Microsoft

Logging in NET Core

Orestis Meikopoulos
Microsoft Azure Cloud & AI Consultant

https://www.linkedin.com/in/ormikopo

Agenda

- Logging in .NET Core
- Application Insights

Logging Providers

- .NET Core supports a logging API out-of-the-box
- **Logging providers** store logs
  - Except for the Console provider which displays logs
  - e.g. the Azure Application Insights provider stores logs in Azure Application Insights
  - Multiple providers can be enabled
- The default ASP.NET Core web app templates
  - Call **CreateDefaultBuilder**, which adds the following default logging providers
    - **Console**
    - **Debug**
    - **EventSource**
    - **EventLog**: Windows only

Logging output from dotnet run and Visual Studio

- Logs created with the default logging providers are displayed
  - In the **Debug output window** of **Visual Studio** when debugging
  - In the **Console window** when the app is run with **dotnet run**
- Logs that begin with "**Microsoft**" categories are from ASP.NET Core framework code
- ASP.NET Core and application code use the same logging API and providers

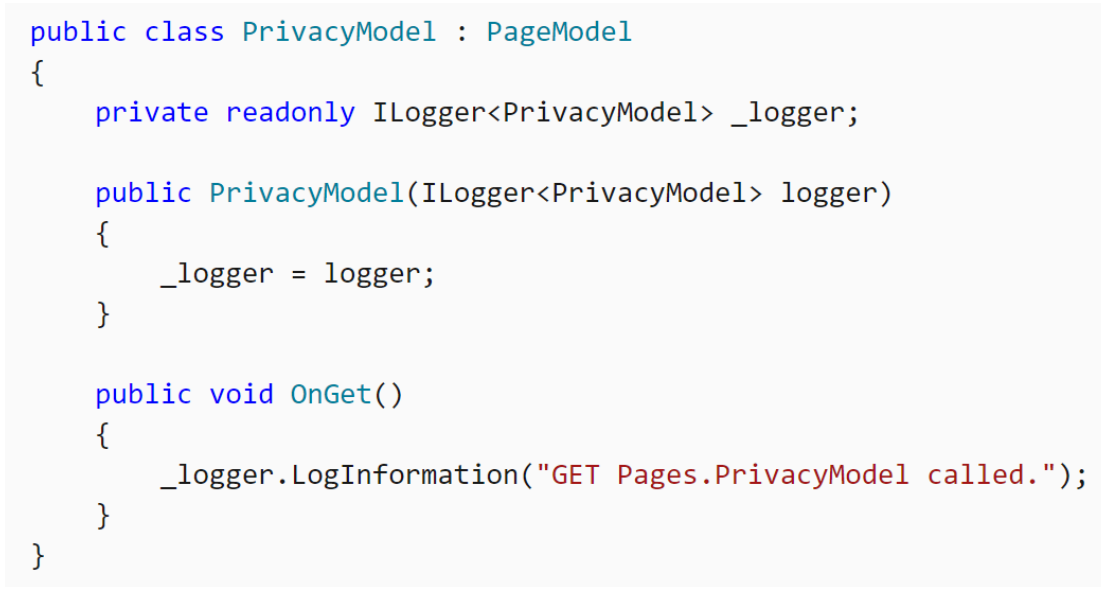
Create Logs

- To create logs, use an ILogger<TCategoryName> object from dependency injection (DI)
- The following example
  - Creates a logger, **ILogger<AboutModel>**
  - Calls **LogInformation** to log at the Information level

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }
    public string Message { get; set; }

    public void OnGet()
    {
        Message = $"About page visited at {DateTime.UtcNow.ToLongTimeString()}";
        _logger.LogInformation(Message);
    }
}
```

Log Category

- When an **ILogger** object is created, a **log category** is specified
  - That category is included with each log message created by that instance of ILogger
  - The category string is arbitrary, but the convention is to use the class name, which will produce a log category in the form of **{AssemblyName}{Namespace}{ClassName}**
    - e.g. "TodoApi.Controllers.TodoController
  - The use of the log category allows us to categorize our different logging messages

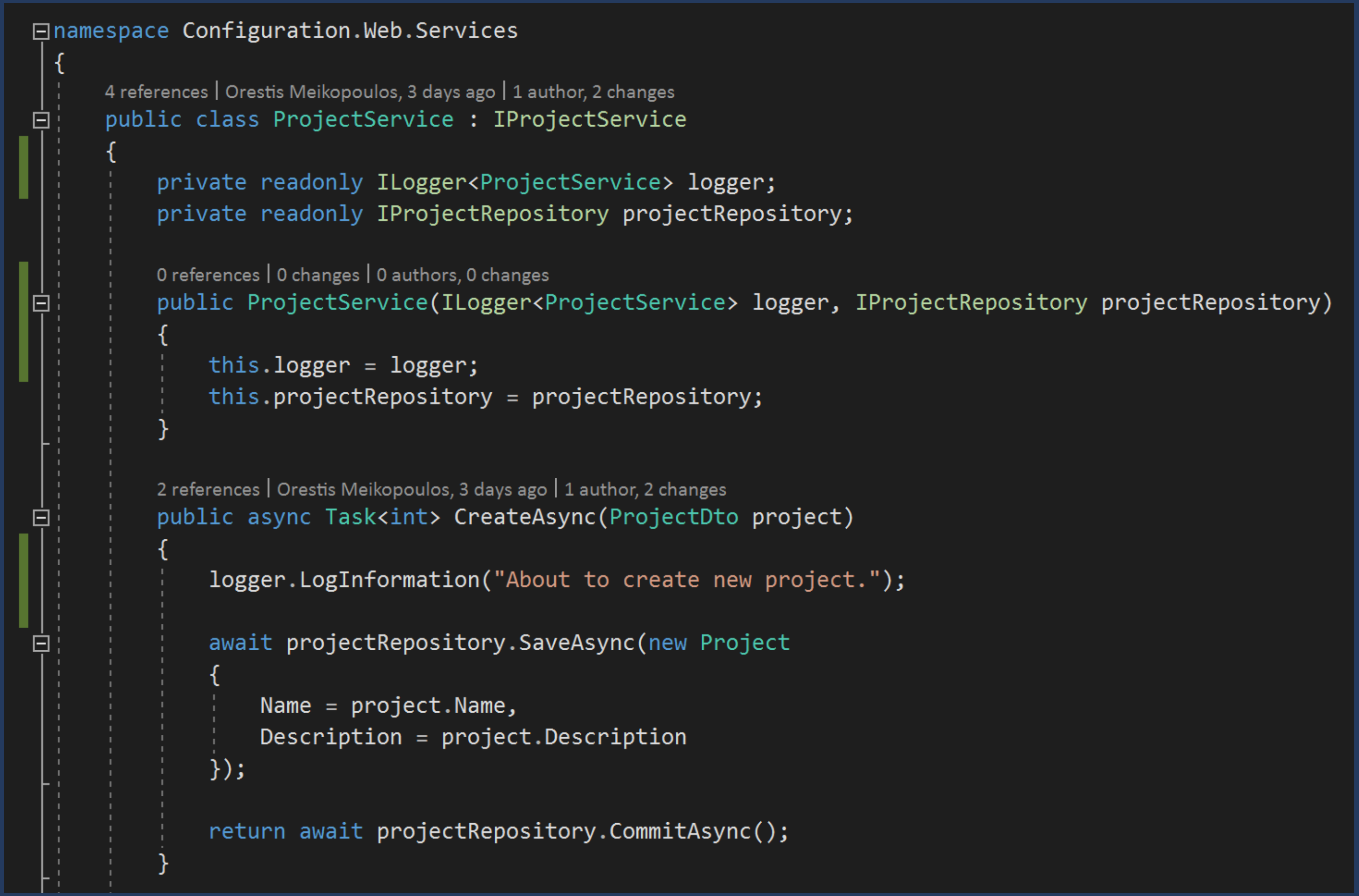


```
public class PrivacyModel : PageModel
{
    private readonly ILogger<PrivacyModel> _logger;

    public PrivacyModel(ILogger<PrivacyModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        _logger.LogInformation("GET Pages.PrivacyModel called.");
    }
}
```

Log Level

LogLevel	Value	Method	Description
Trace	0	LogTrace	Contain the most detailed messages. These messages may contain sensitive app data. These messages are disabled by default and should ***not*** be enabled in production.
Debug	1	LogDebug	For debugging and development. Use with caution in production due to the high volume.
Information	2	LogInformation	Tracks the general flow of the app. May have long-term value.
Warning	3	LogWarning	For abnormal or unexpected events. Typically includes errors or conditions that don't cause the app to fail.
Error	4	LogError	For errors and exceptions that cannot be handled. These messages indicate a failure in the current operation or request, not an app-wide failure.
Critical	5	LogCritical	For failures that require immediate attention. Examples: data loss scenarios, out of disk space.
None	6		Specifies that a logging category should not write any messages.

Logging Basics



```
namespace Configuration.Web.Services
{
    4 references | Orestis Meikopoulos, 3 days ago | 1 author, 2 changes
    public class ProjectService : IProjectService
    {
        private readonly ILogger<ProjectService> logger;
        private readonly IProjectRepository projectRepository;

        0 references | 0 changes | 0 authors, 0 changes
        public ProjectService(ILogger<ProjectService> logger, IProjectRepository projectRepository)
        {
            this.logger = logger;
            this.projectRepository = projectRepository;
        }

        2 references | Orestis Meikopoulos, 3 days ago | 1 author, 2 changes
        public async Task<int> CreateAsync(ProjectDto project)
        {
            logger.LogInformation("About to create new project.");

            await projectRepository.SaveAsync(new Project
            {
                Name = project.Name,
                Description = project.Description
            });

            return await projectRepository.CommitAsync();
        }
```

Audience Question 1

What will be the log category value written in messages coming from the ILogger<ProjectService> instance based on the above image? Which the log level of our choice here?

Configure Logging

- By using **Logging** section of **appsettings.{Environment}.json** files
  - The "**Default**", "**Microsoft**", and "**Microsoft.Hosting.Lifetime**" categories are specified
  - The "**Microsoft**" category applies to all categories that start with "**Microsoft**” and logs at log level **Warning** and higher
  - The "**Microsoft.Hosting.Lifetime**" category is more specific than the "**Microsoft**" category and logs at log level "**Information**" and higher
  - Specific log provider is not specified, so **LogLevel** applies as default to all the enabled logging providers

```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

Configure Logging

- The **“Logging”** property can have **“LogLevel”** and **log provider** (“Debug, “Console” etc.) properties
- The **LogLevel** specifies the **minimum level to log for selected categories**
  - When a LogLevel is specified, logging is enabled for messages at the specified level and higher
- A **log provider** property can also specify a LogLevel property
  - LogLevel under a provider specifies levels to log for that provider, and overrides the non-provider log settings

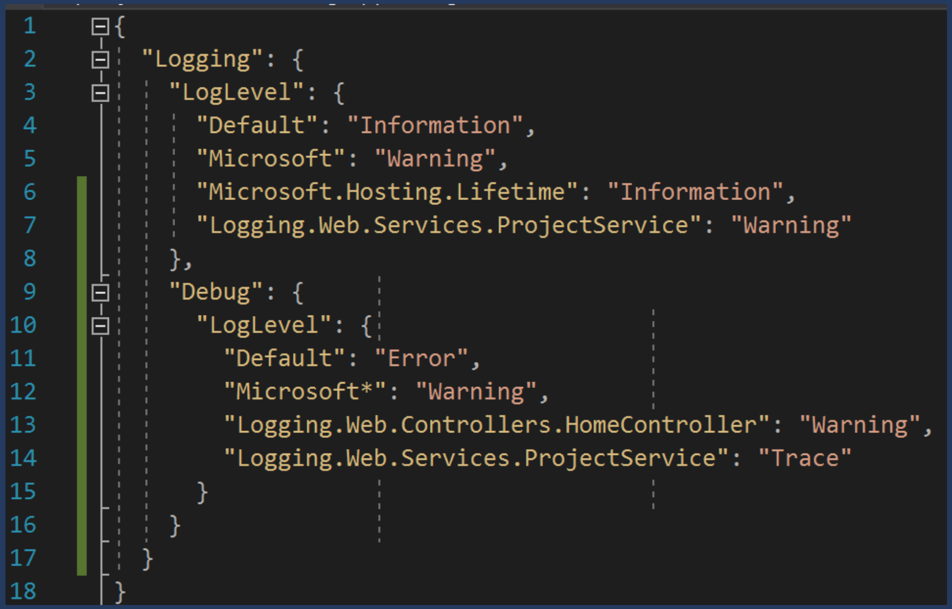
```
{
  "Logging": {
    "LogLevel": { // All providers, LogLevel applies to all the enabled providers.
      "Default": "Error", // Default logging, Error and higher.
      "Microsoft": "Warning" // All Microsoft* categories, Warning and higher.
    },
    "Debug": { // Debug provider.
      "LogLevel": {
        "Default": "Information", // Overrides preceding LogLevel:Default setting.
        "Microsoft.Hosting": "Trace" // Debug:Microsoft.Hosting category.
      }
    },
    "EventSource": { // EventSource provider
      "LogLevel": {
        "Default": "Warning" // All categories of EventSource provider.
      }
    }
  }
}
```

Logging Configuration

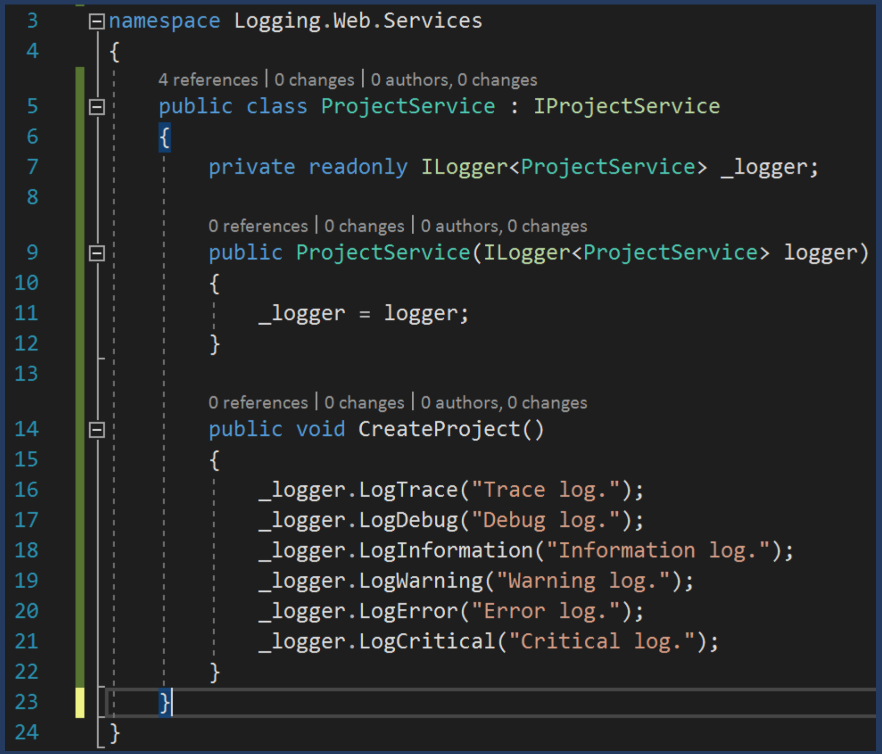
Audience Question 2

We have created a project named Logging.Web and suppose inside the Services folder we create a ProjectService.cs class. Inside it we are injecting an ILogger<ProjectService> instance inside a class property called _logger. In CreateProject() method we are calling logger.LogInformation("Information log."). Based on the appSettings.json file of this slide:

- Will this log message be logged in Debug VS window if we debugged the app through VS?
- Will it be logged in Console if we ran the app with dotnet run?



```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Logging.Web.Services.ProjectService": "Warning"
    },
    "Debug": {
      "LogLevel": {
        "Default": "Error",
        "Microsoft*": "Warning",
        "Logging.Web.Controllers.HomeController": "Warning",
        "Logging.Web.Services.ProjectService": "Trace"
      }
    }
  }
}
```

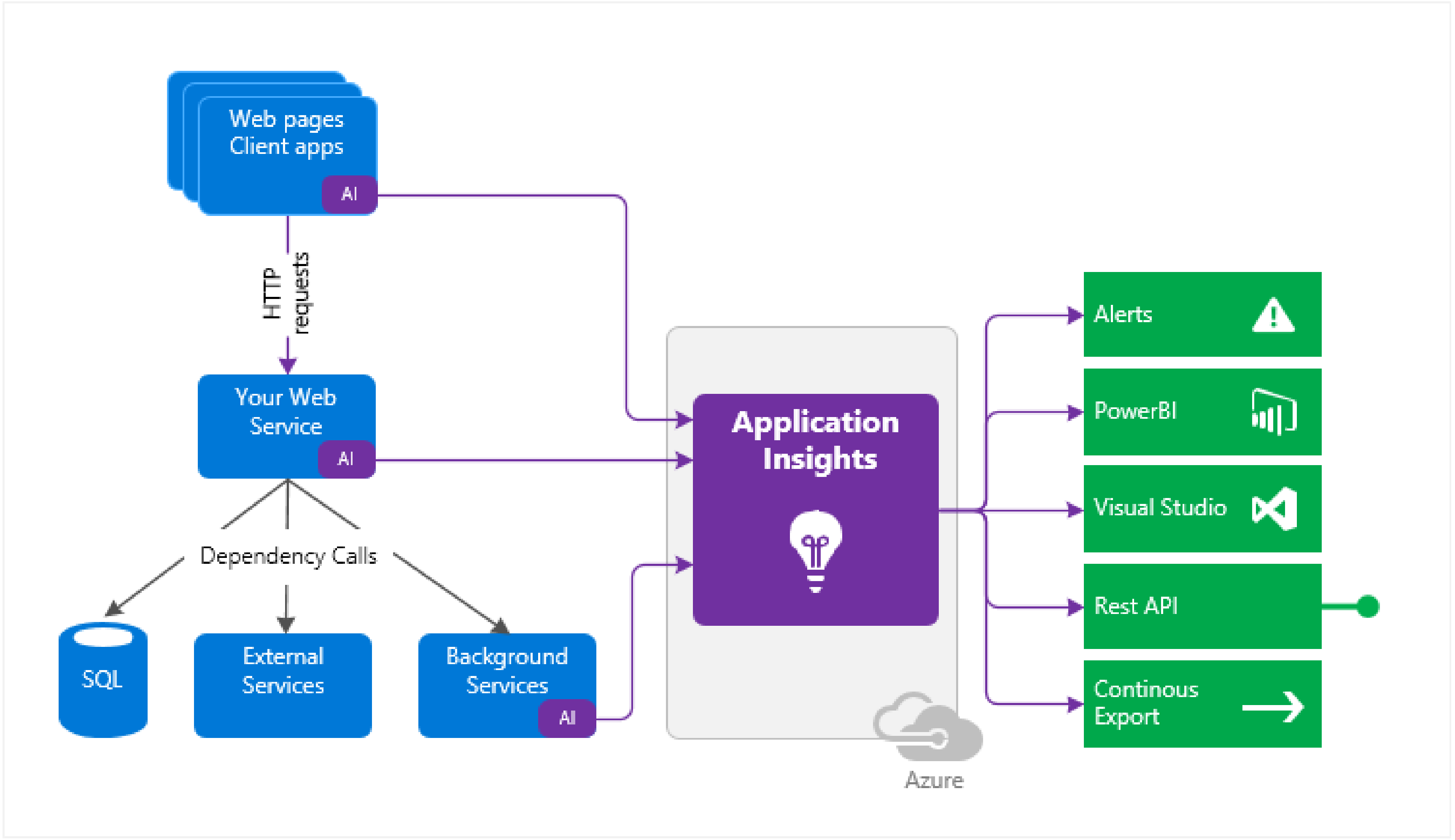


```csharp
namespace Logging.Web.Services
{
    4 references | 0 changes | 0 authors, 0 changes
    public class ProjectService : IProjectService
    {
        private readonly ILogger<ProjectService> _logger;

        0 references | 0 changes | 0 authors, 0 changes
        public ProjectService(ILogger<ProjectService> logger)
        {
            _logger = logger;
        }

        0 references | 0 changes | 0 authors, 0 changes
        public void CreateProject()
        {
            _logger.LogTrace("Trace log.");
            _logger.LogDebug("Debug log.");
            _logger.LogInformation("Information log.");
            _logger.LogWarning("Warning log.");
            _logger.LogError("Error log.");
            _logger.LogCritical("Critical log.");
        }
    }
}
```

Application Insights - Overview

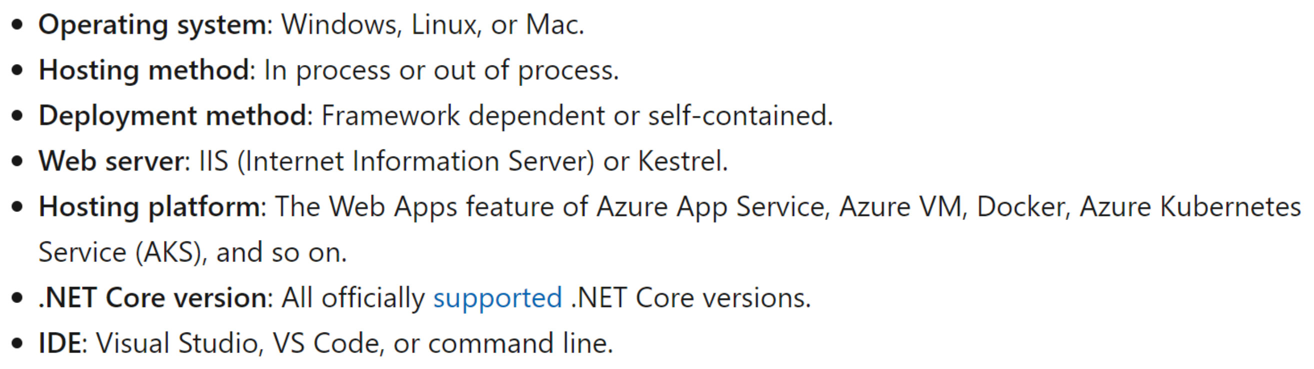
- Application Insights is a **platform as a service (PAAS)** provided from Microsoft Azure as a Cloud Service
- Some capabilities provided are:
  - Data auto collection
    - Request rates, response times, and failure rates
    - Dependency rates, response times, and failure rates
    - AJAX calls from web pages
  - Diagnose exceptions, performance issues & ensure application’s availability
  - Analyze your application’s usage
    - Page views
    - User and session counts
    - Custom events
  - Visual Studio integration
  - Alerting, Export, Azure Integration

Application Insights - How it works



Application Insights in ASP.NET Core apps

- The **Application Insights SDK for ASP.NET Core** can monitor your applications no matter where or how they run
  - If your application is running and has network connectivity to Azure, telemetry can be collected
  - Application Insights monitoring is supported everywhere .NET Core is supported
- Prerequisites
  - A functioning ASP.NET Core application
  - A valid Application Insights instrumentation key. This key is required to send any telemetry to Application Insights



- **Operating system**: Windows, Linux, or Mac.
- **Hosting method**: In process or out of process.
- **Deployment method**: Framework dependent or self-contained.
- **Web server**: IIS (Internet Information Server) or Kestrel.
- **Hosting platform**: The Web Apps feature of Azure App Service, Azure VM, Docker, Azure Kubernetes Service (AKS), and so on.
- **.NET Core version**: All officially supported .NET Core versions.
- **IDE**: Visual Studio, VS Code, or command line.

Application Insights Integration

Microsoft

Thank You

- For the opportunity
- For participating
- For listening