Сборник цитат

* Все дороги ведут либо в REST, либо в серверные объекты, либо в сентябрьскую пересдачу © Саня Рубашек, Сеня Шилов, Дима Трубач
* Не знаешь жизненный цикл контроллера, идешь нахуй © Смелов
* Все мы там будем (пересдача)
* событие, не зависящее от воли и возраста людей © Тест Подручного, но подходит под все случаи жизни

1. Сеть Интернет: сетевые протоколы, службы, организационная структура, документация, форматы данных.

**Интернет: СРАЗУ ЕБАШЬТЕ 4 ЭТИХ КОМПОНЕНТА!!!!!!  
Если сказали, что это сеть на основе TCP/IP то сразу говорите про компоненты, а компоненты это и есть ЭТИ 4 ШТУКИ!!!!**

1) Сеть на основе стека протоколов TCP/IP

2) Службы интернет — **сервер + протокол** / протокол, который описывает, как работать с этим сервером. Стандартные серверы, которые *прослушивают хорошо известные порты (от 0 до 1024).*

3) Набор организаций, обеспечивающих и улучшающих работу Интернета Консорциум - организации, для согласования стандартов

ISOC: Internet Society – международная организация (офисы США, RFC-стандартов. ISOC обеспечивает правовую поддержку и финансирует все другие организации, связанные с деятельностью Internet (IETF, IAB,…).

IETF - организация, инженерная группа, которая предназначена для проектирования интернет, выпуска документации. RFC 4677. Публикует RFC

IAB - совет по архитектуре интернета

ICANN - организация по управлению доменными именами и IP-адресами.

IANA - Администрация адресного пространства Internet. Под контролем ICANN. Кроме того регистрирует типы данных MIME.

W3C **World Wide Web Consortium –** организация разрабатывающая и внедряющая web-стандарты (HTTP, HTML, URI/URL, CSS, DOM, XML, PNG, SVG,…).

ISO - международная стандартизирующая организация

 4) Документация RFC (Request for Comments) и STD

Примеры: DNS, E-mail(SMTP, POP3, IMAP), IRC(обмен сообщений в реальном времени), FTP, Telnet (управление удаленным компьютером в терминальном режиме), **WWW**.

**Internet-служба**:  другое название Internet-сервис, один из видов Internet-ресурса, имеющий специальное назначение (DNS, WWW, E-mail, FTP, ICQ, Telnet)

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

**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)**:
   * **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)**:
   * **Функции**: Протокол для отправки электронной почты.
6. **IMAP/POP3 (Internet Message Access Protocol / Post Office Protocol)**:
   * **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)**:
   * **Функции**: Протокол для передачи файлов между клиентом и сервером.
8. **DNS (Domain Name System)**:
   * **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.

**Основные форматы данных**

1. **HTML (HyperText Markup Language)**:
   * Используется для разметки и отображения веб-страниц.
2. **XML (eXtensible Markup Language)**:
   * Используется для описания данных и обмена информацией между системами.
3. **JSON (JavaScript Object Notation)**:
   * Легкий формат обмена данными, широко используемый в веб-приложениях для передачи данных между клиентом и сервером.
4. **CSS (Cascading Style Sheets)**:
   * Используется для описания внешнего вида и форматирования веб-страниц.
5. **JPEG, PNG, GIF**:
   * Форматы изображений, используемые для передачи графического контента в Интернете.
6. **MP3, AAC, OGG**:
   * Форматы аудио данных для передачи и воспроизведения звукового контента.
7. **MPEG, MP4, WebM**:
   * Форматы видеоданных для передачи и воспроизведения видео контента.
8. **CSV (Comma-Separated Values)**:
   * Простой текстовый формат для представления табличных данных.

MIME - форматы пересылаемых данных



2. Протокол HTTP: версии, основные свойства.

**HTTP** (англ. HyperText Transfer Protocol — «протокол передачи гипертекста») протокол прикладного уровня передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных.

**Основные свойства**

- версии HTTP/1.1 – действующий (текстовый), HTTP/2 – черновой (не распрост, 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 (HTTPS);

- для адресации используется URI;

- поддерживается W3C, описан в нескольких RFC.

Структура HTTP-запроса.

* метод;
* URI;
* версия протокола (HTTP/1.1);
* заголовки (пары: имя/заголовок);
* параметры (пары: имя/заголовок);
* расширение.

Параметры можно передавать:

- query string

- урлой так query string и так в урле

- в теле

Структура HTTP-ответа.

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

**Stateless**-протокол: не сохраняет состояние между запросами.

**Какой эксперимент нужно провести, чтобы доказать, что состояние не сохраняется?** – Создаем обработчик, заводим переменную на уровне класса, при первом запросе обработчик считывает предыдущее значение и ставит новое. При повторном обращении при считывании значение не сохранится. TCP - сохраняет состояние, потому что должен помнить номер пакета, чтобы расположить в нужной последовательности.

**HTTP 2.0**

Данные передаются в бинарном формате в виде фреймов.

**Cмелов: Мультиплексирование - в рамках одного соединения несколько каналов на разных частотах**

Мультиплексирование: в рамках одного соединения много каналов/потоков/стримов. У каждого фрейма есть stream id. Последовательность фреймов в рамках одного стрима важна, а в рамках разных стримов не фиксирована.

Если бы ты писал писал приложение, как бы реализовал мультиплексирование?: для каждого пользователя свой поток и фреймы передаются в рамках одного потока, у фрейма есть свой номер и айдишник потока в котором он передается. И все эти потоки передаются через 1 канал

Сжатие заголовков (HPACK) - заголовки передаются в сжатом виде.

push-сервер – сервер может отсылать данные клиенту (браузеру) для записи в кэш.

**Cмелов: Push сервер - когда делаем ответ на сторону клиентa, мы можем вместе с респонсом отправить данные которые закешируюся на стороне клиента**

**HTTP 3.0**

В качестве транспорта используется UPD. Для обеспечения безопасности используется QUIC (обеспечивает подлинность и правильный порядок пакетов).

QUIC позволяет мультиплексировать несколько потоков данных между двумя компьютерами, работая поверх протокола UDP, и содержит возможности шифрования, эквивалентные TLS и SSL.



3. НTML: назначение, применение, структура страницы, DOM, API.

HTML (от англ. HyperText Markup Language — «язык гипертекстовой разметки») — стандартизированный язык разметки веб-страниц. Код HTML интерпретируется браузерами; полученная в результате интерпретации страница отображается на экране монитора компьютера или мобильного устройства. Спецификации HTML5 формулируются в терминах DOM (объектной модели документа).

**Назначение HTML**

HTML (HyperText Markup Language) — это основной язык разметки, используемый для создания веб-страниц и веб-приложений. Его основное назначение заключается в структурировании контента и данных, которые будут отображаться в веб-браузерах

**Применение HTML**

HTML применяется для:

1. **Создания веб-страниц**: Разметка текста, изображений, видео, таблиц и других элементов.
2. **Создания структурированной информации**: Использование заголовков, параграфов, списков, цитат и других элементов.
3. **Гипертекстовые ссылки**: Создание ссылок на другие страницы и ресурсы.
4. **Формы**: Создание интерактивных форм для ввода и отправки данных.

**Структура HTML-страницы**

HTML-документ состоит из различных элементов, которые организованы в логическую структуру. Основные части HTML-документа:

1. **<!DOCTYPE html>**: Определяет тип документа и версию HTML.
2. **<html>**: Корневой элемент, который содержит все другие элементы.
3. **<head>**: Секция, содержащая метаданные документа (например, заголовок страницы, ссылки на стили и скрипты).
4. **<title>**: Определяет заголовок документа, отображаемый в заголовке браузера.
5. **<body>**: Основная часть документа, содержащая весь контент, который отображается пользователю.

**Семантические теги: article, section, nav, aside, footer, header, figure, figcaption**

**модель DOM (W3C):** представление HTML-документа web-браузером, интерфейс JavaScript для доступа к содержимому HTML-документа. Что делает: парсинг в объекты, предоставляет API доступа к этой древовидной структуре.

**Модель DOM**: четыре уровня DOM0, DOM1, DOM2, DOM3; современные браузеры уровня 2 с элементами уровня 3

Для лучшего понимания: **https://www.geeksforgeeks.org/dom-document-object-model/**

DOM позволяет не только просматривать содержимое страницы, но и взаимодействовать с ним, изменять. Это не статичное отображение, а живой ресурс.

**Основные характеристик**и объектной модели:

• основана на валидном HTML-коде;

• может быть модифицирована из JavaScript;

• не включает псевдоэлементы, созданные из CSS;

• включает скрытые элементы (display: none).

Он представляет документ как дерево объектов, что позволяет программам и скриптам изменять структуру, стиль и содержимое документа.

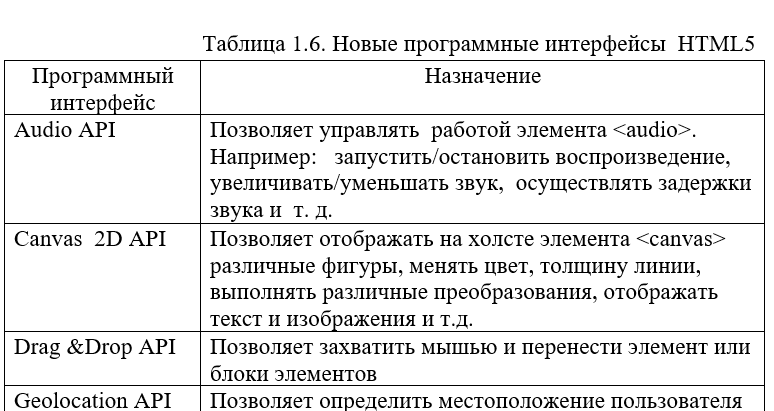
Основные аспекты DOM:

1. **Дерево узлов**: HTML-документ представлен как дерево узлов, где каждый узел соответствует элементу, атрибуту или тексту в документе.
2. **Доступ к элементам**: С помощью DOM можно получить доступ к элементам HTML, изменять их свойства, содержимое и стили.
3. **События**: DOM позволяет назначать и обрабатывать события, такие как клики, загрузка страницы, изменения формы и другие взаимодействия пользователя.

HTML API (Application Programming Interface) предоставляет методы и свойства для работы с HTML-документами через JavaScript. API включает в себя функции для взаимодействия с элементами, управления их поведением и изменениями.

Некоторые важные HTML API:

1. **Element API**: Позволяет взаимодействовать с HTML-элементами, изменять их атрибуты, содержимое и стиль.
2. **Event API**: Позволяет управлять событиями, такими как клики, загрузка страницы, изменение содержимого и т.д.
3. **Form API**: Специфический для работы с HTML-формами, включает методы и свойства для управления полями ввода, отправкой форм и валидацией данных.
4. **Canvas API**: Предоставляет методы для рисования и анимации на веб-странице.





4. CSS: назначение, применение, методы размещения на странице, селекторы.

**Где хранятся CSS в аспах: в wwwroot**

Стиль — это совокупность правил, применяемых к элементу гипертекста и определяющих способ его отображения. Стиль включает все типы элементов дизайна: шрифт, фон, текст, цвета ссылок, поля и расположение объектов на странице.

Таблица стилей — это совокупность стилей, применимых к гипертекстовому документу.

Каскадирование — это порядок применения различных стилей к веб-странице. Браузер, поддерживающий таблицы стилей, будет последовательно применять их в соответствии с приоритетом: сначала связанные, затем внедренные и, наконец, встроенные стили. Другой аспект каскадирования — наследование (inheritance), — означает, что если не указано иное, то конкретный стиль будет применен ко всем дочерним элементами гипертекстового документа. Например, если вы примените определенный цвет текста в теге <div>, то все теги внутри этого блока будут отображаться этим же цветом.

Использование каскадных таблиц дает возможность разделить содержимое и его представление и гибко управлять отображением гипертекстовых документов путем изменения стилей.

**Методы Размещения CSS на Странице**

1. **Встроенные стили (Inline Styles)**: <p style="color: red;">Это красный текст.</p>

2. **Внутренние стили (Internal Styles)**: Определяются в секции **<style>** внутри **<head>** HTML-документа.

3. **Внешние стили (External Styles)**: Определяются в отдельном CSS-файле, который подключается к HTML-документу с помощью тега **<link rel=’stylesheet’ href=’style.css’>**

**4. !!!ТОЛЬКО ЕСЛИ СПРОСИТ!! С помощью js**

Каскадная таблица стилей.

**CSS (Cascading Style Sheets)** – это язык стилей, используемый для описания внешнего вида и форматирования документов, написанных на языках разметки, таких как HTML. Основная задача CSS – отделить содержимое документа от его представления, что позволяет улучшить управление стилями

Селекторы:

Типы селекторов:

Элемент (Типовой селектор): применяет стиль ко всем элементам указанного типа.

p {

color: blue;

}

Идентификатор (ID-селектор): применяет стиль к элементу с конкретным идентификатором.

#header {

background-color: grey;

}

Класс (Классовый селектор): применяет стиль к элементам с указанным классом.

.highlight {

background-color: yellow;

}​

Атрибутный селектор: применяет стиль к элементам с указанными атрибутами.

input[type="text"] {

border: 1px solid black;

}

Комбинированные селекторы: позволяют применять стиль к элементам, соответствующим нескольким условиям.

div p {

color: red;

}

5. Web-приложение: архитектура, основные свойства.

Веб-приложения — клиент-серверное приложение в котором клиент взаимодействует с сервером по протоколу HTTP.

**Cross-Origin Resource Sharing (CORS)** — механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность агенту пользователя получать разрешения на доступ к выбранным ресурсам с сервера на источнике (домене), отличном от того, что сайт использует в данный момент. Корс это политика по вызову сторонних ресурсов.

CORS обеспечивается ***браузером.*** Реагирует, когда джс делает запрос на другой домен. Тогда браузер не дает это сделать. Это он делает: он делать **OPTIONS** запрос на другой домен и спрашивает будешь ли ты принимать запрос с другого домена и если он соглашается, то тогда ему отправляет

HTTP - протокол прикладного уровня, который нужен для того чтобы описывать правила. Эти правила нужны для того, чтобы клиент и сервер могли пересылать друг другу сообщения. HTTP прослушивается через 80 порт или 443 порт – это HTTPS.

Порт — идентификатор .

Веб-сервер представляет собой: HTTP-сервер + файлы. Именно HTTP-сервер взаимодействует с клиентом. Asp.net framework должен быть установлен только на сервере, на клиенте - его нет. Веб-приложение включает все: клиент + сервер.

Web-приложения представляют собой особый тип программ, построенных по архитектуре "клиент-сервер". Особенность их заключается в том, что само Web-приложение находится и выполняется на сервере - клиент при этом получает только результаты работы. Работа приложения основывается на получении запросов от пользователя (клиента), их обработке и выдачи результата. Передача запросов и результатов их обработки происходит через Интернет.

На стороне сервера Web-приложение выполняется специальным программным обеспечением (Web-сервером), который и принимает запросы клиентов, обрабатывает их, формирует ответ в виде страницы, описанной на языке HTML, и передает его клиенту

**Web ресурс приложения:** сущность, расположенная на стороне сервера и имеющая URL/URI, к которой можно сделать http-запрос и получить http-ответ. Одно web-приложение представлено одним или более ресурсов.

**Web ресурсы приложения: статические** - отправляются клиенту без изменения (html-страницы, рисунки, видео-файлы, …), **динамические** – динамически (программно) формируются на сервере и отправляются клиенту (сервлеты, JSP, http-обработчики, aspx-страницы,…). Ресурс может быть статическим относительно сервера и динамическим относительно клиента (html-страницы с JavaScript).

**Общие принципы построения web-приложений:**

1. Web-ресурсы приложения

Web-ресурсы представляют собой различные файлы и данные, доступные через веб-приложение. Это могут быть:

* Статические файлы: HTML, CSS, JavaScript, изображения, видео и другие файлы, которые клиентское приложение загружает и отображает в браузере.
* Динамические данные: Генерируемые сервером данные, такие как JSON, XML или другие форматы, которые используются для обмена данными между клиентом и сервером.

2. Запросы и ответы

Запросы и ответы являются основой взаимодействия между клиентской частью (браузером) и серверной частью (веб-сервером) в web-приложении:

* Запросы: Отправляются клиентским браузером к веб-серверу для получения данных или выполнения операций. Обычно используются HTTP или HTTPS протоколы. Запросы содержат метод (GET, POST, PUT, DELETE и др.), URL-адрес, заголовки и иногда тело запроса.
* Ответы: Возвращаются сервером в ответ на запросы. Ответы содержат HTTP статус (например, 200 OK, 404 Not Found), заголовки и тело ответа, которое может быть HTML, JSON, XML и т.д.

3. Соединение (Сессия)

Соединение (или сессия) — это временная связь между клиентом и сервером в рамках одного веб-приложения:

* HTTP Сессия: Веб-серверы могут поддерживать состояние сессии через cookies или токены аутентификации для идентификации клиента и сохранения его состояния между запросами.

4. Конфигурационный файл приложения

Конфигурационный файл приложения содержит параметры и настройки, необходимые для его работы:

* Настройки подключения к базам данных, конфигурации сетевых параметров, параметры безопасности и другие важные данные.

5. Контекст приложения

Контекст приложения представляет собой окружение, в котором выполняется веб-приложение. Он включает в себя:

* Переменные окружения, конфигурационные данные и другие параметры, которые определяют поведение и состояние приложения.

6. Фильтры

Фильтры (или middleware) используются для обработки запросов и ответов в приложении:

* Промежуточное программное обеспечение, которое выполняет предварительную и постобработку данных, проверку безопасности, логирование и другие операции перед передачей запроса контроллерам и после получения ответа от них.

7. Кэш (данных и вывода)

Кэширование данных и вывода улучшает производительность приложения:

* Кэш данных: Хранит результаты вычислений или запросов к базе данных для повторного использования без повторного выполнения операций.
* Кэш вывода: Хранит скомпилированные или предварительно сформированные HTML страницы для быстрого ответа на запросы клиентов.

8. Слушатели событий

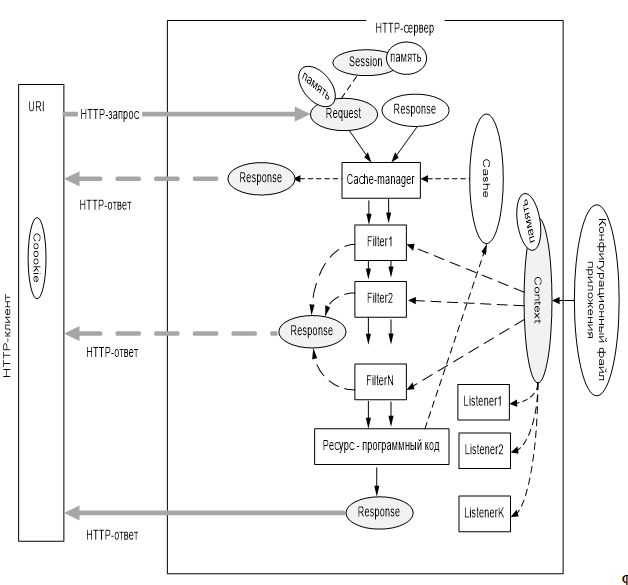
Слушатели событий отслеживают действия пользователя или изменения состояния приложения:

* На клиентской стороне: JavaScript обрабатывает кликовые события, отправку форм, загрузку страницы и другие действия пользователя.
* На серверной стороне: Примеры включают слушатели для обработки HTTP запросов, асинхронных задач и обновлений данных.

9. Принципы безопасности

Безопасность — критически важный аспект веб-приложений:

* Аутентификация и авторизация: Проверка подлинности пользователей и управление их доступом к функционалу приложения.
* Защита от уязвимостей: Обеспечение безопасности данных, защита от XSS (межсайтовый скриптинг), CSRF (межсайтовая подделка запроса) и других угроз.



6. Web-сервер: основные серверные объекты.

**Контекст web-приложения**: серверный резидентный объект, предназначенный для хранения информации об одном web-приложении, общий для всех файлов. Как правило, формируется сразу при загрузке web-сервера, основные данные (параметры приложения) копируются из конфигурационного файла приложения, общий для всех сессий приложения, Обычно контекст предоставляет возможность хранить данные в формате ключ/значение.

Контекст сервера хранит информацию между сессиями.

**Фильтр (Filter)**: серверный объект – препроцессор запроса, предназначен для предварительной обработки объекта Request. К одному ресурсу может быть построена цепочка фильтров, последний в цепочке – ресурс. Фильтр может прервать цепочку и сам сформировать ответ клиенту. Один и тот же фильтр может быть применен к нескольким ресурсам. В качестве параметров фильтр получает объекты Request и Response, которые он передает дальше по цепочке или обрывает цепочку и заполняет объект Response.

**Слушатели событий (Lister):** серверные объекты – для обработки событий жизненного цикла web-приложения.

**Кэш (Cache):** серверный объект, предназначенный для временного хранения данных с целью ускорения выполнения запроса. Кэширование – процессы записи и извлечения данных в/из Cache. Различают кэширование данных и кэширование вывода. **Кэширование данных** – кэширование часто используемых данных. **Кэширование вывода** – кэширование объекта Response.

**Запрос(Request):** серверный объект, который образуется в результате обработки сервером http-запроса, поступающего от клиента и передается серверному программному коду для обработки. Содержит: всю информацию из http-запроса: метод, коллекция заголовков, коллекция параметров, поток данных … Обычно объект Request предоставляет возможность хранить данные в формате ключ/значение.

**Ответ(Response):** серверный объект, который автоматически формируется сервером, при получении http-запроса (одновременно с объектом Request), заполняется данными серверными программным кодом, преобразуется в http-ответ и отправляется клиенту. Содержит: всю информацию, которая должна быть помещена в http-ответ: статус, коллекция заголовков, поток данных, …

**Сессия (Session)**: серверный объект, хранящий информацию о соединении с клиентом, создается при первом обращении время жизни: timeout (системный параметр, обычно равен 10 – 30 минутам) – максимальное время между запросами клиента. Если timeout превышен, то Session разрушается и при следующем запросе создается новый экземпляр. Каждая сессия имеет собственный идентификатор (Session ID, 16 или более байт). Каждый Request принадлежит, какой-то сессии (имеет ссылку на объект Session или содержит Session ID). Обычно объект Session предоставляет приложению возможность хранить данные в формате ключ/значение.

!!!!!! **Про группу запросов с клиента**: Когда один клиент отправляет группу запросов на сервер, все запросы будут ассоциированы с одной и той же сессией, используя один и тот же идентификатор сессии (если клиент поддерживает cookies и сервер правильно настроен для работы с сессиями).!!!!!!

**Пул соединений с базой данных:** несколько предварительно и постоянно открытых соединений с сервером СУБД, которые используют приложения. Выбор подключения из пула по Open, возврат в пул Close. Если все подключения пула заняты, запрос на соединение ставится в очередь. Применение пула позволяет увеличить производительность за счет отсутствия процесса подключения к серверу. Альтернатива: держать постоянное открытое соединение для каждого подключения (в общем случае для web-приложения не приемлемо) или открывать/закрывать соединение при каждом запросе (большие накладные расходы на установку соединения).

**Кто является инициатором создания Куки? - сервер**

**Что кеширует сервер? - Response.**

**Как сервер обрабатывает запрос? -** Мы отправляем http-запрос от клиента к серверу. Общий вид запроса - битовая последовательность. Http-сервер состоит из http-драйвера и ресурс-программный код. Драйвер первый принимает http-запрос. Он формирует 2 серверных объекта: реквест и респонс (пустой). Далее проверяется это новая сессия или нет. Дальше кэш-менеджер смотрит, надо ли отправлять на ресурс или нет. Допустим нет, тогда из кэша берет данные и назад их отсылает. А если в кэше нет данных, то идет на фильтр. Фильтр может отослать на другой фильтр (тогда будет цепочка) или на ресурс, а может решить, что не будет никуда отсылать. Дальше request + response переходят на программу http-сервера и в этой программе есть http-обработчики (хэндлеры) и эти обработчики принимают request и response. Из request программа, которая находится на сервере понимает, что именно клиент хочет получить от нее, эта программа заполняет response. Обработчик формирует нам response и шлет обратно на http-драйвер. Http-драйвер преобразовывает в битовую последовательность и отправляет клиенту. Клиент получает ответ в виде битовой последовательности, преобразовывает в ответ, который понятен для него - html-разметка

7. Web-браузер: структура, основные объекты, DOM, BOM.

**Браузер** прикладное программное средство, предназначенное:

* формирования и выполнения http-запросов;
* получения и обработки http-ответов;
* отображения компьютерных файлов;
* интерпретации js-файлов.

**Браузерное программирование** – разработка приложений, работающих в рамках браузера.

**основные объекты**

**BP: User** **Interface** – модуль пользовательского интерфейса, позволяющий пользователю управлять работой браузера (вводить URI, движение вперед/назад по истории, закладки и пр.)

**BP: Browser** **Engine** – модуль управления браузером: управление закладками, скачивание, проверка орфографии, поиск на странице, …).

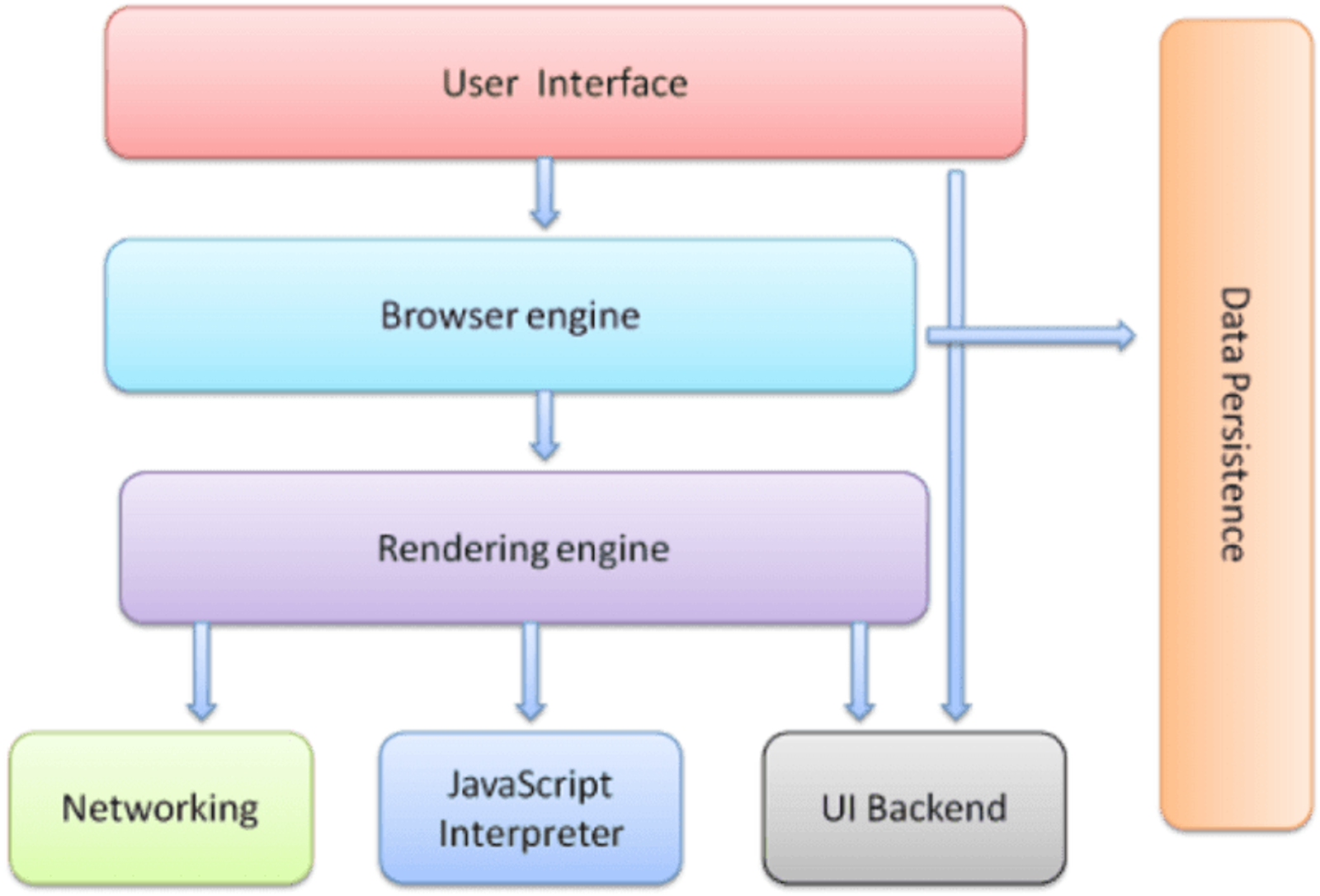
**BP: Render** **Engine** – модуль отображения контента: WebKit, Blink, Gecko. Стандарты HTML, CSS, XML, DOM

**BP: Data Persistence** – модуль обеспечивающий работу с хранилищем: Local storage, Session Storage, Cookies, WebSQL, IndexedDB, File System, AppCache, Service Workers.

**BP: JS Engine** – модуль-интерпретатор JavaScript: V8,Gecko, SpiderMonkey, Rhino, Tamarin, Chakra, Carakan. Стандарт JS и Browser API.

**BP: Networking** – модуль взаимодействия с сетью.

**BP:** программисту браузер представляется в виде модели **Browser** **Object** **Model**(BOM), модели **Document** **Object** **Model** (DOM) и **Browser** **API**.



**BOM, DOM**

**Модель DOM (Document Object Model)** DOM представляет собой объектную модель, которую браузеры используют для представления и взаимодействия с HTML-документами. DOM позволяет JavaScript манипулировать содержимым, структурой и стилями документов.

\*\***Модель BOM (Browser Object Model)** \*\*BOM предоставляет интерфейсы для взаимодействия с браузером вне контекста HTML-документа. BOM включает объекты, такие как **window**, **navigator**, **screen**, **location**, и **history**.

[BOM (Browser Object Model)](https://learn.javascript.ru/browser-environment#bom-browser-object-model)

**Чем характеризуется браузер?** – DOM, BOM (windows, localStorage, navigator, history), XMLHttpRequest, CSS (версии), JavaScript, HTML5 API (canvas, webstorage, webworkers, audio, video, geolocation, filesystem API’s), какие протоколы поддерживает.

8. Методология AJAX: асинхронные запросы, объекты XMLHTTPREQUEST и FETCH, форматы JSON, XML.

(Лекция 3) **AJAX: Asynchronous** **JavaScript** **and** **XML** – асинхронный JavaScript and XML – методология (подход) построения динамических приложений, основанная на асинхронных запросах при которых не осуществляется полная перезагрузка html-страниц.

**Асинхронный запрос:**

1. Заявка на выполнение
2. Ожидание результата

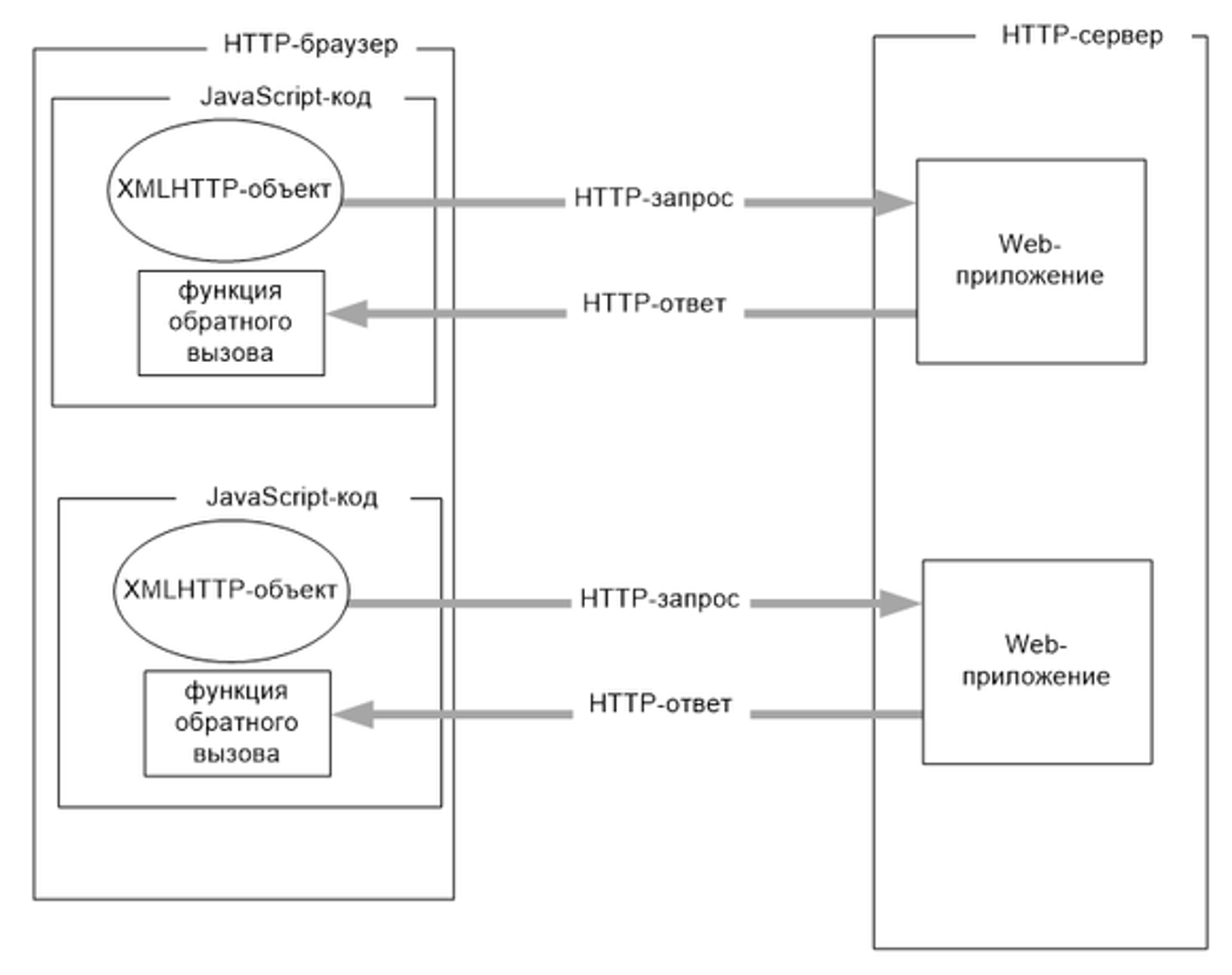
**Объекты XMLHTTPREQUEST и FETCH**

[XMLHTTPREQUEST](https://learn.javascript.ru/xmlhttprequest) — это **встроенный в браузер объект**, который даёт возможность делать HTTP-запросы к серверу без перезагрузки страницы. (существует только в браузере)

**XMLHttpRequest.readyState** возвращает текущее состояние объекта XMLHttpRequest. Объект XHR может иметь следующие состояния:



* status: Свойство, которое содержит HTTP-статус ответа сервера (например, 200 для успешного ответа).
* responseText или response: Свойство, которое содержит текст ответа сервера или объект с ответом, в зависимости от заданного типа ответа (например, XML или JSON).



**Как обработать ответ? -** в свойство onReadyStateChange записать ссылку на callback функцию, которая будет вызываться браузером в зависимости от состояний. На 4 стадии можно обрабатывать сам ответ. Можно использовать для progress-bar.

**Что такое Callback?** - это функция, которая передается в другую функцию в качестве аргумента и вызывается после выполнения определенных операций или событий.

**В C++** реализуются с использованием указателей на функции или функциональных объектов. Это позволяет передать функцию в качестве аргумента в другую функцию, которая затем может вызвать переданную функцию в нужный момент.

**В C#** callback-функции реализуются с помощью делегатов. Делегаты представляют ссылки на методы и позволяют передавать методы как параметры в другие методы. Это позволяет использовать делегаты для реализации callback-функциональности. В C# также есть предопределенные делегаты, такие как Action и Func, которые облегчают работу с callback-функциями.

**Методы:**

Метод open (URI + метод, bool async, sync) - инициализирует новый запрос.

Send(data) - отправляет заявку.

Fetch API: Это современный API для выполнения сетевых запросов в браузере. Он предоставляет более гибкий и мощный способ выполнения запросов по сравнению с XMLHttpRequest

fetch() - это метод, встроенный в браузер. Возвращает **promise**.

Fetch в себе содержит  [XMLHttpRequest](https://ru.wikipedia.org/wiki/XMLHttpRequest).

let promise = fetch(url, [options]),

где url - URL для отправки запроса, options - дополнительные параметры: метод, заголовки и так далее.

**Как зациклить феч? -** Артем сказал никак????

**Promise** - это js-объект, представляющий асинхронную операцию и её результат.

Promise может находиться в трёх состояниях:

* *ожидание (pending)*: начальное состояние, не исполнен и не отклонён.
* *исполнено (fulfilled)*: операция завершена успешно.
* *отклонено (rejected)*: операция завершена с ошибкой.

Форматы: JSON, XML

JSON - [текстовый формат](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82) [обмена данными](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BC%D0%B5%D0%BD_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%BC%D0%B8), основанный на [JavaScript](https://ru.wikipedia.org/wiki/JavaScript)-объектах.

**XML** (*e****X****tensible* ***M****arkup* ***L****anguage*)— «расширяемый [язык разметки](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D1%80%D0%B0%D0%B7%D0%BC%D0%B5%D1%82%D0%BA%D0%B8)».

**Пространство имён в XML** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *XML namespace*) — это стандарт, описывающий именованную группу имён элементов и атрибутов, служащую для обеспечения их уникальности в [XML](https://ru.wikipedia.org/wiki/XML)-документе.

**🤠**

9. Платформа .NET CORE: свойства, назначение, применение.

.NET Core – это платформа для разработки ПО (Фреймворк ОС)

Платформа – это набор библиотек и инструментов для разработки

**Фреймворк ОС** - ***набор библиотек (функций) и порядок работы с ним, которые являются посредниками между разрабатываемым ПО и ядром ОС.***

Другие определения:

1. Совокупность библиотек и среды выполнения, облегчающее разработку большого программного проекта.

2. Набор библиотек и правила работы с ними, предназначены для интерфейсов выполнения программ с ядром ОС.

3. Фреймворк операционной системы предоставляет структуру и набор API (Application Programming Interfaces), которые разработчики могут использовать для создания, тестирования и развертывания приложений, которые могут работать в определенной операционной системе.

Свойства: **кроссплатформенность** и открытый код

Инструменты:

· Core CLR – загружает сборку (метаданные + IL код) и с помощью JIT-компилятора компилирует IL в машинный (нативный) код и сразу исполняет его (это происходит на этапе исполнения программы).

· JIT-компилятор – во время выполнения JIT-компилятор преобразует инструкции MSIL в машинный код. Во время этой компиляции выполняется проверка кода и метаданных MSIL с целью установить, можно ли для них определить, является ли код типобезопасным.

.NET Core: альтернатива .NET, своя реализация CoreCLR и библиотеки CoreFX.

.**NET Core = CoreCLR + библиотеки CoreFX.**

!!!Главная особенность – 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.

ASP.NET Core:

- является фреймворк с открыт hiым кодом

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

- приложения могут работать под IIS (Windows) или под web-сервером Kestrel (кроссплатформенный вариант).

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

**-NuGet – программный менеджер для установки компонентов приложения оформленных в виде пакетов**

- модульность – …

- pay-for-play – подтягиваем только то, что используем

- CLR и JIT также устанавливаются как компонент

- типы приложений MVC UI, Web API, Web Pages. Web Forms не поддерживается.

**- есть встроенное внедрение зависимостей**

- есть интерфейс командной строки CLI, с помощью которого можно создавать и запускать приложения, там есть команда dotnet

- в основе asp.net core лежит интерфейс OWIN

- последняя версия 8.0

Развертывание приложений:

1. FDD – Зависящее от платформы развертывание (Framework-Dependent Deployment, FDD) — это метод развертывания .NET приложений, при котором приложение зависит от уже установленной на целевой системе платформы .NET. В этом случае приложение использует общий runtime, установленный на машине.

2. SCD – Самодостаточное развертывание (Self-Contained Deployment, SCD) — это метод развертывания .NET приложений, при котором приложение включает в себя все необходимые компоненты, включая .NET runtime. Таким образом, приложение не зависит от установленной на целевой системе платформы .NET. Приложение включает все необходимые зависимости и может быть запущено на любой системе без необходимости устанавливать .NET runtime.

.NET runtime – набор приложений и библиотек необходимых для работы приложения

dotnet.exe точка входа для любого asp приложения

любое запускаемое приложение запускается дотнетом.ехе

мы всегда разрабатываем дллку, а затем запускаем через дотнет

других способов запуска приложения на шарпах (и не только) кроме как через дотнет.ехе нет. можно даже просто запустить дотнет в program files и указать свою дллку и это будет работать

Фреймворк находится в с/програм файлс/дотнет/shared

EXE-шник, получаемый при компиляции шарпов - это переименованный dotnet.exe, а функционал лежит в DLL

10. Платформа ASP.NET CORE: свойства, назначение, применение.

ASP.NET Core – программная платформа для разработки **web-приложений** на основе .NET Core

ASP.NET Core: программная платформа, разработанная Microsoft и предназначается для разработки web-приложений. Является развитием технологии OWIN (The Open Web Interface for.NET). Katana – OWIN-совместимый хост, разработанный Microsoft.

OWIN

Open Web Interface OWIN – описывает взаимодействие компонентов **серверной** части.

OWIN: Open Web Interface for NET. OWIN – интерфейс между net-web-сервером и серверным приложением (обработчиками запросов). Основная цель OWIN отделить web-сервер от серверного приложения. Можно разрабатывать отдельно сервер и приложение.

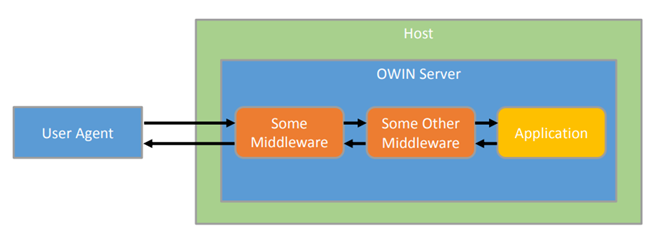
Приложение состоит из 4 компонентов

1. Host – приложение-процесс операционной системы, управляющий жизненным циклом OWIN Server.

2. OWINServer - http-сервер, реализующий интерфейс OWIN. Превращает побитовые запросы клиента в Request, а Response преобразует в побитовый ответ клиенту

3. Application (обработчик запросов)

4. middleware – конвейер обработки запросов и ответов; подключенные компоненты (модули), предназначенные для обработки запросов.



Описание схемы от чатика:  
 Эта схема иллюстрирует архитектуру OWIN (Open Web Interface for .NET) и её взаимодействие с пользователем и приложением. Давайте разберём её подробнее:

1. **User Agent**: Это пользовательский агент, который посылает запросы на сервер. Обычно это веб-браузер или любое другое приложение, взаимодействующее с веб-сервером.
2. **Host**: Это окружение, в котором работает OWIN сервер. Host отвечает за запуск и выполнение OWIN серверов и приложений.
3. **OWIN Server**: Это сервер, который принимает запросы от пользовательского агента и передает их через последовательность компонентов OWIN. Он также отвечает за передачу ответов обратно к пользовательскому агенту.
4. **Some Middleware**: Middleware (посредническое программное обеспечение) - это компоненты, которые обрабатывают запросы и ответы. Они могут выполнять различные задачи, такие как аутентификация, логирование, обработка ошибок и т.д. В схеме показаны два компонента Middleware:
   * **Some Middleware**: Первый компонент посредника, который получает запрос от OWIN сервера, обрабатывает его и передает дальше.
   * **Some Other Middleware**: Второй компонент посредника, который получает запрос от первого компонента посредника, обрабатывает его и передает дальше.
5. **Application**: Это само приложение, к которому в конечном итоге поступает запрос. Приложение обрабатывает запрос, формирует ответ и передает его обратно через все компоненты Middleware к OWIN серверу, а затем к пользовательскому агенту.

Есть 5 компонент – внедрение зависимостей

Суть: стандартизация – разработка отдельных компонентов. Проще поддерживать и расширять.

Условие: компоненты как можно более независимы, чтобы их можно было перенести.

**OWIN:** обеспечивает интерфейсы: между приложением и http-сервером, между http-сервером и Host, между http-сервером и middleware.

**Katana:** Microsoft-реализация OWIN сервера (говорят проект Katana). В качестве Host можно использовать IIS или self-hosting. Кроме того, позволять подключать модули middleware; предоставляет набор классов для работы с сервером и механизм подключения приложения к серверу.

**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, перечислены в файле [AppName].deps.json
* **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 названию проекта (по умолчанию названию каталога) и описывает все его настройки. 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.
* **Program.cs**: главный файл приложения, с которого и начинается его выполнение. Код этого файла настраивает и запускает веб-приложение
* **Папка wwwroot**

Объект **Microsoft.AspNetCore.Builder.WebApplication**. Этот объект настраивает всю конфигурацию приложения, его маршруты, используемые зависимости и т.д

App – резидентный объект, висит внутри хоста.

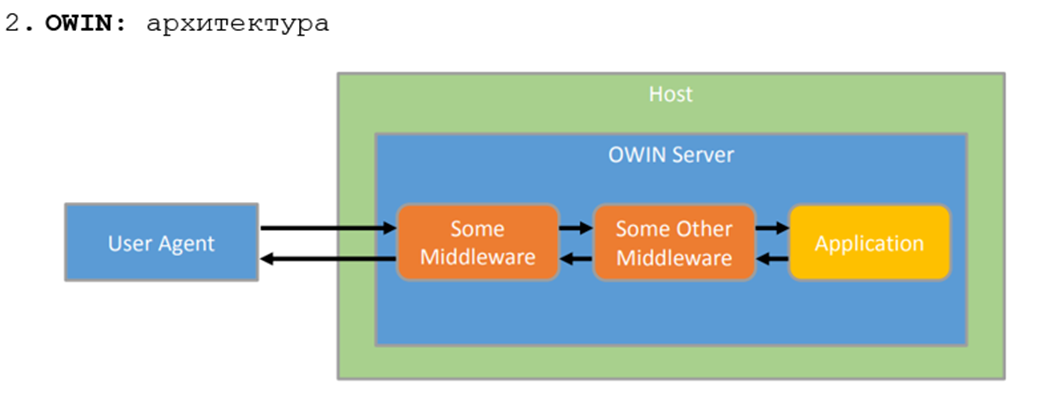
Context – объект типа HttpContext, инкапсулирует всё, что связано с http.

11. Платформа ASP.NET CORE: общая структура приложения.

OWIN

Open Web Interface OWIN – описывает взаимодействие компонентов **серверной** части.  
Зачем owen? OWEN: стандартизация, структурирование, любой разработчик может разработать свой компонент

OWIN: Open Web Interface for NET. OWIN – интерфейс между net-web-сервером и серверным приложением (обработчиками запросов). Основная цель OWIN отделить web-сервер от серверного приложения. Можно разрабатывать отдельно сервер и приложение.



Приложение состоит из 4 компонентов

1) host(.net приложение)

2) http-сервер(программа, которая умеет принимать битовые последовательности от клиента и их превращать в объект Request, умеет принимать ответ Response и преобразовывать его в битовую последовательность)

3) Middleware(это конвейер обработки запроса и ответа)

4) Приложение(обработчик запросов)

Фреймворк встраивается в Приложение или Middleware

Есть 5 компонент – внедрение зависимостей

**OWIN:** обеспечивает интерфейсы: между приложением и http-сервером, между http-сервером и Host, между http-сервером и middleware.

**Katana:** Microsoft-реализация OWIN сервера (говорят проект Katana). В качестве Host можно использовать IIS или self-hosting. Кроме того, позволять подключать модули middleware; предоставляет набор классов для работы с сервером и механизм подключения приложения к серверу.

**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 названию проекта (по умолчанию названию каталога) и описывает все его настройки. 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.
* **Program.cs**: главный файл приложения, с которого и начинается его выполнение. Код этого файла настраивает и запускает веб-приложение
* **Папка wwwroot**

Объект **Microsoft.AspNetCore.Builder.WebApplication**. Этот объект настраивает всю конфигурацию приложения, его маршруты, используемые зависимости и т.д

<https://metanit.com/sharp/aspnet6/1.3.php>

<https://metanit.com/sharp/aspnet6/2.1.php>

12. Платформа ASP.NET CORE: простейший обработчик запросов.

Простейшее приложение - обработчик HTTP-запросов

Пример обработчика:

| app.MapGet("/example", async (context) =>  {  HttpResponse response = context.Response;  HttpRequest request = context.Request;  string parmA = request.Query["ParmA"];  string parmB = request.Query["ParmB"];  response.ContentType = "text/plain";  if (parmA != null && parmB != null)  {  await response.WriteAsync($"GET-Http-XXX: ParmA = {parmA}, ParmB = {parmB}");  }  else  {  await response.WriteAsync("Parameters cannot be null");  }  }); |
| --- |

Когда мы обращаемся к приложению, то посылаем ему запрос определенного типа - get, post, put и т.д. Для обработки запросов в Razor Pages можно определять соответствующие методы, которые называются по шаблону *On[Тип\_запросы]*, либо можно использовать их асинхронные двойники *On[Тип\_запросы]*. Например, запрос Get обрабатывается методами **OnGet()** и **OnGetAsync()**, а запросы Post - методами **OnPost()** и **OnPostAsync()**.

13. Платформа ASP.NET CORE: WebSocket - сервер.

WebSocket: Протокол передачи данных, основанный на TCP, для установки длительного соединения и обмена сообщениями между клиентом и сервером в режиме дуплексной связи. Находится на прикладном уровне модели OSI.

Соединение устанавливается следующим образом:

1.  Клиент посылает обычный HTTP-запрос, называемый рукопожатием, с заголовками Upgrade и Connection (Connection: Upgrade, Upgrade: websocket)

2.  Сервер отвечает HTTP-ответом со статусом 101 (Switching Protocols), показывая, что он согласен переключить протокол на WebSocket. Ответ сервера содержит заголовок Upgrade: websocket и ключи для подтверждения протокола (например, Sec-WebSocket-Accept), сгенерированные на основе ключа, отправленного клиентом в заголовке Sec-WebSocket-Key.

Если не поддерживает, то наверное 400 Bad Request

3. Сразу после отправки сервером ответа,  TCP-соединение остается открытым, клиент и сервер могут начинать двунаправленный обмен сообщениями  по этому же TCP-соединению.

При этом протокол определяет две URL-схемы: ws – для незашифрованного (80 порт дефолт) соединения и  wss – для зашифрованного (443 порт дефолт).

Применения WebSockets

WebSockets находят применение в:

* **Чат-приложениях**: Обеспечивают моментальный обмен сообщениями между пользователями.
* **Реального времени обновлениях**: Например, в приложениях для отслеживания фондовых рынков или спортивных событий.

**Онлайн-играх**: Обеспечивают низкую задержку и высокий уровень взаимодействия между игроками.

WebSocket — протокол связи поверх TCP-соединения,предназначенный для обмена сообщениями между браузером и веб-сервером в режиме реального времени.

Дуплексный протокол – 2 однонаправленных протокола в разные стороны, которые работают независимо друг от друга.

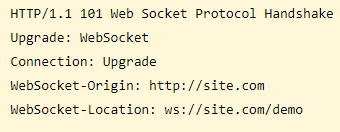
Изначально синхронный HTTP протокол, построенный по модели «запрос — ответ», становится **полностью асинхронным и симметричным**. Теперь уже нет клиента и сервера с фиксированными ролями, а есть два равноправных участника обмена данными. Каждый работает сам по себе, и когда надо отправляет данные другому. Одна сторона отправит данные и продолжит работу дальше, ничего ждать не надо. Вторая сторона ответит, когда захочет — может не сразу, а может и вообще не ответит.

Подключение:

Браузер подключается по протоколу TCP на 80 порт сервера и дает немного необычный GET-запрос:



Если сервер поддерживает Вебсокеты, то он отвечает таким образом:



Когда сервер и клиент послали **handshake** запросы, и проверка пройдена, то начинается этап обмена данными.

Если браузер это устраивает, то он просто оставляет TCP-соединение открытым.

Как только одна сторона хочет передать другой какую-то информацию, она отправляет дата-фрейм следующего вида:



То есть просто строка текста — последовательность байт, к которой спереди приставлен нулевой байт 0x00, а в конце — 0xFF. И все — никаких заголовков, метаданных! Что именно отправлять, разработчики полностью оставили на ваше усмотрение: хотите XML, хотите JSON.

С помощью WebSockets так же можно передавать и бинарные данные и картинки.

Объект WebSocket имеет несколько состояний (**readyState**), которые отражают текущее состояние соединения между клиентом и сервером. Эти состояния определяются согласно спецификации WebSocket API (RFC 6455) и доступны через свойство **readyState** объекта WebSocket в JavaScript. Вот список возможных состояний:

1. **CONNECTING (0)**:
   * Состояние, когда инициировано установление соединения, но рукопожатие ещё не завершено.
   * WebSocket объект переходит в это состояние сразу после создания.
2. **OPEN (1)**:
   * Состояние, когда соединение установлено и готово для обмена данными.
   * WebSocket объект переходит в это состояние, когда рукопожатие завершено успешно.
3. **CLOSING (2)**:
   * Состояние, когда начат процесс закрытия соединения.
   * WebSocket объект переходит в это состояние, когда вызван метод **close()**, и он начал отправлять закрывающее сообщение серверу.
4. **CLOSED (3)**:
   * Состояние, когда соединение закрыто или не удалось установить.
   * WebSocket объект переходит в это состояние, когда соединение закрыто явно вызовом метода **close()** или в результате ошибки при установке соединения.

[**Http-обработчик для взаимодействия с клиентом по протоколу WebSocket**].

С помощью свойства IsWebSocketRequest объекта context.WebSockets можно определить является ли пришедший запрос, запросом на соединение по протоколу WebSocket. Если это так, устанавливается соединение и управление передается методу WebSocketRequest.

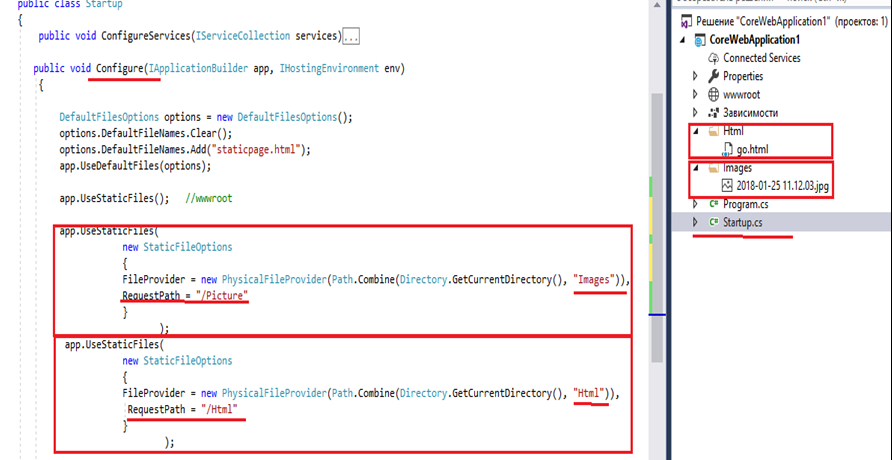
| app.Map("/ws", async (context) =>  {  WebSocket socket;  if (context.WebSockets.IsWebSocketRequest)  {  socket = await context.WebSockets.AcceptWebSocketAsync();  await WebSocketRequest(socket);  }  });  private static async Task WebSocketRequest(WebSocket socket)  {  string s = await Receive(socket);  await Send(socket, s + " " + DateTime.Now.ToString("HH:mm:ss"));  while (true)  {  while (socket.State == WebSocketState.Open)  {  Thread.Sleep(2000);  await Send(socket, DateTime.Now.ToString("HH:mm:ss"));  }  }  }  private static async Task<string> Receive(WebSocket socket)  {  var buffer = new ArraySegment<byte>(new byte[512]);  var result = await socket.ReceiveAsync(buffer, CancellationToken.None);  return Encoding.UTF8.GetString(buffer.Array, 0, result.Count);  }  private static async Task Send(WebSocket socket, string s)  {  var sendBuffer = new ArraySegment<byte>(Encoding.UTF8.GetBytes($"Ответ: {s}"));  await socket.SendAsync(sendBuffer, WebSocketMessageType.Text, true, CancellationToken.None);  } |
| --- |

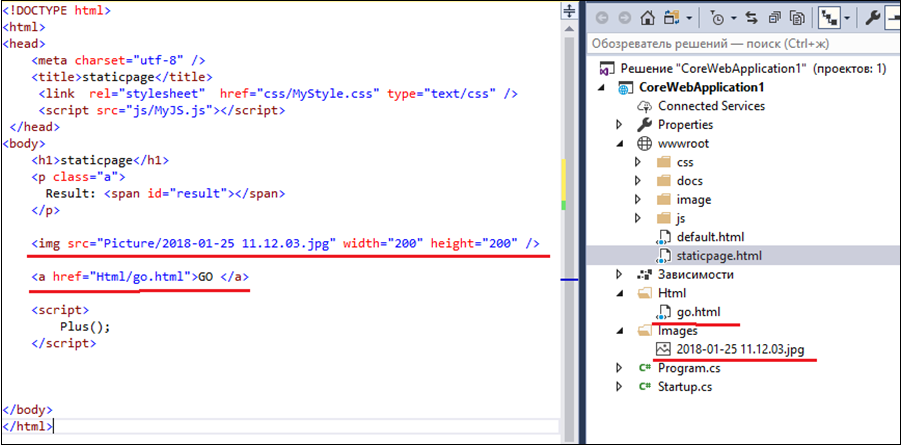
**Как проверить, что WebSocket соединение дуплексное? -** Создать два потока, в одном читать в другом отправлять по вебсокету. Вроде как-то так

14. Платформа ASP.NET CORE: статические данные и стартовые страницы.

Отображение статического контента, необходимо подключить дополнительные nuget-пакеты; при создании Core-проекта автоматически подключен мегапакет Microsoft.AspNetCore.All в котором все есть.

**Добавление папок для статических файлов:**





**UseDefaultFiles:** Этот middleware отвечает за обработку запросов на каталог (например, /), когда в URL не указан конкретный файл. Он пытается найти и обслужить файлы по умолчанию (например, index.html, default.htm, index.htm и т.д.).Если запрос удовлетворяет файлу по умолчанию, middleware завершает обработку и возвращает файл клиенту

**UseStaticFiles:** Этот middleware служит для обслуживания статических файлов (например, HTML, CSS, JavaScript, изображения и т.д.) из указанного каталога (например, wwwroot).Если запрос не удовлетворяет файлу по умолчанию (обработанному UseDefaultFiles), UseStaticFiles попытается найти и вернуть запрашиваемый файл из каталога статических файлов.

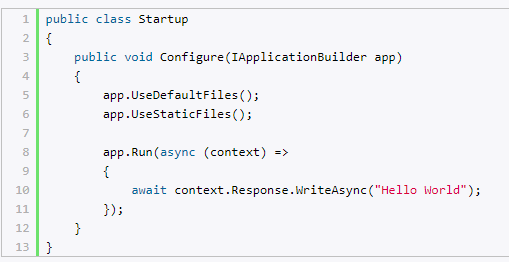
**Почему важен порядок?**

Если UseStaticFiles объявлено раньше UseDefaultFiles, то запросы на каталог (например, /) будут напрямую переданы UseStaticFiles. В этом случае, если в каталоге нет конкретного файла, который соответствует запросу (например, index.html), UseStaticFiles не сможет обработать такой запрос, и клиент получит ошибку 404 (файл не найден).

UseDefaultFiles перед UseStaticFiles гарантирует, что сначала будет проверяться наличие файлов по умолчанию в каталоге. Если файл по умолчанию найден (например, index.html), то UseDefaultFiles вернет этот файл, и UseStaticFiles уже не будет обрабатывать запрос.

**Работа с файлами:**

C помощью специального метода расширения UseDefaultFiles() можно настроить отправку статических веб-страниц по умолчанию без обращения к ним по полному пути:

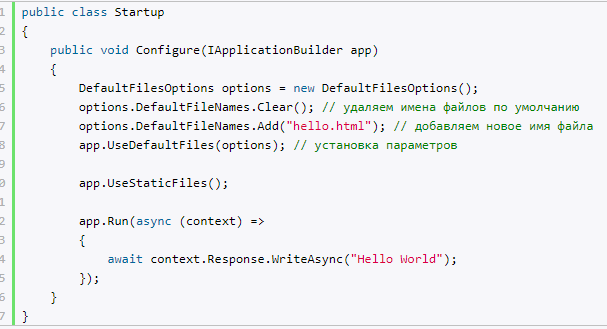


В этом случае при отправке запроса к корню веб-приложения типа http://localhost:xxxx/ приложение будет искать в папке wwwroot следующие файлы:

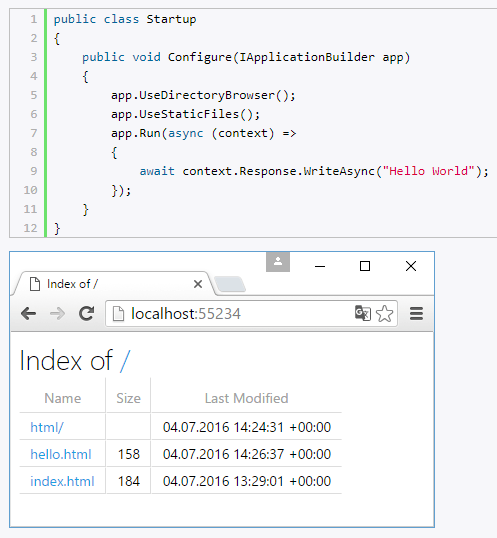
1. default.htm
2. default.html
3. index.htm
4. index.html

Если файл будет найден, то он будет отправлен в ответ клиенту. Если же файл не будет найден, то продолжается обычная обработка запроса с помощью следующих компонентов middleware. То есть фактически это будет аналогично, как будто мы обращаемся к файлу: http://localhost/index.html

Если же мы хотим использовать файл, название которого отличается от вышеперечисленных, то нам надо в этом случае применить объект DefaultFilesOptions:



Метод **UseDirectoryBrowser** позволяет пользователям просматривать содержимое каталогов на сайте:



Данный метод имеет перегрузку, которая позволяет сопоставить определенный каталог на жестком диске или в проекте с некоторой строкой запроса и тем самым потом отобразить содержимое этого каталога:

app.UseDirectoryBrowser(new DirectoryBrowserOptions(){

FileProvider = new PhysicalFileProvider(Path.Combine(

Directory.GetCurrentDirectory(),

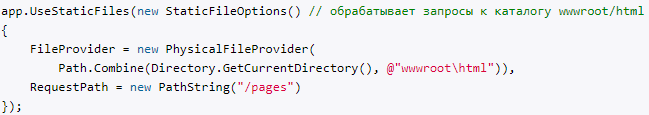
@”wwwroot\html”)),

RequestPath = new PathString(“/pages”)

});

В качестве параметра метод **UseDirectoryBrowser**() принимает объект **DirectoryBrowserOptions**, который позволяет настроить сопоставление путей к файлам с каталогами. Так, в данном случае путь типа http://localhost:56431/pages/ будет сопоставляться с каталогом "wwwroot\html".

Перегрузка метода **UseStaticFiles**() позволяет сопоставить пути с определенными каталогами:



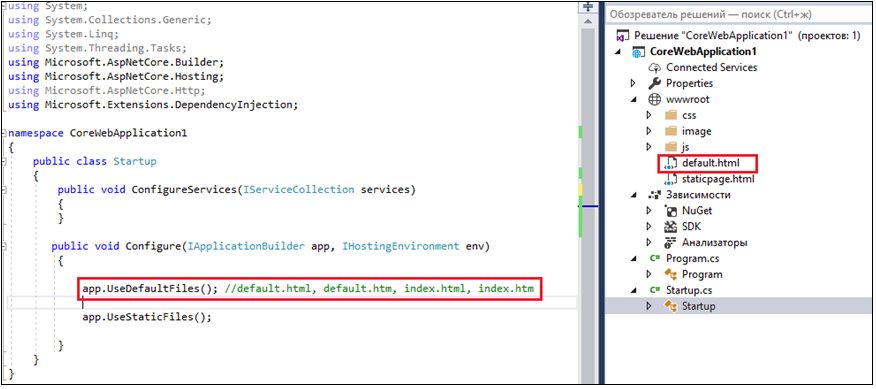
Метод **UseFileServer()** объединяет функциональность сразу всех трех вышеописанных методов UseStaticFiles, UseDefaultFiles и UseDirectoryBrowser:



**Добавление заголовков:**

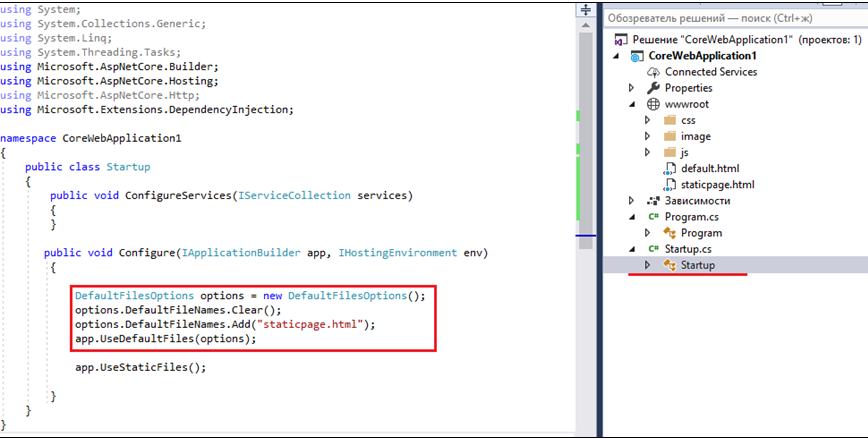


**Стартовые страницы:**



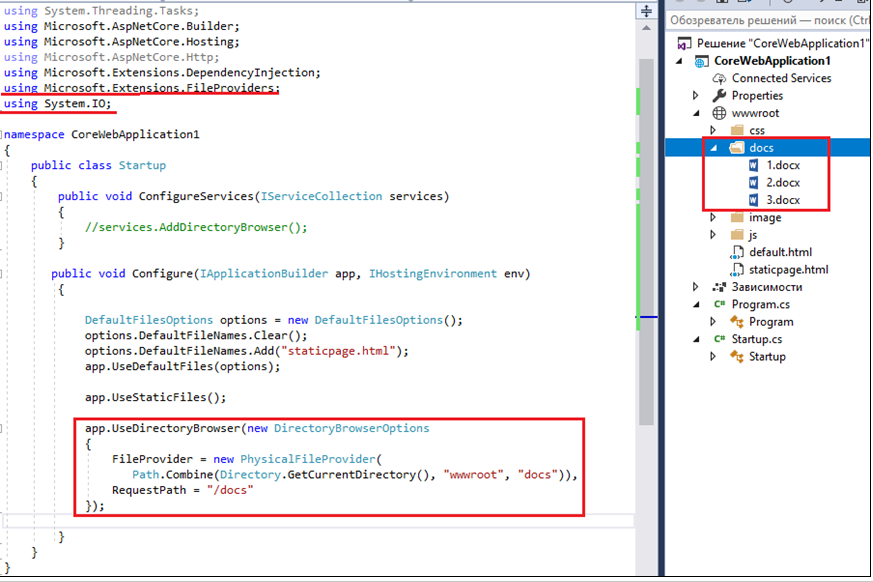
В данном случае у нас будут использоваться страницы default.html, default.htm, index.html, index.htm в качестве страниц по умолчанию.

Для пользовательских страниц по умолчанию используется следующий подход:

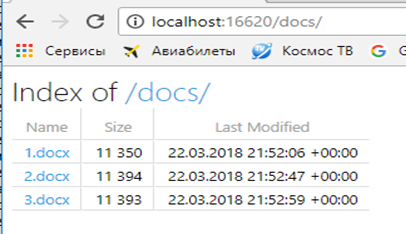


Будет использована страница staticpage.html, как страница по умолчанию.

**Файлы для скачивания:**



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:

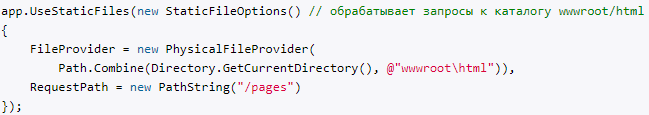


И может скачать файлы из данного каталога.

15. Платформа ASP.NET CORE: приложение для скачивания статических файлов.

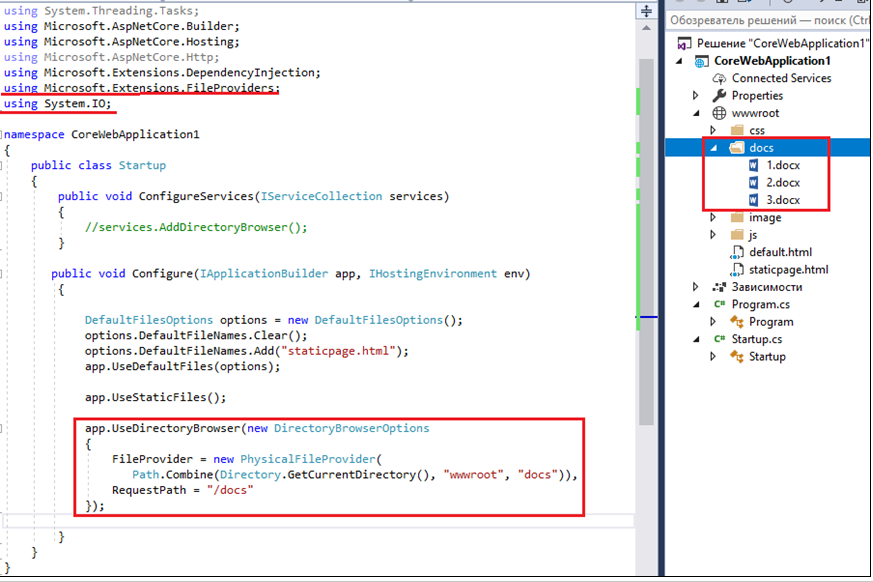
**Где образуются статические файлы и что именно является статическим файлом? -** Статические файлы хранятся в wwwroot по умолчанию. Пример файлов: html, css, js, картинки, документы и т.п.

**Как поменять директорию по умолчанию для статических файлов? -** Так что ли:

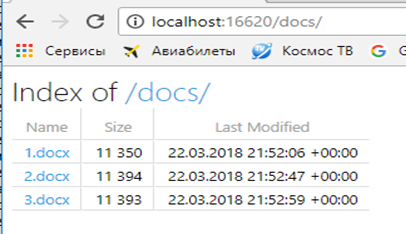


**Как добраться до css если он лежит в папке css в wwwroot? -** Прописать в URI http://localhost:port/css/file.css

**Файлы для скачивания:**



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:

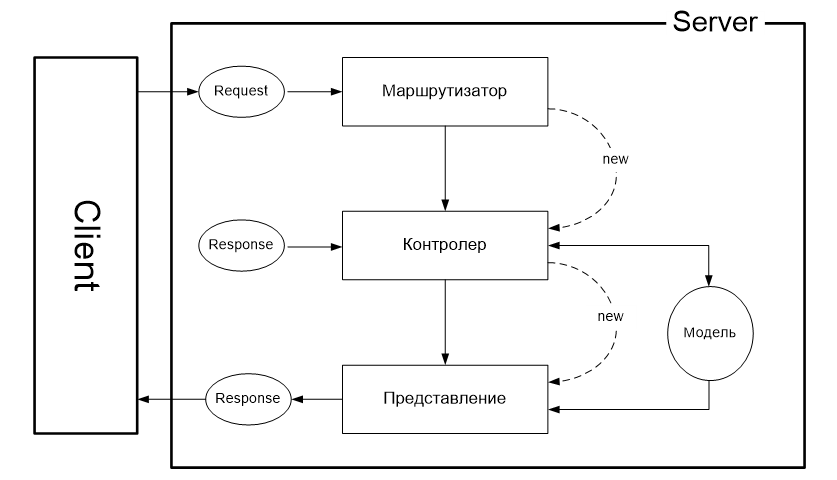
zerocomission

И может скачать файлы из данного каталога.

16. Платформа ASP.NET CORE: структура MVC-приложения.

**MVC: Model-View-Controller** – архитектурный паттерн; Включает три компоненты:

1. модель – данные;
2. представление – отображение модели;
3. контролер – обработка запросов, координация взаимодействия модели и представления.
4. Маршрутизатор – выбор контроллера и действия.



**По Смелову:**

**Как запрос доходит до контроллера? -** тут нужно сказать, что битовая последовательность разбирается HTTP-сервером, создаются объекты Request, Response. Потом они проходят через Middleware. Потом наверно рассказать херню про маршрутизатор.

**Маршрутизатор (он в Middleware)** парсит запрос, разбирает URI и затем выбирает контроллер. С помощью таблицы маршрутизации, определяет имя контроллера и action, и после передает управление объекту фабрики контроллеров. Фабрика контроллеров создает объект класса контроллера и вызывает его action (метод контроллера). Всё это происходит с помощью **рефлексии.** Параметры action = параметры запроса. Action вызывает специальный компонент view-engine (razor-engine) (через метод View). Razor-engin’у передаётся имя файла с расширением cshtml. Razor-engine берёт файл и превращает его в объект c#. Методы объекта (execute) заполняют Response. Такой объект генерируется один раз, он есть всё время в памяти.

Строки в таблице маршрутизации просматриваются сверху вниз. Каждая строка содержит шаблон. Фабрикой контроллеров используется **рефлексия**. Рефлексия позволяет пройтись по метаданным сборки и определить, есть ли такой класс. Фабрику контроллеров можно подменить.

Как фабрика работает:

1. По метаданным сборки определяет, есть ли такой класс.

2. С помощью рефлексии создает объект этого контроллера.

3. Вызывает метод контроллера и передает ему параметры.

**Модель** использует EntityFramework.

EntityFramework – это реализация ORM-модели.

ORM – это методология построения интерфейса с базой данных, которая предполагает, что объектам базы данных ставится в соответствие программный объект. Таблице соответствует DbSet;

структуре таблицы – класс-модель,в котором прописаны свойства,

и каждое свойство соответствует столбцу таблицы;

строке соответствует элемент коллекции DbSet.

**Что такое репозиторий моделей или репозиторий в EF?** – это слой ПО, который надстраивается над API EF. Это реализация интерфейса (dll). Эти dll можно легко заменять. Имеет контекстный тип методов (имена типа FindStudents).

View **могут** иметь доступ к модели, но их **нельзя** пускать в модель. Контроллер должен получать данные через параметры. Зависимость между Model и Controller должна уменьшаться через **репозитории** или **DI(**dependency injection - автоматическое создание объектов, и управление их жизненым циклом).

**View**: Метод View() унаследован от класса Controller, вызывает razor-engine. Он получает в качестве параметра имя контроллера и имя action. В случае, если в метод View не был передан параметр, то имя view будет совпадать с именем action.

Razor-engine: компонент ASP.NET Core, задача которого взять cshtml и преобразовать его в объект C#. Абстрактный класс RazorPage. Объект имеет метод execute, который генерирует разметку и помещает body в response.

Cshtml = html + razor-разметка.

@model - тип

@Model - доступ к объекту модели

Заталкивается через ViewBag.

ViewData – можно обращаться через индексатор => ViewData[“A”]=ViewBag.A

ViewBag – обёртки над ViewData.

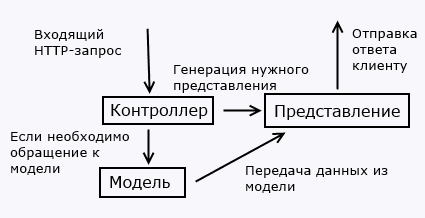
@inherits – производный от класса RazorPage.

Layout. Компоновка – это представление, предназначенное для макетирования страниц.

Из метанита:

* **Модель (model)**: описывает используемые в приложении данные, а также логику, которая связана непосредственно с данными, например, логику валидации данных. Как правило, объекты моделей хранятся в базе данных.  
  В MVC модели представлены двумя основными типами: модели представлений, которые используются представлениями для отображения и передачи данных, и модели домена, которые описывают логику управления данными.  
  Модель может содержать данные, хранить логику управления этими данными. В то же время модель не должна содержать логику взаимодействия с пользователем и не должна определять механизм обработки запроса. Кроме того, модель не должна содержать логику отображения данных в представлении.
* **Представление (view)**: отвечают за визуальную часть или пользовательский интерфейс, нередко html-страница, через который пользователь взаимодействует с приложением. Также представление может содержать логику, связанную с отображением данных. В то же время представление не должно содержать логику обработки запроса пользователя или управления данными.
* **Контроллер (controller)**: представляет центральный компонент MVC, который обеспечивает связь между пользователем и приложением, представлением и хранилищем данных. Он содержит логику обработки запроса пользователя. Контроллер получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления, наполненного данными моделей.

Отношения между компонентами паттерна можно описать следующей схемой:

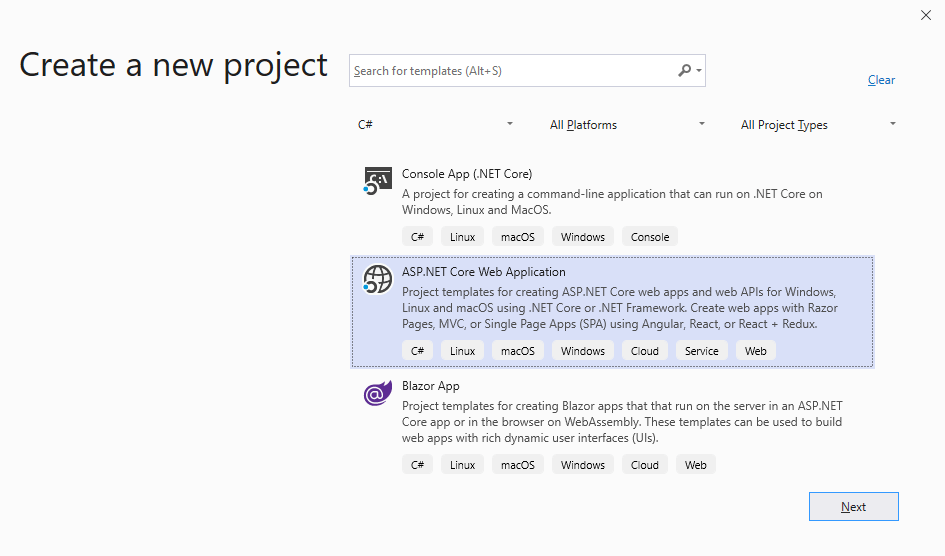
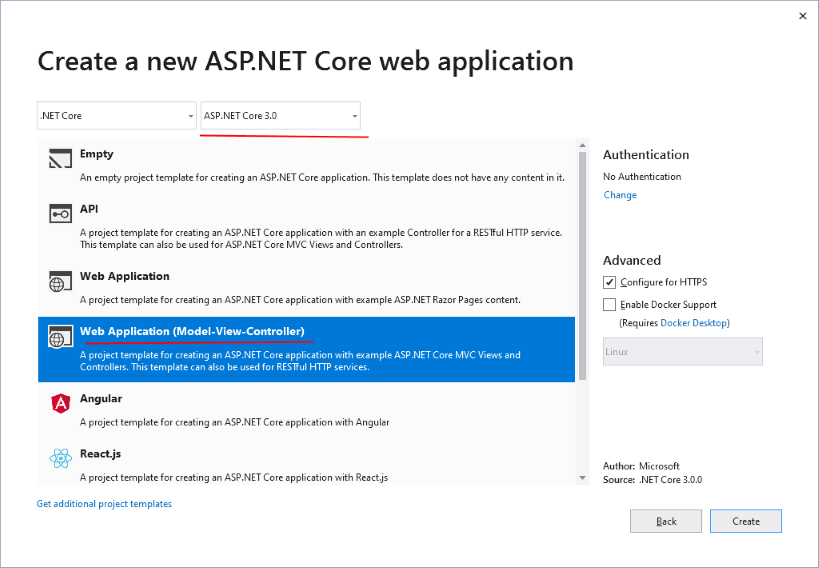


В этой схеме модель является независимым компонентом - любые изменения контроллера или представления никак не влияют на модель. Контроллер и представление являются относительно независимыми компонентами. Так, из представления можно обращаться к определенному контроллеру, а из контроллера генерировать представления, но при этом нередко их можно изменять независимо друг от друга.

Такое разграничение компонентов приложения позволяет реализовать концепцию разделение ответственности, при которой каждый компонент отвечает за свою строго очерченную сферу. В связи с чем легче построить работу над отдельными компонентами. И благодаря этому приложение легче разрабатывать, поддерживать и тестировать отдельные компоненты. Допустим, если нам важна визуальная часть или фронтэнд, то мы можем тестировать представление независимо от контроллера. Либо мы можем сосредоточиться на бэкэнде и тестировать контроллер.

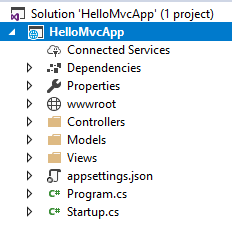
Однако неверно полностью ассоциировать всю платформу ASP.NET Core с MVC. MVC - это лишь паттерн, который реализуется в рамках платформы. Мы можем создать проект по шаблону Empty где не будет никаких контроллеров, моделей, представлений, где будет один класс Startup. И через этот класс построить всю обработку запроса. Но естественно применение MVC облегчает разработку приложений.

Для создания проекта, применяющего MVC, нам надо выбрать шаблон **ASP.NET Core Web App (Model-View-Controller)**:

Оставим все настройки по умолчанию и нажмем на ОК. И Visual Studio создаст новый проект MVC.

Структура создаваемого проекта будет отличаться от структуры проекта по типу Empty. В частности, мы увидим ряд новых папок и файлов:



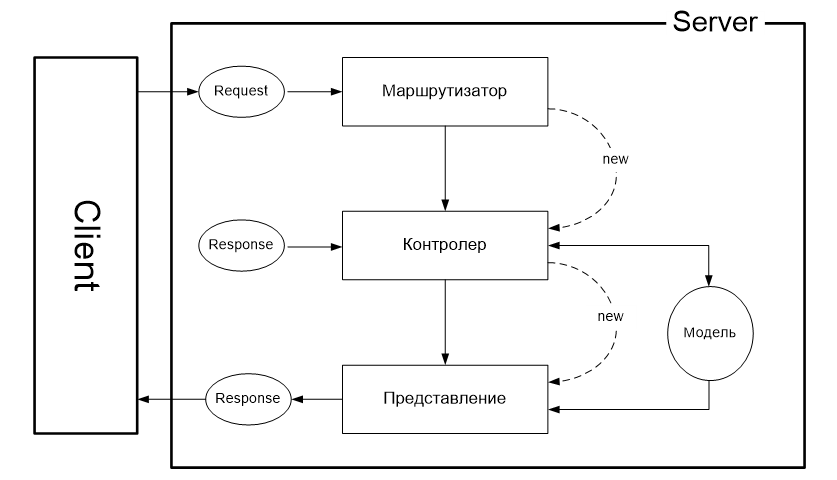
* **Dependencies**: все добавленные в проект пакеты и библиотеки
* **wwwroot**: этот узел (на жестком диске ему соответствует одноименная папка) предназначен для хранения статических файлов - изображений, скриптов javascript, файлов css и т.д., которые используются приложением. Цель добавления этой папки в проект по сравнению с другими версиями ASP.NET, состоит в разграничении доступа к статическим файлам, к которым разрешен доступ со стороны клиента и к которым доступ запрещен.
* **Controllers**: папка для хранения контроллеров, используемых приложением
* **Models**: каталог для хранения моделей
* **Views**: каталог для хранения представлений
* **appsettings.json**: хранит конфигурацию приложения
* **Program.cs**: файл, определяющий класс Program, который инициализирует и запускает хост с приложением.
* **Startup.cs**: файл, определяющий класс Startup, с которого начинается работа приложения. То есть это входная точка в приложение.

Фактически эта та же структура, что и у проекта по типу Empty за тем исключением, что здесь также добавлены по умолчанию папки для ключевых компонентов фреймворка MVC: контроллеров и представлений. А также есть дополнительные узлы и файлы для управления зависимостями клиентской части приложения.

17. Платформа ASP.NET CORE: MVC-приложение, маршрутизатор, атрибуты Route, констрейны маршрутов.

**MVC: Model-View-Controller** – архитектурный паттерн; Включает три компоненты:

1. модель – данные;(есть 3 типа модели: model-это модель внешнего источника данных, action model- это набор параметров, которые поступают в акшин, view model - это набор данных, которые можно передать во view)
2. представление – отображение модели;
3. контролер – обработка запросов, координация взаимодействия модели и представления.
4. Маршрутизатор – выбор контроллера и действия.



<https://metanit.com/sharp/aspnet5/11.1.php>

**Зачем MVC?**: разгр. логику, проще расш. приложение, сделать компоненты независимыми( тоесть замена одного компонента не поввлияет на другие компоненты)

**Как запрос доходит до контроллера? -** тут нужно сказать, что битовая последовательность разбирается HTTP-сервером, создаются объекты Request, Response. Потом они проходят через Middleware. Потом наверно рассказать херню про маршрутизатор.

**Маршрутизатор (он в Middleware)** парсит запрос, разбирает URI и затем выбирает контроллер. С помощью таблицы маршрутизации, определяет имя контроллера и action, и после передает управление объекту фабрики контроллеров. Фабрика контроллеров создает объект класса контроллера и вызывает его action (метод контроллера). Всё это происходит с помощью **рефлексии.** Параметры action = параметры запроса. Action вызывает специальный компонент view-engine (razor-engine) (через метод View). Razor-engin’у передаётся имя файла с расширением cshtml. Razor-engine берёт файл и превращает его в объект c#. Методы объекта (execute) заполняют Response. Такой объект генерируется один раз, он есть всё время в памяти.

Строки в таблице маршрутизации просматриваются сверху вниз. Каждая строка содержит шаблон. Фабрикой контроллеров используется **рефлексия**. Рефлексия позволяет пройтись по метаданным сборки и определить, есть ли такой класс. Фабрику контроллеров можно подменить.

Как фабрика работает:

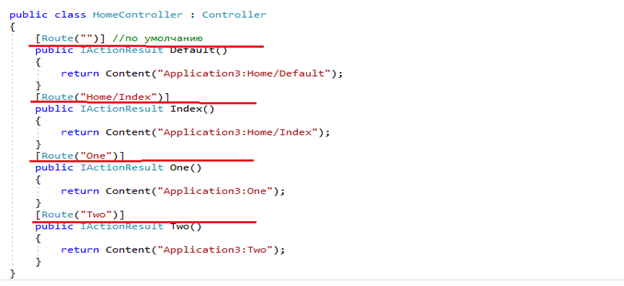
1. По метаданным сборки определяет, есть ли такой класс.

2. С помощью рефлексии создает объект этого контроллера.

3. Вызывает метод контроллера и передает ему параметры.

Какие есть типы маршрутизации?- табличные(эндпоинты), атрибутные(route)

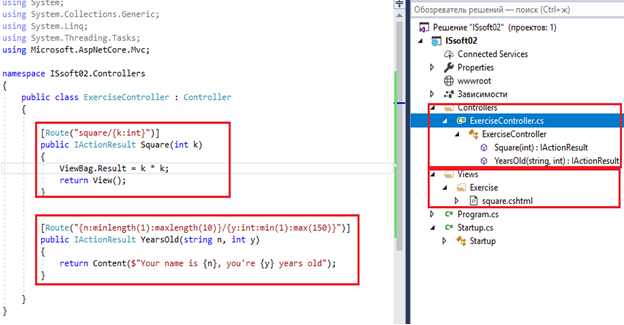
**Атрибут Route**

****

**Может быть несколько:**

****

**Ограничения(КОНСТРЕЙН!!!!!):**

****

18. Платформа ASP.NET CORE: MVC-приложение, контроллер, жизненный цикл контроллера. событие OnAction, атрибуты HttpGet, HttpPost, …, AcceptVerb, принцип обработки параметров запроса, методы передачи данных в представление.

<https://metanit.com/sharp/aspnet5/5.1.php>

**Жизненный цикл контроллера**:

1. **Получение запроса:** Получаем **побитовый** запрос, он обрабатывается маршрутизатором, маршрутизатор решает какой контроллер использовать
2. **Создание контроллера:** Создаем экземпляр контроллера с помощью фабрики контролеров и **рефлексии.**
3. **Выполнение действий контролера**
4. **Возврат ответа**
5. **Уничтожение контроллера**

Для создания контроллера достаточно унаследовать свой класс от базового класса Controller. Но если необходима некоторая общая логика, можно определить свой базовый класс контроллера и уже от него наследовать остальные контроллеры. Либо можно переопределить некоторые методы базового класса Controller.

В базовом классе Controller кроме прочих есть следующие методы:

- **OnActionExecuting()** выполняется при вызове метода контроллера до его непосредственного выполнения;

- **OnActionExecuted()** выполняется после выполнения метода контроллера;

- **OnActionExecutionAsync()** представляет асинхронную версию метода OnActionExecuting().

Методы в рамках одного действия могут обслуживать разные запросы. Для указания метода запроса HTTP нам надо применить к методу один из атрибутов: **[HttpGet], [HttpPost], [HttpPut], [HttpDelete] и [HttpHead]**. Если атрибут явным образом не указан, то метод может обрабатывать все типы запросов: GET, POST, PUT, DELETE и др.

Представляет атрибут, определяющий, на какие HTTP-команды будет отвечать метод действия. Т.е. в параметрах атрибута можно указать необходимые методы.

// Метод действия, который обрабатывает только GET и POST запросы

[("GET", "POST")]

**Основные способы передачи параметров в метод действия**

**1.** **Через параметры маршрута** products/{id}

2. **Через строку запроса** в URL после знака вопроса (?) и разделяются амперсандами (&).

**3.** **Через тело запроса**

**4.** **Через заголовки HTTP**

**5.** **Через форму данных**

Параметры могут представлять примитивные типы (int, string) или же более сложные классы. Передавать значения параметров можно различными способами.

При отправке GET-запроса значения передаются через строку запроса. Например,<http://localhost:1111/Home/Sum?a=2&b=4>.

public string Sum(int a, int b) {

return $”Sumr8 {a} и {b}”; }

Если же не используем параметры в строке запроса, то для параметров будут передаваться значения по умолчанию:

[http://localhost:1111/Home/Sum](http://localhost:1111/Home/Sum?a=2&b=4)/

public string Sum(int a = 2, int b = 4) {

return $”Sum {a} и {b}”; }

Система привязки MVC сопоставляет параметры запроса и параметры метода по имени, при этом также должно быть соответствие по типу передаваемых данных.

***Передача сложных объектов***

Класс Input определяет 3 свойства. Теперь в контроллере метод А6 принимает параметр Input. Здесь параметры строки запроса должны соответствовать по имени свойствам объекта. Регистр названий при этом не учитывается.

***Передача массивов***

<http://localhost:1111/Home/Sum?nums=1&nums=2&nums=3>

Теперь метод Sum принимает массив объектов класса Geometry:

[*http://localhost:57086/Home/Sum?geoms[0].altitude=10&geoms[0].height=3&geoms[1].altitude=16&geoms[1].height=2*](http://localhost:57086/Home/Sum?geoms%5B0%5D.altitude=10&geoms%5B0%5D.height=3&geoms%5B1%5D.altitude=16&geoms%5B1%5D.height=2)

***Передача данных в запросе POST***

Как правило, POST-запросы отправляются через формы, но принципы передачи данных такие же как и у GET-запросов.

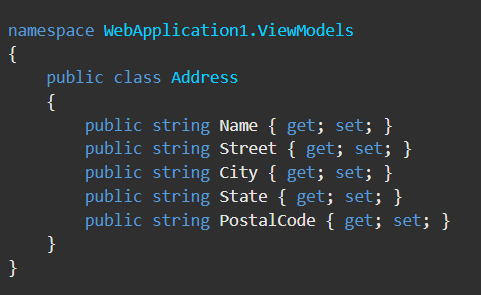
Для начала в представлении определяем форму, которая устанавливает метод отправки - post, адрес отправки - Home/Area и два поля для ввода чисел. Чтобы система могла связать параметры метода и данные формы, необходимо, чтобы атрибуты name у полей формы соответствовали названиям параметров. Правила привязки те же.

**Способы передачи данных на View:**

1) ViewData (глобальный key-value словарь, выглядит примерно так: ViewData[“Foo”])

2) ViewBag (обертка над ViewData, но использует свойства вроде ViewBag.Foo)

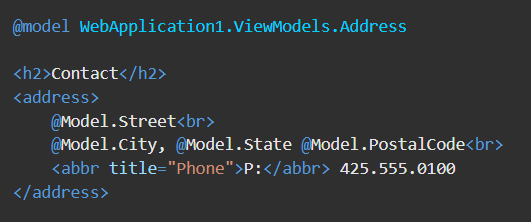
3) ViewModel. Создать специальный класс для передачи данных:



Передать объект класса как параметр метода View():



Доступ:



4) Передача модели через конструктор

5) TempData???? Говорить только если спросит

@model означает, что данные, переданные в представление, имеют тип, следующий за данной директивой

@Model - это обращение к публичным свойствам представления (объекта представления) (например, @Model.a обращается с свойству a модели)

Даже если параметры передаются через конструктор представления, они сами запихиваются во ViewBag (\*точнее, записываются в свойство ViewBag.Model\*)

19. Платформа ASP.NET CORE: MVC-приложение, представление, жизненный цикл представления, хэлперы, частичные представления, директивы @model, @inherits, @function, @inject, @using, @addTagHelper, @removeTagHelper, @RenderSection, @RenderBody.

[**https://metanit.com/sharp/aspnet5/5.1.php**](https://metanit.com/sharp/aspnet5/5.1.php)

**Tag Helpers –** классы предназначенные для генерации Razor-разметки, расширяющей семантику html-разметки. В Razor: новый тег или html-тег с новыми атрибутами (при этом стандартные атрибуты могут быть сохранены).

**Жизненный цикл представления:**

Через reflection ищем папку views, потом ищем папку с именем контроллера, потом файл с именем экшена …  
 Action вызывает специальный компонент view-engine (razor-engine) (через метод View). Razor-engin’у передаётся имя файла с расширением cshtml. Razor-engine берёт файл и превращает его в объект c#. Методы объекта (execute) заполняют Response. Такой объект генерируется один раз, он есть всё время в памяти.

**Директива**- это инструкции для указания необязательных настроек, таких как регистрация пользовательского элемента управления и языка страниц.

Как правило, директива изменяет способ анализа представления или открывает доступ к дополнительным функциям.

**@model** означает, что данные, переданные в представление, имеют тип, следующий за данной директивой

В C# оператор *using* позволяет обеспечить использование какого-то объекта. В Razor для создания вспомогательных функций HTML, содержащих дополнительное содержимое, используется тот же механизм. В следующем коде вспомогательные функции HTML используют оператор **@using** для создания тега <form>:

| @using (Html.BeginForm())  {  <div>  Email: <input type="email" id="Email" value="">  <button>Register</button>  </div>  } |
| --- |
| **@using директива с блоком кода**: Здесь **@using** используется не для импортирования пространств имен, а для создания блока кода, в котором инициализируется объект, который будет автоматически освобожден в конце блока. Это похоже на использование **using** в C# для управления ресурсами |

**Или**

Директива @using в ASP.NET Core MVC используется в представлениях Razor **для импортирования пространств имен**, что позволяет вам использовать классы и методы из этих пространств имен без необходимости указывать их полные квалифицированные имена

@using System

**Директива @using в файлах \_ViewImports.cshtml**

Чтобы не повторять директиву **@using** в каждом представлении, вы можете определить общие директивы в файле **\_ViewImports.cshtml**, который будет применяться ко всем представлениям в текущей папке и вложенных папках:

@using MyApp.Models

Директива **@functions** позволяет определить функции, которые могут применяться в представлении.

| @*functions* {  public string GetHello()  {  return "Hello";  }  }  <div>From method: @GetHello()</div> |
| --- |

**@*inherits*** – позволяет создать собственный базовый класс для представления; базовый класс должен быть производным от RazorPage или RazorPage<ModelType>.:

Директива **@inherits** в ASP.NET Core MVC используется в Razor представлениях для указания базового класса, от которого будет наследоваться представление.

| using Microsoft.AspNetCore.Mvc.Razor;  public abstract class InhClass<TModel> : RazorPage<TModel>  {  public string Text{ get; } = "Example of inherits";  } |
| --- |

*CSHTML*

| @inherits CustomRazorPage<TModel>  <div>Custom text: @CustomText</div> |
| --- |

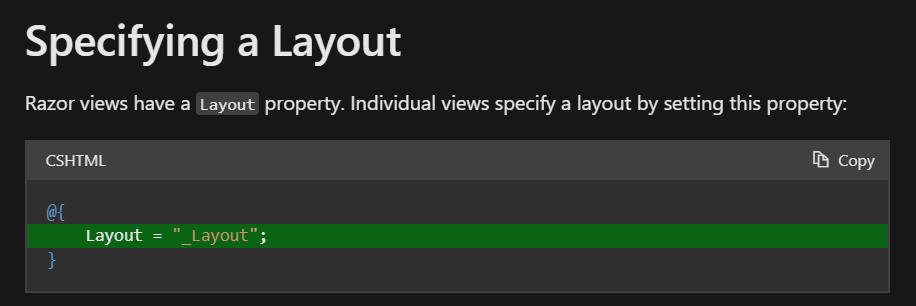
**@*inject***  - директива для внедрения зависимости в представление

| public class VuiwInject{  private string txt=”empty”;  public ViewInject()  { Console.WriteLine(“constructor”);}    public string Set(string s)  { Console.WriteLine(this.txt=s);  retrun this.txt; }    public string Get(string s)  { Console.WriteLine(this.txt);  retrun this.txt; }  } | public void ConfigureServices(IserviceCollection services){  services.addMvc();  services.AddSingleton<Namespace.Inject.ViewInject>()  }    CSHTML  @***inject*** Namespace.Injects.ViewInject VI  @{ Layout=null;}  <body>  <p>VI get = @VI.Get()</p> @{VI.Set(“ok”);}  <p>VI get = @VI.Get()</p>  </body> |
| --- | --- |

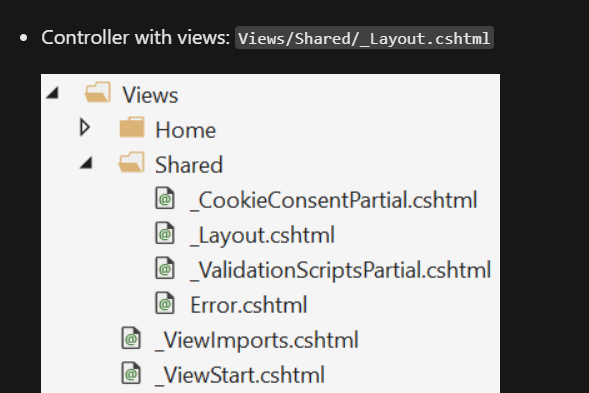
@addTagHelper - подключение TagHelper из указанной сборки

@removeTagHelper - отключение указанных TagHelper

Layout. Компоновка – это представление, предназначенное для макетирования страниц.

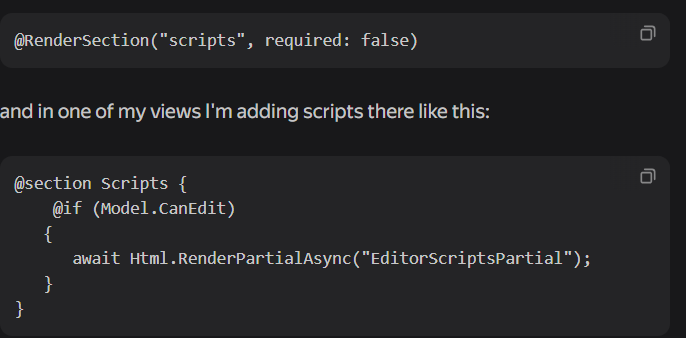


Местонахождение:



RenderBody() – метод представления-макета, предназначенный для отображения содержимого страниц, использующих данный макет.

RenderSection() – метод представления-макета, предназначенный для рендеринга секций. Секции могут определять местоположение содержимого в макете.



Partial View (частичное представление) – фрагмент Razor-разметки, предназначенный для повторного применения; любое представление может быть частичным представлением, но в этом случае не применяется \_ViewStart.cshtml.

Partial, PartialAsync, RenderPartial, RenderPartialAsync

20. Платформа ASP.NET CORE: MVC-приложение, модель источника данных. Паттерн ORM.

**Модель** использует EntityFramework.

EntityFramework – это реализация ORM-модели.

**ORM** – это методология построения интерфейса с базой данных, которая предполагает, что объектам базы данных ставится в соответствие программный объект. Таблице соответствует DbSet; структуре таблицы – класс-модель, в котором прописаны свойства, и каждое свойство соответствует столбцу таблицы; строке соответствует элемент коллекции DbSet.

- **таблице** соответствует **DBSet**

- **базе данных** - **ContextDB**, в котором прописаны DBSetы

- **структуре таблицы** - **модель, класс**, в котором прописаны свойства, каждое из которых соответствует столбцу

- **строке таблицы** - **элемент коллекции DBSet**

- репозиторий - слой ПО, надстраивающийся над API Entity Framework

**Что такое репозиторий моделей или репозиторий в EF?** – это слой ПО, который надстраивается над API EF. Это реализация интерфейса (dll). Эти dll можно легко заменять. Имеет контекстный тип методов (имена типа FindStudents).

View могут иметь доступ к модели, но их нельзя пускать в модель. Контроллер должен получать данные через параметры. Зависимость между Model и Controller должна уменьшаться через репозитории или DI.

***Какая-то Смеловская херня на всякий случай:***

**DTO** – класс, содержит данные **без логики** (могут быть свойства и поля). Используется для передачи данных между приложениями или слоями приложения. JSON объект – сериализация dto.

**Value Object** – при DDD подходе, данные объединяются в объекты, используются для передачи, агрегации данных. Похож на Entity Object, но **не содержит идентификатора**. Два Value Object между собой не различаются. Может содержать методы.

**POCO** (Plain old CLR Object): аналог POJO (Java). Может содержать **логику** (методы), **может быть и Value и Entity.**

DAO: Data Access Object: паттерн проектирования – абстрактный интерфейс к БД, класс содержит CRUD.

DTO – экземпляр сущности, отделяет бизнес-логику от механизмов сохранения. Бизнес-объект работает не с бд, а с сущностями DTO.

**Abstract Factory:** DAO можно представить как фабрику.

Repository: паттерн проектирования. Находится на стороне бизнес-логики. Набор классов и методов, которым соответствует некоторая логика. **Не** **всегда** реализует CRUD. Настроен на бизнес. Ассоциируется с Domain Drive Design.

Интерфейс – это контракт.

**Data Mapper** – сопоставляет данные на уровне бизнес-логики на данные, которые лежат в хранилище данных.

ORM, DAO – идут со стороны бд. Строим модель над данными. Паттерн проектирования, применяется для работы с данными. Концепция бд/ооп, ООП-интерфейс с бд.

**Eager loading** – загрузка связанных объектов сущности одновременно с основным объектом.

**Lazy loading** – ленивая загрузка, связанные объекты только при необходимости, оптимизация процесса чтения.

**Identity Map** – жадное получение объекта из бд 1 раз в рамках сессии, позволяет избегать противоречия в данных, возникает из-за запросов.

ODBC – платформозависимый (это вроде драйвер или что это вообще)

21. Платформа ASP.NET CORE: внедрение зависимостей.

<https://metanit.com/sharp/aspnet5/6.1.php>

DI – механизм, позволяющий создавать в автоматическом режиме объекты некоторого класса или объекты реализации интерфейса и задавать их жизненный цикл. Часть паттерна IoC.

Нередко для установки зависимостей в подобных системах используются специальные контейнеры - IoC-контейнеры (**Inversion of Control**). Такие контейнеры служат своего рода фабриками, которые устанавливают зависимости между абстракциями и конкретными объектами и, как правило, управляют созданием этих объектов.

И если раньше в ASP.NET 4 и других предыдущих версиях надо было использовать различные внешние IoC-контейнеры для установки зависимостей, такие как Ninject, Autofac, Unity, Windsor Castle, StructureMap, то ASP.NET Core уже имеет встроенный контейнер внедрения зависимостей, который представлен интерфейсом **IServiceProvider**. А сами зависимости еще называются сервисами, собственно поэтому контейнер можно назвать **провайдером сервисов**. Этот контейнер отвечает за сопоставление зависимостей с конкретными типами и за внедрение зависимостей в различные объекты.

**Есть два цикла:** **Transient** (новый экземпляр будет создаваться при каждом упоминании) и **Singleton** (будет сохранять состояние между вызовами).

Создание объекта и передача его в конструктор происходит в **фабрике контроллеров**. Т.е. если у конструктора контроллера есть параметр, который был добавлен как DI, то создается новый объект и передается контроллеру в конструктор.

22. Платформа ASP.NET CORE: простейшее Web API-приложение.

**Web API** - веб сервис. **Web сервис** - такие же веб-приложения ,которые предназначены для работы других веб-приложений

Требования к сервисам:

- на стороне сервера кешировать нельзя

Средство Web API основано на добавлении в приложение ASP.NET MVC Framework контроллера специального вида. Эта разновидность контроллеров, которая называется контроллером API, обладает двумя характеристиками:

● Методы действий возвращают объекты моделей, а не объекты типа ActionResult.

● Методы действий выбираются на основе HTTP-метода, используемого в запросе.

23. Платформа ASP.NET CORE: реализация REST-приложения.

**REST** (Representational State Transfer): Архитектурный стиль взаимодействия компонентов распределенного приложения.

**RESTful**: описание web-службы, поддерживающей REST-интерфейс в полном объеме (со строгим соблюдением правил).

**REST**: нет официального стандарта, но REST использует стандарты HTTP, URL/URI, XML, JSON.

**Какая альтернатива для REST? - RPC !!! ВАЖНО !!!**

**REST:** два основныхтипа ресурса – коллекции и элемент коллекции: /api/users, /api/users/288.

REST в ASP.NET Core реализуется **WebAPI** приложением !!!!!!!

шесть обязательных ограничений:

1. модель клиент-сервер;
2. отсутствие состояния на стороне сервера, сохранение состояния допускается на стороне **клиента**, допускается сохранение состояния в другом сервисе (например, в БД);
3. кэширование на стороне клиента, сервер явно управляет кэшированием;
4. единообразие интерфейсов (идентификация ресурсов (URI), манипуляция ресурсами через представления, самодостаточные сообщения, HATEOAS);
5. для клиента сервер должен представляться конечным; (многоуровневость архитектуры )
6. код по требованию: допускается (необязательно) выгрузка на клиент апплетов или сценариев для расширения его функциональности

**HATEOAS: Hypermedia As The Engine Of Application State** – гипермедиа в качестве управления состоянием.

**Гипермедиа**: технология обработки, структурирования информации и произвольного доступа к ее элементам с помощью гиперсвязей (Тед Нильсон, 1965), WWW – реализация гипермедиа.

HATEOAS: Hypermedia As the Engine of Application State.

Для чего? Позволяет получить инфу и настроить клиент. Это дает возможность на стороне сервера менять ссылки. Принцип формата ответов, в которых идут дополнительные ссылки на ресурсы этого сервера.

Определение тут наверно вот это: **принцип формата ответов, в которых идут дополнительные ссылки на ресурсы этого сервера.**

**16. REST:** **общепринятые правила**

* 1. Общий префикс для всех ресурсов сервиса … /API/…,

[http://API.BSTU.BY/](http://api.bstu.by/)...

* 1. Два типа ресурсов: коллекция (users, students, …), элемент коллекции /api/users/238, /api/students/ef3d26.
  2. Поддержка иерархических связей между ресурсами …/api/users/238/cars/aah4899
  3. Ограничить количество HTTP-статусов, сопроводить сообщение дополнительным кодом (например 20003, 404001,…),сделать отдельный ресурс (HATEOAS link) для пояснения ошибок <http://ccc/api/errors/20003>.
  4. Подавление статуса ответа …/api/students/ef3d26?status\_code=200. (когда нужно принудительно установить определенный статус ответа для специфических случаев. )
  5. Версионность /api/students/ef3d26?v=7. ( Использование версионирования в URI позволяет поддерживать совместимость и изменять API без нарушения работы старых клиентов.)
  6. Постраничное получение данных: параметры limit, offset.
  7. Сортировка: параметр sort для сортировки результатов запроса
  8. Все фильтры вынести за знак вопроса: …/api/students?minbday=1998101&maxbday=20001231&gender=m.
  9. Пользователь получает только то, что хочет (Клиенты могут выбирать только необходимые поля для получения): …/api/students?field=bday,surname,gender.
  10. Обозначать в запросе формат сообщений через расширение в URI (желательна поддержка нескольких форматов):

…/api/students.**json**?field=bday,surname,gender;

один из форматов должен быть по умолчанию; могут применяться заголовки Accept и Content-Type со значениями application/xml и application/json для запроса или обозначения в ответе формата.

* 1. Глобальный поиск: ../api/search?q=19600107+Иванов.
  2. Документация.

**REST:** недостатки

* нет общепризнанного стандарта RESTful API;
* не все браузеры поддерживают полный словарь REST-методов (PUT, DELETE); на практике часто используется только GET и POSТ(insert, delete, update);
* не однозначны коды состояний.

1. **REST**: **Representational State Transfer** - передача состояния представления.

2. **REST**: представление – это URI (идентификатор ресурса), управление ресурсом с помощью глаголов (GET, POST, PUT, DELETE).

3. **REST**: Архитектурный стиль взаимодействия компонентов распределенного приложения.

4. **REST**: автор **Roy Fielding**, описан в диссертации «Архитектурные стили и дизайн сетевых программных архитектур», 2000 г.

5. **REST**: реализация (альтернатива) RPC.

6. **RESTful**: описание web-службы, поддерживающей REST-интерфейс в полном объеме (со строгим соблюдением правил).

7. **REST**: нет официального стандарта, но REST использует стандарты HTTP, URL/URI, XML, JSON.

8. **REST:** два основныхтипа ресурса – коллекции и элемент коллекции: /api/users, /api/users/288.

9. **REST**: шесть обязательных ограничений:

- модель клиент-сервер;

- отсутствие состояния на стороне сервера, сохранение состояния допускается на стороне клиента, допускается сохранение состояния в другом сервисе (например, в БД);

- кэширование на стороне клиента, сервер явно управляет кэшированием;

- единообразие интерфейсов (идентификация ресурсов, манипуляция ресурсами через представления, самодостаточные сообщения, HATEOAS);

- для клиента сервер должен представляться конечным;

- код по требованию: допускается (необязательно) выгрузка на клиент апплетов или сценариев для расширения его функциональности.

10. **REST:** GET (select), POST(insert), PUT (update), DELETE (delete).

13. **HATEOAS: Hypermedia As The Engine Of Application State** – гипермедиа в качестве управления состоянием.

14. **Гипермедиа**: технология обработки, структурирования информации и произвольного доступа к ее элементам с помощью гиперсвязей (Тед Нильсон, 1965), WWW – реализация гипермедиа.

24. Платформа ASP.NET CORE: Web API/MVC-приложение.

**Web API в ASP.NET Core**

Web API предназначен для создания HTTP-сервисов, которые могут использоваться различными клиентами, такими как веб-приложения, мобильные приложения и другие сервисы. Он позволяет создавать API, которые возвращают данные в формате JSON, XML и других форматах, обычно используя HTTP-методы (GET, POST, PUT, DELETE и т.д.).

**Особенности Web API:**

1. **Архитектурный стиль REST**: Web API часто используется для реализации RESTful API, предоставляющего структурированный способ доступа к данным через стандартные HTTP-методы.
2. **Контроллеры**: В Web API основной компонент – это контроллеры (**Controllers - атрибут [ApiController]**), которые обрабатывают HTTP-запросы и формируют HTTP-ответы.
3. **Модели данных**: Входные и выходные данные передаются в виде моделей данных (DTO - Data Transfer Objects), которые сериализуются в JSON или другие форматы.
4. **Маршрутизация**: Маршрутизация в Web API определяется атрибутами маршрутизации или настройками в **Startup.cs**, что позволяет точно задать, какой контроллер и метод должны обрабатывать определенный запрос.

**MVC (Model-View-Controller) в ASP.NET Core**

MVC в ASP.NET Core предназначен для создания веб-приложений с шаблоном Model-View-Controller, который отделяет бизнес-логику от пользовательского интерфейса и данных.

**Особенности MVC:**

1. **Разделение на слои**: Модель (**Model**) представляет данные и бизнес-логику, Представление (**View**) отвечает за отображение данных пользователю, а Контроллер (**Controller**) обрабатывает пользовательские запросы и управляет взаимодействием между Моделью и Представлением.
2. **Рендеринг представлений**: Представления в MVC часто создаются с использованием Razor синтаксиса для HTML с возможностью внедрения кода C#.
3. **Аннотации и атрибуты**: Атрибуты маршрутизации и аннотации используются для настройки маршрутизации и поведения контроллеров и их методов.

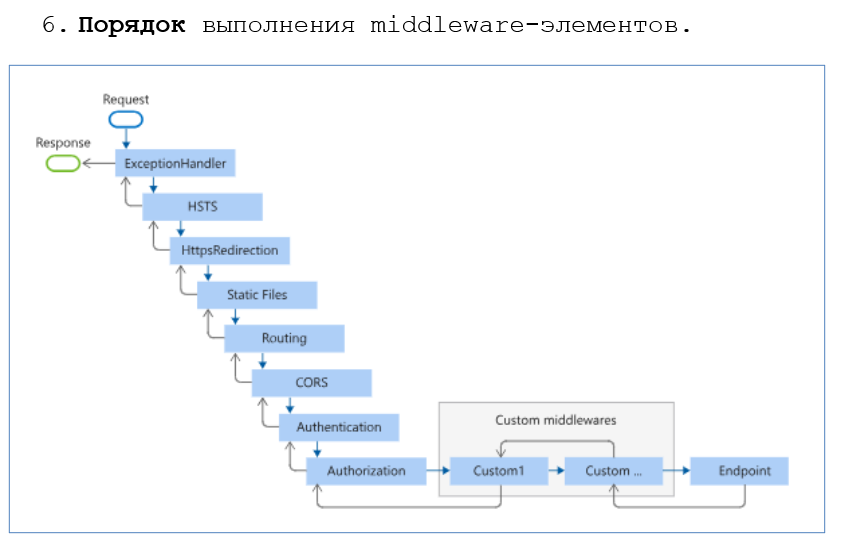
**Различия между Web API и MVC**

* **Цель**: Web API предназначен для создания HTTP-сервисов и API, тогда как MVC используется для создания веб-приложений с шаблоном Model-View-Controller.
* **Формат ответов**: Web API обычно возвращает данные в формате JSON или XML, в то время как MVC часто возвращает HTML для отображения страниц.
* **Маршрутизация**: Web API использует атрибуты маршрутизации или настройки в **Startup.cs**, тогда как MVC часто использует соглашения и шаблоны маршрутизации.

25. Платформа ASP.NET CORE: middleware, разработка объектов middleware.

<https://habr.com/ru/companies/otus/articles/528692/>

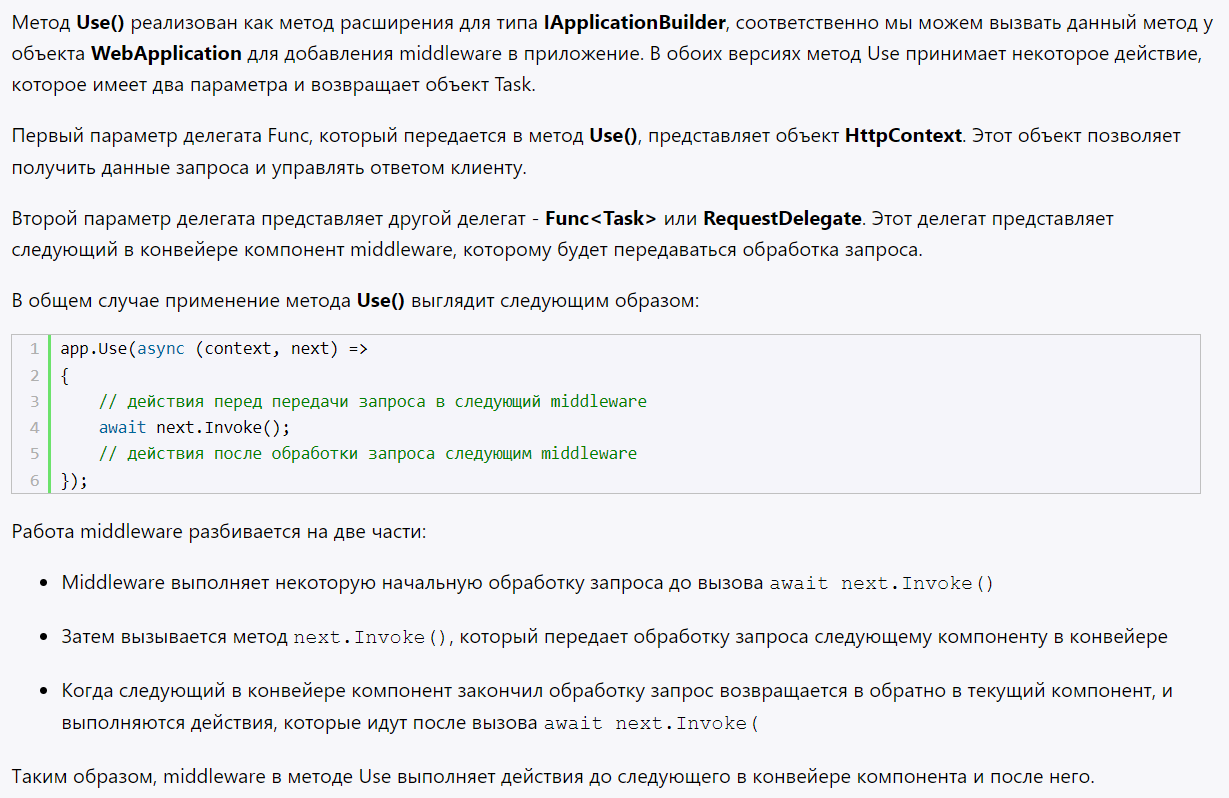
**middleware** - конвейер обработки запроса



**UseEndpoints** конечная точка конвейера.

app.Map… – добавляет конечный элемент конвейера обработки запроса.

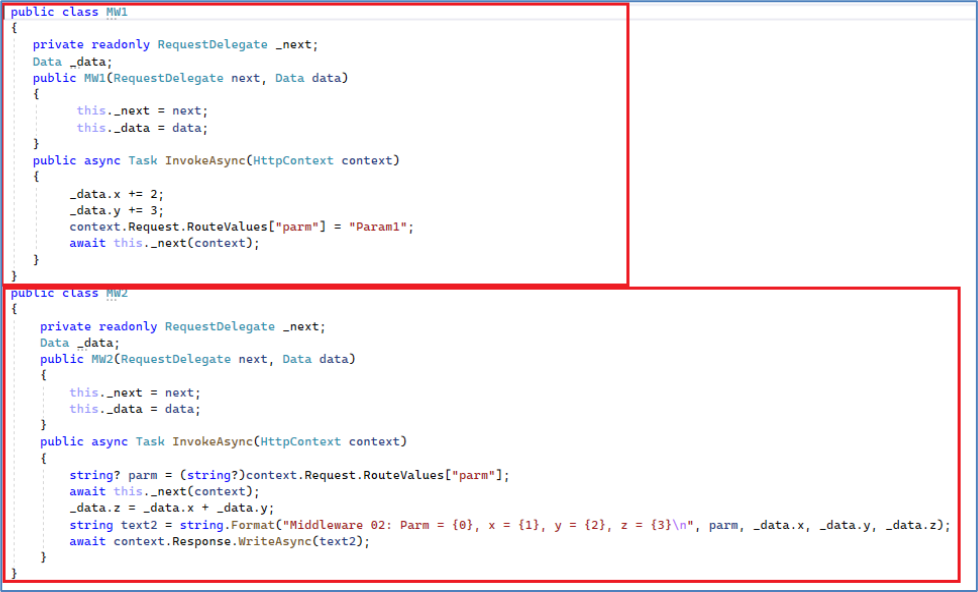
Создание вариант 1:



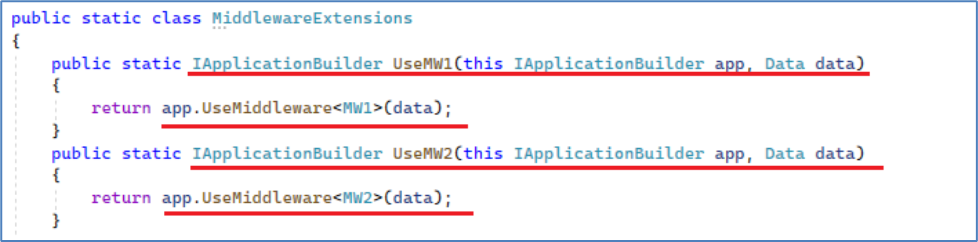
2. Через **UseMiddleware<>**.

Класс middleware должен иметь конструктор, который принимает параметр типа **RequestDelegate**. Через этот параметр можно получить ссылку на тот делегат запроса, который стоит следующим в конвейере обработки запроса.

Также в классе должен быть определен метод, который должен называться либо **Invoke**, либо **InvokeAsync**. Причем этот метод должен возвращать объект **Task** и принимать в качестве параметра контекст запроса - объект **HttpContext**. Данный метод собственно и будет обрабатывать запрос.



3. **Расширение** класса **IApplicationBuilder**.





В ASP.NET Core порядок выполнения middleware-компонентов в конвейере обработки запросов (**pipeline**) зависит от порядка их объявления в методе **Configure** класса **Startup**. Этот порядок действительно важен и определяет, как запросы будут обрабатываться при прохождении через цепочку middleware.