Конвейер обработки запроса и middleware

Обработка запроса в ASP.NET Core устроена по принципу конвейера. Сначала данные запроса получает первый компонент в конвейере. После обработки он передает данные HTTP-запроса второму компоненту и так далее. Эти компоненты конвейера, которые отвечают за обработку запроса, называются **middleware**. В ASP.NET Core для подключения компонентов middleware используется метод Configure из класса Startup.

Компонент middleware может либо передать запрос далее следующему в конвейере компоненту, либо выполнить обработку и закончить работу конвейера. Также компонент middleware в конвейере может выполнять обработку запроса как до, так и после следующего в конвейере компонента.

Компоненты middleware конфигурируются с помощью методов расширений `Run`, `Map` и `Use` объекта **IApplicationBuilder**, который передается в метод `Configure()` класса Startup. Каждый компонент может быть определен как анонимный 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:

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

Рассмотрим метод Configure из класса Startup стандартного проекта по типу Empty:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // если проект в процессе разработки
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
                endpoints.MapGet("/", async context =>
                {
                        await context.Response.WriteAsync("Hello World!");
```

```
                });
        });
}
```

Все вызовы типа `app.UseXXX` как раз и представляют собой добавление компонентов middleware для обработки запроса. То есть у нас получается примерно следующий конвейер обработки:

1. Компонент обработки ошибок - Diagnostics. Добавляется через `app.UseDeveloperExceptionPage()`
2. Компонент маршрутизации - EndpointRoutingMiddleware. Добавляется через `app.UseRouting()`
3. Компонент EndpointMiddleware, который отправляет ответ, если запрос пришел по маршруту "/" (то есть пользователь обратился к корню веб-приложения). Добавляется через метод `app.UseEndpoints()`

При этом порядок определения компонентов играет большую роль. Например, в этом методе сначала добавляются компоненты для встраивания механизма маршрутизации `app.UseRouting()`, а потом только компонент для обработки запроса по определенному маршруту `app.UseEndpoints()`. Если мы изменим порядок, то приложение нормально работать не будет:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // если проект в процессе разработки
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
        app.UseEndpoints(endpoints =>
        {
                endpoints.MapGet("/", async context =>
                {
                        await context.Response.WriteAsync("Hello World!");
                });
        });
        app.UseRouting();
}
```

В этом случае мы получим исключение, так как для работы компонента EndpointMiddleware необходимо, чтобы в приложении уже была определена система маршрутизации с помощью компонента EndpointRoutingMiddleware. Поэтому в конвейер нужно сначала добавлять EndpointRoutingMiddleware (app.UseRouting()) и только потом EndpointMiddleware (app.UseEndpoints()).

По умолчанию ASP.NET Core предоставляет следующие встроенные компоненты middleware:

- `Authentication`: предоставляет поддержку аутентификации
- `Cookie Policy`: отслеживает согласие пользователя на хранение связанной с ним информации в куках
- `CORS`: обеспечивает поддержку кроссдоменных запросов
- `Diagnostics`: предоставляет страницы статусных кодов, функционал обработки исключений, страницу исключений разработчика
- `Forwarded Headers`: перенаправляет зголовки запроса
- `Health Check`: проверяет работоспособность приложения asp.net core
- `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
- `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
- `Static Files`: предоставляет поддержку обработки статических файлов

- `WebSockets`: добавляет поддержку протокола WebSockets

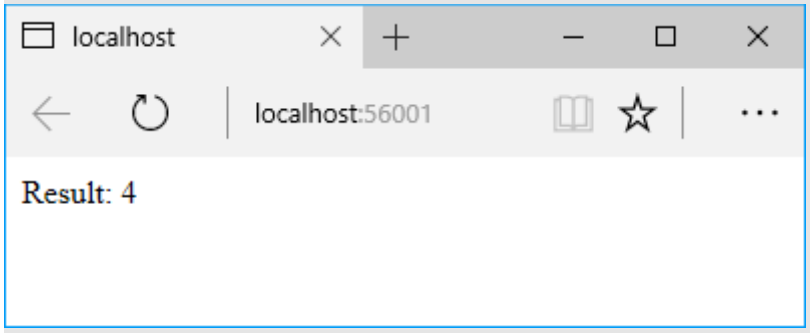
Жизненный цикл middleware

Метод Configure выполняется один раз при создании объекта класса Startup, и компоненты middleware создаются один раз и живут в течение всего жизненного цикла приложения. То есть для последующей обработки запросов используются одни и те же компоненты. Например, определим следующий класс Startup:

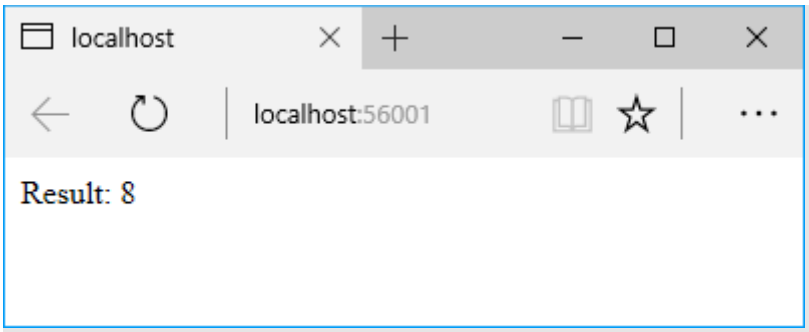
```
public class Startup
{
        public void ConfigureServices(IServiceCollection services)
        {
        }

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

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



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



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.