

# Dynamic Weather Forecasting for Road Trips: Combining GPS Routes with Weather Forecasts

Colin McLean

Dept of Computer Science

MSU Moorhead

Moorhead, Minnesota

colin.mclean@go.mnstate.edu

**Abstract**— Making decisions about long distance road trips can be a complex and stressful task when dealing with potentially hazardous weather along the path. My application aims to ease this burden by creating a user-friendly tool which integrates GPS routing data with real-time weather forecasting data. By matching up the predicted locations the user will be at each 15-minute interval along their path with the weather conditions at that specific time and location, the application offers an accurate weather forecast for the user which accounts for the constant changing of their location. My app leverages modern web development tools like Blazor WebAssembly and Microsoft’s Azure Functions in combination with Azure Maps’ routes API and Open-Meteo’s weather API to create a simple and reliable tool. With this tool, a traveler will no longer have to do all the work of manually looking up their route and cross-referencing it with various locations along the way to get a grasp on what to expect. Instead, this application will do all the work for them and provide them with a unified route weather forecast tailored specifically to where they will be at each point in time along their path.

**Keywords**—*Blazor, Azure, GitHub Actions, route forecasting*

## I. INTRODUCTION

Traditional weather forecasting applications generally focus on providing the weather for one location at a time. This is great if you are staying in that same general area for the rest of the day, but if you are traveling this forecast will quickly become useless. This application aims to remedy this issue of handling dynamic locations when giving weather forecasts. By combining GPS data through route predictions with weather forecasts for locations along the route, this application will be able to give the user a much more accurate view of what weather they will expect to encounter over the course of their trip.

This information will help users be able to better make informed decisions about their travel choices. For example, if a user sees that there is heavy snow along their

route if they were to leave later in the day, they may instead choose an earlier departure time and avoid the bad weather.

While the creation of the application is the primary goal, an important secondary goal of this project is to learn more about emerging web technologies and gain experience in deploying a full-stack website. This technology will involve new trends such as CI/CD (continuous integration, continuous development), web assembly code, and serverless architecture.

## II. TECHNOLOGY USED

The core technologies used in this project are web APIs, Blazor WASM, Azure Functions, and GitHub. An important focus of this project was to choose technologies that integrate well together. This is why I chose to focus on technology all within the Microsoft ecosystem. Azure Map’s API allows for retrieving weather routes. Blazor WASM and Azure Functions are used for the client-server architecture of the website. And GitHub handles version control and deployment. All of these technologies are specifically designed to integrate well with each other.

A major benefit of this choice is that this allowed the application to be written in one consistent language, C#. Both the client and server are written almost entirely in C#. GitHub also is designed to work well with Azure and made it very easy to deploy all components of my web application to the Azure cloud.

### A. Web APIs

At the core of this application is the data. To create route forecasts, it is necessary to have both GPS data and weather data. To retrieve up-to-date data, it was necessary for this application to utilize Web APIs which can provide this information on demand.

The first API utilized is Azure Maps. Azure Maps is responsible for providing both route data and geocoding. Route data gives path information and estimated times for

the trip. Geocoding is responsible for converting location names and latitude/longitude coordinates.

The second key web API used here is from Open-Meteo. Open-Meteo provides a 15-minute forecast for the next 24 hours for a given latitude/longitude.

The combination of these two sources of data is what powers the route forecasting predictions.

### *B. Blazor WASM*

Blazor WASM is a modern web framework focused on creating interactive web applications using reusable components. WASM is short for WebAssembly. WebAssembly is a new web standard which first appeared around 2017 which allows for portable binary-code to be executed within the browser.

The power of using Blazor WASM is that it allows for an alternative to the monopoly of JavaScript on client-side applications. Blazor WASM allows the entire web app to be coded, and compiled from C# and run natively on the user's web browser without any plugins. This allows direct integration of any libraries from the wider .NET ecosystem.

Blazor's role in this web application is to handle the client-side interaction and processing. It is responsible for displaying the content of the weather forecast to the user, accepting user input for new requests, and it handles the processing of the actually combining the GPS and weather data together to create a singular weather forecast for a route. Blazor WASM's compiled nature allows this processing of information to be very fast and efficient. A key benefit of this is that the actual processing of the data can be passed to the client. This places less of a burden of the server side which can instead focus on just retrieving the data.

### *C. Azure Functions*

A major decision in developing this application was the choice of a server-side framework. The API keys necessary for retrieving data from the maps and weather APIs require security. The keys cannot be exposed to the client, Blazor WASM, so a server was required for holding onto this private information and making the requests.

A simple, efficient server technology was the goal and Azure Functions was the perfect solution. Azure Functions is a serverless compute platform. This technology, serverless, has been growing in popularity in recent years. A serverless platform allowed for simplification of the web application's architecture. Instead of having a traditional server, an Azure Function allows you to just create small pieces of code which will be executed on

demand by Azure's servers. This means that your code is only running when it is being used. This on-demand usage is in contrast to traditional servers where the server is continuously running all the time.

A major benefit of Azure Functions for this project is that it is extremely scalable and cost effective. Azure Functions can go from having no users accessing them to a sudden huge demand without an issue because it is Microsoft's larger Azure platform which is responsible for executing the code and not a specific server. This makes for a very cost effective approach for a small project like mine where the vast majority of the time it is not being used. And the code which is executed is generally only executed once per user visit of the web application. This small amount of computation needed made for a perfect match for Azure Functions.

The actual use of Azure Function within the web application is that it takes the start and end location of the user's route from the user input sent to it by the Blazor WASM client, and it gathers data from the APIs to return to the client. It first gets the GPS route from the start to the end, samples twenty equidistant waypoints along this route, and then makes a second GPS route request to get the exact distance and travel time between each waypoint. It then uses these twenty waypoints and their latitude/longitude data to make requests to the weather API to retrieve the 15-minute forecast for the next 24 hours at each waypoint. Once it has all this data, it combines it into one JSON file and returns it to the client so that the client can process it.

By minimizing the data processing on the Azure Function, it can shift the load from the server, which could cost money to compute on, to the client which from the perspective of the web application will be free computation. This allows for the web application to be very cost-efficient.

### *D. GitHub*

GitHub was the choice of version control and deployment platform for this project. GitHub is responsible for storing all of the code for the project and preserving its history. While this was a solo project, it could have also potentially had great benefit if this were collaborative and multiple people were contributing to it.

While the traditional usage of GitHub as storage and version control was very important to the project, one of the key uses of GitHub within this project was their GitHub Actions service. GitHub Actions allow for automated workflows which handle important aspects of the software development lifecycle such as compiling, testing, and deploying.

GitHub Actions was used in this project primarily for its deployment capability. Because GitHub is so well

integrated with the wider Microsoft ecosystem, it was very simple to create a workflow for deploying both the client and server-side. These are split into two folders, Weather.API and Weather.App, within the project structure and are deployed to separate locations. The client side which utilizes Blazor WASM must be deployed to Azure as a static web app while the server side must be deployed to Azure Functions as a function. After a short setup and linking to the corresponding Azure account, GitHub Actions automated workflow was able to easily handle the process of compiling the Blazor WASM app and deploying both it and the Azure Function to the production environment.

By setting up the project this way, it allows for CI/CD which is continuous integration/continuous development. With an automated and quick deployment like this, I can rapidly deploy new iterations of the web application throughout a single day. As soon as I push the new code to GitHub, it automatically deploys it and within minutes the new version is running live. This allows for new features and changes to be made continuously and easily.

### III. SYSTEM ARCHITECTURE

The architecture of this system is relatively simple and involves essentially connecting the user with weather and route GPS data sources. To do this, it's necessary to have the client-side component, Blazor WASM, responsible for interacting the user through their browser and displaying the results, and the server component, Azure Functions, responsible retrieving the data which the user needs.

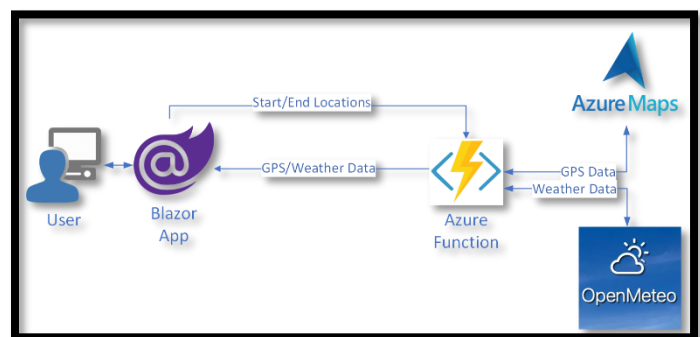
The user will make requests through Blazor, Blazor will retrieve data from Azure Functions, and then Blazor will display the results to the user.

#### A. Data Flow

The data flow is simple in this web application because there is very little user input involved and the retrieval only needs to occur once per route:

1. **User Input:** User will enter a starting location into the from address, such as "Fargo" and an ending location into the to address, such as "Minneapolis." Then to begin the processing, the user will click the "Get Weather Button"
2. **Start/End Location:** These two strings will be sent from the Blazor App to the Azure Function.
3. **Initial GPS Data:** The Azure Function will send the two locations to the Azure Maps API in order to retrieve data for the route.
4. **Process Route:** The Azure Function must next process the route by subdividing it into 20 equidistant waypoints.

5. **Detailed GPS Route Data:** The Azure Function must make a second call to the Azure Maps API to get a more detailed route that specifies the time between each waypoint.
6. **Reverse Geocoding:** The 20 waypoints which are latitude/longitude coordinates must be sent to the Azure Maps reverse geocoding API to retrieve a location name or address that can be used to later describe them on the web app.
7. **Weather Data:** The 20 waypoints must have each of their coordinates sent to the weather API to retrieve the 15 minute forecast for the next 24 hours.
8. **Combined Weather/Route Data:** Now that the Azure Function has gathered the detailed route, the names of the locations, and the waypoints along the way, it will pass all this information back to the Blazor App.
9. **Processing Weather/Route Data:** Finally the Blazor App has all the necessary information and it can lookup the weather data for each location along the route at the corresponding time and create a single 15 minute forecast for the entire route and display it to the user as a forecast table.



Once the initial dataflow has occurred and the Blazor app has all the data, it is easy for the app to continuously process it in different ways to calculate other results. For example, the user can choose to offset their start time if they were going to leave 90 minutes from now instead, and the application would just process the data again with a different offset, rather than go and retrieve any more data from the server. This makes the application very efficient because it makes very few calls to the server.

#### B. Leaflet

One additional feature of the application is that the actual map can also be viewed along with corresponding weather icons and details along the route. To do this it was

necessary to step outside of Blazor WASM and C# and add JavaScript code to the project unfortunately.

Leaflet is JavaScript library that can build interactive maps which can be customized with paths and markers. To accomplish this I leveraged Blazor's JavaScript interoperability which allows Blazor to call functions and send data directly to JavaScript. To accomplish this, the route data including the all the waypoints along the path and their associated weather must be combined into JSON format and passed to a JavaScript function stored separately which is responsible for then parsing the route and displaying the data on the JavaScript map.

While this map did require departing from the Microsoft Azure/.NET ecosystem, it does highlight the power of Blazor WASM still being able to integrate well with JavaScript when necessary.

#### IV. IMPLEMENTATION

While the previous sections have touched on the overall architecture and data flow of the system, I would like to provide some more specific details of some of the actual implementation and the code.

##### A. Azure Functions

The core of an Azure Function involves defining a function's name and the routing to reach it. For this application, there is just one Azure Function, `GetRouteWeather`, which is activated by requesting the route `"weather/{start}/{end}"` where the function then takes whatever is in `{start}` and `{end}` as the locations to retrieve a route for.

To organize the Azure Function there are two service classes, `RouteService` and `WeatherService`. The start location and end location strings are input into the `RouteService`'s `ProcessRoute` function which handles the processing of the waypoints of the route and retrieval of the route data which is handled by another service class called the `AzureMapsClient`. Once the `RouteService` finishes processing the route, it returns a `Route` model containing all necessary information about the route.

Next, the sampled 20 waypoints of the processed `Route` object is input into the `WeatherService`'s `GetForecasts` function which retrieves a forecast for each of the 20 sampled locations of the route.

Once all the processing is complete, all this data is combined into a single `JsonObject` which is then converted into a JSON string and returned as a response. This response is what will be read by the Blazor App.

```
[FunctionName("GetRouteWeather")]
public async Task<ActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = "weather/{start}/{end}"]
    string start,
    string end,
    ILogger log)
{
    log.LogInformation("Received request for route from {Start} to {End}", start, end);
    try
    {
        var route = await routeService.ProcessRoute(start, end);
        var forecasts = await weatherService.GetForecasts(route.SampledWaypoints);

        var result = new JsonObject
        {
            ["route"] = route.RouteData,
            ["locations"] = route.SampledLocations,
            ["weather"] = forecasts
        };

        return new OkObjectResult(result.ToString());
    }
    catch (Exception ex)
    {
        log.LogError(ex, "Failed to process route and weather");
        return new StatusCodeResult(StatusCode.InternalServerError);
    }
}
```

##### B. Blazor WASM

The Blazor WASM application is a single page application so it only involves one razor page which is essentially a combination of C# and HTML. This `Home.razor` page is responsible for defining the layout of the webpage and the initial data retrieval.

While the full details of the Blazor WASM would be well beyond the scope of this paper, I will speak to the core functions of this component of the application. The main processing is triggered through the "Get Weather" button which calls the `GetRouteWeather` function.

```
private async Task GetRouteAndWeather()
{
    isLoading = true;
    try
    {
        route = await WeatherForecastService.GetWeatherForecastAsync(fromAddress, toAddress);
        forecasts = WeatherForecastService.GetWeatherForecastByTimeAndInterval(route, 0, 15);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    isLoading = false;
}
```

A service class, `WeatherForecastService`, was created to organize the various functionality required for this application and is utilized in this function. First `GetWeatherForecastAsync` is called which is what makes an HTTP request to the Azure Function with the start and end location and retrieves the response. This response is then immediately parsed through a large series of other functions which convert the large set of disorganized JSON data from the Azure Function into a single `Route` model which contains the 20 legs of the route as a list of

Leg models and each leg contains the full 24 hours of weather forecasts in the form of a WeatherForecast model.

Once this data has been parsed and converted into a single Route model, the GetRouteAndWeather function then calls the GetWeatherForecastByTimeAndInterval function which is responsible for retrieving the estimated location along the route at each given interval, 15 minutes in this case, and then looking up the corresponding weather that will be happening at that location at the time and returning a list of weather information where each forecast is 15 minutes apart and 15 minutes further down the route.

Once the forecasts for the routes have been calculated, the ForecastTable component is responsible for iterating through the forecasts and creating an HTML table containing the time, location and weather information for each 15 minute period of the route. This is what is then displayed to the user.

While there are a few other features involved with the Blazor WASM side of the application, they generally follow this same pattern of taking the route data and processing it by matching locations along the route with the weather at the corresponding time they will be reached and displaying it to the screen.

V. RESULTS

From the beginning, the goal of this project was to create three primary features for the application. A weather table, a departure based weather grid display, and a map.

A. Forecast Table

The forecast table was the most basic and core feature to include. This mimics the typical look of an hourly forecast you might expect on a traditional website.

























This table clearly shows the location, time and weather of your route and makes it easy to see what you should expect ahead.

Time	Location	Weather Code	Temperature (°F)	Wind Speed (mph)	Feels Like (°F)
7:33 PM	Fargo		44.1° F	2.8 mph	40.6° F
7:48 PM	Glyndon		43.5° F	3.0 mph	39.9° F
8:03 PM	Barnesville		43.0° F	4.2 mph	38.8° F
8:18 PM	Rothsay		42.2° F	5.1 mph	37.5° F
8:33 PM	Fergus Falls		42.4° F	4.8 mph	38.0° F
8:48 PM	Ashby		42.2° F	4.5 mph	38.0° F

B. Weather Grid

The weather grid feature is slightly more complicated in that it displays not just the weather for your route if you were to leave right now, but also the weather for your route at other potential departure times, each 30 minutes apart.

The goal of this feature is to be able to allow users to optimize their departure time to avoid as much better weather as possible. For example, below is a grid showing the forecasts for the route from Fayette, KY to Edinburgh, IN. You can see if you leave at 8:30 PM you will run into heavy rain near the end of your trip. But if you instead leave at 10:30 PM, you will only run into a small amount of rain near the end. While rain may not potentially stop someone from traveling, this feature could be especially useful when the weather is more extreme such as lightning, heavy snow, or torandoes.

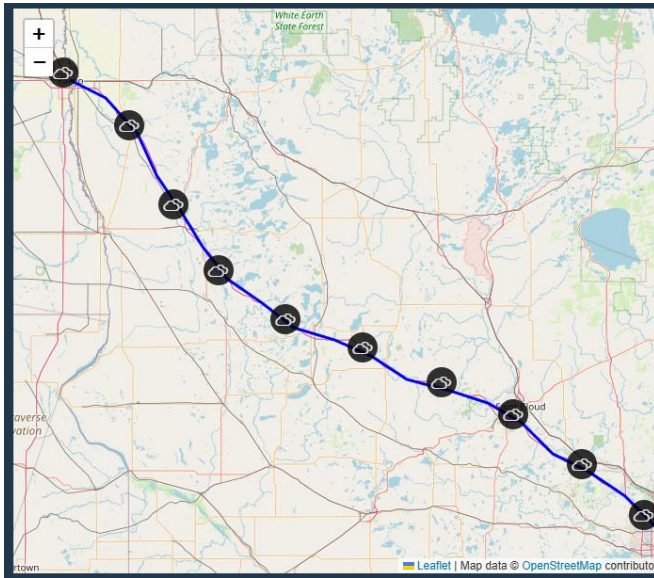
Departure \ Location	0 hr Fayette, KY	30 min Waddy	1 hr Jefferso...	1 hr 30 min Crother...	2 hr Edinburgh
8:30 PM Departure					
9:00 PM Departure					
9:30 PM Departure					
10:00 PM Departure					
10:30 PM Departure					



### C. Weather Map

The map is intended to give a clear visual of the path that it is assuming for all these calculations. It not only provides the path drawn out as a blue line, but weather icons for the weather along the route. As with all other parts of this app, these weather icons are not the current weather, but weather that will be there at the time you reach them along the route.

Additionally, the map has the functionality of being able to scroll over locations along the path and see more detailed weather information and the estimated time of arrival.



## VI. CONCLUSIONS

### A. Lessons Learned

I found this a really educational experience. While class projects and assignments have required us to often make many programs. It is generally within the context of a development environment where it is only deployed or executed locally. I felt that this project gave me the opportunity to learn about actually deploying a full-stack website to a server using the same sort of architecture that a professional developer might use.

Getting the hands-on experience of setting up an Azure cloud and account and configuring my static web app and Azure Function to be able to be deployed was very enlightening to me. Additionally, setting up GitHub actions to be able to automate the workflow of actually deploying my app was another useful learning experience.

Beyond the coding, I think another valuable experience out of all this was participating in the Student Academic Conference. I had never presented at something like this before so to be able to put together a presentation for

others and have them watch and give feedback and questions was a really great experience.

Overall, I think there was just a great deal of hands-on experience both with deploying web apps/servers and giving presentations to others which I think will be very useful going forward in my academic/professional career.

### B. Future Changes

During my presentation of this project, I received some useful feedback some of the audience which would be great future changes:

- Add the option for bikes instead of just cars.
  - It was noted that bikers would be especially interested in weather like rain along their route and they could be potential users of this app.
  - This would adding an option to the Blazor app for selecting car or bike, and then this would be used by the Azure Function when making a call to Azure Maps to get the routes. It would just ask for the route type of bike instead so this would be a very easy change to implement
- Add the option for stops along the route.
  - It was noted that on long car trips it is very common to have to make stops along the way for things such as lunch, and it would useful to be able to take those into account in the route forecast.
  - This change would not require any modification to the retrieval of data from Azure Functions, and would only require a change in the way the client processes data. The user would need the ability to choose a time/location for the stop, and then when calculating forecasts after that stop, an offset corresponding to the length of that time would have to be added to each forecast. That is, if you stop for 30 minutes halfway through the trip then all weather forecasts retrieved after that stop will need to be offset by 30 minutes.

I think both these suggestions were excellent and could be easily implemented within the current framework of the application.

Some additional potential changes that I have thought about:

- Using a different weather API. I used Open-Meteo because it is free, but some of their forecasts lack specific details such as rain probability. A different

API might allow for more useful weather data to be added.

- Automated testing. There is a large amount of complex processing of data going on in the app and it would be difficult to make any changes without there being potential side effects. An automated testing system which assures all the weather is processed as expected to make the application much more robust and easy to modify. This system could also be integrated with GitHub actions so that it will only deploy new versions of the website if it passes the tests.
- Greater precision in waypoints. Twenty sampled waypoints is simply too little for long routes that could potentially stretch 1,000 miles. For long routes, it would be useful to sample more routes to create more precise weather forecasts along the route rather than forecasts only every 50 miles in the case of a 1,000 mile trip.
- A mobile app version. This app would be most naturally used on a mobile phone so it would be ideal if there were a mobile app version which could be integrated with the GPS of the user's current location and update as they progress along their trip. The use of the .NET ecosystem of Blazor WASM makes this a much easier leap to make because much of the same C# code I use for this project could also be used to create a C#-based mobile app using the MAUI framework

By utilizing the feedback both of the audience and some of my own plans, I believe there could be a lot of great improvements to this application which could open

it up to a wider audience. There is certainly a niche element to this app, and so if I were to actually deploy it, it would be very important to try to identify what specific type of user would be most likely to be interested in this and optimize it for them. I think the long distance biker was an especially good suggestion and I think would be a great potential user to look into improving it for.

#### REFERENCES

- [1] "Route - rest API (azure maps)," REST API (Azure Maps) | Microsoft Learn, <https://learn.microsoft.com/en-us/rest/api/maps/route>.
- [2] "Docs | Open-Meteo.com," Open Meteo, <https://open-meteo.com/en/docs/>
- [3] "Mapping weather severity zones," Clear Roads, <https://www.clearroads.org/project/10-02/>
- [4] "How do weather events impact roads?," How Do Weather Events Impact Roads? - FHWA Road Weather Management, [https://ops.fhwa.dot.gov/weather/q1\\_roadimpact.htm](https://ops.fhwa.dot.gov/weather/q1_roadimpact.htm)
- [5] "ASP.NET Core Blazor," Microsoft Learn, <https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-8.0>.
- [6] "Azure functions overview," Azure Functions Overview | Microsoft Learn, <https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview?pivots=programming-language-csharp>.
- [7] "Deploying to azure static web app," GitHub Docs, <https://docs.github.com/en/actions/deployment/deploying-to-your-cloud-provider/deploying-to-azure/deploying-to-azure-static-web-app>.
- [8] E. Flowers, "Erikflowers/weather-icons: 215 weather themed icons and CSS," Weather Icons, <https://github.com/erikflowers/weather-icons>.
- [9] "Documentation - leaflet - a JavaScript library for interactive maps," Leaflet, <https://leafletjs.com/reference.html>.