MICROSOFT ARCHITECT
By Joydip Kanjilal, Columnist, InfoWorld
OCT 30, 2017 3:00 AM PDT

How to work with logging in ASP.Net Core

Take advantage of ASP.Net Core's flexible, customizable and extendable Logging API to plug in a custom logger with ease

Logging is an essential feature in applications for detecting or investigating issues. ASP.Net Core is an open source, cross-platform, lean, and modular framework for building high-performance web applications. Logging is now a first-class citizen in ASP.Net—support for logging is built in.

ASP.Net Core provides support for a wide variety of logging providers; you can plug in your own logging frameworks like Log4Net, NLog, and Elmah.

The Microsoft.Extensions.Logging namespace

Microsoft.Extensions.Logging is ASP.Net Core's logging subsystem that supports a few simple logging providers with just a few lines of code.

[DevSecOps: How to bring security into agile development and CI/CD]

If you are using ASP.Net Core 1.x, you need to include the Microsoft.Extensions.Logging dependencies to your project explicitly. In ASP.Net Core 2.x, these dependencies are included by default.

The Microsoft.Extensions.Logging namespace contains the following built-in classes and interfaces:

- ILogger
- ILoggingFactory
- LoggingFactory
- ILoggingProvider

The ILogger and ILoggerFactory interfaces are available in the Microsoft.Extensions.Logging.Abstractions namespace, and their default implementations reside in the Microsoft.Extensions.Logging namespace. The ILogger interface contains the necessary methods to log data to the underlying log storage. Here's what this interface looks like:

```
public interface ILogger
{
   void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception exceptio
   bool IsEnabled(LogLevel logLevel);
   IDisposable BeginScope<TState>(TState state);
}
```

So, to create your custom logger, you should extend this interface and implement its methods; I will show you how to do that later in this article.

The ILoggerFactory interface is used to create an instance of a type that implements the ILogger interface.

```
public interface ILoggerFactory : IDisposable
{
    ILogger CreateLogger(string categoryName);
    void AddProvider(ILoggerProvider provider);
}
```

ASP.Net Core contains a class called LoggerFactory. This class implements the ILoggerFactory interface. At runtime, the ASP.Net Core framework creates an instance of this class and registers it using its built-in IoC container.

The ILoggerProvider is an interface for creating an instance of the appropriate logger based on a specific category. You can implement the ILoggerProvider interface to build your own custom LoggerProvider:

```
public interface ILoggerProvider : IDisposable
{
   ILogger CreateLogger(string categoryName);
}
```

Configuring logging in ASP.Net Core

You can enable logging support by adding an instance of ILoggerProvider to the ILoggerFactory in the Configure method in the Startup.cs file as shown here:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory l
        {
            loggerFactory.AddProvider(new CustomLoggerProvider(new CustomLoggerProvider(
            {
                LogLevel = LogLevel.Information
            }));
            app.UseMvc();
        }
```

Next, you should create the necessary classes in your project. Here's the CustomLoggerProviderConfiguration class that contains two properties, the LogLevel and EventId. Note how default values have been set for both properties.

```
public class CustomLoggerProviderConfiguration
    {
        public LogLevel LogLevel { get; set; } = LogLevel.Warning;
        public int EventId { get; set; } = 0;
    }
```

Log levels indicate the severity of the messages being logged. ASP.Net Core defines six log levels: Trace, Warning, Debug, Information, Error, and Critical.

The CustomLoggerProvider class implements the ILoggerProvider interface and creates logger instances:

```
public class CustomLoggerProvider : ILoggerProvider
    {
        readonly CustomLoggerProviderConfiguration loggerConfig;
        readonly ConcurrentDictionary<string, CustomLogger> loggers =
         new   ConcurrentDictionary<string, CustomLogger>();
        public CustomLoggerProvider(CustomLoggerProviderConfiguration config)
        {
            loggerConfig = config;
        }
        public ILogger CreateLogger(string category)
        {
            return loggers.GetOrAdd(category,
             name => new CustomLogger(name, loggerConfig));
        }
        public void Dispose()
        {
            //Write code here to dispose the resources
        }
    }
```

Last, you need to create a custom logger class that extends the ILogger interface and implements its methods. I've named this class CustomLogger.

```
public class CustomLogger : ILogger
    {
        readonly string loggerName;
        readonly CustomLoggerProviderConfiguration loggerConfig;
        public CustomLogger(string name, CustomLoggerProviderConfiguration config)
        {
            this.loggerName = name;
            loggerConfig = config;
        }
        public IDisposable BeginScope<TState>(TState state)
        {
            return null;
        }
        public bool IsEnabled(LogLevel logLevel)
        {
            throw new NotImplementedException();
        }
        public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Except
        {
            string message = string.Format("{0}: {1} - {2}", logLevel.ToString(), eventI
            WriteTextToFile(message);
        }
        private void WriteTextToFile(string message)
        {
            string filePath ="D:\\IDGLog.txt";
            using (StreamWriter streamWriter = new StreamWriter(filePath, true))
            {
                streamWriter.WriteLine(message);
                streamWriter.Close();
            }
        }
    }
```

Thanks to ASP.Net Core's built-in support for Dependency Injection, you can get the instance of the ILogger in your controller methods easily. The following code listing illustrates how:

```
public class DefaultController : Controller
  {
        readonly ILogger<ValuesController> logger;
        public DefaultController(ILogger<ValuesController> log)
        {
            logger = log;
        }
        [HttpGet]
        public IEnumerable<string> Get()
        {
            logger.LogInformation("Hello, world!");
            return new string[] { "ASP.Net Core", "Version 2.0" };
        }
  }
```

You can take advantage of the built-in logging API in ASP.Net Core and extend it to log your application's data to a flat file, database, or any other configured log target. You can also use third-party logging providers like NLog or Serilog.

Joydip Kanjilal is a Microsoft MVP in ASP.Net, as well as a speaker and author of several books and articles. He has more than 20 years of experience in IT including more than 16 years in Microsoft .Net and related technologies.

Follow

UNITED STATES

Tim O'Reilly: the golden age of the programmer is over

The 24 highest paying developer roles in 2020

Are industry clouds an opportunity or a distraction?

6 ways Alibaba Cloud challenges AWS, Azure, and GCP

Make life easy with ssh_config

Red Hat Enterprise Linux takes aim at edge computing

3 enterprise AI success stories

Microsoft Visual Studio 2022 preview is coming

The decline of Heroku

When to use Task.WaitAll vs. Task.WhenAll in .NET

8 databases supporting in-database machine learning