Распределенные информационно-аналитические системы

Практическое занятие № 2. «Основы в ASP.NET Core. Часть 2»

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

Учебные вопросы:

1. Создание и запуск приложения. WebApplication и WebApplicationBuilder.

2. Конвейер обработки запроса и middleware.

3. Метод Run и определение терминального middleware.

1. Создание и запуск приложения.

WebApplication и WebApplicationBuilder

В центре приложения ASP.NET находится класс **WebApplication**. Например, если мы возьмем проект ASP.NET по типу ASP.NET Core Empty, то в файле Program.cs мы встретим следующий код:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();
```

Переменная app в данном коде как раз представляет объект **WebApplication**. Однако для создания этого объекта необходим другой объект – **WebApplicationBuilder**, который в данном коде представлен переменной `builder`.

Класс WebApplicationBuilder

Создание приложения по умолчанию фактически начинается с класса **WebApplicationBuilder**. Исходный код доступен по адресу [WebApplicationBuilder.cs]

Для его создания объекта этого класса вызывается статический метод **WebApplication.CreateBuilder()**:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder();
```

Для инициализации объекта **WebApplicationBuilder** в этот метод могут передаваться аргументы командной строки, указанные при запуске приложения (доступны через неявно определенный параметр args):

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args)
```

Либо можно передавать объект **WebApplicationOption**:

```
1  WebApplicationOptions options = new() { Args = args };
2  WebApplicationBuilder builder = WebApplication.CreateBuilder(optic
```

Кроме создания объекта **WebApplication** класс **WebApplicationBuilder** выполняет еще ряд задач, среди которых можно выделить следующие:

Установка конфигурации приложения.

Добавление сервисов.

Настройка логгирования в приложении.

Установка окружения приложения.

Конфигурация объектов IHostBuilder и IWebHostBuilder, которые применяются для создания хоста приложения.

Для реализации этих задач в классе **WebApplicationBuilder** определены следующие **свойства**:

Configuration: представляет объект **ConfigurationManager**, который применяется для добавления конфигурации к приложению.

Environment: предоставляет информацию об окружении, в котором запущено приложение.

Host: объект **IHostBuilder**, который применяется для настройки хоста.

Logging: позволяет определить настройки логгирования в приложении.

Services: представляет коллекцию сервисов и позволяет добавлять сервисы в приложение.

WebHost: объект **IWebHostBuilder**, который позволяет настроить отдельные настройки сервера.

Класс WebApplication

Метод **build()** класса **WebApplicationBuilder** создает объект **WebApplication**:

```
1 WebApplicationBuilder builder = WebApplication.CreateBuilder();
2 WebApplication app = builder.Build();
```

Класс **WebApplication** применяется для управления обработкой запроса, установки маршрутов, получения сервисов и т.д.

Исходный код класса можно найти на Github по адресу WebApplication.cs.

Класс WebApplication применяет три интерфейса:

IHost: применяется для запуска и остановки хоста, который прослушивает входящие запросы.

IApplicationBuilder: применяется для установки компонентов, которые участвуют в обработке запроса.

IEndpointRouteBuilder: применяется для установки маршрутов, которые сопоставляются с запросами.

Для получения доступа к функциональности приложения можно использовать **свойства** класса **WebApplication**:

Configuration: представляет конфигурацию приложения в виде объекта **IConfiguration**.

Environment: представляет окружение приложения в виде **IWebHostEnvironment**.

Lifetime: позволяет получать уведомления о событиях жизненного цикла приложения.

Logger: представляет логгер приложения по умолчанию.

Services: представляет сервисы приложения.

Urls: представляет набор адресов, которые использует сервер.

Для управления хостом класс **WebApplication** определяет следующие **методы**:

Run(): запускает приложение.

RunAsync(): асинхронно запускает приложение.

Start(): запускает приложение.

StartAsync(): запускает приложение.

StopAsync(): останавливает приложение.

Таким образом, после вызова метода Run/Start/RunAsync/StartAsync приложение будет запущено, и мы сможем к нему обращаться:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder();
WebApplication app = builder.Build();
app.Run();
```

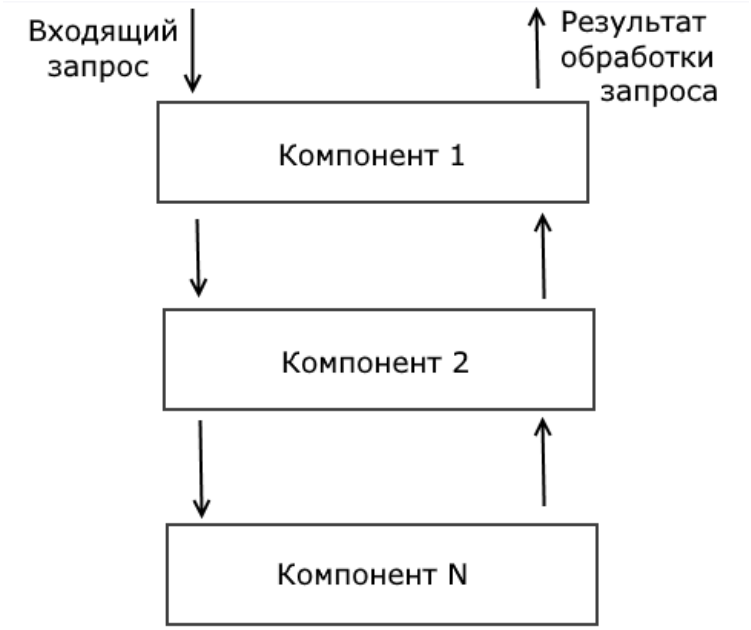
При необходимости с помощью метода **StopAsync()** можно программным способом завершить выполнение приложения:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder();

WebApplication app = builder.Build();

app.MapGet("/", () => "Hello World!");

await app.StartAsync();
await Task.Delay(10000);
await app.StopAsync();  // через 10 секунд завершаем выполнение приложения
```

2. Конвейер обработки запроса и middleware

Одна из основных задач приложения – это обработка входящих запросов. Обработка запроса в ASP.NET Core устроена по принципу конвейера, который состоит из компонентов. Подобные компоненты еще называются **middleware** (в русском языке до сих пор нет адекватного термина для подобным компонента, поэтому далее они именуются преимущественно как "компоненты middleware" или просто middleware).

При получении запроса сначала данные запроса получает первый компонент в конвейере. После обработки запроса компонент **middleware** он может закончить обработку запроса – такой компонент еще называется **терминальным компонентом** (**terminal middleware**). Либо он может передать данные запроса для обработки далее по конвейеру – следующему в конвейере компоненту и так далее. После обработки запроса последним компонентом, данные запроса возвращаются к предыдущему компоненту. Схематически это можно отобразить так:



Конвейер обработки запроса в ASP.NET Core

Компоненты middleware встраиваются с помощью методов расширений Run, Map и Use интерфейса **IApplicationBuilder**. Класс **WebApplication** реализует данный интерфейс и поэтому позволяет добавлять компоненты **middleware** с помощью данных методов.

Каждый компонент middleware может быть определен как метод (встроенный inline компонент), либо может быть вынесен в отдельный класс.

Для создания компонентов middleware используется делегат **RequestDelegate**, который выполняет некоторое действие и принимает контекст запроса – объект **HttpContext**:

```
1 public delegate Task RequestDelegate(HttpContext context);
```

При получении запроса сервер формирует на его основе объект **HttpContext**, которые содержит всю необходимую информацию о запросе. Эта информация посредством объекта **HttpContext** передается всем компонентам middleware в приложении.

Рассмотрим, какую информацию мы можем получить из **HttpContext**. Для этого пройдемся по его **свойствам**:

Connection: представляет информацию о подключении, которое установлено для данного запроса.

Features: получает коллекцию HTTP-функциональностей, которые доступны для этого запроса.

Items: получает или устанавливает коллекцию пар ключ-значение для хранения некоторых данных для текущего запроса.

Request: возвращает объект **HttpRequest**, который хранит информацию о текущем запросе.

RequestAborted: уведомляет приложение, когда подключение прерывается, и соответственно обработка запроса должна быть отменена.

RequestServices: получает или устанавливает объект **IServiceProvider**, который предоставляет доступ к контейнеру сервисов запроса.

Response: возвращает объект **HttpResponse**, который позволяет управлять ответом клиенту.

Session: хранит данные сессии для текущего запроса.

TraceIdentifier: представляет уникальный идентификатор запроса для логов трассировки.

User: представляет пользователя, ассоциированного с этим запросом.

WebSockets: возвращает объект для управления подключениями WebSocket для данного запроса.

Используя эти свойства, мы можем в компоненте middleware получить если не все, то большую часть необходимых данных о запросе и отправить обратно клиенту некоторый ответ.

Встроенные компоненты middleware

Стоит отметить, что ASP.NET Core уже по умолчанию предоставляет ряд встроенных компонентов **middleware** для часто встречающихся **задач**:

Authentication: предоставляет поддержку аутентификации.

Authorization: предоставляет поддержку авторизации.

Cookie Policy: отслеживает согласие пользователя на хранение связанной с ним информации в куках.

CORS: обеспечивает поддержку кроссдоменных запросов.

DeveloperExceptionPage: генерирует веб-страницу с информацией об ошибке при работе в режиме разработки.

Diagnostics: набор middleware, который предоставляет страницы статусных кодов, функционал обработки исключений, страницу исключений разработчика.

Forwarded Headers: перенаправляет заголовки запроса.

Health Check: проверяет работоспособность приложения asp.net core.

Header Propagation: обеспечивает передачу заголовков из HTTP-запроса.

HTTP Logging: логгирует информацию о входящих запросах и генерируемых ответах.

HTTP Method Override: позволяет входящему POST-запросу переопределить метод.

HTTPS Redirection: перенаправляет все запросы HTTP на HTTPS.

HTTP Strict Transport Security (HSTS): для улучшения безопасности приложения добавляет специальный заголовок ответа.

MVC: обеспечивает функционал фреймворка MVC.

OWIN: обеспечивает взаимодействие с приложениями, серверами и компонентами, построенными на основе спецификации OWIN.

Request Localization: обеспечивает поддержку локализации.

Response Caching: позволяет кэшировать результаты запросов.

Response Compression: обеспечивает сжатие ответа клиенту.

URL Rewrite: предоставляет функциональность URL Rewriting.

Endpoint Routing: предоставляет механизм маршрутизации.

Session: предоставляет поддержку сессий.

SPA: обрабатывает все запросы, возвращая страницу по умолчанию для SPA-приложения (одностраничного приложения).

Static Files: предоставляет поддержку обработки статических файлов.

WebSockets: добавляет поддержку протокола WebSockets.

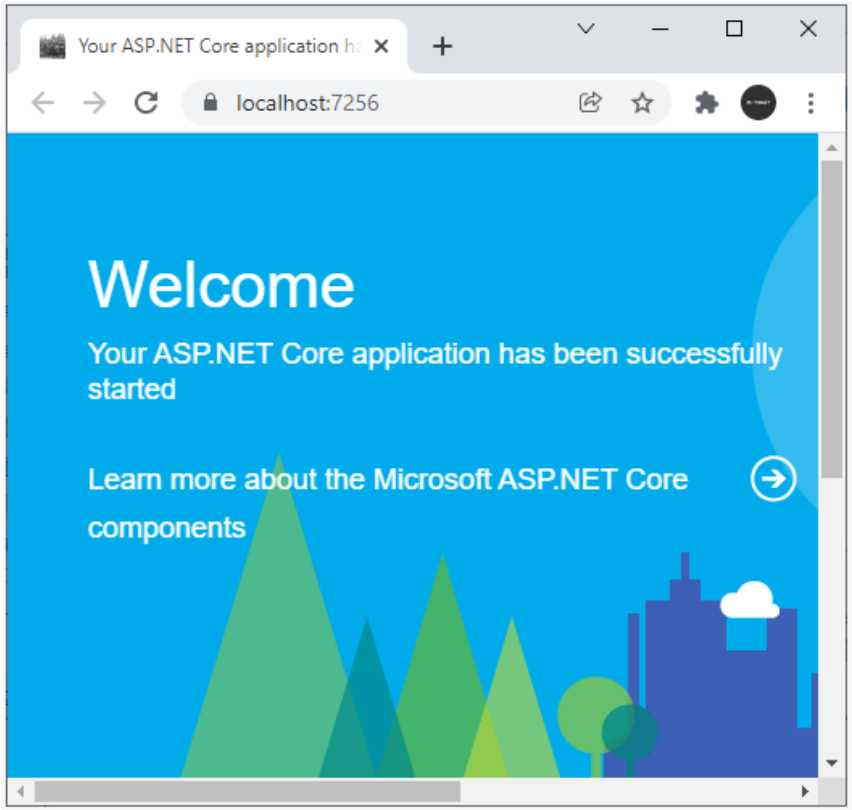
W3CLogging: генерирует логи доступа в соответствии с форматом W3C Extended Log File Format.

Для встраивания этих компонентов в конвейер обработки запроса для интерфейса **IApplicationBuilder** определены методы расширения типа UseXXX.

Например, фреймворк ASP.NET Core по умолчанию предоставляет такой **middleware** как **WelcomePageMiddleware**, который отправляет клиенту некоторую стандартную веб-страницу. Для подключения этого компонента в конвейер запроса применяется метод расширения **UseWelcomePage()**:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.UseWelcomePage();   // подключение WelcomePageMiddleware
app.Run();
```

И при выполнении этого приложения браузер представит нашему взору следующую красочную страницу:



Встроенные middleware в ASP.NET Core и C# и WelcomePageMiddleware

3. Метод Run и определение терминального middleware

Самый простой способ добавления middleware в конвейер обработки запроса в ASP.NET Core представляет метод **Run()**, который определен как метод расширения для интерфейса **IApplicationBuilder** (соответственно его поддерживает и класс **WebApplication**):

```
IApplicationBuilder.Run(RequestDelegate handler)
```

Метод **Run** добавляет терминальный компонент – такой компонент, который завершает обработку запроса. Поэтому соответственно он не вызывает никакие другие компоненты и обработку запроса дальше – следующим в конвейере компонентам не передает. Поэтому данный метод следует вызывать в самом конце построения конвейера обработки запроса. До него же могут быть помещены другие методы, которые добавляют компоненты middleware.

В качестве параметра метод **Run** принимает делегат **RequestDelegate**. Этот делегат имеет следующее определение:

```
public delegate Task RequestDelegate(HttpContext context);
```

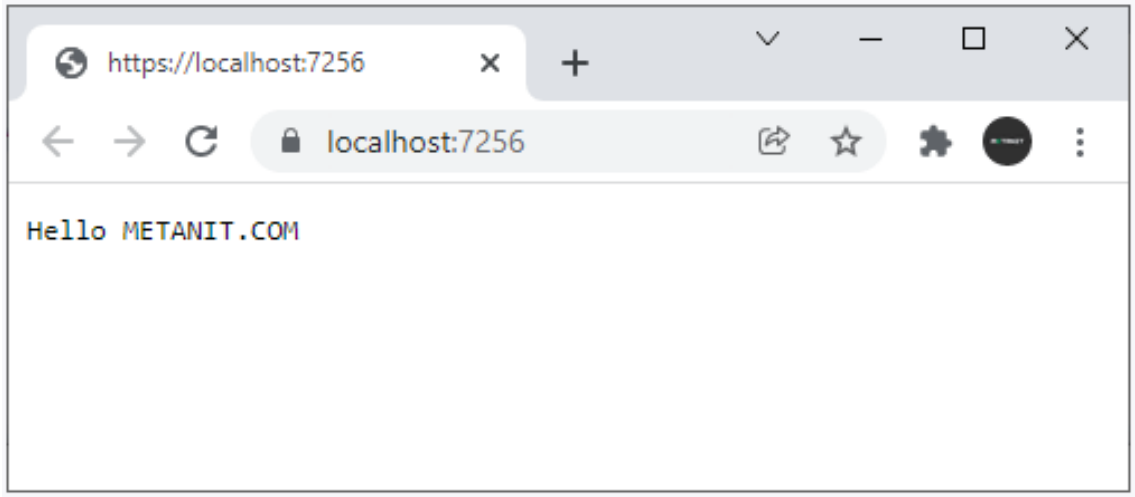
Он принимает в качестве параметра контекст запроса **HttpContext** и возвращает объект **Task**.

Используем этот метод для определения простейшего компонента:

```
var builder = WebApplication.CreateBuilder();

var app = builder.Build();

app.Run(async (context) => await context.Response.WriteAsync("Hello METANIT.COM"));
app.Run();
```

Здесь для делегата **RequestDelegate** передается лямбда-выражение, параметр которого – **HttpContext** можно использовать для отправки ответа. В частности, метод **context.Response.WriteAsync()** позволяет отправить клиенту некоторый ответ – в данном случае отправляется простая строка.

После запуска проекта будет запущено приложение откроется браузер, который выполнит запрос к приложению и получит обратно строку "Hello METANIT.COM".



Установка middleware в ASP.NET Core с помощью метода Run

Здесь следует сделать пару замечаний. Прежде всего, не стоит путать метод **Run()**, который определен в классе **WebApplication** и который запускает приложение, и метод расширения **Run()**, который встраивает компонент middleware. Это два разных метода, которые выполняют разные задачи. И, как видно из кода выше, вызываются оба этих метода.

Второй момент – метод **Run()**, который запускает приложение, вызывается после добавления компонента **middleware**. И мы НЕ можем написать так:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run();  // приложение запущено
// в этой строке уже нет смысла
app.Run(async (context) => await context.Response.WriteAsync("Hello METANIT.COM"));
```

При необходимости естественно мы можем вынести код **middleware** в отдельный метод:

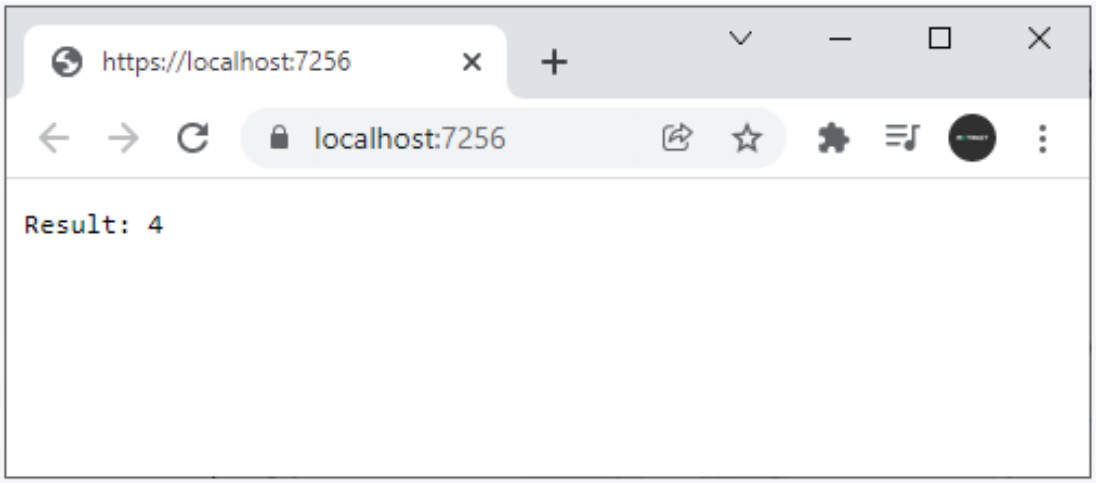
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(HandleRequst);
app.Run();

async Task HandleRequst(HttpContext context)
{
    await context.Response.WriteAsync("Hello METANIT.COM 2");
}
```

Жизненный цикл middleware

Компоненты **middleware** создаются один раз и существуют в течение всего жизненного цикла приложения. То есть для последующей обработки запросов используются одни и те же компоненты. Например, определим в файле **Program.cs** следующий код:

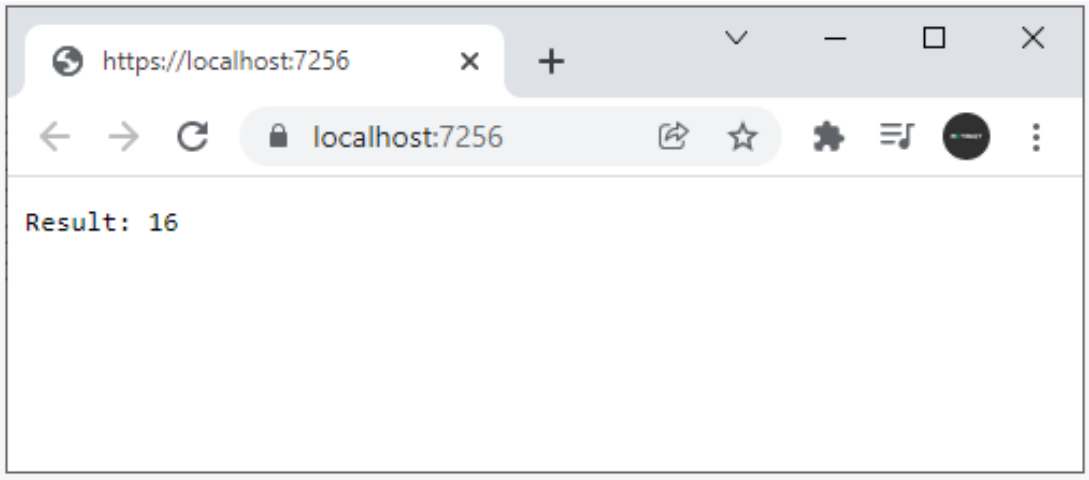
```
var builder = WebApplication.CreateBuilder();

var app = builder.Build();

int x = 2;
app.Run(async (context) =>
{
    x = x * 2;  //  2 * 2 = 4
    await context.Response.WriteAsync($"Result: {x}");
});
app.Run();
```

При запуске приложения мы естественно ожидаем, что браузер выведет число 4 в качестве результата:



Жизненный цикл middleware в ASP.NET Core и C#

Однако при последующих запросах мы увидим, что результат переменной x не равен 4.



Жизненный цикл приложения в ASP.NET Core и C#

Также стоит отметить, что браузер Google Chrome может посылать два запроса – один собственно к приложению, а другой – к файлу иконки favicon.ico, поэтому в Google Chrome результат может отличаться не 2 раза, а гораздо больше.