



# Распределенные информационно-аналитические СИСТЕМЫ

Практическое занятие № 11. «Статические файлы. Логгирование»

Профессор кафедры КБ-2: д.т.н. Шатовкин Р.Р.

# Учебные вопросы:

## 1. Статические файлы.

1.1. Установка каталога статических файлов. `UseStaticFiles`.

1.2. Работа со статическими файлами.

## 2. Логгирование.

2.1. Ведение лога и `ILogger`.

2.2. Фабрика логгера и провайдеры логгирования.

2.3. Конфигурация и фильтрация логгирования.

2.4. Создание провайдера логгирования.

# 1. Статические файлы

## 1.1. Установка каталога статических файлов. UseStaticFiles

Ранее уже рассматривалась отправка статических файлов. В частности, для отправки файлов мы могли использовать метод **SendFileAsync()** объекта **HttpResponse**:

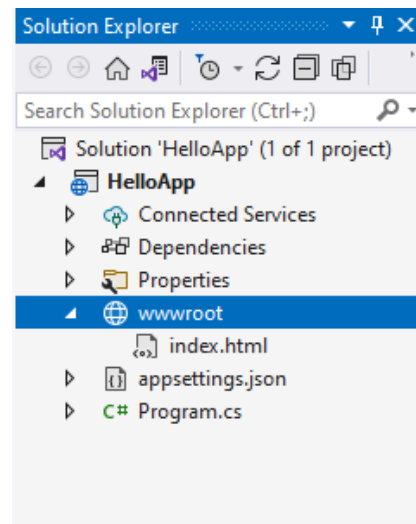
```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.Run(async (context) => await context.Response.SendFileAsync("index.html"));
5
6 app.Run();
```

Однако **ASP.NET Core** также предоставляет специальный встроенный **middleware**, который подключается с помощью метода **UseStaticFiles()** и который упрощает работу со статическими файлами.

При использовании этого **middleware** применяются некоторые условности. В частности, по умолчанию для определения пути хранения статических файлов в проекте используются два параметра **ContentRoot** и **WebRoot**, а сами статические файлы должны помещаться в каталог **ContentRoot/WebRoot**. На стадии разработки параметр **"ContentRoot"** соответствует каталогу текущего проекта. А параметр **"WebRoot"** по умолчанию представляет папку **wwwroot** в рамках каталога **ContentRoot**. То есть, исходя из значений по умолчанию, то статические файлы следует располагать в папке **"wwwroot"**, которая должна находиться в текущем проекте. Однако эти параметры при необходимости можно переопределить.

В разных типах проектов **ASP NET Core** данная папка может уже быть по умолчанию в проекте, а может отсутствовать. Например, в проекте по типу **Empty** данная папка отсутствует, поэтому ее надо добавлять вручную.

Итак, возьмем проект по типу **Empty** и добавим в него новую папку **wwwroot**. Далее добавим в папку **wwwroot** новый файл **index.html**. То есть у нас получится следующий проект:



Изменим код файла **index.html**, например, следующим образом:

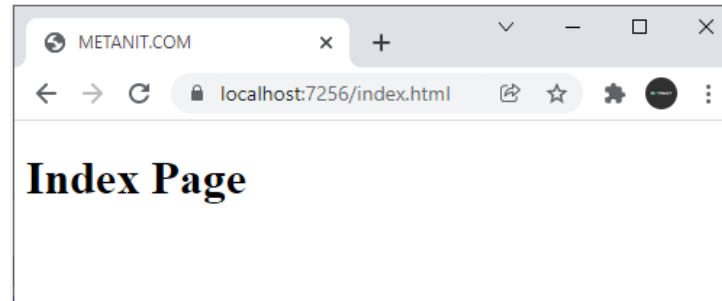
```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta charset="utf-8" />
5     <title>METANIT.COM</title>
6 </head>
7 <body>
8     <h1>Index Page</h1>
9 </body>
10 </html>
```

Для того, чтобы клиенты могли обращаться к этому файлу, подключим соответствующий компонент **middleware** с помощью метода **UseStaticFiles()**:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.UseStaticFiles(); // добавляем поддержку статических файлов
5
6 app.Run(async (context) => await context.Response.WriteAsync("Hello World"));
7
8 app.Run();
```

Для подключения функциональности работы со статическими файлами применяется метод **UseStaticFiles()**, который реализован как метод расширения для типа **IApplicationBuilder**.

Теперь, если мы обратимся к добавленному файлу, например, по пути **/index.html**, то нам отобразится содержимое данной веб-страницы:

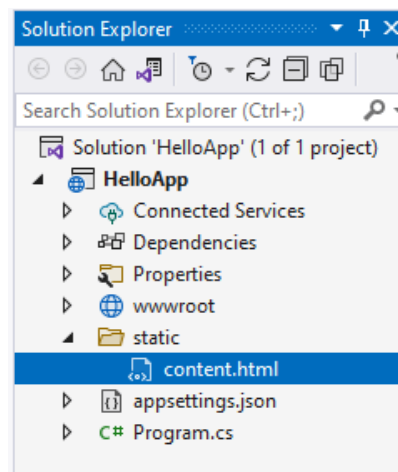


По всем остальным запросам браузер выводил бы строку **"Hello World"**.

Если бы **index.html** находился бы в какой-то вложенной папке, например, в **wwwroot/html/**, то для обращения к нему мы могли бы использовать путь **/html/index.html**. То есть **middleware** для работы со статическими сайтами автоматически сопоставляет запросы с путями к статическим файлам в рамках папки **wwwroot**.

## Изменение пути к статическим файлам

Что делать, если нас не устраивает стандартная папка **wwwroot**. И мы, к примеру, хотим, чтобы все статические файлы в проекте находились в папке **static**. Для этого добавим в проект папку **static** в проект и определим в ней какой-нибудь html-файл. Пусть он будет называться **content.html**:



Допустим, в этом файле будет какое-нибудь простейшее содержимое:

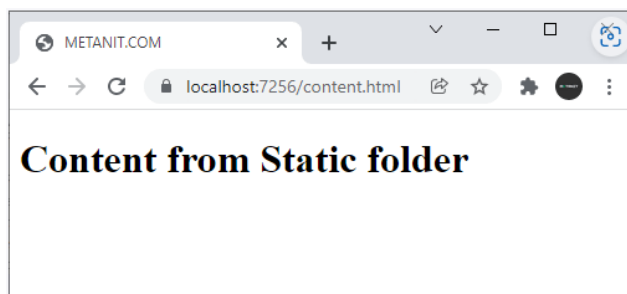
```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta charset="utf-8" />
5     <title>METANIT.COM</title>
6 </head>
7 <body>
8     <h1>Content from Static folder</h1>
9 </body>
10 </html>
```

Чтобы приложение восприняло эту папку, изменим код создания хоста в файле **Program.cs**:

```
1 var builder = WebApplication.CreateBuilder(
2     new WebApplicationOptions { WebRootPath = "static"}); // изменяем папку для хранения статики
3
4 var app = builder.Build();
5
6 app.UseStaticFiles(); // добавляем поддержку статических файлов
7
8 app.Run(async (context) => await context.Response.WriteAsync("Hello World"));
9
10 app.Run();
```

Для добавления пути к файлам используется перегруженная версия метода **CreateBuilder()**, которая в качестве параметра принимает объект **WebApplicationOptions**. Его свойство **WebRootPath** позволяет установить папку для статических файлов.

И после этого мы также сможем обращаться к статическим файлам из папки **static**:

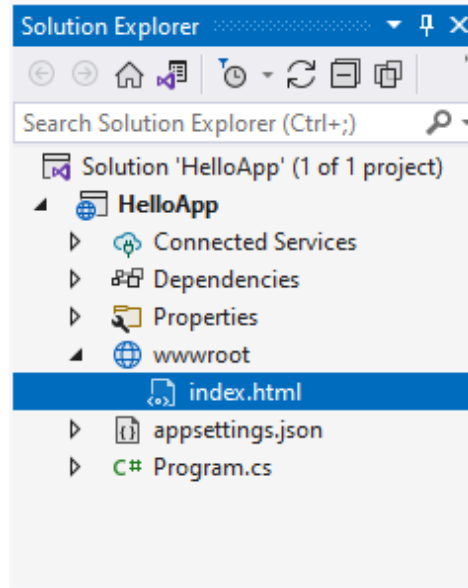


## 1.2. Работа со статическими файлами

Рассмотрим некоторые возможности, которые предоставляет нам **ASP.NET Core** для работы со статическими файлами.

### Файлы по умолчанию

Допустим, в проекте в папке **wwwroot** располагается файл **index.html**:



С помощью специального метода расширения **UseDefaultFiles()** можно настроить отправку статических веб-страниц по умолчанию без обращения к ним по полному пути:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.UseDefaultFiles(); // поддержка страниц html по умолчанию
5 app.UseStaticFiles();
6
7 app.Run(async (context) => await context.Response.WriteAsync("Hello World"));
8
9 app.Run();
```

В этом случае при отправке запроса к корню веб-приложения типа <http://localhost:xxxx/> приложение будет искать в папке **wwwroot** следующие файлы:

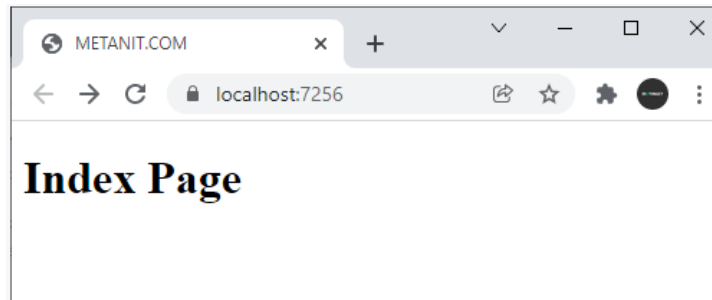
**default.htm**

**default.html**

**index.htm**

**index.html**

Если файл будет найден, то он будет отправлен в ответ клиенту.



Если же файл не будет найден, то продолжается обычная обработка запроса с помощью следующих компонентов **middleware**. То есть фактически это будет аналогично, как будто мы обращаемся к файлу: <http://localhost/index.html>.

Если же мы хотим использовать файл, название которого отличается от вышеперечисленных, то нам надо в этом случае применить объект **DefaultFileOptions**:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 DefaultFileOptions options = new DefaultFileOptions();
5 options.DefaultFileNames.Clear(); // удаляем имена файлов по умолчанию
6 options.DefaultFileNames.Add("hello.html"); // добавляем новое имя файла
7 app.UseDefaultFiles(options); // установка параметров
8
9 app.UseStaticFiles();
10
11 app.Run(async (context) => await context.Response.WriteAsync("Hello World"));
12
13 app.Run();
```

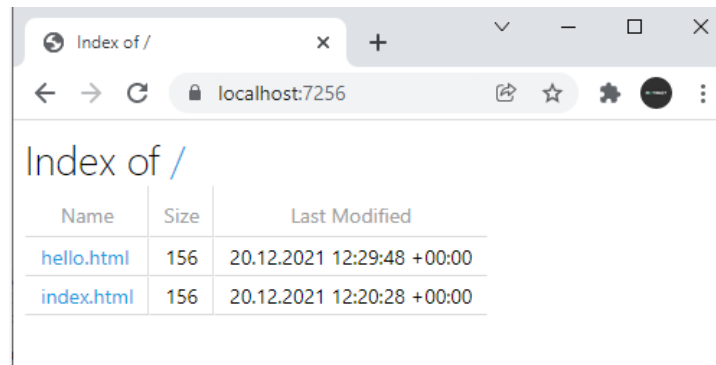
В этом случае в качестве страницы по умолчанию будет использоваться файл **hello.html**, который должен располагаться в папке **wwwroot**.



## Метод UseDirectoryBrowser

Метод **UseDirectoryBrowser** позволяет пользователям просматривать содержимое каталогов на сайте:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.UseDirectoryBrowser();
5 app.UseStaticFiles();
6
7 app.Run();
```

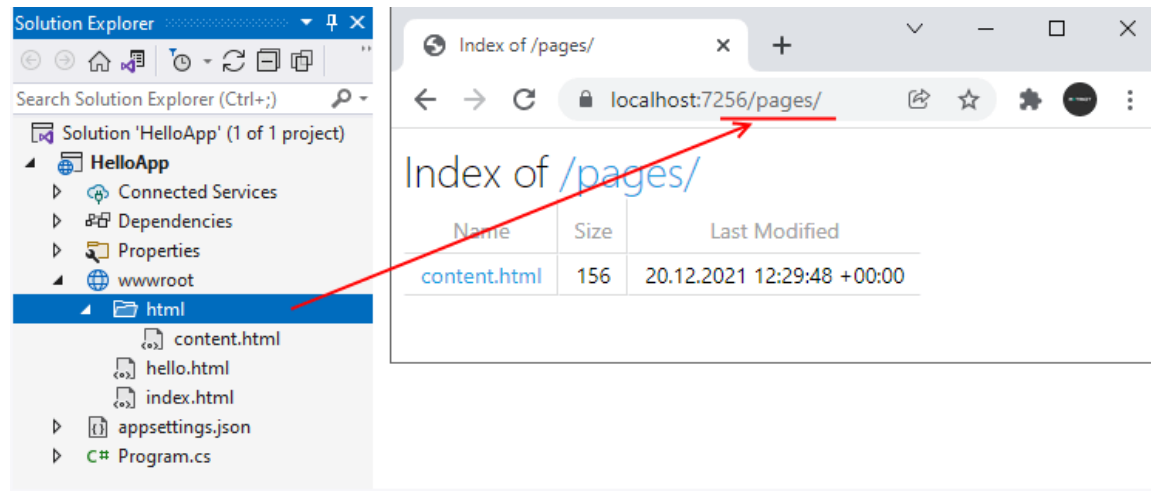


Данный метод имеет перегрузку, которая позволяет сопоставить определенный каталог на жестком диске или в проекте с некоторой строкой запроса и тем самым потом отобразить содержимое этого каталога:

```
1 using Microsoft.Extensions.FileProviders;
2
3 var builder = WebApplication.CreateBuilder();
4 var app = builder.Build();
5
6 app.UseDirectoryBrowser(new DirectoryBrowserOptions()
7 {
8     FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\html")),
9
10    RequestPath = new PathString("/pages")
11 });
12 app.UseStaticFiles();
13
14 app.Run();
```

Чтобы задействовать новый функционал, надо подключить пространство имен **using Microsoft.Extensions.FileProviders**.

В качестве параметра метод **UseDirectoryBrowser()** принимает объект **DirectoryBrowserOptions**, который позволяет настроить сопоставление путей к файлам с каталогами. Так, в данном случае путь типа **http://localhost:xxxx/pages/** будет сопоставляться с каталогом **"wwwroot\html"**.

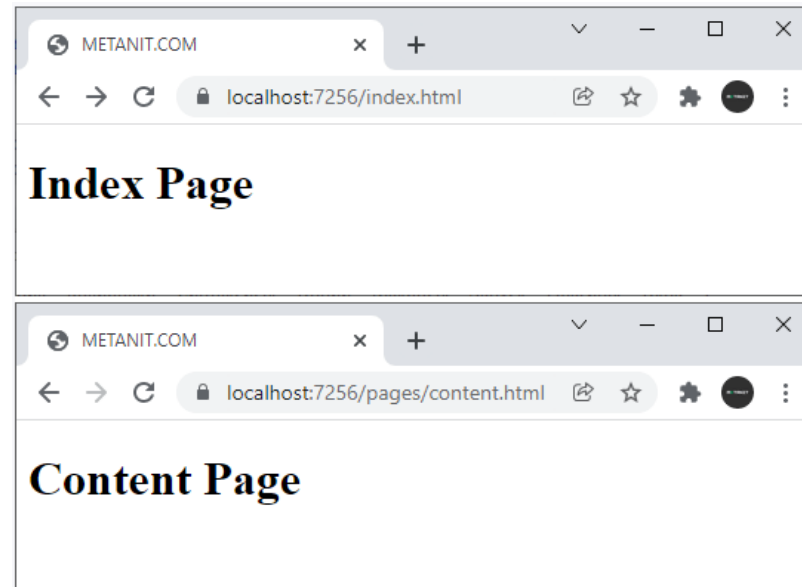


## Сопоставление каталогов с путями

Перегрузка метода **UseStaticFiles()** позволяет сопоставить пути с определенными каталогами:

```
1 using Microsoft.Extensions.FileProviders;
2
3 var builder = WebApplication.CreateBuilder();
4 var app = builder.Build();
5
6
7 app.UseStaticFiles();
8 app.UseStaticFiles(new StaticFileOptions() // обрабатывает запросы к каталогу wwwroot/html
9 {
10     FileProvider = new PhysicalFileProvider(
11         Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\html")),
12     RequestPath = new PathString("/pages")
13 });
14
15 app.Run();
```

Первый вызов **app.UseStaticFiles()** обрабатывает запросы к файлам в папке **wwwroot**. Второй вызов принимает те же параметры, что и метод **app.UseDirectoryBrowser()** в предыдущем примере. И в отличие от первого вызова он обрабатывает запросы по пути **http://localhost:xxxx/pages**, сопоставляя данные запросы с папкой **wwwroot/html**. К примеру, по запросу **http://localhost:xxxx/pages/index.html** мы можем обратиться к файлу **wwwroot/html/index.html**.



## Метод UseFileServer

Метод **UseFileServer()** объединяет функциональность сразу всех трех вышеописанных методов **UseStaticFiles**, **UseDefaultFiles** и **UseDirectoryBrowser**:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.UseFileServer();
5
6 app.Run();
```

По умолчанию этот метод позволяет обрабатывать статические файлы и отправлять файлы по умолчанию типа `index.html`. Если нам надо еще включить просмотр каталогов, то мы можем использовать перегрузку данного метода:

```
1 app.UseFileServer(enableDirectoryBrowsing: true);
```

Еще одна перегрузка метода позволяет более точно задать параметры:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.UseFileServer(new FileServerOptions
5 {
6     EnableDirectoryBrowsing = true,
7     EnableDefaultFiles = false
8 });
9
10 app.Run();
```

Также можно настроить сопоставление путей запроса с каталогами:

```
1 using Microsoft.Extensions.FileProviders;
2
3 var builder = WebApplication.CreateBuilder();
4 var app = builder.Build();
5
6 app.UseFileServer(new FileServerOptions
7 {
8     EnableDirectoryBrowsing = true,
9     FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\html")),
10    RequestPath = new PathString("/pages"),
11    EnableDefaultFiles = false
12 });
13
14 app.Run();
```

В этом случае будет разрешен обзор каталога по пути `http://localhost:xxxx/pages/`, но при этом путь `http://localhost:xxxx/html/` работать не будет.

## 2. Логгирование

### 2.1. Ведение лога и ILogger

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

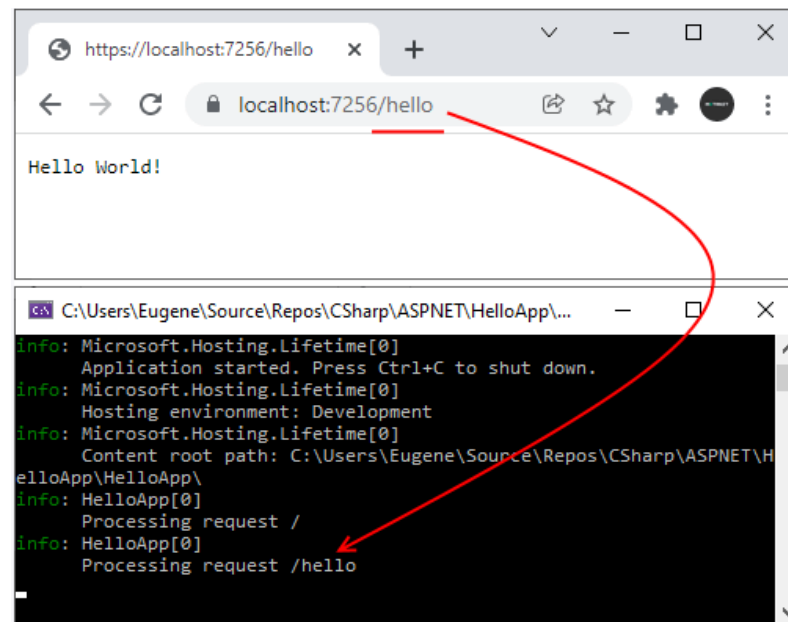
Для логгирования данных нам необходим объект **ILogger<T>**. По умолчанию среда **ASP NET Core** через механизм внедрения зависимостей уже предоставляет нам такой объект. Его можно получить как и любую другую зависимость в приложении. Также этот объект можно получить через свойство **Logger** объекта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

Например, используем встроенный логгер для логгирования на консоль приложения:

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

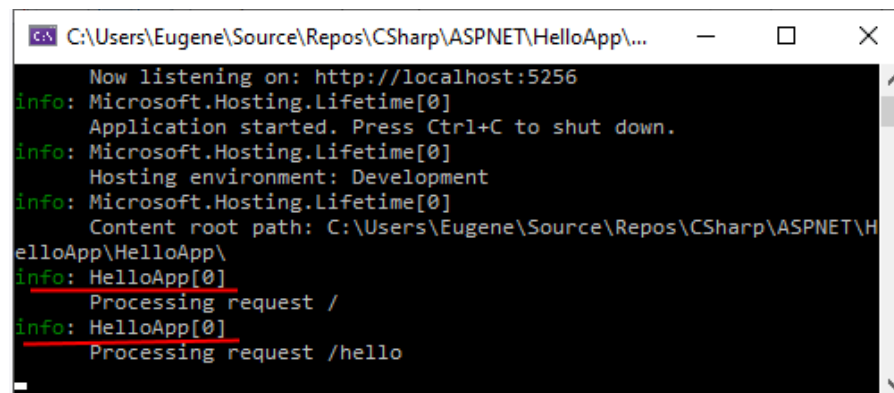


## Категория логгера

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 логирование. Указывая в качестве категории текущий класс, в последствии в логе мы можем увидеть, в каком классе именно было создано данное сообщение логa. Поэтому, как правило, в качестве категории указывается текущий класс, но в принципе это необязательно.

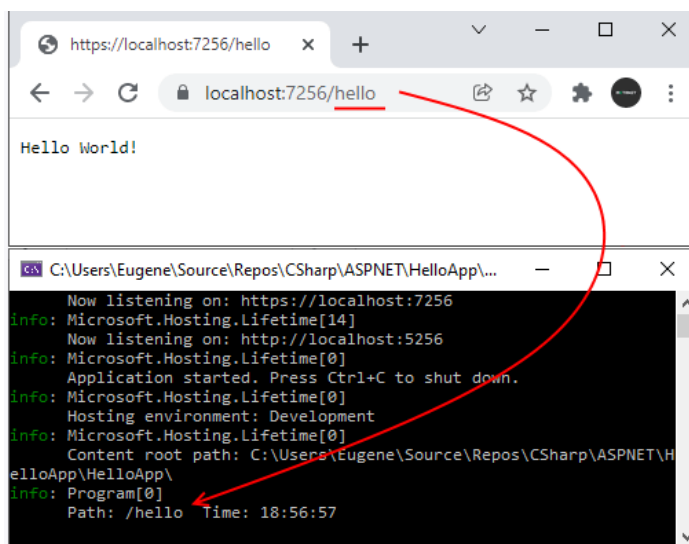
## Получение логгера через внедрение зависимостей

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

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

В данном случае при обращении по адресу **"/hello"** сработает конечная точка, в обработке 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 логируемых событий.

**string format**: строковое сообщения для лога, которое может содержать параметры.

**object[] args**: набор параметров для строкового сообщения.

**Exception error**: логируемый объект исключения.



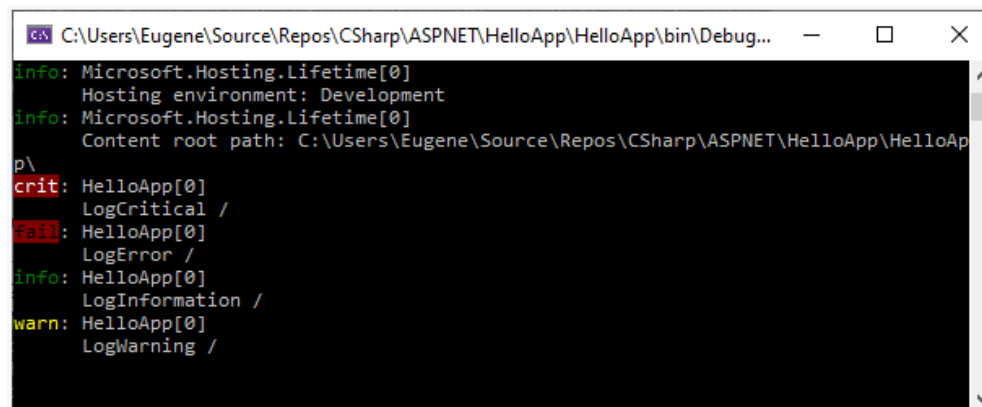
Также для логгирования определен общий метод **Log()**, который позволяет определить уровень логгера через один из параметров:

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

мы получим следующий лог на консоль:



```
C:\Users\Eugene\Source\Repos\CSharp\ASPNET\HelloApp\HelloApp\bin\Debug...
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\Eugene\Source\Repos\CSharp\ASPNET\HelloApp\HelloApp\p\
crit: HelloApp[0]
      LogCritical /
fail: HelloApp[0]
      LogError /
info: HelloApp[0]
      LogInformation /
warn: HelloApp[0]
      LogWarning /
```

## 2.2. Фабрика логгера и провайдеры логгирования

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

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

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

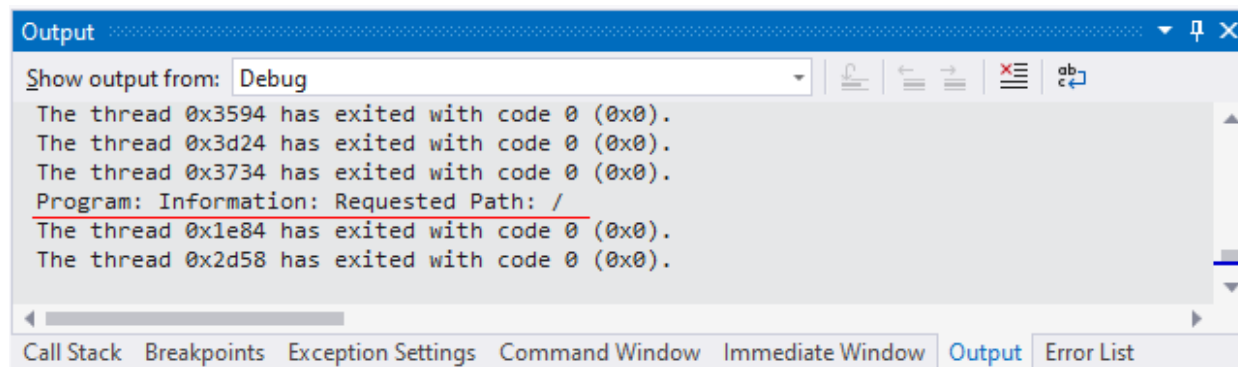
**Debug**: использует для ведения записей лога класс **System.Diagnostics.Debug** и, в частности, его метод **Debug.WriteLine**. Соответственно все записи лога мы можем увидеть в окне **Output** в Visual Studio. Устанавливается методом **AddDebug()**. Стоит отметить, что данный способ работает только при запуске проекта в режиме отладки.

**EventSource**: на Windows введет логгирование в лог **ETW (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 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 var loggerFactory = LoggerFactory.Create(builder => builder.AddDebug());
5 ILogger logger = loggerFactory.CreateLogger<Program>();
6 app.Run(async (context) =>
7 {
8     logger.LogInformation($"Requested Path: {context.Request.Path}");
9     await context.Response.WriteAsync("Hello World!");
10 });
11
12 app.Run();
```



## 2.3. 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   "Logging": {
3     "LogLevel": {
4       "Default": "Information",
5       "Microsoft.AspNetCore": "Warning"
6     }
7   },
8   "AllowedHosts": "*"
9 }
```

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   "Logging": {
3     "LogLevel": {
4       "Default": "Information",
5     }
6   },
7   "AllowedHosts": "*"
8 }
```

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

```

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

```

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

```

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

```

То есть для провайдера **Debug** определено логгирование сообщения уровня **Debug** для всех категорий. Для провайдера **Console** определено два правила:

```

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

```

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 "LogLevel": {  
2   "Default": "Error"  
3 }
```

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

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

```
1 builder.AddFilter("System", LogLevel.Information)
```

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

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

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 "Logging": {  
2   "Debug": {  
3     "LogLevel": {  
4       "Microsoft": "Trace"  
5     }  
6   }  
}
```

Можно определять фильтры сразу и в коде С#, 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 var loggerFactory = LoggerFactory.Create(builder =>  
2 {  
3   builder.AddDebug();  
4   builder.AddFilter("System", LogLevel.Debug)  
5     .SetMinimumLevel(LogLevel.Debug); // Определение MinimumLevel  
6 }  
7 });
```



## Глобальная настройка логгирования

Для глобальной установки настроек логгирования у объекта **WebApplicationBuilder** определено свойство **Logging**, которое представляет тип **ILoggingBuilder**. 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 var builder = WebApplication.CreateBuilder();
2 builder.Logging.ClearProviders(); // удаляем все провайдеры
3 builder.Logging.AddConsole();    // добавляем провайдер для логгирования на консоль
4 var app = builder.Build();
```

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

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 указывает, доступен ли логгер для использования. Здесь можно задать 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 параметров 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 public class FileLoggerProvider : ILoggerProvider
2 {
3     string path;
4     public FileLoggerProvider(string path)
5     {
6         this.path = path;
7     }
8     public ILogger CreateLogger(string categoryName)
9     {
10         return new FileLogger(path);
11     }
12
13     public void Dispose() {}
14 }
```

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

**CreateLogger**: создает и возвращает объект логгера. Для создания логгера используется путь к файлу, который передается через конструктор.

**Dispose**: управляет освобождением ресурсов. В данном случае пустая реализация.

Теперь создадим вспомогательный класс **FileLoggerExtensions**:

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

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

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

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

