Вопросы к экзамену по дисциплине

«Программирование интернет-серверов»

ПОИТ-3, ЛЕТО 2024

Ну тут часть нагенерина ботом, но про платформы .net +- с конспекта лекции

Оглавление

[1. Сеть Интернет: сетевые протоколы, службы, организационная структура, документация, форматы данных. 1](#_Toc168439850)

[3. НTML: назначение, применение, структура страницы, DOM, API. 4](#_Toc168439851)

[4. CSS: назначение, применение, методы размещения на странице, селекторы. 8](#_Toc168439852)

[5. Web-приложение: архитектура, основные свойства. 9](#_Toc168439853)

[6. Web-сервер: основные серверные объекты. 12](#_Toc168439854)

[7. Web-браузер: структура, основные объекты, DOM, BOM. 13](#_Toc168439855)

[8. Методология AJAX 15](#_Toc168439856)

[9. Платформа .NET CORE: свойства, назначение, применение. 21](#_Toc168439857)

[10. Платформа ASP.NET CORE: свойства, назначение, применение. 22](#_Toc168439858)

[11. Платформа ASP.NET CORE: общая структура приложения. 24](#_Toc168439859)

[OWIN 24](#_Toc168439860)

[12. Платформа ASP.NET CORE: простейший обработчик запросов. 26](#_Toc168439861)

[13. Платформа ASP.NET CORE: WebSocket - сервер. 26](#_Toc168439862)

[14. Платформа ASP.NET CORE: статические данные и стартовые страницы. 29](#_Toc168439863)

[15. Платформа ASP.NET CORE: приложение для скачивания статических файлов. 36](#_Toc168439864)

[20. Платформа ASP.NET CORE: MVC-приложение, модель источника данных. Паттерн ORM. 37](#_Toc168439865)

[22. Платформа ASP.NET CORE: простейшее Web API-приложение. 37](#_Toc168439866)

[23. Платформа ASP.NET CORE: реализация REST-приложения. 38](#_Toc168439867)

[25. Платформа ASP.NET CORE: middleware, разработка объектов middleware. 42](#_Toc168439868)

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

**Интернет:** Всемирная компьютерная сеть, построенная на основе стека протоколов TCP/IP

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 стандартные 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, Швейцария), занимающаяся развитием сети Internet. ISOC владелец 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)

Сервер + протокол. Протокол, описывающий доступ к этому серверу. Специфический порт: из диапазона хорошо известных портов (0 - 1024). Хост интернета.

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 видео данных для передачи и воспроизведения видео контента.
8. **CSV (Comma-Separated Values)**:
   * Простой текстовый формат для представления табличных данных.
9. Протокол HTTP: версии, основные свойства.
10. 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.

**модель DOM (W3C):** представление HTML-документа web-браузером, интерфейс JavaScript для доступа к содержимому HTML-документа

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

*По сути, уровень DOM просто представляет собой что-то вроде релизной версии. Это может быть связано с его эволюцией: последующие уровни (спецификации) строятся на предыдущих, добавляются какие-то новые модули, методы и поведения... до тех пор, пока он не остановился на DOM 4-го уровня. Стандарт больше не описывается как набор слоев, а как моментальный снимок живой спецификации.*

содержит описание JavaScript (ECMAScript)

**DOM** – это объектное представление исходного HTML-документа, попытка преобразовать его структуру и содержимое в объектную модель, с которой смогли бы работать различные программы.

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;

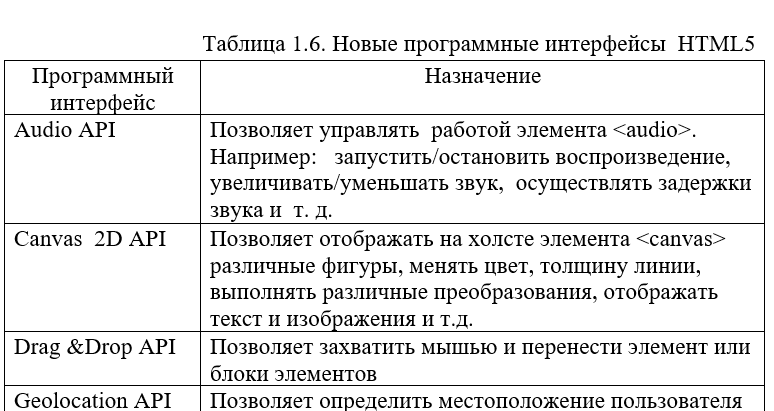
• включает скрытые элементы (display: none).

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

CSS (англ. Cascading Style Sheets — каскадные таблицы стилей) — технология описания внешнего вида документа, оформленного языком разметки.

Каскадные таблицы стилей (Cascading Style Sheets, CSS) — это стандарт, определяющий представление данных в браузере. Если HTML предоставляет информацию о структуре документа, то таблицы стилей сообщают как он должен выглядеть.

Селектор – это часть CSS-правила, которая сообщает браузеру, к какому элементу (или элементам) веб-страницы будет применён стиль.

# - id, . - class, - tags, [] - atributs, \* - all

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>**

Селекторы CSS

Селекторы используются для выбора элементов HTML, к которым будут применены стили. Существует множество различных типов селекторов.

1. **Селекторы по тегу**:
2. **Селекторы по классу**:
3. **Селекторы по идентификатору (ID)**:
4. **Комбинированные селекторы**:
5. Web-приложение: архитектура, основные свойства.

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

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 — программа на сервере, которая прослушивает входящие сообщения.

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

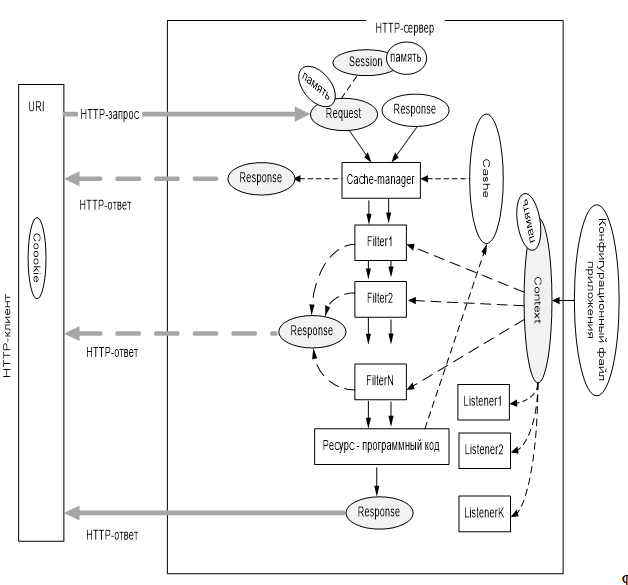
Слушатели событий отслеживают действия пользователя или изменения состояния приложения:

* 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.



1. Web-сервер: основные серверные объекты.

Контекст, фильтр, listener, кеш, request, response, session.

Порядок создания объектов сервера:

Контекст => listener => filter. Затем уже сессия и д.р.

В лк ОПИП 3

1. 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.

Части:

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

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

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

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

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

**Networking** – модуль взаимодействия с сетью



**Browser API:** Geolocation/HTML5 – программный интерфейс позволяющий определить географические координаты месторасположения пользователя

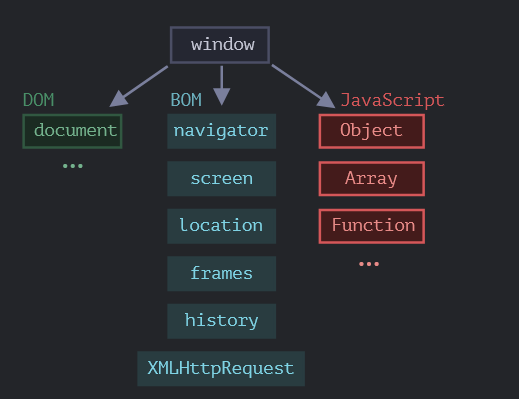
**Browser API:** Web Storage/HTML5 – программный интерфейс позволяющий сохранять данные домена (URL) в хранилище данных (Data Persistence) браузера (Local Storage, Session Storage).

**Browser API:** Drag&Drop/HTML5 - программный интерфейс позволяющий пользователю захватить мышью элемент и перенсти).

\*\***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)

Объектная модель браузера (Browser Object Model, BOM) – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа.



Как мы видим, имеется корневой объект window, который выступает в 2 ролях:

1. Во-первых, это глобальный объект для JavaScript-кода, об этом более подробно говорится в главе [Глобальный объект](https://learn.javascript.ru/global-object).
2. Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.
3. Объект [navigator](https://developer.mozilla.org/en/DOM/window.navigator) содержит общую информацию о браузере и операционной системы. Особенно примечательны два свойства:navigator.userAgent– содержит информацию о браузере иnavigator.platform содержит информацию о платформе, позволяет различать Windows/Linux/Mac и т.п.
4. Объект [location](https://developer.mozilla.org/en-US/docs/Web/API/Window.location) содержит информацию о текущем URL страницы и позволяет перенаправить посетителя на новый URL.
5. Функции alert/confirm/promptтоже входят в BOM.
6. Методология AJAX

**!!!ИЗ ЛЕКЦИИ: AJAX – это методология (принципы и правила написания кода) разработки, которая основывается на возможности объекта XMLHttpRequest выдавать асинхронные запросы**

**AJAX – асинхронный JavaScript and XML – методология (подход) построения динамических приложений, которая основана на возможности XMLHttpRequest выдавать асинхронные запросы. Следствие: не осуществляется полная перезагрузка html-страниц. AJAX: XMLHTTPRequest, DOM, формат: XML и JSON.  XMLHttpRequest не относится к BOM, DOM, но в рамках браузера.**

Метод open (метод, URI, bool async | sync)

Send метод отправляет заявку.

Получить ответ: в свойство onChange записать ссылку на callback функцию, которая будет вызываться браузером в зависимости от 4 состояний. На 4 стадии можно обрабатывать сам ответ. Можно использовать для progress-bar.

Из книжки смелова

Для построения эффективных web-приложений до появления HTML5 применялась технология AJAX (Asynchronous JavaScript and XML), которая основывалась на возможности доступного из JavaScript объекта XMLHTTPRequest асинхронно отправлять http-запросы серверу и обрабатывать его ответы. В качестве формата передачи данных в AJAX, как правило, используется XML и JSON. Основное достоинство AJAX-приложения заключается в его динамичности: применение AJAX позволяет сократить сетевой трафик и избежать лишних перезагрузок html-страниц.

Основной недостаток AJAX заключается в том, что он основывается на протоколе HTTP, который требует работы в режиме «запрос-ответ». Другими словами, получить порцию данных от сервера можно только сделав к нему запрос. В приложениях, основная задача которых реагировать на события, происходящие на сервере, клиентской части приложения приходится периодически выполнять запросы к серверу, чтобы «увидеть» эти события. Для того, чтобы «не пропустить» серверное событие и своевременно на него отреагировать, приходится увеличивать частоту запросов. Это приводит к загрузке сетевого трафика и нерациональному расходу ресурса сервера, который вынужден постоянно отвечать на присланные клиентом запросы.

**AJAX**: Asynchronous JavaScript and XML – асинхронный JavaScript and XML – методология (подход) построения динамических приложений, при которых не осуществляется полная перезагрузка html-страниц. AJAX: XMLHTTPRequest, DOM, формат: XML и JSON.

**В основе методологии Ajax лежат следующие технологии:** язык HTML, язык JavaScript, язык XML, модель DOM, протокол HTTP, протокол JSON, объект XMLHttpRequest.

СТРУКУТУРА:

**HTML–** гипертекстовый язык разметки. Интерпретируется браузером. В Ajax динамически изменяется содержимое html-документа.

**JavaScript**– скриптовый язык, предназначенный для создания сценариев поведения браузера. Интерпретируется браузером. В Ajax html-документ динамически изменяется на стороне клиента с помощью сценариев написанных на языке JavaScript.

**DOM –** объектная модель, позволяющая сценариям JavaScript получить доступ (читать и изменять содержимое) к элементам html-документа (к атрибутам и содержимому тегов). В Ajax ответ сервера ―встраивается‖ с помощью JavaScript-сценария в загруженную ранее браузером страницу. При этом доступ к элементам html-документа осуществляется в соответствии с моделью DOM.

**HTTP –**сетевой протокол передачи гипер текста. Используется для обмена данными между двумя приложениями (клиентом и сервером). В Ajax обмен данными между JavaScript-сценарием на клиенте и серверным приложением (например, сервлетом) осуществляется по правилам HTTP.

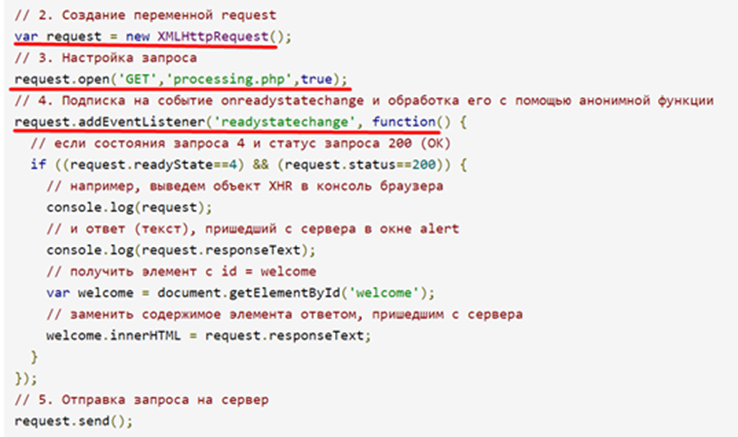
**XML –** расширяемый язык разметки данных. Предназначен для структуризации данных с целью хранения или/и передачи. В Ajax язык XML является одним из форматов, который используется для структуризации данных пересылаемых между JavaScript-сценарием и серверным приложением.

**JSON (JavaScript Object Notation)** - текстовый формат обмена данными, применяемый обычно в сценариях JavaScript**.** В Ajax формат JSON является одним из форматов, который используется для структуризации данных пересылаемых между JavaScript сценарием и серверным приложением. Формат JSON основывается на функции **eval()** языка JavaScript.

**XMLHttpRequest**– **браузерный объект,** специальный API (предопределенный объект), используемый в языке JavaScript для обмена данными между сценарием JavaScript и серверным приложением по протоколу HTTP. В Ajax методы объекта XMLHttpRequestиспользуется для отправки и получения данных между JavaScript-сценарием и серверным приложением. Данные могут получены в виде XML-документа и виде обыкновенного текста (в частном случае могут быть представлены в формате JSON).

*Структура AJAX*

Для каждой части страницы должен создавать свой объект xmlHttpRequest (он живёт в браузере).



А как теперь получить ответ, когда мы отравили запрос? Любая асинхронная обработка всегда устроена так, что обработка ответов осуществляется с помощью функции обратного вызова (функция callback).

Прежде чем, мы с помощью объекта XMLHttpRequest выполним метод Send(), который отправляет асинхронный запрос к серверу, мы этому объекту скажем: а вот когда придет ответ, ты вызови эту функцию, которую мы укажем Т.е. мы ему заранее подсказываем: вызови такую функцию, когда придет ответ и мы с вами продолжаем работать: отправили запрос, продолжаем дальше работать и когда приходит ответ, он вызывает эту функцию и она выполняется, в ней обрабатывается ответ. Причем, эта функция вызывается 5 раз в разные этапы выполнения запроса и только на 4 этапе (нумерация с 0, по счету 5ый раз получается) мы получаем статус ответа, где мы можем выяснить, нормально или ненормально закончился ответ. Это то, что лежит в глубине ajax.

**Методы объекта XmlHttpRequest**

* getAllResponseHeaders() — получить все заголовки ответа от сервера.
* getResponseHeader(«имя\_заголовка») — получить указаный заголовок.
* setRequestHeader(«имя\_заголовка»,«значение») — установить значения заголовка запроса.
* open(«тип\_запроса»,«URL»,«асинхронный»,«имя\_пользователя»,«пароль») — инициализация запроса к серверу, указание метода запроса. Тип запроса и URL — обязательные параметры. Третий аргумент — булево значение. Обычно всегда указывается true или не указывается вообще (по умолчанию — true). Четвертый и пятый аргументы используются для аутентификации (это очень небезопасно, хранить данные об аутентификации в скрипте, так как скрипт может посмотреть любой пользователь). (вызывается первым после создания XMLHttpRequest, вызов open – не открывает соединение. Он настраивает запрос.)
* send(«содержимое») — послать HTTP запрос на сервер и получить ответ. (открывает соединение и отправляет запрос на сервер.)
* abort() — отмена текущего запроса к серверу.

**Свойства объекта XmlHttpRequest**

* Status – http-код ответа.
* StatusText – текстовое описание http-кода.
* ResponseText – текст ответа сервера.
* onreadystatechange — одно из самых главных свойств объекта XMLHttpRequest. С помощью этого свойства задаётся обработчик, который вызывается всякий раз при смене статуса объекта.
* readyState — число, обозначающее статус объекта.

Ответ с сервера приходит в JSON.

5 состояния запроса:

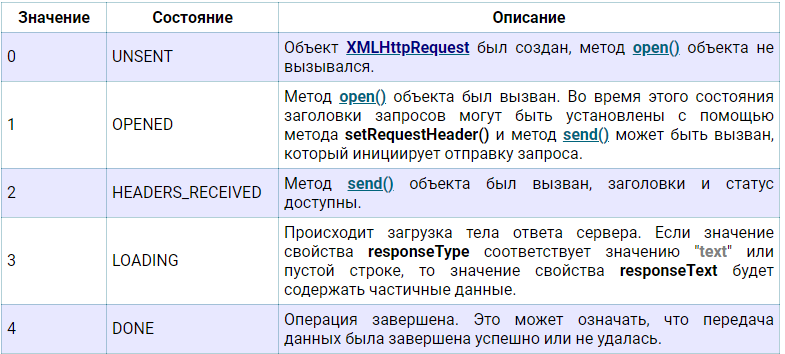
* 0 – начальное состояние
* 1 – вызван open
* 2 – получены заголовки
* 3 – загружается тело
* 4 – запрос завершён

Сейчас существует множество js-библиотек: JQuery, Angular, React, которые скрывают от нас этот механизм, и мы с ним не работаем.

**AJAX в ASP.net представлен в виде:**

* UpdatePanel
* Trigger
* Timer
* UpdateProgress

Когда мы пишем cкрипт, браузер сам подставляет объявление всех его объектов, например: window, [XMLHttpRequest](https://ru.wikipedia.org/wiki/XMLHttpRequest), document.



**AJAX** — не самостоятельная технология, а концепция использования нескольких смежных технологий. AJAX базируется на двух основных принципах:

1. использование технологии динамического обращения к [серверу](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5)) «на лету», без перезагрузки всей страницы полностью, например с использованием [XMLHttpRequest](https://ru.wikipedia.org/wiki/XMLHttpRequest) (основной объект);

* через динамическое создание дочерних фреймов[[1]](https://ru.wikipedia.org/wiki/AJAX#cite_note-1);
* через динамическое создание тега <script>[[2]](https://ru.wikipedia.org/wiki/AJAX#cite_note-JsHttpRequest-2).
* через динамическое создание тега <img>, как это реализовано в Google Analytics.

1. использование [DHTML](https://ru.wikipedia.org/wiki/DHTML) для динамического изменения содержания страницы;

ПРИМЕНЕНИЕ:

**Экономия трафика:** загружает определенную часть страницы или просто получает/передает данные в формате JSON или XML, и затем изменяет содержимое страницы с помощью JS.

**Уменьшение нагрузки на сервер:** говорится, что для загрузки страницы требуется обращение к разным файлам, время на обработку скриптов (иногда запросы к БД) и все это можно заменить загрузкой и генерацией лишь части страницы.

**Ускорение реакции интерфейса:** Поскольку загрузка изменившейся части значительно быстрее, то пользователь видит результат своих действий быстрее и без мерцания страницы (возникающего при полной перезагрузке).

**Возможности для интерактивной обработки:**

Например, при вводе поискового запроса в [Google](https://ru.wikipedia.org/wiki/Google_(%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0)) выводится подсказка с возможными вариантами запроса. На многих сайтах при регистрации пользователь вводит имя, и сразу же видит, доступно это имя или нет. AJAX удобен для программирования [чатов](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%82), [административных панелей](https://ru.wikipedia.org/w/index.php?title=%D0%90%D0%B4%D0%BC%D0%B8%D0%BD%D0%B8%D1%81%D1%82%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BD%D0%B5%D0%BB%D1%8C&action=edit&redlink=1) и других инструментов, которые выводят меняющиеся со временем данные.

[**Мультимедиа**](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BC%D0%B5%D0%B4%D0%B8%D0%B0) **не останавливается :**

Страница не перезагружается, плеер продолжает работать. Потому AJAX ценен на аудио- и видеохос

Асинхронная функция – это функция, после вызова которой JavaScript приложение продолжает работать, потому что функция сразу выполняет возврат. Результат работы асинхронной функции становится известным позже, и для того, чтобы оповестить наше приложении о полученных значениях, асинхронная функция вызывает другую функцию (callback), которую мы передаем в аргументах при запуске.

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

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

где url - URL для отправки запроса,

options - дополниельные параметры: метод, заголовки и так далее.

**Функция обратного вызова: простыми словами:** коллбэк — это функция, которая должна быть выполнена после того, как другая функция завершила выполнение (отсюда и название: callback – функция обратного вызова).

**Чуть сложнее:** В JavaScript функции — это объекты. Поэтому функции могут принимать другие функции в качестве аргументов, а также функции могут возвращать функции в качестве результата. Функции, которые это умеют, называются **функциями высшего порядка**. А любая функция, которая передается как аргумент, называется **callback-функцией**. Чтобы лучше разобраться, давайте посмотрим на примерах, как это выглядит.

loadScript(‘/my/script.js’, function(){

//эта функция вызовется после того, как загрузится скрипт

newFunction(); //теперь всё работает

…

});

В С++ передать указатель на функцию.В С# дилегат.

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

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

Свойства: кроссплатформенность и открытый код, NuGet – программный менеджер, модульность.

**Платформа** – это набор библиотек и инструментов для разработки

.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.

подход pay-for-play, загружается только та функциональность, которая необходима (сборки).

<https://habr.com/ru/companies/nix/articles/327686/>

В NET Core все компоненты программы, которые мы определяем на этапе компиляции, являются зависимостями приложения (включая Core CLR, JIT), которые инфраструктура .NET Core рассматривает как пакеты. Такой пакет называется asset, при этом он может быть как NuGet-пакетом, так и обычным файлом.

Благодаря такой модели .NET Core-приложение состоит из пугающе огромного количества мелких модулей, но это сделано, чтобы уменьшить объем ненужных зависимостей.

Этот подход называется «pay-for-play»; другими словами, приложения загружают только ту функциональность, которая им необходима, но каждая такая функциональность содержится в отдельной сборке.

Portable (FDD)-приложение похоже на традиционное .NET Framework-приложение. В данном случае определенная версия .NET Core-фреймворка (также используются термины shared framework, .NET Core Runtime, redist) должна находиться на целевом компьютере, и при запуске хост процесс загрузит Core CLR, Core FX из папки фреймворка.

В Standalone (SCD)-приложении все компоненты для выполнения (CoreCLR, CoreFX), а также сторонние библиотеки, то есть абсолютно все зависимости, поставляются вместе с самим приложением (чаще всего в одной папке).

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

dodnet.exe точка входа для любого asp приложения

Фреймворк находится в с/програм файлс/дотнет/shared

**NET Core Runtime: dotnet.exe –** утилита для запуска .NET Core-приложения(muxer - мультиплексор), точка входа для любого .NET Core – приложения, host-процесс для выполнения любого .NET Core. dotnet – переименованный corehost.exe

Там на фотке ещё какая-то муть была, но тут без нее

.NET Core — это современная, кроссплатформенная версия .NET.

.NET — это общее название для всей экосистемы разработки программного обеспечения, включающей различные реализации и фреймворки, такие как .NET Framework, .NET Core и др.

ASP.NET — это фреймворк для создания веб-приложений и сервисов, работающий на .NET Framework

ASP.NET Core — это кроссплатформенная версия ASP.NET, работающая на .NET Core.

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

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

Свойства: кроссплатформенность и открытый код

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

Платформа – это набор библиотек и инструментов для разработки

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

.NET Core = CoreCLR + библиотеки CoreFX.

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.

!!!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 открытым кодом

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

- приложения могут работать под IIS (Windows) или под web-сервером Kastrel (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.

- все приложения имеют одну и ту же структуру

- 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 необъодимых для работы приложения

dodnet.exe точка входа для любого asp приложения

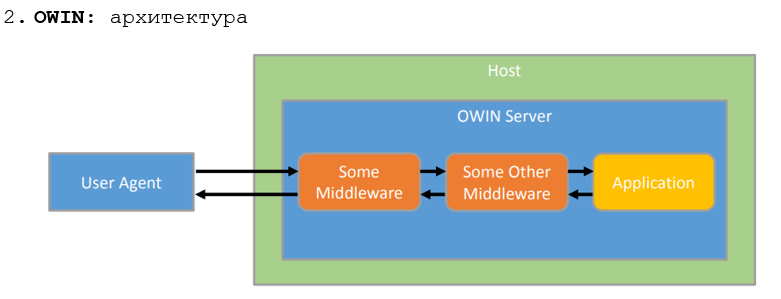
Фреймворк находится в с/програм файлс/дотнет/shared

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

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.

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 назанию 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>

1. Платформа ASP.NET CORE: простейший обработчик запросов.
2. Платформа ASP.NET CORE: WebSocket - сервер.

Протокол WebSocket: назначение, применение.

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 наверно 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.

Протокол определяет управляющие кадры, такие как "ping" и "pong", для поддержания соединения и проверки его состояния.

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.

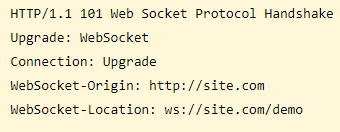
Изначально синхронный HTTP протокол, построенный по модели «запрос — ответ», становится **полностью асинхронным и симметричным**. Теперь уже нет клиента и сервера с фиксированными ролями, а есть два равноправных участника обмена данными. Каждый работает сам по себе, и когда надо отправляет данные другому. 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.

ПО КОДУ

// Включаем поддержку WebSocket

app.UseWebSockets();

// Добавляем миддлвар для обработки WebSocket-запросов

if (context.WebSockets.IsWebSocketRequest)

{

WebSocket ws = await context.WebSockets.AcceptWebSocketAsync();

await EchoWebSocket(context, webSocket);

}

Для отправки сообщения

await ws.SendAsync(bytes,

на html странице

function exe\_start() {

ws = new WebSocket("ws://localhost:5115/ws");

ws.onopen = function () {

messagesElement.innerHTML += `<h1>Open</h1>`;

};

ws.onmessage = function (event) {

messagesElement.innerHTML += `<p>${event.data}</p>`;

}

}

function exe\_stop() {

ws.close(3001, ' stopapp');

ws = null;

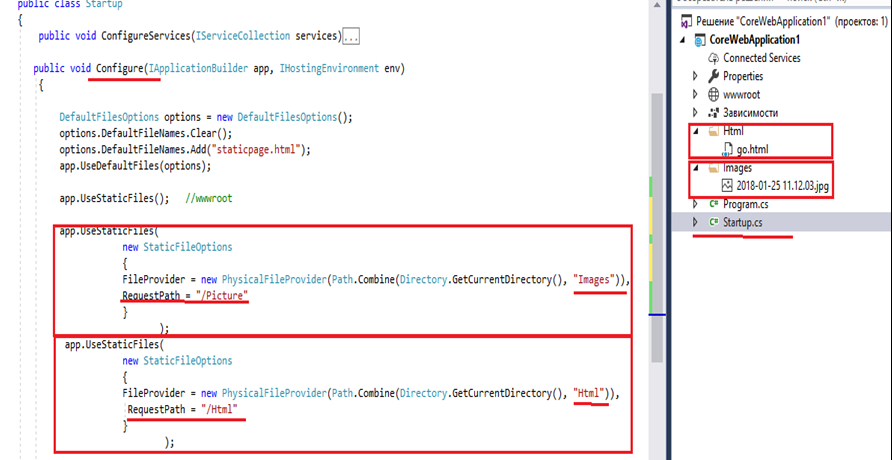
}

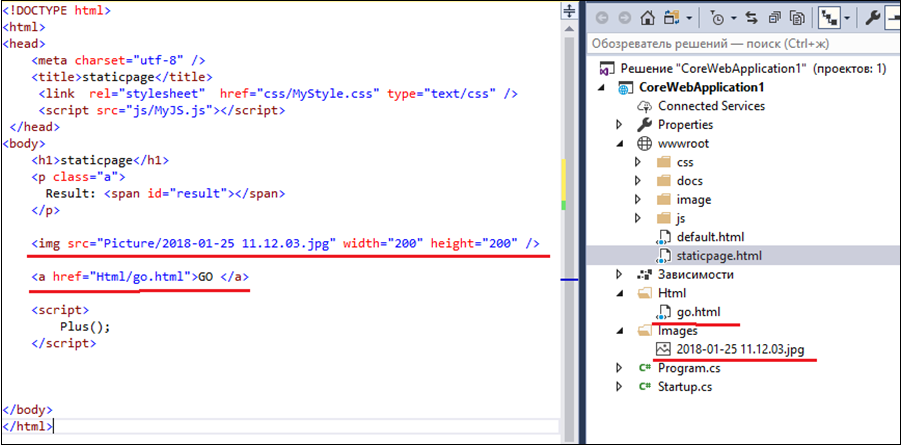
Объект WebSocket имеет несколько состояний (**readyState**), которые отражают текущее состояние соединения между клиентом и сервером. Эти состояния определяются согласно спецификации WebSocket API (RFC 6455) и доступны через свойство **readyState** объекта WebSocket в JavaScript. Вот список возможных 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.
5. 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.

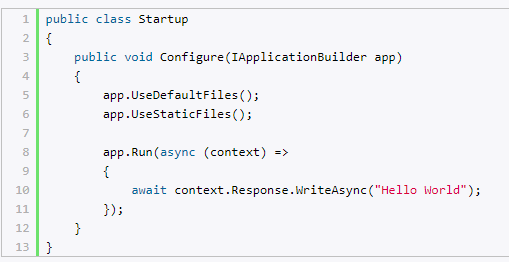
**Почему важен порядок?**

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:

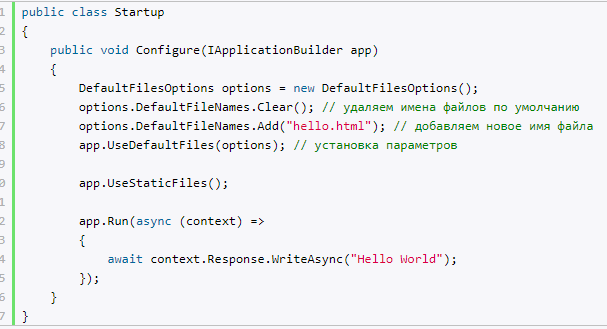


В этом случае при отправке запроса к корню веб-приложения типа http://localhost:xxxx/ приложение будет искать в папке wwwroot следующие файлы:

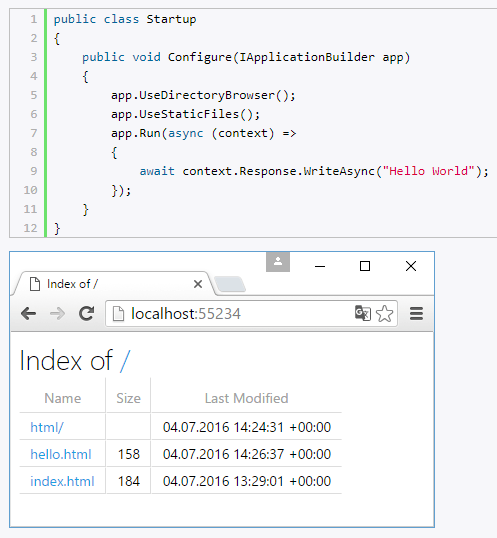
1. default.htm
2. default.html
3. index.htm
4. index.html

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(){

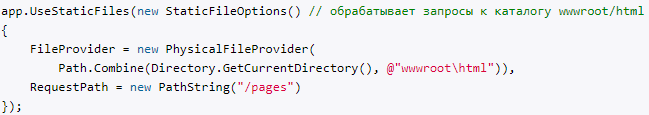
FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), @”wwwroot\html”)),

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:**

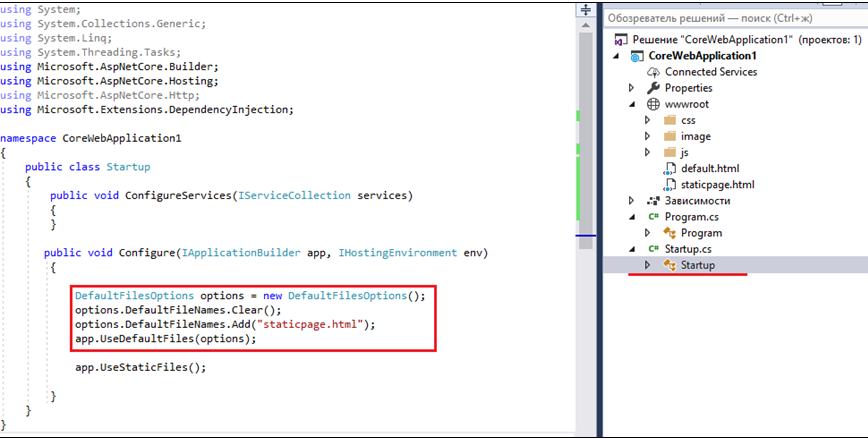


**Стартовые страницы:**

Г

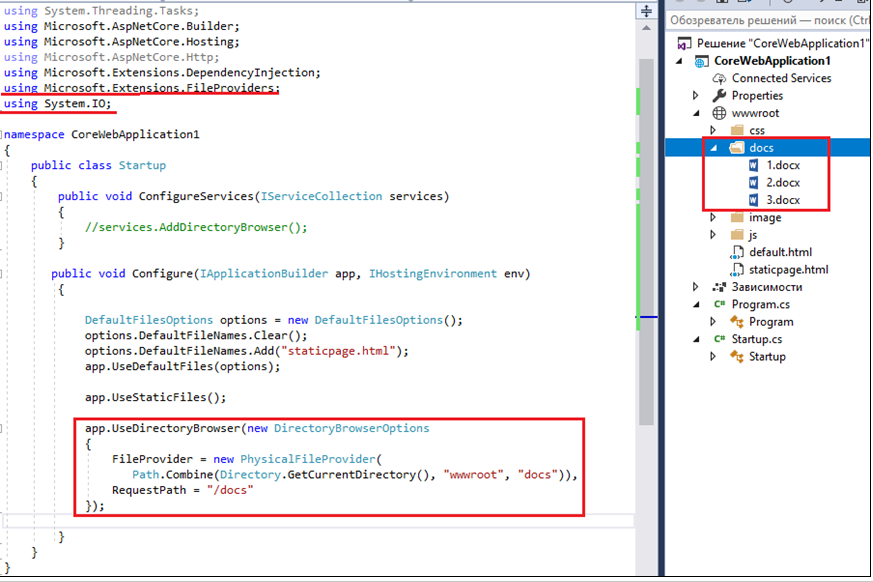
В данном случае у нас будут использоваться страницы default.html, default.htm, index.html, index.htm в качестве страниц по умолчанию.

Для пользовательских страниц по умолчанию используется следующий подход:

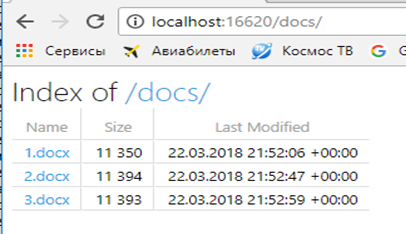


Будет использована страница staticpage.html, как страница по умолчанию.

**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:

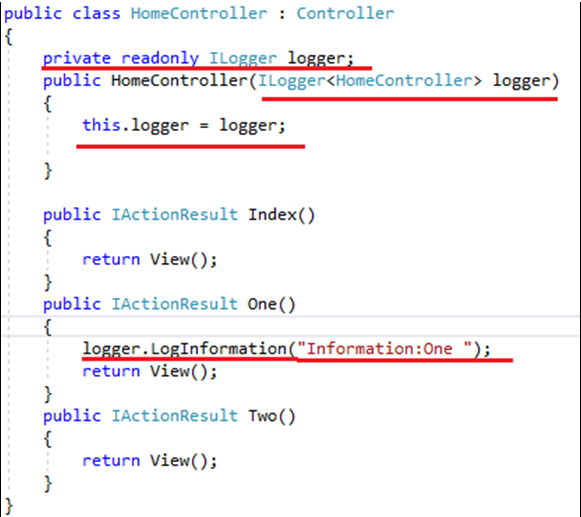


И может скачать файлы из данного каталога.

**Вывод в журнал:**

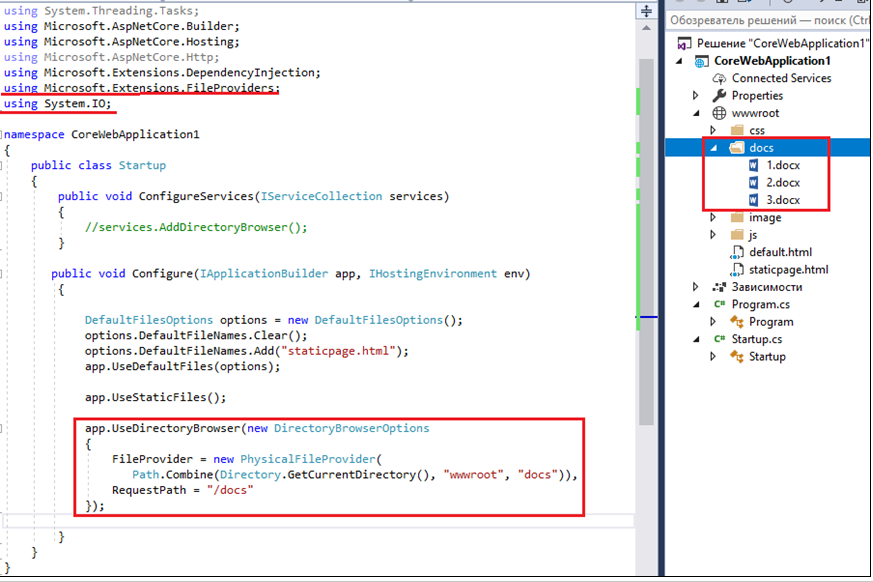


Включаем поддержку логгирования при помощи метода AddConsole(), а затем производим логгирование в самом контроллере.

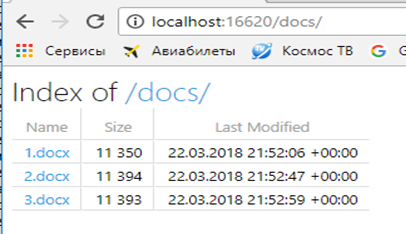


1. Платформа ASP.NET CORE: приложение для скачивания статических файлов.

**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.

1. Платформа ASP.NET CORE: структура MVC-приложения.
2. Платформа ASP.NET CORE: MVC-приложение, маршрутизатор, атрибуты Route, констрейны маршрутов.
3. 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.
4. Платформа ASP.NET CORE: MVC-приложение, представление, жизненный цикл представления, хэлперы, частичные представления, директивы @ model, @inherits, @function, @inject, @using, @addTagHelper, @removeTagHelper, @RenderSection, @RenderBody.
5. Платформа ASP.NET CORE: MVC-приложение, модель источника данных. Паттерн ORM.
6. **ORM:** Object-Relation Mapping, паттерн проектирования, применяется для работы с данными, концепция БД/ООП, ООП-интерфейс с БД, стандарт IT-индустрии.
7. Платформа ASP.NET CORE: внедрение зависимостей.
8. Платформа ASP.NET CORE: простейшее Web API-приложение.

Простейшее Web API-приложение представляет собой веб-приложение, которое предоставляет доступ к данным и функциональности через HTTP протокол с использованием архитектурного стиля REST

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

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

1. **RESTful**: описание web-службы, поддерживающей REST-интерфейс в полном объеме (со строгим соблюдением правил).
2. **REST**: нет официального стандарта, но REST использует стандарты HTTP, URL/URI, XML, JSON.
3. **REST:** два основныхтипа ресурса – коллекции и элемент коллекции: /api/users, /api/users/288.

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, 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
7. **HATEOAS: Hypermedia As The Engine Of Application State** – гипермедиа в качестве управления состоянием.
8. **Гипермедиа**: технология обработки, структурирования информации и произвольного доступа к ее элементам с помощью гиперсвязей (Тед Нильсон, 1965), WWW – реализация гипермедиа.

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

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

<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.

Из чата

**Основные принципы REST API:**

1. **Ресурсы (Resources)**: Ресурсы представляют собой конкретные сущности или объекты, к которым клиенты могут получать доступ через API. Каждый ресурс идентифицируется уникальным URI (Uniform Resource Identifier).
2. **Методы HTTP (HTTP Methods)**: REST API использует стандартные методы HTTP для выполнения операций над ресурсами:
   * **GET**: Получение данных.
   * **POST**: Создание нового ресурса.
   * **PUT**: Обновление существующего ресурса.
   * **DELETE**: Удаление ресурса.

Кроме основных методов, могут использоваться и другие методы, такие как **PATCH** для частичного обновления ресурсов и **OPTIONS** для получения информации о доступных методах для ресурса.

1. **Представление (Representation)**: Клиенты получают представление ресурса в определенном формате данных, например, JSON, XML или других форматах. Формат данных определяется через заголовки HTTP запроса (например, **Accept**).
2. **Состояние (Stateless)**: REST API не хранит состояние между запросами клиентов. Каждый запрос от клиента содержит всю необходимую информацию для обработки запроса. Это повышает масштабируемость и надежность системы, а также упрощает разработку и обслуживание.
3. **Связи (Links)**: Гипермедиа-связи или ссылки предоставляют клиентам информацию о том, какие действия можно выполнить с ресурсом и какие другие ресурсы доступны. Это делает API более самоописывающим и упрощает взаимодействие клиентов с API без необходимости заранее знать его структуру.

"RESTful API" (RESTful Web Services) является подмножеством "REST API" и подразумевает, что API полностью соответствует или следует принципам REST. В частности, для того чтобы API могло быть названо RESTful,

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

ASP.NET Core поддерживает две основные модели разработки веб-приложений: Web API и MVC (Model-View-Controller). Давайте рассмотрим каждую из них подробнее.

**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**), 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.

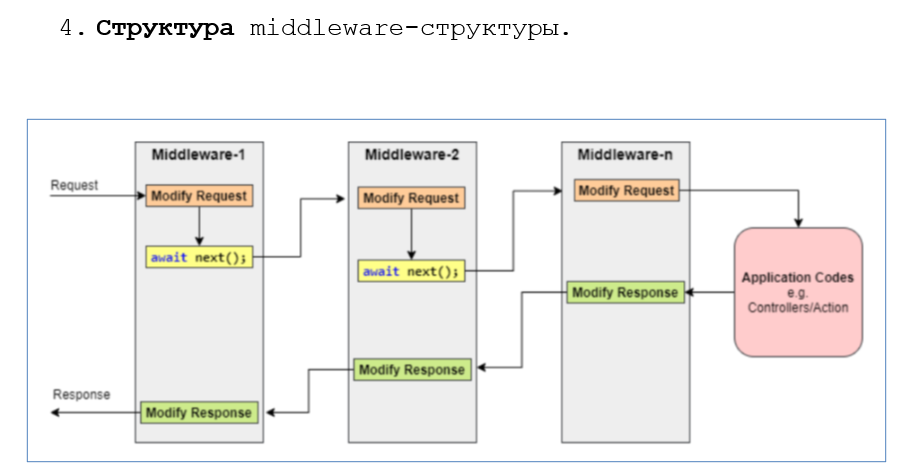
**Особенности MVC:**

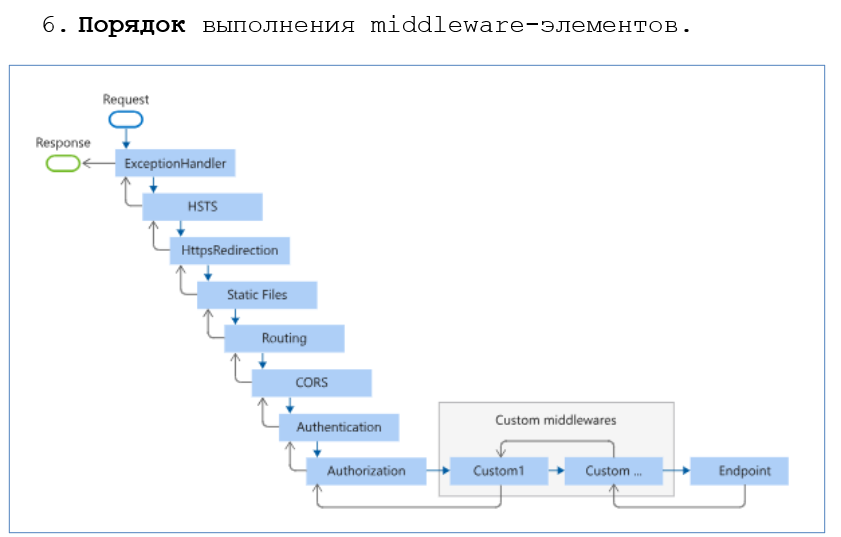
1. **Разделение на слои**: Модель (**Model**) представляет данные и бизнес-логику, Представление (**View**) отвечает за отображение данных пользователю, а Контроллер (**Controller**) обрабатывает пользовательские запросы и управляет взаимодействием между Моделью и Представлением.
2. **Рендеринг представлений**: 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.

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





**UseEdpoints** конечная точка конвейера.

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

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.

**Разработка Middleware**

Для разработки middleware в ASP.NET Core требуется создать класс, который реализует интерфейс IMiddleware или метод расширения RequestDelegate

1/ **Реализация интерфейса IMiddleware**

using Microsoft.AspNetCore.Http;

using System.Threading.Tasks;

public class CustomMiddleware : **IMiddleware**

{

public async Task InvokeAsync(**HttpContext context, RequestDelegate next)**

{

// Код обработки запроса

await context.Response.WriteAsync("Before handling the request.\n");

// Вызываем следующее middleware в конвейере

await next(context);

// Код после обработки запроса

await context.Response.WriteAsync("After handling the request.\n");

}

}

* **IMiddleware** определяет метод **InvokeAsync**, который обрабатывает запрос и передает управление следующему middleware через делегат **RequestDelegate**.

2/ **Метод расширения RequestDelegate**

using Microsoft.AspNetCore.Http;

using System.Threading.Tasks;

public static class CustomMiddlewareExtensions

{

public static async Task CustomMiddleware(**this HttpContext context, RequestDelegate next**)

{

// Код обработки запроса

await context.Response.WriteAsync("Before handling the request.\n");

// Вызываем следующее middleware в конвейере

await next(context);

// Код после обработки запроса

await context.Response.WriteAsync("After handling the request.\n");

}

}

* В этом примере middleware реализуется как метод расширения **CustomMiddleware** для **HttpContext**, который принимает делегат **RequestDelegate**.

**Регистрация Middleware в конвейере**

app.UseMiddleware<CustomMiddleware>(); // Для реализации через IMiddleware

// или

app.UseMiddleware(typeof(CustomMiddlewareExtensions)); // Для реализации через метод расширения