ASP NET CORE

**Загрузка файлов на сервер**

Рассмотрим, как загружать файлы на сервер в ASP.NET Core. Все загружаемые файлы в ASP.NET Core представлены типом IFormFile из пространства имен Microsoft.AspNetCore.Http. Соответственно для получения отправленного файла в контроллере необходимо использовать IFormFile. Затем с помощью методов IFormFile мы можем произвести различные манипуляции файлом - получит его свойства, сохранить, получить его поток и т.д. Некоторые его свойства и методы:

ContentType: тип файла

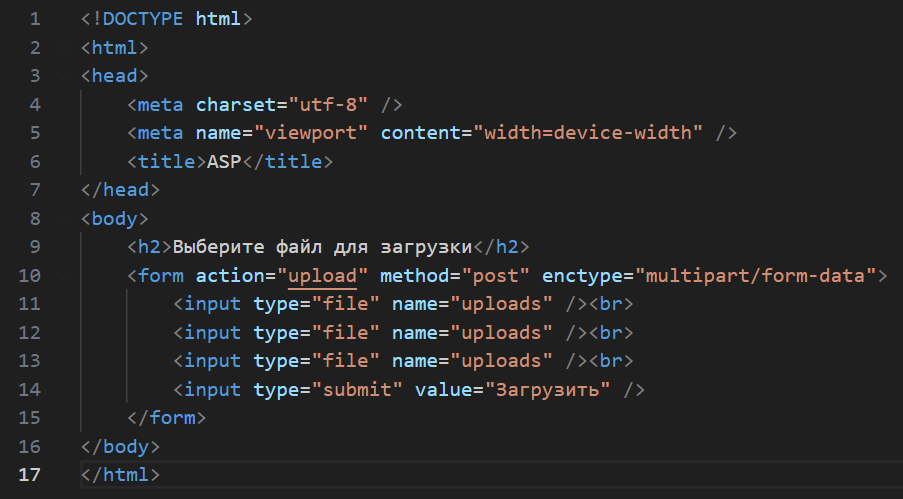
FileName: название файла

Length: размер файла

CopyTo/CopyToAsync: копирует файл в поток

OpenReadStream: открывает поток файла для чтения

Для тестирования данной возможности определим в проекте папку html, в которой создадим файл index.html



В данном случае форма содержит набор элементов с типом file, через которые можно выбрать файлы для загрузки. В данном случае на форме три таких элемента, но их может быть и меньше и больше. А благодаря установке атрибута формы enctype="multipart/form-data" браузер будет знать, что вместе с формой надо передать файлы.

Отправляться файлы будут в запросе типа POST на адрес "/upload".

Теперь в файле Program.cs определим код, который будет получать загружаемые файлы: (pr1)

Здесь если запрос приходит по адресу "/upload", а сам запрос представляет запрос типа POST, то приложение получает коллекцию загруженных файлов с помощью свойства Request.Form.Files, которое представляет тип IFormFileCollection:



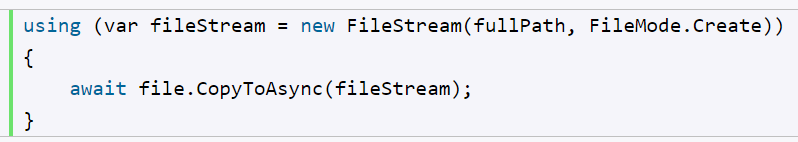
Далее определяем каталог для загружаемых файлов (предполагается, что файлы будут храниться в каталоге "uploads", которая располагается в папке приложения)



Если такой папки нет, то создаем ее. Затем перебираем всю коллекцию файлов.

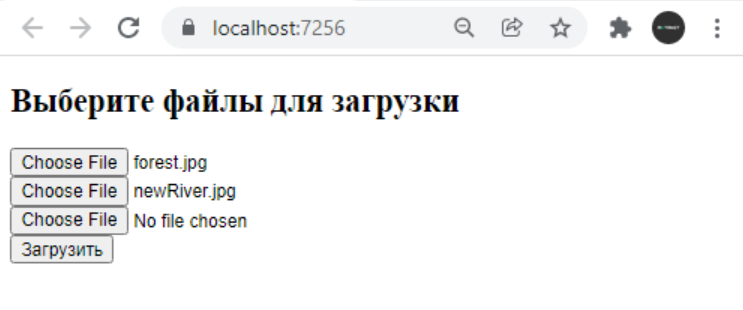


Каждый отдельный файл в этой коллекции представляет тип IFormFile. Для копирования файла в нужный каталог создается поток FileStream, в который записывается файл с помощью метода CopyToAsync.

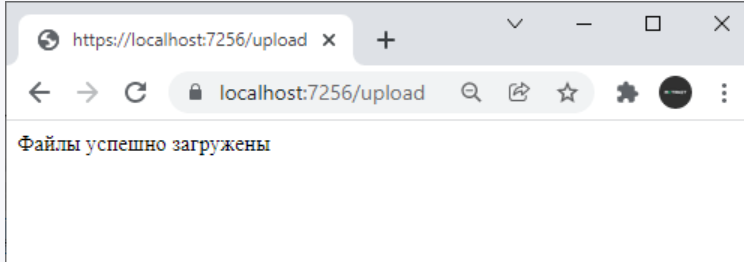


Если запрос идет по другому адресу и/или не представляет тип POST, то отправляем клиенту html-страницу index.html.

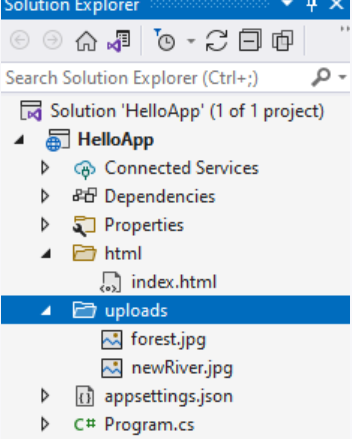
Обратимся к приложению и выберем файлы для загрузки:



И после успешной загрузки нам отобразиться соответствующее сообщение:



А в каталоге проекта будет создана папка uploads, в которой появятся загуженные файлы:



**Метод Use**

Метод расширения Use() добавляет компонент middleware, который позволяет передать обработку запроса далее следующим в конвейере компонентам. Он имеет следующие версии:

**public static IApplicationBuilder Use(this IApplicationBuilder app, Func<HttpContext, Func<Task>, Task> middleware);**

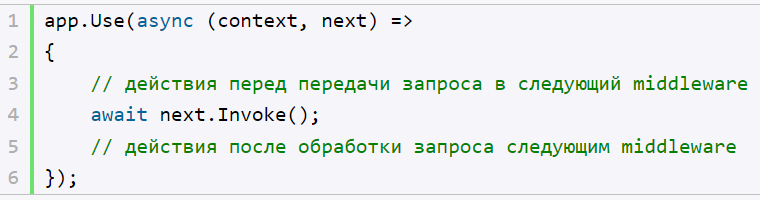
**public static IApplicationBuilder Use(this IApplicationBuilder app, Func<HttpContext, RequestDelegate, Task> middleware)**

Метод Use() реализован как метод расширения для типа IApplicationBuilder, соответственно мы можем вызвать данный метод у объекта WebApplication для добавления middleware в приложение. В обоих версиях метод Use принимает некоторое действие, которое имеет два параметра и возвращает объект Task.

Первый параметр делегата Func, который передается в метод Use(), представляет объект HttpContext. Этот объект позволяет получить данные запроса и управлять ответом клиенту.

Второй параметр делегата представляет другой делегат - Func<Task> или RequestDelegate. Этот делегат представляет следующий в конвейере компонент middleware, которому будет передаваться обработка запроса.

В общем случае применение метода Use() выглядит следующим образом:



Работа middleware разбивается на две части:

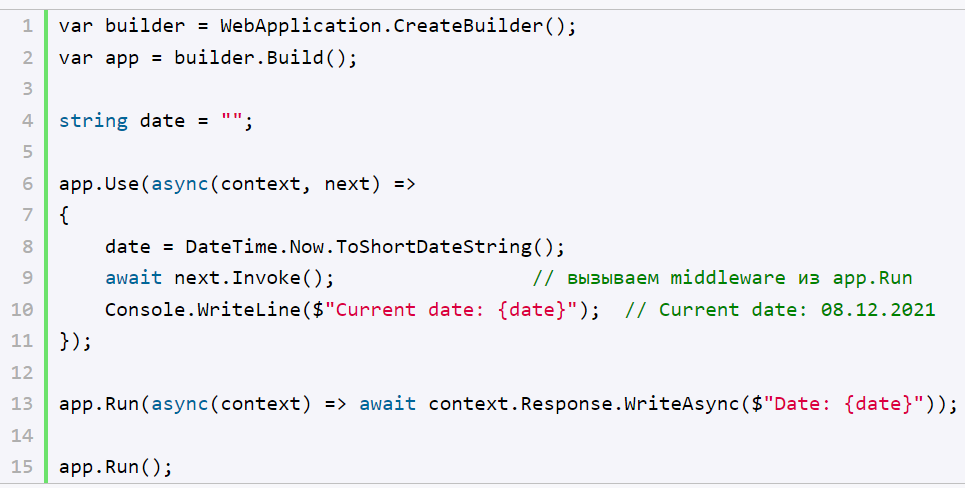
Middleware выполняет некоторую начальную обработку запроса до вызова await next.Invoke()

Затем вызывается метод next.Invoke(), который передает обработку запроса следующему компоненту в конвейере

Когда следующий в конвейере компонент закончил обработку запрос возвращается в обратно в текущий компонент, и выполняются действия, которые идут после вызова await next.Invoke(

Таким образом, middleware в методе Use выполняет действия до следующего в конвейере компонента и после него.

Рассмотрим метод Use() на примере: (pr2)



В данном случае мы используем перегрузку метода Use, которая в качестве параметров принимает контекст запроса - объект HttpContext и делегат Func<Task>, который представляет собой ссылку на следующий в конвейере компонент middleware.

Middleware в методе app.Use() реализует простейшую задачу - присваивает переменной date текущую дату в виде строки и затем передает обработку запроса следующим компонентам middleware в конвейере. То есть при вызове await next.Invoke() обработка запроса перейдет к тому компоненту, который установлен в методе app.Run(). В итоге обработка запроса будет выглядеть следующим образом:

1 Вызов компонента app.Use

2 Установка значения переменной date:



3 Вызов await next.Invoke(). Управление переходит следующему компоненту в конвейере - к app.Run.

4 В middleware из app.Run() отравляет клиенту текущую дату в качестве ответа с помощью метода context.Response.WriteAsync():

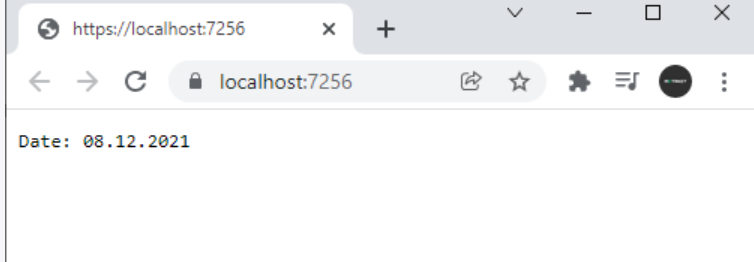


5 Метод app.Run закончил свою работу, и управление обработкой возвращается к middleware в методе app.Use. Начинает выполняться та часть кода, которая идет после await next.Invoke(). В этой части выполняется условное логгирование - на консоль выводится значение переменной date:

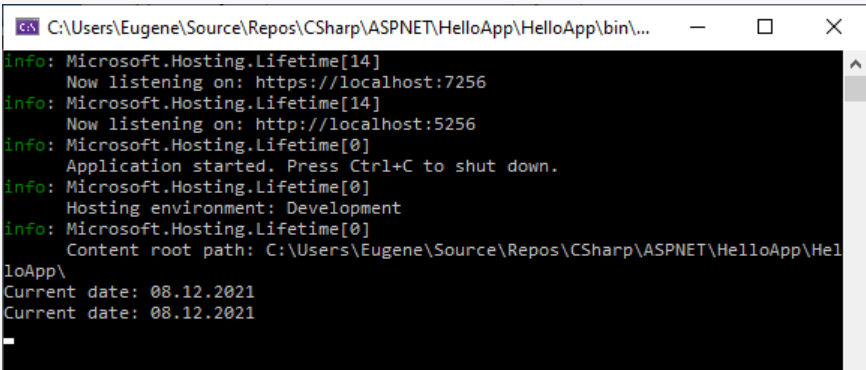


После этого обработка запроса завершена

В итоге в веб-браузере мы увидим следующее сообщение:



А в консоли запущенного приложения мы увидим значение переменной date, которое выводится в middleware из метода app.Use:



Отправка ответа

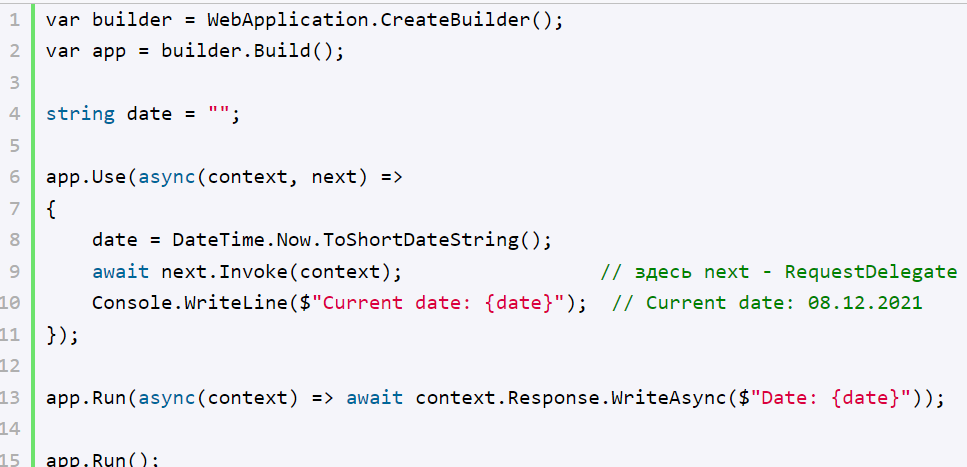
При использовании метода Use и передаче выполнения следующему делегату следует учитывать, что не рекомендуется вызывать метод next.Invoke после метода Response.WriteAsync(). Компонент middleware должен либо генерировать ответ с помощью Response.WriteAsync, либо вызывать следующий делегат посредством next.Invoke, но не выполнять оба этих действия одновременно. Так как согласно документации последующие изменения объекта Response могут привести к нарушению протокола, например, будет послано больше байт, чем указано в заголовке Content-Length, либо могут привести к нарушению тела ответа, например, футер страницы HTML запишется в CSS-файл.

**То есть к примеру следующая обработка запроса не рекомендуется:**



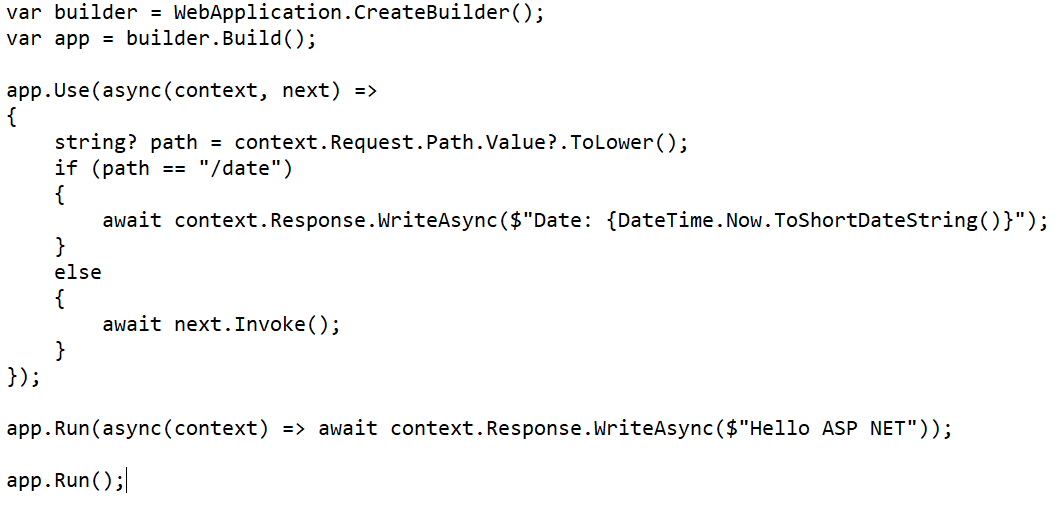
**Использование делегат RequestDelegate**

В примере выше использовалась версия метода Use(), которая использует делегат Func<Task>. Подобным образом можно использовать и другую версию, где используется делегат RequestDelegate. Единственное - при вызове делегата ( то есть фактически следующего в конвейере компонента) необходимо передавать делегату объект HttpContext:

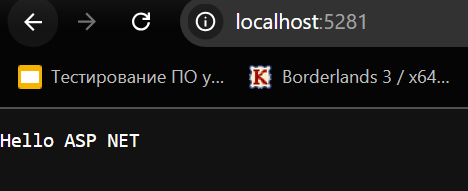


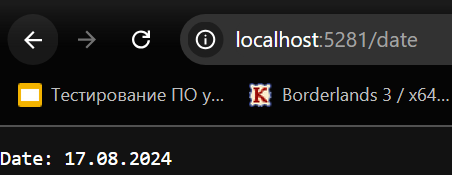
Терминальный компонент middleware

Middleware в методе Use() необязательно должен вызывать к следующему в конвейере компоненту. Вместо этого он может завершить обработку запроса. В этом случае он может выступать в роли такого же терминального компонента middleware, а и компоненты из метода Run(). Например: (pr3)

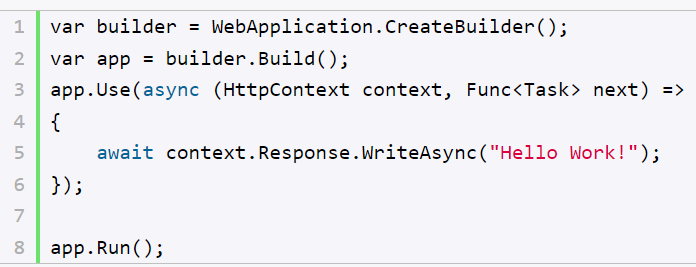


Здесь middleware в app.Use проверяет запрошенный адрес - если он содержит "/date", то клиенту отправляется текущая дата. Иначае обработка запроса передается дальше в app.Run.





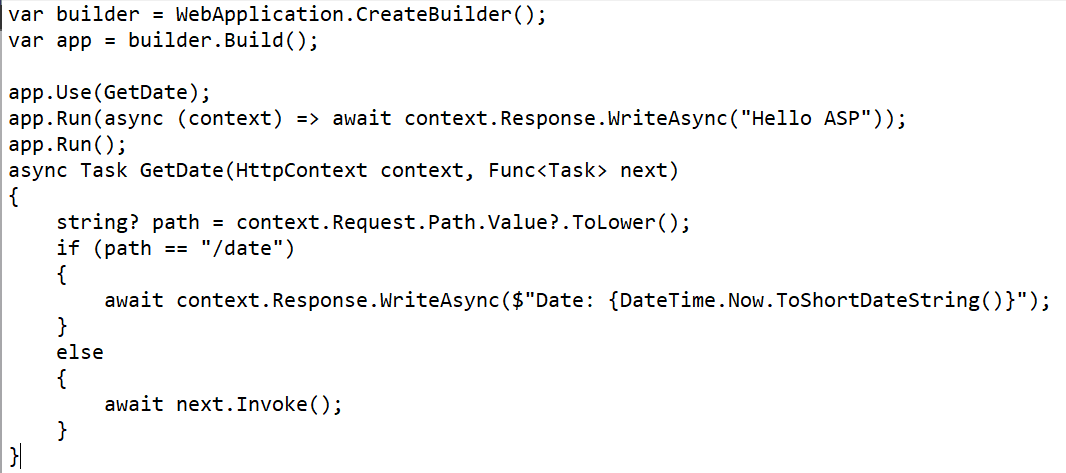
Причем в принципе мы можем использовать компонент в app.Use как единственный и соответственно терминальный:



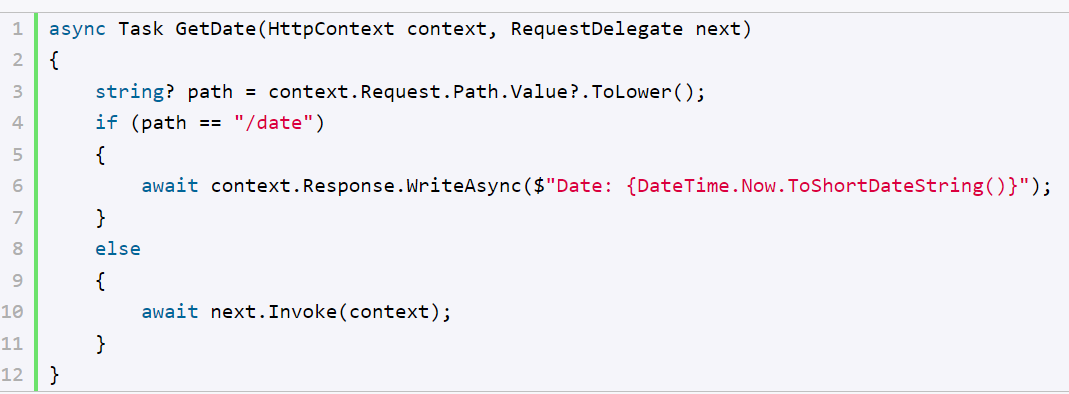
Однако в данном случае для большей производительности лучше использовать app.Run(), если нам надо определить лишь один компонент, который в принципе не передает запрос дальше по конвейеру.

**Вынесение компонентов в методы**

Также можно вынести все inline-компоненты в отдельные методы(pr3)



Подобным образом можно использовать и другую версию метода Use, в которой используется делегат RequestDelegate(pr3)



**Создание ветки конвейера. UseWhen и MapWhen**

UseWhen

Метод UseWhen() на основании некоторого условия позволяет создать ответвление конвейера при обработке запроса:

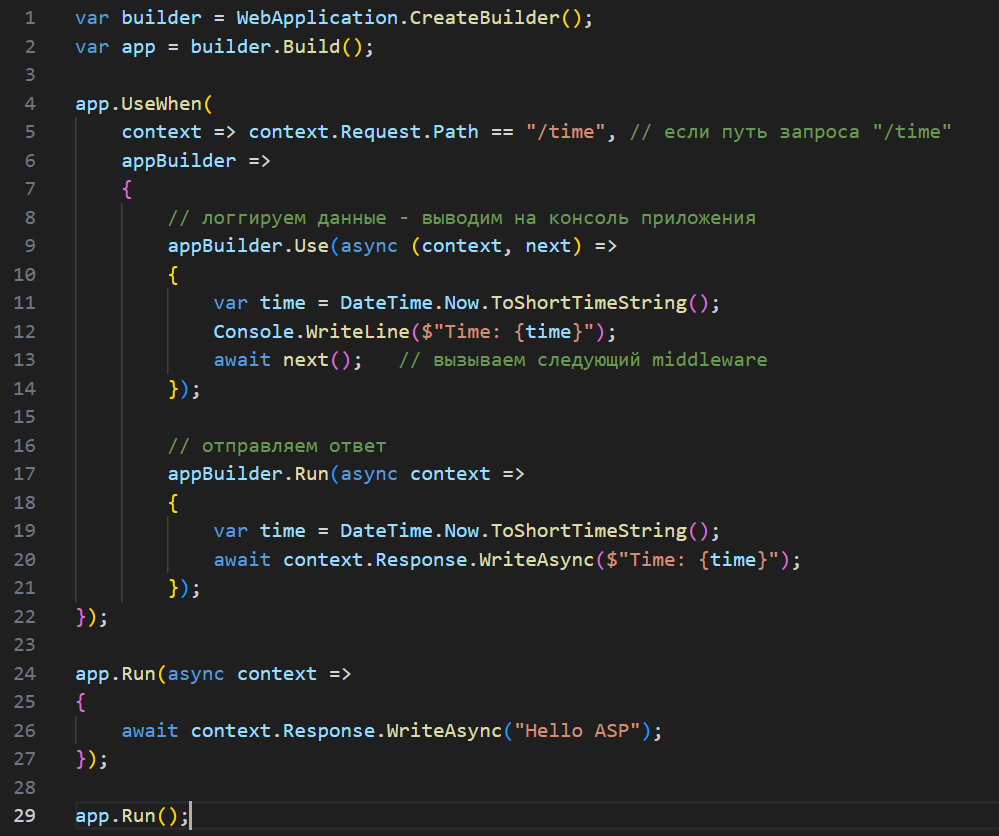
public static IApplicationBuilder UseWhen (this IApplicationBuilder app, Func<HttpContext,bool> predicate, Action<IApplicationBuilder> configuration);

Как и Use(), метод UseWhen() реализован как метод расширения для типа IApplicationBuilder.

В качестве параметра он принимает делегат Func>HttpContext,bool> - некоторое условие, которому должен соответствовать запрос. В этот делегат передается объект HttpContext. А возвращаемым типом должен быть тип bool - если запрос соответствует условию, то возвращается true, иначе возвращаеся false.

Последний параметр метода - делегат Action<IApplicationBuilder> представляет некоторые действия над объектом IApplicationBuilder, который передается в делегат в качестве параметра.

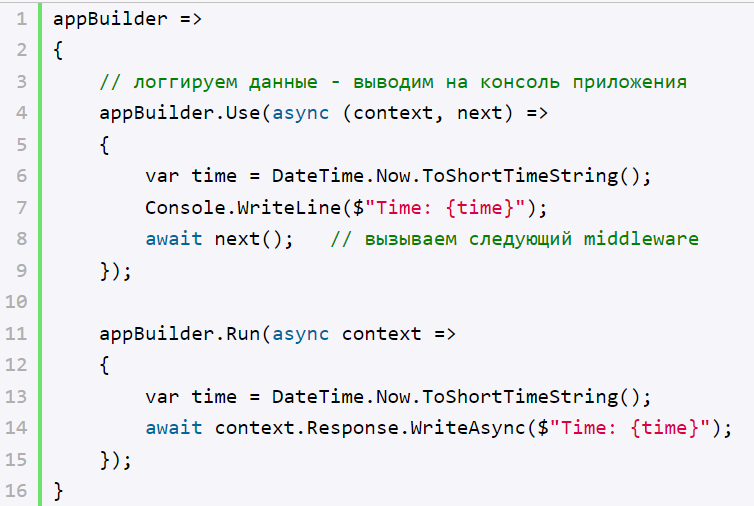
Рассмотрим небольшой пример: (pr4)



В данном случае метод app.UseWhen() в качестве первого параметра получает следующее условие:

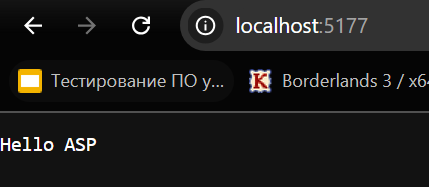


Второй параметр определяет действие, в котором создается ответвление конвейера:

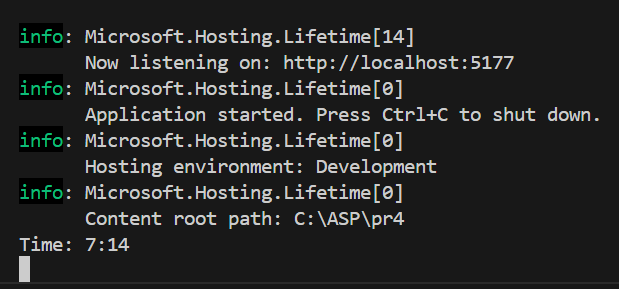


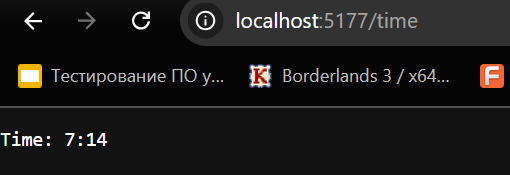
В данном действии в конвейер обработки запроса встраиваются два middleware - с помощью методов Use() и Run(). В первом middleware логгируем это время на консоль приложения. Во втором - терминальном компоненте middleware отправляем информацию о времени в ответ клиенту.

Если мы обращаемся к приложению по пути, который отличается от "/time", то условие в методе UseWhen() ложно, поэтому ответвления конвейера не выполняется. И выполняется middleware из метода app.Run():



Однако если мы обращаемся по пути "/time", то условие в методе app.UseWhen() будет истинно. Соответственно будет выполняться ответвление конвейера, который будет обрабатывать запрос. В итоге на консоль приложения, а также в браузере будет выводиться текущее время.



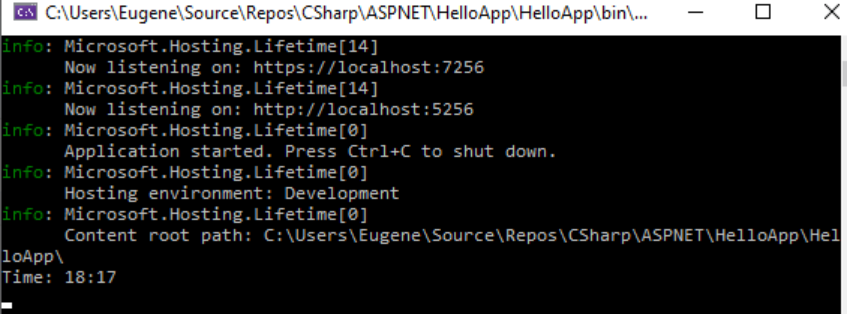


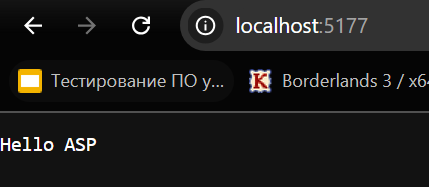
Стоит отметить, что создание ветки происходит один раз при запуске приложения. Например, в примере выше мы видим, что получение времени производится в обоих middleware во встраиваемой ветки. Но что будет, если вынести получение времени во вне и не дублировать в каждом middleware, например, следующим образом (pr5):

В этом случае время будет устанавливаться один раз - при запуске приложения и создании ветки в конвейер. Соответственно вне зависимости от того, сколько раз мы будем обращаться к приложению по пути "/time", мы будем получать одно и то же время.

В примере выше ветка конвейера завершалась терминальным компонентом, поэтому остальные действия в основной части конвейера не выполнялись. Однако мы можем также передать запрос на обработку из ветки в основной поток конвейера: (pr5 – primer 5.1)

В данном случае, если запрос идет по пути "/time", сначала срабатывает ветка конвейера с компонентом, который логгирует время на консоль. А затем выполняется компоненте из app.Run(), который отправляет сообщение "Hello ASP".





Для большей читабельности также можно было бы вынести действия по созданию ветки конвейера в отдельный метод: (pr5 – primer 5.2)

**MapWhen**

Метод MapWhen(), как и метод UseWhen(), на основании некоторого условия позволяет создать ответвление конвейера:

public static IApplicationBuilder MapWhen (this IApplicationBuilder app, Func<HttpContext,bool> predicate, Action<IApplicationBuilder> configuration);

Метод MapWhen() также реализован как метод расширения для типа IApplicationBuilder, принимает те же параметры, что и UseWhen(), и работает во многом аналогичным образом: (pr6)

Здесь опять же, если запрошен путь "/time", то срабатывает ветка конвейера, созданная методом app.MapWhen(), в которой клиенту отправляется текущее время. Если путь запроса другой, то срабатывается основной поток конвейера, в котором отправляется сообщение "Hello ASP".