**Введение в ASP.NET.CORE**

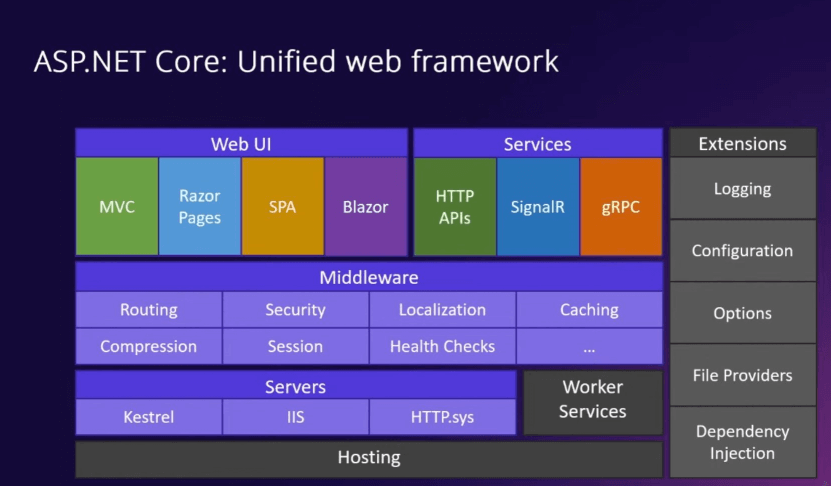
ASP.NET Core представляет технологию для создания веб-приложений на платформе .NET, развиваемую компанией Microsoft. В качестве языков программирования для разработки приложений на ASP.NET Core используются C# и F#.

История ASP.NET фактически началась с выходом первой версии .NET в начале 2002 года и с тех пор ASP.NET и .NET развивались параллельно: выход новой версии .NET знаменовал выход новой версии ASP.NET, поскольку ASP.NET работает поверх .NET. В то же время изначально ASP.NET была нацелена на работу исключительно в Windows на веб-сервере IIS (хотя благодаря проекту Mono приложения на ASP.NET можно было запускать и на Linux).

Однако 2014 год ознаменовал большие перемены, фактически революцию в развитии платформы: компания Microsoft взяла курс на развитии ASP.NET как кроссплатформенной технологии, которая развивается как opensource-проект. Данное развитие платформы в дальнейшем получило название ASP.NET Core, собственно как ее официально именут Microsoft до сих пор. Первый релиз обновленной платформы увидел свет в июне 2016 года. Теперь она стала работать не только на Windows, но и на MacOS и Linux. Она стала более легковесной, модульной, ее стало проще конфигурировать, в общем, она стала больше отвечать требованиям текущего времени.

**Архитектура и модели разработки**

Текущую архитектуру платформы ASP.NET Core можно выразить следующим образом:



На самом верхнем уровне располагаются различные модели взаимодействия с пользователем. Это технологии построения пользовательского интерфейса и обработки ввода пользователя, как MVC, Razor Pages, SPA (Single Page Application - одностраничные приложения с использованием Angular, React, Vue) и Balzor. Кроме того, это сервисы в виде встроенных HTTP API, библиотеки SignalR или сервисов GRPC.

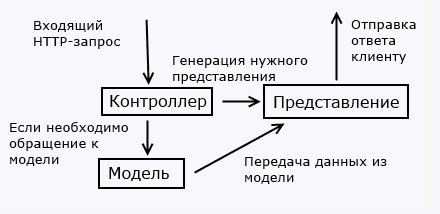
Все эти технологии базируются и/или взаимодействуют с чистым ASP.NET Core, который представлен прежде всего различными встроенными компонентами middleware - компонентами, которые применяются для обработки запроса. Кроме того, технологии высшего уровня также взаимодействуют с различными расширениями, которые не являются непосредственной частью ASP.NET Core, как расширения для логгирования, конфигурации и т.д.

И на самом нижнем уровне приложение ASP.NET Core работает в рамках некоторого веб-сервера, например, Kestrel, IIS, библиотеки HTTP.sys.

Это вкратце архитектура платформы, теперь рассмотрим моделей разработки приложения ASP.NET Core:

Прежде всего это базовый ASP.NET Core, который поддерживает все основные моменты, необходимые для работы соввременного веб-приложения: маршрутизация, конфигурация, логгирования, возможность работы с различными системами баз данных и т.д.. В ASP.NET Core 6 в фреймворк был добавлен так называемый Minimal API - минимизированная упрощенная модель, который еще упростила процесс разработки и написания кода приложения. Все остальные модели разработки работаю поверх базового функционала ASP.NET Core

ASP.NET Core MVC представляет в общем виде построения приложения вокруг трех основных компонентов - Model (модели), View (представления) и Controller (контроллеры), где модели отвечают за работу с данными, контроллеры представляют логику обработки запросов, а представления определяют визуальную составляющую.



Razor Pages представляет модель, при котором за обаботку запроса отвечают специальные сущности - страницы Razor Pages. Каждую отдельную такую сущность можно ассоциировать с отдельной веб-страницей.

ASP.NET Core Web API представляет реализацию паттерна REST, при котором для каждого типа http-запроса (GET, POST, PUT, DELETE) предназначен отдельный ресурс. Подобные ресурсы определяются в виде методов контроллера Web API. Данная модель особенно подходит для одностраничных приложений, но не только.

Blazor представляет фреймворк, который позволяет создавать интерактивные приложения как на стороне сервера, так и на стороне клиента и позволяет задействовать на уровне браузера низкоуровневый код WebAssembly.

**Особенности платформы**

ASP.NET Core работает поверх платформы .NET и, таким образом, позволяет задействовать весь ее функционал.

В качестве языков разработки применяются языки программирования, поддерживаемые платформой .NET. Официально встроенная поддержка для проектов ASP.NET Core есть у языков C# и F#

ASP.NET Core представляет кросс-платформенный фреймворк, приложения на котором могут быть развернуты на всех основных популярных операционных системах: Windows, Mac OS, Linux. И таким образом, с помощью ASP.NET Core мы можем как создавать кросс-платформенные приложения на Windows, на Linux и Mac OS, так и запускать на этих ОС.

Благодаря модульности фреймворка все необходимые компоненты веб-приложения могут загружаться как отдельные модули через пакетный менеджер Nuget.

Поддержка работы с большинством распространенных систем баз данных: MS SQL Server, MySQL, Postgres, MongoDB

ASP.NET Core характеризуется расширяемостью. Фреймворк построен из набора относительно независимых компонентов. И мы можем либо использовать встроенную реализацию этих компонентов, либо расширить их с помощью механизма наследования, либо вовсе создать и применять свои компоненты со своим функционалом.

Богатый инструментарий для разработки приложений. В качестве инструментария разработки мы можем использовать такую среду разработки с богатым функционалом, как Visual Studio от компании Microsoft.

Также можно использовать для разработки среду Rider от компании JetBrains.

Кроме того, имеющаяся оснастка .NET CLI позволяет созадвать и запускать проекты ASP.NET в консоли. И таким образом, для написания кода можно использовать обычных текстовый редактор, например, Visual Studio Code.

**Первое приложение на ASP.NET Core с .NET CLI**

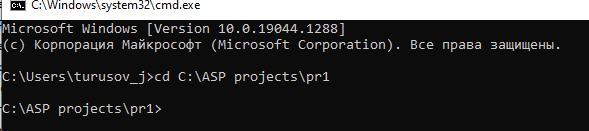
Создадим первую программу на ASP.NET Core. Что нам для этого потребуется? Прежде всего необходим текстовый редактор для написания кода программы. В данном случае я буду использовать в качестве текстового редактора Visual Studio Code

Также для компиляции и запуска программы нам потребуется .NET SDK. Для его установки перейдем на официальный сайт по ссылке .NET SDK

**https://dotnet.microsoft.com/en-us/download**

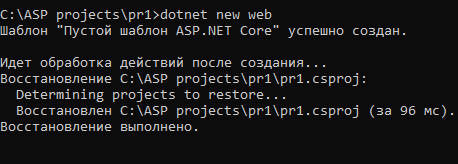


Откроем терминал/командную строку и перейдем к созданной папке проекта с помощью команды cd:



В данном случае мы для создания и запуска проекта мы будем использовать встроенную инфраструктуру .NET CLI, которая устанавливается вместе с .NET SDK.

Для создания проекта в .NET CLI применяется команда **dotnet new**, после которой указывается тип проекта. Для ASP.NET Core есть ряд встроенных типов проектов. В данном случае мы будем использовать самый простейший тип - web. Поэтому введем в терминале команду:



**Структура проекта ASP.NET Core**

Рассмотрим базовую структуру простейшего стандартного проекта ASP.NET Core:

**Dependencies:** все добавленные в проект пакеты и библиотеки, иначе говоря зависимости

**Properties:** узел, который содержит некоторые настройки проекта. В частности, в файле launchSettings.json описаны настройки запуска проекта, например, адреса, по которым будет запускаться приложение.

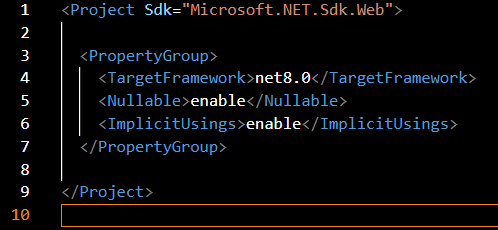
**appsettings.json:** файл конфигурации приложения в формате json

**appsettings.Development.json:** версия файла конфигурации приложения, которая используется в процессе разработки

**helloapp.csproj:** стандартный файл проекта C#, который соответствует назанию проекта (по умолчанию названию каталога) и описывает все его настройки.

**Program.cs:** главный файл приложения, с которого и начинается его выполнение. Код этого файла настраивает и запускает веб-приложение.

Например, посмотрим на содержимое файла pr1.csproj:

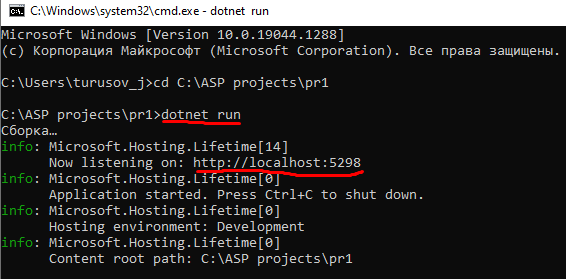


Ключевой компонент здесь - атрибут Sdk="Microsoft.NET.Sdk.Web", который собственно и определяет, что приложение будет использовать SDK "Microsoft.NET.Sdk.Web", который предназначен именно для веб-проектов.

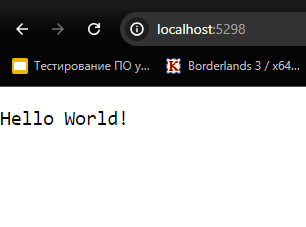
**Запуск проекта**

Проект по умолчанию не представляет какой-то грандиозной функциональности, тем не менее этот проект мы уже можем запустить. Итак, запустим проект. Для этого выполним команду:

**dotnet run**

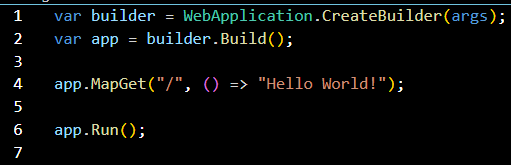


При запуске в консоли мы можем увидеть адрес, по которому мы можем обращаться к приложению. В моем случае это адрес "http://localhost:5298". И я могу обратиться по этому адресу к приложению в браузере и увидеть в нем строку "Hello World!" - результат работы кода по умолчанию из файла Program.cs:



**Запуск приложения и файл Program.cs**

Рассмотрим код файла Program.cs, который создает подобное приложение:



Это так называемое Minimal API - упрощенная минизированная модель для запуска веб-приложения в ASP.NET.

Приложение в ASP.NET Core представляет объект Microsoft.AspNetCore.Builder.WebApplication. Этот объект настраивает всю конфигурацию приложения, его маршруты, используемые зависимости и т.д.

Для создания объекта WebApplication необходим специальный класс-строитель - WebApplicationBuilder. И в файле Program.cs вначале создается данный объект с помощью статического метода WebApplication.CreateBuilder:



В качестве параметра в метод передаются аргументы, которые передаются приложению при запуске.

Получив объект WebApplicationBuilder, у него вызывается метод Build(), который собствено и создает объект WebApplication:



С помощью объекта WebApplication можно настроить всю инфраструктуру приложения - его конфигурацию, маршруты и так далее. В файле Program.cs по умолчанию для приложения определяется один маршрут:



Метод MapGet() в качестве первого параметра принимает путь, по которому можно обратиться к приложению. В данном случае это путь "/", то есть по сути корень веб-приложения - имя домена и порта, после которых может идти слеш, например, https://localhost:7256/

В качестве второго параметра в метод MapGet() передаются обработчик запроса по этому маршруту в виде функции. Здесь это лямбда-выражение, которое возвращает строку "Hello World!". Именно поэтому при обращении к приложению мы увидим данную строку в браузере.

И в конце необходимо запустить приложение. Для этого у класса WebApplication вызывается метод Run():



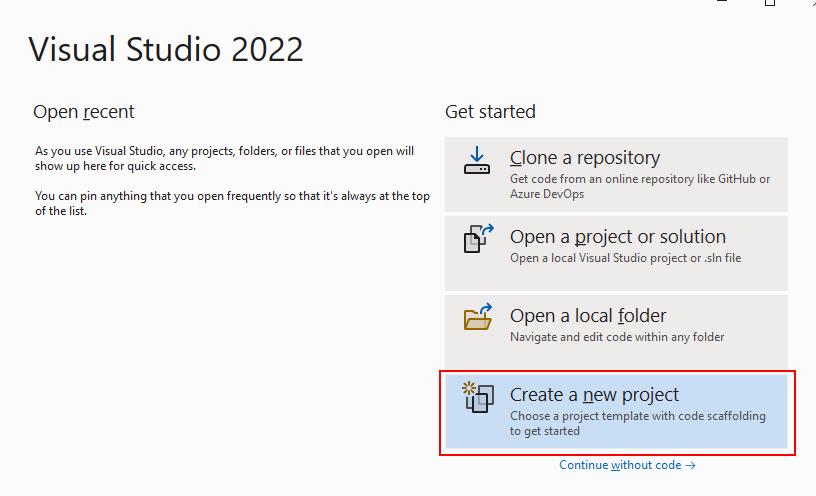
В итоге запустится приложение в виде консоли, и мы сможем обращаться к приложению из различных браузеров.

**Второй вариант создание приложения типа ASP**

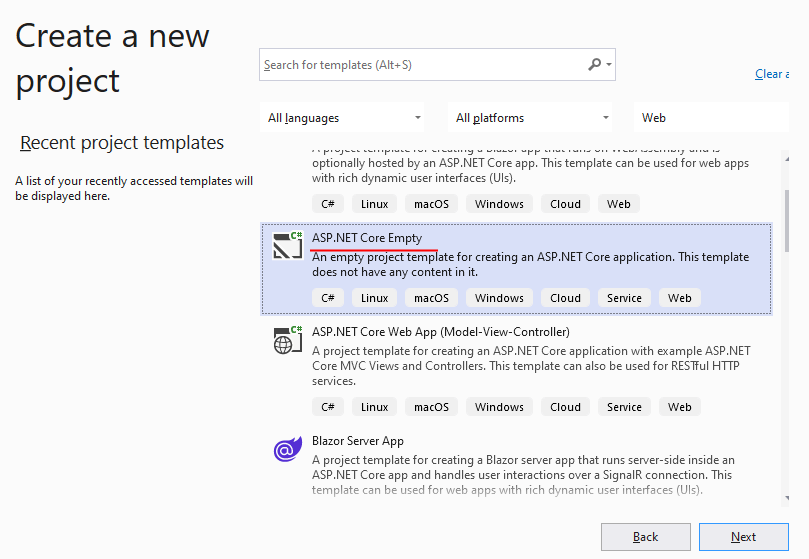
**Через Visual Studio**

В прошлой теме было рассмотрено создание первого проекта ASP.NET Core с помощью .NET CLI с компиляцией и запуском проекта в терминале. Это самый простой способ для создания приложений ASP.NET. Однако также мы можем использовать среду разработки Visual Studio, которая упрощает многие аспекты по работе с проектом ASP.NET. Рассмотрим, как создавать проект в Visual Studio.

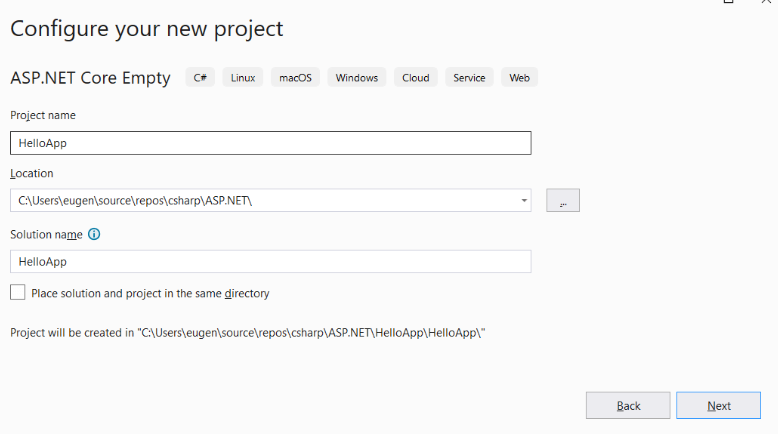
Шаг1:



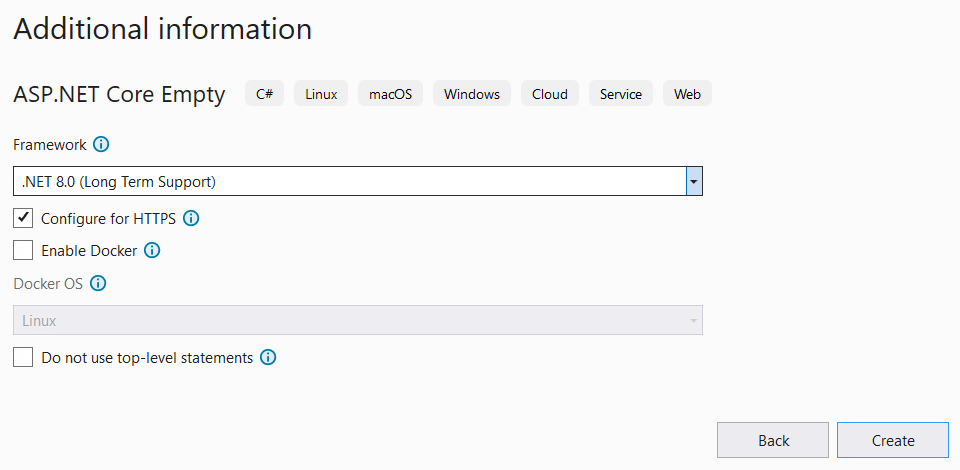
Шаг 2:



Шаг 3:



Шаг 4:

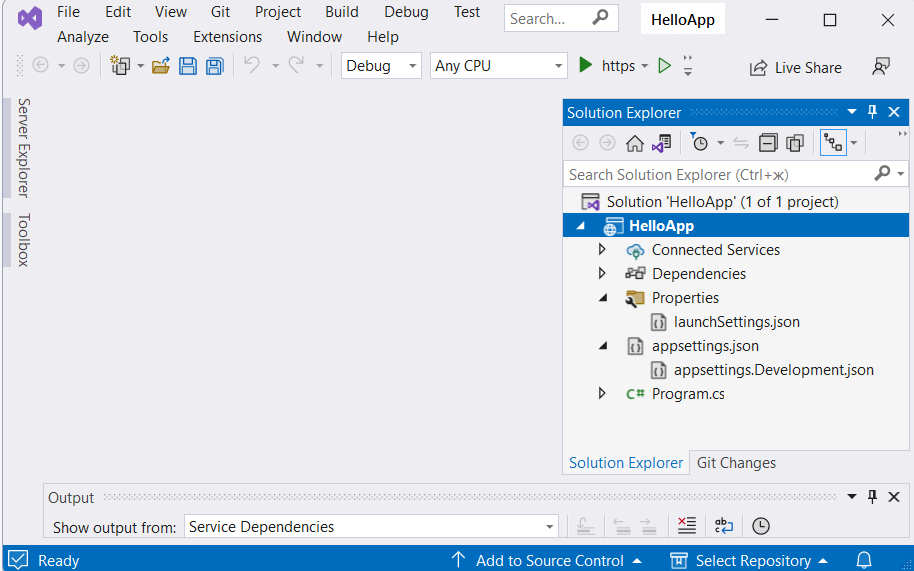


Кроме того, здесь с помощью флажка Configure for HTTPS можно установить использование протокола https. По умолчанию этот флажок отмечен, это значит, что проект по умолчанию будет использовать протокол https.

Другой флажок - Enable Docker позволяет задействовать Docker. Если этот флажок установлен, то в поле ниже можно будет выбрать ОС, которая будет использоваться под Docker. Но в данном случае оставим этот флажок неотмеченным.

Третий флажок - Do not use level statements позволяет, как и в консольных проектах, отключить поддержку выражений верхнего уровня. Этот флажок также оставим неотмеченным.

Оставим все остальные настройки по умолчанию и нажмем на кнопку Create (Создать) для создания проекта. После этого Visual Studio создаст и откроет нам проект:



**Структура проекта ASP.NET Core**

Рассмотрим базовую структуру стандартного проекта ASP.NET Core в Visual Studio. Проект ASP.NET Core Empty содержит очень простую структуру - необходимый минимум для запуска приложения:

**Connected Services:** подключенные сервисы из Azure

**Dependencies:** все добавленные в проект пакеты и библиотеки, иначе говоря зависимости

**Properties:** узел, который содержит некоторые настройки проекта. В частности, в файле launchSettings.json описаны настройки запуска проекта, например, адреса, по которым будет запускаться приложение.

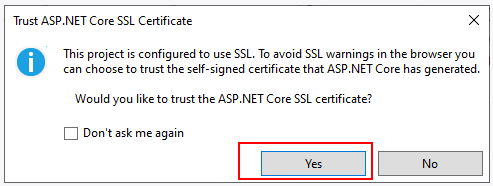
**appsettings.json:** файл конфигурации проекта в формате json

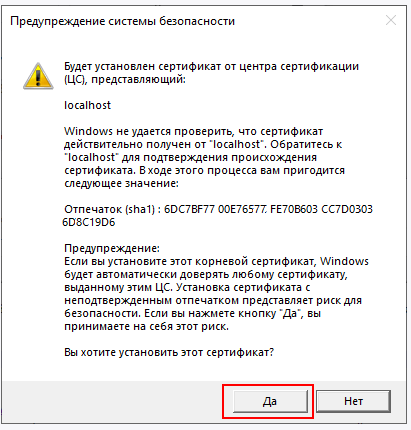
**appsettings.Development.json:** версия файла конфигурации приложения, которая используется в процессе разработки

**Program.cs:** главный файл приложения, с которого и начинается его выполнение. Код этого файла настраивает и запускает веб-приложение.

**Запуск проекта**

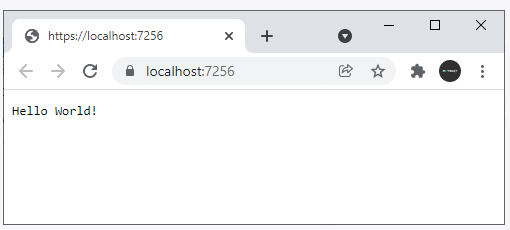
Проект по умолчанию не представляет какой-то грандиозной функциональности, тем не менее этот проект мы уже можем запустить. Итак, запустим проект. При запуске нам может отобразиться окно, где надо подтвердить доверие для серфиката SSL, а также его установку.





И после подтверждения и установки сертификата отобразиться консоль, где выводися некоторая базовая информация о приложении:

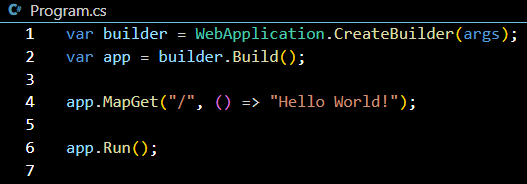
И, кроме того, будет запущен браузер, где мы сможем лицезреть строку "Hello World!" - результат работы кода по умолчанию из файла Program.cs:



**Основы в ASP.NET Core**

**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:



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

**https://github.com/dotnet/aspnetcore/blob/main/src/DefaultBuilder/src/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:

**WebApplicationOptions options = new() { Args = args };**

**WebApplicationBuilder builder = WebApplication.CreateBuilder(options);**

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:

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

**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.

**https://github.com/dotnet/aspnetcore/blob/main/src/DefaultBuilder/src/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 метод Run/Start/RunAsync/StartAsync приложение будет заущено, и мы сможем к нему обращаться:

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

**WebApplication app = builder.Build();**

**app.Run();**

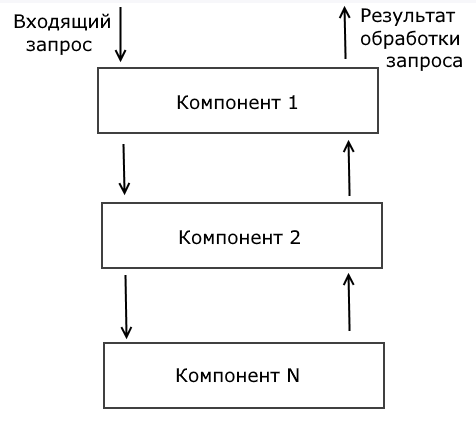
При необходимости с помощью метода StopAsync() можно программным способом завершить выполнение приложения:



**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:



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:

**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)

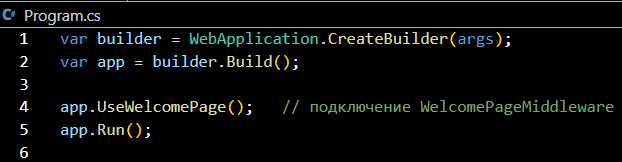
**Static Files:** предоставляет поддержку обработки статических файлов

**WebSockets:** добавляет поддержку протокола WebSockets

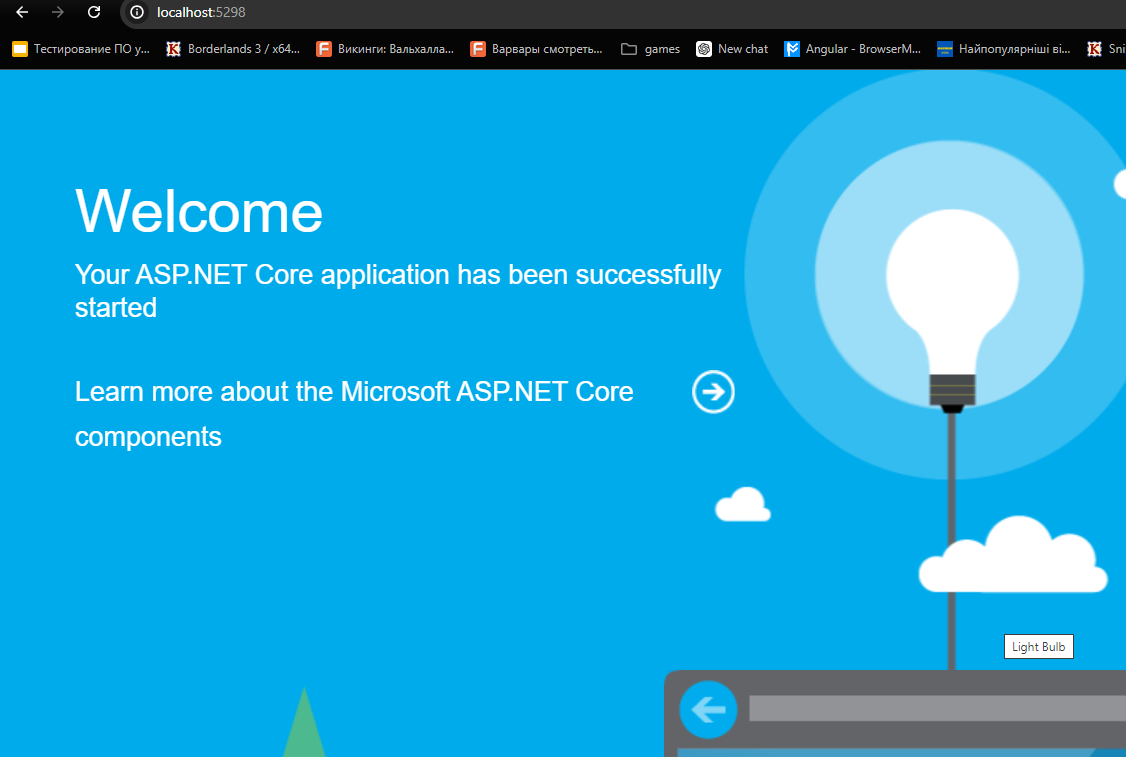
**W3CLogging:** генерирует логи доступа в соответствии с форматом W3C Extended Log File Format.

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():



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



**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 (соответствено его поддерживает и класс WebApplication):

**IApplicationBuilder.Run(RequestDelegate handler)**

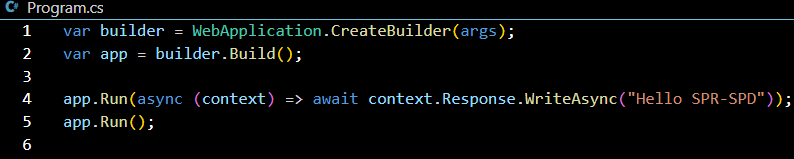
Метод Run добавляет терминальный компонент - такой компонент, который завершает обработку запроса. Поэтому соответствено он не вызывает никакие другие компоненты и обработку запроса дальше - следующим в конвейере компонентам не передает. Поэтому данный метод следует вызывать в самом конце построения 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);**

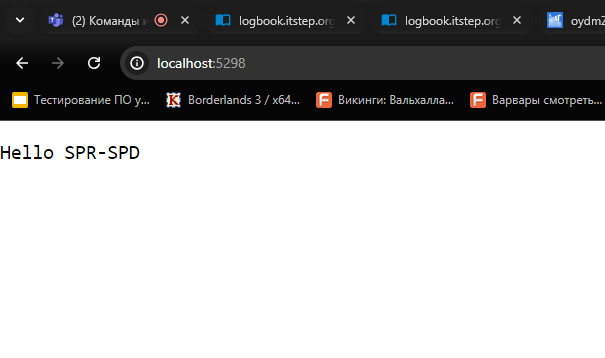
Он принимает в качестве параметра контекст запроса HttpContext и возвращает объект Task.

Используем этот метод для определения простейшего компонента:



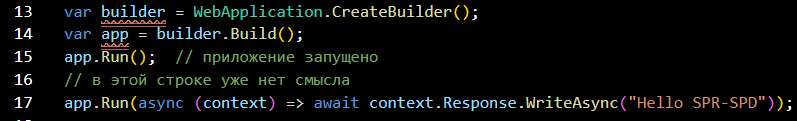
Здесь для делегата RequestDelegate передается лямбда-выражение, параметр которого - HttpContext можно использовать для отправки ответа. В частности, метод context.Response.WriteAsync() позволяет отправить клиенту некоторый ответ - в данном случае отправляется простая строка.

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

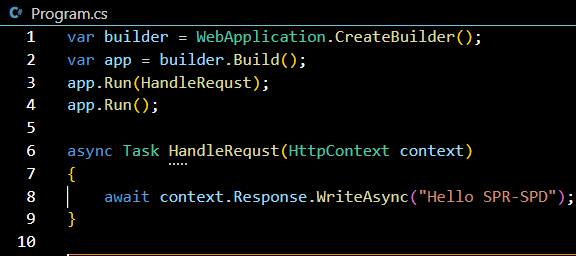


Здесь следует сделать пару замечаний. Прежде всего, не стоит путать метод Run(), который определен в классе WebApplication и который запускает приложение, и метод расширения Run(), который встраивает компонент middleware. Это два разных метода, которые выполняют разные задачи. И, как видно из кода выше,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

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

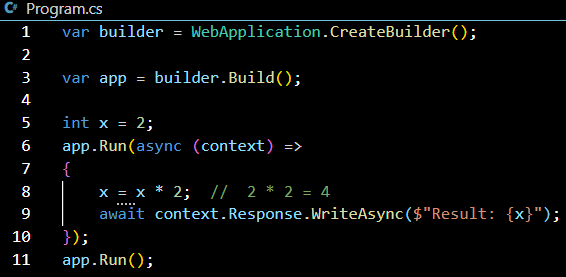


При необходимости естественно мы можем вынести код middleware в отдельный метод:

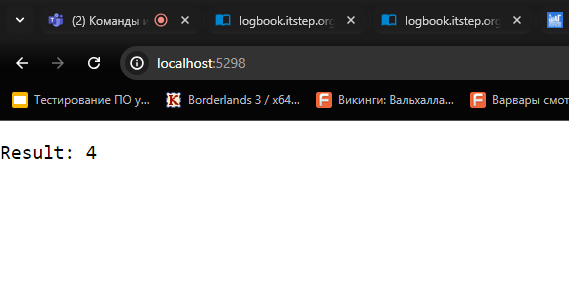


**Жизненный цикл middleware**

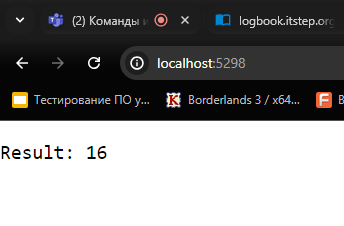
Компоненты middleware создаются один раз и существуют в течение всего жизненного цикла приложения. То есть для последующей обработки запросов используются одни и те же компоненты. Например, определим в файле Program.cs следующий код:



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



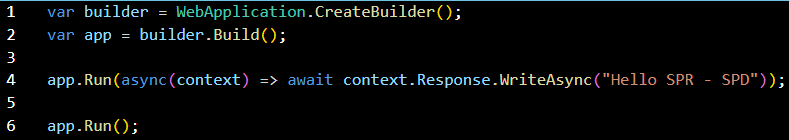
Однако при последующих запросах мы увидим, что результат переменной х не равен 4.



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.

**HttpResponse. Отправка ответа**

Все данные запроса передаются в middleware через объект Microsoft.AspNetCore.Http.HttpContext. Этот объект инкапсулирует информацию о запросе, позволяет управлять ответом и, кроме того, имеет еще много другой функциональности. Например, возьмем простейшее приложение:



Здесь параметр context, который передается в middleware в методе app.Run() как раз представляет объект HttpContext. И через этот объект, точнее через его свойство Response мы можем отправить клиенту некоторый ответ: context.Response.WriteAsync($"Hello SPR - SPD").

Свойство Response объекта HttpContext представляет объект HttpResponse и устанавливает, что будет отравляться в виде ответа. Для установки различных аспектов ответа класс HttpResponse определяет следующие свойства:

**Body:** получает или устанавливает тело ответа в виде объекта Stream

**BodyWriter:** возвращает объект типа PipeWriter для записи ответа

**ContentLength:** получает или устанавливает заголовок Content-Length

**ContentType:** получает или устанавливает заголовок Content-Type

**Cookies:** возвращает куки, отправляемые в ответе

**HasStarted:** возвращает true, если отправка ответа уже началась

**Headers:** возвращает заголовки ответа

**Host:** получает или устанавливает заголовок Host

**HttpContext:** возвращает объект HttpContext, связанный с данным объектом Response

**StatusCode:** возвращает или устанавливает статусный код ответа

Чтобы отправить ответ, мы можем использовать ряд методов класса HttpResponse:

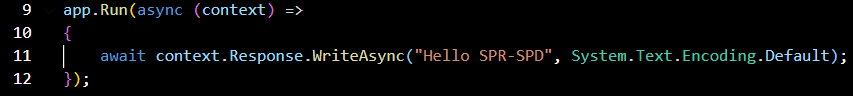
**Redirect():** выполняет переадресацию (временную или постоянную) на другой ресурс

**WriteAsJson()/WriteAsJsonAsync():** отправляет ответ в виде объектов в формате JSON

**WriteAsync():** отправляет некоторое содержимое. Одна из версий метода позволяет указать кодировку. Если кодировка не указана, то по умолчанию применяется кодировка UTF-8

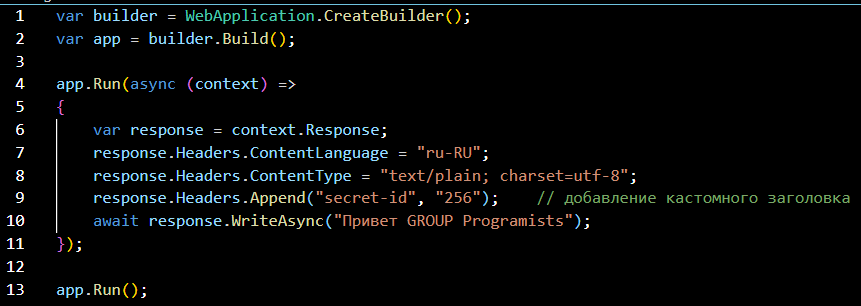
**SendFileAsync():** отправляет файл

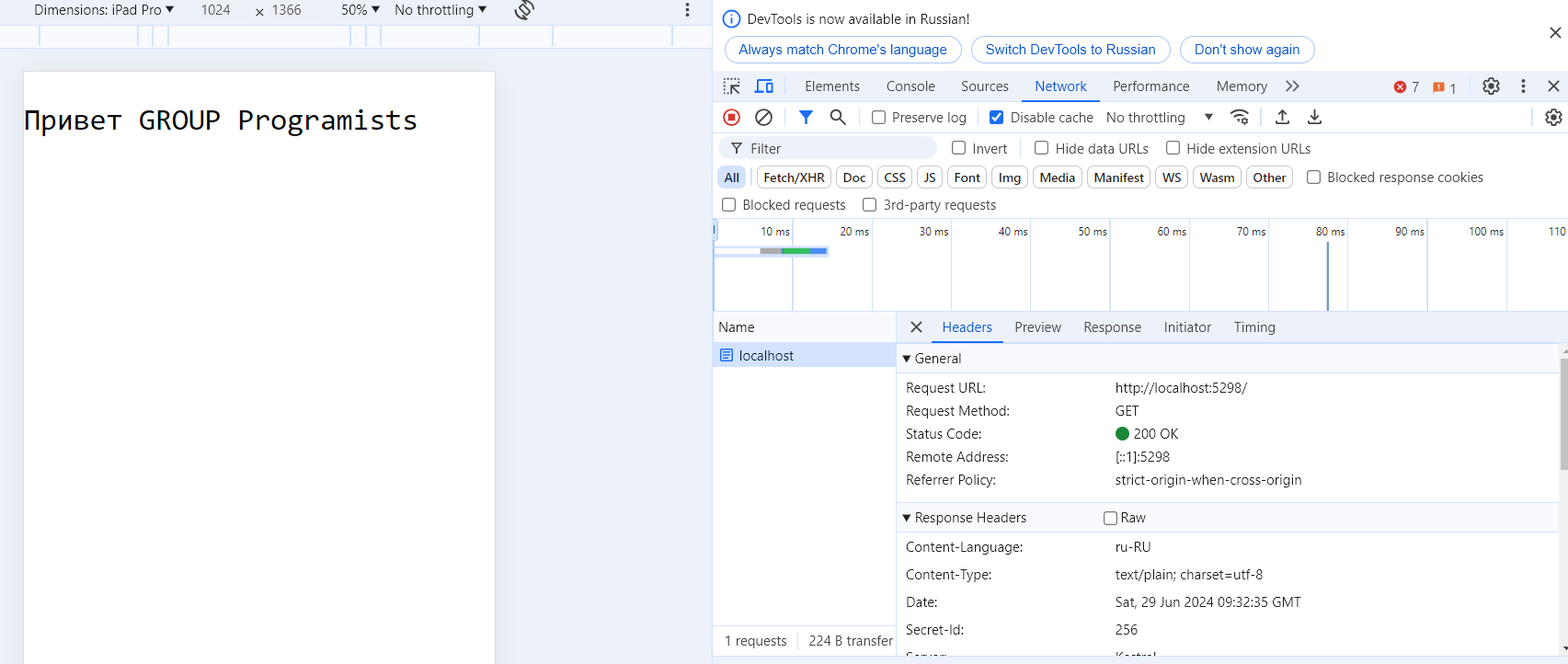
Самый простой способ отправки ответа представляет метод WriteAsync(), в который передается отправляемые данные. В качестве дополнительного параметра мы можем указать кодировку:



**Установка заголовков**

Для установки заголовков применяется свойство Headers, которое представляет тип IHeaderDictionary. Для большинства стандартных заголовков HTTP в этом интерфейсе определены одноименные свойства, например, для заголовка "content-type" определено свойство ContentType. Другие, в том числе свои кастомные заголовки можно добавить через метод Append(). Например:





Стоит отметить, что для вывода кириллицы желательно устанавливать заголовок ContentType, в том числе кодировку, которая применяется в отправляемом содержимом (в примере выше это "text/plain; charset=utf-8").

Также стоит отметить, что вместо

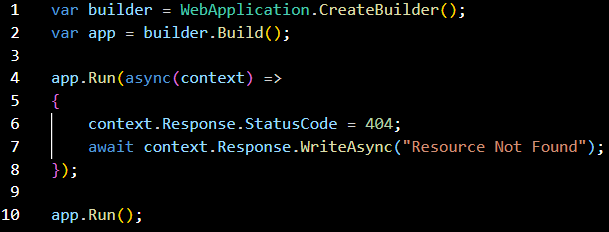
**response.Headers.ContentType = "text/plain; charset=utf-8";**

можно было бы написать

**response.ContentType = "text/plain; charset=utf-8";**

**Установка кодов статуса**

Для установки статусных кодов применяется свойство StatusCode, которому передается числовой код статуса:



В данном случае устанавливается код 404, который указывает, что ресурс не найден.



**Отправка html-кода**

Если необходимо отправить html-код, то для этого необходимо установить для заголовка Content-Type значение text/html:

