.NET OpenTelemetry Cheatsheet

01 - Packages to install

In order to get your applications setup with the OpenTelemetry protocol exporter for an ASP.NET Core application, you'll need the following packages:

```
OpenTelemetry.Exporter.OpenTelemetryProtocol
OpenTelemetry.Exporter.OpenTelemetryProtocol.Logs
OpenTelemetry.Extensions.Hosting
OpenTelemetry.Instrumentation.AspNetCore
OpenTelemetry.Instrumentation.Http
OpenTelemetry.Instrumentation.Runtime
```

Note: Some packages maybe in a pre-release state if you cannot see them on NuGet.org straightaway

02 - Insecure Channel Setup

If your application is targeting .NET Core 3.1, and you are using an insecure (HTTP) endpoint, the following switch must be set before adding OtlpExporter

```
AppContext.SetSwitch("System.Net.Http.SocketsHttpHandler
.Http2UnencryptedSupport", true);
```

03 - Resource setup

We will need to setup a new resource and name our service appropriately:

```
var resource = ResourceBuilder.CreateDefault()
    .AddService("CodeWithStu", serviceVersion: "1.0.0");
```

04 - Logging Setup

Remember to change the endpoint below to come from something like appsettings.json:

```
builder.Logging.AddOpenTelemetry(options =>
{
    options.SetResourceBuilder(resource);
    options.AddOtlpExporter(otlp =>
    {
        otlp.Endpoint = new Uri("https://myendpoint");
    });
});
```

05 - Metrics Setup

Remember to change the endpoint below to come from something like appsettings.json:

```
builder.Services.AddOpenTelemetryMetrics(options =>
{
    options.SetResourceBuilder(resource);
    options.AddOtlpExporter(otlp =>
    {
        otlp.Endpoint = new Uri("https://myendpoint");
    });

    options.AddHttpClientInstrumentation();
    options.AddAspNetCoreInstrumentation();
    options.AddRuntimeInstrumentation();
});
```

.NET OpenTelemetry Cheatsheet

06 - Tracing Setup

Remember to change the endpoint below to come from something like appsettings.json:

```
builder.Services.AddOpenTelemetryTracing(options =>
{
    options.SetResourceBuilder(resource);
    options.AddOtlpExporter(otlp =>
    {
        otlp.Endpoint = new Uri("https://myendpoint");
    });

    options.AddAspNetCoreInstrumentation();
    options.AddHttpClientInstrumentation();
});
```

07 - Adding Custom Metrics

Meters are the entry point for libraries to create a named group of instruments, such as counters and histograms. Each library can and should create it's own meter, storing it either in a static variable or DI container. Once created, call CreateXXX and store the result in the same way.

```
using System.Diagnostics.Metrics;

var myMeter = new Meter("CodeWithStu", "1.0.0");

builder.Services.AddOpenTelemetryMetrics(options =>
{
    // ...
    options.AddMeter(myMeter.Name);
});
```

08 - Other Exporters

The majority of this code can be re-used with other exporters for logs / traces / metrics. Simply replace **AddOtlpExporter(...)** with the equivalent from one of the other exporters listed list below:

- Console
- Geneva [*contrib*]
- Jaeger
- Instana [*contrib*]
- Prometheus
- StackDriver [*contrib*]
- Zipkin
- ZPages

Note: *requires additional NuGet package installation*

Note 2: *"contrib" means projects that don't meet the express scope of the core OpenTelemetry for .NET SDK's but are otherwise contributed to the project and actively maintained.*

09 - More Information

For a more detailed look at how to configure OpenTelemetry in a .NET application, watch this playlist:

https://bit.ly/CodeWithStu-OTel

Last updated & Checked: 2022-07-15