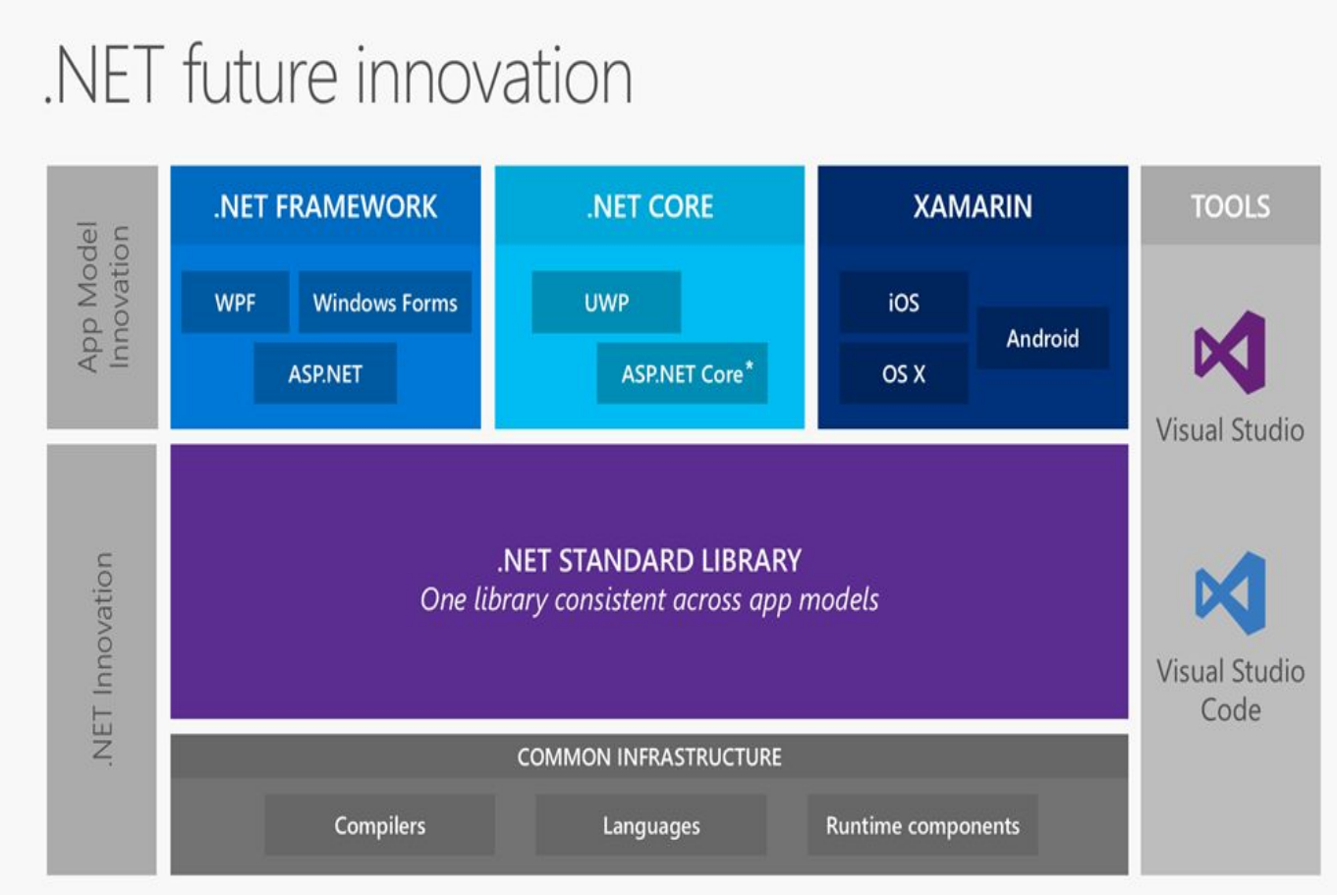
Введение в ASP.Net Core MVC

Платформа **ASP.NET Core** - технология от компании Microsoft для создания веб-приложений.

ASP.NET Core является *opensource* фреймворком. Все исходные файлы доступны на GitHub.

Версия	**Дата**
MVC 1	13 Марта 2009
MVC 2	10Марта 2010
MVC 3	13 Января 2011
MVC 4	15 Августа 2012
MVC 5	17 Октября 2013
MVC 6 Core	2016
Core 2.1	май 2018
ASP.NET Core 3	декабрь 2019 года

ASP.NET Core включает в себя фреймворк MVC, который объединяет функциональность MVC, Web API и Web Pages.



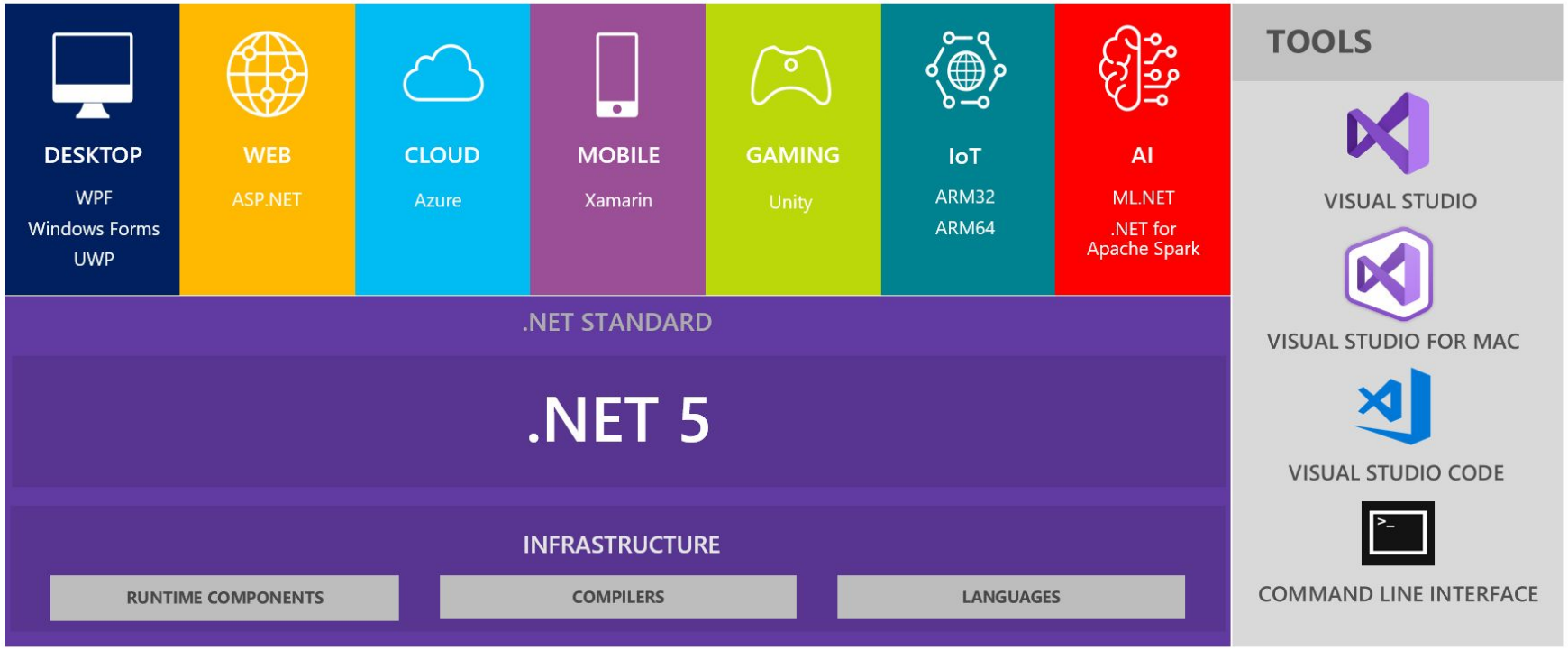
https://dou.ua/lenta/articles/net-evolution/

Эволюция .NET-стека: что изменилось за последние несколько лет

https://habr.com/ru/company/raiffeisenbank/blog/451136/

Представлен .NET 5

.NET – A unified platform



Вышла новая версия платформы .NET под номером 5.0
https://tproger.ru/news/dotnet-5-0-released/

Для развертывания веб-приложения можно использовать традиционный **IIS** или кросс-платформенный веб-сервер **Kestrel**.

(**Internet Information Services**) — программное обеспечение для развертывания веб-сервера.

Входит в состав Windows. Поддерживает работу по протоколам HTTP, HTTPS, FTP, SMTP, POP3.

По сравнению с предыдущими версиями ASP.NET при работе с ASP.NET Core IIS не использует инфраструктуру **System.Web**, что значительно повышает производительность приложения

Kestrel — это новый, ***открытый, кроссплатформенный*** (Windows, Linux, Mac) веб-сервер на базе Libuv. Libuv — это кроссплатформенная компонента для поддержки асинхронных операций ввода\вывода: асинхронная работа с TCP\UDP сокетами, DNS резервация, операции с файловой системой, управление потоками и др.

Kestrel по умолчанию включается в проект ASP.NET Core.

При развертывании на Windows Kestrel может применять IIS в качестве прокси-сервера, а при развертывании на Linux как прокси-серверы могут использоваться Apache и Nginx. Также Kestrel может работать самостоятельно внутри своего процесса без IIS.

По умолчанию в Visual Studio доступны для запуска два профиля:

- IIS Express (запуск с проксированием через IIS Express)

- Профиль, который совпадает с названием проекта - тот пункт, который позволяет запускать приложение в отдельном процессе без IIS. В данном случае приложение будет запускать именно Kestrel. Причем по умолчанию приложение будет запускаться на **5000**-порту.

HTTP.sys (раньше WebListener)

Это HTTP-сервер для ASP.NET Core, который работает только в ОС Windows и дает возможность хостить ASP.NET приложения без IIS. Он работает напрямую с Http.Sys (драйвер Kernel). Его основное достоинство — поддержка большинства интерфейсов IIS, но при этом малая ресурсоемкость.

Kestrel рекомендуется запускать вместе с проксирующими веб-серверами типа IIS, Apache для обеспечения большей безопасности.

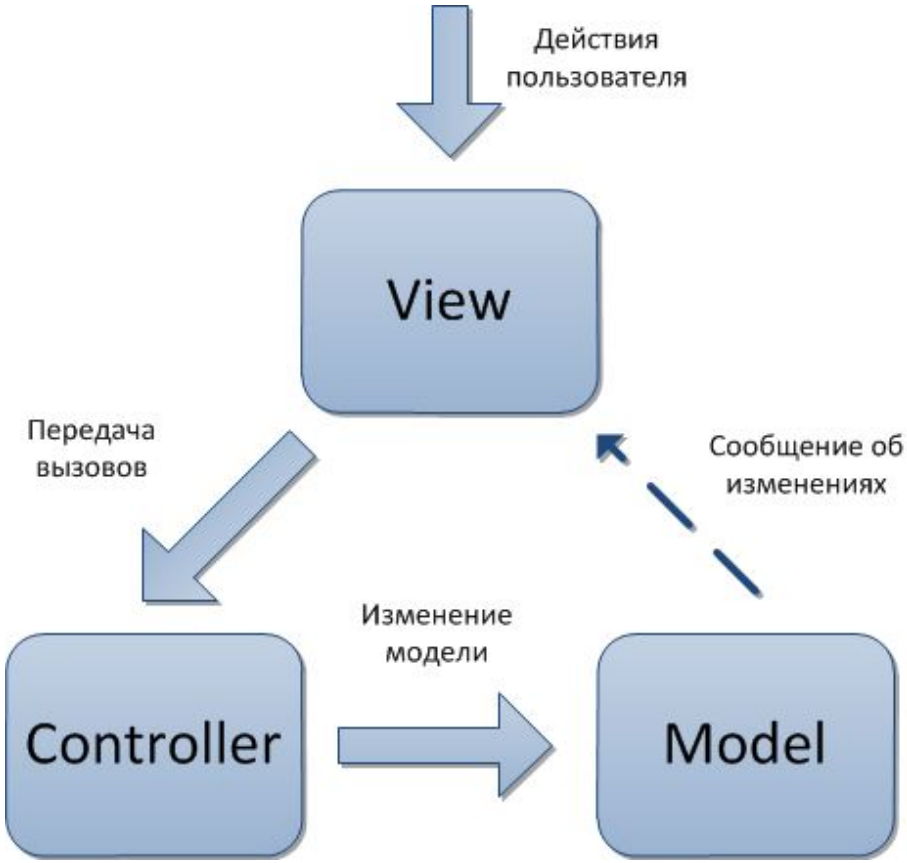
Для HTTP.sys не требуется проксирующего веб-сервера (IIS и не может работать с HTTP.sys), так как Http.Sys позволяет обеспечить безопасность и надежность.

Архитектура Модель – Вид – Контроллер

ASP.NET Core MVC – платформа для разработки веб-приложений, реализующих паттерн **«модель - представление - контроллер»**

Нужно различать архитектурный паттерн MVC и реализацию ASP.NET Core MVC. Паттерн MVC появился в 1978 году и связан с проектом Smalltalk в Xerox PARC.

Model-View-Controller (**MVC**) — шаблон, позволяющий разделять данные приложения, пользовательский интерфейс и управляющую логику на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо


Действия
пользователя
View
Передача
вызовов
Сообщение об
изменениях
Изменение
модели
Controller
Model

Основная идея этого паттерна в том, что и контроллер и представление зависят от модели, но модель никак не зависит от этих двух компонент.

Модель (*Model*) предоставляет данные и реагирует на команды контроллера, изменяя своё состояние.

Представление (*View*) отвечает за отображение данных модели пользователю, реагируя на изменения модели.

Контроллер (*Controller*) интерпретирует действия пользователя, оповещая модель о необходимости изменений.

Так как MVC не имеет строгой реализации, то реализован он может быть по-разному.

Нет общепринятого определения, где должна располагаться бизнес-логика.

Она может находиться как в контроллере, так и в модели. В последнем случае, модель будет содержать все бизнес-объекты со всеми данными и функциями.

Некоторые фреймворки жестко задают где должна располагаться бизнес-логика, другие не имеют таких правил.

Также не указано, где должна находиться проверка введённых пользователем данных.

Простая валидация может встречаться даже в представлении, но чаще они встречаются в контроллере или модели

Контроллер определяет, какое представление должно быть отображено в данный момент.

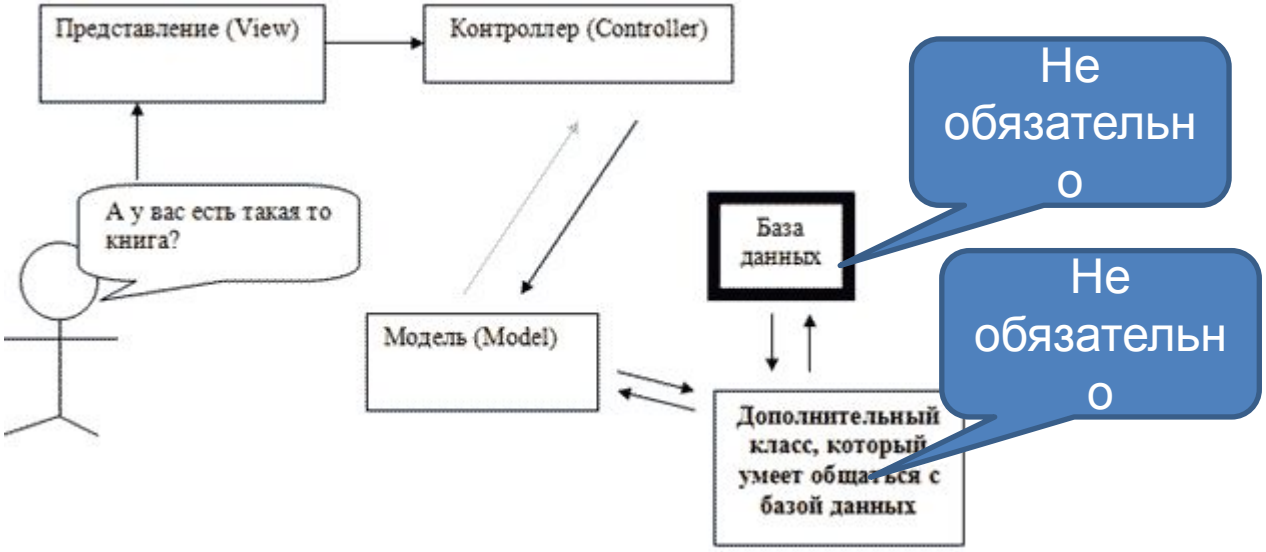
События представления могут повлиять только на контроллер. Контроллер может повлиять на модель и определить другое представление.

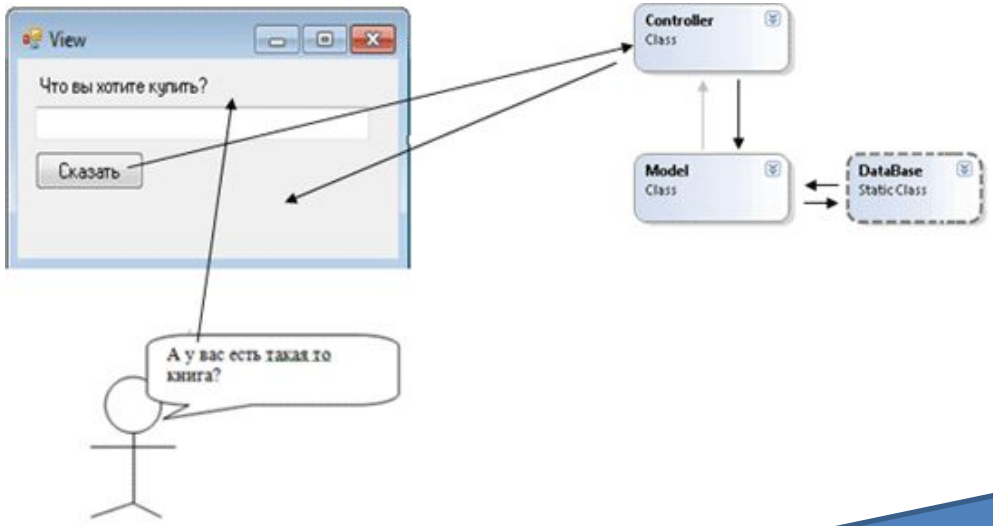
Возможно несколько представлений только для одного контроллера

Пример.

http://skillcoding.com/Default.aspx?id=230

Покупатель в книжном магазине





Представление

```
public partial class View : Form
    {
        // создаем объект класса Controller
        Controller controller = new Controller();
        public View()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (textBox1.Text != "")
                // выводим результат
                MessageBox.Show(controller.Question(textBox1.Text));
        }
    }
```

```
class Controller
    {
        public string Question(string msg)//msg - то что ищем
        {
            Model model =  new Model();
            return "На даный момент у нас товар "
                + model.GetAnser(msg);
        }
    }


class Model
    {
        //question - то что ищем
        public string GetAnser(string question)
        {
            return DataBase.GetAnser(question);
        }
    }
```

Инфраструктура ASP.NET Core MVC реализует паттерн MVC. На самом деле в ASP.NET Core MVC внедрена разновидность паттерна MVC, которая особенно хорошо подходит для веб-приложений.

В инфраструктуре ASP.NET Core MVC контроллеры - это классы C#, обычно производные от класса **Microsoft. AspNetCore.Mvc.Controller.** Каждый открытый метод в производном от Controller классе является методом действия. который ассоциирован с каким-то URL.

Когда запрос посылается по URL, связанному с методом действия, операторы в данном методе действия выполняются, чтобы провести некоторую операцию над моделью предметной области, а затем выбрать представление для отображения клиенту.

В приложении ASP.NET Core MVC представление является файлом, содержащим HTML-элементы и код C#, который обрабатывается для генерации ответа.

Преимущества Asp.Net Core MVC

- *Расширяемость*

ASP.NET Core и ASP.NET Core MVC построены в виде последовательности независимых компонентов, которые реализуют интерфейс .NET или созданы на основе абстрактного базового класса. Основные компоненты можно легко заменять другими компонентами с собственной реализацией.

Для каждого компонента MVC Framework разработчику представляет три возможности:

- Использование **стандартного** компонента
- **Порождение подкласса** от стандартной реализации
- Полная замена компонента новой **реализацией интерфейса**

-Жесткий контроль над HTML и HTTP

• Генерация разметки, которая соответствует стандартам
• Сгенерированные ASP.NET Core MVC страницы не содержат никаких данных View State, поэтому они меньше типовых страниц ASP.NET Web Forms по размеру.
• Полный контроль над запросами

ASP.NET Core MVC, позволяет легко использовать наилучшие клиентские библиотеки: jQuery, Angular или Bootstrap CSS. Компания Microsoft включила их поддержку как встроенных частей стандартного шаблона проектов для веб-приложений.

- *Тестируемость*

Разнесение различных задач приложения по разным, независимым друг от друга частям программного обеспечения, позволяет строить легко тестируемые приложения.

- *Мощная система маршрутизации*

В ASP.NET MVC применяется средство, известное как **маршрутизация URL**, которое обеспечивает предоставление понятных URL-адресов по умолчанию.

- *Построение на основе лучших частей платформы ASP.NET*

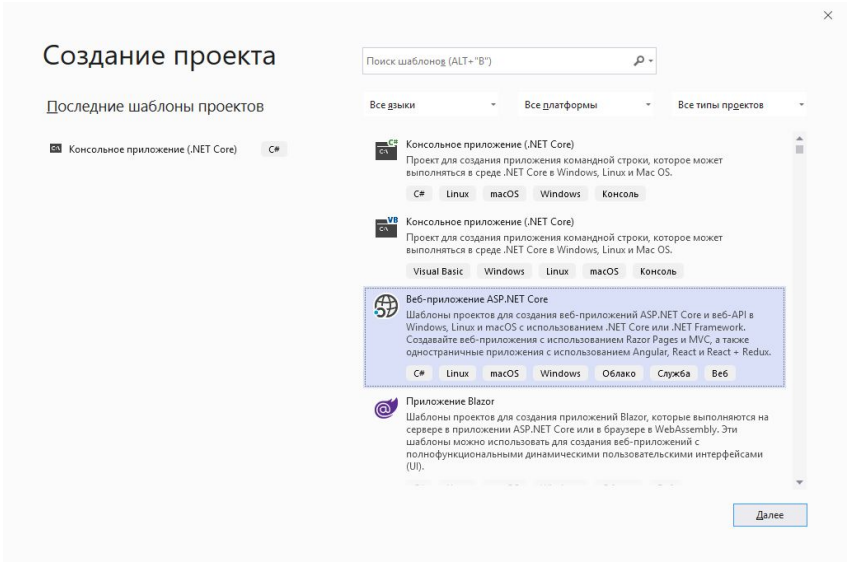
Код можно писать на любом языке .NET и при этом иметь доступ к одним и тем же функциям, которые определены не только в MVC Framework, но и в библиотеке классов .NET, а также в широком множестве библиотек .NET от независимых разработчиков.

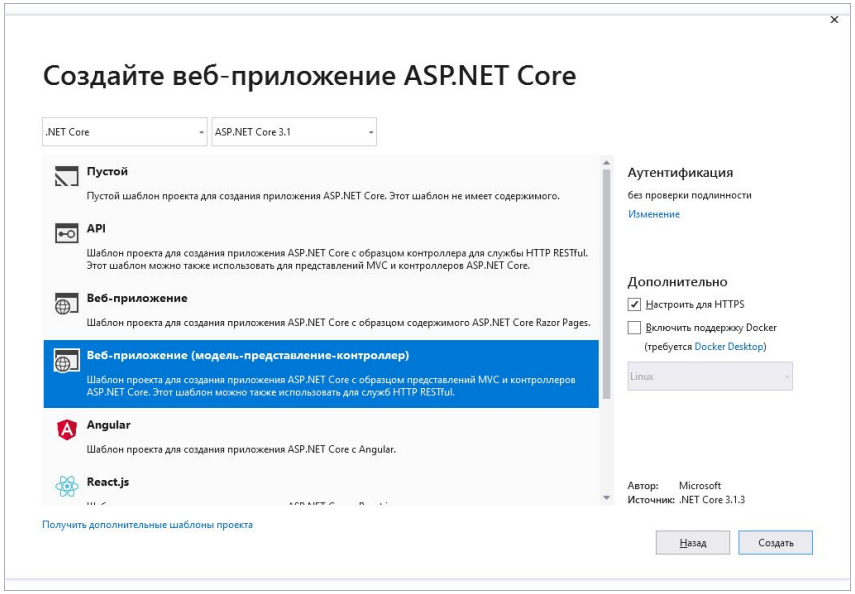
- *Инфраструктура ASP.NET Core MVC имеет открытый код*

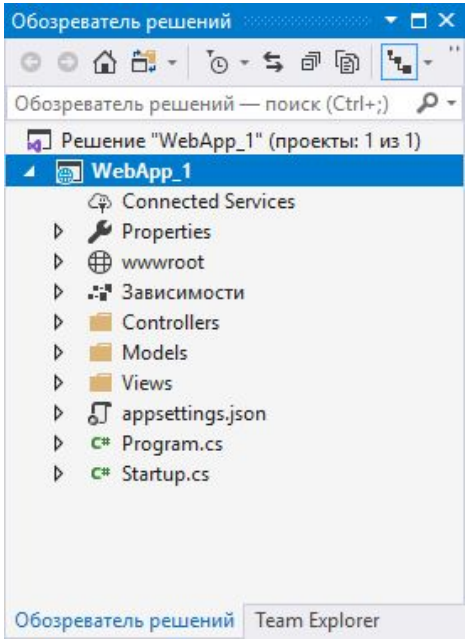
Исходный код можно изменять, развертывать и даже распространять в виде производного проекта. Исходный код ASP.NET Core и ASP.NET Core MVC доступен для загрузки по адресу: https://github .com/aspnet .

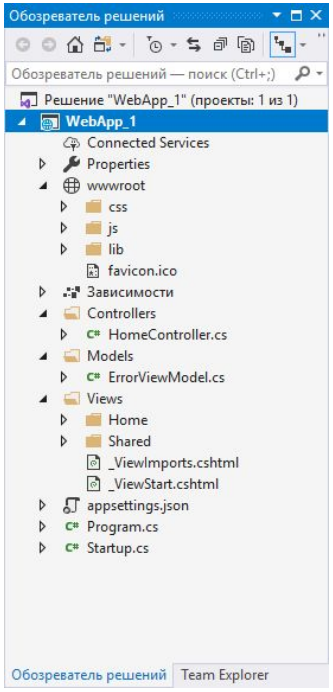
Неверно отождествлять ASP.NET Core с фреймворком ASP.NET Core MVC. Фреймворк ASP.NET Core MVC работает поверх платформы ASP.NET Core, и предназначен для того, чтобы упростить создание приложения. Но можно не использовать MVC, а применять чистый ASP.NET Core и на нем всецело выстраивать логику приложения.

Создание простого проекта

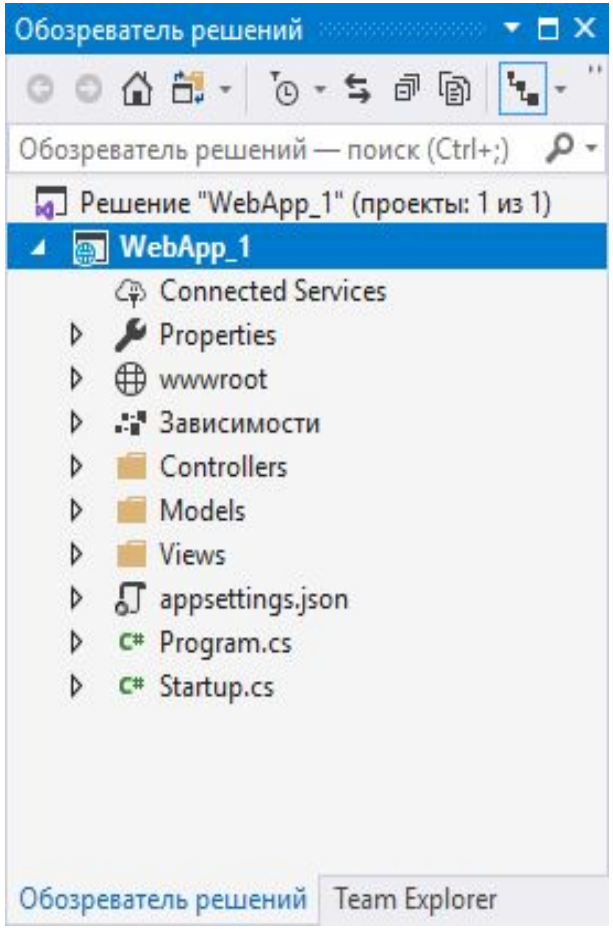








Структура проекта



Dependencies: все добавленные в проект пакеты и библиотеки

wwwroot: этот узел предназначен для хранения статических файлов - изображений, скриптов javascript, файлов css и т.д., которые используются приложением.

Controllers: содежит файлы классов контроллеров. По умолчанию в эту папку добавляется HomeController

Models: содержит файлы моделей. По умолчанию Visual Studio добавляет пару моделей, служащих для аутентификации пользователя

Views: здесь хранятся представления. Все представления группируются по папкам, каждая из которых соответствует одному контроллеру. После обработки запроса контроллер отправляет одно из этих представлений клиенту. Каталог Shared содержит общие для всех представления

Startup.cs: файл, определяющий класс Startup. Этот класс производит конфигурацию приложения, настраивает сервисы, которые приложение будет использовать, устанавливает компоненты для обработки запроса или middleware.

appsettings.json: хранит конфигурацию приложения

Program.cs: файл, определяющий класс Program, который инициализирует и запускает хост с приложением.

Соглашения в MVC

В проекте MVC применяются два вида соглашений. Соглашения первого вида - это просто предположения о том, как может выглядеть структура проекта.
Например:

В папке **wwwroot** должны находится изображения, CSS файлы и т.д.
В папке **Content** изображения

Но это не является обязательным. Просто именно в соответствующих папках рассчитывают обнаружить указанные элементы другие разработчики, использующие MVC. Но, например, классы моделей могут быть определены где угодно в текущем проекте или вообще вынесены в отдельный проект.

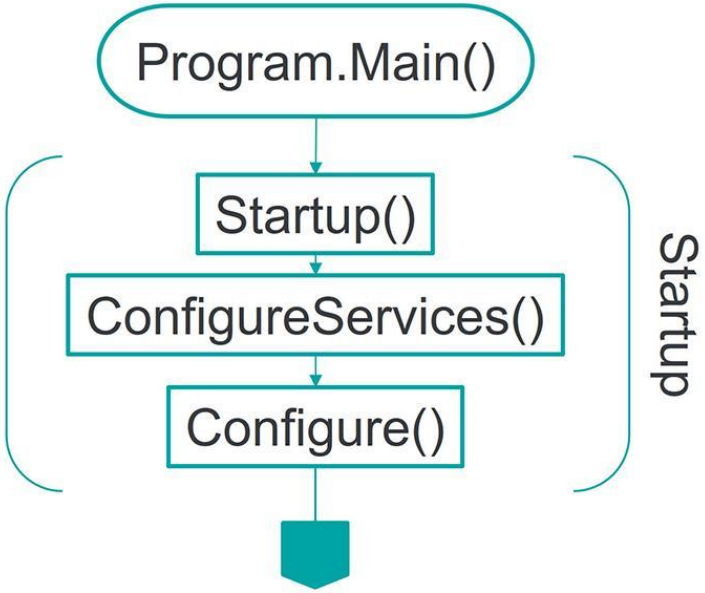
Соглашения второго вида - это *соглашения по конфигурации (convention over configuration)* . Соглашение по конфигурации означает, что не нужно явно конфигурировать, к примеру, ассоциации между контроллерами и их представлениями. Нужно просто следовать определенному соглашению об именовании для файлов - и все будет работать.

Классы контроллера должны иметь имена, заканчивающиеся словом **Controller.**
Например, ProductController, AdminController, HomeController

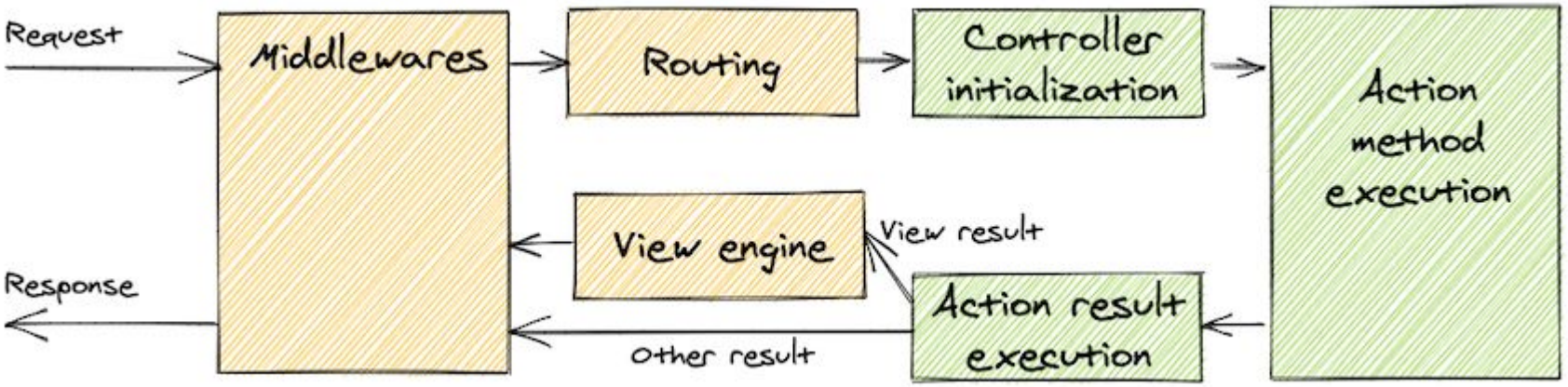
Представления должны располагаться в папке **/Views/Имя_контроллера**
Например, представления для ProductController должны находится в каталоге
/Views/Product

Жизненный цикл приложения и запроса

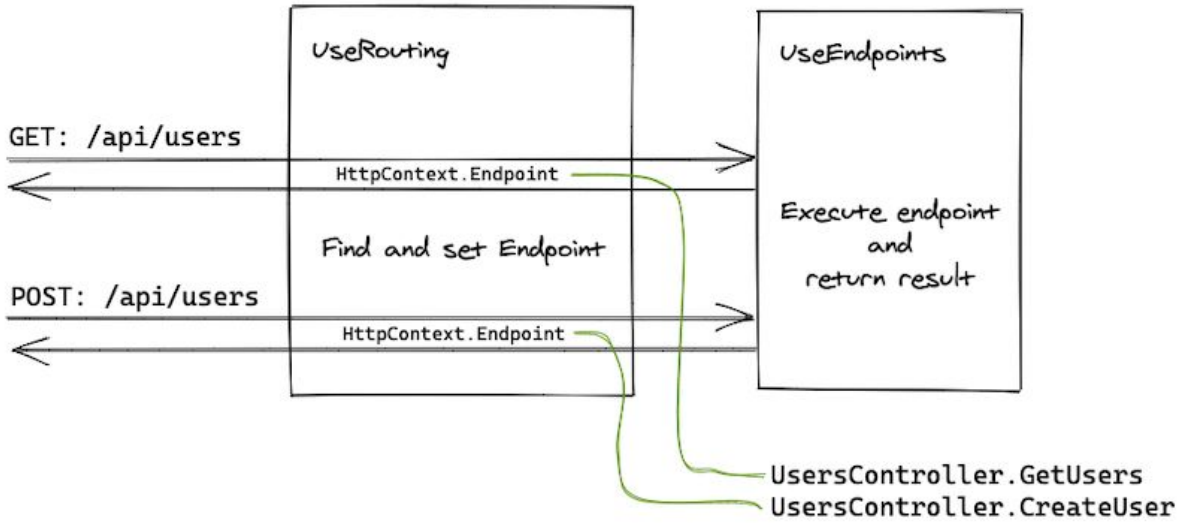
https://stefaniuk.website/all/zhiznenny-cikl-zaprosov-v-asp-net-core-mvc/ **Богдан Стефанюк**



Приложения ASP.NET Core представляют собой приложение .NET Core Console, которое вызывает специальные библиотеки ASP.NET. Это фундаментальное изменение в разработке основных приложений ASP.NET. Вместо размещения приложения в IIS все библиотеки размещения ASP.NET выполняются из Program.cs.



Middlewares (промежуточное программное обеспечение - software that acts as a bridge between an operating system or database and applications, especially on a network.) представляют из себя базовые блоки, с помощью которых строится конвейер, который обрабатывает каждый запрос. Каждый блок получает запрос и смотрит на него, если может предоставить ответ — возвращает его, если нет, передаёт запрос следующему блоку.

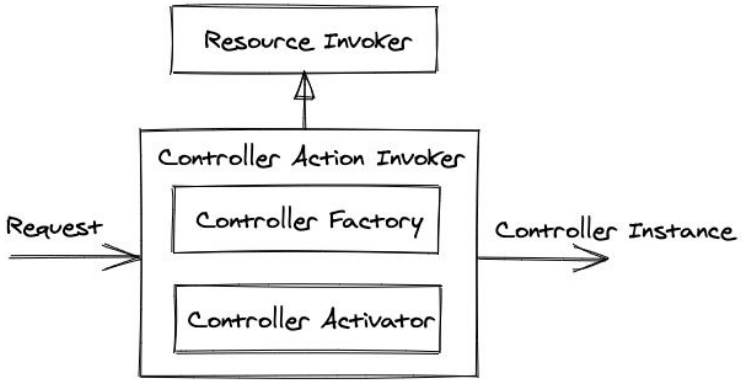


Маршрутизация (Routing) позволяет найти для каждого URL подходящий обработчик, а также извлекает все параметры из URL. Для маршрутизации используются две middleware:

UseRouting

UseEndpoints

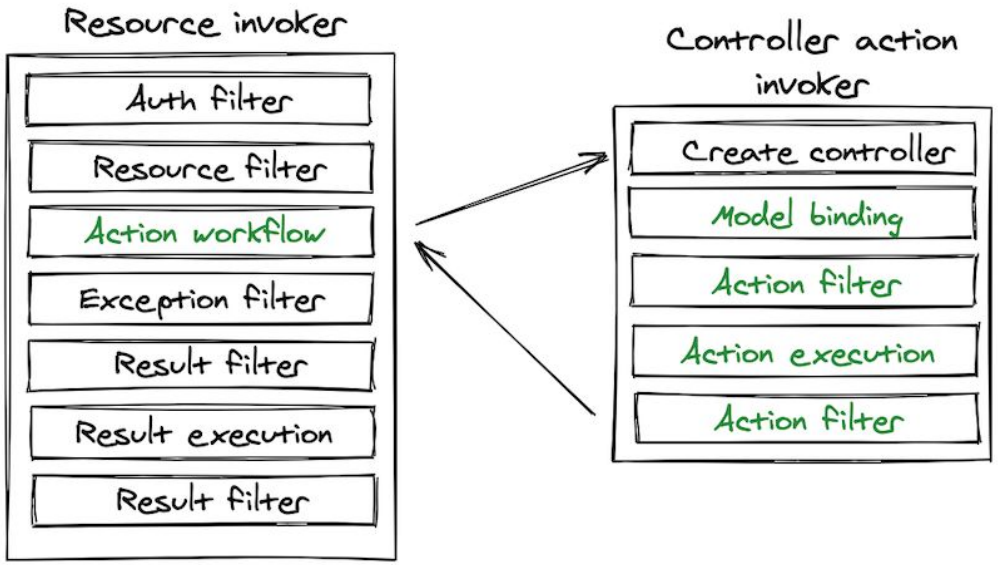
Инициализация контроллера:



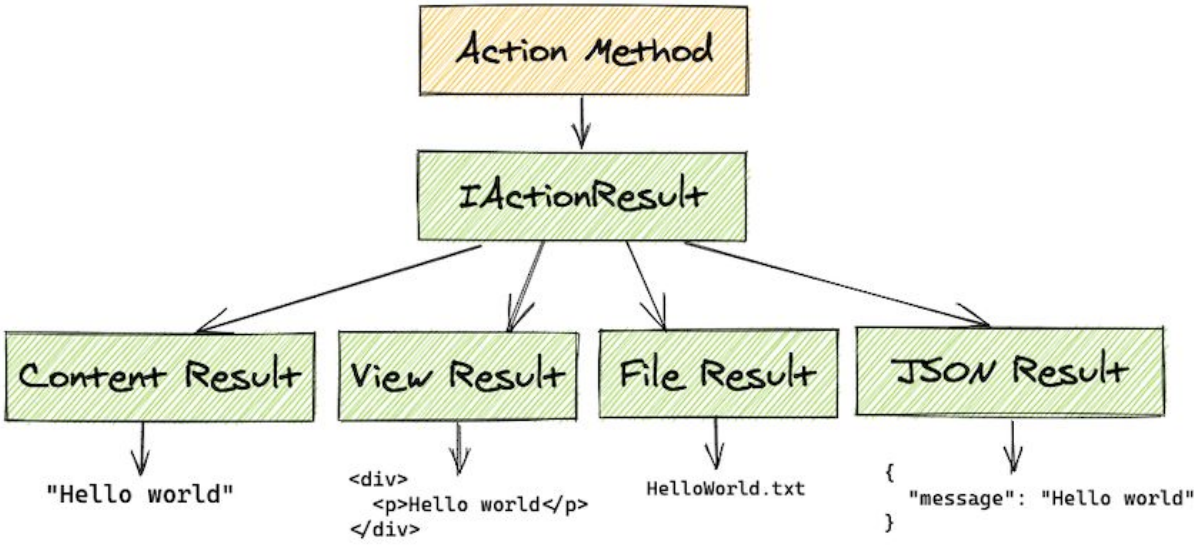
Работа каждого метода контроллера состоит из своего жизненного цикла.

Общий вид этого жизненного цикла:

Фильтры авторизации
Фильтры ресурсов
Привязки моделей
Фильтры действий
Выполнение метода контроллера
Фильтры исключений
Фильтры результатов
Выполнение результата
Фильтры результатов

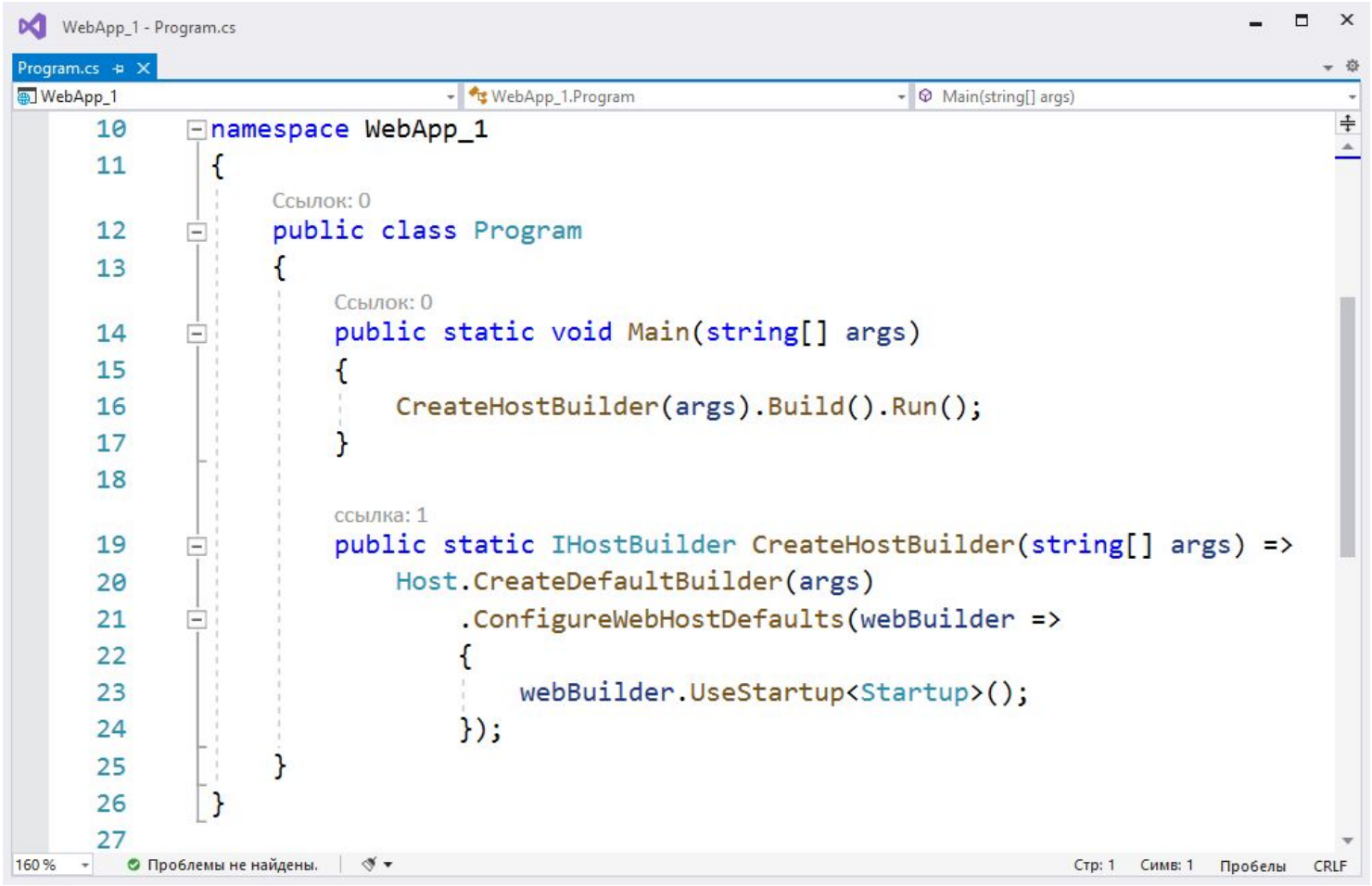


Методы контроллера возвращают объекты результата, которые в дальнейшем преобразовываются в соответствующее представление.



ASP.NET Core обеспечивает полный контроль над жизненным циклом приложения.

Раннер:



При запуске приложение ASP.NET Core создает *хост* (объект **IHost**) .

Хост инкапсулирует ресурсы приложения, такие как:

- Реализация HTTP-сервера
- Компоненты промежуточного программного обеспечения (**middleware**)
- логирование
- Услуги внедрения зависимостей (DI)
- конфигурация

В методе Main вызывается метод **Build()** у созданного объекта IHostBuilder , который создает хост - объект **IHost**, а затем для непосредственного запуска у IHost вызывается метод **Run.**

После этого приложение запущено, и веб-сервер начинает прослушивать все входящие HTTP-запросы.

Для создания IHost применяется объект **IHostBuilder**.

Создание IHostBuilder производится с помощью метода **Host.CreateDefaultBuilder(args)**. *Тип результата – IHostBuilder.*

Данный метод

- Устанавливает корневой каталог
- Устанавливает конфигурацию хоста. Для этого загружаются переменные среды с префиксом "DOTNET_" и аргументы командной строки
- Устанавливает конфигурацию приложения (загружается содержимое из файлов appsettings.json и appsettings.{Environment}.json), а также переменные среды и аргументы командной строки.
- Добавляет провайдеры логирования
- Если проект в статусе разработки, то также обеспечивает валидацию сервисов

Затем вызывается метод **ConfigureWebHostDefaults()**. *Тип результата – IHostBuilder.*

Этот метод выполняет конфигурацию параметров хоста:

- Загружает конфигурацию из переменных среды с префиксом "ASPNETCORE_"
- Запускает и настраивает веб-сервер Kestrel, в рамках которого будет разворачиваться приложение
- Добавляет компонент Host Filtering, который позволяет настраивать адреса для веб-сервера Kestrel
- Если для работы приложения требуется IIS, то данный метод также обеспечивает интеграцию с IIS

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.

Самым важным компонентом является класс **Startup**, который применяется для создания служб (объектов, предоставляющих общую функциональность повсюду в приложении) и компонентов промежуточного программного обеспечения (ПО) , используемых для обработки HTTP-запросов.

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

Метод Configure используется для указания того, как приложение отвечает на HTTP-запросы. Конвейер запросов настраивается путем добавления компонентов промежуточного программного обеспечения в экземпляр IApplicationBuilder.

Хостинг создает IApplicationBuilder и передает его непосредственно Configure.

WebApp_1 - Startup.cs

Startup.cs

WebApp_1 | WebApp_1.Startup | Startup(IConfiguration configuration)

```
using Microsoft.Extensions.Hosting;

namespace WebApp_1
{
    Ссылок: 2
    public class Startup
    {
        Ссылок: 0
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        ссылка: 1
        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        Ссылок: 0
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        Ссылок: 0
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
                // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }
            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}

```

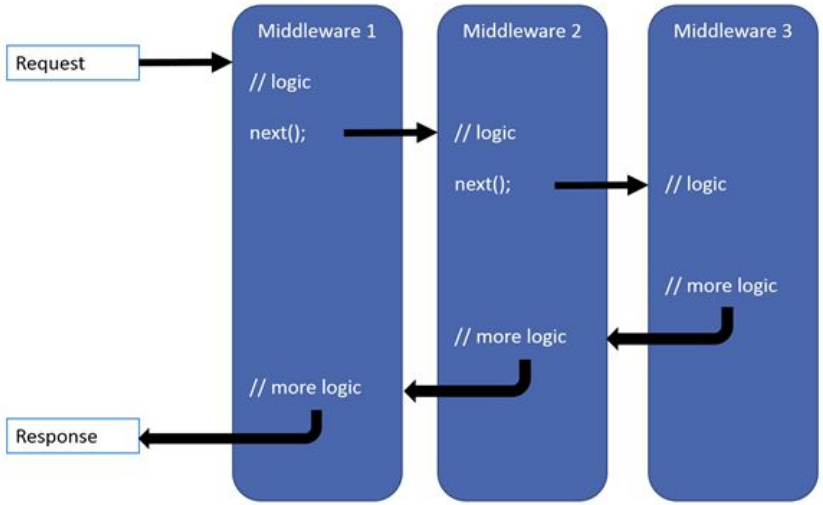
90 % | Проблемы не найдены. | Стр: 1 | Симв: 1 | Пробелы | CRLF

Метод **ConfigureServices()** регистрирует сервисы, которые используются приложением. После добавления в коллекцию сервисов добавленные сервисы становятся доступными для приложения.

Конвейер обработки запроса и middleware

Обработка запроса в ASP.NET Core устроена по принципу конвейера. Данные запроса по очереди получают компоненты в конвейере, которые называются **middleware**.

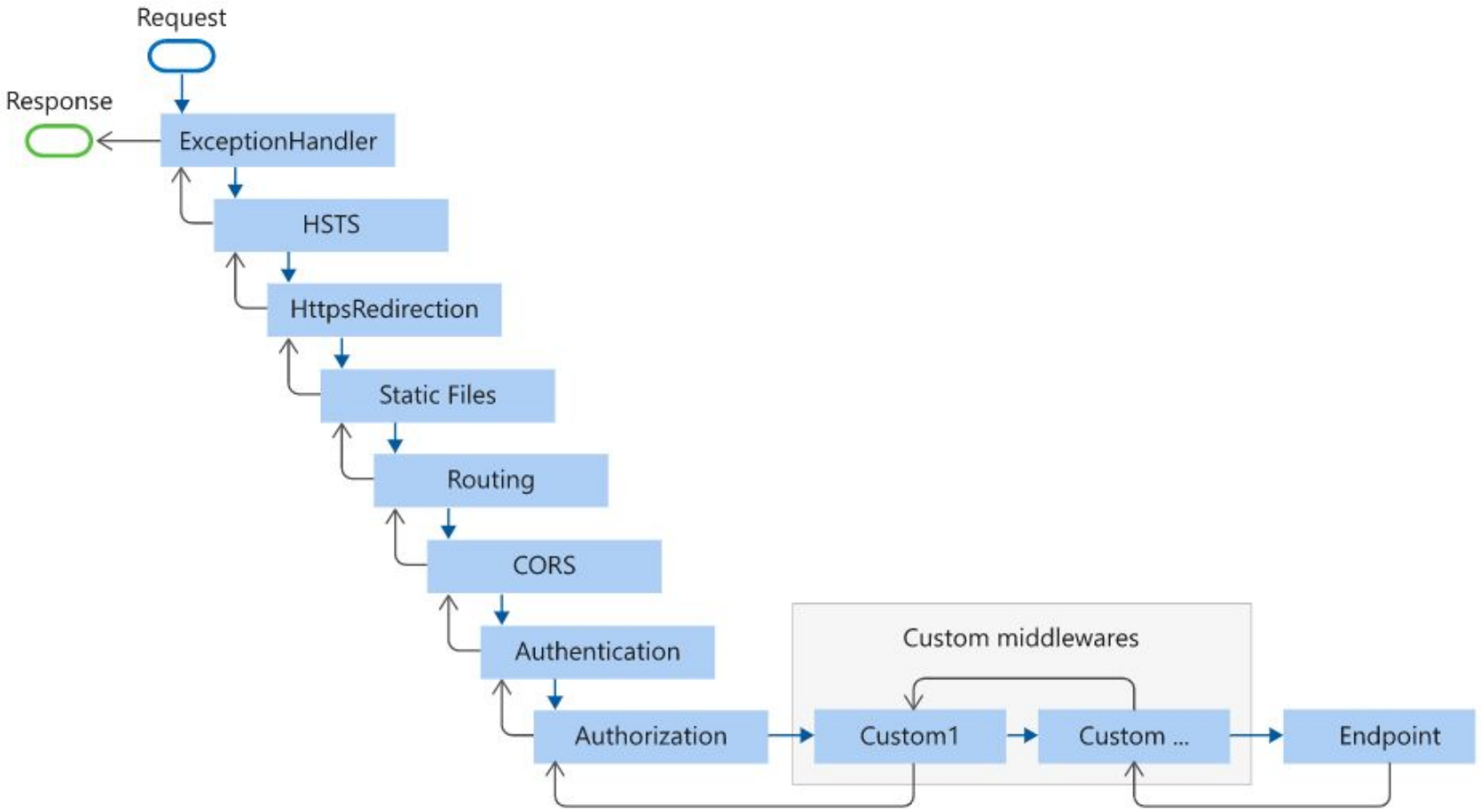
В ASP.NET Core для подключения компонентов middleware используется метод Configure из класса Startup.



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.

Порядок добавления компонентов промежуточного программного обеспечения в Startup.Configure методе определяет порядок, в котором компоненты промежуточного программного обеспечения вызываются по запросам, и обратный порядок ответа. Порядок имеет **решающее значение** для безопасности, производительности и функциональности.

Полный конвейер обработки запросов для приложений ASP.NET Core MVC и Razor Pages



Модели

Все сущности в приложении принято выделять в отдельные модели. В зависимости от поставленной задачи и сложности приложения можно выделить различное количество моделей.
Модели представляют собой простые классы и располагаются в проекте в каталоге *Models*. Модели описывают логику данных.

В приложении ASP.NET MVC Core модели можно разделить по степени применения на несколько групп:

- Модели, объекты которых хранятся в специальных хранилищах данных (например, в базах данных, файлах xml и т.д.)
- Модели, которые используются для передачи данных представление или наоборот, для получения данных из представления. Такие модели называются **моделями представления**
- Вспомогательные модели для промежуточных вычислений

POCO — это класс, который не прибит гвоздями к архитектуре какой-либо библиотеки. Программист сам волен выбирать иерархию классов (или отсутствие оной). Например, библиотека для работы с БД не будет заставлять наследовать "пользователя" от "сущности" или "активной записи". В идеале чистоты классов не нужны даже атрибуты.
Подобный подход развязывает руки программистам и позволяет строить удобную им архитектуру, использовать уже имеющиеся классы для работы со сторониими библиотеками и т. п. Впрочем, не обходится и без проблем, например, использование POCO может требовать магии во время выполнения: генерации унаследованных классов в памяти и т. п.
Примером POCO является любой класс, который не унаследован от специфического для некоторой библиотеки базового класса, не загромождён конвенциями и атрибутами, но который тем не менее может этой библиотекой полноценно использоваться.
DTO — это класс с данными, но без логики. Он используется для передачи данных между слоями приложения и между приложениями, для сериализации и аналогичных целей.
Примером DTO является любой класс, который содержит только поля и свойства. Он не должен содержать методов для получения и изменения данных.
https://habr.com/ru/post/268371/