

## **Diving into .NET 9, Blazor, and Observability with Coralogix**

So, there I was, a newbie to .NET 9, Blazor, and Coralogix, standing on the precipice of observability in a world of bugs and mysteries. As an Agile enthusiast, I'm well versed in all things "observability" and how it's a game-changer for root cause analysis, especially in today's rapid, iterative development cycles. Here's my journey creating a Blazor app with OpenTelemetry (OTEL) and Coralogix — with some unexpected fun along the way.

### **First, Baby Steps: Blazor and .NET 9.0 (Literally!!)**

Starting with .NET 9.0 and Blazor was a lot like making lemonade...if you'd never seen a lemon before - I haven't used .Net since .Net 5.0, so learning curve central – oh and I have a MacBook too. With Microsoft's help, though, I got my "DemoBlazor" app up and running pretty quickly. The default project came with a basic WeatherForecastService — a nifty mock weather service that could fetch forecasts. Nothing fancy, but enough to get the ball rolling and for this it meant I didn't need to think about creating my own.

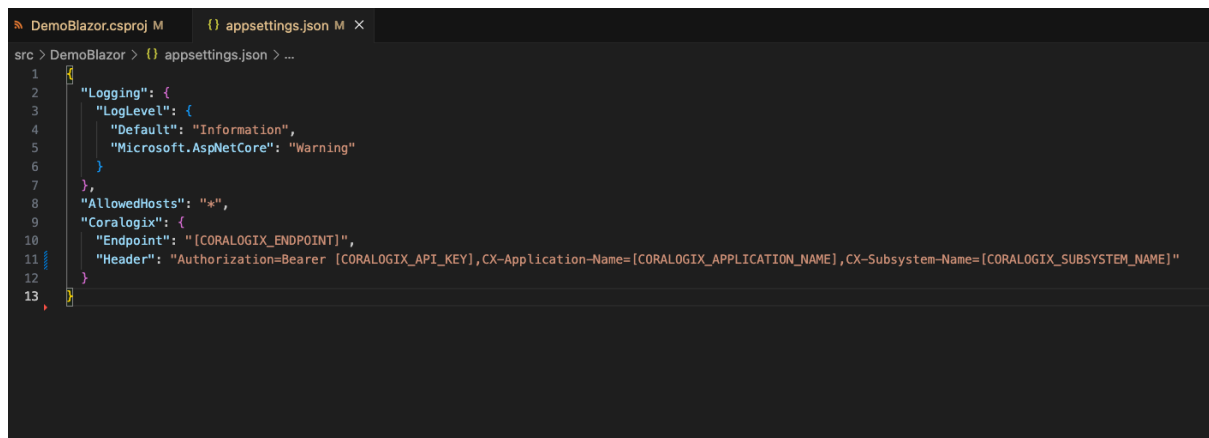
```
src > DemoBlazor > Data > WeatherForecastService.cs > WeatherForecastService
1 namespace DemoBlazor.Data;
2
3 7 references
4 public class WeatherForecastService
5 {
6     2 references
7     private static readonly string[] Summaries = new[]
8     {
9         "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
10    };
11
12    3 references
13    public Task<WeatherForecast[]> GetForecastAsync(DateOnly startDate)
14    {
15        return Task.FromResult(Enumerable.Range(1, 5).Select(index => new WeatherForecast
16        {
17            Date = startDate.AddDays(index),
18            TemperatureC = Random.Shared.Next(-20, 55),
19            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
20        }).ToArray());
21    }
22 }
```

From here, I wanted to enhance this basic setup with some real-world monitoring capabilities. This is where **observability** comes in: rather than relying on gut feeling (or the infamous "it works on my machine" excuse – this causes me pain), observability lets us know *exactly* what's happening under the hood when things go wrong. And in Agile, where bugs are lurking in every sprint, observability is invaluable.

### **Getting Cozy with OTEL and Coralogix**

Now, the goal was to pump some telemetry (traces, metrics, and logs) from my Blazor app to Coralogix using OTEL. With OpenTelemetry being a kind of universal

adapter for observability, the integration was actually pretty smooth. I just needed to install a few packages and configure endpoints and API keys in appsettings.json.



```
src > DemoBlazor > {} appsettings.json > ...
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft.AspNetCore": "Warning"
6      }
7    },
8    "AllowedHosts": "*",
9    "Coralogix": {
10     "Endpoint": "[CORALOGIX_ENDPOINT]",
11     "Header": "Authorization=Bearer [CORALOGIX_API_KEY], CX-Application-Name=[CORALOGIX_APPLICATION_NAME], CX-Subsystem-Name=[CORALOGIX_SUBSYSTEM_NAME]"
12   }
13 }
```

This setup would give me a full view of my app's health in Coralogix — but first, I had to make sure I was collecting data worth viewing!

## Adding Some Flavour: Metrics, Tracing, and Logging

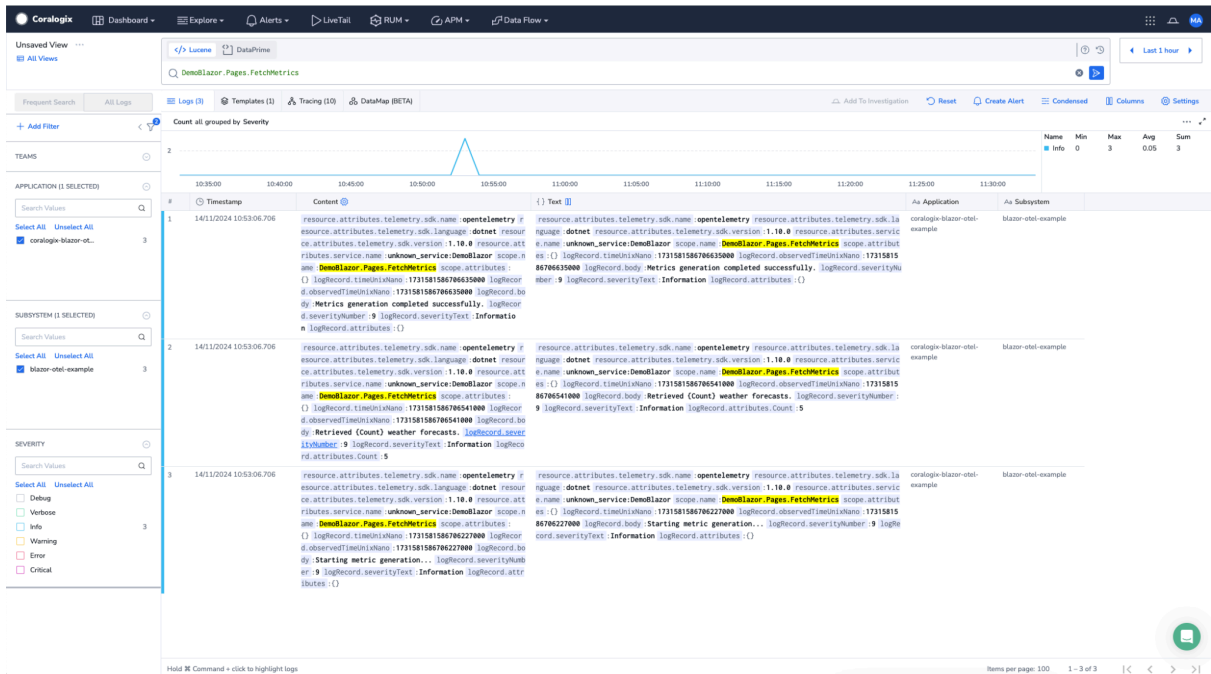
To make the observability data interesting, I created three pages: **FetchMetrics**, **FetchTracing**, and **FetchLogging**.

The first page, FetchMetrics, generated CPU and memory metrics. Nothing fancy, just some simulated loops and mock data to showcase what metrics could look like in Coralogix.

```

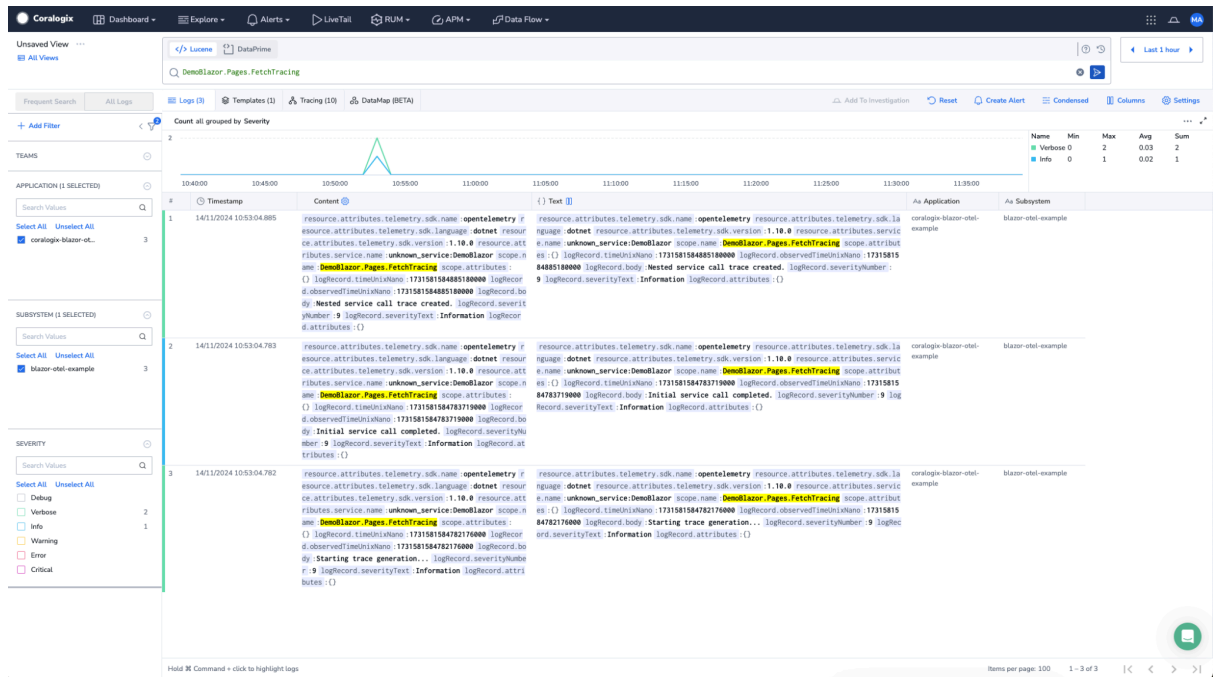
src > DemoBlazor > Pages > FetchMetrics.razor
1  @page "/fetch-metrics"
2  @using DemoBlazor.Data
3  @inject ILogger<FetchMetrics> Logger
4  @inject WeatherForecastService ForecastService
5
6  <h3>Fetch Metrics</h3>
7
8  <button class="btn btn-primary" onclick="GenerateMetrics">Generate Metrics</button>
9
10 <p>@statusMessage</p>
11
12 @code {
13
14     private string statusMessage = "Click the button to generate metrics.";
15
16     private async Task GenerateMetrics()
17     {
18         try
19         {
20             Logger.LogInformation("Starting metric generation...");
21
22             // Simulate a memory allocation metric
23             var forecasts = await ForecastService.GetForecastAsync(DateOnly.FromDateTime(DateTime.Now));
24             Logger.LogInformation("Retrieved {Count} weather forecasts.", forecasts.Length);
25
26             // Simulate a CPU intensive task for metrics demonstration
27             for (int i = 0; i < 10000; i++)
28             {
29                 Math.Sqrt(i);
30             }
31
32             Logger.LogInformation("Metrics generation completed successfully.");
33             statusMessage = "Metrics generated and sent to Coralogix.";
34         }
35         catch (Exception ex)
36         {
37             Logger.LogError(ex, "An error occurred while generating metrics.");
38             statusMessage = "An error occurred during metric generation.";
39         }
40     }
41

```



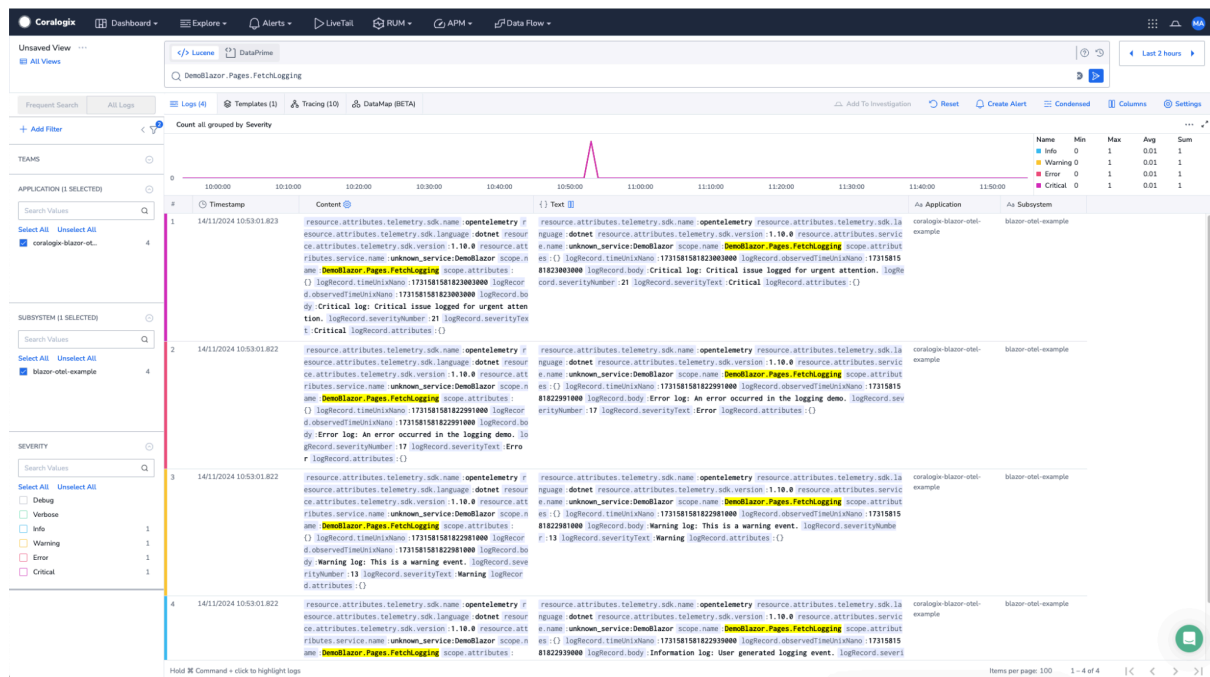
FetchTracing was where things got interesting. I set up a nested call to simulate real-world tracing — the kind you'd encounter when debugging multi-step processes. With OTEL, I could follow each call step-by-step and see where things were slowing down or breaking.

```
src > DemoBlazor > Pages > FetchTracing.razor
1  @page "/fetch-tracing"
2  @using DemoBlazor.Data
3  @inject ILogger<FetchTracing> Logger
4  @inject WeatherForecastService ForecastService
5
6  <h3>Fetch Tracing</h3>
7
8  <button class="btn btn-primary" @onclick="GenerateTrace">Generate Trace</button>
9
10 <p>@statusMessage</p>
11
12 @code {
13     4 references
14     private string statusMessage = "Click the button to generate a trace.";
15
16     2 references
17     private async Task GenerateTrace()
18     {
19         try
20         {
21             Logger.LogInformation("Starting trace generation...");
22
23             // Initial service call trace
24             var forecasts = await ForecastService.GetForecastAsync(DateOnly.FromDateTime(DateTime.Now));
25             Logger.LogInformation("Initial service call completed.");
26
27             // Nested service call to simulate complex tracing
28             await Task.Delay(100); // Simulate async work
29             Logger.LogInformation("Nested service call trace created.");
30
31             statusMessage = "Trace generated and sent to Coralogix.";
32         }
33         catch (Exception ex)
34         {
35             Logger.LogError(ex, "An error occurred while generating tracing data.");
36             statusMessage = "An error occurred during trace generation.";
37         }
38     }
39 }
```



FetchLogging, on the other hand, lets me experiment with logs of all levels: Info, Debug, Warning, and my personal favourite — Critical! With all three pages in place, my humble weather app was now generating enough observability data to make Sherlock Holmes proud.

```
src > DemoBlazor > Pages > FetchLogging.razor
1  @page "/fetch-logging"
2  @using DemoBlazor.Data
3  @inject ILogger<FetchLogging> Logger
4
5  <h3>Fetch Logging</h3>
6
7  <button class="btn btn-primary" @onclick="GenerateLogs">Generate Logs</button>
8
9  <p>@statusMessage</p>
10
11  @code {
12      3 references
13      private string statusMessage = "Click the button to generate logs.";
14
15      2 references
16      private void GenerateLogs()
17      {
18          Logger.LogTrace("Trace log: Verbose tracing log for Coralogix.");
19          Logger.LogDebug("Debug log: Debugging information for development.");
20          Logger.LogInformation("Information log: User generated logging event.");
21          Logger.LogWarning("Warning log: This is a warning event.");
22          Logger.LogError("Error log: An error occurred in the logging demo.");
23          Logger.LogCritical("Critical log: Critical issue logged for urgent attention.");
24
25          statusMessage = "Logs generated and sent to Coralogix.";
26      }
27  }
```



## Observability's Secret Weapon: Root Cause Analysis

Here's the beauty of observability: when something goes wrong, you don't have to rely on guesswork. All that OTEL data flows to Coralogix, where I can trace issues back to their root causes with actual data, not assumptions. In an Agile setting, this kind of insight can save days of back-and-forth troubleshooting. Having logs, traces, and metrics at my fingertips means I can spot the exact point of failure, down to the millisecond.

## Taking It Further

For those already running a Blazor app, adding observability could be your next power move. Just bring in OTEL for traces and metrics, set up Coralogix, and add custom metrics for key user actions. Want even more control? Try adding alerting in Coralogix to automatically notify you of anomalies. Trust me, once you get a taste of observability, there's no going back. Who knew debugging could be this fun?

In the end, I may be new to .NET, but with the power of Blazor, OTEL, and Coralogix on my side, I feel like I could tackle any bug that comes my way. And hey, if I can do it, anyone can.

