

BlazorDemoUI

About

Home

Counter

Fetch data






People

Weather forecast

This component demonstrates fetching data from a REAL WEATHER Service.

333 W Broad St, Columbus, OH 43215

Get Forecast


	Date	Temp	Wind	Summary
	This Afternoon	68	14 mph	Mostly sunny, with a high near 68. North wind around 14 mph.
	Monday	73	2 to 7 mph	Sunny, with a high near 73. South wind 2 to 7 mph.
	Tuesday	86	3 to 10 mph	Mostly sunny, with a high near 86. Southwest wind 3 to 10 mph.
	Wednesday	90	6 to 9 mph	A chance of rain showers between 11am and 2pm, then a chance of showers and thunderstorms. Partly sunny, with a high near 90. Chance of precipitation is 40%.
	Thursday	85	2 mph	A chance of showers and thunderstorms. Partly sunny, with a high near 85. Chance of precipitation is 50%.

Data Sources:

Geocoding Service from Census.gov

National Weather Service using api.weather.gov

Extension for chrome:



JSON Viewer 0.8.17

Validates and makes JSON documents easy to read. Open source.

ID: aimiinbnnkboelefkjlenlgimcabobli

Inspect views background page,1 more...

Details

Remove

Creating a Real Weather Forecast in the Blazor Template

This Blazor (server side) code is (just about) the minimum required to access a real weather forecast within Blazor.

Introduction

As someone new to Blazor, .NET, C#, and json, I have found that many of the examples in tutorials were too complicated to be helpful to a real beginner.

This is not an example of best coding practices! But, it illustrates how pages, classes, httpclients, and services interact with json data to produce a result, sort of putting this all together in a very simplistic, but real example.

This code can be “plugged” right into the Blazor template to create a Real weather forecast.

Notes:

- (1) the National Weather Service API appears to be a bit under construction, and some areas (particularly metro NY) are coming up with missing weather data. Other services are available for free weather APIs; hopefully this example can illustrate enough to be able to use any of them.
- (2) The US Census geocoding app is also limited in its ability to recognize addresses, especially compared to Google. But, it doesn't (as yet) require an API key, and it is free, so it was also a good place to start to show how this could work.

My goal was to modify the Blazor Template as little as possible to make this work, and keep it as simple as possible without totally hard-coding in everything (such as location and date).

Overview of Steps

I want the User to enter an address (in an EditForm), press a button, and have the system display the forecast information that I have pre-selected in a table.

I. User Interface

- a. Use an **EditForm** section to capture the user-entered address, with a **submit** button.
- b. Display a 5-day forecast, just for the daytime forecasts (or a nice error message).

II. Processing

- a. Link the form to the task to get the forecast ("**OnValidSubmit**").
- b. Pass the user-entered address to the Weather Real Forecast Service task as a parameter.
- c. Execute code in another file (a "Service") to get the forecast. Because of the APIs I have decided to use, this Service must:
 - i. Get GPS coordinates of selected address
 - ii. Get NWS forecast office and grid
 - iii. Get the corresponding forecast
- d. Send the selected forecast data back to the user page.

These 3 steps are similar processes, using different APIs and parameters.

III. Data Models

- a. myLUAddressModel : An in-page **model** for the user-entered address, to "bind" it to the code, so that I can use it as a parameter when I call the service.
- b. "WeatherRealForecast" : A model of the parameters available from the weather service which may be used in the output. The JSON deserializer parses all items as strings, except for the boolean value of IsDateTime (true/false) from the forecast API.

<https://geocoding.geo.census.gov/geocoder/locations/onlineaddress?address=333+W+Broad+St,+Columbus,+OH+43215&benchmark=9&format=json>

Output (additional lines not used deleted for brevity). You can enter this above url into Google Chrome with JSON extension to see these results:

```
{
  • result:
    {
      ○ input:
        {
          ▪ benchmark:
            {
              ▪ id: "9",
              ▪ benchmarkName: "Public_AR_Census2010",
              ▪ benchmarkDescription: "Public Address Ranges - Census 2010 Benchmark",
              ▪ isDefault: false,
            },
          ▪ address:
            {
              ▪ address: "333 W Broad St, Columbus, OH 43215"
            },
        },
      ○ addressMatches:
        [
          ▪ {
            ▪ matchedAddress: "333 E Broad St, COLUMBUS, OH, 43215",
            ▪ coordinates:
              {
                ▪ x: -82.99035,
                ▪ y: 39.963387,
              },
            ▪ tigerLine:
              {
                ▪ tigerLineId: "218649171",
                ▪ side: "R",
              },
            ▪ addressComponents:
              {
                ▪ fromAddress: "399",
                ▪ toAddress: "301",
                ▪ preQualifier: "",
                ▪ preDirection: "E",
                ▪ preType: "",
                ▪ streetName: "Broad",
                ▪ suffixType: "St",
                ▪ suffixDirection: "",
                ▪ suffixQualifier: "",
                ▪ city: "COLUMBUS",
                ▪ state: "OH",
                ▪ zip: "43215",
              },
          },
        ]
    }
}
```

<https://api.weather.gov/points/39.963387,-82.99035>

Output (additional lines above and below not used deleted for brevity). You can enter this above url into Google Chrome with JSON extension to see these results:

```
},
- properties: {
  @id: "https://api.weather.gov/points/39.9634,-82.9903999",
  @type: "wx:Point",
  cwa: "ILN",
  forecastOffice: "https://api.weather.gov/offices/ILN",
  gridX: 84,
  gridY: 80,
  forecast: "https://api.weather.gov/gridpoints/ILN/84,80/forecast",
  forecastHourly: "https://api.weather.gov/gridpoints/ILN/84,80/forecast/hourly",
  forecastGridData: "https://api.weather.gov/gridpoints/ILN/84,80",
  observationStations: "https://api.weather.gov/gridpoints/ILN/84,80/stations",
- relativeLocation: {
  type: "Feature",
  - geometry: {
    type: "Point",
    - coordinates: [
      -82.984882,
      39.983938,
    ]
  }
}
```

<https://api.weather.gov/gridpoints/ILN/84,80/forecast>

Output (additional lines above and below not used deleted for brevity). You can enter this above url into Google Chrome with JSON extension to see these results:

```
    unitCode: "unit:m",
  },
  - periods: [
    - {
      number: 1,
      name: "This Afternoon",
      startTime: "2020-05-31T14:00:00-04:00",
      endTime: "2020-05-31T18:00:00-04:00",
      isDaytime: true,
      temperature: 68,
      temperatureUnit: "F",
      temperatureTrend: null,
      windSpeed: "14 mph",
      windDirection: "N",
      icon: "https://api.weather.gov/icons/land/day/sct?size=medium",
      shortForecast: "Mostly Sunny",
      detailedForecast: "Mostly sunny, with a high near 68. North wind around 14 mph.",
    },
    - {
      number: 2,
      name: "Tonight",
      startTime: "2020-05-31T18:00:00-04:00",
      endTime: "2020-06-01T06:00:00-04:00",
      isDaytime: false,
      temperature: 43,
      temperatureUnit: "F",
      temperatureTrend: null,
      windSpeed: "2 to 12 mph",
      windDirection: "NE",
      icon: "https://api.weather.gov/icons/land/night/few?size=medium",
      shortForecast: "Mostly Clear",
      detailedForecast: "Mostly clear, with a low around 43. Northeast wind 2 to 12 mph.",
    },
    - {
      number: 3,
```

WeatherRealForecast.cs

```
namespace BlazorDemoUI.Data
{
    public class WeatherRealForecast
    {
        // THESE ARE THE FIELDS FROM THE JSON FILE THAT I MIGHT WANT TO USE
        public string Number { get; set; } // I HAD TO START EACH WITH A CAPITAL LETTER
        public string Name { get; set; }
        public string StartTime { get; set; }
        public string EndTime { get; set; }
        public bool IsDaytime { get; set; }
        public string Temperature { get; set; }
        public string TemperatureUnit { get; set; }
        public string TemperatureTrend { get; set; }
        public string WindSpeed { get; set; }
        public string WindDirection { get; set; }
        public string Icon { get; set; }
        public string ShortForecast { get; set; }
        public string DetailedForecast { get; set; }
    }
}
```

RealWeatherFetchData.razor

```
@page "/fetchdata2"

@using BlazorDemoUI.Data
@inject WeatherRealForecastService ForecastService 1

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a REAL WEATHER Service.</p>

<EditForm Model=@newAddress OnValidSubmit="@ExecuteGetForecast">
    <InputText id="Name" size="100" placeholder="Address to Look Up" @bind- 2
    Value="newAddress.myLUAddress" :null /><br /><br />
    <button type="submit" class="btn btn-primary">Get Forecast</button>
</EditForm>
<br />

@if (forecasts == null)
{
    <p><em></em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th></th>
                <th>Date</th> 3
                <th>Temp</th>
                <th>Wind</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                if (forecast.IsDaytime) // I only want daytime forecasts 5
                {
                    <tr>
                        <td></td>
                        <td>@forecast.Name</td>
                        <td>@forecast.Temperature</td>
                        <td>@forecast.WindSpeed</td> 4
                        <td>@forecast.DetailedForecast</td>
                    </tr>
                }
                else { }
            }
        </tbody>
    </table>
}
```


RealWeatherFetchData.razor (continued, this is the @code section)

```
@code {
private WeatherRealForecast[] forecasts;

public class myLUAddressModel // this class lets me use an input field at the top to
{
public string myLUAddress { get; set; } = null;    // have user input an address
}

public myLUAddressModel newAddress = new myLUAddressModel();
        // my class instance is named newAddress- referenced by the EditForm above

public async Task ExecuteGetForecast()
{
    forecasts = await ForecastService.ExecuteLookupForecast(newAddress.myLUAddress);
}

//    protected override async Task OnInitializedAsync() THIS WAS THE ORIGINAL TASK FROM THE TEMPLATE
//    {
//        forecasts = await ForecastService.ExecuteLookupForecast()    BEFORE I ADDED THE ADDRESS
//        PARAMETER AND BUTTON
//    }
}
```

6

7

1. Inject allows me to use a class of C# code that is in another file, as a “service”. In my Blazor page, I am going to “call it” (when I press the button) using the pseudonym ForecastService. That class/service has a task called ExecuteLookupForecast, which I will refer to as ForecastService.ExecuteLookupForecast.
2. The EditForm lets me accept user input (I skipped the validation options). The “@” items link my form and the user input to the C# code below.
3. I made a few changes to the template headings so I could use different data from the real weather forecast. Also, no need for a heading for the weather image.
4. The values in each column come from the WeatherRealForecast array I am calling forecasts on this page, which were returned by Task ExecuteLookupforecast in the ForecastService service. Each of these fields is constructed by the WeatherRealForecast model.
5. I only want daytime forecasts to appear, so I am evaluating the IsDaytime field of each row in the array to see if it is true, only displaying the row of data if it is true. Otherwise (else) it will display nothing.
6. I define myLUAddress as a string, and the only element in myLUAddressModel. Defining this within a Model allows me to use EditForm to obtain the value. The “link” between EditForm and the model is newAddress, defined here as an instance of the myLUAddressModel.
7. Finally, I call ExecuteLookupForecast to get the forecast info, sending the user-entered address. The original template’s call to get the forecast was “OnInitializedAsync”, but here I need to have the user enter the address and then press the button, so I changed it to a user-initiated task.

WeatherRealForecastService.cs

```
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Net.Http;
using System.Net.Http.Headers;
using Newtonsoft.Json.Linq; // NEEDED NEWTONSOFT JSON TO PARSE THE JSON OUTPUTS

namespace BlazorDemoUI.Data
{
    public class WeatherRealForecastService
    {
        1 HttpClient clientgeocode = new HttpClient(); // to get GPS coordinates
        HttpClient clientGetGrid = new HttpClient(); // to get NWS forecast office and grid
        HttpClient clientforecast = new HttpClient(); // to get the forecasts

        string strLongitude { get; set; } // these are set within a try/catch section so must be
        string strLatitude { get; set; } // declared out of that section in order to be used later

        2 public async Task<WeatherRealForecast[]> ExecuteLookupForecast(string mylocation)
        {
            // 1. GET/READ/PARSE GPS COORDINATES FOR LOCATION
            string myAddress = "?address=" + mylocation.Replace(" ", "+"); 3
            string urlgeocode =
                "https://geocoding.geo.census.gov/geocoder/locations/onlineaddress{0}&benchmark=9&format=json";

            4 try
            {
                var responsegeocode = await clientgeocode.GetAsync(string.Format(urlgeocode, myAddress));
                string myresultgeocode = await responsegeocode.Content.ReadAsStringAsync();
                //Console.WriteLine("my resultgeocode is " + myresultgeocode);

                dynamic datageocode = JObject.Parse(myresultgeocode);
                strLongitude = datageocode.result.addressMatches[0].coordinates.x; 5
                strLatitude = datageocode.result.addressMatches[0].coordinates.y;

            }
            catch
            {
                6 return Enumerable.Range(0, 1).Select(index => new WeatherRealForecast
                {
                    IsDaytime = true,
                    Name = "Location not found. Try entering only the Street Address and Zip Code"
                }).ToArray();
            }
        }

        // 2. GET/READ/PARSE NWS GRID AND OFFICE FOR THOSE COORDINATES
        string urlgetgrid = "https://api.weather.gov/points/{0},{1}";
        string myurlgrid = string.Format(urlgetgrid, strLatitude, strLongitude);
        Console.WriteLine("my myurl is " + myurlgrid);
        clientGetGrid.DefaultRequestHeaders.Add("User-Agent", "any text string");
        // will soon require an API key
        7 var responsegrid = await clientGetGrid.GetAsync(myurlgrid);
        string myresultgrid = await responsegrid.Content.ReadAsStringAsync();
        Console.WriteLine("myresultgrid is " + myresultgrid);

        dynamic datagrid = JObject.Parse(myresultgrid);
        string strForecastUrl = datagrid.properties.forecast;
        Console.WriteLine("my strForecastURL is " + strForecastUrl);
    }
}
```

```
// 3. GET/READ/PARSE FORECAST FOR THAT OFFICE AND GRID, AND RETURN ARRAY TO PAGE

try
{
    clientforecast.DefaultRequestHeaders.Add("User-Agent", "any text string"); // will
soon require an API key
    var response = await clientforecast.GetAsync(string.Format(strForecastUrl));
    string myresult = await response.Content.ReadAsStringAsync();
    Console.WriteLine("my result is " + myresult);

    dynamic data = JObject.Parse(myresult);

    return Enumerable.Range(0, 10).Select(index => new WeatherRealForecast
    {
        Name = data.properties.periods[index].name,
        Temperature = data.properties.periods[index].temperature,
        WindSpeed = data.properties.periods[index].windSpeed,
        DetailedForecast = data.properties.periods[index].detailedForecast,
        Icon = data.properties.periods[index].icon,
        IsDaytime = data.properties.periods[index].isDaytime
    }).ToArray();
}
catch
{
    return Enumerable.Range(0, 1).Select(index => new WeatherRealForecast
    {
        IsDaytime = true,
        Name = "Forecast not available. Try looking out your window."
    }).ToArray();
}
}
```

(end of WeatherRealForecastService)

1. Define the HttpClient. I created one for each of my API calls (I could have consolidated into at least just 2). Need `using System.Net.Http;` and `using System.Net.Http.Headers;` for this to work.
2. This is my Task, which takes in an address string (which the template did not have) and outputs a forecast array (similar to the template).
3. String manipulation to be compatible with the census geocoding API. Spaces not allowed, replaced with + signs. Address will be parameter zero.
4. Execute the call to the API to get the GPS coordinates. The resulting text response in JSON format has much more information than just the coordinates, all of which is "dumped" into `myresultgeocode`. If you view this string in Google Chrome with the JSON extension installed, hovering over/clicking on the value will show the fully qualified reference name in the bottom status bar, e.g., `result.addressMatches[0].coordinates.x` to show you how to reference to pick out those items.
5. Pick out just the x (longitude) and y (latitude) coordinates from the JSON string by item name. This is a `Newtonsoft.Json` command.
6. In case the geocoder couldn't find the GPS coordinates. This system has a few different options for collecting the address info, and with additional validation included can be made to be more forgiving.

7. This gets the forecast office, box and grid string to use that correspondes to the GPS latitude and longitude. The NWS web site says they will soon be moving to requiring an APIKEY, but currently only needs a dummy header with the tag "User-Agent". So, I assign the header to the httpclient, and then as in the 1st API call, I
 - create the url string,
 - run the API query call,
 - get the results into a JSON string, and
 - pull out the "forecast" parameter, which is a pre-populated url string with the office, grid X and grid y already built in.
8. This takes the forecast URL, with the User-Agent header added to the httpclient (I could have used the same httpclient), and again
 - use the url string,
 - run the API query call,
 - get the results into a JSON string, and
 - pull out all of the forecast fields I want to show, and load them into an array of type WeatherRealForecast (my model).
9. Catch and send back an error message rather than havwe the system spit out an ugly error message.
10. Before any of this will work, add the service to the Startup.cs file:

Modify Startup.cs to add the RealForecastService:

```
0 references
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddServerSideBlazor();
    services.AddSingleton<WeatherForecastService>();
    services.AddSingleton<WeatherRealForecastService>();
}
```