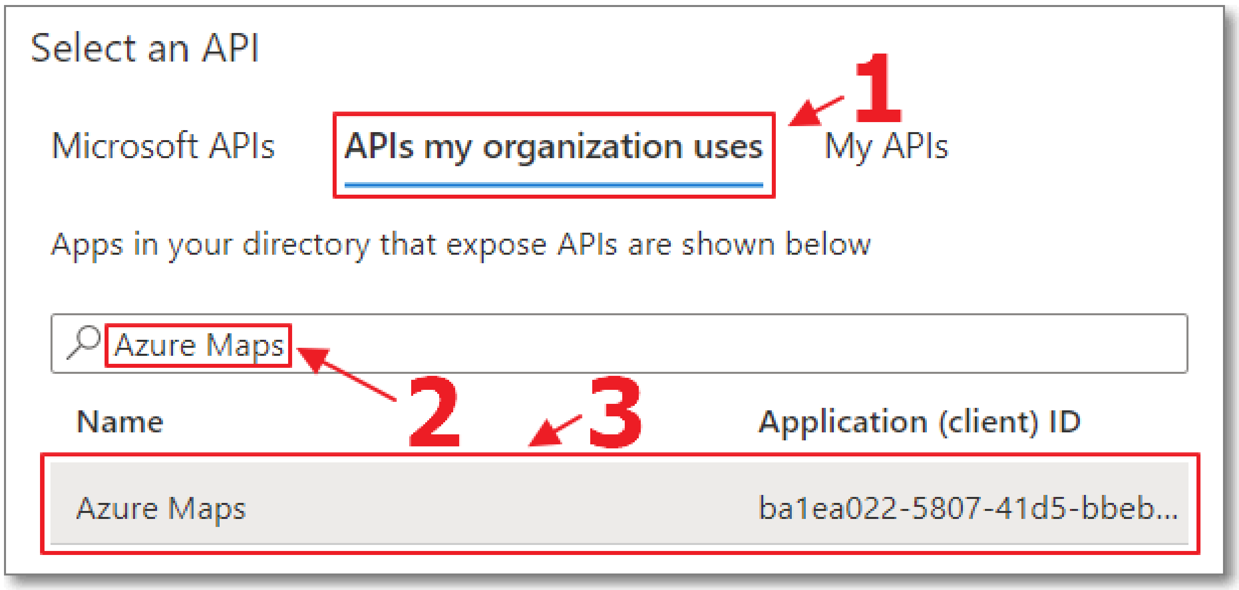


Figure 20: Add a Permission

Click **API permissions** to navigate to that section and click **Add a permission**.


Select an API
Microsoft APIs
APIs my organization uses
1
My APIs
Apps in your directory that expose APIs are shown below
Azure Maps
2
3

Name	Application (client) ID
Azure Maps	ba1ea022-5807-41d5-bbeb...

Figure 21: Select Azure Maps API

Select **APIs my organization uses**, search for **Azure Maps**, and click it to select it.

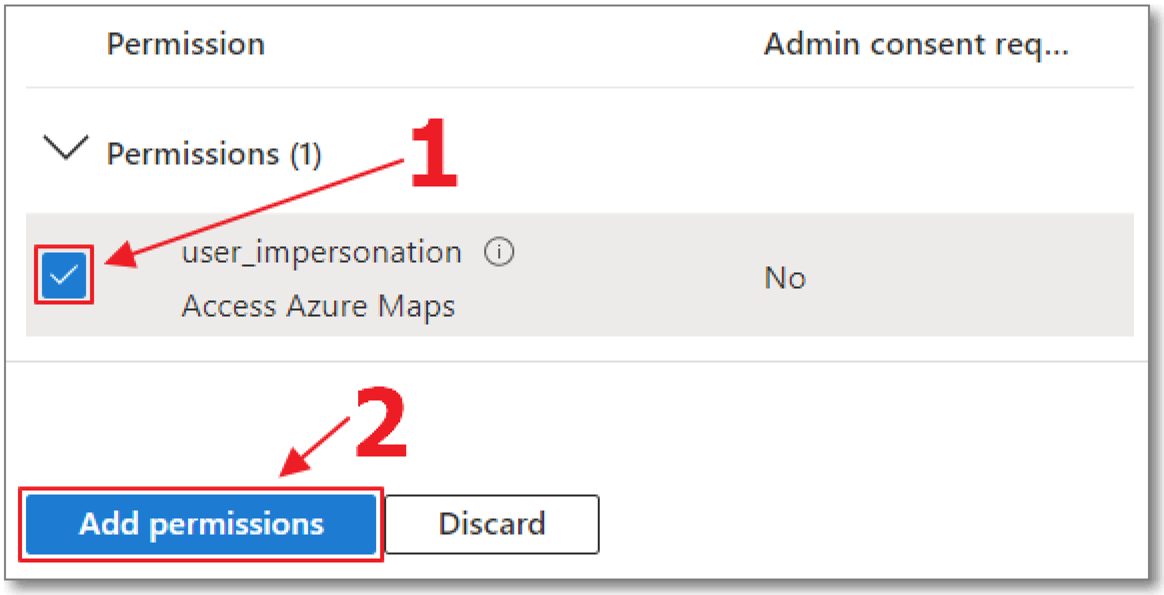


Figure 22: Add Permissions

Check the box next to **Access Azure Maps** and click **Add permissions**.

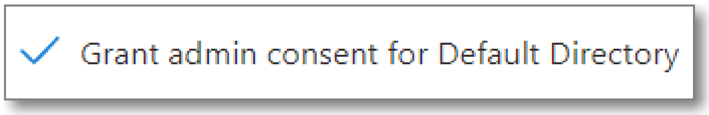


Figure 23: Grant Admin Consent Button

Finally, click the **Grant admin consent** button.

Assign the service principal to the Azure Maps account

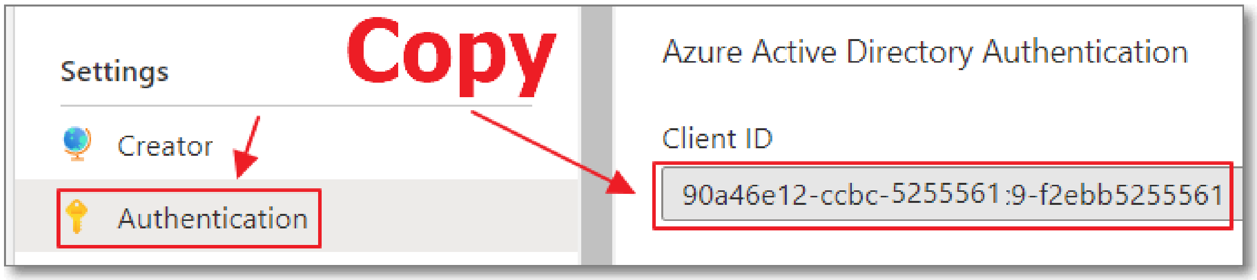

Settings
Copy
Creator
Authentication
Azure Active Directory Authentication
Client ID
90a46e12-ccbc-5255561:9-f2ebb5255561

Figure 24: Copy Azure Maps Client ID

Return to the Azure Maps account created earlier, select the **Authentication** section, and copy and save the **Client ID** (you will need it in a later step).

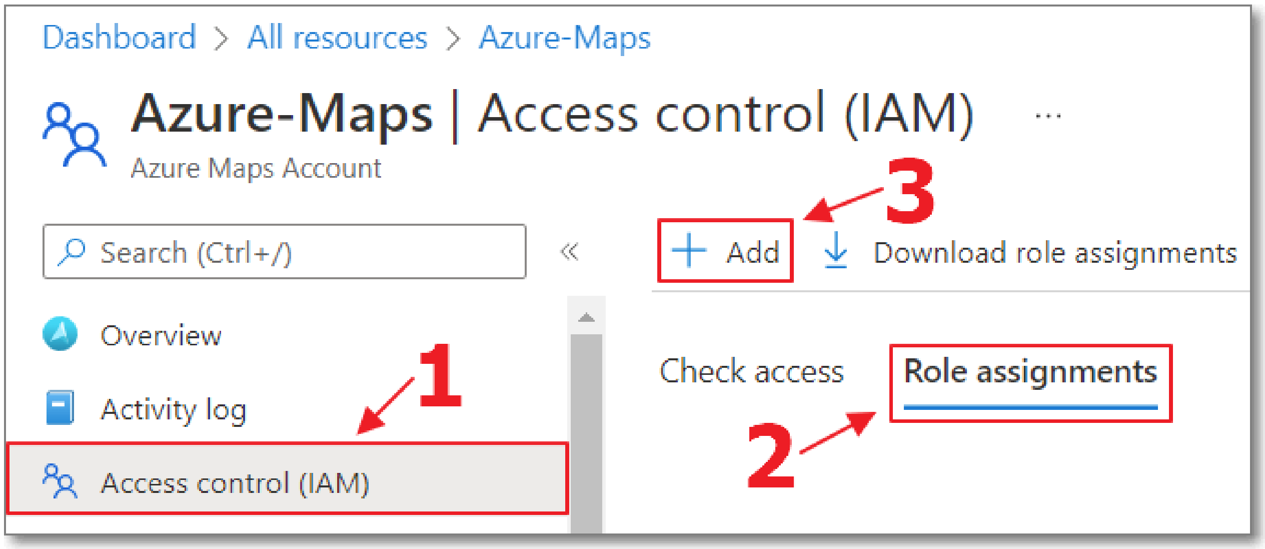
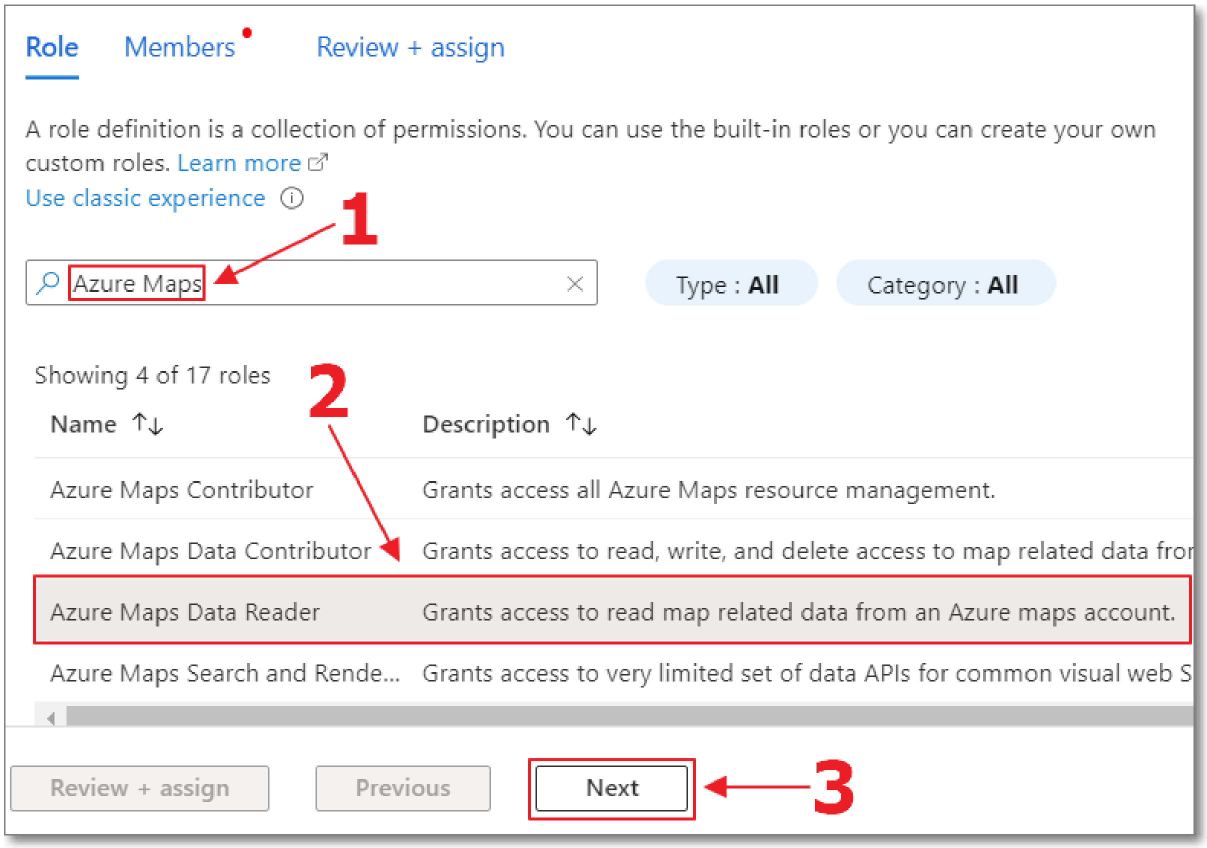


Figure 25: Add Role Assignment

Select **Access control (IAM)** > **Role assignments**. Click **Add** and select **Add role assignment**.



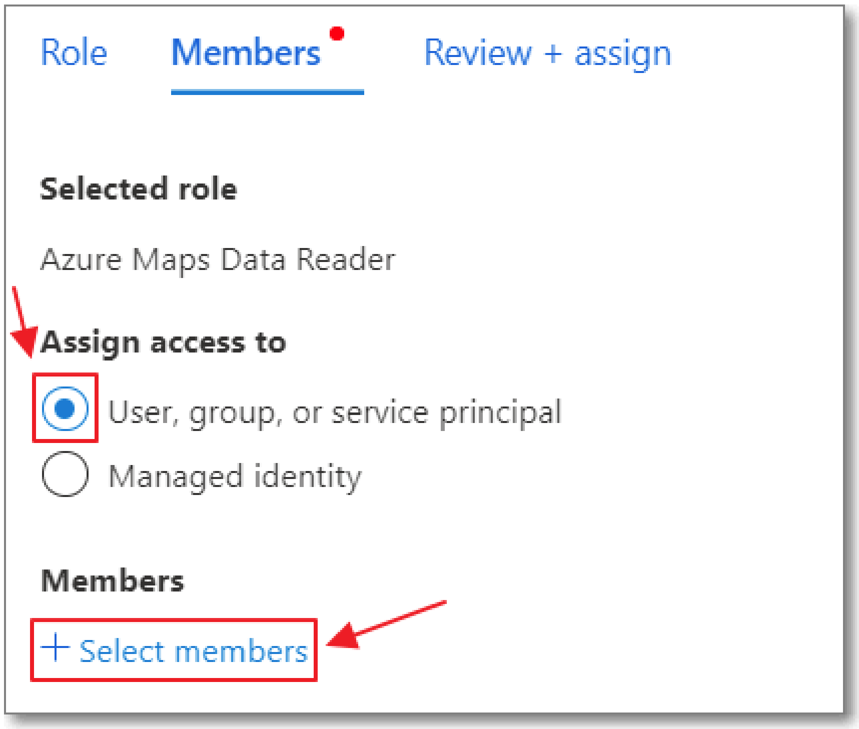
A role definition is a collection of permissions. You can use the built-in roles or you can create your own custom roles.

Showing 4 of 17 roles

Name	Description
Azure Maps Contributor	Grants access all Azure Maps resource management.
Azure Maps Data Contributor	Grants access to read, write, and delete access to map related data fro
Azure Maps Data Reader	Grants access to read map related data from an Azure maps account.
Azure Maps Search and Rende...	Grants access to very limited set of data APIs for common visual web S

Figure 26: Assign Azure Maps Data Reader

Search for **Azure Maps**, then select the **Azure Maps Data Reader** role, and click **Next**.



Selected role

Azure Maps Data Reader

Assign access to

[✓ User, group, or service principal]

Managed identity

Members

+ Select members

Figure 27: Select Members

Under **Assign access to**, select **User, group, or service principal**. Then click **Select members**.

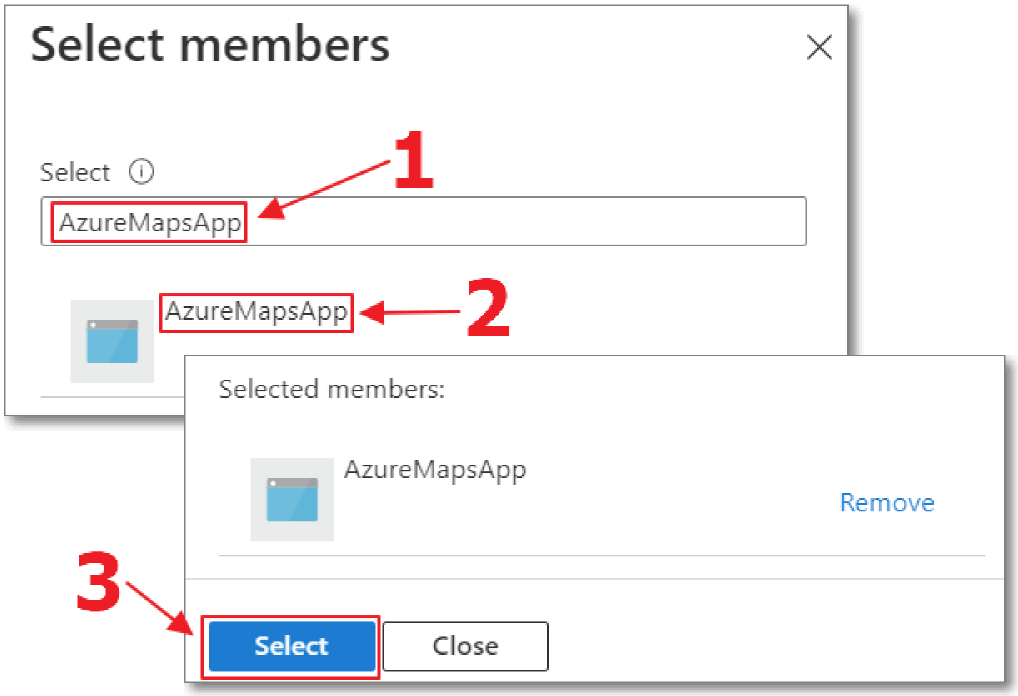
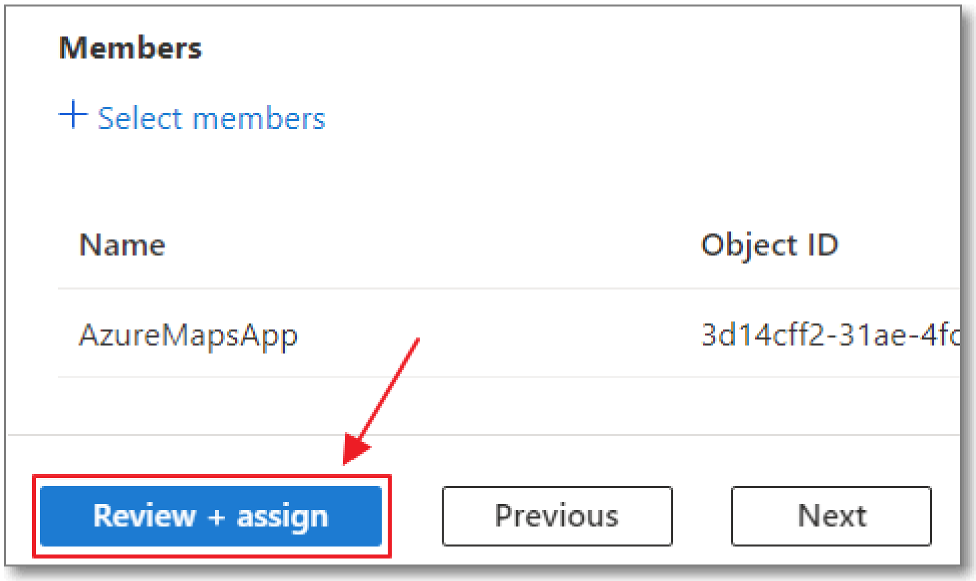


Figure 28: Select AzureMapsApp

Search for the name of the Azure app registration, created earlier, and select it. Then, click **Select**.



Name	Object ID
AzureMapsApp	3d14cff2-31ae-4fc

Figure 29: Review and Assign

Click **Review + assign**, then on the next screen, click **Review + assign** again.

Store and retrieve the service principal settings

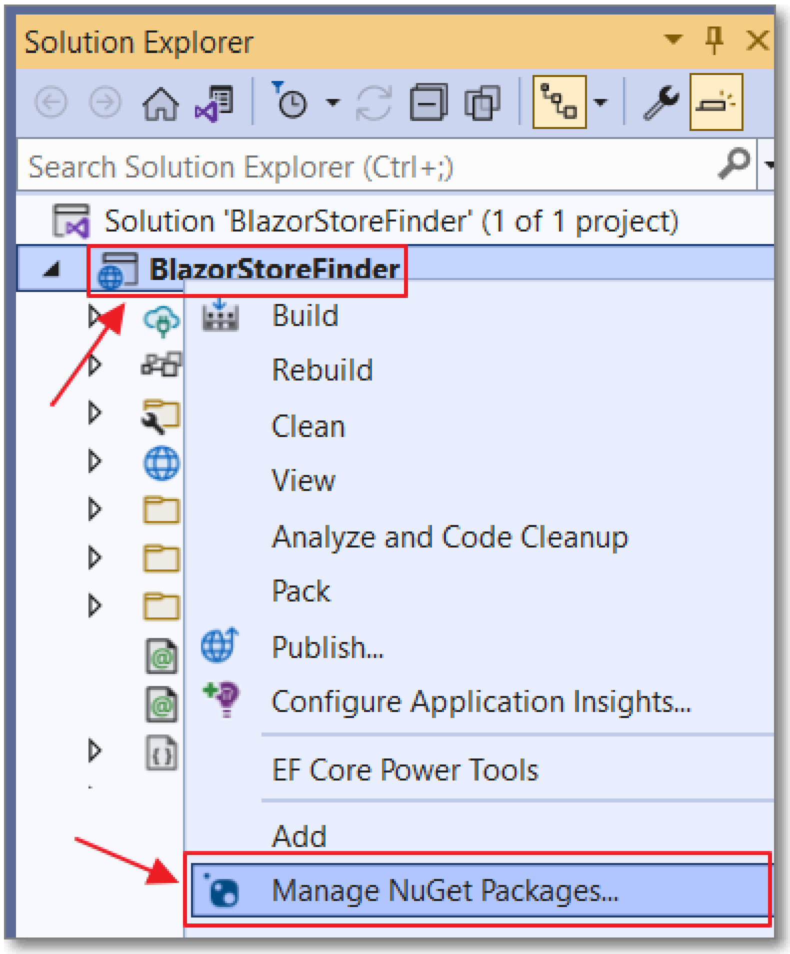


Figure 30: Manage NuGet Packages

In Visual Studio, right-click the project node and select **Manage NuGet Packages**.

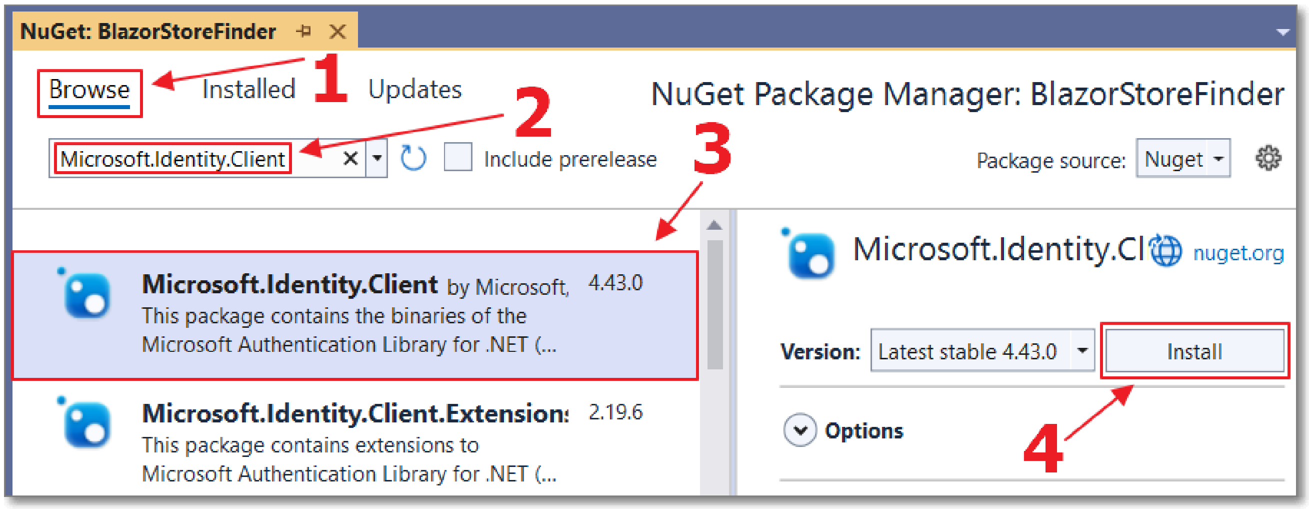

NuGet: BlazorStoreFinder
Browse
Installed
Updates
1
2
3
4
NuGet Package Manager: BlazorStoreFinder
Microsoft.Identity.Client
Include prerelease
Package source: Nuget
Microsoft.Identity.Client by Microsoft, 4.43.0
This package contains the binaries of the Microsoft Authentication Library for .NET (...
Microsoft.Identity.Client.Extension: 2.19.6
This package contains extensions to Microsoft Authentication Library for .NET (...
Microsoft.Identity.Cl nuget.org
Version: Latest stable 4.43.0
Install
Options

Figure 31: Install Microsoft.Identity.Client

Install the **Microsoft.Identity.Client** NuGet package.

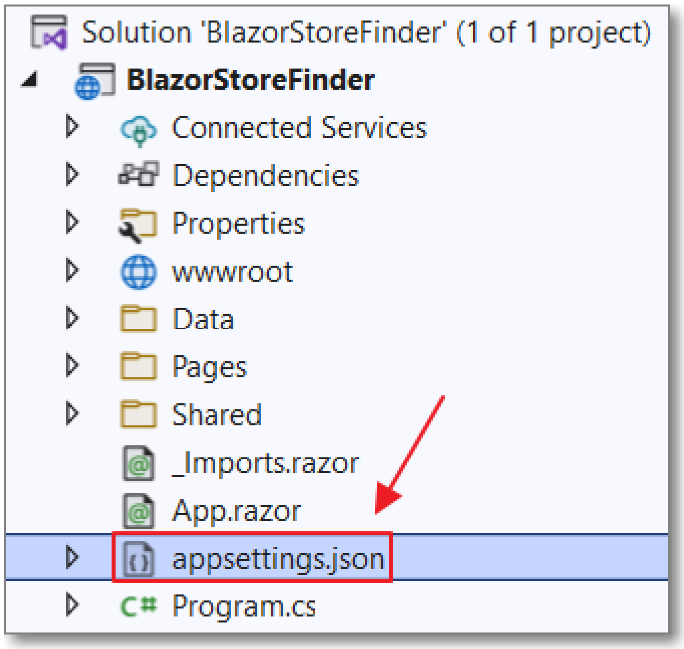


Figure 32: Open appsettings.json

Next, we will store the required values, gathered earlier, in the application settings file. Open the **appsettings.json** file and replace all the contents with the following (replacing the highlighted yellow sections with your own values).

Code Listing 6: appsettings.json

```
{
  "ConnectionStrings": {
    "DefaultConnection": "** To be filled in later **"
  },
  "AzureMaps": {
    "ClientId": "{{ Client ID from Azure Maps Account }}",
    "AadTenant": "{{ Azure Tenant ID }}",
    "AadAppId": "{{ Client ID from Azure App Registration }}",
    "AppKey": "{{ AppKey (secret) from Azure App Registration }}"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Note: The `ClientId` from the Azure Maps account will be consumed in the "Displaying Azure Maps" section covered later.

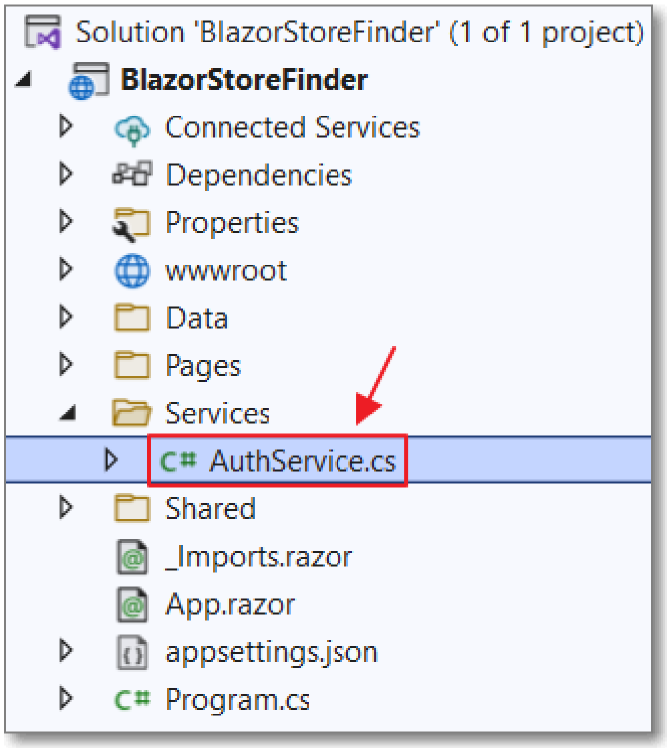


Figure 33: AuthService.cs

Create a **Services** directory and a **`AuthService.cs`** static class using the following code.

Code Listing 7: AuthService

```
using Microsoft.Identity.Client;
using Microsoft.JSInterop;

namespace BlazorStoreFinder
{
    public static class AuthService
    {
        private const string AuthorityFormat =
            "https://login.microsoftonline.com/{0}/oauth2/v2.0";

        private const string MSGraphScope =
            "https://atlas.microsoft.com/.default";

        public static string? ClientId;
        public static string? AadTenant;
        public static string? AadAppId;
        public static string? AppKey;

        internal static void SetAuthSettings(IConfigurationSection AzureMaps)
        {
            // A call in Program.cs to this method is made to set the values.
            ClientId = AzureMaps.GetValue<string>("ClientId");
            AadTenant = AzureMaps.GetValue<string>("AadTenant");
            AadAppId = AzureMaps.GetValue<string>("AadAppId");
            AppKey = AzureMaps.GetValue<string>("AppKey");
        }

        [JSInvokable]
        public static async Task<string> GetAccessToken()
        {
            // Create a confidential client.
            IConfidentialClientApplication daemonClient;

            // Create a builder for the confidential client.
            daemonClient = ConfidentialClientApplicationBuilder
                .Create(AadAppId)
                .WithAuthority(string.Format(AuthorityFormat, AadTenant))
                .WithClientSecret(AppKey)
                .Build();

            // Get the access token for the confidential client.
            AuthenticationResult authResult =
            await daemonClient.AcquireTokenForClient(new[] { MSGraphScope })
            .ExecuteAsync();

            // Return the access token.
            return authResult.AccessToken;
        }
    }
}
```

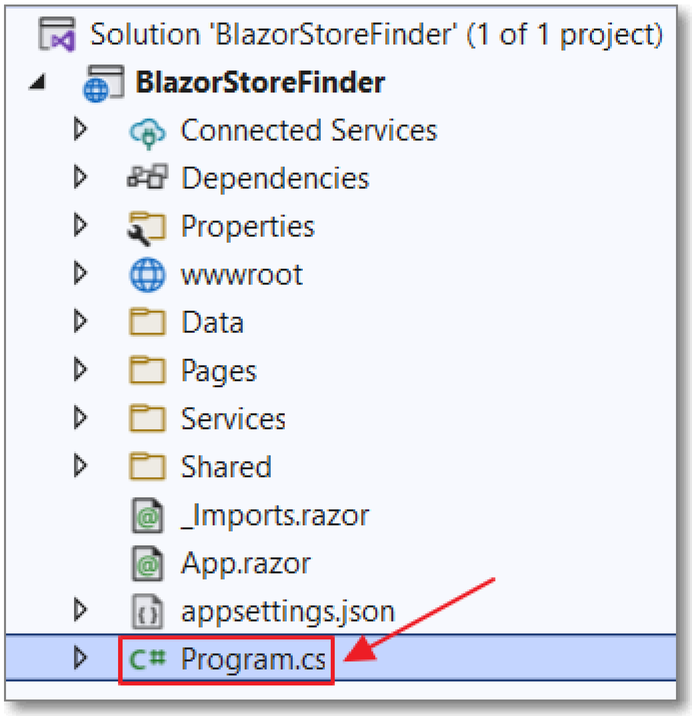


Figure 34: Update Program.cs

Next, open the **Program.cs** file and replace the line of code:

`var app = builder.Build();`

With the following code that will get the settings from the **appsettings.json**, call the original **`Build()`** code and initialize the static **`AuthService`** class with the settings from the **appsettings.json** file.

Code Listing 8: AzureMaps Settings

```
// Get the Azure Maps settings from appsettings.json.
var AzureMaps = builder.Configuration.GetSection("AzureMaps");

// Build the application and return the startup type.
var app = builder.Build();

// Initialize the AuthService so the later calls to
// GetAccessToken will work.
BlazorStoreFinder.AuthService.SetAuthSettings(AzureMaps);
```

Display the tile

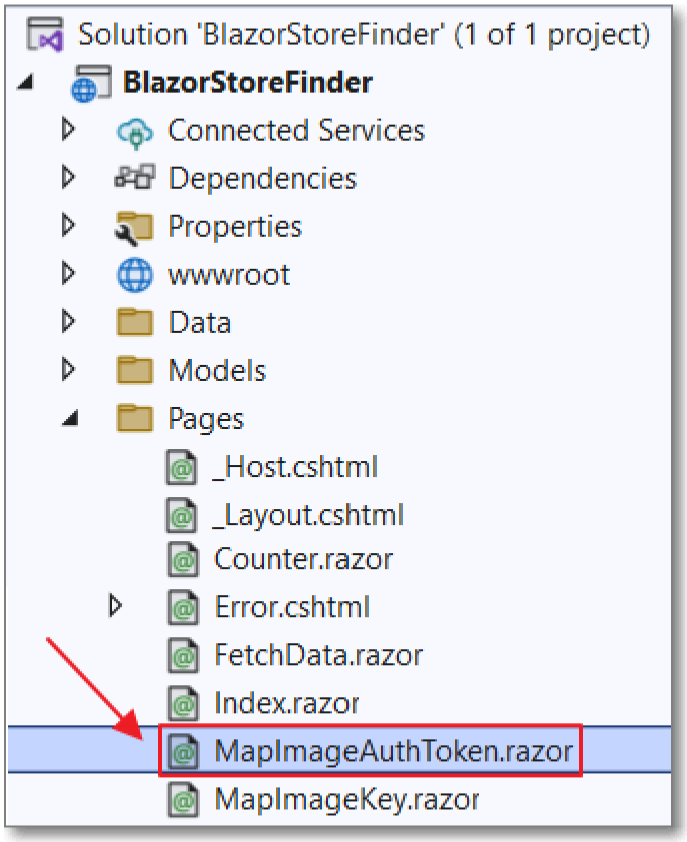


Figure 35: Create MapImageAuthToken.razor

Create a new Razor control called **`MapImageAuthToken.razor`** using the following code.

Code Listing 9: MapImageAuthToken.razor

```
@page "/mapimageauthtoken"
@using System.Text;

<div class="row">
    <div style="width: 300px; height: 300px;">
        <img src="@PngImage" height="300" width="300">
    </div>
</div>

@code {
    string PngImage = string.Empty;
    string Latitude = "";
    string Longitude = "";

    protected override async Task OnAfterRenderAsync(bool firstRender)
```

```
{
    if (firstRender)
    {
        Latitude = Convert.ToDouble("34.0522").ToString();
        Longitude = Convert.ToDouble("-118.2437").ToString();

        // Call GetTile() method to get the PNG image.
        PngImage = await GetTile();

        StateHasChanged();
    }
}

public async Task<string>
GetTile()
{
    String PngImage = "";

    // Create a HTTP client to make the REST call.
    using (var client = new System.Net.Http.HttpClient())
    {
        // Get an access token from AuthService.
        var AccessToken = await AuthService.GetAccessToken();

        client.DefaultRequestHeaders.Accept.Clear();
        // Pass the Azure Maps Client Id.
        client.DefaultRequestHeaders.Add("x-ms-client-id",
        AuthService.ClientId);

        // Pass the access token in the auth header.
        client.DefaultRequestHeaders.Authorization =
        new System.Net.Http.Headers
        .AuthenticationHeaderValue("Bearer", AccessToken);

        // Build the URL.
        StringBuilder sb = new StringBuilder();

        // Request a PNG image.
        sb.Append("https://atlas.microsoft.com/map/static/png?");
        // Specify the API version, layer type, and zoom level.
        sb.Append("api-version=1.0&layer=basic&style=main&zoom=12&");
        // Pass the latitude and longitude.
        sb.Append($"&center={Longitude},%20{Latitude}");
        // Request that a pin be placed at the latitude and longitude.
        sb.Append($"&pins=default%7C%7C{Longitude}+{Latitude}");

        // Set the URL.
        var url = new Uri(sb.ToString());
```

```
            // Call Azure Maps and get the response.
            var Response = await client.GetAsync(url);

            // Read the response.
            var responseContent = await Response.Content.ReadAsByteArrayAsync();

            // Convert the response to an image.
            PngImage =
           $"data:image/png;base64,{Convert.ToBase64String(responseContent)}";
        }

        return PngImage;
    }
}
```

Add the following code to **NavMenu.razor** to add a link to the control.

Code Listing 10: Map (Token) Link

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="mapimageauthtoken">
        <span class="oi oi-map-marker" aria-hidden="true"></span> Map (Token)
    </NavLink>
</div>
```

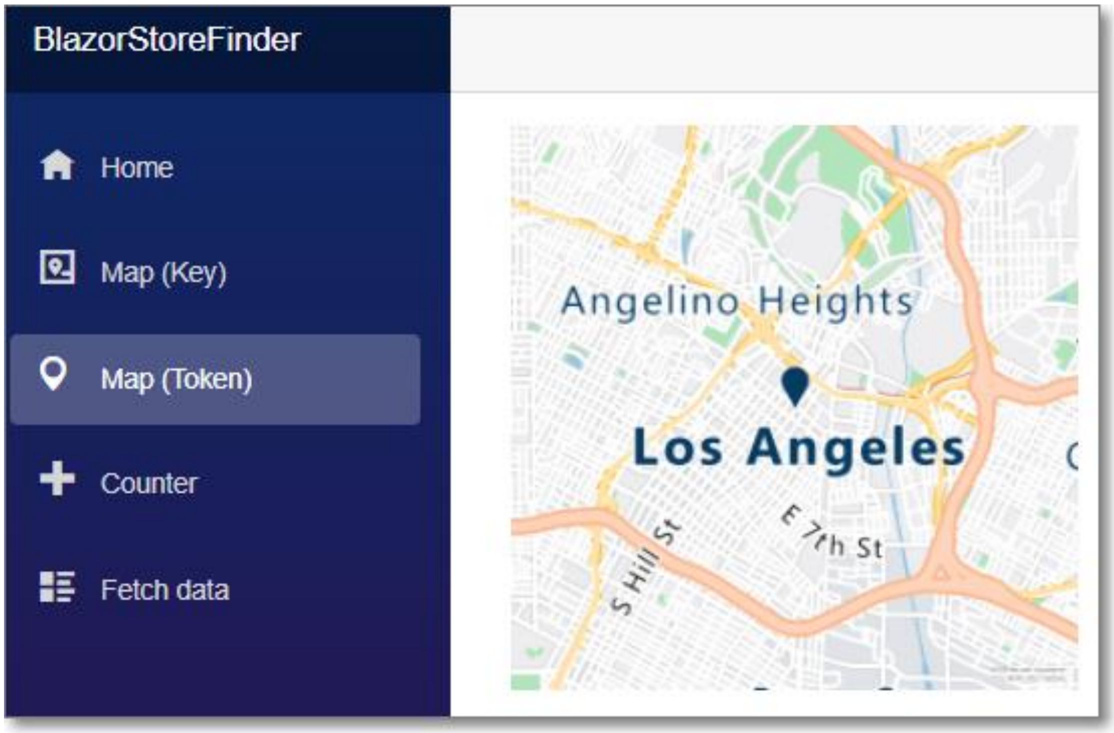


Figure 36: Tile Using Auth Token

Run the application, click the **Map (Token)** link, and view the result.

Retrieve your current location

We will now update the code to determine automatically the user's current latitude and longitude, and display a map tile of that location.

Install the following NuGet package: **Darnton.Blazor.DeviceInterop**.

Open the **Program.cs** file and add the following code (before the **`var app = builder.Build()`** line).

Code Listing 11: Add Darnton.Blazor.DeviceInterop

```
// Configure the Darnton Geolocation component
builder.Services
.AddScoped<
Darnton.Blazor.DeviceInterop.Geolocation.IGeolocationService,
Darnton.Blazor.DeviceInterop.Geolocation.GeolocationService
>();
```

Return to the **`MapImageAuthToken.razor`** control and add the following lines of code under the **`@using System.Text`** line.

Code Listing 12: Darnton Using and Inject Statements

```
@using Darnton.Blazor.DeviceInterop.Geolocation;
@inject IGeolocationService GeolocationService
```

Next, alter all the UI markup code to the following.

Code Listing 13: Darnton Markup

```
@if (CurrentPositionResult != null)
{
    <div class="row">
        <div style="width: 300px; height: 300px;">
            <img src="@PngImage" height="300" width="300">
        </div>
    </div>
}
else
{
    @if (isLoading)
    {
        <div class="spinner-border text-primary" role="status"></div>
    }
}
```

Add the following fields to the **`@code`** section.

Code Listing 14: Darnton Fields

```
    bool isLoading = true;
    protected GeolocationResult? CurrentPositionResult { get; set; }
```

Finally, change the entire **`OnAfterRenderAsync`** method to the following.

Code Listing 15: Darnton OnAfterRenderAsync Method

```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        // Get current location
        // will cause a popup to show to ask permission.
        CurrentPositionResult = await GeolocationService.GetCurrentPosition();

        if (CurrentPositionResult.IsSuccess)
        {
            // Get latitude and longitude.
            string? CurrentLatitude =
            CurrentPositionResult.Position?.Coords?.Latitude.ToString("F2");

            string? CurrentLongitude =
            CurrentPositionResult.Position?.Coords?.Longitude.ToString("F2");

            // Set latitude and longitude
            // (to be consumed by GetTile() method).
            if (CurrentLatitude != null && CurrentLongitude != null)
            {
                Latitude = Convert.ToDouble(CurrentLatitude).ToString();
                Longitude = Convert.ToDouble(CurrentLongitude).ToString();
            }

            // Call GetTile() method to get the PNG image.
            PngImage = await GetTile();

            isLoading = false;
            StateHasChanged();
        }
    }
}
```

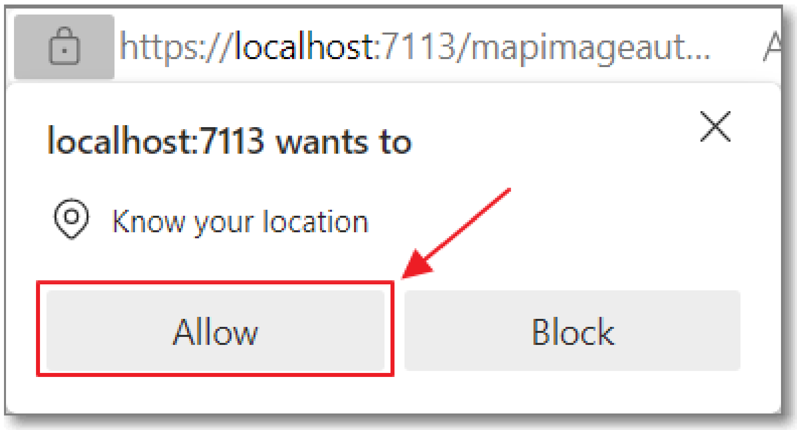


Figure 37: Allow Location Pop-up

Run the application and navigate to the **Map (Token)** page. A pop-up will appear asking for permission to obtain your current location. Click **Allow**.

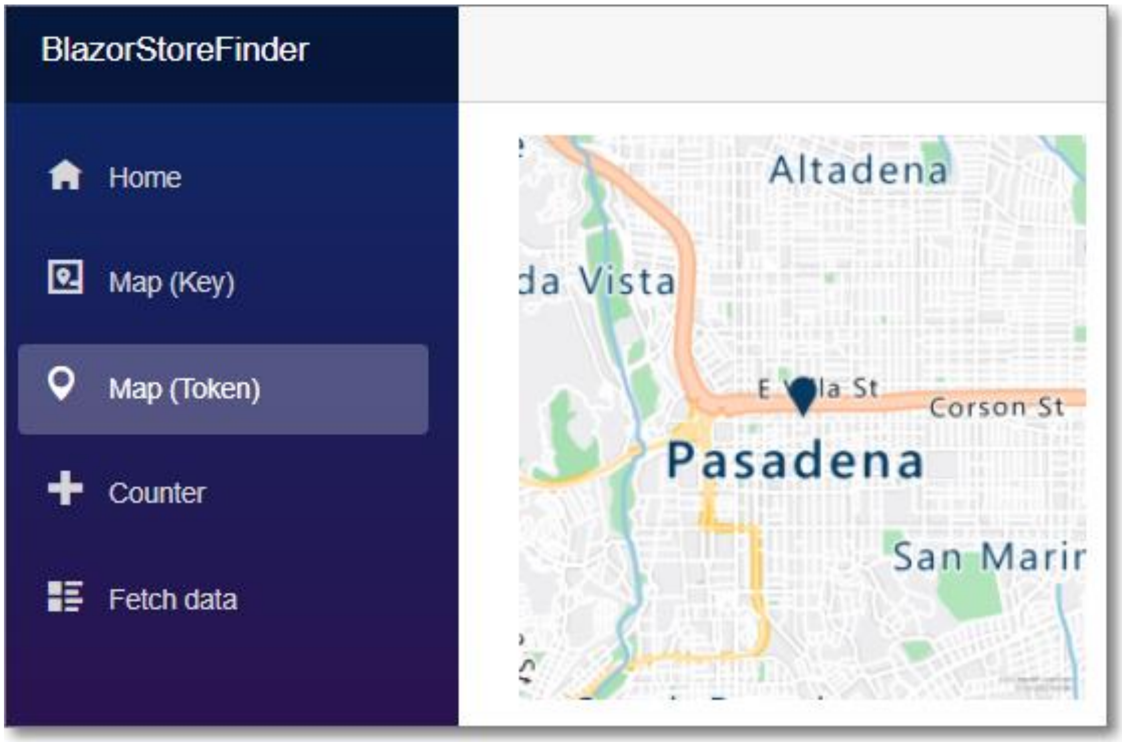


Figure 38: Tile of Your Current Location

A map tile of your current location will be displayed.

Render map using AzureMapsControl.Components

In the final example for this chapter, we will display an interactive Azure map. To do this, we will use the open source **`AzureMapsControl.Components`** control. This is a Microsoft Blazor component that wraps the Azure Maps JavaScript control in an API that allows programmatic interaction using C# rather than JavaScript.

Install the following NuGet package: **AzureMapsControl.Components**.

Open the **Program.cs** file and add the following **`using`** statement.

Code Listing 16: AzureMapsControl Using Statement

```
using AzureMapsControl.Components;
```

Next, add the following code (before the **`var app = builder.Build()`** line).

Code Listing 17: AddAzureMapsControl Configuration

```
// This code configures anonymous authentication
// for the Azure Maps API.
// An auth token will be required to access the maps.
builder.Services.AddAzureMapsControl(
    configuration => configuration.ClientId =
    AzureMaps.GetValue<string>("ClientId"));
```

The preceding code passes the **`ClientId`** from the Azure Maps account, stored in the appsettings.json file, to the Blazor **`AzureMapsControl`** control that will be used to display the map.

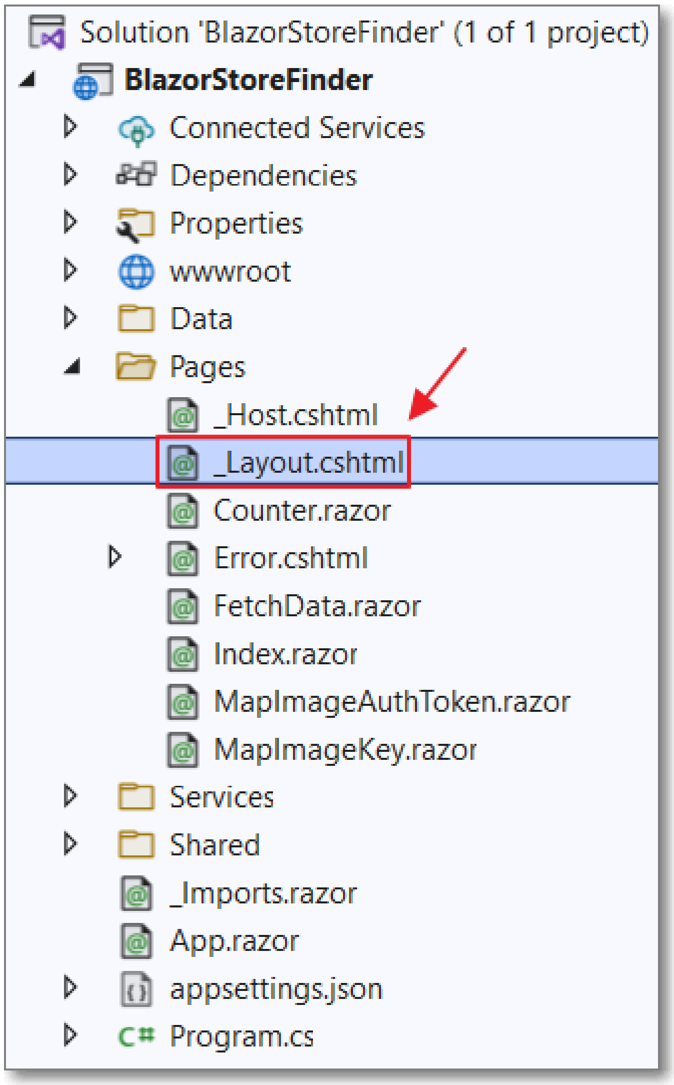


Figure 39: AzureMapsControl Layout Page Configuration

Open the **Pages/_Layout.cshtml** file and before this line:

`<component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />`

add the following code to the **`head`** section.

Code Listing 18: AzureMapsControl Stylesheet Code

```
    <link rel="stylesheet"

href="https://atlas.microsoft.com/sdk/javascript/mapcontrol/2/atlas.min.css
"
    type="text/css" />
    <link rel="stylesheet"
    href="https://atlas.microsoft.com/sdk/javascript/drawing/0.1/atlas-
drawing.min.css"
    type="text/css" />
    <link rel="stylesheet"
    href="https://atlas.microsoft.com/sdk/javascript/indoor/0.1/atlas-
indoor.min.css"
    type="text/css" />
    <style>
        #map {
            position: absolute;
            width: 60%;
            min-width: 300px;
            height: 60%;
            min-height: 250px;
        }
    </style>
```

Finally, add the following code before the closing **body** tag on the page.

Code Listing 19: AzureMapsControl JavaScript Code

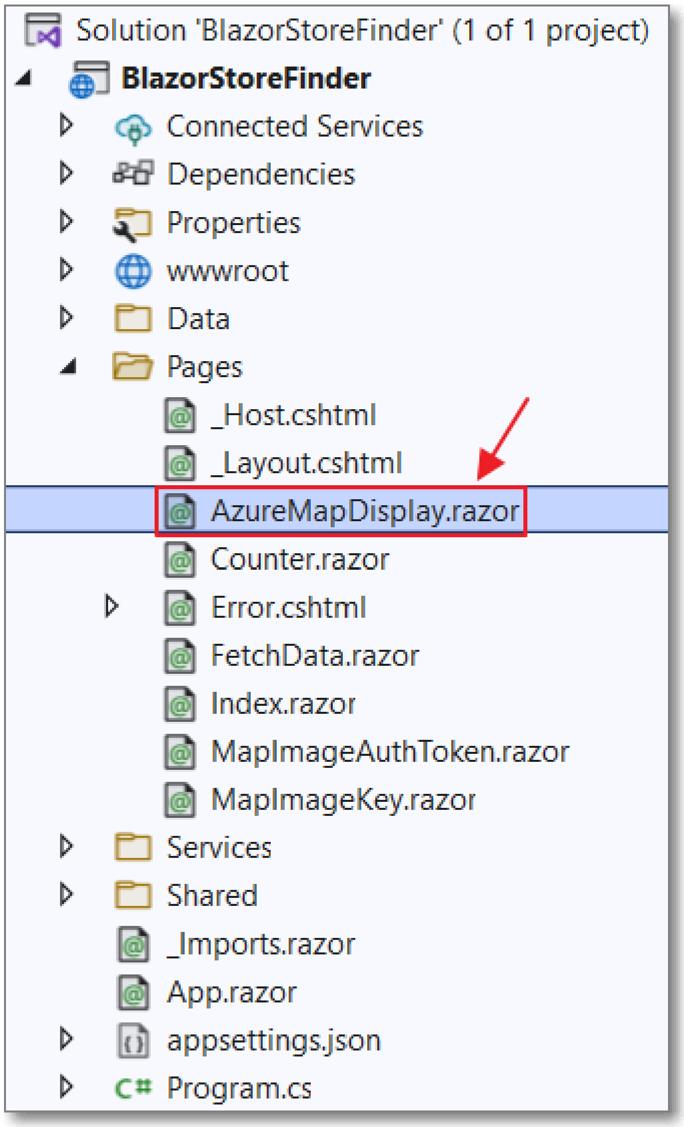
```
<script
src="https://atlas.microsoft.com/sdk/javascript/mapcontrol/2/atlas.min.js">
</script>
<script
src="https://atlas.microsoft.com/sdk/javascript/drawing/0.1/atlas-
drawing.min.js">
</script>
<script
src="https://atlas.microsoft.com/sdk/javascript/indoor/0.1/atlas-
indoor.js">
</script>
<script
src="_content/AzureMapsControl.Components/azure-maps-control.js"></script>
<script type="text/javascript">
    azureMapsControl.Extensions.getTokenCallback = (resolve, reject, map)
=> {
        DotNet.invokeMethodAsync('BlazorStoreFinder', 'GetAccessToken')
        .then(function (response) {
            return response;
        }).then(function (token) {
            resolve(token);
        });
    };
</script>
```

This code adds the required JavaScript libraries, as well a custom JavaScript method, that will retrieve an access token by calling the `GetAccessToken` method in the `AuthService` class created earlier.

This is required to allow the Blazor `AzureMapsControl` to pass the auth token to the Azure Maps JavaScript API.

Note: For more information on *`the DotNet.invokeMethodAsync`* ***method used in the previous code, see: "Call .NET methods from JavaScript functions in ASP.NET Core Blazor" in this article.***

Display an Azure map



```
Solution 'BlazorStoreFinder' (1 of 1 project)
  BlazorStoreFinder
    Connected Services
    Dependencies
    Properties
    wwwroot
    Data
    Pages
      _Host.cshtml
      _Layout.cshtml
      AzureMapDisplay.razor
      Counter.razor
      Error.cshtml
      FetchData.razor
      Index.razor
      MapImageAuthToken.razor
      MapImageKey.razor
    Services
    Shared
    _Imports.razor
    App.razor
    appsettings.json
    Program.cs
```

Figure 40: AzureMapDisplay.razor Control

Create a new Razor control called **`AzureMapDisplay.razor`** using the following code.

Code Listing 20: AzureMapDisplay Using Statements

```
@page "/azuremapdisplay"
@using AzureMapsControl.Components.Map
@using Darnton.Blazor.DeviceInterop.Geolocation;
@inject IGeolocationService GeolocationService
```

Add the following UI markup code to display a Show Traffic button. When clicked, it will call a method (to be created later) that will display the current traffic conditions on the map.

Code Listing 21: Show Traffic Button

```
<button @onclick="ShowTraffic">Show Traffic</button>
```

Next, add the following UI markup code to instantiate the **`AzureMap`** control.

Code Listing 22: AzureMap Control

```
<AzureMap Id="map"
          CameraOptions="new CameraOptions { Zoom = 10 }"
          StyleOptions="new StyleOptions { ShowLogo = false }"
          EventActivationFlags="MapEventActivationFlags.None()
                               .Enable(MapEventType.Ready)"
          TrafficOptions="new
AzureMapsControl.Components.Traffic.TrafficOptions
                           {
                               Incidents = false,
                               Flow =
AzureMapsControl.Components.Traffic
                                      .TrafficFlow.Relative
                           }"
          OnReady="OnMapReadyAsync" />
```

Add the following code to create needed fields.

Code Listing 23: AzureMapDisplay Fields

```
@code {
    MapEventArgs? myMap;
    protected GeolocationResult? CurrentPositionResult { get; set; }


}
```

Add the following method that will run when the **`AzureMap`** control invokes the **`onMapReadyAsync`** event.

This method will determine your current location and set your current latitude and longitude on the map.

Code Listing 24: AzureMapDisplay OnMapReadyAsync Method

```
public async Task OnMapReadyAsync(MapEventArgs eventArgs)
{
    myMap = eventArgs;

    // Get current location
    // will cause a popup to show to ask permission.
    CurrentPositionResult = await
GeolocationService.GetCurrentPosition();

    if (CurrentPositionResult.IsSuccess)
    {
        // Get latitude and longitude.
        string? CurrentLatitude =
        CurrentPositionResult.Position?.Coords?.Latitude.ToString("F2");

        string? CurrentLongitude =
        CurrentPositionResult.Position?.Coords?.Longitude.ToString("F2");

        if (CurrentLatitude != null && CurrentLongitude != null)
        {
            // Set the latitude and longitude as the map camera center.
            await eventArgs.Map.SetCameraOptionsAsync(
                options => options.Center =
                new AzureMapsControl.Components.Atlas.Position
                (
                    Convert.ToDouble(CurrentLongitude),
                    Convert.ToDouble(CurrentLatitude)
                ));
        }
    }
}
```

Lastly, add the following method that will enable the map to display the current traffic conditions when triggered by the **Show Traffic** button.

Code Listing 25: Show Traffic Method

```
public async Task ShowTraffic()
{
    if (myMap != null)
    {
        // Enable traffic and set options.
        await myMap.Map
        .SetTrafficOptionsAsync(options => options.Incidents = true);
    }
}
```

Add the following code to **NavMenu.razor** to add a link to the control.

Code Listing 26: AzureMap Link

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="azuremapdisplay">
```

```
        <span class="oi oi-aperture" aria-hidden="true"></span> Azure Map
    </NavLink>
</div>
```

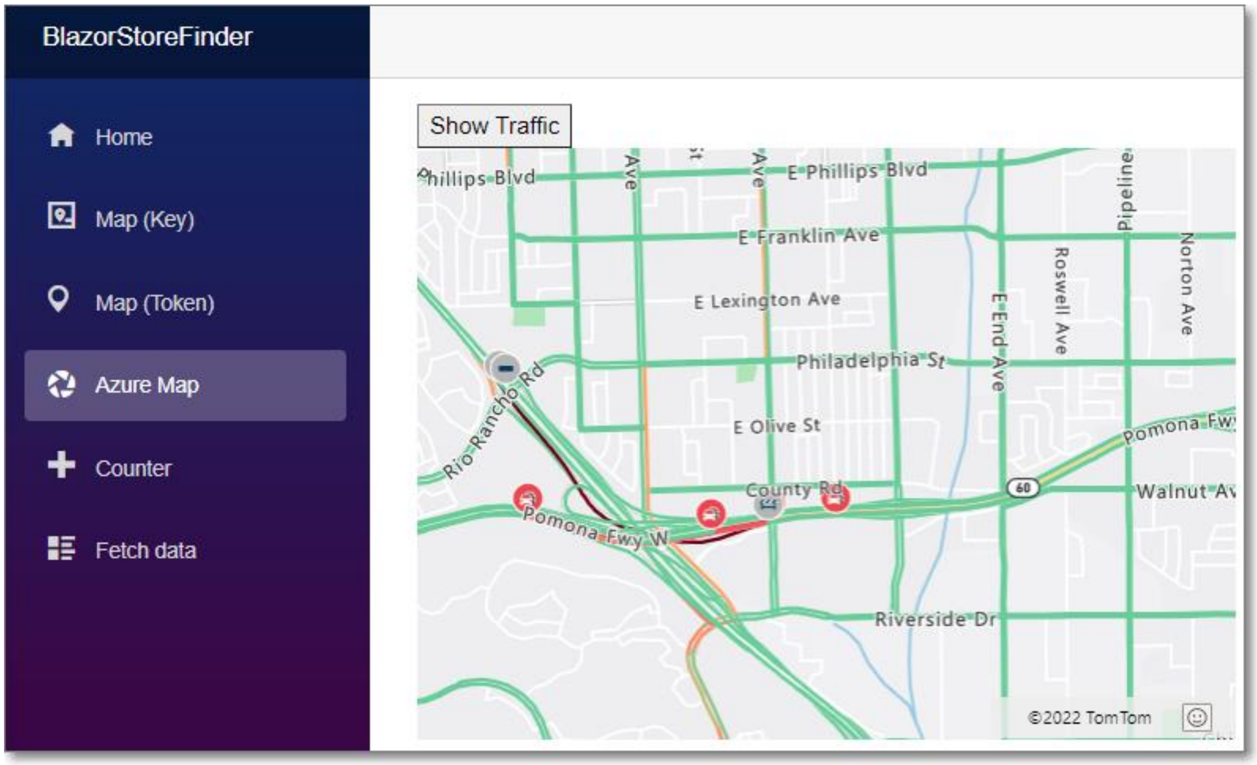
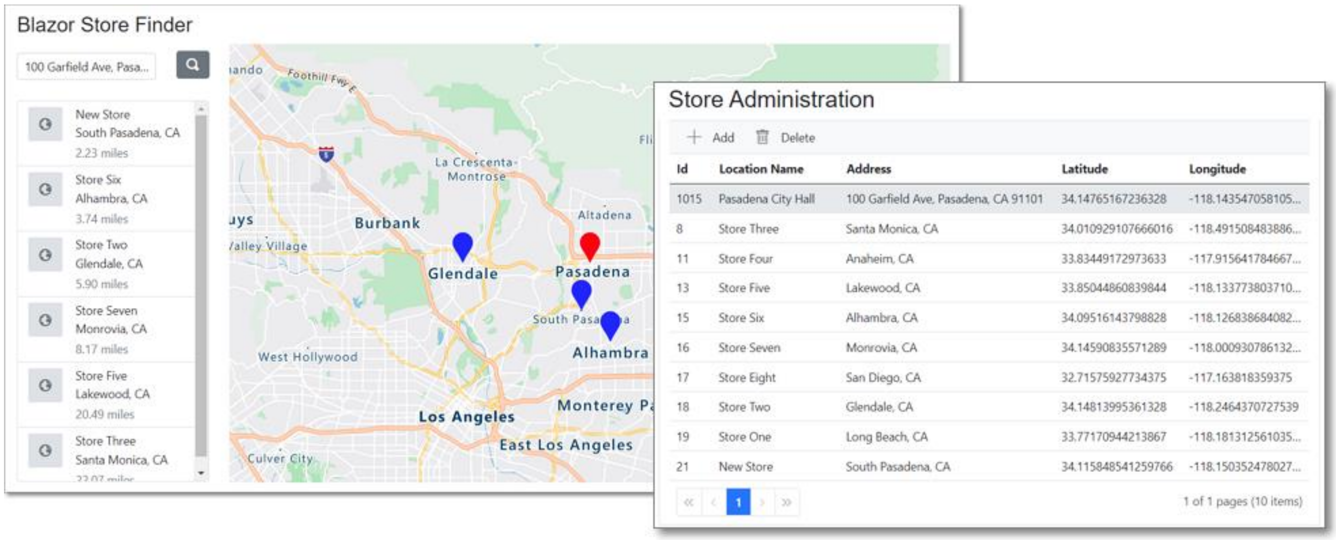


Figure 41: Display Map with Traffic

Run the application and navigate to the **Azure Map** page.

The map will appear displaying your current location. Clicking the **Show Traffic** button will display current traffic conditions.

Chapter 3 The Store Finder Application



Id	Location Name	Address	Latitude	Longitude
1015	Pasadena City Hall	100 Garfield Ave, Pasadena, CA 91101	34.14765167236328	-118.143547058105...
8	Store Three	Santa Monica, CA	34.010929107666016	-118.491508483886...
11	Store Four	Anaheim, CA	33.83449172973633	-117.915641784667...
13	Store Five	Lakewood, CA	33.85044860839844	-118.133773803710...
15	Store Six	Alhambra, CA	34.09516143798828	-118.126838684082...
16	Store Seven	Monrovia, CA	34.14590835571289	-118.000930786132...
17	Store Eight	San Diego, CA	32.71575927734375	-117.163818359375
18	Store Two	Glendale, CA	34.14813995361328	-118.2464370727539
19	Store One	Long Beach, CA	33.77170944213867	-118.181312561035...
21	New Store	South Pasadena, CA	34.115848541259766	-118.150352478027...

1 of 1 pages (10 items)

Figure 42: The Store Finder Application in Action

In the remaining chapters of this book, we will construct **Blazor Store Finder**, a real-world application using the Azure Maps API.

This will resemble the traditional store locator application that allows an administrator to create and geocode store locations, and then allow an end user to see a map of stores near their location.

The structure of the application is as follows:

- Store administration
  - Add and edit store locations.
    - Geocode store locations.
- Store locator
  - Determine your current location and set as the default location.
    - Search for stores.
    - Display stores near your selected location.

Chapter 4 Create the Application

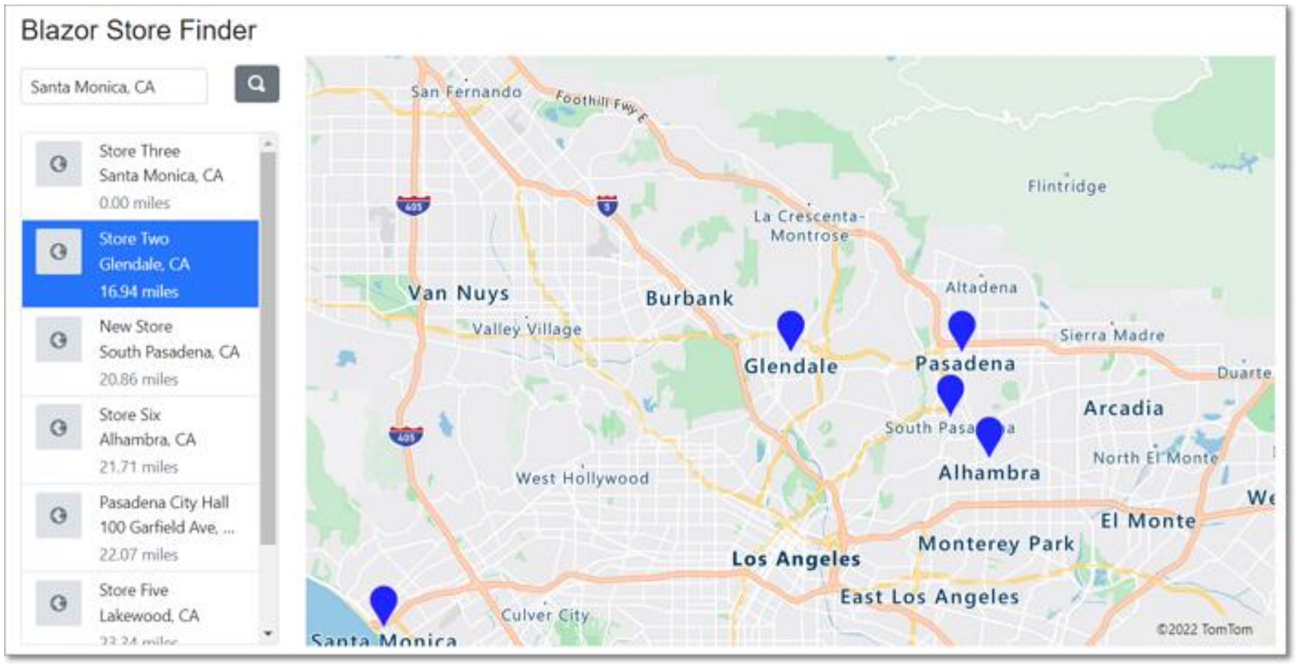
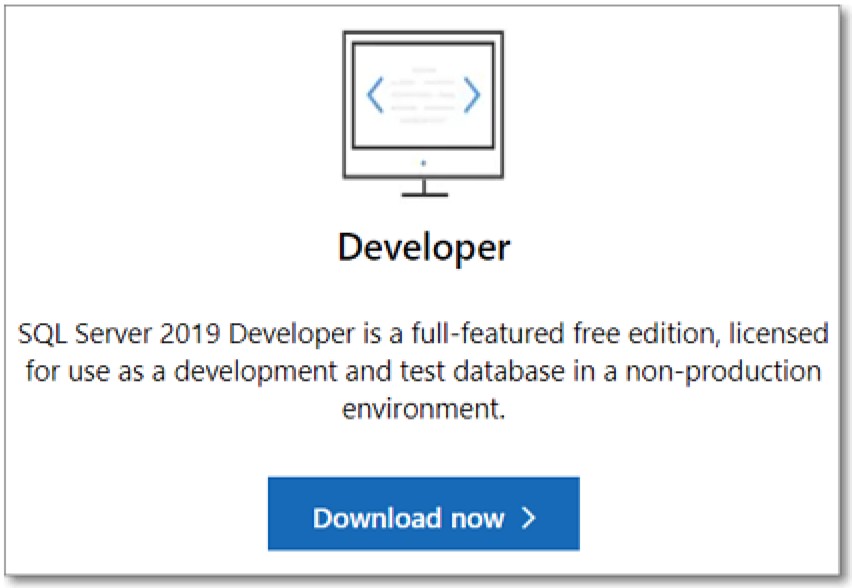


Figure 43: Store Finder Application

In this chapter, we will cover the steps to set up the development of the Blazor Store Finder application.

Note: The source code for the completed application is available on [GitHub](#).

Install SQL Server


Developer

SQL Server 2019 Developer is a full-featured free edition, licensed for use as a development and test database in a non-production environment.

Figure 44: SQL Server

The application requires a database to store the data. Download and install the free SQL Server 2019 Developer Edition here. Or, if you have access to it, use the full SQL Server.

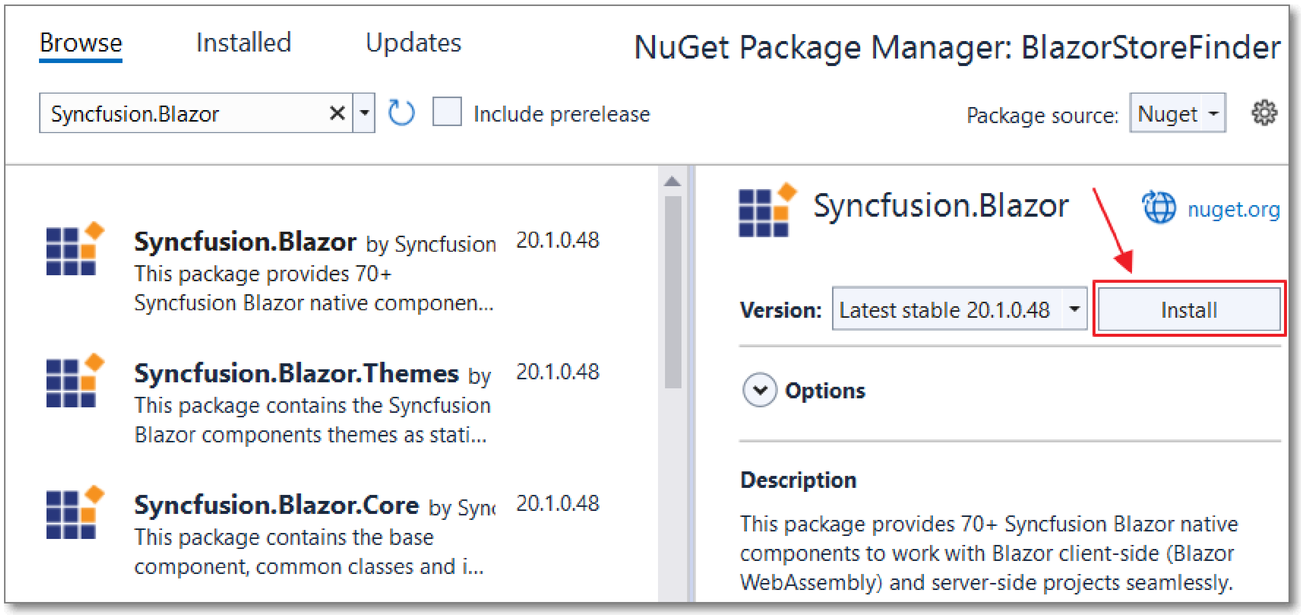
Add Syncfusion

The Syncfusion controls allow us to implement advanced functionality easily, with a minimum amount of code. The Syncfusion controls are contained in a NuGet package.

In this section, we will cover the steps to obtain that package and configure it for our application.

Note: You can find the latest requirements to use Syncfusion here.

Install and configure Syncfusion



Syncfusion.Blazor by Syncfusion 20.1.0.48
This package provides 70+ Syncfusion Blazor native componen...

Syncfusion.Blazor.Themes by 20.1.0.48
This package contains the Syncfusion Blazor components themes as stati...

Syncfusion.Blazor.Core by Sync 20.1.0.48
This package contains the base component, common classes and i...

Version: Latest stable 20.1.0.48

Description

This package provides 70+ Syncfusion Blazor native components to work with Blazor client-side (Blazor WebAssembly) and server-side projects seamlessly.

Figure 45: Install Syncfusion NuGet Package

Install the following NuGet packages:

- Syncfusion.Blazor
- Syncfusion.Blazor.Themes

Open the **_Imports.razor** file and add the following.

Code Listing 27: Syncfusion in _Imports.razor

```
@using Syncfusion.Blazor
```

Open the **Program.cs** file and add the following **`using`** statement.

Code Listing 28: Syncfusion Program.cs Using Statement

```
using Syncfusion.Blazor;
```

Next, add the following code (before the **`var app = builder.Build()`** line).

Code Listing 29: AddSyncfusionBlazor

```
// Add Syncfusion
builder.Services.AddSyncfusionBlazor();
```

Add the following to the **`head`** element of the **_Layout.cshtml** page.

Code Listing 30: Syncfusion in _Layout.cshtml

```
<link href="_content/Syncfusion.Blazor.Themes/bootstrap5.css" rel="stylesheet" />

<script src="_content/Syncfusion.Blazor/scripts/syncfusion-blazor.min.js"
    type="text/javascript"></script>
```

Note: When you run the application, you will see a message:

This application was built using a trial version of Syncfusion Essential Studio. Please include a valid license to permanently remove this license validation message. You can also obtain a free 30-day evaluation license to temporarily remove this message during the evaluation period. Please refer to this help topic for more information.

Click the link in the message for instructions on obtaining a key to make the message go away.

Create the database

From the toolbar, select **View**, and then choose SQL Server Object Explorer.

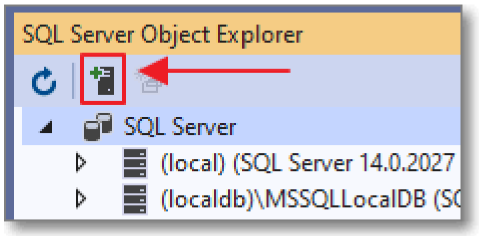

SQL Server Object Explorer
SQL Server
(local) (SQL Server 14.0.2027
(localdb)\MSSQLLocalDB (SC

Figure 46: Add SQL Server

Click the **Add SQL Server** button to add a connection to your database server, if you don't already have it in the **SQL Server** list.

Note: For this example, we do not want to use the (localdb) connection (for SQL Express) that you may see in the list.

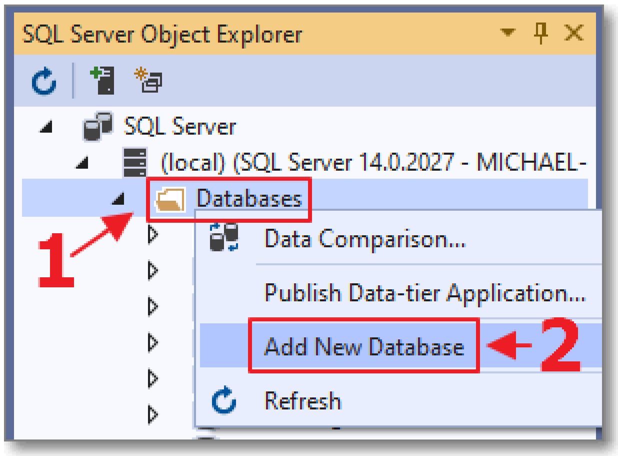

SQL Server Object Explorer
SQL Server
(local) (SQL Server 14.0.2027 - MICHAEL-
Databases
1
Data Comparison...
Publish Data-tier Application...
Add New Database
2
Refresh

Figure 47: Add New Database

Expand the tree node for your SQL server, then right-click **Databases** and select **Add New Database**.

Name the database **BlazorStoreFinder**.

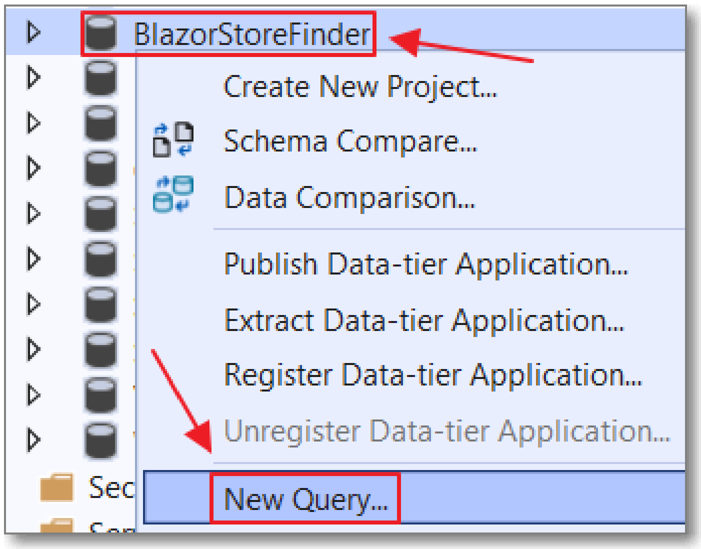

BlazorStoreFinder
Create New Project...
Schema Compare...
Data Comparison...
Publish Data-tier Application...
Extract Data-tier Application...
Register Data-tier Application...
Unregister Data-tier Application...
New Query...
Sec

Figure 48: Create New Query

We will now create the **StoreLocations** table that will contain the stores. After the database has been created, right-click it and select **New Query**.

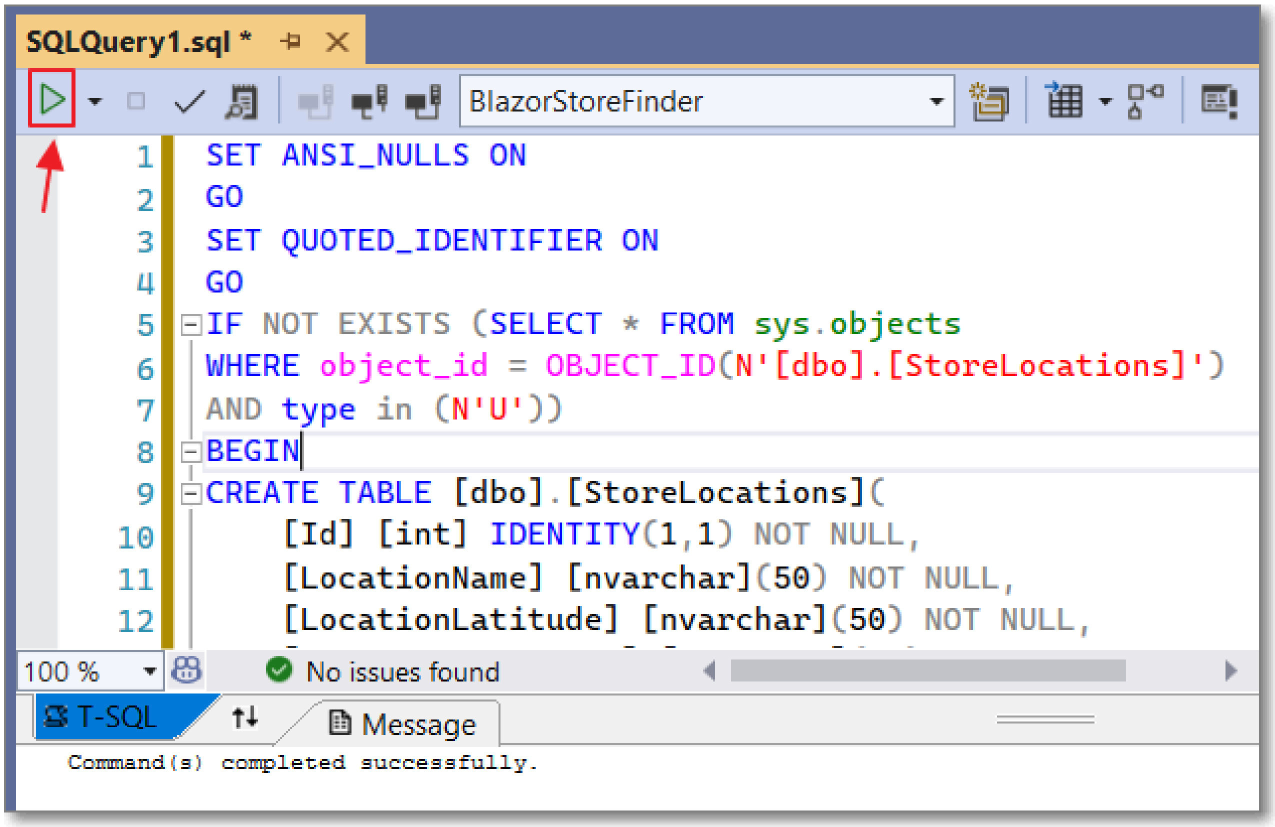
Paste in the following script.

Code Listing 31: StoreLocations Table SQL Script

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
```

```
GO
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[StoreLocations]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[StoreLocations](
      [Id] [int] IDENTITY(1,1) NOT NULL,
      [LocationName] [nvarchar](50) NOT NULL,
      [LocationLatitude] [nvarchar](50) NOT NULL,
      [LocationLongitude] [nvarchar](50) NOT NULL,
      [LocationAddress] [nvarchar](250) NOT NULL,
      [LocationData] [geography] NOT NULL,
PRIMARY KEY CLUSTERED
(
      [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
END
GO
```

Click the **Execute** icon.

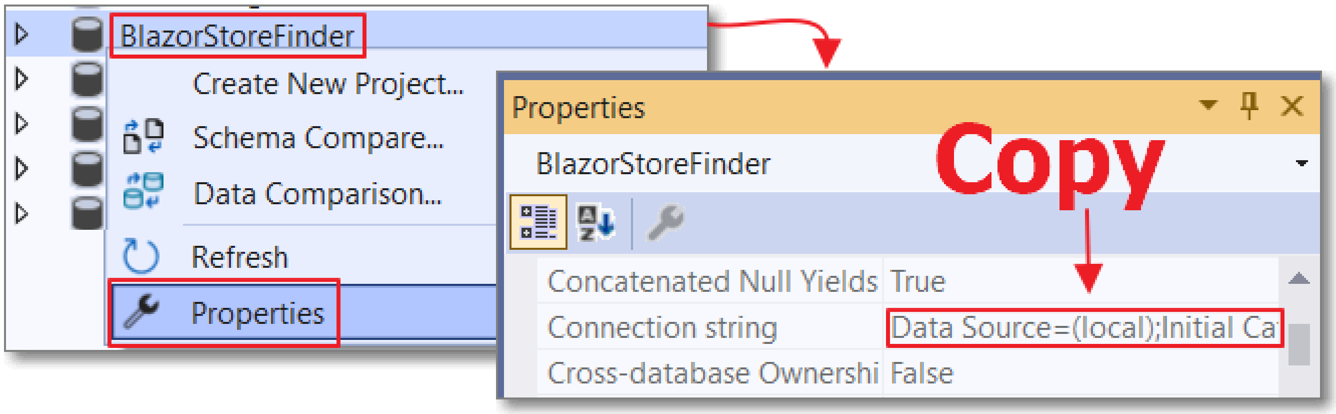


```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[StoreLocations]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[StoreLocations](
     [Id] [int] IDENTITY(1,1) NOT NULL,
     [LocationName] [nvarchar](50) NOT NULL,
     [LocationLatitude] [nvarchar](50) NOT NULL,
```

```
Command(s) completed successfully.
```

Figure 49: Execute SQL Script to Create StoreLocations Table

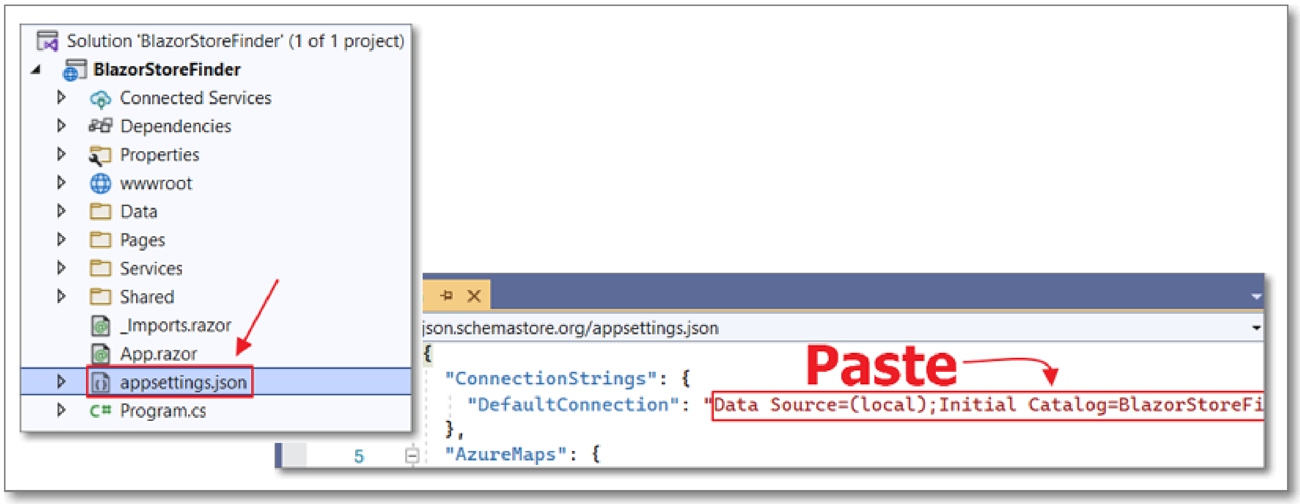
The **StoreLocations** table will be created.



Concatenated Null Yields	True
Connection string	Data Source=(local);Initial Ca
Cross-database Ownershi	False

Figure 50: Copy Connection String

Right-click the database and select **Properties**. In the **Properties** window, copy the **Connection string**.



```
{
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=(local);Initial Catalog=BlazorStoreFi
  },
  "AzureMaps": {
```

Figure 51:Paste Connection String

Open the **appsettings.json** file and paste the connection string in the **DefaultConnection** property.

Save and close the file.

Chapter 5 Creating Store Administration

In this chapter, we will construct an administration page that will allow store locations to be geocoded and added to the database.

Create the data layer

We will now create the data context code that will allow the data service, created in the following steps, to communicate with the database tables we just added.

A data context is a layer of code that sits between the database and the C# code that communicates with the database.

Install the following NuGet packages:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.SqlServer.NetTopologySuite

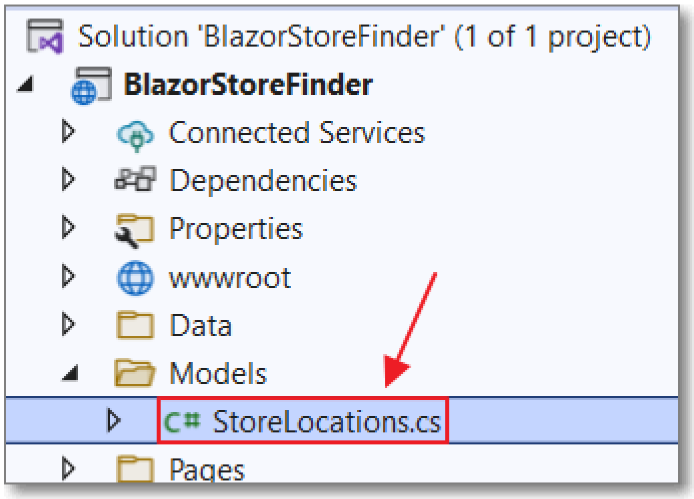


Figure 52: Add StoreLocations.cs

Create a **Models** folder and add a new class named **`StoreLocations.cs`** using the following code.

Code Listing 32

```
#nullable disable
namespace BlazorStoreFinder
{
    public partial class StoreLocations
    {
        public int Id { get; set; }
        public string LocationName { get; set; }
        public string LocationLatitude { get; set; }
        public string LocationLongitude { get; set; }
        public string LocationAddress { get; set; }
        public NetTopologySuite.Geometries.Point LocationData { get; set;
}
    }
}
```

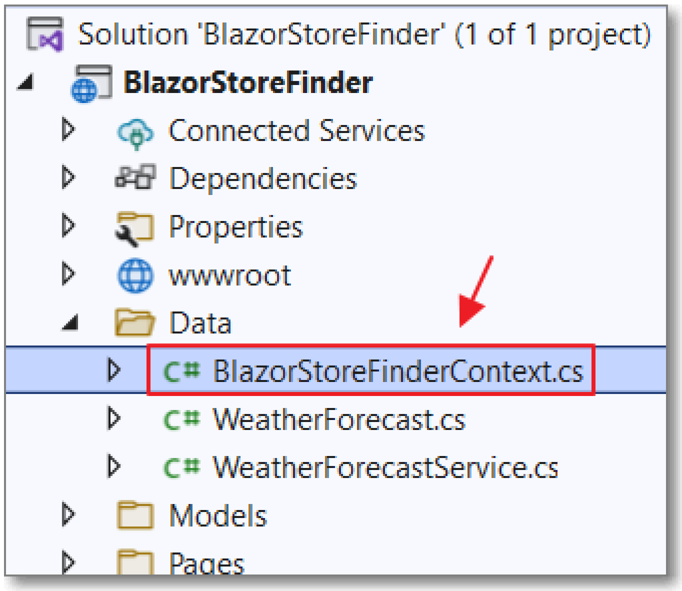

Solution 'BlazorStoreFinder' (1 of 1 project)
BlazorStoreFinder
Connected Services
Dependencies
Properties
wwwroot
Data
BlazorStoreFinderContext.cs
WeatherForecast.cs
WeatherForecastService.cs
Models
Pages

Figure 53: Create BlazorStoreFinderContext.cs

Add a **`BlazorStoreFinderContext.cs`** class in the **Data** folder using the following code.

Code Listing 32: BlazorStoreFinderContext.cs File

```
#nullable disable
using System;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata;

namespace BlazorStoreFinder
{
    public partial class BlazorStoreFinderContext : DbContext
    {
        public BlazorStoreFinderContext()
        {
        }
        public BlazorStoreFinderContext(
            DbContextOptions<BlazorStoreFinderContext> options)
            : base(options)
        {
        }

        public virtual DbSet<StoreLocations> StoreLocations { get; set; }

        protected override void OnModelCreating(ModelBuilder 
modelBuilder)
        {
            modelBuilder.Entity<StoreLocations>(entity =>
            {
                entity.Property(e => e.LocationAddress)
                .IsRequired()
                .HasMaxLength(250);

                entity.Property(e => e.LocationLatitude)
                .IsRequired()
                .HasMaxLength(50);

                entity.Property(e => e.LocationLongitude)
                .IsRequired()
                .HasMaxLength(50);

                entity.Property(e => e.LocationName)
                .IsRequired()
                .HasMaxLength(50);

                entity.Property(e => e.LocationData)
                .IsRequired();
            });

            OnModelCreatingPartial(modelBuilder);
        }

        partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
    }
}
```

The preceding data context class leverages Entity Framework Core to communicate with the database. The class contains two constructors. The second constructor allows a database connection string to be passed. We will do that in the **Program.cs** file.

Open the **Program.cs** file and add the following **`using`** statements.

Code Listing 33: Data Layer Using Statements

```
using BlazorStoreFinder;
using Microsoft.EntityFrameworkCore;
```

Next, add the following code (after the **`var builder = WebApplication.CreateBuilder(args)`** line).

Code Listing 34: Data Connection Configuration

```
builder.Services.AddDbContext<BlazorStoreFinderContext>(options =>
options.UseSqlServer(
    builder.Configuration.GetConnectionString("DefaultConnection"),
x => x.UseNetTopologySuite()));
```

This code retrieves the database connection string from the appsettings.json file and passes it to the **`BlazorStoreFinderContext`** data context class created earlier. It also configures the **`NetTopologySuite`** component that will allow access to spatial data types in the database.

Store location service

We will now create a **`StoreLocationService`** class that will use the data context to read and write to the database. We will create data access methods that will be called by Razor controls to be created later.

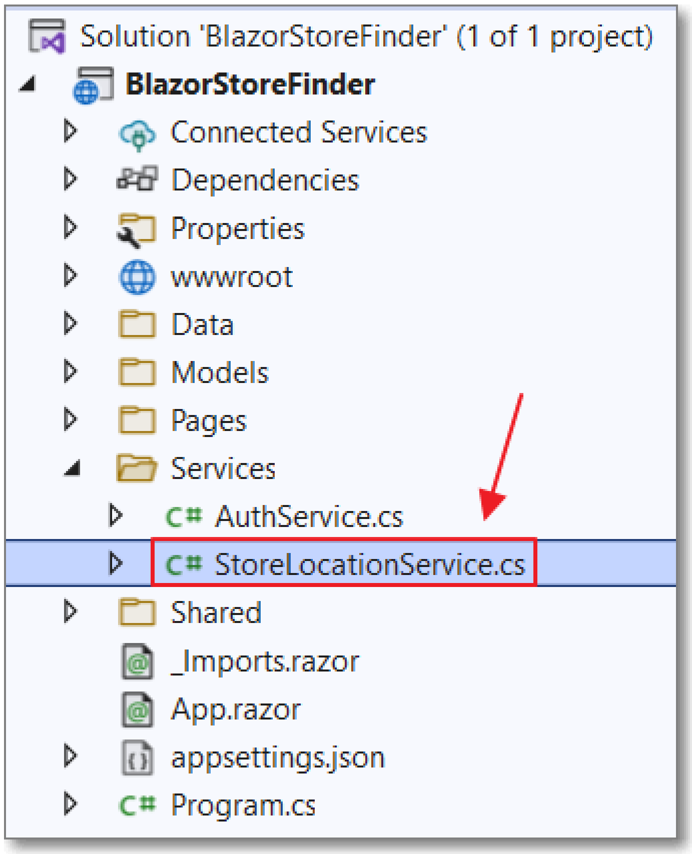


Figure 54: Create StoreLocationService.cs

Add a **`StoreLocationService.cs`** class in the **Services** folder using the following code.

Code Listing 35: StoreLocationService Class

```
#nullable disable
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;
using NetTopologySuite.Geometries;
using Newtonsoft.Json;
using System.Data;
using System.Net.Http.Headers;
using System.Text;

namespace BlazorStoreFinder
{
    public class StoreLocationService
    {
        private readonly BlazorStoreFinderContext _context;
        public StoreLocationService(BlazorStoreFinderContext context)
        {
            _context = context;
        }
    }
}
```

Next, add the following method to the class that will return a collection of all store locations in the database.

Code Listing 36: GetStoreLocations

```
    public async Task<List<StoreLocations>> GetStoreLocations()
    {
        return await
            _context.StoreLocations.OrderBy(x => x.Id).ToListAsync();
    }
```

Add the following code that will return a single store location when passed the record ID.

Code Listing 37: GetStoreLocation

```
    public async Task<StoreLocations> GetStoreLocation(int id)
    {
        return await _context.StoreLocations.FindAsync(id);
    }
```

Add the following code to allow a store location to be inserted into the database.

Code Listing 38: AddStoreLocation

```
    public async Task<StoreLocations> AddStoreLocation
        (StoreLocations storeLocation)
    {
        _context.StoreLocations.Add(storeLocation);
        await _context.SaveChangesAsync();
        return storeLocation;
    }
```

Also, add the following method that will delete a store location when passed the record ID.

Code Listing 40: DeleteStoreLocation

```
    public async Task DeleteStoreLocation(int id)
    {
        var storeLocation =
            await _context.StoreLocations.FindAsync(id);

        _context.StoreLocations.Remove(storeLocation);
        await _context.SaveChangesAsync();
    }
```

Finally, to register the store location service, add the following code to the **Program.cs** file before the **`var app = builder.Build()`** line.

Code Listing 39: Register StoreLocationService

```
// Register StoreLocationService
builder.Services.AddScoped<StoreLocationService>();
```

Geocode store locations

We will now add a method to the store location service that will accept an address, geocode it, and return a coordinate containing the latitude and longitude. This will then be stored in the database using the **`AddStoreLocation`** method created earlier.

To geocode the address, we will call the Azure Maps API. This API method will return a **`SearchAddressResult`** object containing more than just the coordinates.

The first step is to create a class, matching the properties of the **`SearchAddressResult`** object, that will be used to deserialize the response from the Azure Maps API.

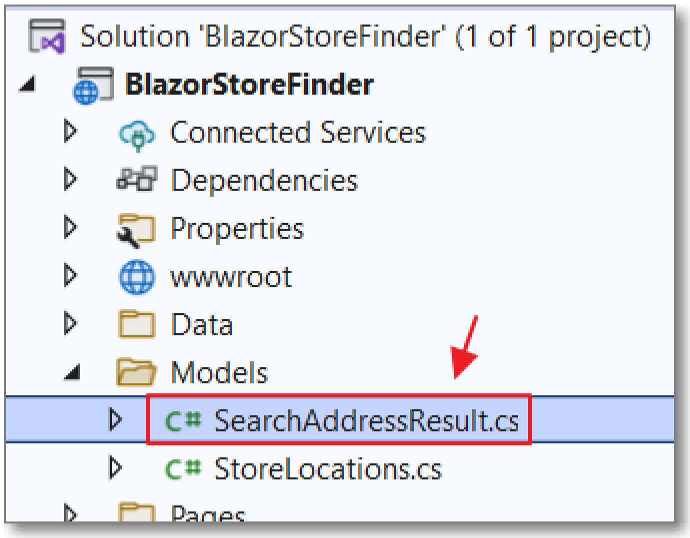


Figure 55: Create SearchAddressResult.cs

In the **Models** folder, add a **`SearchAddressResult.cs`** class using the following code.

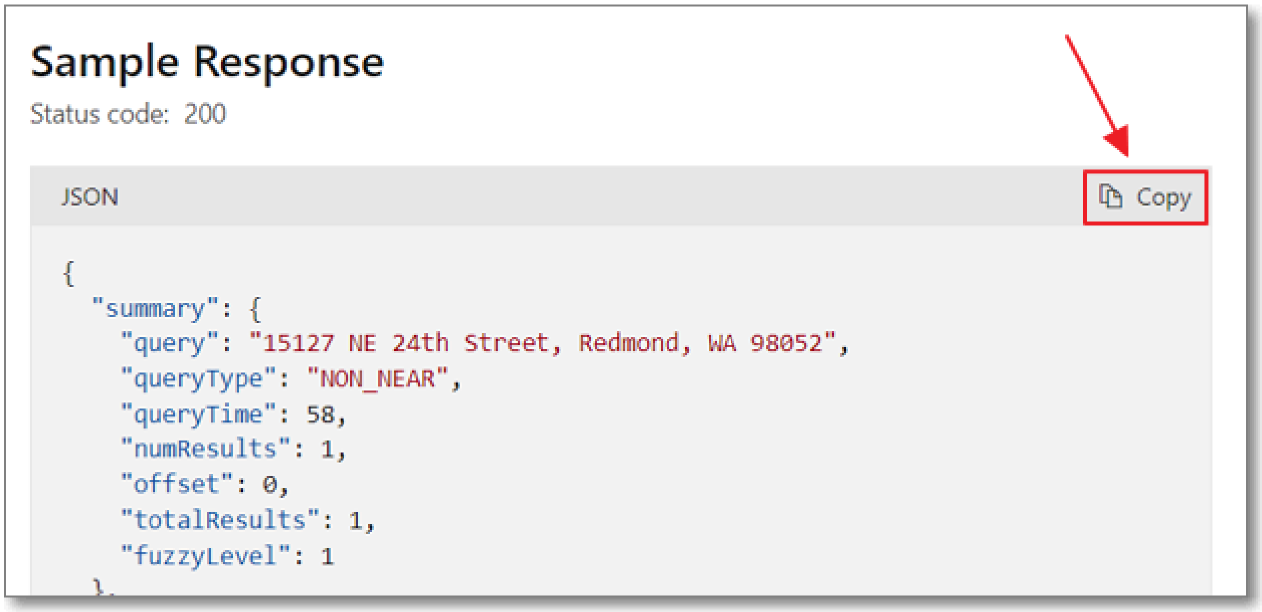
Code Listing 40: SearchAddressResult.cs

```
#nullable disable
```

```
namespace BlazorStoreFinder.Result
{

}
```

In your web browser, navigate to the following URL:

https://docs.microsoft.com/en-us/rest/api/maps/search/get-search-address



```
{
  "summary": {
    "query": "15127 NE 24th Street, Redmond, WA 98052",
    "queryType": "NON_NEAR",
    "queryTime": 58,
    "numResults": 1,
    "offset": 0,
    "totalResults": 1,
    "fuzzyLevel": 1
  },
```

Figure 56: Copy SearchAddressResult

Scroll down to the **Sample Response** section and use the **Copy** button to copy it.

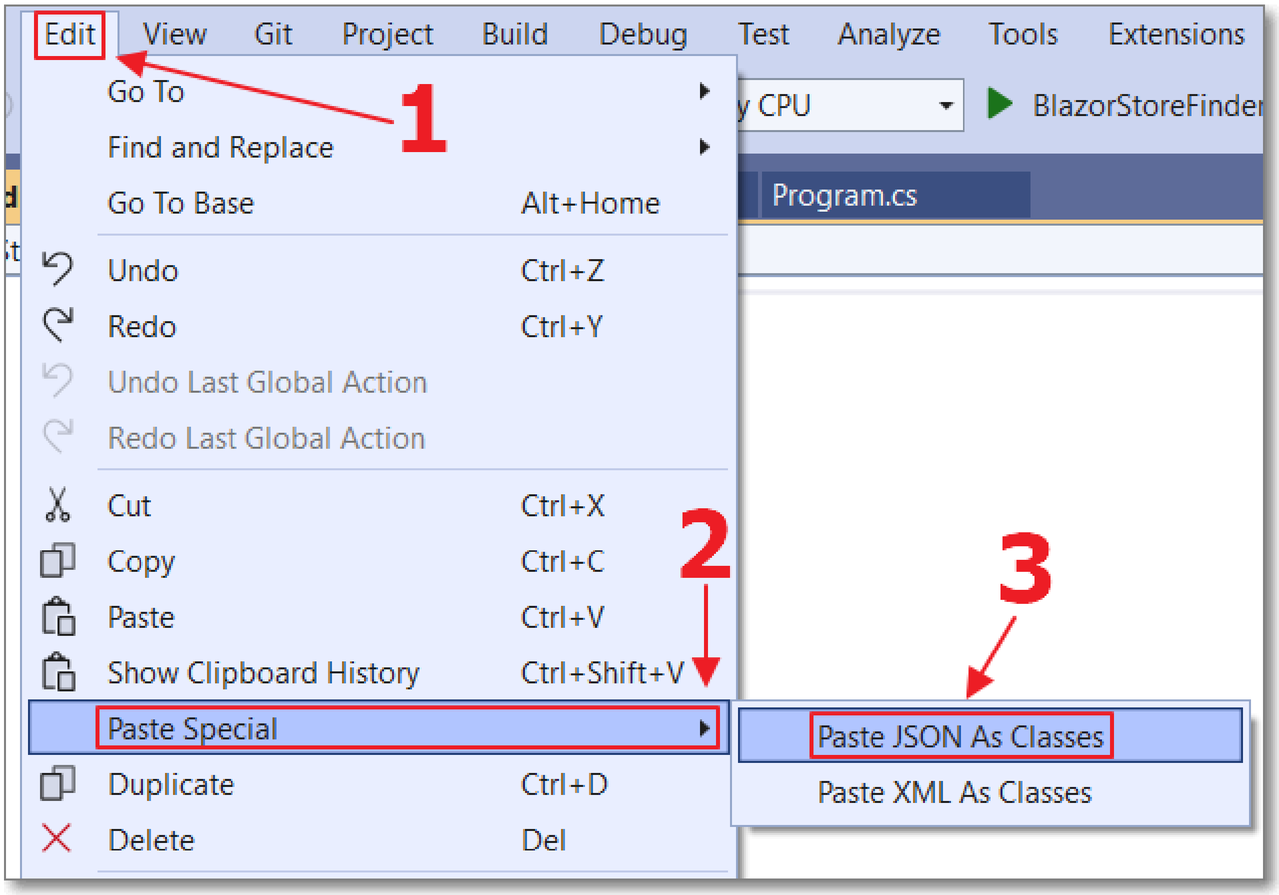
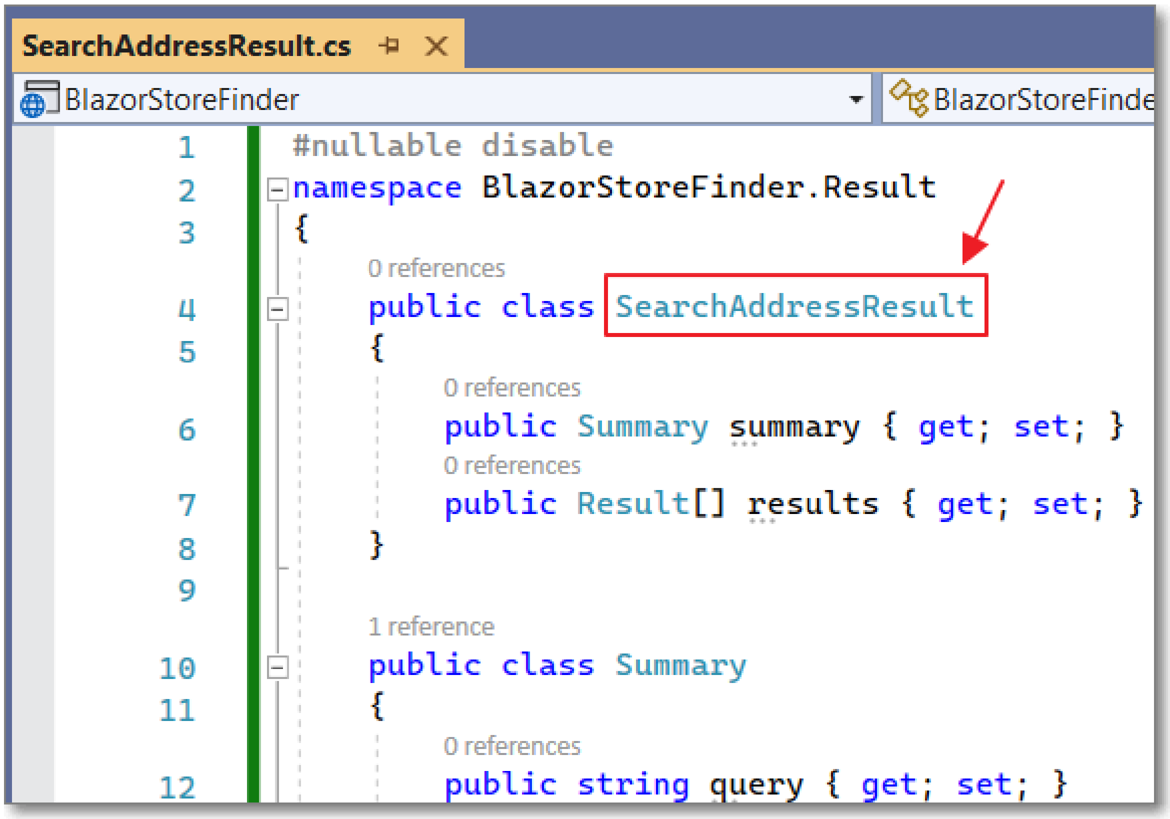


Figure 57: Paste JSON As Classes

In **Visual Studio**, select **Edit** from the toolbar, then **Paste Special** > **Paste JSON As Classes** to paste the contents inside the namespace declaration.



```
#nullable disable
namespace BlazorStoreFinder.Result
{
    0 references
    public class SearchAddressResult
    {
        0 references
        public Summary summary { get; set; }
        0 references
        public Result[] results { get; set; }
    }

    1 reference
    public class Summary
    {
        0 references
        public string query { get; set; }
```

Figure 58: Change Rootobject to SearchAddressResult

The pasted code will set that root class name to **`Rootobject`**. Change **`Rootobject`** to **`SearchAddressResult`**.

Next, add the following **`using`** statement to the store location service class (**`StoreLocationService.cs`**).

Code Listing 413: BlazorStoreFinder.Result Using Statement

```
using BlazorStoreFinder.Result;
```

Finally, add the following method to the class that will call the Azure Maps API and return a **`Coordinate`** object containing latitude and longitude when passed an address.

Code Listing 424: GeocodeAddress

```
public async Task<Coordinate> GeocodeAddress(string address)
{
    Coordinate = new Coordinate();

    // Create an HTTP client to make the REST call.
    // Search - Get search address
    // https://bit.ly/3JER1ii

    // Best practices for Azure Maps Search Service
    // https://bit.ly/3JFQkFt

    using (var client = new System.Net.Http.HttpClient())
    {
        // Get an access token from AuthService.
        var AccessToken = await AuthService.GetAccessToken();

        client.DefaultRequestHeaders.Accept.Clear();
        client.DefaultRequestHeaders.Accept.Add(
            new MediaTypeWithQualityHeaderValue("application/json"));

        // Pass the Azure Maps client Id.
        client.DefaultRequestHeaders.Add("x-ms-client-id",
            AuthService.ClientId);
        // Pass the access token in the auth header.
        client.DefaultRequestHeaders.Authorization =
        new System.Net.Http.Headers.AuthenticationHeaderValue(
            "Bearer", AccessToken);

        // Build the URL.
        StringBuilder sb = new StringBuilder();

        // Request an address search.
        sb.Append("https://atlas.microsoft.com/search/address/json?");
        // Specify the API version and language.
        sb.Append("&api-version=1.0&language=en-US");
        // Pass the address.
        sb.Append($"&query={address}");

        // Set the URL.
        var url = new Uri(sb.ToString());

        // Call Azure Maps and get the response.
        var Response = await client.GetAsync(url);

        // Read the response.
        var responseContent = await Response.Content.ReadAsStringAsync();
        var AddressResult =
            JsonConvert.DeserializeObject<SearchAddressResult>(
                responseContent);

        // Create coordinate.
        coordinate = new Coordinate(
            Convert.ToDouble(
                AddressResult.results.FirstOrDefault()?.position.lon),
            Convert.ToDouble(
                AddressResult.results.FirstOrDefault()?.position.lat));
    }

    return coordinate;
}
```

Add and edit store locations

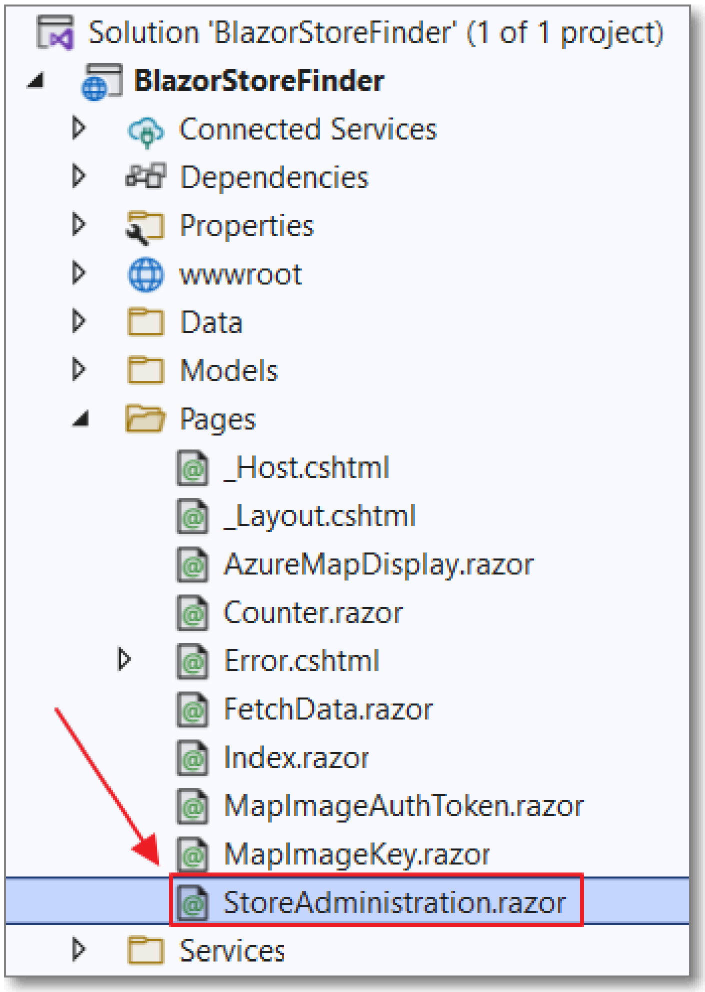


Figure 59: StoreAdministration.razor

In the **Pages** folder, create a new control named **`StoreAdministration.razor`** using the following code.

Code Listing 435: Administration Control

```
@page "/storeadmin"
@using Syncfusion.Blazor.Grids
@using Syncfusion.Blazor.Calendars
@using Syncfusion.Blazor.DropDowns
@using Syncfusion.Blazor.Inputs
@using Syncfusion.Blazor.Popups
@using System.Text;
@using NetTopologySuite.Geometries
@using Newtonsoft.Json
@using System.Net.Http.Headers
@inject StoreLocationService _StoreLocationService
@inherits OwningComponentBase<StoreLocationService>
<h3>Store Administration</h3>



@code {
    List<StoreLocations> storelocations =
    new List<StoreLocations>();

    private DialogSettings DialogParams =
    new DialogSettings { MinHeight = "250px", Width = "450px" };

}
```

Add the following code to the UI section. This will create a Syncfusion **`DataGrid`** component that contains the store locations and allows you to add and delete them.

The data grid also allows editing entries using a pop-up.

Code Listing 446: Syncfusion DataGrid Component

```
<div class="col-lg-12 control-section">
    <div class="content-wrapper">
    <div class="row">
    <SfGrid DataSource="@storelocations"
            Toolbar="@(new List<string>() { "Add", "Delete" })"
AllowPaging="true">
        <GridEvents OnActionBegin="ActionBeginHandler"
TValue="StoreLocations"></GridEvents>
        <GridEditSettings AllowAdding="true" AllowDeleting="true"
                                Mode="@EditMode.Dialog"
Dialog="DialogParams">
            <Template>
                @{
                    var Store = (context as StoreLocations) ?? new
StoreLocations();
```

```
                }
                <div>
                    <div class="form-row">
                        <div class="form-group col-md-12">
                            <SfTextBox ID="Location Name" @bind-
Value="@(Store.LocationName)"
                                       TValue="string"
FloatLabelType="FloatLabelType.Always"
                                       Placeholder="Location
Name"></SfTextBox>
                        </div>
                    </div>
                    <div class="form-row">
                        <div class="form-group col-md-12">
                            <SfTextBox ID="Address" @bind-
Value="@(Store.LocationAddress)"
                                       TValue="string"
FloatLabelType="FloatLabelType.Always"

Placeholder="Address"></SfTextBox>
                        </div>
                    </div>
                </div>
            </Template>
        </GridEditSettings>
        <GridColumns>
            <GridColumn Field=@nameof(StoreLocations.Id)
                        HeaderText="Id" IsPrimaryKey="true"
                        ValidationRules="@(new ValidationRules{
Number=true})" Width="50">
            </GridColumn>
            <GridColumn Field=@nameof(StoreLocations.LocationName)
                        HeaderText="Location Name"
                        ValidationRules="@(new ValidationRules{
Required=true})" Width="150">
            </GridColumn>
            <GridColumn Field=@nameof(StoreLocations.LocationAddress)
                        HeaderText="Address"
                        ValidationRules="@(new ValidationRules{
Required=true})">
            </GridColumn>
            <GridColumn Field=@nameof(StoreLocations.LocationLatitude)
                        HeaderText="Latitude" Width="150">
            </GridColumn>
            <GridColumn Field=@nameof(StoreLocations.LocationLongitude)
                        HeaderText="Longitude" Width="150">
            </GridColumn>
        </GridColumns>
```

```
    </SfGrid>
    </div>
    </div>
</div>
```

Note: For more information on the Syncfusion DataGrid component, see this article.

Add the following method to the code section. This will call the **`GetStoreLocations()`** method, created earlier, to retrieve the store locations from the database and populate the data grid.

Code Listing 457: Calling Service.GetStoreLocations

```
    protected override async Task OnInitializedAsync()
    {
        // We access StoreLocationService using @Service
        storelocations = await Service.GetStoreLocations();
    }
```

Add the **`ActionBeginHandler`** method that will respond to the Syncfusion DataGrid toolbar action event. Based on the option selected, it will call the **`Save`** or **`Delete`** method (to be created later).

Code Listing 468: Save and Delete Handler

```
    public async Task ActionBeginHandler(
        ActionEventArgs<StoreLocations> args)
    {
        if (args.RequestType.ToString() == "Save")
        {
            await Save(args.Data);
        }

        if (args.RequestType.ToString() == "Delete")
        {
            await Delete(args.Data);
        }
    }
```

The following method will delete a store location from the database.

Code Listing 479: Administration Delete Method

```
    public async Task Delete(StoreLocations store)
    {
        await Service.DeleteStoreLocation(store.Id);
        storelocations = await Service.GetStoreLocations();
    }
```

The following **Save** method calls the **GeocodeAddress** method, created earlier, to geocode the address and save the result to the database.

Code Listing 50: Geocode and Save Location

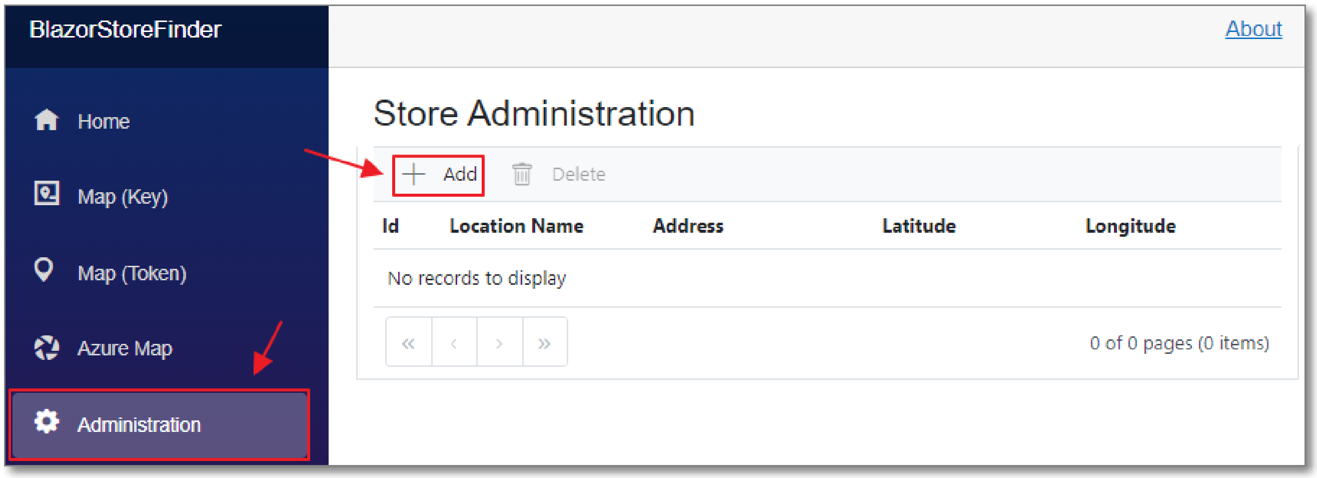
```
    public async Task Save(StoreLocations store)
    {
        // Geocode address.
        Coordinate =
        await Service.GeocodeAddress(store.LocationAddress);

        // Create a new store location.
        store.LocationLatitude = coordinate.Y.ToString();
        store.LocationLongitude = coordinate.X.ToString();

        // SRID 4326 (WGS 84) is the most standard
        // in cartography and GPS systems.
        store.LocationData = new Point(coordinate) { SRID = 4326 };

        // Save a new store location.
        await Service.AddStoreLocation(store);
    }
```

Finally, add the following code to **NavMenu.razor** to add a link to the control.

Code Listing 48: Administration Link

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="storeadmin">
        <span class="oi oi-cog" aria-hidden="true"></span> Administration
    </NavLink>
</div>
```

Run the application and navigate to the **Administration** page.



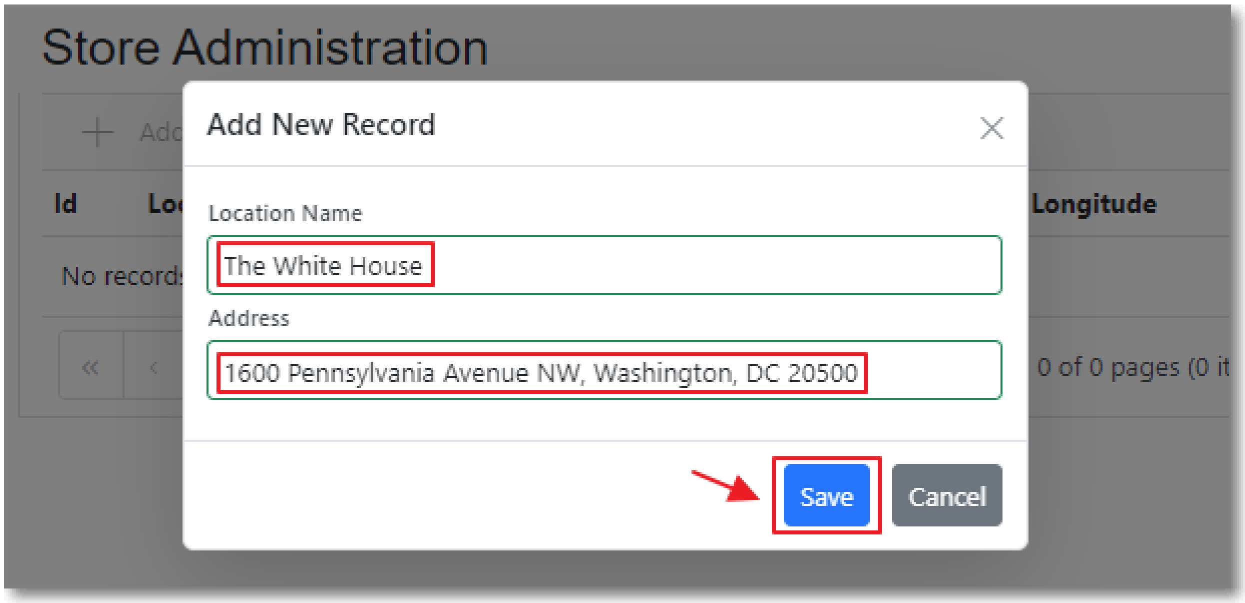
Store Administration

Id	Location Name	Address	Latitude	Longitude
No records to display				

0 of 0 pages (0 items)

Figure 60: Administration Page—Add New Location

Click **Add** to add a new store location.

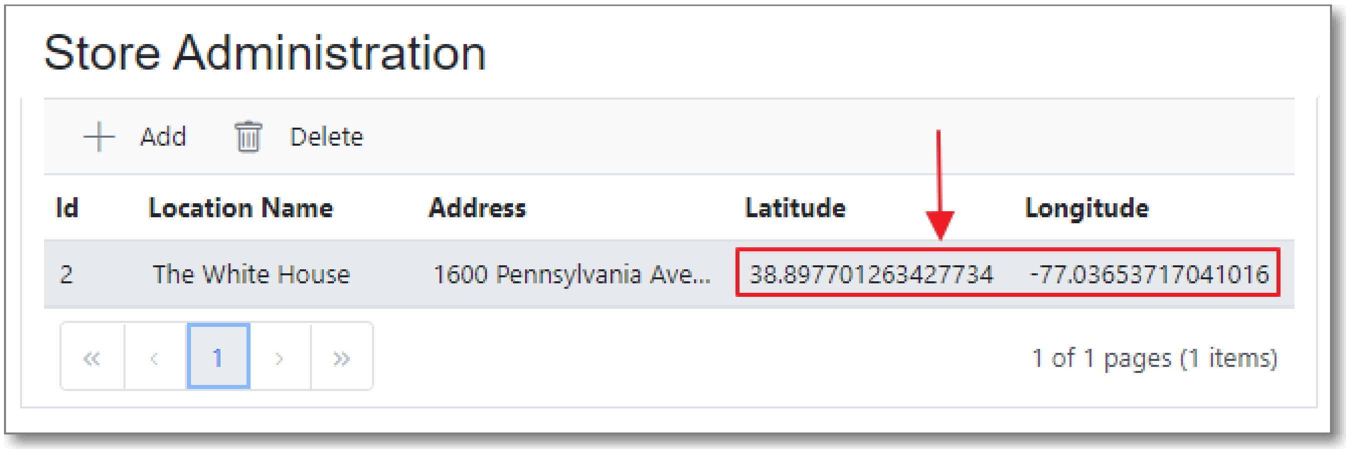


Location Name: The White House

Address: 1600 Pennsylvania Avenue NW, Washington, DC 20500

Figure 61: Add a Store Location

Enter a name and an address for the location and click **Save**.



Store Administration

Id	Location Name	Address	Latitude	Longitude
2	The White House	1600 Pennsylvania Ave...	38.897701263427734	-77.03653717041016

Figure 62: View Latitude and Longitude

The latitude and longitude for the address will appear in the DataGrid (and be saved to the database).

Chapter 6 Create the Store Search Page

In this final chapter, we will construct a search page that will allow the user to search for store locations and display those store locations on an interactive map.

Reverse-geocode a location

As the first step, we will add a method to the store location service that will accept the latitude and longitude coordinates and return an address. This is called *reverse geolocation*. To perform this, we will call the Azure Maps API. The API method we will call will return a **`ReverseSearchAddressResult`** object.

Later, the return result from this method will be consumed by the store search page to display the address of the user's current location.

As we did with **`SearchAddressResult`** in the previous chapter, the first step is to create a class, matching the properties of the **`ReverseSearchAddressResult`** object.

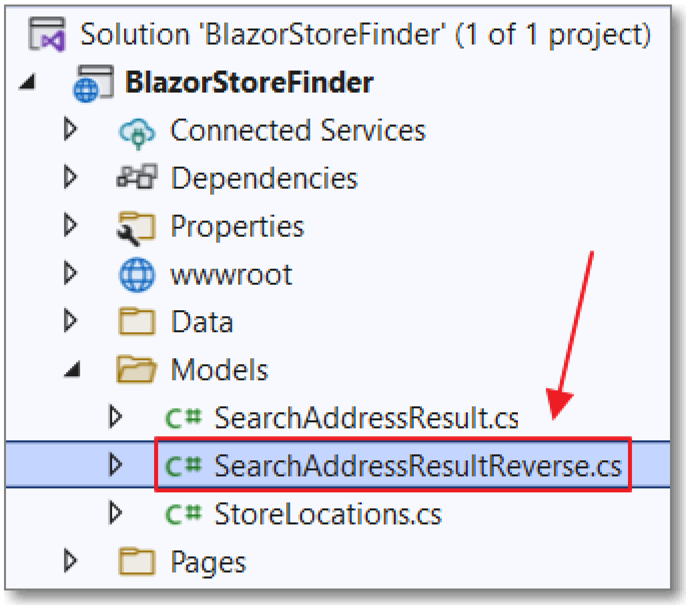


Figure 63: Create SearchAddressResultReverse.cs

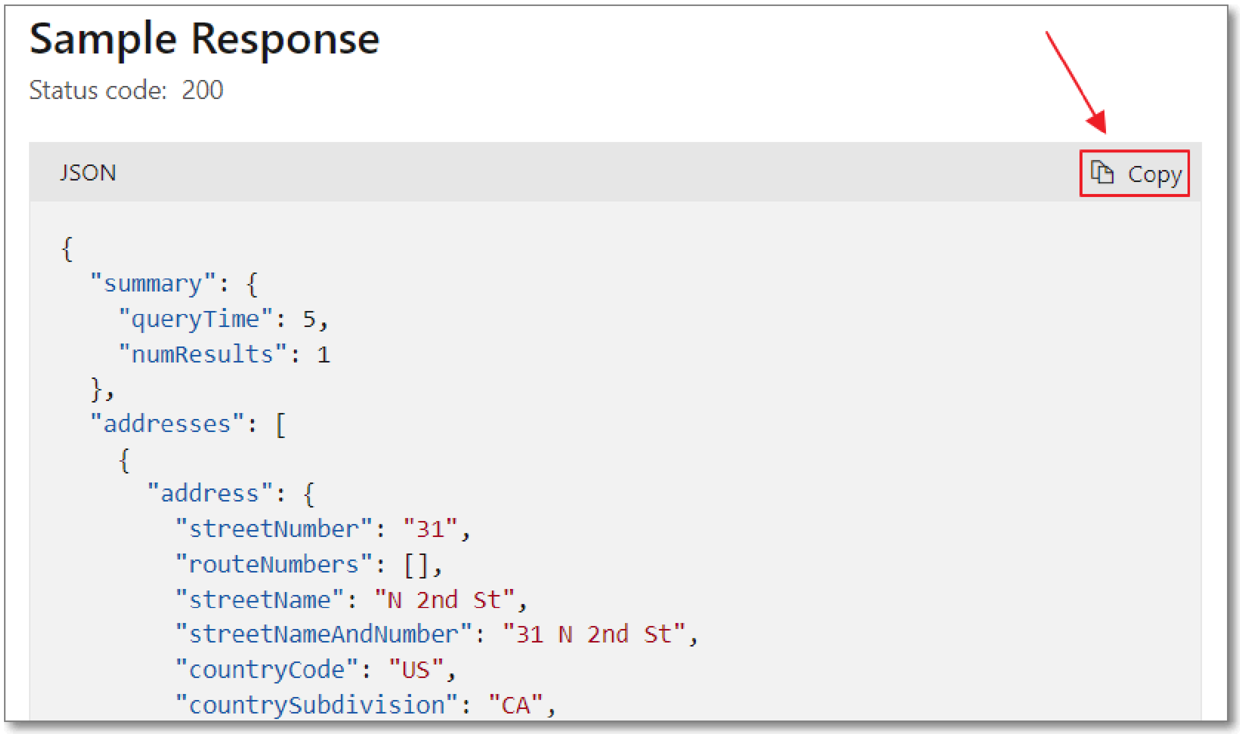
In the **Models** folder, add a **`SearchAddressResultReverse.cs`** class using the following code.

Code Listing 492: SearchAddressResultReverse.cs

```
#nullable disable
namespace BlazorStoreFinder.Reverse
{
}
```

In your web browser, navigate to the following URL:

https://docs.microsoft.com/en-us/rest/api/maps/search/get-search-address-reverse


Sample Response

Status code: 200

```
{
  "summary": {
    "queryTime": 5,
    "numResults": 1
  },
  "addresses": [
    {
      "address": {
        "streetNumber": "31",
        "routeNumbers": [],
        "streetName": "N 2nd St",
        "streetNameAndNumber": "31 N 2nd St",
        "countryCode": "US",
        "countrySubdivision": "CA",
```

Figure 64: Copy ReverseSearchAddressResult

Scroll down to the **Sample Response** section and click **Copy**.

In **Visual Studio**, select **Edit** from the toolbar, then **Paste Special** > **Paste JSON As Classes** to paste the contents inside the namespace declaration.

The pasted code will set that root class name to **`Rootobject`**. Change **`Rootobject`** to **`SearchAddressResultReverse`**.

Next, add the following **`using`** statement to the store location service class (**`StoreLocationService.cs`**).

Code Listing 50: BlazorStoreFinder.Reverse Using Statement

```
using BlazorStoreFinder.Reverse;
```

Finally, add the following method to the class that will call the Azure Maps API and return an address when passed latitude and longitude coordinates.

Code Listing 51: GeocodeReverse

```
public async Task<SearchAddressResultReverse>
    GeocodeReverse(Coordinate paramCoordinate)
{
    SearchAddressResultReverse result = new SearchAddressResultReverse();

    // Create a HTTP client to make the REST call.

    // Search - Get search address reverse
    // https://bit.ly/3Nuz5cP
    using (var client = new System.Net.Http.HttpClient())
    {
        // Get an access token from AuthService.
        var AccessToken = await AuthService.GetAccessToken();

        client.DefaultRequestHeaders.Accept.Clear();
        client.DefaultRequestHeaders.Accept.Add(
            new MediaTypeWithQualityHeaderValue("application/json"));

        // Pass the Azure Maps client ID.
        client.DefaultRequestHeaders.Add(
            "x-ms-client-id", AuthService.ClientId);

        // Pass the access token in the auth header.
        client.DefaultRequestHeaders.Authorization =
        new System.Net.Http.Headers.AuthenticationHeaderValue(
            "Bearer", AccessToken);

        // Build the URL.
        StringBuilder sb = new StringBuilder();

        // Request an address search.
        sb.Append(
            "https://atlas.microsoft.com/search/address/reverse/json?");
        // Specify the API version and language.
        sb.Append("api-version=1.0");
        // Pass latitude.
        sb.Append($"&query={paramCoordinate.X}");
        // Pass longitude.
        sb.Append($",{paramCoordinate.Y}");

        // Set the URL.
        var url = new Uri(sb.ToString());

        // Call Azure Maps and get the response.
        var Response = await client.GetAsync(url);

        // Read the response.
        var responseContent =
            await Response.Content.ReadAsStringAsync();

        result =
            JsonConvert.DeserializeObject<SearchAddressResultReverse>(
                responseContent);
    }

    return result;
}
```

Update store location service

The final method we will add to the store location service will accept latitude and longitude coordinates, search the database of store locations, and return the nearest locations and their distances from the search location.

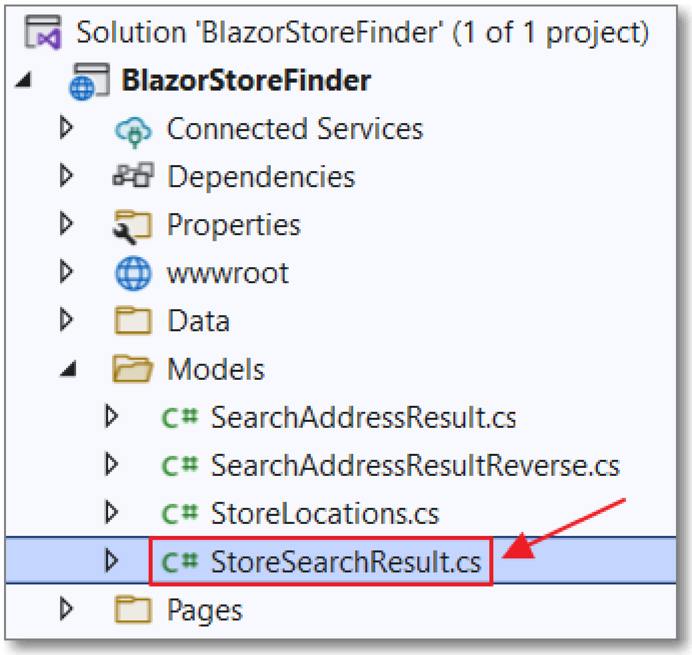


Figure 65: Create StoreSearchResult.cs

In the **Models** folder, add a new class called **`StoreSearchResult.cs`** using the following code.

Code Listing 52: StoreSearchResult

```
namespace BlazorStoreFinder
{
    public class StoreSearchResult
    {
        public string? LocationName { get; set; }
        public string? LocationAddress { get; set; }
        public double LocationLatitude { get; set; }
        public double LocationLongitude { get; set; }
        public double Distance { get; set; }
    }
}
```

Now, to allow the search to be performed, add the following code to the store location service class (**`StoreLocationService.cs`**).

Code Listing 56: GetNearbyStoreLocations

```
public List<StoreSearchResult> GetNearbyStoreLocations(
    Coordinate paramCoordinate)
{
    List<StoreSearchResult> colStoreLocations =
        new List<StoreSearchResult>();

    // Using a raw SQL query because sometimes NetTopologySuite
    // cannot properly translate a query.

    StringBuilder sb = new StringBuilder();

    // Set distance to 25 miles.
    sb.Append("declare @Distance as int = 25 ");
    // Declare the coordinate.
    sb.Append($"declare @Latitude as nvarchar(250) = ");
    sb.Append($"'{paramCoordinate.Y}' ");
    sb.Append($"declare @Longitude as nvarchar(250) = ");
    sb.Append($"'{paramCoordinate.X}' ");
    // Declare the geography.
    sb.Append("declare @location sys.geography ");
    sb.Append(" ");
    // Set the geography to the coordinate.
    sb.Append("set @location = ");
    sb.Append("geography::STPointFromText('POINT");
    sb.Append("(' + @Longitude + ' ' + @Latitude + ')', 4326) ");
    sb.Append(" ");
    // Search for store locations within the distance.
    sb.Append("SELECT ");
    sb.Append("[LocationName], ");
    sb.Append("[LocationAddress], ");
    sb.Append("[LocationLatitude], ");
    sb.Append("[LocationLongitude], ");
    sb.Append("[LocationData].STDistance(@location) ");
    sb.Append("/ 1609.3440000000001E0 AS [DistanceInMiles] ");
    sb.Append("FROM [StoreLocations] ");
    sb.Append("where [LocationData].STDistance(@location) ");
    sb.Append("/ 1609.3440000000001E0 < @Distance ");
    sb.Append("order by [LocationData].STDistance(@location) ");
    sb.Append("/ 1609.3440000000001E0 ");

    using (SqlConnection connection =
    new SqlConnection(_context.Database.GetConnectionString()))
    {
        SqlCommand command = new SqlCommand(sb.ToString(), connection);
        connection.Open();
        SqlDataReader reader = command.ExecuteReader();

        while (reader.Read())
        {
            colStoreLocations.Add(new StoreSearchResult
            {
                LocationName =
                reader["LocationName"].ToString(),

                LocationAddress =
                reader["LocationAddress"].ToString(),

                LocationLatitude =
                Convert.ToDouble(reader["LocationLatitude"]),

                LocationLongitude =
                Convert.ToDouble(reader["LocationLongitude"]),

                Distance =
                double.Parse(reader["DistanceInMiles"].ToString())
            });
        }

        reader.Close();
    }

    return colStoreLocations;
}
```

Search for store locations

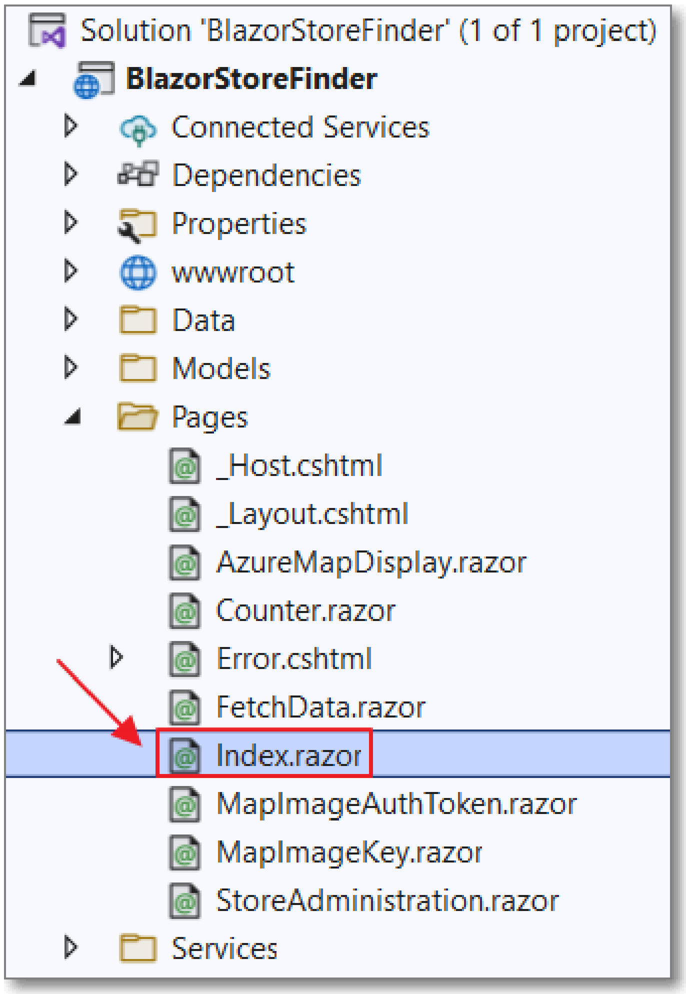


Figure 66: Index.razor

We will construct the store search page on the existing **Index.razor** page.

Replace all the existing code with the following code.

Code Listing 53: Index Control

```
@page "/"
@using BlazorStoreFinder
@using NetTopologySuite.Geometries
@using Syncfusion.Blazor.Layouts
@using Syncfusion.Blazor.Buttons
@using Syncfusion.Blazor.Inputs
@using Syncfusion.Blazor.SplitButtons
@using Syncfusion.Blazor.Lists
@using Darnton.Blazor.DeviceInterop.Geolocation;
```

```
@using AzureMapsControl.Components.Map
@inject IGeolocationService GeolocationService
@inject StoreLocationService _StoreLocationService
@inherits OwningComponentBase<StoreLocationService>
<h4>Blazor Store Finder</h4>

@code {
    MapEventArgs? myMap;
    SfListView<StoreSearchResult>? StoreSearchResultListBox;
    protected GeolocationResult? CurrentPositionResult { get; set; }

    List<StoreSearchResult> colStoreLocations =
    new List<StoreSearchResult>();

    Coordinate CurrentCoordinate = new Coordinate();
    string CurrentLocation = "";
    bool searching = false;

}
```

Add the following code to the UI section. This code uses a Syncfusion **`TextBox`** component to allow the user to enter an address to use for the search. It also uses a Syncfusion **`ListView`** component to display a list of the store locations that are the result of the search.

Finally, it uses an **`AzureMap`** control to display the stores on an interactive map.

Code Listing 54: Search Results and Azure Map

```
<div class="row" style="width:1500px">
<div class="col-xs-2 col-sm-2 col-lg-2 col-md-2">
    <div class="row">
        <div class="col-xs-9 col-sm-9 col-lg-9 col-md-9"
        style="margin-top:10px;">
            <SfTextBox Placeholder="Location"
            @bind-Value="CurrentLocation"></SfTextBox>
        </div>
        <div class="col-xs-3 col-sm-3 col-lg-3 col-md-3">
            @if (searching)
            {
                <div class="spinner-border text-primary" role="status"
                    style="margin-top:8px;"></div>
            }
            else
            {
                <button class="e-control e-btn e-lib" @onclick="Search"
                    style="width:auto; margin-top:8px;">
                    <span class="oi oi-magnifying-glass"></span>
                </button>
            }
        </div>
    </div>
    <br />
    @if (colStoreLocations.Count > 0)
    {
        <SfListView @ref="StoreSearchResultListBox"
                    DataSource="@colStoreLocations"
                    Height="450px" CssClass="e-list-template listview-
template">
            <ListViewFieldSettings TValue="StoreSearchResult"
Id="LocationName"
                                Text="Text"
Child="Child"></ListViewFieldSettings>
            <ListViewTemplates TValue="StoreSearchResult">
                <Template>
                    @{
                    StoreSearchResult currentData =
(StoreSearchResult)context;

                    <div class="e-list-wrapper e-list-avatar e-list-
multi-line">
                        <span class="e-avatar"><span class="oi oi-
globe"></span>
                        </span>
                        <span class="e-list-item-header">
                        @currentData.LocationName</span>
                        <span class="e-list-item-header">
                        @currentData.LocationAddress</span>
                        <span class="e-list-content">
                            @currentData.Distance.ToString("F") miles
                        </span>

                    </div>
                }
            </Template>
        </ListViewTemplates>
    </SfListView>
    }
</div>

<div class="col-xs-10 col-sm-10 col-lg-10 col-md-10">
    <AzureMap Id="map"
                CameraOptions="new CameraOptions { Zoom = 10 }"
                StyleOptions=
                "new StyleOptions { ShowLogo = false, ShowFeedbackLink =
false }"
                EventActivationFlags=
"MapEventActivationFlags.None().Enable(MapEventType.Ready)"
                OnReady="OnMapReadyAsync" />
</div>
</div>
```

Add the following method to the code section. This retrieves the user's current latitude and longitude. It will then pass it to the **ReverseGeocode** method created earlier. The address returned will be set as the default search address, and the **Search** method (to be implemented later) will perform a search of stores in the database near that address.

Code Listing 55: OnAfterRenderAsync

```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        // Get current location
        // will cause a popup to show to ask permission.
        CurrentPositionResult =
        await GeolocationService.GetCurrentPosition();

        if (CurrentPositionResult.IsSuccess)
        {
            // Get latitude and longitude.
            string? CurrentLatitude =

CurrentPositionResult.Position?.Coords?.Latitude.ToString("F2");

            string? CurrentLongitude =

CurrentPositionResult.Position?.Coords?.Longitude.ToString("F2");

            // Set latitude and longitude
            // (to be consumed by GetTile() method)
            if (CurrentLatitude != null && CurrentLongitude != null)
            {
                CurrentCoordinate.X = Convert.ToDouble(CurrentLatitude);
                CurrentCoordinate.Y = Convert.ToDouble(CurrentLongitude);

                // Reverse geocode coordinate and set location.
                var SearchAddressResult =
                await Service.GeocodeReverse(CurrentCoordinate);

                if
(SearchAddressResult.addresses[0].address.freeformAddress
                    != null)
                {
                    CurrentLocation =
SearchAddressResult.addresses[0].address.freeformAddress;

                    // Search for nearby stores.
                    await Search();
                }
            }

            StateHasChanged();
        }
    }
}
```

Next, add the following method that will run when the **`AzureMap`** control fires its **`OnReady`** event. This allows programmatic access to the **`AzureMaps`** control through the **`myMap`** variable.

Code Listing 60: OnMapReadyAsync

```
    public async Task OnMapReadyAsync(MapEventArgs eventArgs)
    {
        await Task.Run(() => myMap = eventArgs);
    }
```

Finally, add the following method that will search for store locations in the database and populate the **`colStoreLocations`** collection that lists results on the page. This will also show a map marker for the current location and icons for each store location on the **`AzureMap`** control.

Code Listing 56: Search Store Locations

```
public async Task Search()
{
    searching = true;
    StateHasChanged();

    // Clear location results.
    colStoreLocations = new List<StoreSearchResult>();

    // Geocode address.
    Coordinate CurrentCoordinate =
    await Service.GeocodeAddress(CurrentLocation);

    if (CurrentCoordinate != null)
    {
        // Find nearby stores.
        colStoreLocations =
        Service.GetNearbyStoreLocations(CurrentCoordinate);
    }

    searching = false;
    StateHasChanged();

    if (myMap != null && CurrentCoordinate != null)
    {
        // Center map to current location.
        await myMap.Map.SetCameraOptionsAsync(
                options => options.Center =
                new AzureMapsControl.Components.Atlas.Position
                (CurrentCoordinate.X, CurrentCoordinate.Y));

        // Add icon for current location.
        await myMap.Map.ClearHtmlMarkersAsync();

        var HomeIcon = new
AzureMapsControl.Components.Markers.HtmlMarker(
            new AzureMapsControl.Components.Markers.HtmlMarkerOptions
                {
                    Position = new AzureMapsControl.Components.Atlas
                               .Position(
                                   CurrentCoordinate.X,
CurrentCoordinate.Y
                               ),

                    Draggable = false,
                    Color = "#FF0000"
                });

        await myMap.Map.AddHtmlMarkersAsync(HomeIcon);

        // Add icons for search results.
        foreach (var store in colStoreLocations)
        {
            var StoreIcon = new
AzureMapsControl.Components.Markers.HtmlMarker(
                new AzureMapsControl.Components.Markers.HtmlMarkerOptions
                    {
                        Position = new AzureMapsControl.Components.Atlas
                                   .Position(
                                       store.LocationLongitude,
store.LocationLatitude
                                   ),

                        Draggable = false,
                        Color = "#0000FF"
                    });
            await myMap.Map.AddHtmlMarkersAsync(StoreIcon);
        }
    }
}
```

Run the application and navigate to the **Home** page.

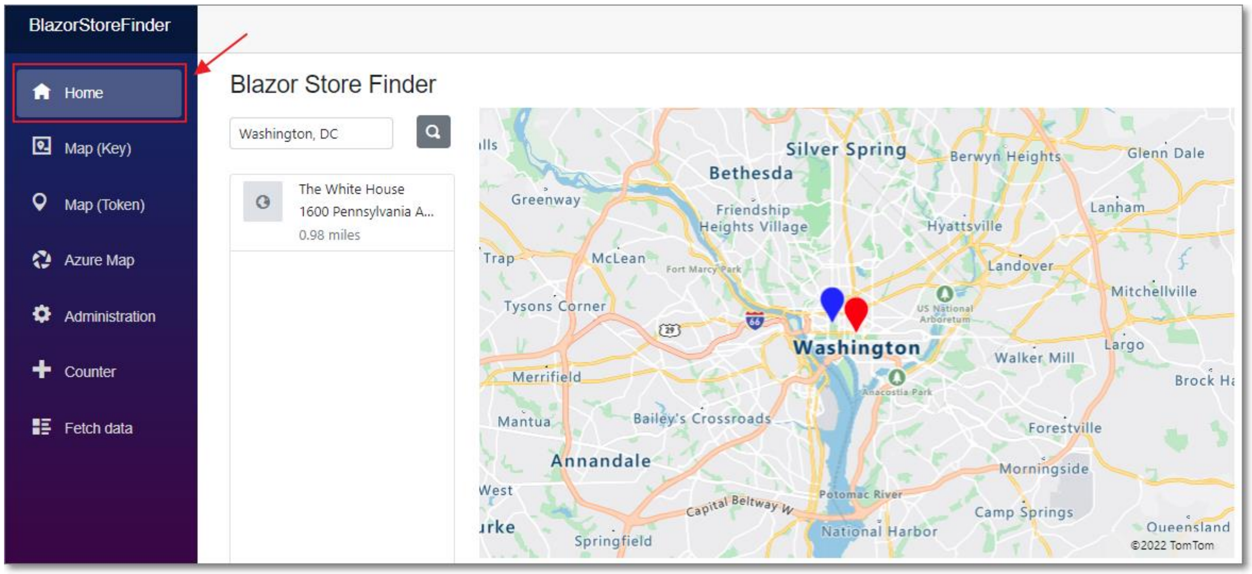


Figure 67: Search for Store Locations

The search box will be populated with a location that matches your current location. The map will display, centered at the current location with a red icon. Blue icons will display for each nearby store location from the database.

Re-center map on selected store

In the final example, to demonstrate how we can dynamically interact with the **`AzureMaps`** control, we will center the map on a store location when it is selected in the search result list.

In the UI, change the following line of code.

Code Listing 57: Update Div

```
~~<div class="e-list-wrapper e-list-avatar e-list-multi-line">~~
```

To the following code, add an **`onclick`** event handler that will call the **`OnStoreSelect`** method (to be created later), passing the currently selected store location.

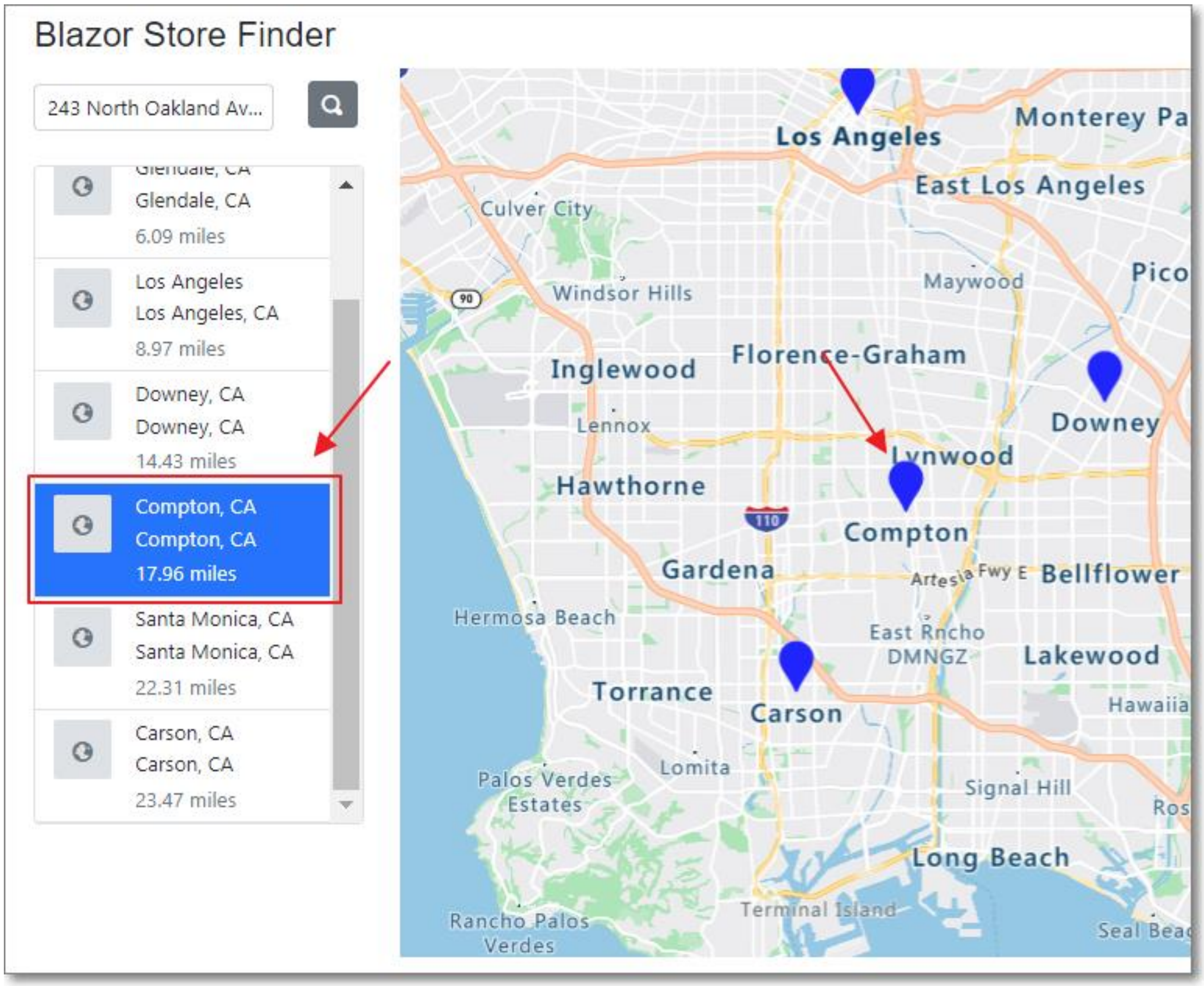
Code Listing 58: Add Onclick Handler to Div

```
<div class="e-list-wrapper e-list-avatar e-list-multi-line"
@onclick="(e => OnStoreSelect(currentData))">
```

Finally, add the following **`OnStoreSelect`** method that will center the **`AzureMaps`** control on the selected icon.

Code Listing 59: OnStoreSelect

```
async Task OnStoreSelect(StoreSearchResult SearchResult)
{
    // Center map on store.
    if (myMap != null && CurrentCoordinate != null)
    {
        await myMap.Map.SetCameraOptionsAsync(
                options => options.Center =
                new AzureMapsControl.Components.Atlas.Position
                (SearchResult.LocationLongitude,
                    SearchResult.LocationLatitude));
    }
}
```



Blazor Store Finder

243 North Oakland Av...

Glendale, CA
Glendale, CA
6.09 miles

Los Angeles
Los Angeles, CA
8.97 miles

Downey, CA
Downey, CA
14.43 miles

Compton, CA
Compton, CA
17.96 miles

Santa Monica, CA
Santa Monica, CA
22.31 miles

Carson, CA
Carson, CA
23.47 miles

Figure 68: Center Map on Selected Store Location

Run the application.

When you click a store location in the search results on the left-hand side of the page, the map will center on that location.

We've now seen how Azure Maps, together with Syncfusion controls, enables you to create sophisticated applications using Blazor. Try it for yourself.