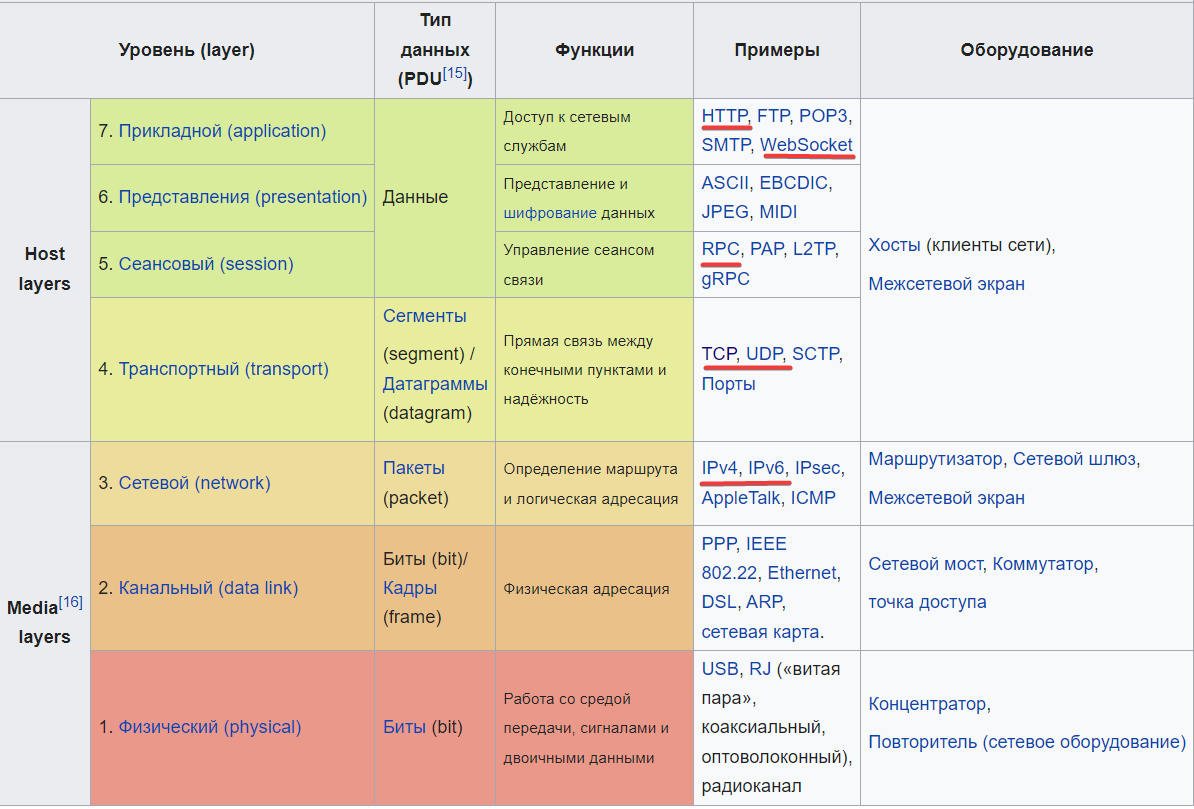
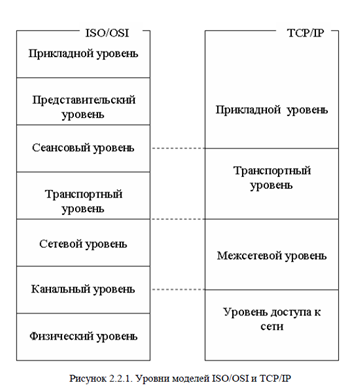
1. Интернет: определение, принцип устройства и организации, службы Интернет, понятие Интернет-сервера.



**OSI (Open Systems Interconnection)** — это модель, разработанная Международной организацией по стандартизации (ISO) для стандартизации взаимодействия различных сетевых систем.



1. **Сервер** – программное обеспечение, которое принимает и обрабатывает запросы.
2. Интернет – Всемирная компьютерная сеть, построенная на основе стека протоколов TCP/IP.
3. **Интернет** – 4 компонента:
   1. Стек протоколов TCP-IP — это основа Интернета (индустриальный стандарт, поддерживается всеми операционными системами).
   2. Интернет службы (DNS, DHCP, SMTP, POP3, FTPи т.д.).
   3. Документация в формате RFC и STD.
   4. Система организаций, которые поддерживают сеть (*в порядке убывания важности, как сказал Смелов*):
      * ISOC (развитие, внедрение и распространение новых интернет-технологий)
      * IETF (разрабатывает стандарты интернета в RFC, выпускает всю документацию)
      * IAB (занимается архитектурой)
      * ICANN (распространяет доменные имена)
      * IANA (регистрирует MIME)
      * W3C (WWW Consortium, некоммерческая организация для согласования стандартов) — организация разрабатывающая и внедряющая web-стандарты (HTTP, HTML, URI/URL, CSS, DOM, XML, PNG, SVG,…).
4. **Интернет-служба** – это программа, система, предоставляющая услуги клиентам. Сервер + протокол (протокол, который описывает, как работать с этим сервером). Стандартные серверы, которые прослушивают стандартные порты (*от 0 до 1024*).

**Основные сетевые протоколы**

1. **IP (Internet Protocol)**:
   * **Версии**: IPv4 и IPv6.
   * **Функции**: Адресация и маршрутизация пакетов данных от источника к назначению.
2. **TCP (Transmission Control Protocol)**:
   * **Функции**: Обеспечение надежной передачи данных, управление потоком и контроль ошибок. TCP гарантирует, что данные будут доставлены в правильном порядке и без потерь.
3. **UDP (User Datagram Protocol)**:
   * **Функции**: Обеспечение быстрой, но ненадежной передачи данных. Используется в приложениях, где важна скорость, а не надежность, например, в потоковой передаче аудио и видео.
4. **HTTP/HTTPS (HyperText Transfer Protocol / Secure HTTP)**:-80 443
   * **Функции**: Используются для передачи гипертекстовых документов в Интернете (веб-страниц). HTTPS обеспечивает защищенную передачу данных с использованием SSL/TLS.
5. **SMTP (Simple Mail Transfer Protocol)**:25
   * **Функции**: Протокол для отправки электронной почты.
6. **IMAP/POP3 (Internet Message Access Protocol / Post Office Protocol)**:143 110
   * **Функции**: Протоколы для получения электронной почты. IMAP позволяет работать с почтой на сервере, а POP3 загружает почту на локальное устройство.
7. **FTP (File Transfer Protocol)**:-21
   * **Функции**: Протокол для передачи файлов между клиентом и сервером.
8. **DNS (Domain Name System)**: 53
   * **Функции**: Преобразование доменных имен в IP-адреса, позволяя пользователям использовать удобные имена вместо цифровых адресов.
9. **DHCP (Dynamic Host Configuration Protocol)**:
   * **Функции**: Автоматическое назначение IP-адресов и других сетевых параметров устройствам в сети.

**RFC (ЭЙ-ЭФ-СИ)**: жизненный цикл описан в RFC 2026, этапы: Draft Internet, Proposed Standard, Draft Standard, Internet Standard (RFC/STD), Historic (вышедшие из употребления). RFC может содержать не только стандарты, но и концепции (Experimental – результаты экспериментов, Informational – информационные, Best Current Practice – опыт применения).

RFC — это серия документов, которые определяют стандарты, протоколы, методологии и инновации, связанные с работой сети Интернет (инет)

**MIME(МАЙМ)**: **Multipurpose Internet Mail Extensions** - многоцелевые расширения Internet-почты. Используется и как стандарт кодирования Internet-сообщений.

1. HTTP: определение, основные свойства, структура запроса, структура ответа, статусы ответа, заголовки (примеры).

**HTTP** – *полудуплексный* протокол *прикладного* уровня (используется для обмена данными между веб-сервером и клиентом).

Создатель HTTP и HTML – Тим Бёрнерс-Ли (Также он глава W3C).

Версии HTTP:

* 1. *0.9* (Работает поверх TCP. Только гипертекст (нельзя передавать файлы). Нет заголовков, status codes, полного URL. Methods: GET)
  2. *1.0* (Есть кеширование, аутентификация, заголовки (e.g. Content-Type), можно передавать медиа (фото, видео, аудио), есть коды ответа.   
     Нет постоянного соединения. Methods: GET, HEAD, POST)
  3. *1.1* (Действующая версия. Имеет постоянное соединение (Connection: keep-alive), есть SSL/TLS, chunked responses, content negotiation.)
  4. *2.0* (Не распространен. Есть параллельные запросы в рамках одного соединения и сжатие одинаковых заголовков. *Бинарный*: (это значит, что в 2.0 всё изначально закодировано и сжато, а в 1.1 изначально всё в строковом значении, а только потом сжимается))
  5. *3.0* (Не распространен. Главная особенность – универсальный транспортный протокол QUIC, работающий поверх UDP. Замена TCP)  
     <https://habr.com/ru/companies/sbermarket/articles/703048/>  
     <https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP>

1. **HTTP:** основные свойства

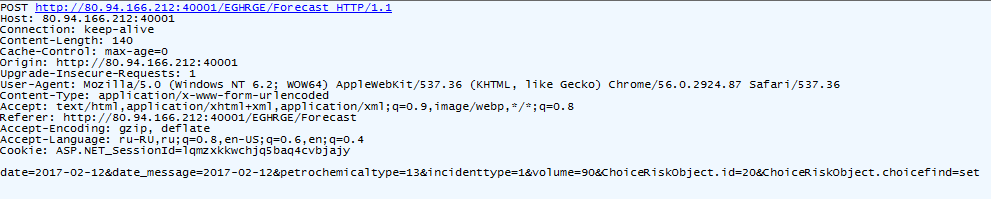
* версии HTTP/1.1 – действующий (текстовый), HTTP/2 – черновой (не распространен, бинарный);
* два типа абонентов: клиент и сервер;
* два типа сообщений: request и response;
* от клиента к серверу – request;
* от сервера к клиенту – response;
* на один request всегда один response, иначе ошибка;
* одному response всегда один request, иначе ошибка;
* TCP-порты: 80, 443;
* для адресации используется URI или URN;
* поддерживается W3C, описан в нескольких RFC.

1. **Request**:

* Метод (get..post..put..delete);
* URI;
* версия протокола (HTTP/1.1);
* заголовки (пары: имя/заголовок);
* параметры (пары: имя/заголовок);

Параметры могут передаваться в URL или в теле запроса, например:

* В URL: https://example.com/search?q=example
  + Параметр q с значением example.
* В теле запроса (для POST): { "name": "value" }
  + Параметр name с значением value.
* расширение.



1. **Response:**

* версия протокола (HTTP/1.1);
* код состояния (1xx, 2xx, 3xx, 4xx, 5xx);
* пояснение к коду состояния;
* заголовки (пары: имя/заголовок);
* расширение.

**Типы заголовков:**

* + General – общие для запросов и ответов (напр. Connection, Date)
  + Request (напр. Accept, Cookie, Host, Authorization)
  + Response (напр. Set-Cookie, Server)
  + Entity – для сущностей в запросах и ответах (напр. Content-Type, Content-Length, Content-Encoding: все, что начинается с “Content”)
  + User – заголовок, который не описан в протоколе HTTP   
    (должен начинаться с “x-header\_name”)

1. **Response: Код состояния:**

* **1xx**: информационные сообщения;
* **2xx**: успешный ответ;
* **3xx**: переадресация;
* **4xx**: ошибка клиента;
* **5xx**: ошибка сервера.

- 100 — Продолжить

- 101 — Переключение протоколов

- 200 — OK (Успешно)

- 201 — Создано

- 202 — Принято

- 203 — Информация не авторитетна

- 204 — Нет содержимого

- 205 — Сбросить содержимое

- 206 — Частичное содержимое

- 300 — Множественный выбор

- 301 — Перемещено навсегда

- 302 — Найдено

- 303 — Смотреть другое

- 304 — Не изменялось

- 305 — Использовать прокси

- 307 — Временное перенаправление

- 400 — Неверный запрос

- 401 — Неавторизован

- 402 — Требуется оплата

- 403 — Запрещено

- 404 — Не найдено

- 405 — Метод не разрешен

- 406 — Не приемлемо

- 407 — Требуется аутентификация прокси

- 408 — Время ожидания запроса истекло

- 409 — Конфликт

- 410 — Удалено

- 411 — Требуется длина

- 412 — Предусловие не выполнено

- 413 — Слишком большое содержимое запроса

- 414 — URI запроса слишком длинный

- 415 — Неподдерживаемый тип медиа

- 416 — Запрошенный диапазон недоступен

- 417 — Ожидание не выполнено

- 500 — Внутренняя ошибка сервера

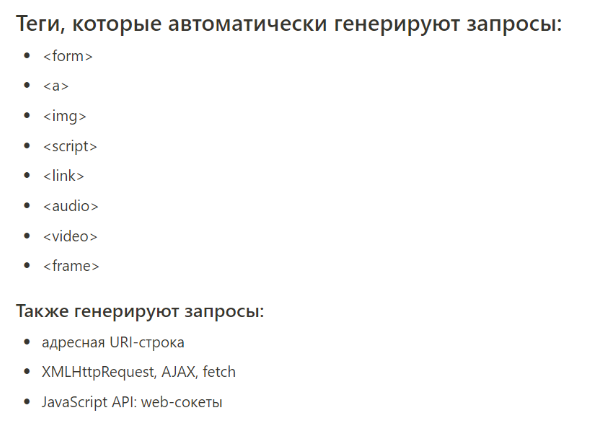
- 501 — Не реализовано

- 502 — Плохой шлюз

- 503 — Сервис недоступен

- 504 — Время ожидания шлюза истекло

- 505 — Версия HTTP не поддерживается



1. Web-приложение: определение, структура, общая схема web-приложения.

**Internet-ресурс:** сущность в сети Internet, имеющая адрес (опубликованная в Internet сущность).

**Программирование в Internet:** разработка распределенного (сетевого) приложения, представляющего собой один или несколько связанных Internet-ресурсов. Сайт – частный случай приложения, Вычислительное облако – частный случай приложения.

**Программная пратформа – это набор прог библиотек, правил работы с доку-цией и инструментов**

* **LAMP** (Linux, Apache, MySQL, PHP/Perl, кроссплатформенная технология);
* **Java EE (кроссплатформенная технология на основе Java);**
* **ASP.NET** (ASP.NET CORE – кроссплатформенная версия);
* **NODE.JS** (кроссплатформенная технология, на основе JavaScript);
* **Ruby on Rails**(кроссплатформенная технология на основе Ruby и фреймворка MVC для web-разработки);
* **Python Django** (кроссплатформенная технология на основе Python и фреймворка MVC для web-разработки).
* **Web-приложение:** клиент-серверное приложение, применяющее для обмена данными протокол HTTP; может быть просто web-приложением (HTML+HTTP) или web-службой (API, HTTP-транспорт, формат XML, JSON).
* **Порт** — программа на сервере, которая прослушивает входящие сообщения.
* **Web-программирование:** разработка клиент-серверных приложений, компоненты которого взаимодействуют по протоколу HTTP-протокола; web-программирование – частный случай программирования в Internet.
* **Архитектура web-приложения**:



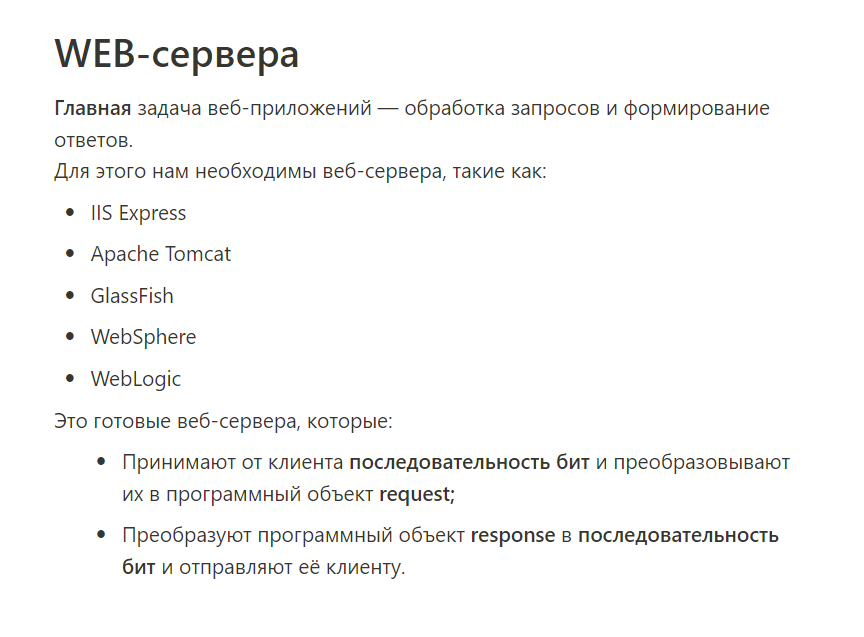








1. **URI: Uniform Resource Identifier** – унифицированный идентификатор ресурса (документ, изображение, файл, служба, электронная почта,…).
2. **URL: Uniform Resource Location** - унифицированный локатор ресурса , содержащий местонахождение ресурса и способ обращения (протокол) к ресурса, описывает множество URI.
3. **URN: Uniform Resource Name** - унифицированное имя ресурса – URI, имя ресурса, не содержащее месторасположение и способ доступа к ресурсу. В будущем URN должен заменить URL (для решения проблем с перемещением ресурсов в Internet).



1. ASP.NET CORE: назначение, основные свойства, кроссплатформенность, общая структура приложения, маршрутизация, типы приложений ASP.NET CORE, программный менеджер NuGet, встроенные компоненты.

**ASP.NET CORE:** программная платформа для разработки web-приложений**.**

**ASP.NET CORE:** кроссплатформенный фреймворк (Windows, Linux, Mac OS) с открытым кодом для разработки web-приложений на платформе .NET.

Свойства:

- является фреймворк с открытым кодом (доступен для публичного просмотра, изменения и распространения)

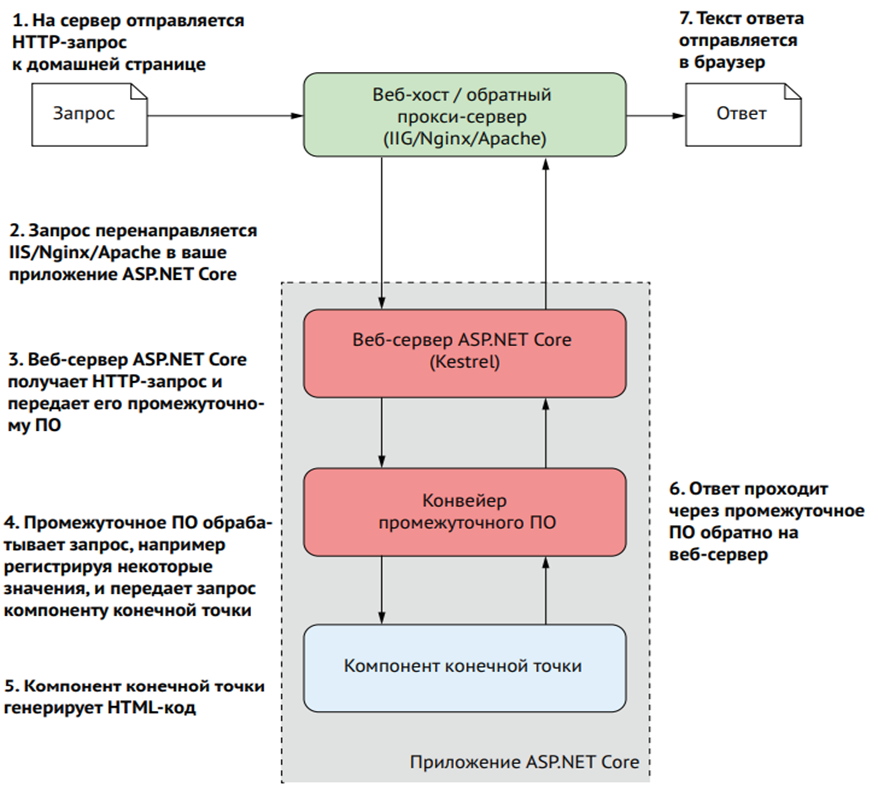
- может работать над .NET Core и над полной .NET.

- приложения могут работать под IIS (Веб-сервер Microsoft) или под web-сервером Kestrel (Кроссплатформенный веб-сервер, встроенный в ASP.NET Core.).

- благодаря модульности, все отдельные компоненты загружаются через Nuget, нет необходимости применять библиотеку System.Web.dll (основная библиотека для приложений ASP.NET)

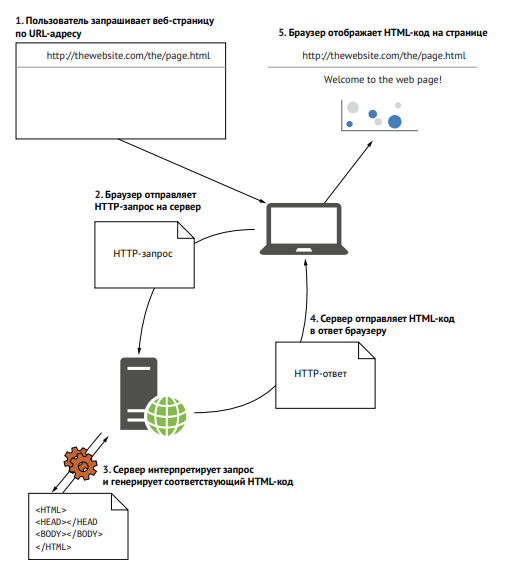
Кроссплатформенность: Она достигается за счёт существования разных версий этой платформы для разных ОС. Она состоит из CoreCLR + библиотеки CoreFX и эти 2 компонента машинозависимы и существуют для разных ОС.

Общая структура приложения:

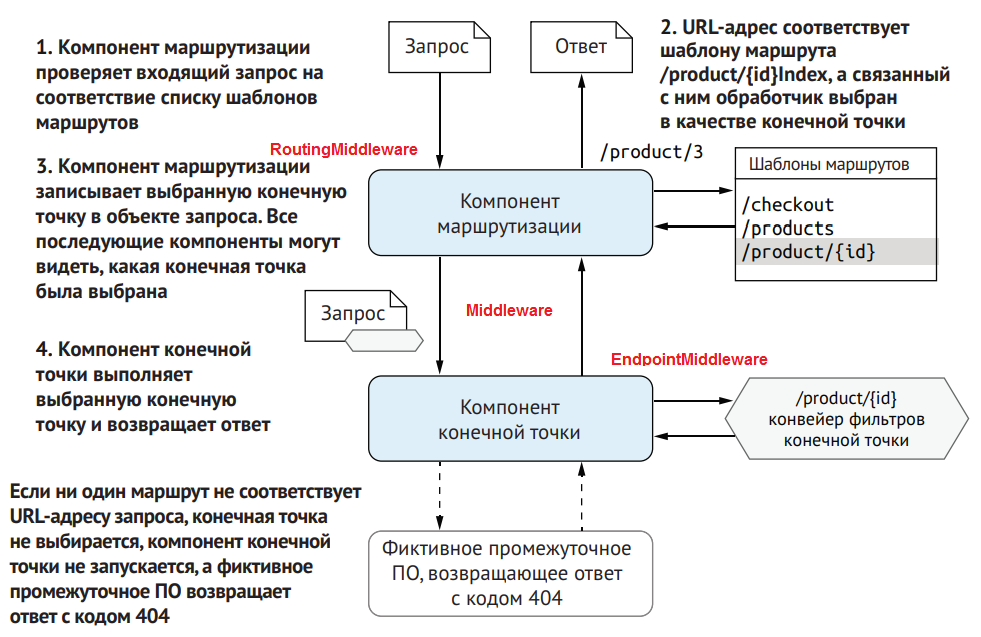


### Общая структура приложения на ASP.NET Core:

1. **HTTP-запрос от клиента**:
   * Пользователь отправляет HTTP-запрос через веб-браузер к веб-серверу. Запрос может быть на получение страницы, данных или выполнения какого-либо действия.
2. **Веб-хост / обратный прокси-сервер**:
   * Запрос поступает на веб-хост, такой как IIS, Nginx или Apache. Эти сервера могут обрабатывать запросы и передавать их дальше в приложение.
3. **Веб-сервер ASP.NET Core (Kestrel)**:
   * Веб-сервер ASP.NET Core (Kestrel) принимает HTTP-запросы от обратного прокси-сервера. Kestrel — это легковесный, кроссплатформенный веб-сервер, встроенный в ASP.NET Core.
4. **Конвейер промежуточного программного обеспечения (middleware)**:
   * Запрос проходит через конвейер middleware, где выполняется обработка. Middleware может включать компоненты для аутентификации, авторизации, обработки ошибок, маршрутизации и другие. Каждый компонент может изменять запрос или ответ, а также передавать управление следующему элементу в цепочке.
5. **Компонент конечной точки**:
   * После обработки middleware запрос передается к конечной точке приложения, такой как контроллер или обработчик. Здесь происходит основная логика обработки запроса, например, получение данных из базы.
6. **Ответ**:
   * После обработки запрос формируется ответ, который отправляется обратно через конвейер middleware, затем обратно в Kestrel, а затем веб-хост. В итоге текст ответа отправляется обратно в браузер пользователя.
7. цикл обработки запроса



**Маршрутизация** - механизм выбора обработчика HTTP-запроса



**Типы приложений ASP NET Core**:  MVC UI, Web API, Web Pages.

**NuGet** – программный менеджер для установки компонентов

приложения оформленных в виде пакетов

Встроенные компоненты:

1. Middleware(компоненты, которые обрабатывают HTTP-запросы и формируют ответы, формируя конвейер обработки запросов. )

2. Система маршрутизации (Routing)

3. Контроллеры и MVC

4. Razor Pages

5.  Dependency Injection(Встроенный контейнер сервисов для регистрации и получения зависимостей )

6. Логирование(Встроенная система логирования )

1. ASP.NET CORE: проект, назначение файлов проекта: lanchsetting.json, appsettings.json, Program.cs. Основные настраиваемые (builder) компоненты. Объект HttpContext. Директорий wwwroot.

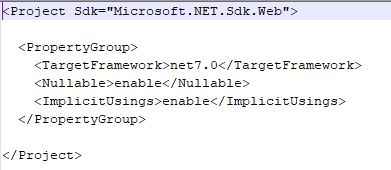
**ASP.NET CORE:** приложениешаблона ASP.NET Core Empty

**launchSettings.json** - как запустить (run) проект

**appsettings.json** -конфигурация в режиме выполнения;

**Program**.cs:Этот файл содержит точку входа в приложение. Он отвечает за конфигурацию и запуск веб-сервера.

**.csproj:**



Хранит в себе информацию о проекте, выходной тип (TargetType), использующиеся пакеты, сборки и ссылки на проекты, использующуюся версию .NET.

### launchSettings.json

"profiles": {

"IIS Express": {

"commandName": "IISExpress",

"launchBrowser": true,

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

},

"MyApp": {

"commandName": "Project",

"launchBrowser": true,

"applicationUrl": "http://localhost:5000",

"environmentVariables": {

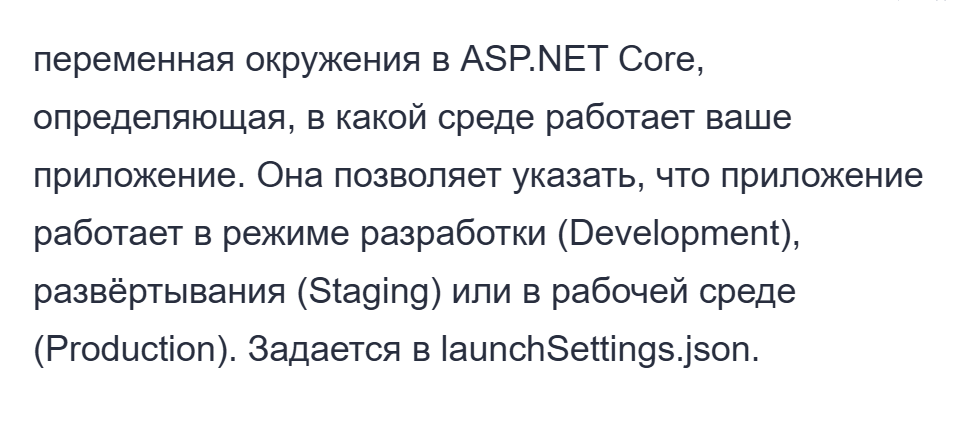
"ASPNETCORE\_ENVIRONMENT": "Development"

}

}

}

1. **commandName**:
   * Определяет, как будет запущено приложение (Project – как проект .net, IISExpres - Запускает приложение с использованием IIS Express, Executable -как исполняемый файл).
2. **launchBrowser**:
   * Указывает, нужно ли автоматически открывать браузер при запуске.
3. **applicationUrl**:
   * URL-адрес, по которому будет доступно приложение.
4. **environmentVariables**:
   * Переменные среды, которые будут доступны во время выполнения приложения. Например, ASPNETCORE\_ENVIRONMENT определяет режим (Development, Staging, Production).



### appsettings.json

#### Основные элементы

1. **Конфигурационные параметры**:
   * Определяет различные настройки, такие как строки подключения, параметры логирования и другие настройки приложения.

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MyAppDb;Trusted\_Connection=True;"

},

"AllowedHosts": "\*"

}

1. **Logging**:
   * Настройки для системы логирования, включая уровни логирования (Information, Warning и т.д.).
2. **ConnectionStrings**:
   * Строки подключения к базам данных, которые могут использоваться в приложении.
3. **AllowedHosts**:
   * Указывает, какие хосты разрешены для доступа к приложению.

#### Основные настраиваемые (builder) компоненты

При настройке приложения ASP.NET Core можно использовать различные компоненты:

* **WebHostBuilder**: Настраивает веб-сервер и его параметры.
* **IServiceCollection**: Позволяет регистрировать зависимости для внедрения (Dependency Injection).
* **IConfiguration**: Управляет конфигурационными настройками приложения.
* **ILoggerFactory**: Настраивает логирование.
* **ASP.NET CORE: WebApplicationBuilder:** конфигурирование, журналирование, сервисы, **Kestrel**.**(Этот объект настраивает всю конфигурацию приложения, его маршруты, используемые зависимости и т.д)**

**HttpContext**  объект, создаваемый HTTP-сервером, содержит исчерпывающую информацию о соединении, запросе, будущем ответе:

**wwwroot** – корневой каталог хранения статических файлов сайта по умолчанию(html,css,js,)

1. **ASP.NET CORE: WebApplication:** резидентный объект (ядро приложения, содержащее http-сервер **Kestrel**); параметризация на основе паттерна builder (экземпляр **WebApplicationBuilder**);

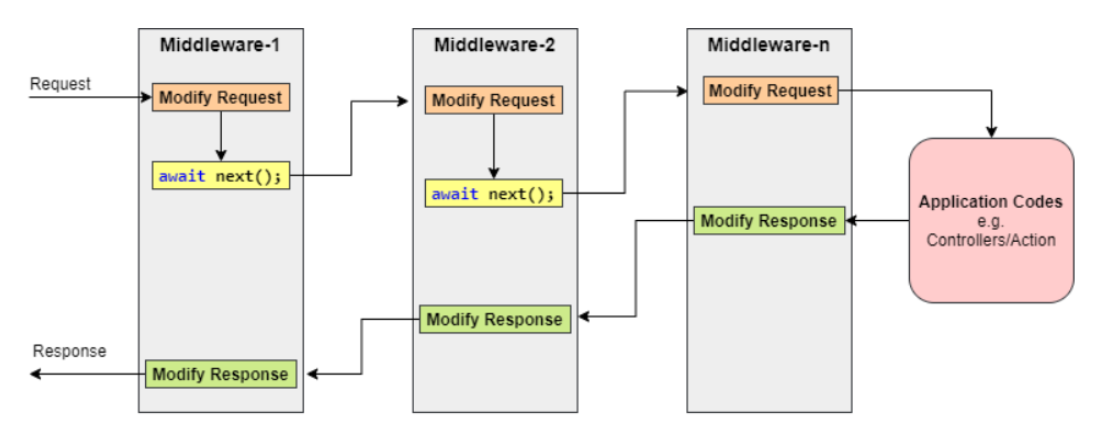
экземпляр создаётся методом **WebApplicationBuilder.Build();**

приложение запускается методом **WebApplication.Run().**

1. **ASP.NET CORE: WebApplication.UseXXX:** подключение middleware-элементов.
2. **ASP.NET CORE: WebApplication.Environment:** переменные окружения (**ASPNETCORE\_ENVIRONMENT**).
3. **ASP.NET CORE: WebApplicationBuilder:** конфигурирование, журналирование, сервисы, **Kestrel**.В предыдущих версиях: классы Startup, IHostBuilder, IHost.
4. **ASP.NET CORE: Program.cs:** протоколирование (журналирование), конечная точка, обработчик запросов, маршрутизация.
5. ASP.NET CORE: middleware: определение, назначение, принцип устройства, встроенные элементы. Принцип разработки пользовательского элемента. Встроенные компоненты платформы, подключаемые как middleware –элементы.

1)цепочка программных объектов, предназначенных для последовательной обработки запросов и ответов, реализация паттерна Chain of Responsibility  
2)конвейер обработки запросов и ответов

Например, у нас может быть middleware-компонент для аутентификации пользователя, middleware-компонент для обработки ошибок и еще один middleware-компонент для обслуживания статических файлов



1. **ASP.NET CORE: Middleware:** Middleware**-**объект может одно из:

*на прямом пути*

1. получить request, сгенерировать response и передать его предшествующему Middleware**-**объекту или http-серверу;
2. получить request,изменить request и передать его следующему Middleware**-**объекту;

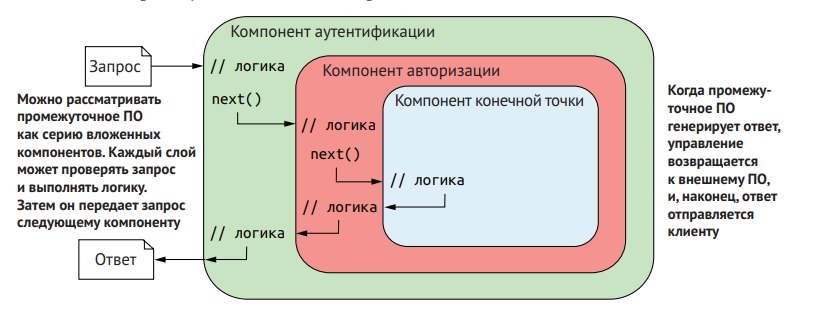
*на обратном пути*

1. получить исходящий response, изменить response, передать его следующему Middleware**-**объекту или http-серверу;

Принцип устройства Middleware

* Middleware вызывается последовательно в порядке регистрации в конвейере.
* Каждый компонент получает контекст запроса (HttpContext).
* Middleware может:
  + Выполнить свои действия **до** вызова следующего (например, логировать запрос).
  + Вызвать следующий middleware.
  + Выполнить действия **после** вызова следующего (например, логировать ответ).
  + Прервать конвейер и вернуть ответ сразу.

1. **ASP.NET CORE: Middleware: *app*.Use*XXXXX*()**– имена Middleware-элнмент: **app.UseHttpLogging()**.
2. **ASP.NET CORE: Middleware:** класс **EndpointMiddleware** (конечная точка): **app.MapGet( *endpoint* ), app.MapPost( *endpoint*** ), …
3. **ASP.NET CORE: Middleware:** типичное применение: выполнение при каждом запросе (сквозная задача приложения): журналирование, аутентификация, авторизация, маршрутизация запросов(**app.Map*XXX,*** MVC**-**паттерн**), …**
4. **ASP.NET CORE: Middleware:** данные о запросе и ответе упакованы в экземпляре **HttpContext**.
5. **ASP.NET CORE: Middleware:** встроенные **Middleware**-элементы (UseHttpLogging, UseStaticFiles, …), пользовательские Middleware-элементы (экземпляры классов, производных от класса UseMiddleware).
6. **ASP.NET CORE: Middleware:** request может не дойти по middleware-цепочке до конечной точки. Например, при аутентификации и авторизации.
7. **ASP.NET CORE: Middleware:** элемент, замыкающий конвейер и возвращающий ответ, называется терминальным Middleware**-** элементом.
8. **ASP.NET CORE: Middleware:** порядок определения Middleware**-** элементом имеет смысл.
9. **ASP.NET CORE: Middleware:** конвейер - последовательность вложенных вызовов Middleware**-**элементов.

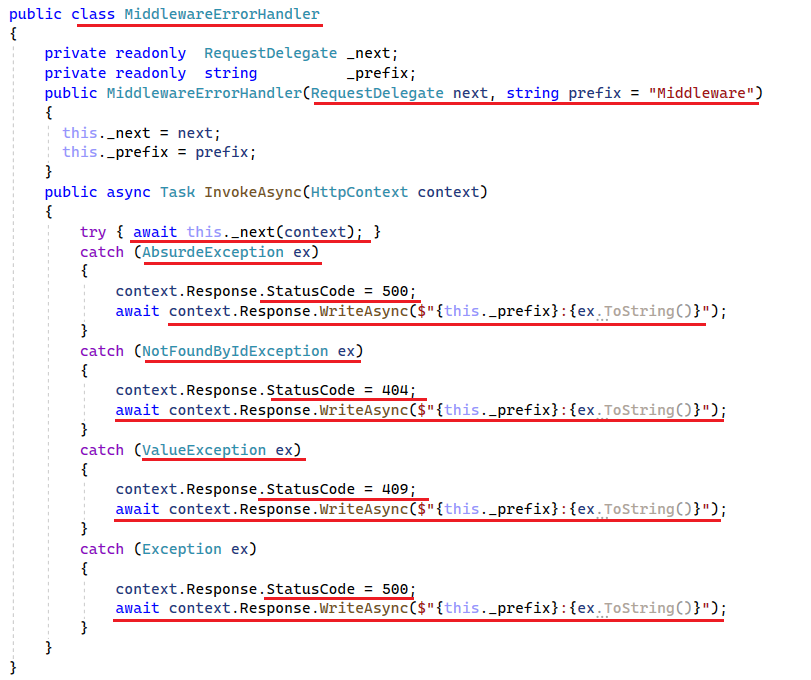
****

#### Встроенные элементы

ASP.NET Core предоставляет множество встроенных Middleware-элементов, включая:

* **UseStaticFiles**: Обработка статических файлов (CSS, JS, изображения).
* **UseRouting**: Настройка маршрутизации для обработки запросов.
* **UseAuthentication**: Аутентификация пользователей.
* **UseAuthorization**: Авторизация пользователей.
* **UseHttpLogging**: Журналирование HTTP-запросов.

#### Принцип разработки пользовательского элемента

:

Создать класс с методом Invoke или InvokeAsync, принимающим HttpContext и возвращающим Task.

1. В методе реализовать логику обработки запроса.
2. Для передачи управления следующему middleware вызвать await \_next(context).



#### Встроенные компоненты платформы,подключаемые как middleware –элементы.

Встроенные компонент платформы(то, что уже реализовано заранее и не требует написания кода для подключения), подключаемые как middleware –элементы:

### 1. UseRouting()

* **Описание**: Активирует маршрутизацию, позволяя сопоставлять HTTP-запросы с конечными точками (контроллерами и действиями).

### 2. UseStaticFiles()

* **Описание**: Обслуживает статические файлы (например, CSS, JavaScript, изображения) из директории wwwroot, позволяя пользователям загружать эти ресурсы.

### 3. UseExceptionHandler()

* **Описание**: Обрабатывает исключения, перенаправляя пользователей на страницу ошибок при возникновении необработанных исключений.

### 4. UseCors()

* **Описание**: Включает поддержку CORS, позволяя контролировать, какие домены могут делать запросы к вашему приложению, что особенно важно для API.

1. ASP.NET CORE: статические файлы: назначение, применение, скачивание статических файлов.

#### Применение

Статические файлы обычно используются для:

* **Предоставления стилей**: CSS-файлы для оформления веб-страниц.
* **Интерактивности**: JavaScript-файлы, которые добавляют функциональность на клиентской стороне.
* **Изображений и медиа**: PNG, JPEG, GIF и другие форматы изображений, а также видео и аудио файлы.
* **Шрифтов**: Загружаемые шрифты для улучшения типографики.

#### Скачивание статических файлов

Статические файлы — файлы, которые оправляются клиенту без изменений и обработки со стороны сервера(html,css,js,изображения и документы).

**UseStaticFiles:**

### Использование UseStaticFiles для настройки статических файлов в ASP.NET Core

В ASP.NET Core вы можете использовать UseStaticFiles для обслуживания статических файлов из различных директорий. Вот как можно настроить метод для обслуживания файлов из корневого каталога или других каталогов.

#### 1. Обслуживание файлов из корневого каталога

По умолчанию статические файлы обслуживаются из каталога wwwroot. Если вы хотите настроить другое имя каталога, вы можете использовать метод UseWebRoot.

public class Program

{

public static void Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

// Установка другого корневого каталога

builder.WebHost.UseWebRoot("CustomRoot");

var app = builder.Build();

// Обслуживание статических файлов

app.UseStaticFiles();

app.Run();

}

}

#### 2. Обслуживание файлов из других директорий

Если вы хотите обслуживать статические файлы из других каталогов, можете использовать StaticFileOptions. Вот пример, как это сделать:

app.UseStaticFiles(new StaticFileOptions

{

FileProvider = new PhysicalFileProvider(

Path.Combine(builder.Environment.ContentRootPath, "MyStaticFiles")),

RequestPath = "/StaticFiles"

});

### Объяснение кода

* **FileProvider**: Указывает, откуда будут загружаться статические файлы. В данном случае файлы загружаются из папки MyStaticFiles, находящейся в корневом каталоге приложения.
* **RequestPath**: Определяет путь, по которому файлы будут доступны через HTTP. В примере файлы будут доступны по URL http://yourdomain/StaticFiles/yourfile.ext.

метод, который в случае отсутствия параметров помечает файлы в корневом каталоге(по умолчанию wwwroot, но можно изменить с помощью UseWebRoot) как обслуживаемые. Также с помощью параметров типа StaticFileOptions можно указать другие директории(не относящиеся к корневой), например:  
app.UseStaticFiles(new StaticFileOptions

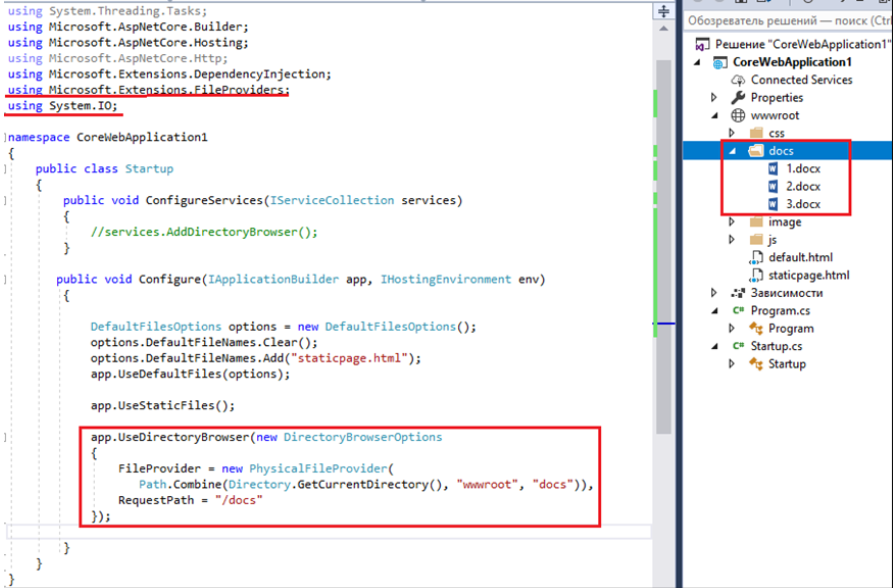
{

    FileProvider = new PhysicalFileProvider(

           Path.Combine(builder.Environment.ContentRootPath, "MyStaticFiles")),

    RequestPath = "/StaticFiles"

});



### Настройка UseDirectoryBrowser()

1. **Подключите необходимые middleware**:
   * Включите UseStaticFiles() для обслуживания статических файлов.
   * Используйте UseDirectoryBrowser() для включения просмотра содержимого каталога.

public void Configure(IApplicationBuilder app)

{

// Указываем путь к статическим файлам

app.UseStaticFiles();

// Включаем просмотр содержимого каталога docs

app.UseDirectoryBrowser(new DirectoryBrowserOptions

{

FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), "docs")),

RequestPath = "/docs"

});

// Обработка запросов на скачивание файлов

app.Map("/download/{\*filename}", appBuilder =>

{

appBuilder.Run(async context =>

{

var filename = context.Request.RouteValues["filename"].ToString();

var filePath = Path.Combine(Directory.GetCurrentDirectory(), "docs", filename);

var mime = "application/octet-stream"; // По умолчанию

if (System.IO.File.Exists(filePath))

{

context.Response.Headers.Add("Content-Disposition", $"attachment; filename=\"{filename}\"");

await context.Response.SendFileAsync(filePath);

}

else

{

context.Response.StatusCode = 404;

await context.Response.WriteAsync("File not found.");

}

});

});

}

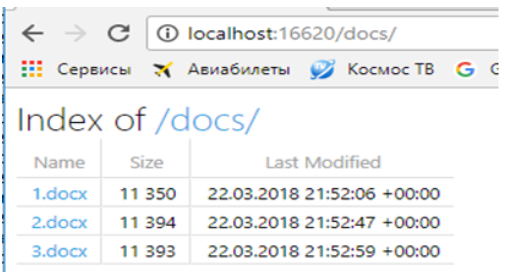
### Объяснение кода

1. **UseStaticFiles()**:
   * Позволяет обслуживать статические файлы из каталога wwwroot.
2. **UseDirectoryBrowser()**:
   * Позволяет пользователям просматривать содержимое каталога docs по URL /docs.
3. **Map("/download/{\*filename}", ...)**:
   * Настраивает маршрут для скачивания файлов. {\*filename} позволяет захватывать имя файла из URL.
4. **Обработка запроса**:
   * Проверяет, существует ли файл по указанному пути.
   * Если файл существует, добавляет заголовок Content-Disposition, указывая, что файл должен быть загружен с указанным именем.
   * Использует SendFileAsync() для отправки файла клиенту.
   * Если файл не найден, возвращает статус 404 и сообщение об ошибке.-====================

Используется метод UseDirectoryBrowser(), который позволяет

пользователю просматривать содержимое каталога, в данной реализации

мы прописываем каталог docs и после перехода по соотв. url пользователь видит следующее



По нажатию на файл должны срабатывать следующие строки кода:

 context.Response.Headers.Add("Content-Disposition", $"attachment; filename=\"{filename}\"");

                return Results.File(filePath, mime);

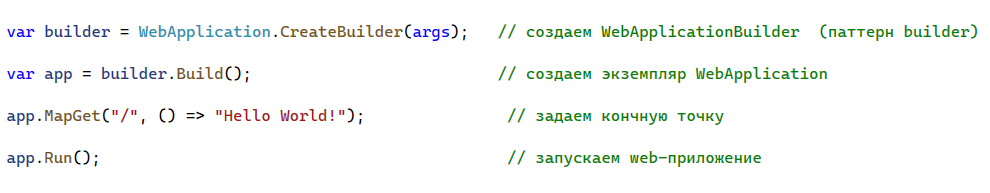
Добавляем в заголовки информацию о  том, что тело будет содержат

ь файл с названием, и в само тело добавляем файл.

1. ASP.NET CORE: приложение miniAPI: структура приложения, конечные точки MapXXX, UseExceptionHandler, Fallback-обработчик, обработка параметров и входных данных, заголовки Content-Type, Accept. Маршрутизация запросов с помощью MapGroup.

miniAPI – способ написания простейших веб-приложений, без использования контроллеров, атрибутов [HttpGet] [HttpPost] и т.д. В нём используются готовые методы MapXXX, а реализация логики обработки запросов выполняется с помощью лямба-выражений.

Структура приложения miniAPI:



1. **var builder = WebApplication.CreateBuilder(args);**
   * Создаёт экземпляр WebApplicationBuilder, который используется для конфигурации приложения.
2. **Регистрация служб**
   * Можно регистрировать необходимые сервисы в контейнере зависимостей.
3. **var app = builder.Build();**
   * Создаёт экземпляр WebApplication, который будет использоваться для определения маршрутов и запуска приложения.
4. **Методы MapGet, MapPost, MapPut, MapDelete**
   * Определяют маршруты и логику обработки запросов. Лямбда-выражения реализуют необходимую логику прямо в месте определения маршрута.
5. **app.Run();**
   * Запускает приложение и начинает прослушивание входящих HTTP-запросов.

MiniApi использу.т конечные точки:

MapGet

MapPost

MapPut

MapDelete

Для определения  маршрутов и обработки http запросов

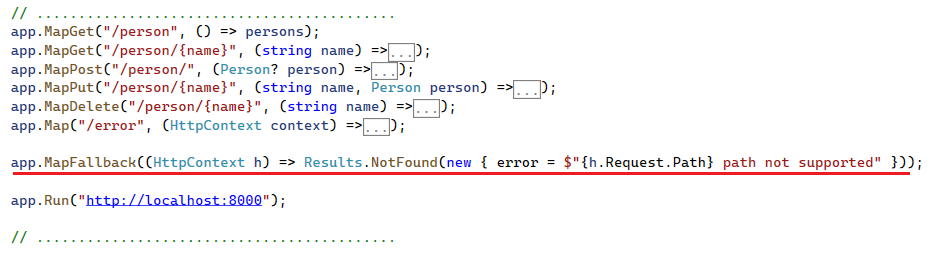
Синтаксис: app.MapXXX(метод)(“путь”, делегат/функция) - создание конечной точки. Возвращает RouteHandlerBuilder для дальнейшей настройки

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

UseExceptionHandler: В случаях, когда необходимо настроить собственный обработчик ошибок, можно воспользоваться этим методом. Он подключает ExceptionHandlerMiddleware, который, в свою очередь, «ловит» и логирует необработанных исключения, а также перенаправляет при возникновении исключения на некоторый адрес. Например : // если приложение не находится в процессе разработки // перенаправляем по адресу "/error"

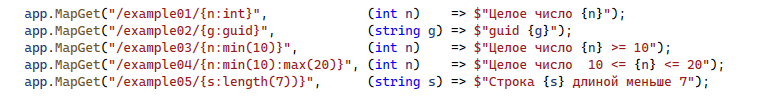
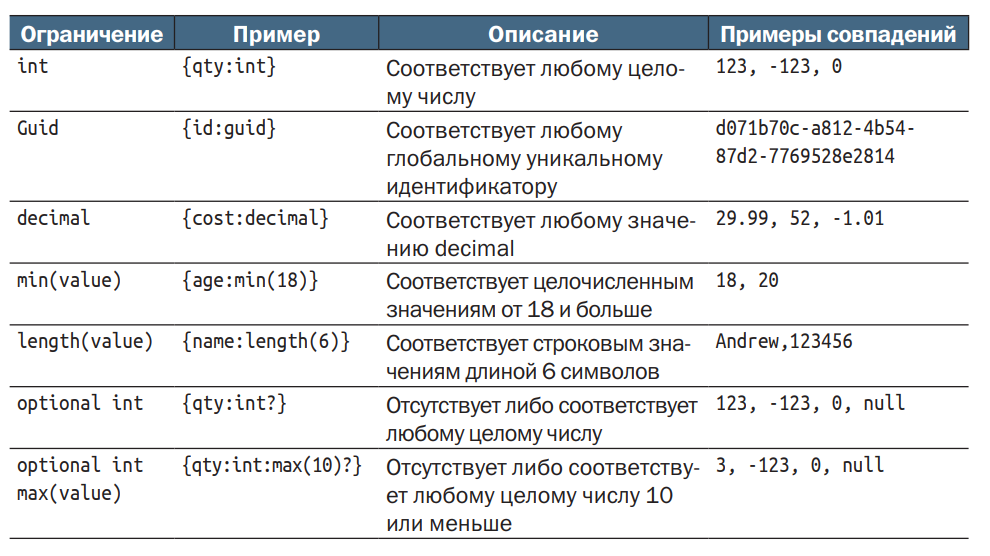
if (!app.Environment.IsDevelopment()) { app.UseExceptionHandler("/Error"); } // middleware, которое обрабатывает исключение

 app.Map("/error", app => app.Run(async context => { context.Response.StatusCode = 500; await context.Response.WriteAsync("Error 500. DivideByZeroException occurred!"); }));

Fallback-обработчик: 

Обработчик неподходящих ни под один обработчик http запросов. Cрабатывает при получении сервером запроса, который не может быть обработан существующими обработчиками.

Обработка параметров и входных данных:



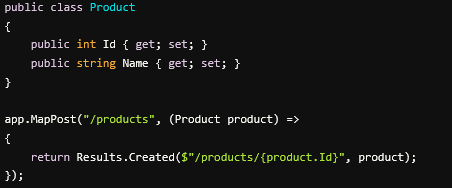
Обработка параметров из URL

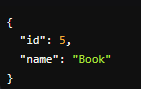
Обработка параметров из Query(часть URL после знака вопроса):



/search?q=текст

Обработка параметров из Body:



 - тело запроса

Json-объект будет автоматически десериализован в объект Product

Если нужно явно указать источник - можно использовать теги из Mvc(FromRoute], [FromQuery], [FromHeader(Name=XXX)], [FromService])

**Заголовки content-type и accept**

content-type это MIME-тип ресурса в request body или response body. *(Entity header)*

Accept - указывает  какой тип данных клиент ожидает получить(request-header)

**Машрутизация запросов MapGroup**

Это способ группировать несколько маршрутов(конечных точек) с одним префиксом и общими настройками Инициализация группы:



celebrities.MapGet("/{id:int:min(1)}", (IRepository repo, int id) =>

         {

             …

         });

таким образом, этот mapget будет доступен по localhost:5000/api/celebrities/3

Для каждой из группы можно указать свой middleware, RequireCors, отдельная обработка параметров и т.д.

#### ASP.NET CORE: приложение miniAPI: фильтры конечных точек, обработка параметров (GetArgument), делегат next. Фильтры конечных точек

Фильтры в ASP.NET Core представляют собой серверные объекты, которые обрабатывают HTTP-запросы и могут выполнять предварительную обработку объектов Request и Response. Они позволяют организовать цепочку обработки запросов, где каждый фильтр может модифицировать запрос или ответ, а также принимать решения о продолжении обработки.

**Основные характеристики фильтров**

1. **Цепочка фильтров**:
   * Фильтры могут быть организованы в цепочку, где каждый фильтр обрабатывает запрос и может передавать его дальше или прерывать цепочку.
   * Последний элемент в цепочке — это сам ресурс (например, контроллер или конечная точка).
2. **Прерывание цепочки**:
   * Фильтр может прервать дальнейшую обработку запроса и самостоятельно сформировать ответ клиенту. Это позволяет, например, возвращать ошибки или редиректы без обращения к основному ресурсу.
3. **Многоразовое использование**:
   * Один и тот же фильтр может быть применен к нескольким ресурсам, что делает код более модульным и повторно используемым.
4. **Параметры фильтра**:
   * Фильтры получают объекты Request и Response, что позволяет им взаимодействовать с данными запроса и формировать ответ.

Фильтры конечных точек в MiniAPI позволяют добавлять дополнительную логику перед или после обработки запроса. Это может быть полезно для аутентификации, авторизации, логирования и других задач.

Пример применения фильтров:

csharp

Copy

app.MapGet("/api/items", [Authorize] () =>

{

return new List<string> { "Item1", "Item2" };

});

В этом примере фильтр [Authorize] используется для проверки аутентификации пользователя перед выполнением конечной точки.

**Обработка параметров с помощью GetArgument**

При обработке запросов в MiniAPI можно использовать метод GetArgument, чтобы получить параметры, переданные в конечную точку. Это позволяет извлекать данные из URL, тела запроса или заголовков.

Пример:

csharp

Copy

app.MapGet("/api/items/{id}", (int id) =>

{

// Логика для получения элемента по ID

return $"Item with ID: {id}";

});

Здесь параметр id будет автоматически извлечен из URL.

**Делегат next**

Делегат next представляет следующий элемент в конвейере обработки запросов. Он используется для передачи управления следующему Middleware или конечной точке после выполнения определенной логики.

Пример использования делегата next:

csharp

Copy

app.MapGet("/api/items", async (HttpContext context, RequestDelegate next) =>

{

// Логика перед передачей управления

await next(context); // Передача управления следующему элементу

// Логика после обработки ответа

});

В этом примере сначала выполняется логика перед вызовом следующего элемента, а затем выполняется логика после получения ответа.

1. ASP.NET CORE: приложение miniAPI: скачивание файлов.

В ASP.NET Core MiniAPI можно легко реализовать функциональность скачивания файлов. Для этого используется метод File, который позволяет отправлять файлы клиенту с соответствующими заголовками.

#### Шаги для реализации скачивания файлов

1. **Создание папки для файлов**:
   * Создайте папку в вашем проекте, например, wwwroot/files, где будут храниться файлы для скачивания.
2. **Настройка Middleware для статических файлов**:
   * Убедитесь, что вы включили Middleware для обслуживания статических файлов в Program.cs:

csharp

Copy

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.UseStaticFiles(); // Обслуживание статических файлов

1. **Определение конечной точки для скачивания файлов**:
   * Используйте метод MapGet для создания конечной точки, которая будет обрабатывать запросы на скачивание файлов.

Пример:

app.MapGet("/download/{fileName}", (string fileName) =>

{

var filePath = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot/files", fileName);

if (!System.IO.File.Exists(filePath))

{

return Results.NotFound("File not found.");

}

var fileBytes = System.IO.File.ReadAllBytes(filePath);

return Results.File(fileBytes, "application/octet-stream", fileName);

});

В этом примере:

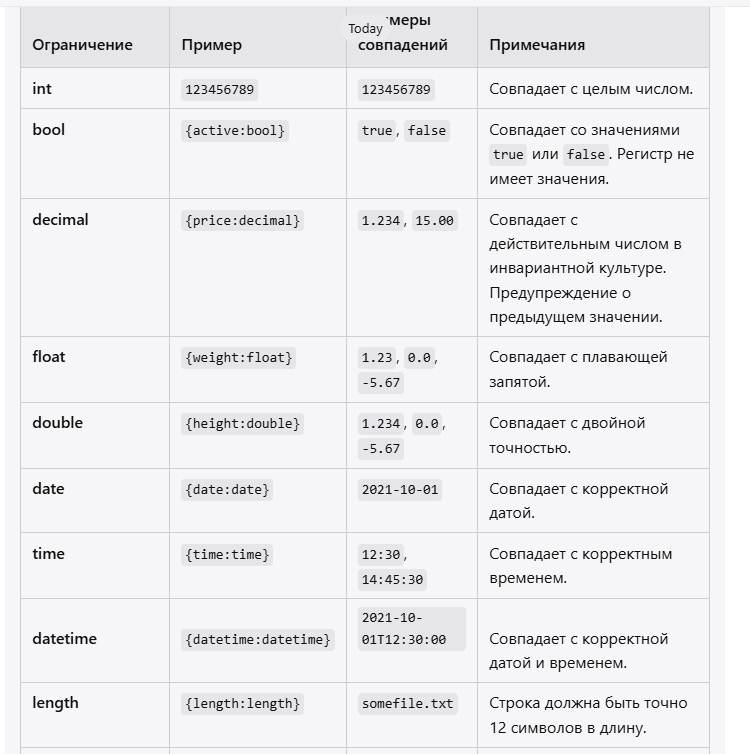
* + fileName передается как параметр из URL.
  + Проверяется существование файла.
  + Если файл существует, он загружается в память и отправляется клиенту с заголовком Content-Disposition, который указывает, что файл должен быть скачан.

1. **Тестирование**:

http://localhost:5000/download/example.txt

* + Если файл example.txt существует в папке wwwroot/files, браузер начнет его скачивание.

1. ASP.NET CORE: приложение miniAPI: шаблоны и констрейнты маршрутов.

****

В ASP.NET Core MiniAPI маршруты позволяют определять шаблоны URL и применять ограничения (констрейнты) для параметров маршрутов. Это помогает контролировать, какие запросы обрабатываются, и обеспечивает более гибкое и безопасное взаимодействие с API.

#### Шаблоны маршрутов

Шаблоны маршрутов — это строки, которые описывают, как должны выглядеть URL. Они могут содержать параметры, которые будут извлечены из URL при обработке запроса.

**Пример простого шаблона маршрута**:

csharp

Copy

app.MapGet("/api/items/{id}", (int id) =>

{

return $"Item ID: {id}";

});

В этом примере {id} — это параметр маршрута, который будет заменен фактическим значением из URL.

#### Констрейнты маршрутов

Констрейнты позволяют ограничить значения параметров маршрутов, чтобы они соответствовали определенным критериям. Например, можно указать, что параметр должен быть целым числом или соответствовать определенному регулярному выражению.

**Примеры констрейнтов**:

1. **Тип данных**:
   * Констрейнт int требует, чтобы параметр был целым числом.

csharp

Copy

app.MapGet("/api/items/{id:int}", (int id) =>

{

return $"Item ID: {id}";

});

1. **Регулярные выражения**:
   * Можно использовать регулярные выражения для более сложных ограничений.

csharp

Copy

app.MapGet("/api/users/{username:regex(^[a-zA-Z0-9\_-]+$)}", (string username) =>

{

return $"Username: {username}";

});

В этом примере параметр username должен соответствовать регулярному выражению, которое допускает буквы, цифры, дефисы и подчеркивания.

#### Комбинирование шаблонов и констрейнтов

Можно комбинировать шаблоны и констрейнты, чтобы создать более сложные маршруты. Например:

csharp

Copy

app.MapGet("/api/products/{category}/{id:int}", (string category, int id) =>

{

return $"Category: {category}, Product ID: {id}";

});

В этом примере маршрут принимает два параметра: category (строка) и id (целое число).

1. ASP.NET CORE: приложение miniAPI: применение сервиса LinkGenerator, поименованные конечные точки (WithName)переадресация с помощью RedirectToRoute.

### ASP.NET Core: Приложение MiniAPI — Применение сервиса LinkGenerator и поименованные конечные точки

В ASP.NET Core MiniAPI вы можете использовать сервис LinkGenerator для создания URL-адресов на основе маршрутов вашего приложения. Это позволяет динамически генерировать ссылки на конечные точки, что особенно полезно для переадресации и работы с поименованными конечными точками.

#### Применение сервиса LinkGenerator

Сервис LinkGenerator позволяет создавать URL-адреса с использованием определенных маршрутов и параметров. Чтобы использовать его, необходимо внедрить LinkGenerator в ваш обработчик.

Пример использования LinkGenerator:

csharp

Copy

app.MapGet("/api/items", (LinkGenerator linkGenerator) =>

{

var url = linkGenerator.GenerateLink("getItem", new { id = 1 });

return Results.Ok(new { Url = url });

});

Здесь GenerateLink создает URL для конечной точки с именем getItem и параметром id.

#### Поименованные конечные точки

Для того чтобы использовать LinkGenerator, необходимо задать имя конечной точки с помощью метода WithName. Это позволяет ссылаться на конечную точку по имени.

Пример задания поименованной конечной точки:

csharp

Copy

app.MapGet("/api/items/{id:int}", (int id) =>

{

return $"Item ID: {id}";

}).WithName("getItem");

В этом примере конечная точка, обрабатывающая запросы по маршруту /api/items/{id}, получает имя getItem.

#### Переадресация с помощью RedirectToRoute

Вы можете использовать RedirectToRoute для переадресации клиента на другую конечную точку, используя ее имя. Это удобно для создания более чистых и управляемых маршрутов.

Пример переадресации:

csharp

Copy

app.MapGet("/api/redirect/{id:int}", (int id, LinkGenerator linkGenerator) =>

{

var url = linkGenerator.GenerateLink("getItem", new { id });

return Results.Redirect(url);

});

В этом примере, когда клиент обращается по маршруту /api/redirect/{id}, он будет переадресован на конечную точку getItem, создавая чистую и понятную логику маршрутизации.

1. ASP.NET CORE: приложение miniAPI: понятие привязки модели Model binding), атрибуты FromXXX, применение собственных типов в шаблонах маршрута, необязательные параметры, массивы параметров.

**Привязка модели** — это механизм, который преобразует данные из HTTP-запроса в объекты .NET, такие как POCO (Plain Old CLR Object). Это позволяет разработчикам работать с объектами, а не с сырыми данными из запросов.

#### Основные типы объектов

1. **POCO**: Обычные классы, которые могут содержать логику и представляют бизнес-объекты.
2. **DTO (Data Transfer Object)**: Классы без логики, предназначенные для передачи данных между частями приложения.
3. **Value Object**: Объекты из области проектирования DDD (Domain-Driven Design), которые не имеют идентичности и могут содержать логику, но передают только данные.
4. **Entity**: Объекты с уникальной идентичностью, представляющие конкретные экземпляры в модели домена.

Привязка модели в ASP.NET Core выполняет следующие задачи:

* Извлекает данные из различных источников, таких как:
  + Данные маршрута
  + Поля формы
  + Строки запроса
* Предоставляет данные контроллерам и страницам через параметры методов и общедоступные свойства Razor.
* Преобразует строковые данные в типы .NET.
* Обновляет свойства сложных типов (например, объекты с вложенными свойствами).

### Этапы привязки модели

Привязка модели осуществляется на этапе **EndpointMiddleware**,

### Применение собственных типов в шаблонах маршрута

В ASP.NET Core можно использовать собственные типы в шаблонах маршрута, что позволяет создавать более выразительные и понятные URL. Для этого необходимо реализовать интерфейс IConvertible или создать собственный механизм преобразования.

public class UserId

{

public int Id { get; }

public UserId(int id)

{

Id = id;

}

public static UserId FromString(string value)

{

return new UserId(int.Parse(value));

}

}

app.MapGet("/users/{userId}", (UserId userId) => $"User ID: {userId.Id}");

В этом примере UserId используется в маршруте, и ASP.NET Core будет автоматически преобразовывать строку из URL в объект UserId.

### Необязательные параметры

Необязательные параметры могут быть указаны в маршруте, добавляя знак вопроса (?) в шаблоне. Это позволяет создавать более гибкие маршруты.

app.MapGet("/greet/{name?}", (string? name) =>

{

return name is null ? "Hello, Guest!" : $"Hello, {name}!";

});

В этом примере параметр name является необязательным. Если он не указан, будет возвращено приветствие для гостя.

### Массивы параметров

Можно принимать массивы параметров в конечных точках, что позволяет передавать несколько значений из URL.

app.MapGet("/items", (string[] ids) =>

{

return $"Item IDs: {string.Join(", ", ids)}";

});

Если запрос включает ?ids=1&ids=2&ids=3, будет возвращён список ID элементов.

* FromRoute: Получает значения из маршрута запроса (например, из URL).
* FromQuery: Получает значения из строки запроса.
* FromBody: Получает значения из тела запроса (чаще всего используется для JSON или XML).
* FromHeader: Получает значения из заголовков HTTP-запроса.
* FromServices: Внедряет зависимость, зарегистрированную в системе DI (Dependency Injection).

1. ASP.NET CORE: приложение miniAPI: хорошо известные типы в параметрах.

### Stream

**Stream** — это абстракция для работы с последовательностью байтов, которая позволяет читать и записывать данные. Это основной базовый тип для различных классов потоков, таких как:

* **FileStream**: Для работы с файлами на диске.
* **MemoryStream**: Для работы с данными в оперативной памяти.
* **NetworkStream**: Для работы с сетевыми соединениями.

**Произношение**: **"Стрим"**

### PipeReader

**PipeReader** — это современный способ работы с потоками данных, который более производителен по сравнению с StreamReader. Он используется в высокопроизводительных приложениях, таких как Kestrel (веб-сервер ASP.NET Core), и позволяет эффективно считывать данные из потока.

**Произношение**: **"Пайп Ридер"**

### CancellationToken

**CancellationToken** — это структура, используемая для отмены операций, как асинхронных, так и длительных. Она является частью механизма отмены задач и часто применяется в методах с использованием async/await, позволяя безопасно отменять выполнение задач.

**Произношение**: **"Кэнселяшн Токен"**

### ClaimsPrincipal

**ClaimsPrincipal** — это класс, представляющий текущего пользователя и его утверждения (claims). Он используется для аутентификации и авторизации, содержая информацию о ролях, логине, правах доступа пользователя и других данных.

**Произношение**: **"Клеймс Принсипал"**

1. ASP.NET CORE: приложение miniAPI: пользовательские привязки, атрибут AsParameter.
2. ASP.NET CORE: приложение miniAPI: валидация

параметров запроса с помощью встроенного механизма (MinimaApisExtentions).

1. ASP.NET CORE: принцип разработки middleware-обработчика ошибок (Use, UseMiddleware, расширение IApplicationBuilder).
2. ASP.NET CORE: понятие и принцип применения Dependency Injection, создание Transient, Singleton b Scope-сервисы, атрибут FromServices.
3. ASP.NET CORE: приложение Razor Page: паттерн MVVM, структура приложения (View, Model), файл Program.cs (builder.Services, app.Map….), маршрутизация, порядок прохождения и обработки запроса, обработчики зпросов генерации ответа, применение Dependency Injection в обработчиках.
4. ASP.NET CORE: приложение Razor Page: Razor-разметка @{…}, @page, @using, @inject, @model, @funtions, @if, @foreach.
5. ASP.NET CORE: приложение Razor Page: Layout-макетирование, применение файла Layout.cshtml.css., применение директив @RenderBody, @RenderSection, @section.
6. ASP.NET CORE: приложение Razor Page: привязка параметров к модели - отображение (mapping).
7. ASP.NET CORE: приложение Razor Page: схемы работы с формами, обработка параметров формы, пересылка (upload) файлов с клиента на сервер с помощью формы
8. ASP.NET CORE: приложение Razor Page: понятие tag helpers, применение частичных представлений,
9. ASP.NET CORE: приложение MVC: определение, назначение, схема приложения, структура проекта.
10. ASP.NET CORE: приложение MVC: маршрутизация, таблица маршрутизации, шаблоны маршрутов.
11. ASP.NET CORE: приложение MVC: контроллеры, жизненный цикл контроллера, фильтры котроллеров, применение Dependency Injection в контроллерах, обработка параметров, способы передачи данных во View.
12. ASP.NET CORE: приложение MVC: представление, жизненный цикл представления, применение Dependency Injection в контроллерах в представлениях, макетирование представлений, @Razor-разметка.
13. ASP.NET CORE: приложение MVC: встроенные и пользовательские Html-helpers.
14. ASP.NET CORE: приложение Web API: понятие REST, назначение Web API, схема приложения, возможные реализации Web API, особенности контроллеров API.