**Логгирование**

**Ведение лога и ILogger**

ASP.NET Core имеет встроенную поддержку логгирования, что позволяет применять логгирование с минимальными вкраплениями кода в функционал приложения.

Для логгирования данных нам необходим объект **ILogger<T>**. По умолчанию среда ASP NET Core через механизм внедрения зависимостей уже предоставляет нам такой объект. Его можно получить как и любую другую зависимость в приложении. Также этот объект можно получить через свойство **Logger** объекта WebApplication.

Например, используем встроенный логгер для логгирования на консоль приложения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    app.Run(async (context) =>  {      // пишем на консоль информацию      app.Logger.LogInformation($"Processing request {context.Request.Path}");        await context.Response.WriteAsync("Hello World!");  });    app.Run(); |

В данном случае через свойство **app.Logger** получаем встроенный логгер и с помощью его метода logger.LogInformation передаем на консоль некоторую информацию.

При обращении к приложению с помощью следующего запроса *http://localhost:xxxxx/hello* на консоль будет выведена информация, переданная логгером:

**Категория логгера**

При создании логгера для него указывается категория. Обычно в качестве категории логгера выступает класс, в котором используется логгер. В этом случае логгер типизируется классом-категории. Например, логгер, для которого в качестве категории выступает класс Program:

|  |  |
| --- | --- |
| 1 | ILogger<Program> |

В чем смысл категории? Категория задает текстовую метку, с которой ассоциируется сообщение логгера, и в выводе лога мы ее можем увидеть.

Где это может быть полезно? Например, у нас есть несколько классов middleware, где ведется логгирование. Указывая в качестве категории текущий класс, в последствии в логе мы можем увидеть, в каком классе именно было создано данное сообщение лога. Поэтому, как правило, в качестве категории указывается текущий класс, но в принципе это необязательно.

**Получение логгера через внедрение зависимостей**

Поскольку логгер добавляется в сервисы приложения, то мы можем получить его как и любой другой сервис через систему внедрения зависимостей. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    app.Map("/hello", (ILogger<Program> logger) =>  {      logger.LogInformation($"Path: /hello  Time: {DateTime.Now.ToLongTimeString()}");      return "Hello World";  });    app.Run(); |

В данном случае при обращении по адресу "/hello" сработает конечная точка, в обработчике которой через механизм внедрения зависимостей можно получить объект логгера. Стоит учитывать, что в этом случае для логгера надо определить категорию. Здесь в качестве категории применяется класс Program (неявный класс, в котором и запускается приложение).

В самом обработчике логгер выводит на консоль путь запроса и время запроса:

**Уровни и методы логгирования**

При настройке логгирования мы можем установить уровень детализации информации с помощью одного из значений перечисления **LogLevel**. Всего мы можем использовать следующие значения:

* Trace: используется для вывода наиболее детализированных сообщений. Подобные сообщения могут нести важную информацию о приложении и его строении, поэтому данный уровень лучше использовать при разработке, но никак не при публикации
* Debug: для вывода информации, которая может быть полезной в процессе разработки и отладки приложения
* Information: уровень сообщений, позволяющий просто отследить поток выполнения приложения
* Warning: используется для вывода сообщений о неожиданных событиях, например, ошибках, которые не останавливают выполнение приложения, но в то же время должны быть исследованы
* Error: для вывода информации об ошибках и исключениях, которые возникли при текущей операции и которые не могут быть обработаны
* Critical: уровень критических ошибок, которые требуют немедленной реакции - ошибками операционной системы, потерей данных в бд, переполнение памяти диска и т.д.
* None: вывод информации в лог не применяется

Для вывода соответствующего уровня информации у объекта ILogger определены соответствующие методы расширения:

* LogDebug()
* LogTrace()
* LogInformation()
* LogWarning()
* LogError()
* LogCritical()

Так, в примере выше для вывода информации на консоль использовался метод LogInformation().

Вывод сообщений уровня Trace по умолчанию отключен.

Каждый такой метод имеет несколько перегрузок, которые могут принимать ряд различных параметров:

* string data: строковое сообщение для лога
* int eventId: числовой идентификатор, который связан с логом. Идентификатор должен быть статическим и специфическим для определенной части логгируемых событий.
* string format: строковое сообщения для лога, которое может содержать параметры
* object[] args: набор параметров для строкового сообщения
* Exception error: логгируемый объект исключения

Также для логгирования определен общий метод **Log()**, который позволяет определить уровень логгера через один из параметров:

|  |  |
| --- | --- |
| 1 | logger.Log(LogLevel.Information, $"Requested Path: {context.Request.Path}"); |

При стандартном логгировании на консоль для каждого уровня/метода определен своя метка и цветовой маркер, которые позволяют сразу выделить сообщение соответствующего уровня. Например, при запуске следующего кода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    app.Run(async (context) =>  {      var path = context.Request.Path;      app.Logger.LogCritical($"LogCritical {path}");      app.Logger.LogError($"LogError {path}");      app.Logger.LogInformation($"LogInformation {path}");      app.Logger.LogWarning($"LogWarning {path}");        await context.Response.WriteAsync("Hello World!");  });    app.Run(); |

**Фабрика логгера и провайдеры логгирования**

В примерах в прошлой теме мы получали объект логгера, который добавляется через DI. Но мы можем также использовать фабрику логгера для его создания:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    ILoggerFactory loggerFactory = LoggerFactory.Create(builder => builder.AddConsole());  ILogger logger = loggerFactory.CreateLogger<Program>();  app.Run(async (context) =>  {      logger.LogInformation($"Requested Path: {context.Request.Path}");      await context.Response.WriteAsync("Hello World!");  });    app.Run(); |

В данном случае с помощью метода **LoggerFactory.Create** создается фабрика логгера в виде объекта **ILoggerFactory**. В качестве параметра метод принимает делегат, который устанавливает некоторые настройки логгирования. В частности, метод **AddConsole()** объекта ILoggingBuilder устанавливает вывод сообщений лога на консоль. Затем метод CreateLogger() фабрики собственно создает логгер:

|  |  |
| --- | --- |
| 1 | ILogger logger = loggerFactory.CreateLogger<Program>(); |

Метод CreateLogger() типизируется классом, который представляет категорию. В данном случае это класс Program, в котором неявно выполняется данный код. Но в качестве альтернативы название категории можно передать в метод в качестве параметра в виде строки:

|  |  |
| --- | --- |
| 1 | ILogger logger = loggerFactory.CreateLogger("WebApplication"); |

В итоге мы получим тот же вывод сообщений на консоль. Но преимущество использования фабрики логгеров состоит в том, что мы можем дополнительно настроить различные параметры логгирования, в частности, провайдер логгирования.

**Получение фабрики логгера через dependency injection**

Как и логгер, фабрика логгера доступна в приложении в виде сервиса, соответственно ее можно получит через механизм внедрения зависимостей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    app.Map("/hello", (ILoggerFactory loggerFactory)=>{        // создаем логгер с категорией "MapLogger"      ILogger logger = loggerFactory.CreateLogger("MapLogger");      // логгируем некоторое сообщение      logger.LogInformation($"Path: /hello   Time: {DateTime.Now.ToLongTimeString()}");      return "Hello World!";  });    app.Run(); |

**Провайдеры логгирования**

В примере выше логгирование шло на консоль. Вообще путь логгирования определяется провайдером логгирования. По умолчанию ASP.NET Core предоставляет следующие провайдеры:

* **Console**: вывод информации на консоль. Устанавливается методом **AddConsole()**
* **Debug**: использует для ведения записей лога класс **System.Diagnostics.Debug** и в частности его метод Debug.WriteLine. Соответственно все записи лога мы можем увидеть в окне Output в Visual Studio. Устанавливается методом **AddDebug()**. Стоит отметить, что данный способ работает только при запуске проекта в режиме отладки
* **EventSource**: на Windows введет логгирование в лог [ETW](https://docs.microsoft.com/en-us/windows/win32/etw/event-tracing-portal) (Event Tracing for Windows), для просмотра которого может использоваться инструмент PerfView (или аналогичный инструменты). Хотя данный провайдер задумывался как кроссплатформенный, для Linux и MacOS пока назначение лога не определено. Устанавливается методом **AddEventSourceLogger()**
* **EventLog**: записывает в Windows Event Log, соответственно работает только при запуске на Windows. Устанавливается методом **AddEventLog()**

Например, вместо консоли зададим вывод лога в окне Output в Visual Studio:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    var loggerFactory = LoggerFactory.Create(builder => builder.AddDebug());  ILogger logger = loggerFactory.CreateLogger<Program>();  app.Run(async (context) =>  {      logger.LogInformation($"Requested Path: {context.Request.Path}");      await context.Response.WriteAsync("Hello World!");  });    app.Run(); |

**Конфигурация и фильтрация логгирования**

Для логгера можно задать конфигурацию с помощью одного из следующих источников:

* Файлы (json, xml)
* Аргументы командной строки
* Переменные среды окружения
* Объекты .NET
* Незашифрованное хранилище Secret Manager

Также можно создать свой провайдер конфигурации логгера.

Например, по умолчанию в проект по типу Empty добавляется файл **appsettings.json**, который предназначен для конфигурации проекта и в том числе может определять и по умолчанию уже содержит конфигурацию логгера:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | {    "Logging": {      "LogLevel": {        "Default": "Information",        "Microsoft.AspNetCore": "Warning"      }    },    "AllowedHosts": "\*"  } |

Узел **"Logging"** задает настройки логгирования. Далее узел **LogLevel** задает **минимальный уровень** логгирования для двух категорий. Категория "Default" представляет универсальное применение ко всем категориям. То есть по умолчанию будут логгироваться все сообщения уровня Information. Но для логгера с категорией "Microsoft.AspNetCore" будут логгироваться только сообщения уровня Warning.

Мы можем, например, убрать логгирование для категории "Microsoft.AspNetCore":

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {    "Logging": {      "LogLevel": {        "Default": "Information",      }    },    "AllowedHosts": "\*"  } |

Или установить логгирование связки различных уровней и категорий для различных провайдеров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | {    "Logging": {      "Debug": {        "LogLevel": {          "Default": "Debug"        }      },      "Console": {        "LogLevel": {          "Default": "Information",          "Microsoft.AspNetCore": "Warning"        }      },      "LogLevel": {        "Default": "Error"      }    },    "AllowedHosts": "\*"  } |

Здесь определяются четыре правила. Одно правило определено для провайдера Debug:

|  |  |
| --- | --- |
| 1  2  3  4  5 | "Debug": {      "LogLevel": {          "Default": "Debug"      }  } |

То есть для провайдера Debug определено логгирование сообщения уровня Debug для всех категорий.

Для провайдера Console определено два правила:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | "Console": {      "LogLevel": {          "Default": "Information",          "Microsoft.AspNetCore": "Warning"      }  } |

В данном случае для всех категорий будут логироваться сообщения уровня "Information". Для категории "Microsoft.AspNetCore" логируются сообщения уровня "Warning".

И также определено одно общее правило для всех провайдеров:

|  |  |
| --- | --- |
| 1  2  3 | "LogLevel": {      "Default": "Error"  } |

Логгирование сообщений уровня "Error" для всех категорий.

Следует учитывать, что настройки узла Logging.{имя\_провайдера}.LogLevel переопределяют настройки в узле Logging.LogLevel

Также можно настроить применяемые фильтры программно с помощью фабрики логгера:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.Extensions.Logging.Debug;  //....................    var loggerFactory = LoggerFactory.Create(builder =>  {      builder.AddDebug();      builder.AddConsole();      // настройка фильтров      builder.AddFilter("System", LogLevel.Information)              .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Trace);  });  ILogger logger = loggerFactory.CreateLogger("WebApplication"); |

Фильтр логгирования задается с помощью метода **AddFilter()**, который позволяет задать для определенной категории определенный уовень логгирования. Например:

|  |  |
| --- | --- |
| 1 | builder.AddFilter("System", LogLevel.Information) |

устанавает для категории "System" уровень LogLevel.Information для всех провайдеров. Это все равно, если бы мы написали в файле appsettings.json:

|  |  |
| --- | --- |
| 1  2  3  4 | "Logging": {      "LogLevel": {        "System": "Information"  } |

Второй вызов AddFilter обобщенный - обобщенный класс указывает на провайдер, для которого задается фильтр. То есть для провайдера DebugLoggerProvider для категории "Microsoft" устанавливается уровень LogLevel.Trace:

|  |  |
| --- | --- |
| 1 | AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Trace); |

Этот вызов аналогичен следующему определению в файле конфигурации

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | "Logging": {      "Debug": {        "LogLevel": {          "Microsoft": "Trace"      }  } |

Можно определять фильтры сразу и в коде C#, и в файле конфигурации. При создании логгера объект ILoggerFactory выбирает и применяет одно правило для каждого провайдера. Выбор нужного правила состоит из ряда этапов:

1. Выбираются все правила, которые соответствуют провайдеру. Если таких правил нет, то выбираются все правила, общие для всех провайдеров.
2. Из выбранных на предыдущем шаге правил выбирается правило, которое имеет наиболее длинное соответствие имени категории. Если такого нет, то выбираются все правила, общие для всех категорий.
3. Если на предыдущем шаге выбрано несколько правил, из них выбирается самое последнее (Порядковый номер правила соответствует его порядку определения в файле конфигурации или в коде. Причем правила из файла конфигурации предшествуют правилам, определенным в коде).
4. Если на предыдущем шаге не выбрано никаких правил, то применяется настройка MinimumLevel. Данную опцию также можно определить в коде, например, при настройке фабрики логгера:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var loggerFactory = LoggerFactory.Create(builder =>  {      builder.AddDebug();      builder.AddFilter("System", LogLevel.Debug)              .SetMinimumLevel(LogLevel.Debug);   // Определение MinimumLevel    }); |

**Глобальная настройка логгирования**

Для глобальной установки настроек логгирования у объекта WebApplicationBuilder определено свойсво **Logging**, которое представляет тип **ILogingBuilder**. Для управления логгирования он предоставляет ряд методов:

* AddConfiguration(): добавляет конфигурацию логгера в виде объекта IConfiguration
* AddConsole(): добавляет консольный логгер
* AddConsoleFormatter(): добавляет объект форматирования для консольного вывода
* AddJsonConsole(): добавляет форматирование сообщений консольного логгера в формат "json"
* AddSimpleConsole(): добавляет простое форматирование для логгирования на консоль
* AddDebug(): добавляет логгер отладки
* AddEventLog(): добавляет логгер для вывод в журнал Windows Event Log
* AddEventSourceLogger(): добавляет логгер для вывода в лог Event Tracing for Windows
* AddFilter(): добавляет фильтрацию для сообщений логгера
* AddProvider(): добавляет провайдер логгирования
* ClearProviders(): удаляет все зарегистрированные провайдеры логгирования
* SetMinimumLevel(): устанавливает минимальный уровень логгирования

Например:

|  |  |
| --- | --- |
| 1  2  3  4 | var builder = WebApplication.CreateBuilder();  builder.Logging.ClearProviders();   // удаляем все провайдеры  builder.Logging.AddConsole();   // добавляем провайдер для логгирования на консоль  var app = builder.Build(); |

**Создание провайдера логгирования**

Стандартная инфраструктура ASP NET Core предоставляет, возможно, не самые удобные способы логгирования - на консоль, в окне Output в Visual Studio. Однако в то же время ASP.NET Core позволяет полностью определить свою логику ведения лога. Допустим, мы хотим сохранять сообщения в текстовом файле.

Вначале добавим в проект новый класс **FileLogger**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | public class FileLogger : ILogger, IDisposable  {      string filePath;      static object \_lock = new object();      public FileLogger(string path)      {          filePath = path;      }      public IDisposable BeginScope<TState>(TState state)      {          return this;      }        public void Dispose() { }        public bool IsEnabled(LogLevel logLevel)      {          //return logLevel == LogLevel.Trace;          return true;      }        public void Log<TState>(LogLevel logLevel, EventId eventId,                  TState state, Exception? exception, Func<TState, Exception?, string> formatter)      {          lock (\_lock)          {              File.AppendAllText(filePath, formatter(state, exception) + Environment.NewLine);          }      }  } |

Класс логгера должен реализовать интерфейс **ILogger**. Этот интерфейс определяет три метода:

* BeginScope: этот метод возвращает объект IDisposable, который представляет некоторую область видимости для логгера. В данном случае нам этот метод не важен, поэтому возвращаем значение this - ссылку на текущий объект класса, который реализует интерфейс IDisposable.
* IsEnabled: возвращает значения true или false, которые указыват, доступен ли логгер для использования. Здесь можно здать различную логику. В частности, в этот метод передается объект LogLevel, и мы можем, к примеру, задействовать логгер в зависимости от значения этого объекта. Но в данном случае просто возвращаем true, то есть логгер доступен всегда.
* Log: этот метод предназначен для выполнения логгирования. Он принимает пять параметров:
  + LogLevel: уровень детализации текущего сообщения
  + EventId: идентификатор события
  + TState: некоторый объект состояния, который хранит сообщение
  + Exception: информация об исключении
  + formatter: функция форматирования, которая с помощью двух предыдущих параметов позволяет получить собственно сообщение для логгирования

И в данном методе как раз и производится запись в текстовый файл. Путь к этому файлу передается через конструктор

Далее добавим в проект класс **FileLoggerProvider**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class FileLoggerProvider : ILoggerProvider  {      string path;      public FileLoggerProvider(string path)      {          this.path = path;      }      public ILogger CreateLogger(string categoryName)      {          return new FileLogger(path);      }        public void Dispose() {}  } |

Этот класс представляет провайдер логгирования. Он должен реализовать интерфейс **ILoggerProvider**. В этом интерфейсе определны два метода:

* **CreateLogger**: создает и возвращает объект логгера. Для создания логгера используется путь к файлу, который передается через конструктор
* **Dispose**: управляет освобождение ресурсов. В данном случае пустая реализация

Теперь создадим вспомогательный класс **FileLoggerExtensions**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public static class FileLoggerExtensions  {      public static ILoggingBuilder AddFile(this ILoggingBuilder builder, string filePath)      {          builder.AddProvider(new FileLoggerProvider(filePath));          return builder;      }  } |

Этот класс добавляет к объекту ILoggingBuilder метод расширения AddFile, который будет добавлять наш провайдер логгирования.

Теперь используем провайдер в файле **Program.cs**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var builder = WebApplication.CreateBuilder();  // устанавливаем файл для логгирования  builder.Logging.AddFile(Path.Combine(Directory.GetCurrentDirectory(), "logger.txt"));  // настройка логгирования с помошью свойства Logging идет до  // создания объекта WebApplication  var app = builder.Build();    app.Run(async (context) =>  {      app.Logger.LogInformation($"Path: {context.Request.Path}  Time:{DateTime.Now.ToLongTimeString()}");      await context.Response.WriteAsync("Hello World!");  });    app.Run(); |

За глобальную установку настроек логгирования отвечает свойство **Logging** класса WebApplicationBuilder. Это свойство представляет объект ILoggingBuilder и предоставляет ряд методов для управления логгированием. И в данном случае с помошью выше определенного метода AddFile добавляем логгирование в файл.

Стоит отметить, что глобальная настройка логгирования должна идти до создания объекта WebApplication.

Теперь для логгирования также будет использоваться файл logger.txt, который будет создаваться в папке проекта.