Класс Startup

Класс **Startup** является входной точкой в приложение ASP.NET Core. Этот класс производит конфигурацию приложения, настраивает сервисы, которые приложение будет использовать, устанавливает компоненты для обработки запроса или middleware.

Если мы обратимся к файлу Program.cs, то там есть такие строки:

```
public static void Main(string[] args)
{
        CreateHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                        webBuilder.UseStartup<Startup>();
                });
```

Метод `webBuilder.UseStartup<Startup>()` устанавливает класс Startup в качестве стартового. И при запуске приложения среда ASP.NET будет искать в сборке приложения класс с именем Startup и загружать его.

Однако в принципе необязательно, что класс назывался именно Startup. Так мы можем изменить соответствующий вызов в файле Program.cs на следующий:

```
webBuilder.UseStartup<Proccessor>()
```

Теперь среда будет искать при запуске приложения класс Proccessor. И в этом случае нам надо будет определить в проекте класс с именем Proccessor, который будет аналогичен файлу Startup.

Класс Startup должен определять метод **Configure()**, и также опционально в Startup можно определить конструктор класса и метод **ConfigureServices()**.

При запуске приложения сначала срабатывает конструктор, затем метод `ConfigureServices()` и в конце метод `Configure()`. Эти методы вызываются средой выполнения ASP.NET.

В проекте ASP.NET Core по шаблону Empty класс Startup выглядит следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace HelloApp
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseRouting();

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

Метод ConfigureServices

Необязательный метод `ConfigureServices()` регистрирует сервисы, которые используются приложением. В качестве параметра он принимает объект **IServiceCollection**, который и представляет коллекцию сервисов в приложении. С помощью методов расширений этого объекта производится конфигурация приложения для использования сервисов. Все методы имеют форму `Add[название_сервиса]`.

В проекте по типу Empty данный метод не выполняет каких-либо действий:

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

А, к примеру, в проекте по типу **Web Application (Model-View-Controller)** данный метод имеет следующее определение

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
}
```

Метод `services.AddControllersWithViews();` добавляет в коллекцию сервисов сервисы, которые необходимы для работы контроллеров MVC. После добавления в коллекцию сервисов добавленные сервисы становятся доступными для приложения. Как правило, встроенные методы, которые добавляют встроенные сервисы, начинаются с префикса `Add`, например, `AddControllersWithViews()`.

Метод Configure

Метод Configure устанавливает, как приложение будет обрабатывать запрос. Этот метод является обязательным. Для установки компонентов, которые обрабатывают запрос, используются методы объекта **IApplicationBuilder**. Объект IApplicationBuilder является обязательным параметром для метода Configure.

Кроме того, метод нередко принимает еще один необязательный параметр - объект **IWebHostEnvironment**, который позволяет получить информацию о среде, в которой запускается приложение, и взаимодействовать с ней.

Но в принципе в метод Configure в качестве параметра может передаваться любой сервис, который зарегистрирован в методе ConfigureServices или который регистрируется для приложения

по умолчанию (например, IWebHostEnvironment).

Метод `Configure()` в проекте по типу Empty непосредственно обрабатывает запрос:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
        // если приложение в процессе разработки
    if (env.IsDevelopment())
    {
                // то выводим информацию об ошибке, при наличии ошибки
        app.UseDeveloperExceptionPage();
    }
        // добавляем возможности маршрутизации
        app.UseRouting();

        // устанавливаем адреса, которые будут обрабатываться
        app.UseEndpoints(endpoints =>
        {
                // обработка запроса - получаем констекст запроса в виде
объекта context
                endpoints.MapGet("/", async context =>
                {
                        // отправка ответа в виде строки "Hello World!"
                        await context.Response.WriteAsync("Hello World!");
                });
        });
}
```

Разберем по шагам, что делает этот метод:

1. Выражение `if (env.IsDevelopment())` проверяет, находится ли приложение в состоянии/статусе разработки. Что это значит? Для проекта можно указать, например, через настройки, что он находится в состоянии разработки. Вообще условно есть три состояния или стадии проекта: в состоянии разработки (Development), в состоянии подготовки к развертыванию (Staging) и в состоянии полноценного использования (Production), когда он уже развернут на каком-нибудь сервере, и пользователи могут к нему обращаться. По умолчанию Visual Studio устанавливает для проекта состояние разработки. И данное выражение как раз проверяет состояние.

Если проект находится в состоянии разработки, то, возможно, мы захотим применять некоторые действия, которые не нужны, когда приложение уже развернуто. Так, по умолчанию вызывается метод

```
app.UseDeveloperExceptionPage();
```

который выводит подробные сообщения об ошибках. Подобные сообщения нежелательны и могут раскрывать некоторые чувствительные данные, когда приложение уже развернуто на сервере и с ним могут работать пользователи, поэтому они подобные сообщения по умолчанию выводятся только в состоянии разработки.

2. Вызов `app.UseRouting()` добавляет некоторые возможности маршрутизации, благодаря чему приложение может соотносить запросы с определенными маршрутами.

3. Далее идет вызов `app.UseEndpoints(endpoints =>`, который позволяет определить маршруты, которые будут обрабатываться приложением.

4. Цепочка вызовов завершается выражением

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

Это выражение указывает, что для всех запросах по маршруту "/" (то есть к корню веб-приложения) в ответ будет отправляться строка "Hello World!".

В итоге при запуске проекта по типу Empty мы увидим в браузере строку "Hello World!".

Большинство встроенных методов IApplicationBuilder имеют форму `Use[название_сервиса]`. Например, `app.UseRouting()` настраивает систему маршрутизации в приложении.

Конструктор Startup

Конструктор является необязательной частью класса Startup. В конструкторе, как правило, производится начальная конфигурация приложения.

Если мы создаем проект ASP.NET Core по типу Empty, то класс Startup в таком проекте по умолчанию не содержит конструктор. Но при необходимости мы можем его определить.

Можно создать конструктор без параметров, а можно в качестве параметров передать сервисы IWebHostEnvironment (передает информацию о среде, в которой запускается приложение) и IConfiguration (передает конфигурацию приложения), которые доступны для приложения по

умолчанию. К примеру, можно получить доступный для приложения по умолчанию сервис IWebHostEnvironment, сохранить его в переменную и использовать при обработке запроса:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace HelloApp
{
    public class Startup
    {
        IWebHostEnvironment _env;
        public Startup(IWebHostEnvironment env)
        {
            _env = env;
        }
        public void ConfigureServices(IServiceCollection services)
        {
        }

        public void Configure(IApplicationBuilder app)
        {
            app.UseRouting();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapGet("/", async context =>
                {
                    await context.Response.WriteAsync($"Application Name:
{_env.ApplicationName}");
                });
            });
        }
    }
}
```

В данном случае в браузере будет выводиться название приложения, которое хранится в свойстве `_env.ApplicationName`