**1ый вопрос**

1. Протокол HTTP: клиент-сервер; типы сообщений, структура запроса, структура ответа, статус (серии значений), методы, заголовки, параметры. Понятие stateless-протокола.

**HTTP** – прот.перед.гипертекста – прот.прикл.ур-ня перед.д-х (изнач.– в виде гипертекст.док-ов).

Основа – технол.«К-С», т.е. предполаг.сущ-ние К, кот. инициир. соед. и посыл.запрос, и С, кот. ожид. соед. для получ.запроса, вып.необх.д-вия и возвращ. обратно сообщ.с рез-том.

**Спецификация HTTP**

* Название: HyperText Transfer Protocol
* Уровень (по модели OSI): прикладной
* Семейство: TCP/IP
* Порт/ID: 80/TCP
* Назначение протокола: доступ к гипертексту, ныне стал универсальным
* Спецификация: RFC 1945, RFC 2616

|  |  |
| --- | --- |
| Достоинства | Недостатки |
| \*простота – леко созд.К прилож  \*расширяемость  \*распространенность | \*большой р-р сообщ  \*нет поддержки распределнности |

**!Осн. свойства HTTP:**

- версии http/1.1 (текстовый), http/2 (бинарный – сейчас к нему переход)

- 2 вида сообщений: req, res => асинхр (неравноцен К и С, разный формат сообщ)

- 2 типа абонентов: К –req–> <–res– С

- 1 req : 1 res и наоборот (иначе ошибка)

- TCP-порты: **80** (для серверов, не поддерж. шифр) и **443** (поддерж шифр; HTTPS)

- для адресации исп. URI (часть без http://имя\_пк)

- поддерж. W3C, описан в неск RFC (2616)

|  |  |
| --- | --- |
| **!Структура Request:**  - метод (метка get/post)  - URI  - версия протокола (http/1.1)  - заголовки (имя+знач)  - параметры (имя+знач)  - расширение (тело) | **!Структура Response:**  - версия протокола  - код сост (1хх, 2хх, 3хх, 4хх, 5хх)  - пояснение к коду сост  - заголовки (имя+знач)  - расширение |

Заголовки и тело м.отсутств., но старт.строка – обяз, т.к.указ тип запроса/отв

Старт.строка запроса: метод, uri, версия

Старт.строка ответа: версия, код состояния, пояснение

**Request-методы:**

\*OPTIONS – исп.для опред-ния возм-стей веб-С или пар-ров соед для рес-са.

\*GET – узнать содержимое ресурса; м. начать какой-то процесс и передать инфу

\*HEAD – анал.get. Отличие – отсутствует тело в ответе С.

\*POST – для передачи пользовательских д-х задан.ресурсу.

\*PUT – загрузки содержимого на указ.URI (не было – созд; есть – измен.)

\*DELETE – удаляет указ.рес-с

\*TRACE – вовзращ.запрос так, что К м.увидеть какую инфу промеж.С доб/измен в запросе

connect, extension-method

|  |  |
| --- | --- |
| **Заголовки:**  \* general (д.ключ.в люб.сообщ К/С)  \* request (т. в запросах)  \* response (исп т. в ответах)  \* entity (опис содержимое тела – ContentType) | **Код сост. Response:**  \* 1хх: информ. сообщение  \* 2хх: успешный ответ  \* 3хх: переадресация  \* 4хх: ошибка клиента (404)  \* 5хх: ошибка сервера |

**URI** (Uniform Resourse ID) – унифицир. id ресурса (док, изобр, файл, служба, email)

**URL** (Uniform Resourse Location) – унифиц. локатор ресурса

**URN** (Uniform Resourse Name) – унифиц. имя ресурса

**URI** – имя ресурса, не содержащее месторасп. и способа доступа к ресурсу (можем опр. месторасп. ресурсов в глобальной БД)

**STD** – там прописан URI, STD0 – список всех STD

1. Протокол HTTPS: TLS, шифронаборы, сертификаты, процедура рукопожатия.

**Идентификация** – заявление пользователя о себе.

**Аутентификация –** процедура, направленная на подтверждение идентификатора пользователя (логин и пароль, как правило). На основании какого-то секрета.

**Авторизация** – после аутентификации; проверка есть ли права на выполнение тех или иных действий.

**HTTPS** — протокол безоп.пер.данных, поддерживает технологию шифрования TLS/SSL.

HTTP передаёт д-е в открытом виде. Злоумышленники могут “вклиниться” в передачу — изменить или перехватить данные.

В HTTPS для перед. данных созд.защищённый канал.

HTTPS не является отдельным протоколом. Это обычный HTTP, работающий через шифрованные транспортные механизмы SSL и TLS. Он обеспечивает защиту от атак, основанных на прослушивании сетевого соединения.

HTTPS = TLS + HTTP

\*TLS – прокладка между HTTP и TCP

\*есть рукопожатие

**TLS** – transport layer security – протокол защиты транспортного уровня – криптографический протокол, кот обесп защищ.передачу д-х между узлами в сети Интернет. Исп ассим.шифрование для аутентификации, симм.шифр для конфиденциалности и коды аутентичности сообщений для сохранения целостности сообщ.

Исп-ся в прилож., кот раб.с сетью Интернет, таких как веб-браузеры, работа с эл.почтой, обмен мгновенными сообщ и т.д.

Дает возм-сть К-С прилож.осущ.связь в сети т.о., что нельзя производить прослушивание пакетов и осущ.несанк.доступ.

**TLS – аутентификация**

Аутентификация, которая осуществляется на основе сертификата x.509

**x.509** – стандартный формат хранения и транспортировки атрибутов безс-ти.

- стандарт, кот.разработан международным институтом телекоммун., кот.лежит в основе HTTPS и TLS-аутентификации.

- эл.док, кот. выдается Центром Сертификации.

Содержит: имя держателя, адрес, серийный номер сертификата, даты проверки, открытый ключ держателя.

Вместе с сертификатом выдается секретный и публичный ключ.

Публичный находится на самом сертификате.

Секретный ключ выдается отдельно.

***Схема работы TLS:***

**Клиент** выдает запрос серверу (Client Hello).

**Сервер** подписывает свой сертификат и высылает клиенту (Server Hello).

**Клиент** проверяет сертификат в центре сертификации, которому доверяет.

**Клиент** сравнивает данные сертификата с информацией центра сертификации.

**Клиент** сообщает серверу, какие ключи шифрования он поддерживает.

**Сервер** выбирает подходящую длину ключа.

**Клиент** генерирует симметричный ключ, шифрует его открытым ключом.

**Сервер** получает симметричный ключ и расшифровывает его закрытым ключом.

Для шифрования исп сеанс.ключ. получение общего сеанс.ключа К и С произв.по протоколу Диффи-Хелмана.

На этом заканчивается процедура подтвержения связи. Между К и С установлено безопасное соед., д-е, кот будут передаваться по нему, шифр и расшифр с исп-нием симм.криптосистемы до тех пор, пока соед.не будет завершено.



HTTPS позв аутентиф.C, К (если надо), созд. безопасный канал связи (зашифр)

в основе протокол TSL - обесп аутентификацию + шифрование д-х

**как изготовить сертификат и получить временный?**

этот сертификат в основе TSL-протокола  
предполаг, что К и С доверяют общему центру сертификации

цифровая подпись зашифрована приватным ключом  
подпись мб прочитана с пом. откр ключа

на стороне К дб сертификат удостоверяющего центра с публ ключом

в Сертификата обязательно есть публ ключ  
*необх условие:* чтобы К мог проверить правильность сертификата со стороны С надо наличие на стороне К сертификата удостовр. центра с публ ключом  
тогда он м прочитать цифр подпись, кот-ю сделал удостовр. центр на сертификате сервера  
т.о. он м проверить правильность сертификата

**где хранятся сертификаты?**в ОС в спец. хранилище (м добраться 2 способами: через браузер, через косоль ОС)

чтобы браузер мог делать HTTPS запросы к др. серверам, у него дб сертификаты этих удост центров с откр ключами

шифронабор:

- какие шифры будут исп?....

**Почему сервер должен подтверждать?**мы ему указываем свои данные, кредитные карточки….

**HTTPS**

**Генерация – лк 22**

***options компоненты:***  
1) приватный ключ сервера  
2) сертификат серевра, заверенный центром (т.е. там стоит цифр.подпись центра сертификации, которая зашифрована приват.ключом центра сертификации + центр ему выдал публ.ключ)

Если Сервер хочет поддерживать HTTPS Протокол он должен поддерживать эти вещи выше

TLS ниже протокола HTTP  
мы не замечаем что есть шифрование… работаем как с обычным HTTP, скрыто на более низком уровне стека протоколов

***Как изготовить сертификат?***Общепризнанное ПО для работы с SSL/TSLявл библиотека OpenSSL, устан на ПК  
м.исп нв 2 режимах:   
\* писать код на C/C++  
\* исп ф-ции библиотеки для шифр, созд/провер сертифи…. всё что связано с протоколов SSL, TSL

м. устан сокетное соединение, но исполь security-сокеты, исп шифрования

1. HTML. Структура HTML-страницы. Каскадные таблицы стилей (CSS). Модель DOM. Пример.

**HTML** — **это** язык гипертекстовой разметки документов (HyperText **Markup** **Language**)

HTML-документ — это обычный текстовый документ, может быть создан как в обычном текстовом редакторе **(Блокнот)**, так и в специализированном, с подсветкой кода **(Notepad++, Visual Studio Code и т.п.)**. HTML-документ имеет расширение .html.

HTML-документ состоит из дерева HTML-элементов и текста. Каждый элемент обозначается в исходном документе начальным (открывающим) и конечным (закрывающим) тегом (за редким исключением).

**Начальный тег** показывает, где начинается элемент, конечный — где заканчивается. **Закрывающий тег** образуется путем добавления слэша / перед именем тега: <имя тега>…</имя тега>. Между начальным и закрывающим тегами находится содержимое тега — контент.

Теги могут вкладываться друг в друга, например, <p><i>Текст</i></p>. При вложении следует соблюдать порядок их закрытия **(принцип «матрёшки»)**, например, следующая запись будет неверной: <p><i>Текст</p></i>.

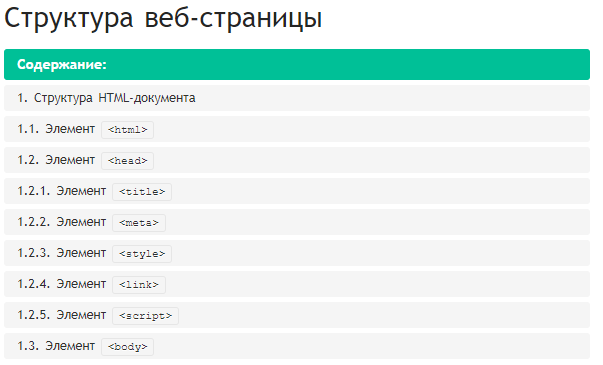
HTML-элементы могут иметь атрибуты (глобальные, применяемые для всех HTML-элементов, и собственные). Атрибуты прописываются в открывающем теге элемента и содержат имя и значение, указываемые в формате имя атрибута="значение". Атрибуты позволяют изменять свойства и поведение элемента, для которого они заданы.

Каждому элементу можно присвоить несколько значений class и только одно значение id. Множественные значения class записываются через пробел, <div class="nav top">. Значения class и id должны состоять только из букв, цифр, дефисов и нижних подчеркиваний и должны начинаться только с букв или цифр.

Браузер просматривает (интерпретирует) HTML-документ, выстраивая его структуру (DOM) и отображая ее в соответствии с инструкциями, включенными в этот файл (таблицы стилей, скрипты). Если разметка правильная, то в окне браузера будет отображена HTML-страница, содержащая HTML-элементы — заголовки, таблицы, изображения и т.д.

Процесс интерпретации **(парсинг)** начинается прежде, чем веб-страница полностью загружена в браузер. Браузеры обрабатывают HTML-документы последовательно, с самого начала, при этом обрабатывая CSS и соотнося таблицы стилей с элементами страницы.

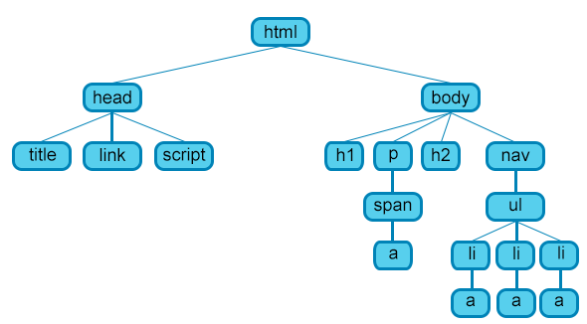
HTML-документ состоит из двух разделов — заголовка — между тегами <head>…</head> и содержательной части — между тегами <body>…</body>.



Язык HTML следует правилам, которые содержатся в файле объявления типа документа *(Document Type Definition, или DTD)*. DTD представляет собой XML-документ, определяющий, какие теги, атрибуты и их значения действительны для конкретного типа HTML. Для каждой версии HTML есть свой DTD.

**DOCTYPE** отвечает за корректное отображение веб-страницы браузером. DOCTYPE определяет не только версию HTML (например, html), но и соответствующий DTD-файл в Интернете.

Элементы, находящиеся внутри тега <html>, образуют дерево документа, так называемую **объектную модель документа, DOM (document object model)**. При этом элемент <html> является корневым элементом.



**Предок** — элемент, который заключает в себе другие элементы. На рисунке 1 предком для всех элементов является <html>. В то же время элемент <body> является предком для всех содержащихся в нем тегов: <h1>, <p>, <span>, <nav> и т.д.

**Потомок** — элемент, расположенный внутри одного или более типов элементов. Например, <body> является потомком <html>, а элемент <p> является потомком одновременно для <body> и <html>.

**Родительский элемент** — элемент, связанный с другими элементами более низкого уровня, и находящийся на дереве выше их. На рисунке 1 <html> является родительским только для <head> и <body>. Тег <p> является родительским только для <span>.

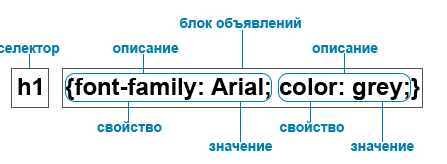
**Дочерний элемент** — элемент, непосредственно подчиненный другому элементу более высокого уровня. На рисунке 1 только элементы <h1>, <h2>, <p> и <nav> являются дочерними по отношению к <body>.

**Сестринский элемент** — элемент, имеющий общий родительский элемент с рассматриваемым, так называемые элементы одного уровня. На рисунке 1 <head> и <body> — элементы одного уровня, так же как и элементы <h1>, <h2> и <p> являются между собой сестринскими.

**CSS (Cascading Style Sheets)** — язык таблиц стилей, который позволяет прикреплять стиль (например, шрифты и цвет) к структурированным документам (например, документам HTML и приложениям XML).

Каскадные таблицы стилей описывают правила форматирования элементов с помощью свойств и допустимых значений этих свойств. Для каждого элемента можно использовать ограниченный набор свойств, остальные свойства не будут оказывать на него никакого влияния.

Объявление стиля состоит из двух частей: **селектора** и **объявления**. В HTML имена элементов нечувствительны к регистру, поэтому «h1» работает так же, как и «H1». Объявление состоит из двух частей: имя свойства (например, color) и значение свойства (grey). Селектор сообщает браузеру, какой именно элемент форматировать, а в блоке объявления (код в фигурных скобках) перечисляются форматирующие команды — свойства и их значения.



1. Протокол WebSockets: принципы работы и применения. Пример.

**WebSocket** – \*новый протокол

\*формат передачи д-х

\*надстройка над TCP для обмена сообщ в режиме реал.времени

(подробнее в устан.соед.)

\*стандарт RFC 6455

\*дуплексный (2 канала связи)

\*пакет require(‘ws’)

\*широковещ.сообщ

\*каналы – *потоки*

\*механизм ping/pong

\*JSON, XML и др.форматы передачи сообщ.

\*RPC

**Каналы связи:**

1. **Дуплексный –** одновременно и прием, и передача (TCP, WebSocket)
2. **Полудуплексный –** одноврем.только прием или только передача (HTTP)

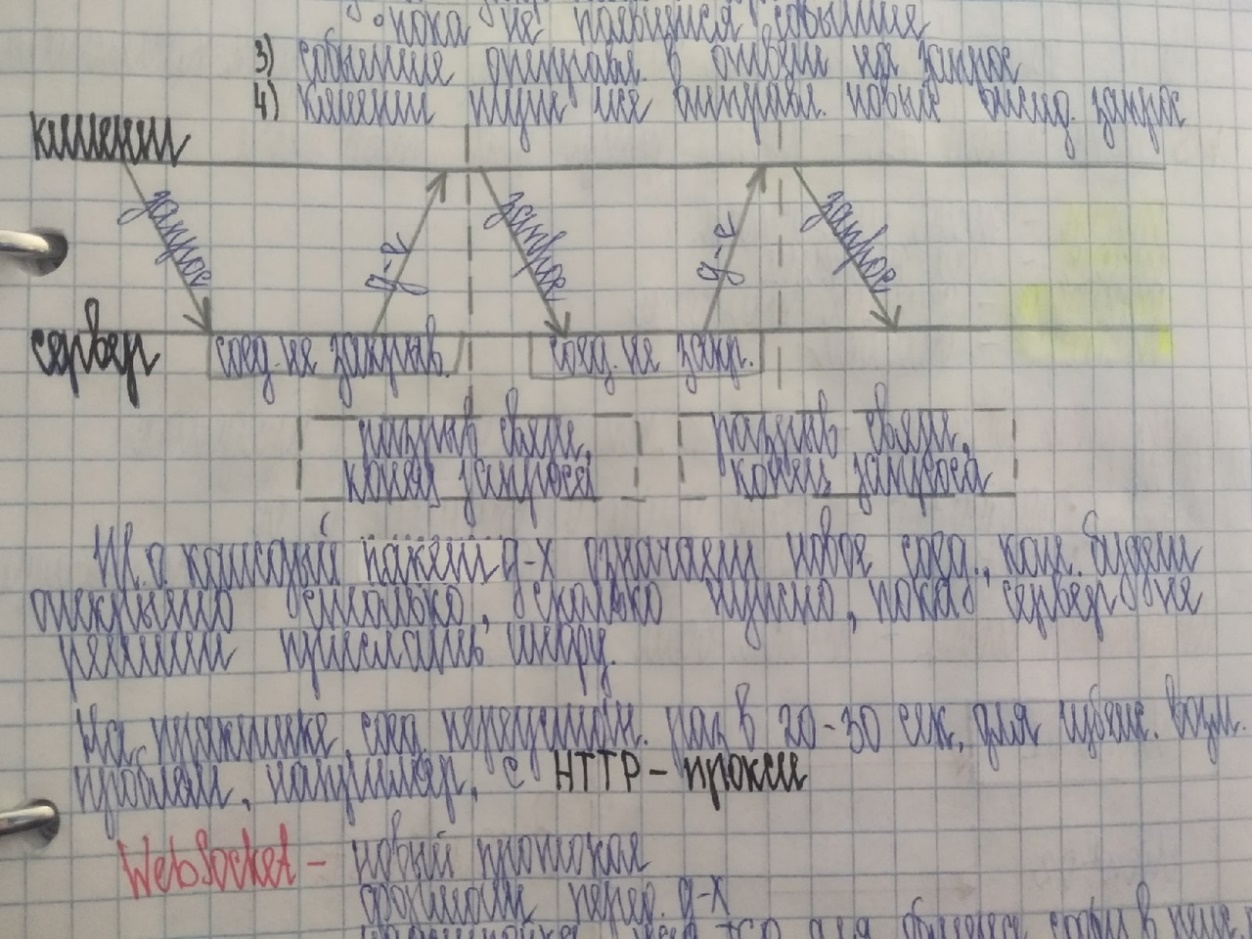
**WebSocket API** – новый программный интерфейс, кот.основан на протоколе дуплексной связи WS.

Протокол работает над TCP-соединением и предназначен для обмена сообщениями между браузмером и web-сервером.

Поддержка вебсокетов осуществляется технологиями корпорации Microsoft: IIS, ASP.NET и WCF.

**Схема:**

1. отправляем запрос на С
2. соед.не закрывается сервером (пока не появится событие)
3. событие отправляется в ответ на запрос
4. К тут же отправл.новый ожид.запрос



Т.о. каждый пакет д-х означает новое соед., кот.будет открыто столько, сколько нужно, пока С не решит прислать инфу. На практике соед.переустан.1 раз в 20-30 сек, для избеж.возм. проблем, например, с HTTP-прокси.

**Процесс установки соед:**

**Upgrade** – К просит С переключится на новый протокол (WebSocket)

**Процесс «рукопожатия»** - процесс установки соед по новому протоколу:

К посыл обычный HTTP-запрос на С с заголовком **upgrade** кот сообщ.С, что К хочет устан.WS-соед.

1. JavaScript. Основные стандарты. Типы данных. Программные структуры. Принцип применения. Пример.

**JS –** то, что делает живым веб-страницы.

JS регистрозависим!!

Подключение js на страницу делается в <script> (либо в head, либо в body)

Когда браузер получает веб-страницу с кодом html и javascript, то он ее интерпретирует. Результат интерпретации в виде различных элементов - кнопок, полей ввода, текстовых блоков и т.д., мы видим перед собой в браузере. Интерпретация веб-страницы происходит последовательно сверху вниз.

Когда браузер встречает на веб-странице элемент <script> с кодом javascript, то вступает в действие встроенный интерпретатор javascript. И пока он не закончит свою работу, дальше интерпретация веб-страницы не идет.

JS практически всегда вставляется перед закрытием body когда практически вся страница загружена – повыш.произв.

Код состоит из инструкций, к.из кот заверш. «;».

Комменты - // либо /\*\*/

Подключить файл - <script src="js/myscript.js"></script>

\* атрибут src – путь к файлу скрипта

**Переменные** предназн.для хранения данных.

Создать переменную – var / let;

**Константы –** const – нельзя изменять.

**Инициализация** (присвоение нач.знач) - с пом =

**5 примитивных ТД:**

* String – предст.строку
  + М.исп.как 2ый так и 1нарные кавычки
* Number – числ.знач.
  + Целые (+/-)
  + Дробные (с плав.точкой (+/-); в качестве разделителя - точка)
* Boolean – логич.знач тру/фолс
* Undefined – знач.не установлено
* Null – неопред.знач. (им.значение, но оно не определено)

Все что не попадает под эти 5 – тип *object* – сложный объект.

Пример: var user = {name:”Tom”, age:24} //свойства

JS – язык со слабой типизацией. Т.е.переменные могут динам.менять тип

var xNumber;  // тип undefined

console.log(xNumber);

xNumber = 45;  // тип number

console.log(xNumber);

xNumber = "45"; // тип string

console.log(xNumber);

**Оператор typeof** – позвол.получить тип пер-ной.

var name = "Tom";

console.log(typeof name); // string

**Мат.операции:**

* Сложение (+)
* Вычитание (-)
* Умножение (\*)
* Деление (/)
* Деление по модулю (%)
* Инкремент
  + Префиксный ++х - увел на 1, а затем вернуть значение.
  + Постфиксный х++– вернуть значение, а затем увел.на 1

// префиксный инкремент

var x = 5;

var z = ++x;

console.log(x); // 6

console.log(z); // 6

// постфиксный инкремент

var a = 5;

var b = a++;

console.log(a); // 6

console.log(b); // 5

* Декремент – уменьшить на 1 (тож префиксный и постфиксный)

// префиксный декремент

var x = 5;

var z = --x;

console.log(x); // 4

console.log(z); // 4

// постфиксный декремент

var a = 5;

var b = a--;

console.log(a); // 4

console.log(b); // 5

**Приоритет:** сначала инкремент/декремент, затем умножение/деление, затем +/-

**Преобразование данных:** PArseInt() – строка в число, PasreFloat() – строка в дробное число

var num1 = "123hello";

var num2 = parseInt(num1);

console.log(num2); // 123

Если преобразование выполнить не получилось – NaN (Not a Number)

isNaN – проверить, представляет ли строка число (число – false, не число - true)

**Массив –** new Array()

alert – выскакивает уведомление

promt – диал.окно с предложение ввести значение

function – для определения функции

function display(){

     document.write("функция в JavaScript");

}

Display(); - вызов функции (ОБЯЗАТЕЛЬНО)

1. Методология Ajax. Структура Ajax-приложения, принципы разработки и применения. Пример.

**AJAX** (Asynchronous JS and XML) – технология, которая основ.на возм-сти XMLTHHPRequest асинхронно отправлять запросы С и обрабатывать ответы.

* Asynchronous означ, что когда К запрашивает д-е с С, браузер не завис.пока не получ.ответ. Пользователь м.перемещаться по стр. Как только С вернул ответ, в фон.режиме ответ нач. обраб.. соотв.ф--циями
* JS – ЯП, который исп для создания объекта запроса AJAX, парсинга ответа и обновления DOM страницы
* XML – для отпраки запроса на С клиент исп xmlhttprequest/XHR API. API – набор методов, кот.задают правила общения между 2 заинтерес.сторонами. Тем не менее, приходящие из AJAX запроса д-е мб в любом формате, не только ХМЛ.

AJAX – Технология обращения к С без перезагрузки страницы. Уменьшается время отклика.

Формат передачи д-х – XML/JSON

Достоинство AJAX – динамичность: применение аякс позволяет сократить сетевой трафик и избежать перегрузок html-страниц.

Недостаток – то, что он основан на хттп, который требует работу в режиме «запрос-ответ». Т.е.для того, чтобы получить порцию данных от С необх.сделать запрос. В прилож., задача кот реагировать на соб, кот происх.на С, клиентской части приходится период.выполнять запросы к С, чтобы увидеть эти события. Для того, чтобы не пропустить серверное событие необх.увеличить частоту запросов.

**Объект XMLHTTPRequest**

var request = new XMLHttpRequest();

После создания м.отправл.запросы к С, но для этого надо вызвать метод open для иниц..

request.open("GET", "<http://localhost/hello.txt>", false);

Метод приним.3 пар-ра:

* тип запроса (гет, пост, пут, делит);
* адрес запроса
* лог.пар-р true/false – указ.булет ли запрос осущ.в ас.режиме (тру – ас, фолс – с)

Также мб бот пар-ры – логин и пароль – если для выполнения нужна аутетиф.

Синхр и ас запрос отлич.тем, что запрос в синхр режиме пока не выполнится запрос – ост.код JS не будет выполнятся. В ас.запросе м.параллельнно с запросов выполнять другий JS.

Отправить запрос – send()

**Свойства XMLHttpRequest** позвол.контролить вып-ние запроса

* status – статусный код ответа ХТТП от С
* statusText – текст статуса
* responseType – тип ответа (text, json, document, blob, arraybuffer, “”)
* response – ответ С
* responceText – текст ответа
* responceXML – xml, если формат ответа от С – хмл

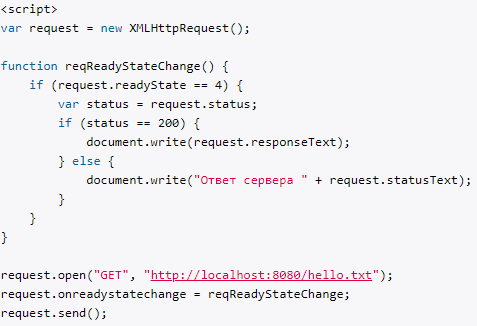


**Ас.запросы**

Работы сложнее чем с синхр., т.к.надо обраб.событие redystatechange объекта xmlhttprequest.

При ас.запросе xmlhttprequest исп св-во readystate для хранения сост.запроса

* 0: объект XMLHttpRequest создан, но метод open() еще не был вызван для инициализации объекта
* 1: метод open() был вызван, но запрос еще не был отправлен методом send()
* 2: запрос был отправлен, заголовки и статус ответа получены и готовы к использованию
* 3: ответ получен от сервера
* 4: выполнение запроса полностью завершено (даже если получен код ошибки, например, 404)

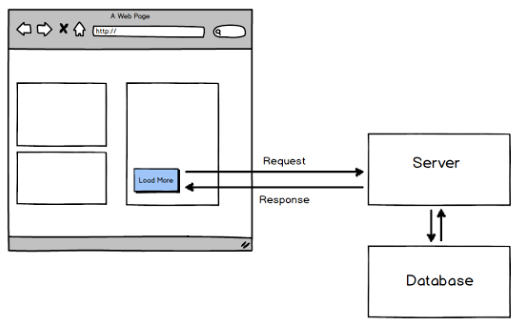


**Что можно делать с помощью AJAX:**

* Элементы интерфейса – полезен для форм, кот связаны с действиями – добавить в корзину
* Динамическая подгрузка страницы
* Живой поиск

С пом AJAX К(браузер) общается с С и запрашивает у него д-е. Полученный ответ обрабатывается в стр вносятся изменения без полной перезагрузки.

**AJAX сценарий**



* Пользователь хочет увидеть больше статей, и он или она кликает на нужную кнопку. Данное событие запускает вызов AJAX.
* Запрос посылается на сервер. С запросом можно передать различные данные. Запрос может посылаться в статический файл (к примеру, example.php), хранящийся на сервере. Также можно выполнять динамические скрипты (functions.php), на каждом этапе которых будет происходить общение с базой данных (или другой системой) для извлечения необходимой информации.
* База данных отсылает обратно запрошенные статьи на сервер. А сервер посылает их в браузер.
* JavaScript парсит ответ и обновляет часть DOM (структура страницы). В нашем примере обновится только сайдбар. Остальные части страницы не меняются.

1. Web-приложение. Архитектура web-приложения. Особенности реализации web-приложения. Web-сервер и web-клиент. Пример.

***КС прил*** – прил, сост из 2 компонент: К и С

К и С взаимод. в соотв. с заданными правилами (*протоколами*)

***Клиент*** – часть прил, явл. инициатором соед

Между К и С дб устан *соединение*

***Web-прил*** – прил с КС архитектурой, где К и С взаимод. по протоколу HTTP (прикл.ур)

**сеть Интернет – 4 компонента:**

1) сеть на основе ст.пр. TCP/IP

2) набор служб (протокол + сервер, все № портов до 1024) : DNS, FTP, TelNet, POP3

3) стандарты (там прописаны протоколы) : RFC, STD

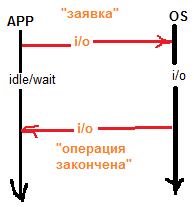
4) организации, упр-щие этой сетью : IAB, IANA, ICANN, ISOC

*backend* – серверная часть, *frontend* – клиентская часть

***Узел интернет*** – у-во, имеющее IP + подключение к сети Интернет

к. узел хар-ся своей ***прогр.-аппарат. платформой*** (аппаратура + ОС)



**Web-сервер:**

принимает запросы, обраб их и отпр ответы К

чтение http-запроса i/o

парсинг http-запроса cpu

запросы к БД i/o

запросы к др. серверам i/o

вычисления cpu

форматиров. ответа cpu

отправка http-ответа i/o

В основном, web-сервер делает в(ы)вод, мало использует процессор

Под всеми к-дами в(ы)вода скрыт системный вызов функции ядра ОС!

Пока ждем ответ ядра ОС, это время надо рационально исп.

*“проблема блокирующего в(ы)вода”* – если ничего в это время не делаем – **синхр. операция** (процессор простаивает)

1. ASP.NET: публикация ASP.NET-приложения, структура и параметры узла IIS, реальный и виртуальный каталоги, процедура настройки web-узла.

ПО умолч.пирлож ASP.NET работают над сервером IIS.

IIS(Internet Information Services) используется для публикации приложений.

WWW Publishing Service – служба ИИС, которая отвечает за упр-ние и работу веб сайтов и веб прилож

**Архитектура IIS:**



**Автоматическая публикация:**

Здесь нам доступно несколько вариантов публикации:

* Microsoft Azure App Service: публикация в облаке Azure
* IIS, FTP, etc: публикация через FTP
* Folder: публикация в виде отдельного пакета в файловой системе текущей рабочей машины
* Import Profile: импорт профиля, который содержит настройки публикации
* Microsoft Azure Virtual Machines: публикация в облаке Azure, по сравнению с первой опцией обладает большими возможностями по управлению инфраструктурой развертывания

Создаем пустую папку. Создаем веб-узел на ИИС. Далее в проекте жмем опубликовать и выбираем Folder для публикации в виде отдельного пакета. Укажем путь к той же папке и нажмем на кнопку опубликовать.

По умолчанию все узлы IIS автоматически запускаются при запуске ПК.

Остановка узла – остановку служб интенета и выгразка узла из памяти пк.

Приостновка – не позвол службам интернета принимать новые подключения, но не влияет на уже обрабатываемые запросы.

В **окне свойств** задаются параметры идентификации веб-узла:

* Описание
  + Пользователь может ввест любое имя севера
* Ip – адрес
* TCP порт – по умол.80, но мб другой
* Подключения
  + Не ограничены – нет ограничений (входите кому надо)
  + Предельное число - макс число 1врем.подключ
* Время ожидания – промежут в сек, после чего С отключается неактивного юзера
* Разрешить поддержку открытых соед.HTTP
  + Устан.флажок помогает К поддерживать откр.подключ к С вместо того, чтобы повторно открывать подключение клиента при каждом запросе
  + Отключить этот режим – быстродействие С на минус
* Вести урнал

**Добавить новый узел:**

* В оснастке IIS выберите компьютер или узел и нажмите кнопку Действие.
* Нажмите кнопку Создать, а затем Веб-узел или FTP-узел для вызова мастера узла.
* Следуйте инструкциям на экране для присвоения новому узлу идентифицирующей информации. Необходимо предоставить адрес порта и путь к домашнему каталогу.

**Домашний каталог**

Каждый веб или FTP узел должен иметь 1 домашний каталог.

**Домашний каталог –** центральное расположение публикуемых страниц. Содержит домашнию страницу или файл указателя с приветсвием. Домашний катало сопоставл.с именем домена или С.

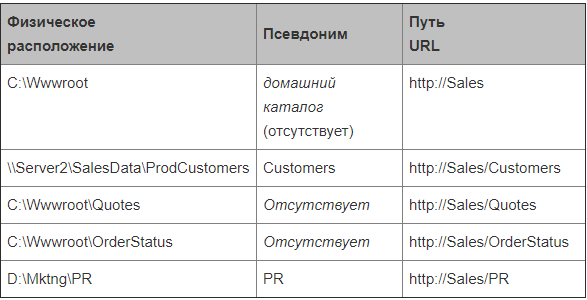
Например, если узел имеет в Интернете имя домена www.microsoft.com и домашний каталог C:\Website\Microsoft, то веб-обозреватели будут использовать адрес URL http://www.microsoft.com/ для доступа к файлам в этом домашнем каталоге

Стандартный домашний каталог создается при установке IIS и при создании нового веб-узла.

**Создание виртуальных каталогов**

Виртуальным каталогом называют каталог, который физически не содержится в домашнем каталоге, но выводится для посетителей веб-узла как его вложенный каталог.

Виртуальный каталог имеет псевдоним, т.е. имя, которое веб-обозреватели используют для доступа к этому каталогу. Поскольку псевдоним обычно оказывается более коротким, чем полное имя каталога, пользователям легче его запоминать и вводить. Применение псевдонимов является также более безопасным. Пользователи не знают, где ваши файлы физически размещаются на сервере, и не могут использовать эту информацию для изменения файлов. Вместо того чтобы изменять адрес URL для каталога, достаточно изменить сопоставление псевдонима и физического адреса каталога.



*Чтобы создать виртуальный каталог*

* В оснастке IIS выберите веб- или FTP-узел, к которому требуется добавить каталог.
* Нажмите кнопку Команда и выберите команду Создать и подкоманду Виртуальный каталог.
* Используйте Мастер создания виртуального каталога для выполнения этой задачи.

*Чтобы удалить виртуальный каталог*

* В оснастке IIS выберите виртуальный каталог, который необходимо удалить.
* Нажмите кнопку Команда и выберите команду Удалить. Удаление виртуального каталога не приводит к удалению соответствующего физического каталога или файлов.

**IIS:**

*Как проходит запрос от браузера (любого http-клиента) до нашего приложения?*   
Любой http-сервер он же является сервером TCP, который слушает 80-порт и работает по правилам протокола http.  
Если мы набираем адресную строку в браузере и нажимаем enter, у нас отправляется запрос. Т.е. у нас есть uri – начинается со схемы (http/https), символ. имя пк. Чтобы присоединиться по tcp, надо знать имя пк, к к-му мы обращаемся. Это симв. имя дб разрешено с пом к-то механизма. Этих механизмов сущ. несколько:  
\* dns – файл hosts  
есть ф-я *get\_host\_by\_addr(name)* – м. получить ip  
*Откуда он знает этот ip?* – в сетевых настройках мы указываем IP dns  
*Если там стоит, что ip адрес получается динамически через dhcp протокол?* откуда тогда берется dns? – dhcp высылает все сетевые настройки, в т.ч. и dns, по к-му м. разрешаться имена

**IIS-сервер** – набор серверов. С каждым узлом связаны свои сет. настройки. Для к. узла указ. протокол, порт, ip. Каждый узел, по большому счету, это отдельный сервер, кот. слушает порт и прослушивает ip указанный в настройках данного узла.

Т.о., мы отправляем запрос, разрешается символ. имя, запрос идет в сторону сервера, попадает на конкр. пк по ip-адресу, в рамках этого пк запрос поступает на конкр. сервер в рамках iis

Т.е., наше приложение – сервер, кот. работает в рамках IIS, их мб много они установлены на узлах этого компа

**Пул приложений** – это окружение нашего прил. – здесь указ № версии .NET. У нас мб неск приложений, каждое работает со своей .NET версией. Его основное назначение: поддержка того окружения, того .NET-фреймворка, в рамках к-го будет работать ASP-приложение + управление потоками ииса  
Дело в том, что iis отъедает часть потоков от windows и организует свю с-му потоков в рамках iis-а. один из эл-тов управления этими потоками – пул приложений

**Сопоставление обработчиков:**  
свои обработчики для каждого узла + в рамках ASP.NET есть ***стандартные обработчики*** – для обработки запросов прил опр. типа (forms, mvc-прил и тд)

1. ASP.NET: http-обработчики, порядок разработки, http-обработчик для взаимодействия с клиентом по протоколу WebSockets. Пример.

**Повторим всё:**

Мы разрабатываем на платформе **ASP.NET**   
**Платформа** – набор библиотек, к которым есть документация (т.е. это API, через которые мы можем добираться к этим библиотекам) + набор инструментов для разработки ПО на основе этой платформы

В vs установлены специальные плагИны для разработки ASP.NET, шаблоны   
**ASP.NET** – платформа 2 уровня, над .NET, для разработки web-приложений  
(*веб-прил* – прил с К-С архитектурой + протокол HTTP)

.NET Framework – прокладка между Windows и нашим приложением, разраб. в ASP.NET, кот работает над платформой .NET Framework  
**.NET Framework** – реализация CLI-спецификации, входит библиотека + CLR (интерпретатор)  
- это DLL, написанная по спецификации CALL, устан. в windows – осн. задача интерпретировать MSIL-код

**Структура:**

Есть .NET платформа. Над ней есть ASP.NET. В рамках ASP.NET платформы м. разрабатывать различные приложения: forms, handler, mvc… Платформа одна, просто разные шаблоны приложений.

**HttpHandler** – простейшее ASP.NET-приложение, предназн. для получения запроса, его обработки, формирования ответа и его отправки.   
у него есть контейнер ***HttpContext*** – его основная задача: хранить запрос и ответ  
в ***config*** прописывается маршрутизация к хендлеру: \* метод, для к-го хэндлер предназначен + path, по к-му к handler-у будет осущ. доступ

**MVC**

  
**Модели** – классы, предст. д-е  
**К-р** – компонент MVC-прил, предст. классом, включ в себя методы, соотв. action

на самом деле есть и 4й компонент – **маршрутизатор**, кот. осн. на таблице маршрутизации  
любой запрос приходит на mvc-сервер (кот. мы вложили в узел iis), запрос встречает маршрутизатор и опред. имя контролера, имя action в рамках этого контроллера, и создает объект контроллер.  
Использ механизм **Reflecton** – ищет в сборке класс, кот. соотв. к-ру (имя к-ра + Controller). Этот класс д. наследовать httpController-класс. С пом Reflection созд объект к-ра и вызываем action этого к-ра + выдергиваем параметры из запроса и передаем этому методу.

***Главное:***  
*\* маршрутизатор создает объект к-ра (с пом* ***Reflection****)   
\* к-р создает объект модели (contextDb в EF)  
\* к-р вызывает готовый объект view ИЛИ впервые вызыв* ***View Engine*** *“Razer” – движок, кот. преобр файл cshtml -> c#-класс View  
\* к-р живет в рамках 1 запроса!  
\* модель живет 1 запрос!  
\* Объект view сохр и живет при след. обращении к нему, до перезагрузки С*

Модель не обяз. реализована с пом. EF, она м. работать с обычным json-файлом (независимость объекта к-ра и модели)

**Global.asax** – спец файл, кот. содержит обработчики ЖЦ приложения, он есть во всех приложж  
он показывает, что приложения (forms, mvc, webapi) все имеют одну и ту же платформу, структуру, и события

Папка **Shared** – там views, общие для неск. к-ров  
Папка <Имя\_кр> – для к-ра  
Если пишем **return View()** без парам, то возвр. view в папке *Views <Имя кр>* с именем action-a

**ViewBag** – коллекция, динам. объект, к к-му м добавлять св-ва – простейший способ передачи д-х между View и к-ром. Есть и другой способ, но данные в любом случае в итоге записаны во ViewBag

|  |  |
| --- | --- |
| В к-ре: | Во View:  м. переприсвоить данные, ибо c# забывает об их типе, лучше переприсвоить. |

@ - переключает с разметки на c# - «возьми этот объект, преобразуй его в строку и подставь ToString от выражения»  
href – м. для перехода между action-ами (?)

Маршрутизатор вызывает action и передает туда параметры на уровне Reflection – т.е. идет сопоставление имен параметров action с именами параметров в запросе. В запросе все парам стринговые и они стараются преобразоваться к типу параметров в action. М задать в action дефолтные параметры (если параметры request не смогут преобразоваться)

Иногда неудобно, если пришел параметр не int, вместо него будет 0 (сопоставления не произошло), мы можем не понять пришел 0 или что-то левое?  
=> поэтому существует **‘int?’** (будут null-типы если ошибка преобразования)

любое приложение связ. с обработкой запроса. Эти запросы м получить с пом объекта Request.

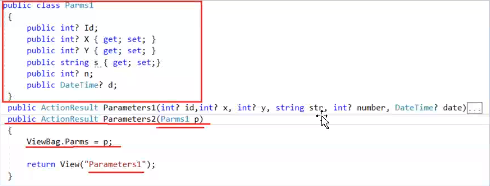
ПЕРЕДАЧА ПАРАМЕТРОВ

Иногда в action передается большой набор параметров (большая форма во view), потом при отладке д-е могут меняться, дополняться => нам придется этот набор переписывать

**способы обработки этих параметров:**  
\* исп. коллекцию Params, вход. в состав Request:  
метод Parameters() – из Request.Params м. получит значение с пом TryParse и запис в пер-ные  
потом заполн структуру new Parms (например)

\* м. аналогично делать во viw (Нежелательно), появл. связь с к-ром. Внутри view получаю this.Request – могу обрабатывать и выдавать значения, кот есть в request

**Параметры в запросе:**  
1) в составе параметров  
2) в составе uri {controller}/{action}/{*id*}

м. объявить класс-структуру Parms1, куда входят поля или свойства  
и вместо набора параметров подставить эту структуру  
этот маршрутизатор будет искать имена в рамках этой структуры  


**3 штуки моделей:**  
1 – *модель action* – набор параметров, кот пришел на action (в виде струкутуры или набора)  
2 – *модель д-х* – то, что мы создаем напр. с пом EF  
3 – *модель view* – return view(“param”) – строка внутри view наз. моделью

**MVC: Controller**

Всегда м. добраться к request, response в нашем ,vc-прил  
в объект Request заталк всякие коллекции (QueryString, Form, Cookies, Params, RouteData….)  
**RouteData** – коллекция, все что касается маршрутизации (имя к-ра, action)

Объект Response – генер д-е можно  
методы Response.Write…

есть **Redirect**-ы (переадрессация)  
301 – временное перемещение  
302 – «ресурс перемещен навсегда»  
если переадресация в рамках сервера, м. коротко: /L03/RWrite  
*RedirectToAction* – указ т. action, если в рамках одного к-ра   
мб переадресации сложнее *RedirectToRoute*(к-р, action, id)

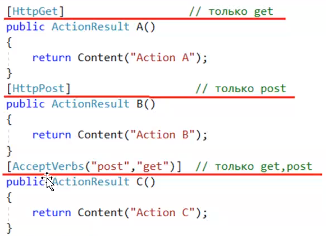
**Content** – способ вернуть из к-ра данные, помещ в body ответа, идет обыкн. текст плэйн, кот. пойдет на сторону К

return **File (“Word.docs”, “appl/msword”, “tt.docx”)** – файл идет в загрузки – происходит download  
чтобы пошел в загрузки, надо чтобы в ответе сущ. спец тег Content-Disposition кот сод. спец. паарметр attachment, иначе браузер будет пытаться его сразу отобразить

**return Json (new {x=1}..)** – подготавливает json, выводит в браузере

если надо динамически формировать скрипт, кот мы передаем Ку на стороне браузера, исп **return JavaScript (“function sum …”)**потом во view м. сделать <scrypt src=”~/L03/JavaScriptx”/> - м. обращ ч/uri  
**HttpStatusCodeResult(403)** – выдать к-то статус

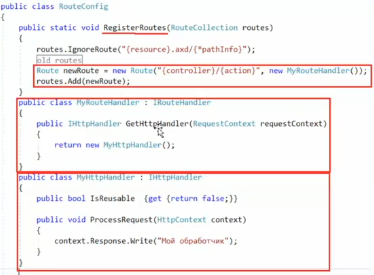
**MVC КОНТРОЛЛЕР**

в к-ре м. сделать маршрутизацию с пом атрибута  
м. указать метод, с пом кот м. заступиться к acton: [HttpGet]  
не указано – с пом. любого метода м. заступиться к action  


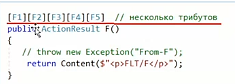
**[Route(“{x:int}/{name}”]** – атрибут маршрутизации, сильно разавита. Если мы использ этот атрубут, то в таблице маршр д. сделать строку: *routes.*M*apMvcAttributeRoutes*() --> сначала проверяет применяем атрибуты маршр. Action д сод те же имена параметров, кот. переданы в атрибуте. Пришедший запрос сначала будет обрабатываться синей строкой «вначале обраб маршруты в соотв. с атрибутами [Route{“x..”}] а потом будет обрабатывать уже маршруты из таблицы маршрутизации. Атрибуты Route приоритетнее.  


лучше указ. маршруты с пом маршрутизатора – тогда механизм образования маршрутов будет в одном месте, иначе будем долго искать где у нас тот action, кот обрабатывает маршрут

**MVC: контроллеры (продолжение). Представление начало**

Что такое атрибут? Буду спрашивать  
***Атрибут*** *– метаинформация, позвол записывать д-е в сборку. Говорим компилятору C#: «Запиши эти данные в сборку». М извлечь инфу из атрибута потом*  
\* позв указать методы, кот приписаны action-у: [HttpGet], [HttpPost], [AcceptVerbs(‘post’, ‘get’)]  
\* *атрибуты маршрутизации* – позв гибко маршрутизировать к-ры и actions. Применять осторожно, только если даст возможность реально сократить код. Иначе потом их надо собирать по всему коду, они не сосредоточены в одном месте.  
для этого надо в м-ре прописать строку: routes.MapMvcAttributeRoutes() – она позволяет потом использ эти атрибуты.  
\* RoutePrefix(‘hi’) – позв задать маршруты с одним общим префиксом, для всего к-ра  
\* м. здесь свой обработчик маршрутов  
в RegisterRoutes, где у нас прописаны все маршруты, мы дополнительно укажем свой пользовательский маршрут:  
  
мы здесь создаем новый Route, и с ним связываю обхект MyRouteHandler, кот. мы сами прописали. Это класс, кот реализует ин-с IRouteHandler – там метод GetHttpHandler, кот возвр HttpHandler (сами его напишем, не впервой)

**ФИЛЬТРЫ ACTION**

**Фильтр** – класс, кот. содержит методы, кот будут вызываться в соотв с нек правилами, кот. соответствуют названиям этого фильтра: есть фильтры *авторизации, аутентификации, акции, результата, исключения*. Каждому фильтру соотв. свой класс, кот мы должны написать. Они делают предвар обработку запроса, кот. мы получаем на какой-то action. Можно для одного action написать неск. разных типов фильтров:  


Сначала вызыв: аутентификации, авториз, action-a, результата. Фильтр результата работает только в одну сторону

**АТРИБУТ КЕШИРОВАНИЯ**

**Кеширование** – процесс копирования данных из медленно-действующей памяти в быстродействующую для повышения производительности.

[OutputCache (Duration=5)] – кешируется response, копия может устареть, если бд уже изменилась. При нек запросе будет получен ответ и он будет уложен в объект «кеш объекта» и при след. запросах в теч **Duration** все запросы будут получать один и тот же ответ. Через **Duration** ответ устраеет и новый результат снова закешируется. Если идет извлечение из БД или зависит от вычислений и времени, кеш будет несколько отставать. Но во многих случаях это вполне приемлимо и существенно повышает производительность сервера, сообенно если вычисления в action-е трудоемки и к нему часто обращаются.

Кешированием можно гибко управлять: на параметры VeryByParm – м. указать имена параметров, кот. дб учитываться для кеширования: какие бы параметры не передавались action-у, все равно будет один и тот же ответ, если нет VeryByParam (он позв вычислять в завис от параметров – будет создан отдельный кеш для конкр значений x,y например)



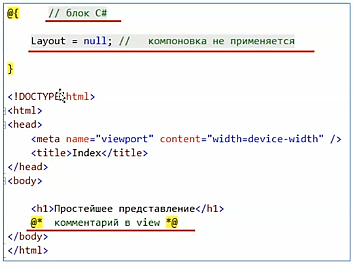
**АТРИБУТ ACTIONNAME**

[ActionName(“H-Name”)] – позв переименовать action, т.е. если будем делать запросы извне и маршрутизировать запросы, эта акция будет действовать т.о. будто она имеет имя H-Name

**Представления View**

В составе ASp.NET есть неск View Engine – ASP X, Razer – их задача: принять на вход файл с разметкой (в завис от engin-a разная), а на выходе образуется c# класс, кот и формирует Response.

**Razer** Engine – движок, вход в состав ASP.NET MVC, на входе обраб cshtml, на выходе объект, кот формирует response. Эта процедура происходит один раз за запуск приложения. Затем если к этому view есть еще запросы, то уже будет использован готовый объект. Создание view-а инициируется к-ром.

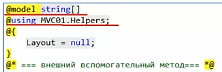
*CHTML файл:*  
  
сод. статический html + разметку, кот. позволяет делать Razer. @{} – переключение разметки с html на c#. @\*\*@ - комменты

В Razer использ компоновка (макет, мастер страниц) – это спец. механизм, позв создать шаблон нашей разметки для неск cshtml-файлов – позв создать общую часть, структуру, общий вид для мн-ва view. Эта компоновка подключ с пом св-ва Layout – оно унаследовано этим классом на картинке (вед view превращ в класс). Этот класс опирается на базовый класс ASP.NET, в этом классе есть св-во Layout. И мы м его использовать, указав компоновку – указать другой cshtml файл, кот. является шаблоном для данного файла. Либо null – шаблон не применяется.

Желательно всегда применять View() без параметров, чтобы директория по умолч была Views/к-р/action.cshtml. Дальше будет искать в папке Shared. View есть простые и строготипизированные.

**Строготипизированные views** – этим представлениям при вызове в action-е передается нек. параметр: return View(new A(3,5)). Это и есть “модель view”, про к-ю мы говорили (3 типа моделей). И потом этот параметр получен внутри view в виде модели, там указан ее тип (имя класса А, кот. мы разработали с к-ре).  
   
Потом во view чтобы обратиться:  
  


Для разработки view MVC предлагает ряд вспомогательных механизмов, позв создавать повторно-выполняемые компоненты:

\* **@helper{}** – вспомогательнный метод с целью повторного применения, м. разработать свой. М разраб внутренние (внутри view) и внешние helper-ы – мб применен в рамках нескольких view. Потом с пом using подтягиваем внешний helper:  
  
Потом м передавать внешний helper CityList (сами его разработали): @Html.CityList(Model)

**MVC View, DI**

Стандартные Helpers:

1) **Html.Action** – возвр string – обращ к action-у, у к-го берет view, передает параметры и забирает у action-а результат, который будет подставлен вместо Html.Action (это разметка – строка или view)  
  
2) **Html.RenderAction** – пишет в response  
 @Html.Action = @{Html.RenderAction}  
3) **Html.Partial** – возвр строку – частич предст – фрагмент предст-я для повтор. использ. Часто их использ совместно с AJAX. В предст вызыв fetch – м. делать запрос к серверу к action-у, кот. возвращает return PartialView(..)  
4) **Html.RenderPartial** (имя част.предст, парам) – то же соотношение, запис в ответ  
5) **Url.Action**(к-р, action) – то же самое, что <http://localhost:14032/V/Contry> например  
6) **Html.ActionLink** (лэйбл, action, к-р) – генерит ссылку, action и к-р при нажатии сработает  
7) **Html.BeginForm** (action, к-р, FormMethod.Post) + **EndForm** – между ними заключена разметка с формой для ввода д-х  
8) **Html.TextBox**, **RadioButton**, **CheckBox, Label** – более компактно записать тэги  
9) **Html.Hidden**, **Password**, **TextArea**   
10) Html.Display, DisplayText  
11) Html.Validation, ValidationSummary (соберет все сообщения о валидации, отобр группой  
12) суффикс FOR: (ориент на модель)

**Применение FORM**

**2 фазы:**  
1) вывести форму для юзера, чтоб заполнил  
2) нажать submit – иниц отправку формы на сторону сервера



Валидация……..

**Компоновка LAYOUT**



сод. статитеский текст +   
+ **RenderSection**(“head”, true) – здесь указана секция head (true – секция обязательна) – именованное место, куда будут подставлены д-е  
+ **RenderBody** – место, куда поступает вся остальная разметка, которую мы не именуем никакой секцией  
if (IsSectionDefined(“foot”)) – м. проверить есть ли секция во view

**MVC: Внедрение зависимостей (DI)**

**Dependency Injection** - Прогр. мех, позв. в авто- режиме создавать прогр. объект с заданным ЖЦ (когда будет создаваться, разрушаться и как он будет внедряться (непосредственно или ч/параметры методов) – мы это м. определить)  
там можно задавать события, инстансирование (созд объекта) и его разрушение  
этот мех. м. исп по-разному:  
\* непосредственно  
\* ч/параметры методов  
м. опред область дв-ия этого мех.

*Для чего?*  
\* чтобы создавать слабосвязанный код  
\* для повторного применения кода  
в ASP, MVC: чаще всего в кач-ве объекта внедрения исп. context db или репозиторий модели д-х, к-ю мы разработали в прил.  
чтоб мб репозиторий (объект модели) подменить и исп. свою (для автомат. тестирования)

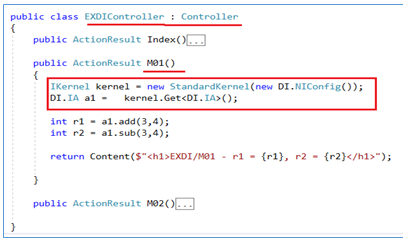
**IoC (Inversion of Control)** – принцип программирования, позвол. снизить зависимость между компонентами программ; DI – один из способов реализации IoC

есть неск. *IoC-контейнеров* (фреймворков), позв. использ. механизм IoC: NinJect, AutoFac, Unity…

*Какие пакеты?*  
надо скачать неск. библ (это мин набор) из Nuget:  
\* Ninject  
\* Ninject.Extensions.NamedScope  
\* Ninject.Web.Common.WebHost  
\* Ninject.Web.Mvc

Прямое (непосредственное) применение DI:  


\* интерфейс с методами IA  
\* класс, кот. их реализует A1 : IA  
\* дальше создаем файл конфиг DI, производный от NinjectModule (он получен из этих пакетов NuGet)  
там реализ. метод Load(), в кот. прописываем Bind (связ. имя IA с именем A1 – указ. механизму DI, что имея ввиду IA, мы подразумеваем что A1 это реализация этого ин-са)

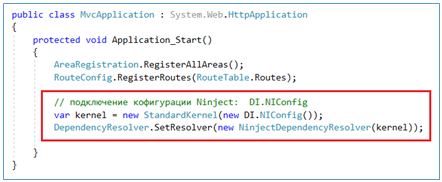
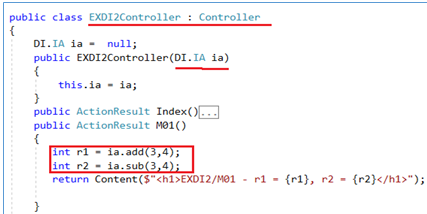


Самый простой метод внедрения DI. есть контроллер, есть Action, в к-м мы : создаем объект StandartKernel (из Nuget) и ему указываю свой конфигуратор  
т.о. в A1 получаем объект a1

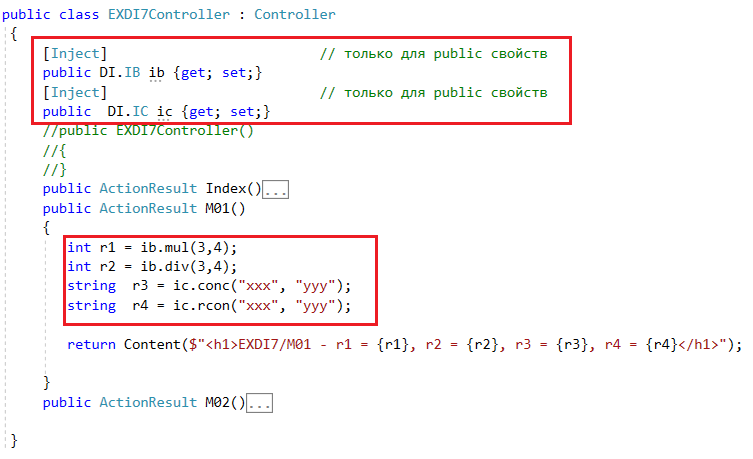
**Зачем всё?**  
мы здесь используем интерфейс, а выше указали, что вместо него имеем ввиду реализацию A1. Эти части нах. в разных частях кода. И если потом потребуется заменить реализацию, мы это сделаем в рис.1. Мы осущ позднее связывание с конкр. объектом  
Потом просто перепишем конфигуратор, свяжем например с классом А2. Появ независимость от конкр реализации объекта, кот реализует ин-с. К-р менять не придется.

DI/Ninject: глобальная регистрация зависимостей Global.asax

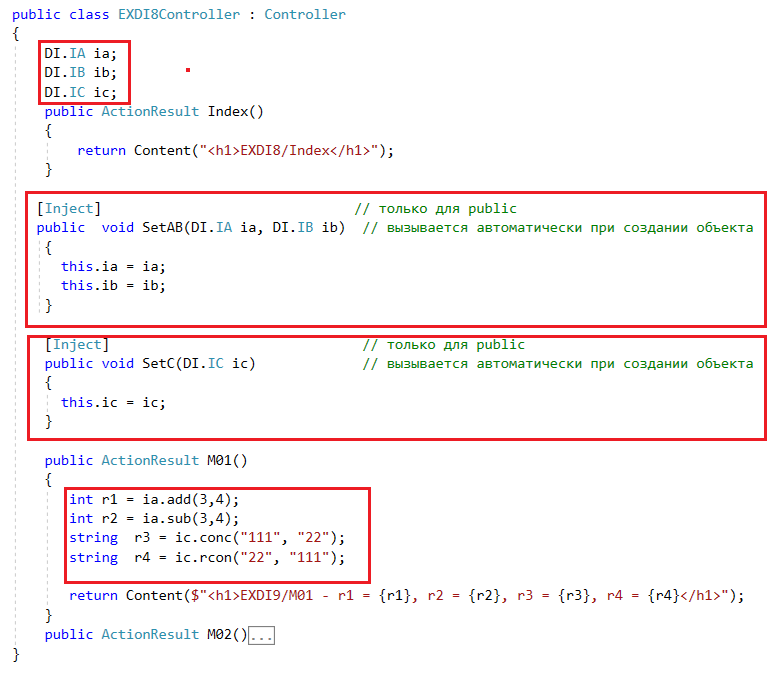
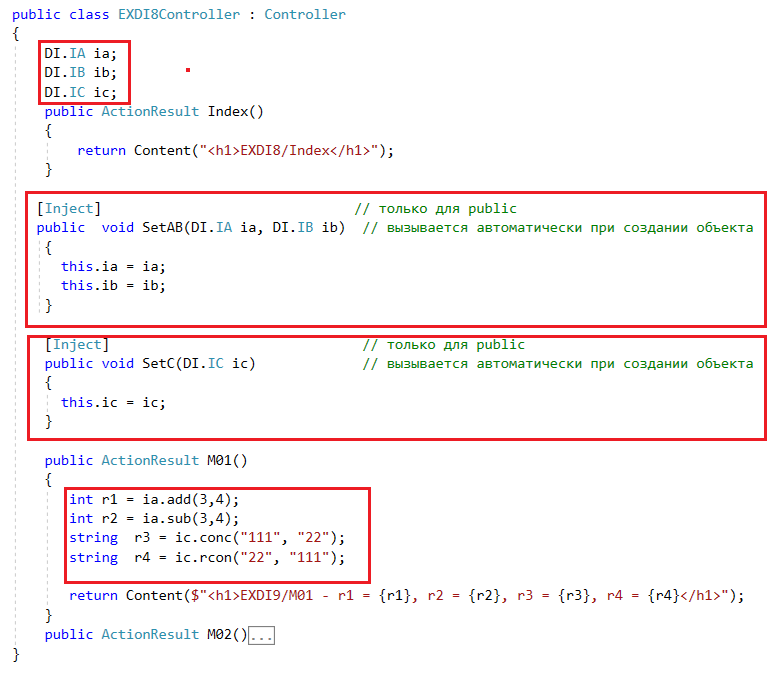
эту часть, где создается kernel, мы перетащим в Global.asax

Global.asax – файл, есть во всех прил ASP.NET, нужен чтоб обраб. события ЖЦ приложения, неважно какое это asp-прил. (handler, mvc, forms…)  
там мы м. подпис. на нек. события  
нас интересует *Applicaton\_Start()* – возникает, когда стартует прил (если устан на IIS, то оно стартует почти сразу авто-) – туда можем подсунуть свой код, например DI (созд kernel и ч/resolver его регистрируем в прил)  
  
после этого м. применять его более удобным способом:  
  
\* в к-ре дописали конструктор с параметром типа IA – когда стартует к-р, ему в кач. параметра передается объект ин-са  
- м. создать несколько классов внедрения, ин-сов  
- м. передавать сколько угодно парам в к-р (несколько DI подставить)  
- м. создать несколько модулей конфигурации – в создании StandartKernel пишем их ч/запятую в парам

Внедрения для свойств [Inject]:

+ исп атрибут [Inject]:  
те же объекты реализации и исп св-во этого ин-са (будет подставлен объект, кот. связан в конфигураторе). Надо явно указывать, что это [Inject]  


Внедрение для методов внутри к-ра [Inject]:

\* не являются action-ами (м явно указать атрибут [NonAction])  
\* [Inject] предполаг применение параметров, кот. дб подставлены авто-  
\* сначала вып конструктор (его м. и не быть), потом пробежится по [Inject]  


**SCOPE (Область действия) InTransientScope:**по умолч: при каждом внедрении (упоминании) создается новый объект (это область д-я InTrancientScope):  
  
**InTransientScope** – новый объект внедрения на каждый вызов  
**InSingletonScope** – сохр объект, исп один и тот же, даже если новые запросы (к-р пересоздается) на весь ЖЦ прил  
**InThreadScope** – сохр объект для потока (поток создается iis-ом, выделяется для вып-я нашего кода. он отрабатывает и мб так, что след запрос будет работать в том же потоке), для к. потока будет создаваться свой экземпляр объекта  
**InRequestScope** – т. для ASP.NET – созд объект внедрения на к. запрос

***Поток*** - средство или механизм диспетчеризации процессора в ОС

***Поток данных*** - абстракция над данными в виде последовательности данны

**15.05 ЛЕКЦИЯ. Web-сервисы (3 технологии)**

1. Компонентные технологии: CORBAB(OMG), COM. DCOM, COM+, RMI, XML-RPC, SOAP, EbXML **EbXML** (Electronic Business XML) – проект спонсируемый UN/CEFACT (центр ООН при поддерржке процедур практики управления коммерции и транспорта) и OASIS (стандартизирующей организации по реализации XML)

**Идея:**Когда мы подключаем компоненты, они подкл в режиме RunTime (позднее связывание). Давайте наделаем эти стандартные компоненты и будем их разным образом компоновать, превратим прогу в индустриальное произв-во – будем разраб отд. компоненты, из кот будем складывать с-му. MS в этом преуспел (спецификация COM, COM+)

Разработали – неплохо было бы их расположить в сети. М с ними работать как с обычными классами. Чтобы юзеру было незаметно – работает он с локальным компонентом или удаленным, мы не замечали. Чтобы компоненты мб перемещать в другие места в сети, но прога юзера оставалась работоспособной все равно.

Появилась спецификация **DCOM** (это протокол, а 3COM – технология) – мб размещать компоненты на разных пк но исп. их можно только в лок. сети. Она очень капризная, если что-то меняется +сложная

Была разработана специифкация **CORBA** – основана на COM+, DCOM, но в ней был новый уровень абстракций – настройка над этими технология, она позволяла более легко создавать распределенные технологии. на ней много очень написали кода и до сих пор нек. с-мы ее используют.

От нее начали отказываться, все признали что идея сделать разработку ПО прозрачным относительно сети, т.е. работали с прогр. объектом удаленно и не замечали, идея оказалась неплодотворной. Перешли к другим технологиям – они осн. на HTTP Протоколе.

**Web-сервис** – прил, предост откр интерфейс для других приложений.Его юзеры – другие приложения. Мы говорим здесь о WEB.API приложениях.

**SOA** – сервис-ориентид-архитектура – наше прил взаимод. по протоколу http с др. приложениями, чтобы из них брать инфу или вып. другие д-вия.

Все веб-сервисы на данный момент м. разбить **на 2 группы**:  
\* **RPC** – обесп интерфейс удал. процедур (Смелов счит. более правильным), ин-с предост. в виде удал объека, у кот-го есть доступные методы кот мы можем вызывать внутри лок. проги  
\* **REST-сервисы** – намного проще. Представлены конечному юзеру как набор uri, к к-м м сделать 4 вида запросов: get, post, put, delete – и предост. правила взаимод. с этим набором uri

**ASMX сервисы** – обесп способ взаимод-я rpc  
**4 технологии, на к-х основ этот веб-сервис**: XML, SOAP, WSDL, UDDI

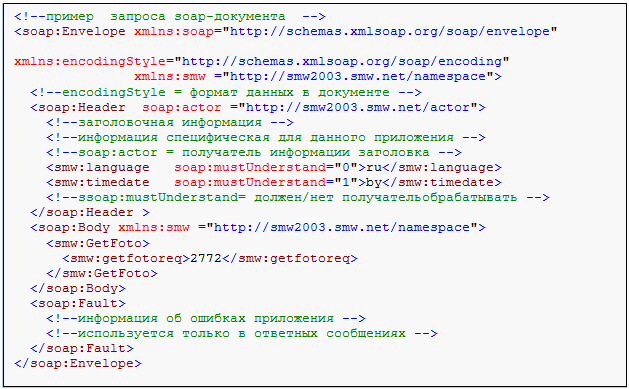
**\* XML** – структура языка разметки, с пом. него мы м. разраб собст. языки для разметки наших д—х, храним эти д-е в фс либо в бд

\* **SOAP** – протокол обмена инфой в технологии ASMX для веб-сервисом и в технологии DAV-CF. Сам протокол прослужил долгую жизнь – много раз изменялся и подходы к нему тоже. Изначально расшифр как simple-object-empties-protocol – Но после версии какой-то предложили это названия перестать расшифровывать. Этот протокол позв. организовать обмен инфой между клиентом и сервером. Клиент м. отсылать запросы по протоколу SOAP и по нему получать ответы. В начале этот протокол был ориентирован на технологию вызова удал. процедур. Но потом его стали применять и для обмена по другим сервисам. Этот протокол подразум. обмен сообщениями в формате xml. Д. создать xml-док (сформир envelope) и его доставить до сервера, С м. его разобрать, посмотреть что мы попросили в запросе и сформир ответ (envilope сформир.) SOAP м работать по протоколам tcp, namedpipe…. Мы собираемся перемещать сообщ в рамках http протокола. Мы soap использует http-протокол в кач-ве транспорта. Это значит, что soap-сообщения будут помещены в тело запроса и в тело ответа. И будем исп спец. заголовки, кот будут говорить что мы перемещаем в нашем запросе/ответе soap-сообщение. Content-type = application/xml+soap.

\* **WSDL** – web service definition language – спец язык, исп формат xml для описания web-сервиса. Он позв нам описать веб-сервис досконально, исчерпывающе. Чтобы мб обратиться к этому сервису с запросами и знать формат ответов кот мы от него получим. Позв созд спец файлы, кот. написаны на языке xml, и кот описывают сервисы. Обычно wsdl сод много комментов, для чего сервис, кому он принадлежит и т.д. Там же нах формальное описание с пом xml-тегов. Имея такое описание, программист может его взять и работать с этим wsdl-описанием и м. обращаться и работать с этими сервисами. Впервые мы с wsdl столкнулись в 2007 г.

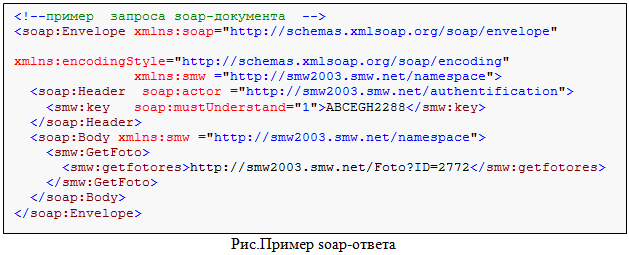
\* **UDDI** – Universal Description Discovery & Integration – эта технология подразумевала создание стандарт. хранилищ в wsdl-описаниях – каталоги, кот. будут содерать описание сервисов, Будут какие-то большие компании (алик) и у них будет оч много сервисов, к к-м другие компании смогут подключать свои прогр. продукты. MS даже сделал этот каталог на основе сервера, он входил в состав ОС какого-то сервера. Там одна из служб была UDDI, но потом эта технология не задалась и развития не получила.

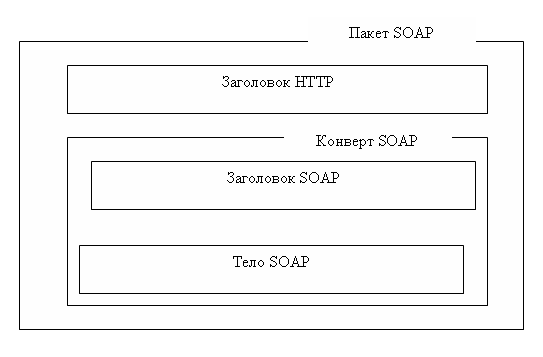
**Структура SOAP-СООБЩЕНИЯ** (БУДУТ НА ЭКЗАМЕНЕ СПРАШИВАТЬ)

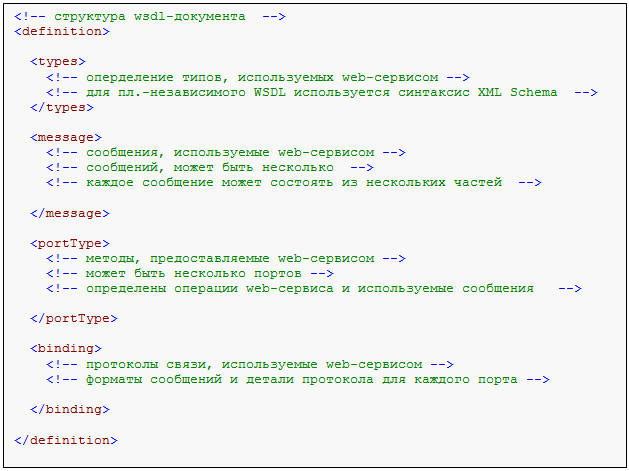


Пример soap-запроса (это xml)

*3 осн. компонента:*  
<Envelope> - корневой тег, кот опред сообщение и пр-во имет, исп-ное в доке. Обязателен толко тег <body> (это запрос от К к С, предполаг что будет вызыв процедура GetFoto). Есть еще <soap:header> (служ инфа для программистов), <soap:Fault> (ошибки)

  
тоже Envelope, Header, Body (отвеч процедура GetFoto – ее результат это response). Т.е. на клиенте есть программа кот вызыв удаленную процедуру GetFoto, передавая ей парам. С отрабаывает GetFoto, возвр строку uri

  
Есть много серверов, кот. поддерж SOAP

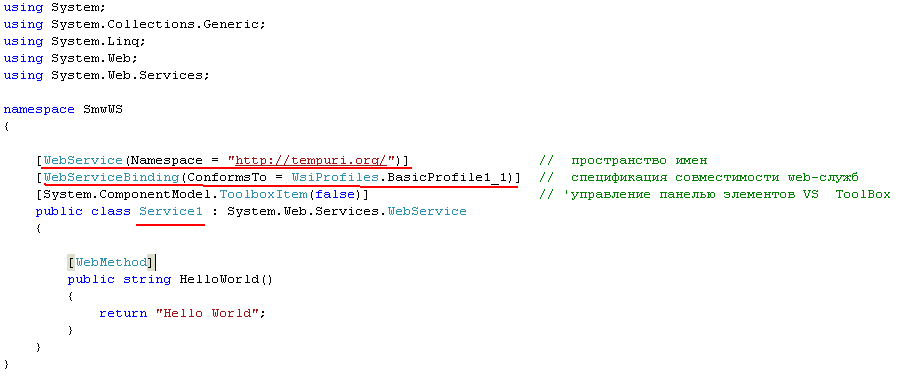
**Структура WSDL дока:** ( НА ЭКЗАМЕНЕ БУДУТ СПРАШИВАТЬ)  


**Разделы WSDL:**  
\* **TYPES** – тд от К к С, мб стандартные и комплексные – структ/класс содержат поля/св-ва, содержащие фундаментальные ТД  
\* **MESSAGE** – опис сообщения между К-С, здесь опис их структура и там исп ТД, кот опис в *types*  
\* **PORTTYPE** – сод. методы удалнного сервера, у сервера есть методы, удал. процедуры кот мы можем вызывать. Эти методы связаны с сообщениями (*messages*), они могут их получать и отправлять, сообщ кот описаны выше  
\* **BINDING** –сод. описание связи soap и протокола транспорта (http), здесь инфа кот позволяет связать soap-протокол с этим http-протоколом, по к-му мы отсылаем сообщение

**ТЕХНОЛОГИИ:**

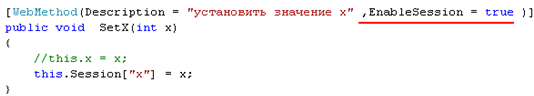
Вначале появилась ASMX, кот основана на SOAP, потом отдельно разрабатывали платформу **WCF** – отд платформа для разработки прилоежния с SOA . В итоге WCF поглотила ASMX (ASMX – частный случай WCF, тем не менее она достаточно простая и удобная, ms оставил ее отдельно). Но с другой стороны она часть общей технологии asmx. В качестве транспорта исп только веб. Если разработаем сервис asmx, то он обязательно подлежит его публикация на iis-сервер.

**Признаки веб-сервиса ASMX:**  
\* SOAP-протокол  
\* HTTP в кач транспорта  
\* исполняется IIS-сервером (дб опублик)

****

методы в классе надо декорировать атрибутом [WebMethod], когда вызываем извне. Вся архитектура SOAP поддерж организацией OASIS.

жмем **Запустить** – создается сервис (dll, кот имеет свой webconfig, и дб опубликована на IIS), запускается внутренний IIS-сервер, на кот публикуется DLL, и он готов для работы  
Создается узел Localhost, и сразу запускает браузер, кот. делает запрос на этот узел и получает инфу о методах, кот. мы создали, и м проверить их работоспособность. К. новый запрос создает новый экземпляр объекта.

Есть укороченная версия ответа, envelope полностью не передается, где нет никаких headers, fault и т.д.  
По умолч состояние не сохраняется, но м использ сессии. М передать параметр EnableSession = true. this.Session[“x”] = x, присваиваем, можем извлечь.   
  
в веб-конфиг, обязательно разрешить куки!  
если в браузере в конце uri службы Поставить ?wsdl, нам вернется wsdl-документ (он созд авто-, либо мы создаем wsdl, а vs сгенерирует программу)

\* созд прилож  
\* добавить службу  
\* вставить адрес на службу (в этот момент она дб запущена)  
если в вс – один экземпляр в подвешенном состоянии дб запущен сервис, в другом экземпляре доб. ссылку на службу  
\* указ пр-во имен, жмем ОК  
\* вс генерирует исх код прокси-класса (он имеет те же методы, что и удал. сервис, но находится здесь, эмулирует работу удал. сервиса, обращ как с обычным классом – это заместитель, кот притворяется что он сервис, им. те же методы)  
\* созд объект этого прокси-класса, вызываю его методы

***Proxy class***

класс, построенный на основе описания сервиса(wsdl)

он обменивается соап сообщениями с веб сервисом вместо тебя, для тебя это выглядит как обращение к объекту этого класса

**Лекция, Сервисы**

**WCF: (Windows Communication Foundation)**

– более общая технология, снимает ограничения ASMX  
– это платформа (набор библ + инструмент для работы с ними), позв создавать приложения с SOA-архитектурой (SOA – сост из потребителей, поставщикаов, реестра сервисов)

**Отличия от ASMX:**  
\* М обесп различные модели взаимод К-С (однонапр, дуплекс, потоковая, П/И)  
\* Поддерж различные протоколы  
\* Поддерж неск точек доступа (asm - 1)  
\* Поддерж self-hosting (м запускать сервис без iis, а встроить его в .NET-прилож (Forms, консольное и т.д.)

– м. отправить сообщение и не ждать результата (однонаправл модель)  
– м. иметь дуплекс, 2 независимых канала связи  
– мб *потоковая модель* взаимод-я – К делает запрос, С посылает неск. ответов  
– м сделать модель Издатель-Подписчик  
– **архитектура**: есть К (м содержать прокси-класс) и С (исполн в рамках хоста). Хост м встраивать в себя сервис, можно встроить в сервис в консольное прил, запустить ч/cmd… Сервис WCF м поддерж неск точек доступа. В ASMX мб только 1 точку доступа. в WCF м обращ по tcp, http-протоколу (ч/различные точки доступа, кот. ДОЛЖНЫ отличаться протоколом доступа). По разным точкам доступа м обеспечить различные сервисы.

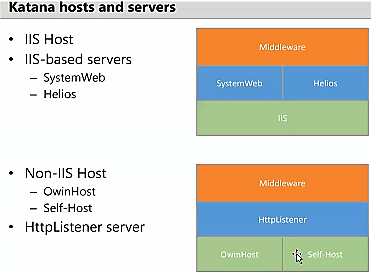
**WEB API:**

**WEB API** – обычное MVC-прилож (веб-сервис), кот обесп REST-интерфейс – набор uri (get, post, put, delete), опис доступ к удаленной коллекции объектов, кот мы м считывать, удалять, корректировать.   
у к-ра другой базовый класс ( : ApiController)  
другая маршрутизация

**OWIN**

**OWIN** **(Open Web INterface for NET)**  
– ин-с между net-web-сервером, серверным приложением (обработчиками запросов) и хостом  
подход к разработке веб-сервером

архитектура:  
  
есть OWIN-сервер (веб-сервер) и есть внутри наше приложение (оранж и желт). OWIN опис как они м/собой взаимод. User Agent – http-клиент, взаимод с нашим приложением. Хост нужен чтобы создать процесс ОС, кот потом м включить в себя объект сервера. Сервер приним запросы, обраб битовую послед-сть, формир респонс и отпр его. OWIN – набор ин-сов, кот. должны реализовать объекты сервера, объекты хоста и объекты приложения. Когда разраб прилож, мы фактически реализуем методы, опис в этих ин-сах.

**Katana** – проект (сервер), кот реализовал спецификацю OWIN. В кач-ве хоста м использ IIS или др. приложение (self-hosting) + этот ин-с позвол подкобчать middleware-модули. Катана предлаг *2 варика реализации сервера:*   
1) осн на IIS – серверу предлаг серверы: SystemWeb или Helios, и middleare, и IIS  
2) без IIS – исп OwinHost или Self-Host (встроенный хост), и 1 сервер HttpListener, и middleware  


Этот подход, известны ин-сы, кот дб реализованы сервером, хостом. И мы м разраб свои компоненты на основе этих ин-сов, и свои конфигурации сервера.

**лекция 29.05 – ASP.NET CORE**

это платформа МС, предн для разраб веб-прил.  
явл. развитием овен? это альтернатива НЕТ у нее своя реализация СЛР  
это фреймворк с откр. кодом, можно скачать  
она м работать в прил, кот м. работать как в НЕТ.КОР так и в .НЕТ  
прил, кот мы разраб на КОРЕ, могут раб. на ИИС и сами (как веб-постинг)   
всё абсолютно что м в нем устанавливать, сделано в Нугет  
веб-пэджис уже не нашел шир. распространения, нет кор больше его не поддер

1. ASP.NET: ASMX-сервисы, WSDL, SOAP, прокси, порядок разработки, принципы применения. Пример.
2. ASP.NET: MVC-приложение, структура MVC-приложения, назначение основных компонентов приложения, маршрутизация. Пример.
3. ASP.NET: MVC-приложение, маршрутизатор, принципы устройства и работы. Пример.
4. ASP.NET: MVC-приложение, маршрутизация с помощью атрибутов, констрейны маршрутизации, принципы работы. Пример.
5. ASP.NET: MVC-приложение, котроллер, жизненный цикл контроллера, взаимодействие с моделью и представлениями. Пример.
6. ASP.NET: MVC-приложение, представление, Razor Engine, жизненный цикл представления. Пример.
7. ASP.NET: MVC-приложение, модель, жизненный цикл модели, репозиторий. Пример.
8. ASP.NET: MVC-приложение, внедрение зависимостей. Пример.
9. ASP.NET: MVC Web API, структура Web API-приложения; назначение основных компонентов приложения, маршрутизация. Пример.
10. WCF-сервисы: WSDL, хост, прокси, модели взаимодействия клиента и сервера, порядок разработки, принципы применения. Пример.

**2ой вопрос**

1. ASP.NET CORE: программная платформа, принципы работы, архитектура. Пример.

**ASP.NET Core** – программная платформа, кот.разраб.Microsoft и предназнач.для разраб.веб-прилож: от небольших веб-сайтов до крупных веб-порталов и веб-сервисов.

Явл.развитием OWIN (The Open Web Interface). Katana – OWIN-совестимый хост, раб.с микрософт.

**.NET CORE –** альтернатива .NET, своя реализация CLR (CoreCLR) и своя библ (CoreFX).

**ASP.NET Core** – фреймв.с откр.кодом (opensource фреймворк) и его м.всегда скачать с гитхаба, где есть полностью исх.код.

Последняя версия – 3.1 (2019г)

Прилож., кот.разраб.на этой платформе могут работать как .NET CORE так и над полной .NET.

Если прилож.раб.над NET Core, то обесп.кроссплатф.над Windows, Linux, Mac OS.

Прилож. ASP .NET Core могут работать под IIS (Windows) или под web-С Kastrel (кроссплатф.варик)

|  |  |  |
| --- | --- | --- |
| ASP.NET 4.6 | ASP .NET Core 2.0(3.1) | |
| .NET Framework 4.6 (Windows) | | .Net Core 2.0(3.1) (Lnix, Windows, Mac OS) |
| .NET FCL | | .Net Core Library |
| Runtime Components – компоненты собственной ОС | | |

Модульность обеспеч.то, что все отдельные компоненты загруж.через NuGet и нет необх.примен. библ.System.Web.dll (осн.библ.для прилож. ASP.NET) => обеспеч.кроссплатформенность, т.к.NuGet раб.на различн.платформах.

Хар-ся расширяемостью. Фреймворк построен из наора относ.независ.компонент. И мы м.исп. встроенную реализ.этих компонентов, либо расшир их с пом.мех.наследования, либо создать и применять свои компоненты со своим функционалом.

МБ разраб.след.***типы прилож***:

* MVC UI, Web API, Web Pages (не нах.шир.примен)
  + ASP.NET Core включает в себя фрейморк MVC, кот.объединяет функциональность MVC, WebAPI и Web Pages. В пред.версиях платформы д-е технологии реализ. Отдельно и поэтому содержали много дублирующей функциональности. Сейчас они объединены в 1 программную модель ASP.NET Core MVC.
  + В модель MVC доб.ряд доп.ф-ций: тег-хелперы, кот позвол.соед.синтаксис html+c#
* Web Form не поддерживается

**ASP .NET Core** поддерж.вижлой начиная с 2015 года, а также вижул студио код (кроссплатф.инстр)

Работая с core мы разрабатываем web-приложение с помощью таких языков, как c#, html, css, js.

Проверить установку .NET Core (ком.строка от админа):

* dotnet --version – проверить версию
* создание приложения – dotnet new console –output FirstApp
  + создается файлик .csproj – предназ.для вижлы и позвол раб.в вижле с проектом
* зайти в папку с приложением – cd FirstApp
* выполнить приложение – dotnet run – все сформируется, скомпилируется и выведет рез.

***Задача*/***Создание веб-приложения:*

Visual C# => Веб => веб-приложение ASP.NET Core => делаем его пустым для того, чтобы посмотреть что там есть в начальном проекте.

*Структура пустого приложения:*

* Connected Services: подключенные сервисы из Azure
* Dependencies: все добавленные в проект пакеты и библиотеки, иначе говоря зависимости
* Properties: узел, кот.сод.нек.настройки проекта.
* appsettings.json: файл конфигурации проекта в формате json
* Program.cs: главный файл приложения, с которого и начинается его выполнение. Код этого файла настривает и запускает веб-хост, в рамках которого разворачивается приложение
* Startup.cs: файл, который определяет класс Startup и который содержит логику обработки входящих запросов

*Program.cs:*

|  |  |
| --- | --- |
|  | создается хост  исп.стрелочная ф-ция, кот запуск.цепочку ф-ций и создает хост и сод.станд. Kastrel http сервер  CreateDefaultBuilder – построение хоста. |

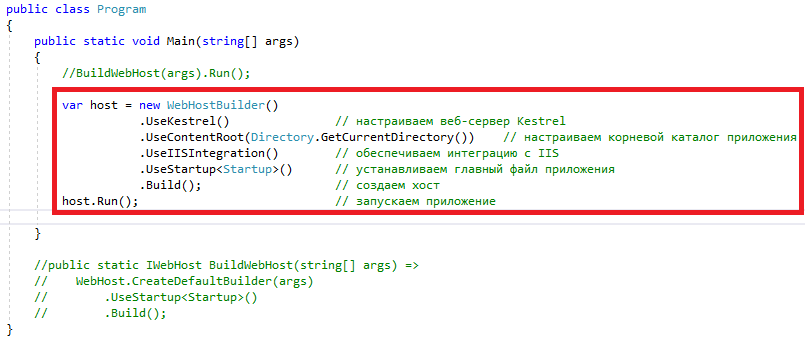
Чтобы запустить прилож.ASP.NET Core, необх.объект **IHost**, в рамках кот.развертывается веб-прилож. Для создания IHost применяется объект **IHostBuilder**. В программе по умолчанию в стат. методе CreateHostBuilder как раз создается и настраивается IHostBuilder. Непосредственно создание IHostBuilder производится с помощью метода **Host.CreateDefaultBuilder(args)**.

Далее вызывается метод **ConfigureWebHostDefaults()**. Этот метод призван выполнять конфигурацию параметров хоста.

А помощью последовательного вызова цепочки методов у объекта IWebHostBuilder производится инициализация веб-сервера для развертывания веб-приложения. В частности, в данном случае у IWebHostBuilder вызывается метод **UseStartup()**. Этим вызовом устанавливается стартовый класс приложения - класс Startup, с которого и будет начинаться обработка входящих запросов.

В методе Main вызывается метод у созданного объекта IHostBuilder вызывается метод **Build()**, который собственно создает хост - объект **IHost**, в рамках которого развертывается веб-приложение. А затем для непосредственного запуска у IHost вызывается метод **Run.** После этого приложение запущено, и веб-сервер начинает прослушивать все входящие HTTP-запросы.

*Подробнее о CreateDefaultBuilder:*

****

*Startup.cs* – класс, кот сод.логик обработки вх.запросов – производит конфигурацию прилож, настраивает сервисы, кот.прилож будет исп, устанавл.компоненты для обработки запроса или middleware.

|  |  |
| --- | --- |
|  | Startup – класс, кот явл главным файлом веб прилож  \*м.сод.необяз.к-р  2 метода:  ConfigureService – необяз., вызыв.после к-ра и примен для регистрации сервисов  Configure – обяз., вызыв после ConfigureService – обработка сообщ кот прих.по протоколу http (запускаем run и формируем response => на все запросы отвечаем “Hello World”) |

Разберем подробно:

* 1. Выражение ***if (env.IsDevelopment())*** проверяет, находится ли прилож в состоянии/статусе разработки. Что это значит? Для проекта можно указать, например, через настройки, что он находится в состоянии разработки.

Вообще условно есть **три состояния** или стадии проекта: в состоянии разработки (Development), в состоянии подготовки к развертыванию (Staging) и в состоянии полноценного использования (Production), когда он уже развернут на каком-нибудь сервере, и пользователи могут к нему обращаться. По умолчанию Visual Studio устанавливает для проекта состояние разработки. И данное выражение как раз проверяет состояние.

Если проект находится в состоянии разработки, то, возможно, мы захотим применять некоторые действия, которые не нужны, когда приложение уже развернуто. Так, по умолчанию вызывается метод *app.UseDeveloperExceptionPage();* который выводит подробные сообщения об ошибках. Подобные сообщения нежелательны и могут раскрывать некоторые чувствительные данные, когда приложение уже развернуто на сервере и с ним могут работать пользователи, поэтому они подобные сообщения по умолчанию выводятся только в состоянии разработки.

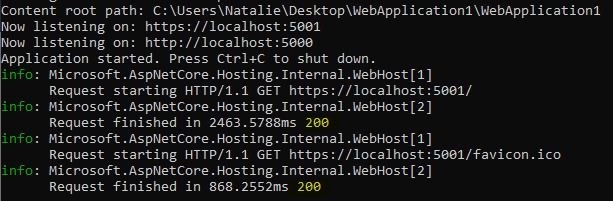
* 1. **Следующий шаг – run()**

Метод Run представляет собой простейший способ для добавления компонентов middleware в конвейер. Однако компоненты, определенные через метод Run, не вызывают никакие другие компоненты и дальше обработку запроса не передают. Он принимает в качестве параметра контекст запроса HttpContext.

Данный метод определяет один единственный делегат запроса, который обрабатывает все запросы к приложению. Причем так как данный метод не передает обработку запроса далее по конвейеру, то его следует помещать в самом конце. До него же могут быть помещены другие методы. В итоге при запуске проекта по типу Empty мы увидим в браузере строку "Hello World!".

Проект мб запущен в 2 режимах:

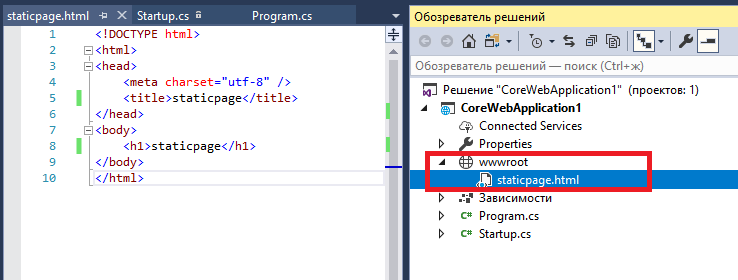
* CoreWebApp – запускаем как консольное приложение – как хостинг?
  + Стартует как консольное приложение
  + Выбает протокол, чтобы отследить какие запросы обраб.



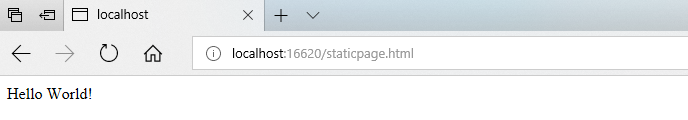
* IIS Express – через ИИС

1. ASP.NET CORE: работа со статическими файлами, добавление заголовков, стартовые страницы, файлы для скачивания, вывод в журнал. Пример.

***Wwwroot*** – папка для статич.контекта (html, css, js); стат.контект не отображ.по умолч.



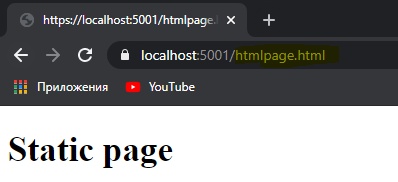
Если перейдем в браузере localhost:16620/staticpage.html – она его будет искать в wwwroot найдет и выдаст HelloWorld



**Для отображения статического контента** необх.подключить NuGet пакеты; при создании net core проекта автоматически подключается Micrisoft.AspNetCore.All в котором есть все

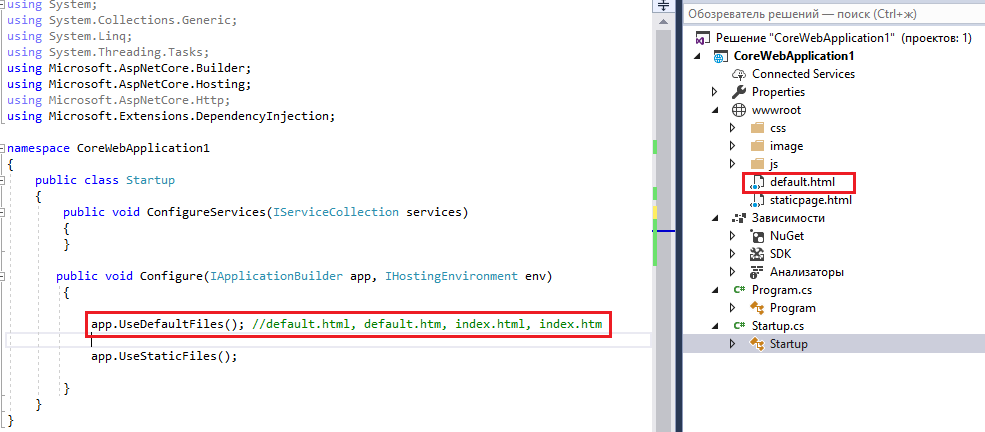
Для того, чтобы указать, что мы хотим использовать статич.файл => Startup.cs => в методе Configure пишем только 1 строку => app.UseStaticFile() – оно работало по умолч., а это более строгое определение и вызывает именно статический файл.;

Делаем localhost:16620/staticpage.html

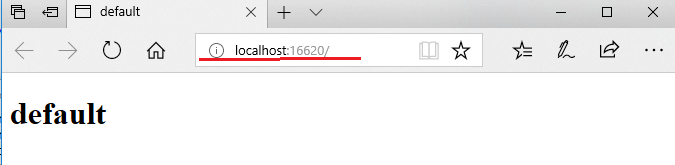
  
Также можно добавить картинку в wwroot и вызвать в html странице:

<img src=”название\_картинки” width=”500”, height = “500”/>

Можно **создать стартовые страницы** по умолчанию. Для этого исп. app.UseDefaultFiles(), кот.позвол устан. имена дефолтных файлов(в этом случае – dafault.html/.htm, index.html/.htm) – именно в таком порядке будут отыскиватсья стартовые файлы в wwwroot

****

Тогда по запросу localhost:16620 выведется содержимое дефолтного файла

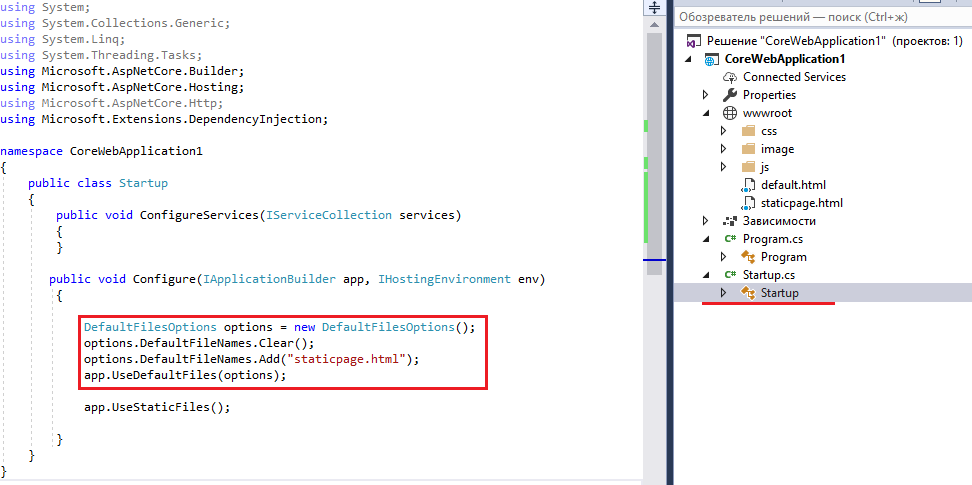
****

Также можно самим составить список дефолтных файлов:

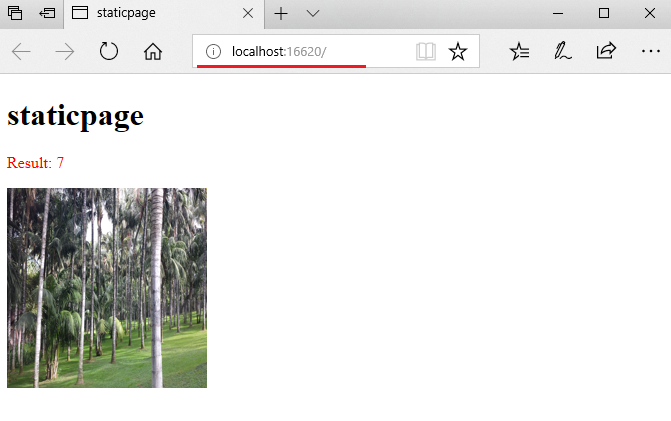
\*создаем объект

\*чистим старый список

\*добавляем свой элемент

****

Запускаем приложение:

****

**Файлы для скачивания**

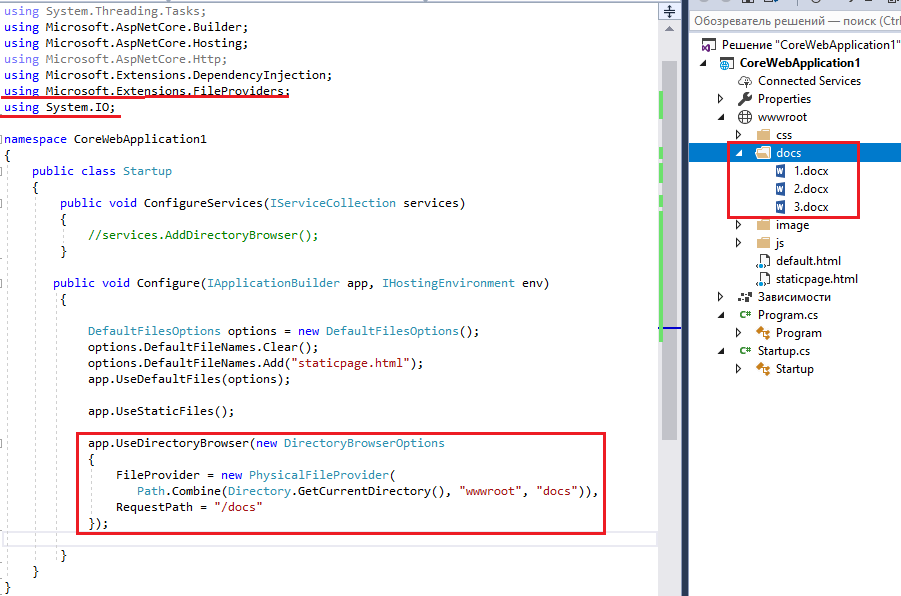
Необходимо добавить нужные namespacы:

* Using System.IO
* Using Microsoft.Extensions.FileProviders

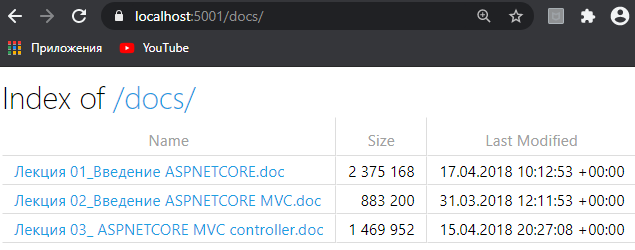
Пишем следующий код и указываем где нах.те файлы, к кот мы хотим дать юзерам доступ

\*указали что папка для скачивания – wwroot/docs

\* указали, что в uri надо указать docs чтобы добраться к этим файлам

  
В папку docs мы закинули пару документов и по запросу выдается:

(если нажмем на файл – он скачается)



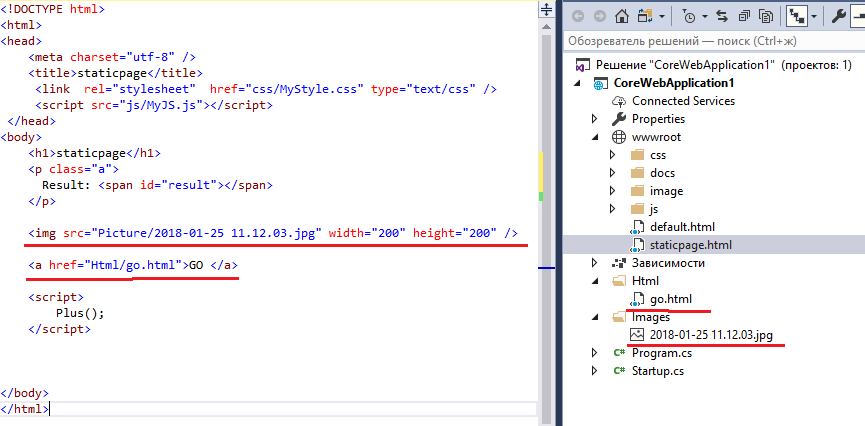
Кроме wwwroot мы можем назначить и другие директории **для хранения стат.файлов**

\* указ отдельно 2 папки

\* указали по какому uri будет доступ к статическому контенту (html/picture)

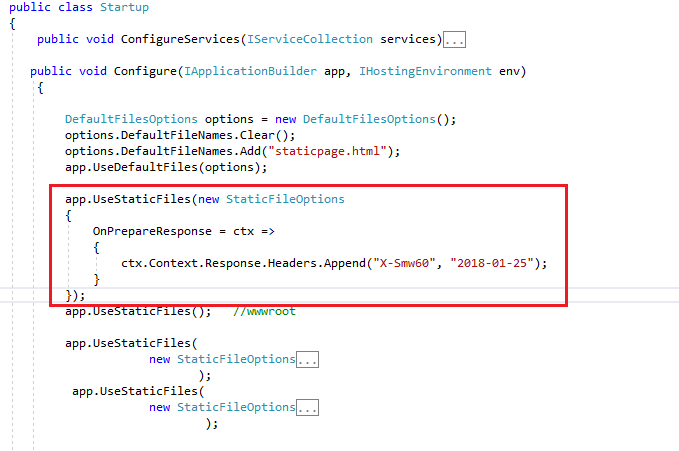
****

дальше в нашем стат.файле в wwwroot м.исп-ть эти файлы в body:

****

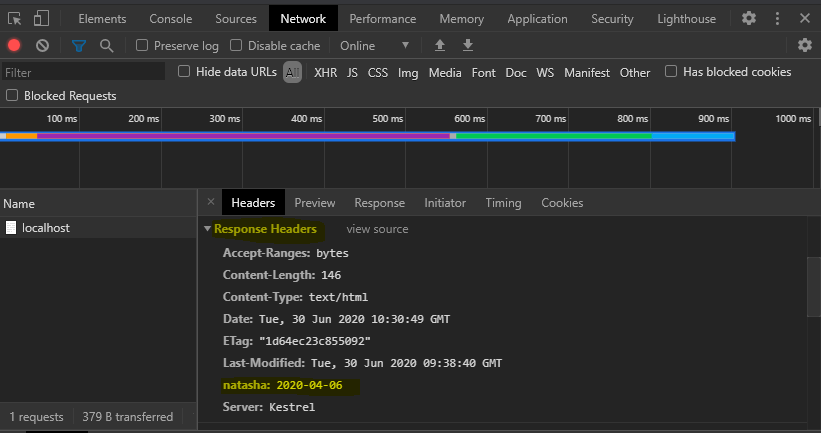
Можем **добавить http-заголовок**, при этом UseStaticFiles должен быть первым

\* через стрелочную функцию добавляем заголовок и значение

****

Чтобы проверить заголовок:

\* необходимо сделать запрос еще раз, чтобы появился «localhost» во вкладке network



Логгер включается для того, чтобы использовать какую-то трассировку в нашем прилож.

1. ASP.NET CORE: MVC, настройка MVC и маршрутизатора, применение атрибута Route для маршрутизации. Пример.

**MVC** (Model View Controller) – шаблон / архитектурный паттерн; включает 3 компоненты:

* Модель – используемые в прилож данные + логика, кот связана непосредственно с д-ми;
* Представление – отображение модели, визуальная часть, интерфейс юзера.
* Контроллер – обработка запросов, координация взаимодействия модели и представления.
* Маршрутизатор – выбор контроллера и действия

Фреймворк ASP.NET Core MVC работает поверх платформы ASP.NET Core, и предназн.для того, чтобы упросить создание прилож.

Поступает запрос – запрос поступает на маршрутизатор – создается контроллер – контроллер формирует респонс – контроллер отрабатывает, использует модель и формируется представление, которое формирует респонс – респонс уходит на сторону клиента.

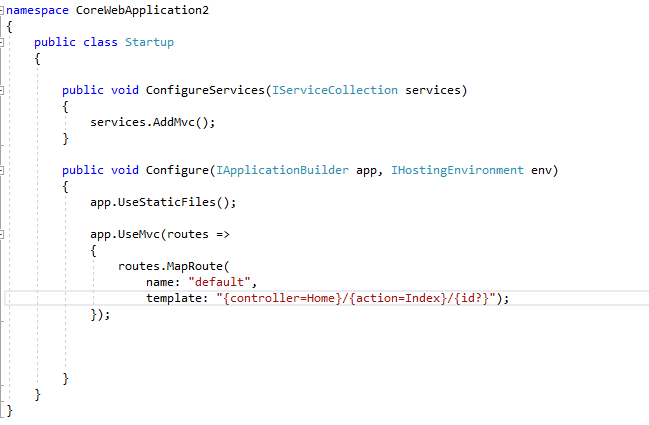


Модель – независ.компонент – любые изменения контроллера или представления никак не влияют на модель. Контроллер и представл.явл.относ.независ.компонентами. Из представления м.обращаться к опред.контроллеру, а из контроллера генерить представления, но их также м. измен независ друг от друга.

*Создание веб-приложения:*

Visual C# => Веб => веб-приложение ASP.NET Core => MVC

Смотрим Startup.cs:

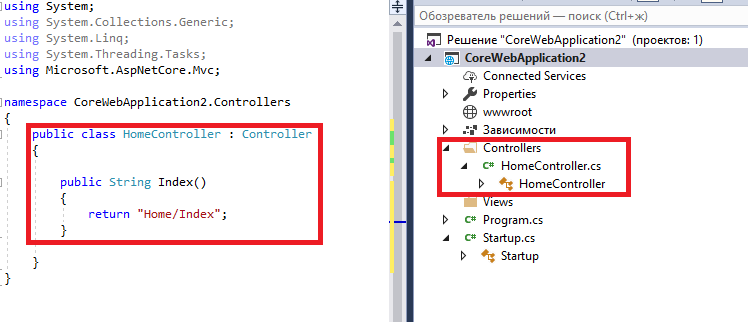
****

\* для того, чтобы исп.MVC используем метод AddMVC(), т.е. подключить сервис MVC

\* В методе Configure подключаем статические файлы, а также используем MVC и здесь же определяем маршрутизатор, в котором используем MapRoute (при написании шаблона маршрута указываем контроллер и его дефолное значение, а также дефолтное зрначение action.

Разрабатываем контроллеры

* У нас это класс, который наследует от Controller
* В .net 2 разных базовых класса для контроллеров mvc и web-api, а также разные маршрутизаторы
* В asp .net core все контроллеры имеют общие контроллеры, а также общие маршрутизаторы (те же для mvc и web-api)

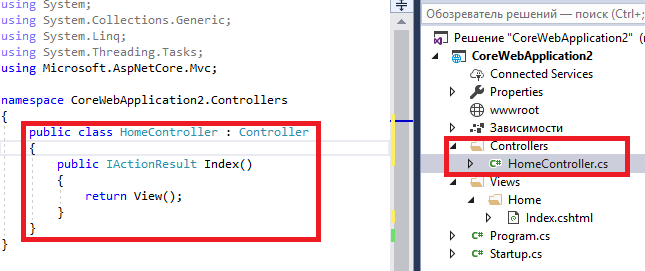
****

Обработка запроса

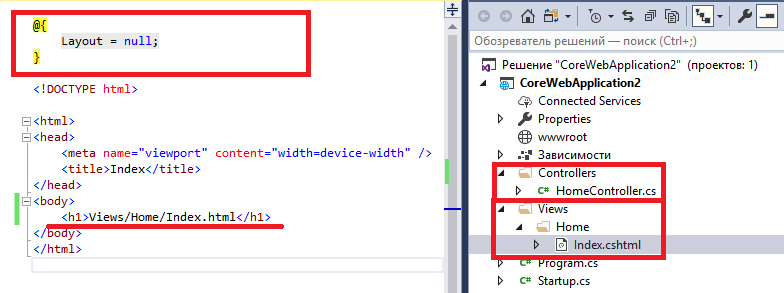
****

**Также можно использовать view**

\* здесь вызывается view по умолч., имя кот.совпад.с именем action

****

Содержимое view:

****

Результат обработки:

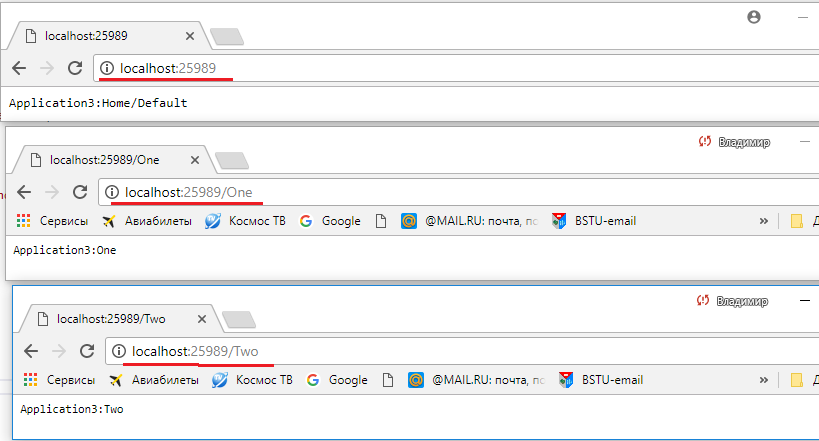
****

Сопоставление url с названием контроллера и его метода происходит благодаря системе маршрутизации. В классе Startup в методе Configure() м.найти определение 1 маршрута – по умолч.

Там доб.1 маршрут с именем default и шаблон {controller=Home}/{action=Index}/{id?}. Данный шаблон устанавл.3сегментную структуру строки запроса: *controller/action/id.* Т.е.в начале идет название контроллера, атем название действия и затем мб необяз.пар-р id.

**Атрибут Route**  - если мы хотим делать маршрутизацию без таблиц маршрутизации, то мы просто указываем маршруты явно для к.action с помощью атрибута Route

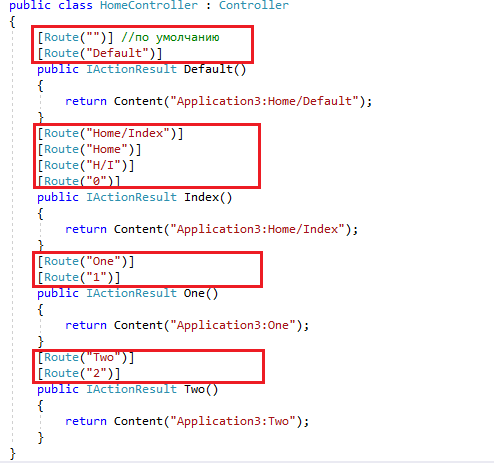
****

****

Также с помощью атрибута Route можно указывать **множественные маршруты.**

Т.е.при переходе по Home/Index, Home, H/I, 0 результат будет 1 и тем же.

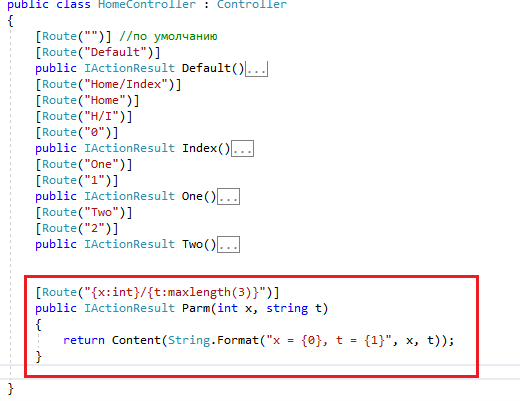
Т.е.к 1 actionу мы м.прописать много маршрутов

****

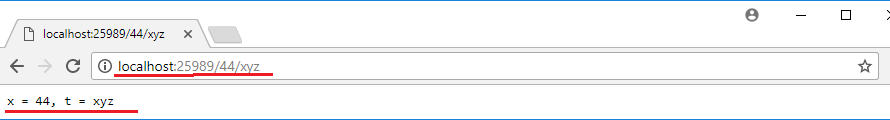
**Обработка сложных маршрутов:**

Используем параметризованный uri и обрабатываем значения как параметры.

Описывается маршрут, кот.обяз.д.сдерж.целочисл.знач, и строку макс из 3 символов

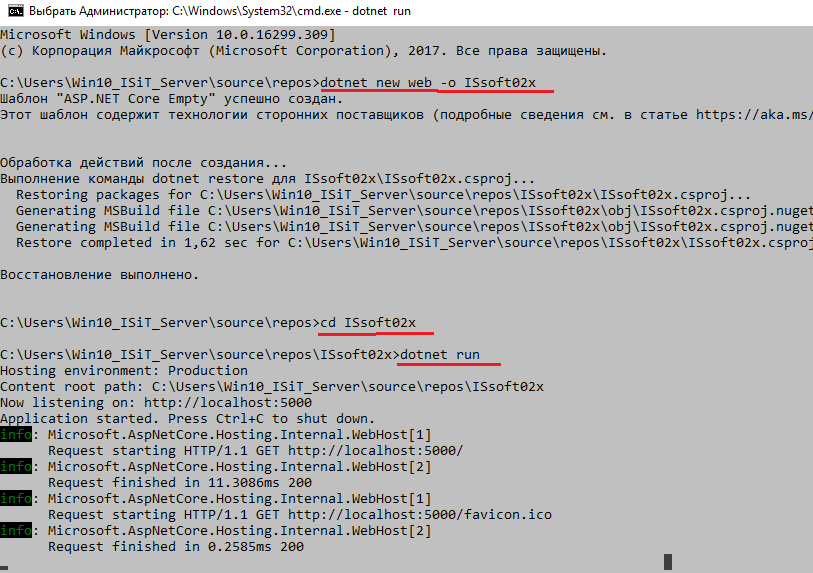
****

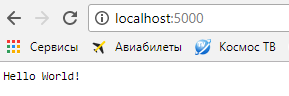
Обработка такого маршрута:

****

**Разработка web-приложения с помощью CLI (ком.строка)**

\* dotnet new wev –o ISsoft02x – создаст готовое веб-приложение

****

****

М. изменить приложение и также запустить через ком.строку.

**ViewBag** – объект, который позвол опред.любую пер-ную и передать ей нек.знач, а затем в представл извлечь это знач.

----------------------------------------------

Возможно, сопоставление по умолч.не всегда удобно. Например, у нас есть класс в папке Controllers, но м не хотим, чтобы он мог обрабатывать запрос и исп-ся как контроллер.

[NonController] - указать, что этот класс не явл.контроллером (пишется над классом)

[NonAction] – пишем над методом, кот.не д.рассм как действие контроллера

[ActionName] – позволяет задать другое имя для действия => по пред имени не будет работать

1. ASP.NET CORE: MVC-котроллер, действия (action) контроллера, контекст контроллера, поддержка сессии, результат работы действия, внедрение зависимостей. Пример.

**MVC Controller -** компонент приложения разработанного в соответствии с паттерном MVC. В одном приложении может быть несколько контроллеров.

**MVC Controller –** программный класс, производный от одного из абстрактных классов: **Controller** или **ControllerBase**.

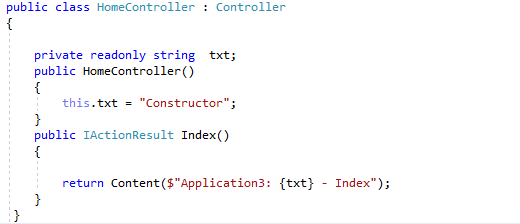
**ControllerBase** не поддерживает view**. Controller** поддерживает view

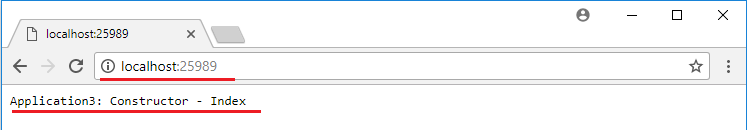
По умолч.HomeController имеет 4 метода, кот.м.назвать **действиями**.

**Действия контроллера** – публичные методы, кот.м.сопоставл.с запросами.

М. применять **конструкторы** и применять стандартные методы

\*Content – выводить д-е в редима text/plain

****

****

Существуют особенности по поиску контроллеров в core.

Существует redlection-поиск контроллера. По каким признакам?

* неабстрактный public-класс;
* не generic;
* не вложен в другие типы;
* не декорирован [NonController];
* наследуется от Control, либо имя класса имеет суффикс Control.

**М.создать контроллер без суффикса Controller**

**контекст контроллера** – св-во, в кот.входят другие св-ва кот.м.нас заинетресовать

ModelState – объект, кот предназнач.для валидации д-х.

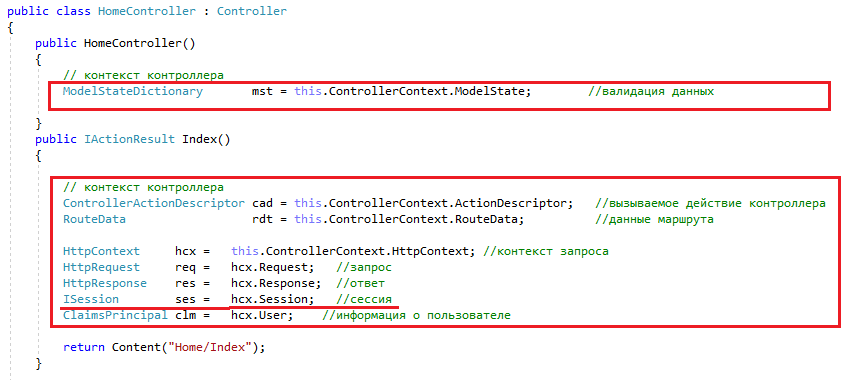
В любом mvc приложении есть 3 типа моделей – для view, для action и для данных.

Модель для экшн образуется из набора пар-ров, кот приходят в экшн.

ModelState и предназначена для валидации д-х, кот приходят в кач-ве пар-ров.

HttpCOntext искапсулирует всю информацию о запросе и состоит из свойств:

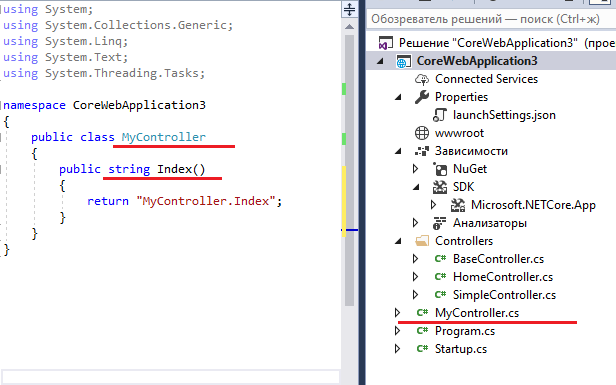
* request – содерж.инфу о тек.запросе
* response – управляет ответом
* user – тек.пользователь, кот обращ.к прилож
* session – объек для работы с сессиями

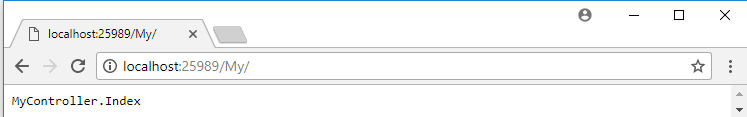
****

POCO-controller - controller не наследник класса Controller; имя с суффиксом Controller;

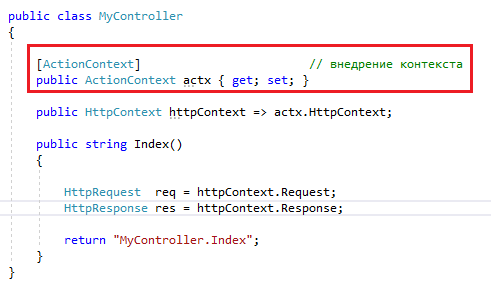
**может находиться в отдельной сборке** **(не получилось);**

**пробовал в отдельную сборку подключать Microsoft.AspnetCore.Mvc – не помогло.**

****

****

Здесь можем внедрить контекст, т.к.тот контекст кот.дб прийти от контроллера не пришел, поэтому можем внедрить свой

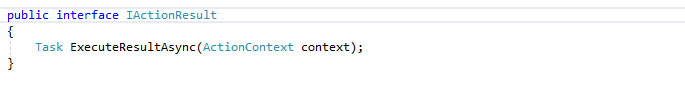
****

**MVC Controller** по умолч.не поддерживает сессию, адля поддержки объекта **Session** исп.AddSession.

**Требования к action (методам контроллера)**

* нестатический публичный метод котроллера;
* не является generic;
* не помечен атрибутом [NonAction];
* перегруженные методы могут быть только отмеченные атрибутами [Action], [ActionName], [AcceptVerbs], [HttpGet], [HttpPost]; - т.е.если мы помечаем данными атрибутами методы, то они могут быть перегружены
* ref и out–параметры в действиях не допустимы;
* возвращает IActionResult – преобразует ответ к этому типу (возвращаем стринг - преобразуется)

интерфейс IActionResult.- асинхронный вызов

****

**Результат действия:**

Объект типа IActionResult предназначен для генерации рез-та д-вий.

Интерфейс IActionResult нах.в пр-ве Microsoft.AspNetCore.Mvc и опред.1 метод:

*Public interface IActionResult {*

*Task ExecuteResultAsync (actionContext context); }*

Этот метод принимает контекст дейсвтия и выполняет генерацию результата.

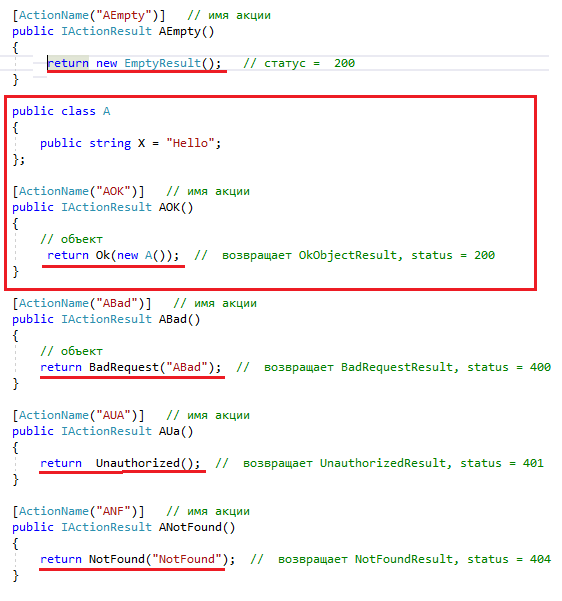
Этот интерфейс затем реализуется абстр.баз.классом ActionResult. И этот класс добавляет метод, который выполняется асинзронно.

Если мы хотим создать свой класс результата действий, то как раз можем наследовать его от ActionResult либо реализовать интерфейс IActionResult.

**возврат кодов**

демонстрируем стандартные методы контроллера

* ContentResult – пишет указанный контент напрямую в ответ в виде строки
* EmptyResult – отправляет пустой ответ в виде статусного кода 200
* NoContentResult – во многом похож на EmptyResult, также отправляет пустой ответ, только в виде статусного кода 204
* **FileResult**: является базовым классом для всех объектов, которые пишут набор байтов в выходной поток. Предназначен для отправки файлов
* **FileContentResult**: класс, производный от FileResult, пишет в ответ массив байтов
* **VirtualFileResult**: также производный от FileResult класс, пишет в ответ файл, находящийся по заданному пути
* **PhysicalFileResult**: также производный от FileResult класс, пишет в ответ файл, находящийся по заданному пути. Только в отличие от предыдущего класса использует физический путь, а не виртуальный.
* **FileStreamResult**: класс, производный от FileResult, пишет бинарный поток в выходной ответ
* **ObjectResult**: возвращает произвольный объект, как правило, применяется в качестве базового класса для других классов результатов.
* **StatusCodeResult**: результат действия, который возвращает клиенту определенный статусный код HTTP
* **UnauthorizedResult**: класс, производный от StatusCodeResult. Возвращает клиенту ответ в виде статусного кода HTTP 401, указывая, что пользователь не прошел авторизацию и не имеет прав доступа к запрошенному ресурсу.
* NotFoundResult: производный от StatusCodeResult. Возвращает клиенту ответ в виде статусного кода HTTP 404, указывая, что запрошенный ресурс не найден
* BadRequestResult: производный от StatusCodeResult. Возвращает статусный код 400, тем самым указывая, что запрос некорректен
* OkResult: производный от StatusCodeResult. Возвращает статусный код 200, который уведомляет об успешном выполнении запроса
* JsonResult: возвращает в качестве ответа объект или набор объектов в формате JSON
* RedirectResult: перенаправляет пользователя по другому адресу URL, возвращая статусный код 302 для временной переадресации или код 301 для постоянной переадресации зависимости от того, установлен ли флаг Permanent.
* RedirectToRouteResult: класс работает подобно RedirectResult, но перенаправляет пользователя по определенному адресу URL, указанному через параметры маршрута
* RedirectToActionResult: выполняет переадресацию на определенный метод контроллера

****

**Переадресация**

В ASP.NET Core MVC для создания переадресации используются классы RedirectResult, LocalRedirectResult, RedirectToActionResult и RedirectToRouteResult. Но в зависимости от типа переадресации их применение будет отличаться.

Протокол HTTP поддерживает **два типа переадресации**:

* постоянная переадресация. При постоянной переадресации сервер будет отправлять браузеру статусный код 301. При данном типе переадресации предполагается, что запрашиваемый документ окончательно перемещен в другое место. И после получения статусного кода 301 браузер может автоматически настраивать запросы на новый ресурс, даже если старый ресурс со временем перестанет применять переадресацию. Поэтому данный способ можно использовать, если вы полностью уверены, что документ на старое место уже не возвратится..
* временная переадресация. При временной переадресации сервер будет отправлять браузеру статусный код 302. При этом считается, что запрашиваемый документ временно перемещен на другую страницу.

**Redirect –** временная переадресация

**RedirectPermanent** – постоянная переадресация

**LocalRedirect –** временная переадресация по локальным адресам

**LocalRedirectPermanent** – постоянная переадресация по локальным адресам

**RedirectToAction –** временная переадресация на определенный метод контроллера

**RedirectToActionPermanent –** постоянная переадресация на определенный метод контроллера

**RedirectToRoute –** временная переадресация с исп-нием маршрутов

**RedirectToRoutePermanent –** постоянная переадресация с исп-нием маршрутов

**Внедрение зависимостей**

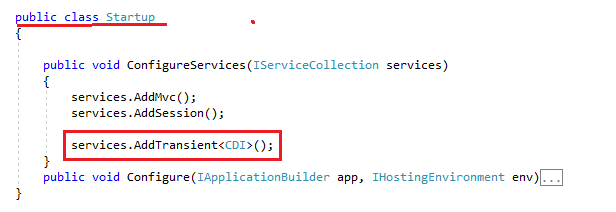
**Dependency injection (DI)** или внедрение зависимостей представляет механизм, который позволяет сделать взаимодействующие в приложении объекты слабосвязанными. Такие объекты связаны между собой через абстракции, например, через интерфейсы, что делает всю систему более гибкой, более адаптируемой и расширяемой.

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.

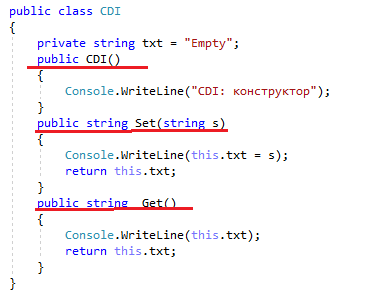
Этот механизм уже встроен а асп.нет кор.

Для внедрения зависимостей используется сервис – AddTransient (использовали для скопа это понятие, в нашем случае мы хотим добавить какой-то объект, в нашем случае указываем имя класса)

Т.е.используем разработанный класс для внедрения

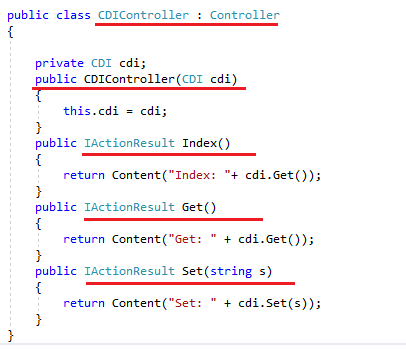
****

Наш разработанный класс

****

Далее можем внедрять наш объект в контроллере через конструктор.

Зпускаем используя пару интерфейс и объект.

****

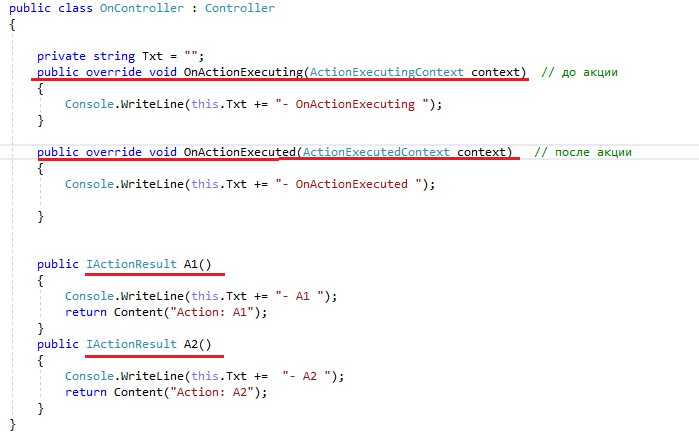
1. ASP.NET CORE: события OnAction, атрибуты HttpGet, HttpPost, …, AcceptVerb, принцип передачи параметров в метод действия.

Фильтры действий позволяют нам проконтролировать входной контекст запроса при доступе к действию, а также выполнить определенные действия по завершению работы метода действий. Например, мы можем изменить выходной результат метода.

OnAction – специальный обработчики, которые могут быть исп-ны для контроллера.

М.сделать спец.обработчики, которые будут работать до акции и после акции.

Если такие обработчики установлены в контролере – они будут срабатывать до к.акции и после.

****

Может потребоваться указать HTTP-метод, при помощи которого предполагается обрабатывать запрос. Для этого есть универсальный атрибус [AcceptVerbs], либо его алиасы [HttpGet], [HttpPost]…..

**Передача параметров**

Передавать значения для пар-ров м.разл спомобами.

При отправке get запроса значения передаются через строку запроса.

Стандартный get запрос принимает следующую форму:

*название\_ресурса?параметр1=значение1&параметр2=значение2*

Если не исп.пар-ры в сроке запроса, то для пар-ров будут передаваться знач.по умолч., кот будут раб. Если через строку запроса не передаются никакие пар-ры.

public string Square(int a = 3, int h = 10)

{

    double s = a \* h / 2;

    return $"Площадь треугольника с основанием {a} и высотой {h} равна {s}";

}

Также можно передавать массив в качестве пар-ра. В таком случае строка запроса будет выглядеть:

*Sum?nums=1&nums=2&nums=3*

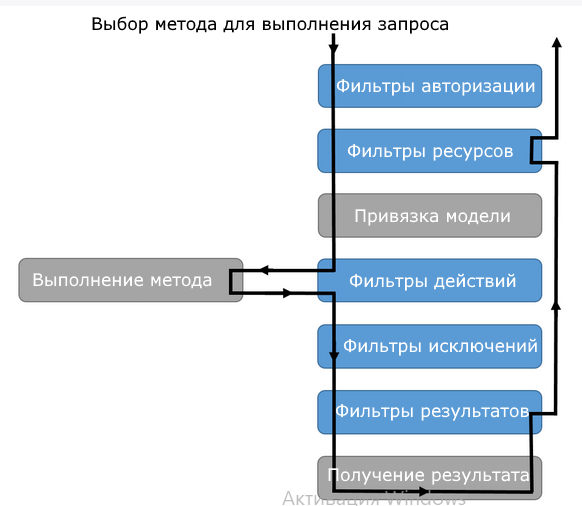
Кроме get запросов также широко применяются post запросы. Как правило, такие запросы отправляются с помощью форм на веб странице. ПРИМЕР

1. ASP.NET CORE: Фильтры: Action Filter, Result Filter, Authorization Filter, Resource Filter, Exception Filetr, пользовательские фильтры действий.

Фильтры позволяют выполнять некоторые действия до или после определенной стадии обработки запроса. В ASP.NET Core имеются следующие типы фильтров:

* Фильтры действий: применяется только к действиям контроллера, запускается после фильтра ресурсов как до, так и после выполнения метода контроллера
* Фильтры результатов действий: фильтр применяется к результатам методов контроллера и страниц Razor Pages, выполняется как до, так и после получения результата
* Фильтры авторизации: определяют, авторизован ли пользователь для выполнения текущего запроса. Если пользователь не авторизованн для доступа к ресурсу, то фильтр завершает обработку запроса.
* Фильтры ресурсов: выполняются после фильтров авторизации. Его метод OnResourceExecuting() выполняется до всех остальных фильтров и до привязки модели, а его метод OnResourceExecuted() выполняется после всех остальных фильтров
* Фильтры исключений: определяют действия в отношении необработанных исключений
* Фильтры RazorPages: применяется только к страницам RazorPages, выполняются перед и после обработки запроса страницей Razor Page

Вместе все эти типы фильтров образуют конвейер фильтров (filter pipeline), который встроен в процесс обработки запроса в MVC и который начинает выполняется после того, как инфраструктура MVC выбрала метод контроллера для обработки запроса. На разных этапах обработки запроса в этом конвейере вызывается соответствующий фильтр:



Несмотря на то, что конвейер фильтров образуют пять разных типов фильтров, которые вызываются на разных этапах и имеют свою строго задачу, тем не менее все они имеют общие моменты реализации. Так, все фильтры поддерживают два способа реализации: синхронную и асинхронную. В зависимости от выбранного способа класс фильтра будет реализовать тот или иной интерфейс.

1. ASP.NET CORE: MVC-представление, обнаружение представления, жизненный цикл представления, методы редеринга представления в web-страницу (методы View контроллера), способы передачи данных из котроллера в представление, строготипизированные представления, директива @model. Пример.

**View** – файл с расширением chtml, кот сод html, css, js и razor-конструкции, кот позвол переходить от кода html к коду на языку c#

Папка Views/Home – папка где хранятся представления и соотв.имени контроллера.

Папка Views/Shared – папка в кот хранятся общие представл.для всех контроллеров. По умолч.сост из

* Layout.chtml – исп.в кач-ве мастер-страницы
* Error.chtml – исп для отображения ошибок
* \_ValidationScripsPartial.chtml – частичн.представл, кот подключ.скрипты валидации формы

В корне каталога View м.найти 2 файла, кот. сод.код,, кот авто добавл ко всем представлениям:

* \_ViewImports.chtml – устан.нек.общие пр-ва имен для всех вьюх
* \_ViewStart.chtml – устан.общую мастер-страницу

Добавить свое представление => Add => New Item => Razor View

Razor View Engine – движок представления, компонент sqp.net core mvc-фреймворка, предназн.для генер содержимого response на осн.содержимого chtml-файла.

Также движок исп для обработки кода, кот.содержит как эл-ты хтмл, так и конструкции c#.

Razor – движок по умолчанию. Можно исп.сторонние движки либо создать свой.

Цель движка – определить переход от хтмл к с шарп.

Синтаксист прост – все его конструкции предваряются символом @, после которого происходит переход к коду c#.

Пример:

* <p>Дата: @DateTime.Now.ToLongDateString()</p>
* <p>@(20 + 30)</p>

Если создаем html код, в кот.@ надо исп не как символ синтаксиса, то его надо продублировать для того, чтобы отобразить:

<p>@@DateTime.Now =@DateTime.Now.ToLongDateString()</p>

Результат:



Блоки кода м.им неск выраж. Блок кода заключ в фиг.скобки, в к.выраж.завершается «;». В таких блоках м.опред.обычные пер-ные и затем их исп.в представлении.

*@{*

*string head = "Привет мир!!!";*

*head = head + " Добро пожаловать на сайт!";*

*}*

*<h3>@head</h3>*

Весь код в пределах блока расценивается как код c#. Однако с помощью конструкции @: мы можем в блоке кода выводить на веб-страницу текст:

*@{*

*string head = "Hello world";*

*@: <b>Привет мир!</b>*

*head = head + "!!";*

*}*

*<p>@head</p>*

Результат:



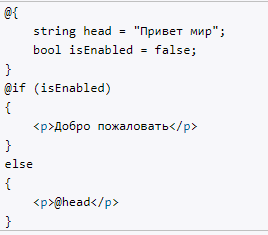
Если надо вывести текст без каких-то html-элементов => исп.снипет <text>

Комменты распола между символами @\*\*@

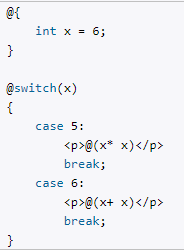
Chtml-фай компилируется в сборку, которая исп.для генерации response-содержимого.

Также м.исп.усл.констрункции:

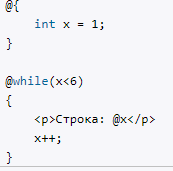
* If



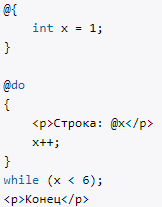
* Switch



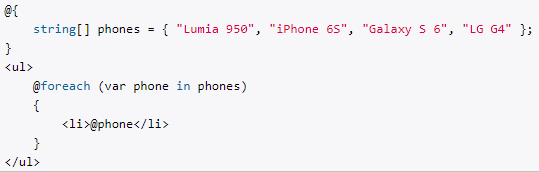
* Цикл for



* Цикл do..while



* Цикл foreach



* Конструкция try..catch..finally

Если в блоке try выбрасывается исключение, то выполняется блок catch. И в любом случае в конце блока try и catch выполняется блок finaly.



За работу с представ.отвеч объект **ViewResult**. Он производит рендеринг представления в веб страницу и возвращ.ее в виде ответа клиенту. Чтобы возвратить объект ViewResult исп метод View:

*public class HomeController : Controller*

*{*

*public IActionResult Index()*

*{*

*return View();*

*}*

*}*

Есть chtml файл, который обраб.enginом, генерируется класс, который помечается в отд.сборку и этот класс занимается генерированием response.



Вызов метода View возвращает объект ViewResult. Затем уже ViewResult производит рендеринг опред представл в ответ. Т.е. контроллер только готовит д-е и выбирает какое представление надо возвратить в кач-ве объекта ViewResult, а объект ViewResult обращается к движку представлений для рендеринга представления в вых.ответ. По умолч. Контроллер производит **поиск представления** в проекте по след.путям:

*/Views/Имя\_контроллера/Имя\_представления.cshtml*

*/Views/Shared/Имя\_представления.cshtml*

Согласно настройкам по умолч., если название представления не указано явно, то в кач-ве представления будет исп.то, имя кот.совпадает с именем действия контроллера. Например, по примеру выше действие Index будет производить поиск представления Index.chtml в папке Views/Home/

Метод View() имеет **4 перегруженные версии**:

* **View():** для генерации ответа используется представление, которое по имени совпадает с вызывающим методом
* **View(string viewName):** в метод передается имя представления, что позволяет переопределить используемое по умолчанию представление

Т.е.мы можем переопределить представление. Если представление нах в той же папке, кот предназначена для данного контроллера, то в метод View()

*public IActionResult Index()*

*{*

*return View("About");*

*}*

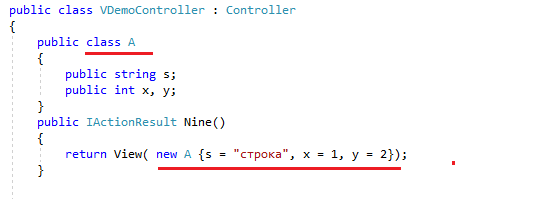
В этом случае метод Index будет использовать представлеие Views/Home/About.chtml

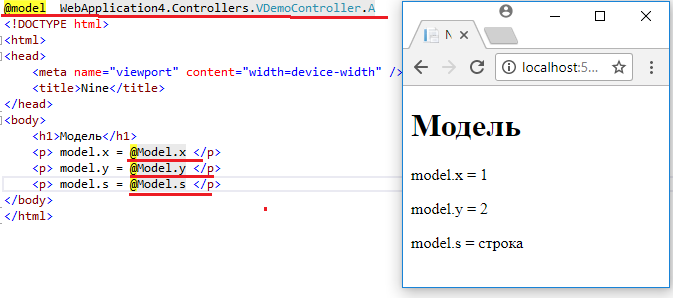
Если представление нах.в жругой папке, то надо передать полный путь к представлению:

*"~/Views/Some/Index.cshtml"*

* **View(object model):** передает в представление данные в виде объекта model
* **View(string viewName, object model):** переопределяет имя представления и передает в него данные в виде объекта model

@model – если передаем какие-то параметры при вызове вью – это рассматривается как модель внутри вью и мы м.исп.переменную Model с большой буквы для доступа к этой информации.

****

****

**Передача данных из контроллера в представление:**

* ***ViewData***

VIewData представляет словарь из пар ключ-значение.

*public IActionResult Index()*

*{*

*ViewData["Message"] = "Hello ASP.NET Core";*

*return View();*

*}*

Здесь динамически опред.объект с ключом «Message» и значением «Hello ASP.NET Core»

После этого мы м.его исп-ть в представлении. Но, при это не обяз.устан.значение в контроллере, а м. именно в представлении.

*@{*

*ViewData["Title"] = "Index";*

*}*

*<h2>@ViewData["Title"].</h2>*

*<h3>@ViewData["Message"]</h3>*

*<p>Random text.</p>*

* ***ViewBag***

ViewBag во многом подобен ViewData. Он позволяет определить различные свойства и присвоить им любое значение.

*public IActionResult Index()*

*{*

*ViewBag.Message = "Hello ASP.NET Core";*

*return View();*

*}*

Второй пример можно переписать так:

*@{*

*ViewBag.Title = "Index";*

*}*

*<h2>@ViewBag.Title.</h2>*

*<h3>@ViewBag.Message</h3>*

*<p>Random text.</p>*

* ***Модель представления***

Модель представления является во многих случаях более предпочтительным способом для передачи данных в представление. Для передачи данных в представление используется одна из версий метода View:

*public IActionResult Index()*

*{*

*List<string> countries = new List<string> { "Бразилия", "Аргентина", "Уругвай", "Чили" };*

*return View(countries);*

*}*

В метод View передается список, поэтому моделью представления Index.cshtml будет тип List<string> (либо IEnumerable<string>). И теперь в представлении мы можем написать так:

*@model List<string>*

*@{*

*ViewBag.Title = "Index";*

*}*

*<h3>В списке @Model.Count элемента</h3>*

*@foreach(string country in Model)*

*{*

*<p>@country</p>*

*}*

В самом начале представления с помощью директивы @model устанавливается модель представления. Тип модели должен совпадать с типом объекта, который передается в метод View() в контроллере.

Установка модели указывает, что объект Model теперь будет представлять объект List<string> или список. И мы сможем использовать Model в качестве списка.

//в рамках представления – одно и то же.

Model = ViewBag.Model = ViewData[“Model”]

**Строго типизированные представления** – представления, для которых определена модель.

1. ASP.NET CORE: MVC-представление, директивы @using, @function, @inherits. Пример.

Директива @functions позвол.опред.ф-ции, которые могут применяться в представлении. Например:

@functions

{

    public int GetFactorial(int n)

    {

        int result = 1;

        for (int i = 1; i <= n; i++)

            result \*= i;

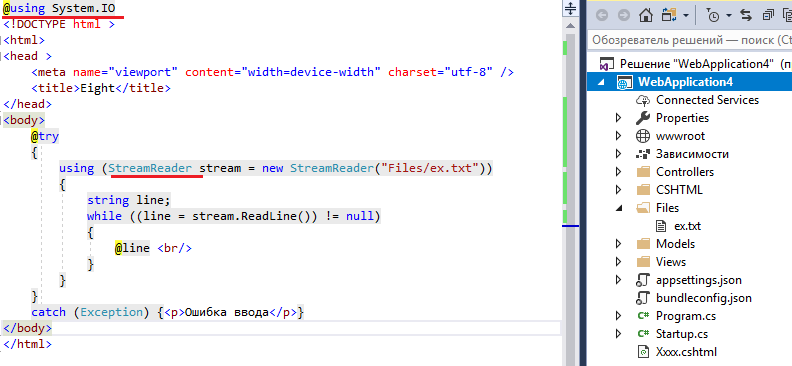
        return result;

    }

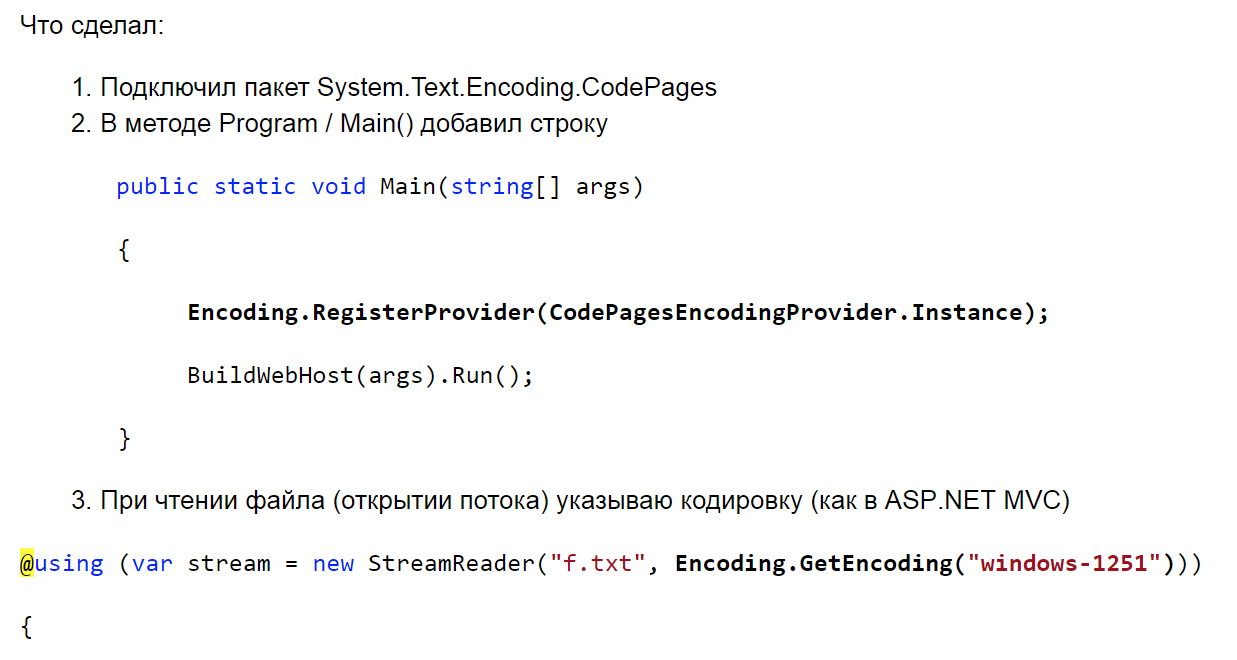
}

<div>Факториал числа 6 равен: @GetFactorial(6)</div>

@using – использовать какие-то namespace в рамках view



**Проблема русского языка**



@inherits – позволяет создать собственный баз.класс для представления; базовый класс дб производным от RazorPage или RazorPage<ModelType>

*Пример с объяснениями в коде*

1. ASP.NET CORE: MVC-представление, директивы @addTagHelper, @removeTagHelper. Пример.

Проект ASP.NET MVC Core уже по умолчанию подключает функциональность tag-хелперов в представления с помощью установки в файле **\_ViewImports.cshtml** следующей директивы:

|  |  |
| --- | --- |
| 1 | @addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers |

Первый параметр директивы указывает на tag-хелперы, которые будут доступны во всех представлениях из папки Views, а второй параметр определяет библиотеку хелперов. В данном случае директива использует синтаксис подстановок - знак звездочки ("\*") означает, что все хелперы из библиотеки **Microsoft.AspNetCore.Mvc.TagHelpers**.

Если вдруг у нас не окажется подобной директивы, то ее добавление в представления позволяет использовать все встроенные tag-хелперы.

По умолчанию эта директива определяется в файле *\_ViewImports.cshtml*, который находится в папке *Views*. Однако мы можем конкретизировать применение хелперов к определенной группе представлений. Например, если у нас есть каталог *Views/Home* - специально для представлений для контроллера HomeController, и мы хотим применить только к ним определенные хелперы. В этом случае мы можем добавить файл *\_ViewImports.cshtml* непосредственно в этот каталог. И любой tag-хелпер, добавленный директивой @addTagHelper из файла *Views/Home/\_ViewImports.cshtml*, будет применяться только к представлениям из каталога *Views/Home*.

Еще одна директива removeTagHelper удаляет ранее добавленные tag-хелперы. Ее применение аналогично:

|  |  |
| --- | --- |
| 1 | @removeTagHelper "\*, Microsoft.AspNetCore.Mvc.TagHelpers" |

Данная директива может быть полезной, если мы, например, захотим ограничить применение хелперов в каком-то одном представлении или группе представлений. Эта директива также определяется в файле *\_ViewImports.cshtml*.

1. ASP.NET CORE: MVC-представление, применение компоновки (Layout) представления, компоновка по умолчанию (\_ViewStart), применение секций @RenderSection, @RenderBody. Пример.

**Макет/компоновка** – представление, кот.предназн.для макетирования страниц.

Макетирование страниц осуществляется с помощью мех.секций.

Любая страница в свою очередь тоже мб компоновкой, т.е.допуск.вложенность.

\_ViewStart.cshtml;

* действует, если вызов осуществляется через ViewResult;
* не действует, если вызов через PartialViewResult;
* если применяется ViewStart.cshtml, то поиск в родном директории, а потом в SHARED.

По умолчанию представления уже подключают мастер-страницу за счет файла \_ViewStart.cshtml. Этот файл можно найти в проекте в папке Views. Код этого файла добавляется в самое начало кода преставлений при их запуске. При этом файлы представлений, к которым применяется \_ViewStart.cshtml, должны находиться с этим файлов в одном каталоге.

В к.представлении хранится св-вой **Layout**, которое хранит ссылку на мастер страницу

Когда будет происходить рендеринг представления, то система будет искать мастер страницу \_Layout по следующим путям:

* /Views/[Название\_контроллера]/\_Layout.cshtml
* /Views/Shared/\_Layout.cshtml

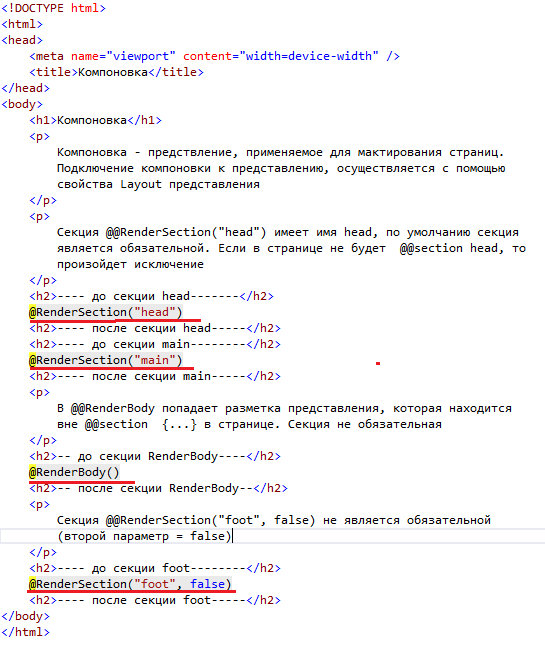
Если в обеих папках: и в /Views/[Название\_контроллера], и в /Views/Shared/ имеется файл с одинаковым именем, например, \_Layout.cshtml, то к представлению применяется файл, который находится с ним в одной папке как более приоритетный. То есть таким образом мы можем определить для представлений каждого отдельного контроллера или представлений, которые находятся в одной папке, свою отдельную мастер-страницу.

Если вдруг мы хотим глобально по всему проекту поменять мастер-страницу на другой файл, который расположен в какой-то другой папке, например, в корне каталога Views, то нам надо использовать полный путь к файлу:

* @{ Layout = "~/Views/\_Layout.cshtml"; }

**RenderSection()** – метод, кот.исп.для вставки секций.

**RenderBody()** – плейсхолдер на место кот.будут подставляться другие представления, кот исп.данную страницу. В итоге для всех представлений м.установить единый стиль оформления.



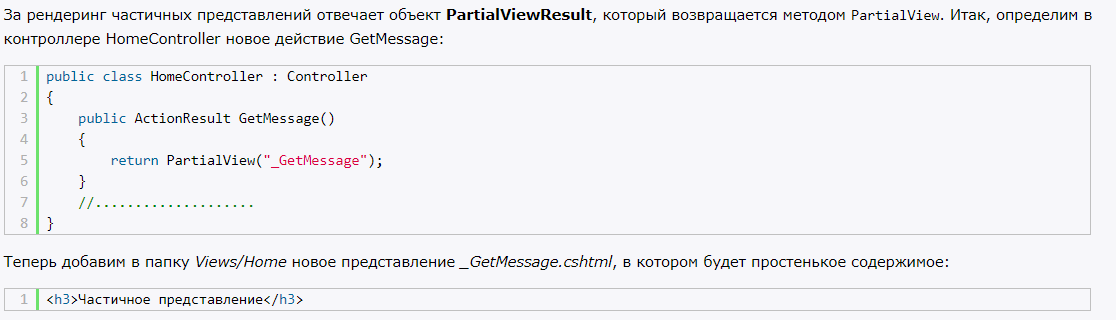
\_ViewImports.cshtml; здесь размещаются общие директивы @using и директивы, связанные с TagHelpers, размещается в родном для представлений директории или в SHARED.

1. ASP.NET CORE: MVC-представление, частичные представления (partial view). Встроенные хелперы. Пример.

**Partial view** – частичное представление; фрагмент Razor-разметки, предназначенный для повторного применения; любое представление мб частичным, но в этом случае не применяется \_ViewStart.cshtml. \_ViewImports.cshtml действует на частичные представления.

**Отличительная особенность** частичных представлений является то, что их можно встраивать в другие обычные представления. По действию похожи на секции. Могут использоваться как обычные, но чаще всего используются для рендеринга результатов AJAX-запросов.

За рендеринг частичных представлений отвечает объект PartialViewResult, который возвращается метолом PartialView.

****

****

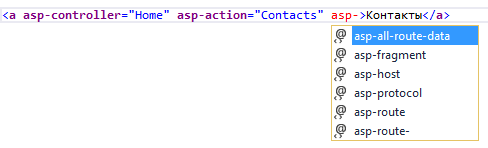
**Tag Helpers** – классы предназначенные для генерации Razor-разметки, расширяющей семантику html-разметки. В Razor: новый тег или html-тег с новыми атрибутами (при этом стандартные атрибуты могут быть сохранены).

Tag-хелперы представляют более удобный способ для генерации html-элементов, нежели обычные html-хелперы, поскольку tag-хелперы во многом выглядят как обычные html-элементы, Visual Studio имеет встроенную поддержку IntelliSense для tag-хелперов

Использовать tag-хелперы довольно просто. Например, определим в представлении следующий код:

<a asp-controller="Home" asp-action="Contacts">Контакты</a>

Внешне данный хелпер напоминает обычную ссылку - стандартный элемент html, однако это не элемент html. И если мы воспользуемся всплывающей подсказкой, то увидим, что кроме обычных для элемента <a /> он имеет ряд других:



Данный хелпер создает ссылку, для которой в качестве контроллера используется Home, а в качестве метода Contact. Такой хелпер будет интуитивно более понятным и привычным, нежели создание ссылки с помощью Html.ActionLink:

|  |  |
| --- | --- |
| 1 | @Html.ActionLink("Контакты", "Contacts", "Home") |

В то же время нам необязательно использовать именно tag-хелперы. Мы можем использовать обычные html-хелперы, если они нам более удобны.

Хотя мы можем создать любой необходимый хелпер, но в большинстве случаев нам не придется писать свои хелперы, потому что фреймворк MVC уже предоставляет большой набор встроенных html-хелперов, которые позволяют генерировать ту или иную разметку, например, код элементов форм.

1. встроенные tag helpers

* <a>
* <script>
* <link>
* <for>
* <input>
* <label>
* <span> для сообщения об ошибке валидации
* <div> для сообщения об ошибках валидации

<environment>

AnchorTagHelper представляет тег-хелпер, который позволяет создавать ссылки. Он может принимать ряд специальных атрибутов:

* **asp-controller**: указывает на контроллер, которому предназначен запрос
* **asp-action**: указывает на действие контроллера
* **asp-area**: указывает на действие область, в которой расположен контроллер или страница RazorPage (если они находятся в отдельной области)
* **asp-page**: указывает на RazorPage, которая будет обрабатывать запрос
* **asp-page-handler**: указывает на обработчик страницы RazorPage, которая будет применяться для обработки запроса
* **asp-host**: указывает на домен сайта
* **asp-protocol**: определяет протокол (http или https)
* **asp-route**: указывает на название маршрута
* **asp-all-route-data**: устанавливает набор значений для параметров
* **asp-route-[название параметра]**: определяет значение для определенного параметра
* **asp-fragment**: определяет ту часть хэш-ссылки, которая идет после символа решетки #. Например, "paragraph2" в ссылке "http://mysite.com/#paragraph2"

**asp-action и asp-controller**

Мы можем создавать ссылки в ASP.NET Core различными способами. Например:

|  |  |
| --- | --- |
| 1  2 | @Html.ActionLink("О сайте","About","Home")  <a href='@Url.Action("About", "Home")'>О сайте</a> |

В первом случае используется html-хелпер, во втором - стандартный элемент ссылки с хелпером Url.Content. Еще один способ предоставляют tag-хелпер AnchorTagHelper:

|  |  |
| --- | --- |
| 1 | <a asp-controller="Home" asp-action="About">О сайте</a> |

В данном случае используется не элемент html <a />, а именно хелпер AnchorTagHelper. Его атрибут **asp-controller** указывает на название контроллера, а **asp-action** определяет действие, которому будет идти запрос. Если указан атрибут asp-action, но не указан asp-controller, то в качестве контроллера используется тот контроллер, который связан с текущим представлением.

Если необходимо установить ссылку на действие контроллера, который находится в другой области, то применяется атрибут **asp-area**:

|  |  |
| --- | --- |
| 1 | <a asp-controller="Home" asp-action="About" asp-route-area="Service">О сайте</a> |

В данном случае предполагается, что контроллер Home находится в области Service.

Если, наоборот, в представлении, которое находится в какой-нибудь области, надо создать ссылку на действие контроллера, который не находится ни в какой области, то указывается пустой атрибут:

|  |  |
| --- | --- |
| 1 | <a asp-controller="Home" asp-action="About" asp-route-area="">О сайте</a> |

**asp-host и asp-protocol**

AnchorTagHelper по умолчанию создает локальную ссылку, если же нам надо создать ссылку на другой домен, то мы можем применить атрибут **asp-host**:

|  |  |
| --- | --- |
| 1 | <a asp-controller="Home" asp-action="About" asp-host="localhost.com" asp-protocol="https">О сайте</a> |

Кроме того, мы можем изменить стандартный протокол на https, использовав атрибут **asp-protocol**. Данный элемент в итоге создает следующую ссылку: *https://localhost.com/Home/About*

**asp-route- и asp-all-route-data**

А что если у нас метод принимает какие-нибудь параметры, которые надо указать в ссылке:

|  |  |
| --- | --- |
| 1 | public IActionResult GetPerson(int id){......} |

В этом случае мы можем использовать атрибут **asp-route-**:

|  |  |
| --- | --- |
| 1 | <a asp-controller="Home" asp-action="GetPerson" asp-route-id="5" >Item5</a> |

Если метод принимает несколько параметров, например:

|  |  |
| --- | --- |
| 1  2  3  4 | public IActionResult GetPerson(int id, string name, int age)  {      return Content($"id={id}  name={name}  age={age}");  } |

то мы можем указать несколько атрибутов asp-route-:

|  |  |
| --- | --- |
| 1 | <a asp-controller="Home" asp-action="GetPerson" asp-route-id="5"  asp-route-age="18" asp-route-name="tom" >Item5</a> |

Чтобы не устанавливать все параметры по отдельности, можно применить атрибут **asp-all-route-data**:

|  |  |
| --- | --- |
| 1 | <a asp-controller="Home" asp-action="GetPerson" asp-all-route-data='new Dictionary<string,string> { { "id", "5" }, {"name", "tom" }, { "age", "18" } }' >Item5</a> |

asp-all-route-data в качестве значения принимает словарь с параметрами и их значениями. В результате будет генерироваться ссылка, аналогичная предыдущей.

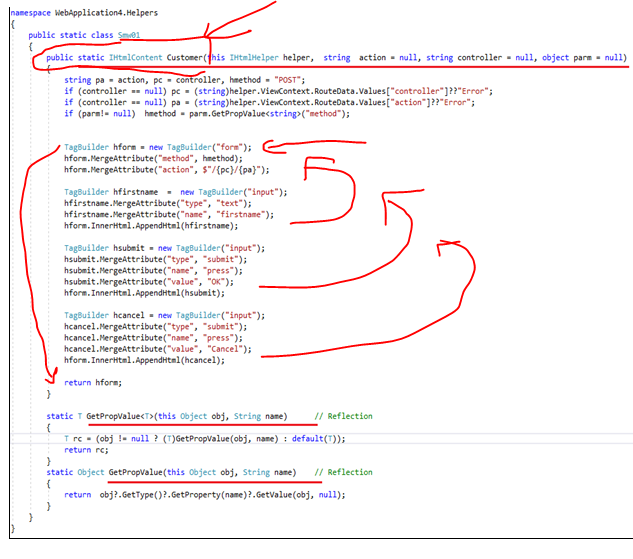
1. ASP.NET CORE: MVC-представление, вспомогательные методы представления (хелперы). Пример.

**Вспомогательные методы** – методы расширения для IHtmlHelper, IHtmlHelper<TModel> (для типизированного представления) или IUriHelper.

вспомогательные методы сохраняют состояние соответственных связанных элементов в ModelState

На картинке ниже можно увидеть Кастомный Вспомогательный метод, который мы создали сами (Внутрь принимаются параметры, на выходе получается html-тег)

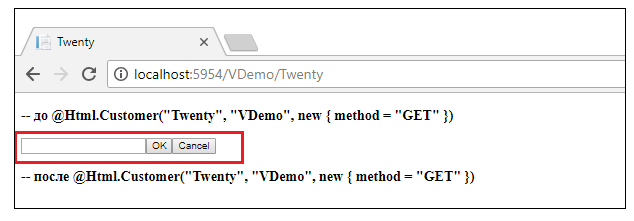
Парсятся приходящие параметры, создается тег **form**; Создаются теги **input**, которые затем встраиваются в тег form. И выводится form.



Ниже вы можете наблюдать встраивание Нашего тега в View страничку и передача внутрь параметров.



Результат: (Как видим Наш Хэлпер готов)



Также существуют уже готовые хэлперы (Аналоги html-кода):





1. ASP.NET CORE: MVC-модель, DB-модель и View-модель. Модель Entity Framework,принцип Code разработки DB-модели. Объект ModelState, назначение и принципы применения. Атрибуты валидации: Required, RegularExpression, пользовательский атрибут валидации. Пример.

**ModelState:** объект, для хранения состояния представления. Заполнение ModelState при заполнении параметров action.

**View Model:** объект, применяемый для передачи данных в представлении.

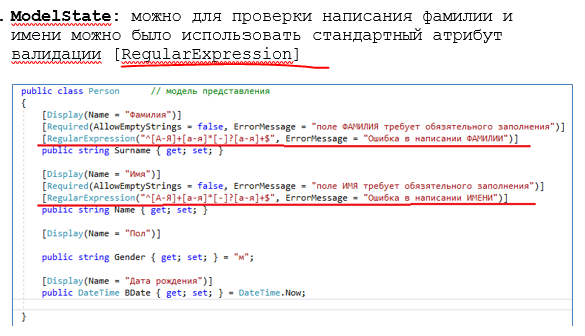
На рисунке снизу показана модель представления с валидацией (стандартные атрибуты c#, но можно и свой сделать)



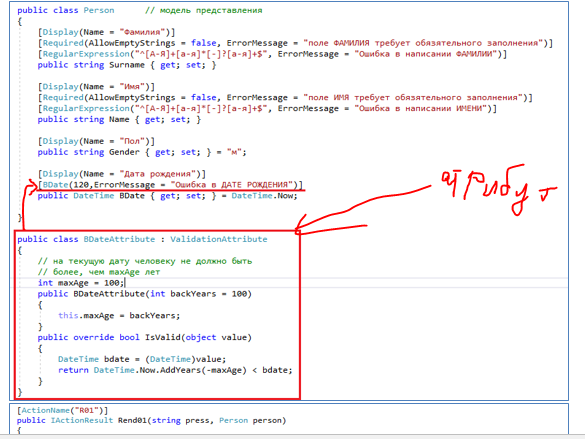
На скрине снизу показан фрагмент записи ошибок валидации в объект ModelState. В случае атрибутов (скрин сверху) они добавляются автоматически



Вот вам примерчик валидации через регулярные выражения (Атрибут RegularExpression):



Можно для проверки даты рождения разработать собственный атрибут валидации. Давайте это сделаем:



1. **MVC Filters**: механизм автоматического выполнения кода в рамках контроллера; фильтры: действий, результатов, исключений, авторизации. Фильтры могут быть в форме атрибутов акции (действует для одну акцию) или атрибутов контроллера (действует для всех акций контроллера). Для однородны фильтров может быть установлен приоритет (Order, чем больше, тем приоритетней)
2. **Action Filter**: позволяет автоматически выполнить методы класса до и после выполнения акции (действия) контроллера.
3. **Result Filter**: позволяет выполнить методы класса, до и после формирования объекта-результата (IActionResult), возвращаемого акцией.
4. **Authorization Filter**: проверяет наличие авторизации текущего пользователя
5. **Resource Filter:** выполняетсяпосле фильтра авторизации и после всех остальных фильтров.
6. **Exception Filter:** обработка исключений в акциях
7. Для однородных фильтров может быть установлен приоритет (Order, чем больше, тем приоритетней). Для этого фильтр должен реализовывать IOrdersFilter (иначе Order = 0).

**Entity Framework** представляет специальную объектно-ориентированную технологию на базе фреймворка .NET для работы с данными. Если традиционные средства ADO.NET позволяют создавать подключения, команды и прочие объекты для взаимодействия с базами данных, то Entity Framework представляет собой более высокий уровень абстракции, который позволяет абстрагироваться от самой базы данных и работать с данными независимо от типа хранилища. Если на физическом уровне мы оперируем таблицами, индексами, первичными и внешними ключами, но на концептуальном уровне, который нам предлагает Entity Framework, мы уже работает с объектами.

Entity Framework представляет прекрасное ORM-решение, которое позволяет автоматически связать обычные классы языка C# с таблицами в базе данных. Entity Framework Core нацелен в первую очередь на работу с СУБД MS SQL Server, однако поддерживает также и ряд других СУБД. В данном случае мы будем работать с базами данных в MS SQL Server.

Также стоит отметить, что здесь мы будем использовать **Entity Framework Core** - кроссплатформенное решение на базе .NET Core, которое отличается от предыдущих версий, например, от Entity Framework 6.

Чтобы взаимодействовать с базой данных через Entity Framework нам нужен контекст данных - класс, унаследованный от класса **Microsoft.EntityFrameworkCore.DbContext**

Свойство DbSet представляет собой коллекцию объектов, которая сопоставляется с определенной таблицей в базе данных. При этом по умолчанию название свойства должно соответствовать множественному числу названию модели в соответствии с правилами английского языка. То есть User - название класса модели представляет единственное число, а Users - множественное число.

Через параметр options в конструктор контекста данных будут передаваться настройки контекста.

В конструкторе с помощью вызова Database.EnsureCreated() по определению моделей будет создаваться база данных (если она отсутствует).

Чтобы подключаться к базе данных, нам надо задать параметры подключения. Для этого изменим файл **appsettings.json**, добавив определенные строки подключения.

И последним шагом в настройке проекта является изменение файла **Startup.cs**. В нем нам надо изменить метод ConfigureServices():

// получаем строку подключения из файла конфигурации

            string connection = Configuration.GetConnectionString("DefaultConnection");

            // добавляем контекст MobileContext в качестве сервиса в приложение

            services.AddDbContext<ApplicationContext>(options =>

                options.UseSqlServer(connection));

            services.AddControllersWithViews();

Добавление контекста данных в виде сервиса позволит затем получать его в конструкторе контроллера через механизм внедрения зависимостей

**Способы взаимодействия с БД**

Entity Framework предполагает три возможных способа взаимодействия с базой данных:

* Database first: Entity Framework создает набор классов, которые отражают модель конкретной базы данных
* Model first: сначала разработчик создает модель базы данных, по которой затем Entity Framework создает реальную базу данных на сервере.
* Code first: разработчик создает класс модели данных, которые будут храниться в бд, а затем Entity Framework по этой модели генерирует базу данных и ее таблиц

**ДОП ИНФА**

ключевые отличия ASP.NET Core от предыдущих версий ASP.NET:

* Новый легковесный и модульный конвейер HTTP-запросов
* Возможность развертывать приложение как на IIS, так и в рамках своего собственного процесса
* Использование платформы .NET Core и ее функциональности
* Распространение пакетов платформы через NuGet
* Интегрированная поддержка для создания и использования пакетов NuGet
* Единый стек веб-разработки, сочетающий Web UI и Web API
* Конфигурация для упрощенного использования в облаке
* Встроенная поддержка для внедрения зависимостей
* Расширяемость
* Кроссплатформенность: возможность разработки и развертывания приложений ASP.NET на Windows, Mac и Linux
* Развитие как open source, открытость к изменениям

---------ДОП ИНФА ----------

**HTTP – сервер:**

**HTTP-сервер** – С часть веб-прилож.

**HTTP-протокол –** формат передачи д-х по TCP.

Полудупл.канал – К посыл.запрос и должен дождаться ответа по тому же каналу.

createServer – для создания С. (вызыв.при поступл.запроса на С).

**Очередь подключ –** очередь заявок на вып-ние accept(выгребает из очереди соед).

Соб.**connection** – возник.когда К выдал ф-цию connect, она раб.при соед.с С и оно поступ о очередь listen. Т.е.вып-ся когда пришло сообщ., что мы хотим соед.

Server.keepAliveTimeout – макс время бездействия К (12000 мс по умолч)

Если есть timeout => close не работает.

**JSON: (JS Object Notation)**

это форма сериализации js-объекта (в строку). JS-объ <-> JSON (м. превращать туда-обратно)

JSON.stringify(obj1) – преобраз js-объект в строку

JSON.parse(json\_obj) – преобраз в js-объект

**HTTP-клиент -** это клиентская часть веб-прилож. К делает запрос и получ ответ,к от.м.обрабат.

\* require(‘http’);

\* созд. объект http.request(options, <ф-я обраб. ответы сервера>)

***options:*** - параметры для обращ. к серверу

- хост (ip или симв. имя gethostbyname)

- парс URI-запроса

- порт

- метод

Если сервер обращ.к др. серверу, в него надо вкладывать http-клиент.

query = require(‘querystring’) --> м. формир. парам-ры

let parms = query.stringify ( {x:3; t:2})

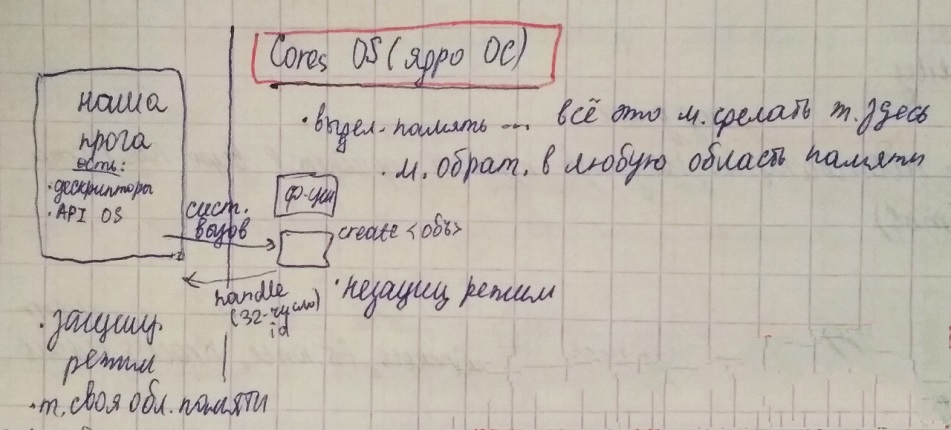
req.write(.) – запих. в тело при POST-запросе

*Директорий* – файл, кот. хранит ссылки на др. файлы

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Сокет –** объект ОС, кот. в своих св-ва хранит (IP, порт, сем-во проток). Мы можем с ним работать через сист. вызовы, используя *дескриптор* **(**32-бит. число – id сист.объекта ядра ОС)

|  |  |
| --- | --- |
| объекты ядра ОС:  файл, сокет, процесс, память… | действия с объектами:  create, open, close |



**Процесс –** ед.работы ОС – объект ОС, кот.связан с каким-то заданием, кот вып-ся в данной ОС

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**TCP:**

Как устроен TCP-сервер?

\* require(‘net’);

\* минимум 2 сокета (1 - для устан.соед, 1 - для обмена сообщ.в режиме реал.времени)

\*listen – переводит С в режпрослуш. + созд.очередь кот ждет подключ

\*accept – выбир.из очереди очередн.эл-т + созд.сокет для обмена сообщ.с К

Как завалить TCP-сервер?

\*подключить столько К, чтобы ост.получили отказ

Во всех ОС мех.сокетов практически одинак. Видим сетевой в-в – за ним скрыв.сокет. Сокеты пропис.в стандарте POSIX.

**Петля –** (127.0.0.1 или 0.0.0.0) – для того, чтобы раб.на лок ПК,

К отлич от С тем, что К – инициатор соед.

session – объект, чтобы хранить инфу о соединении. Если как в лабе неск. клиентов просят вычисление, сессия позволяет серверу отличить их.

**MSSQL**

Npm install mssql

Сервер СУБД – доступ через сокет (как только сервер – сразу сокет).

Для подключ.надо знать:

\*ip /<-- символ.имя //gethostbyname или DNS (в host.ass)

\*порт (sql – 1433, oracle - 1521)

\*имя БД

\*логин, пароль

Динам запрос – знач.в запрос подставл.налету

- нек.код, созд.и сохр в пер-ной, пока не возникн.необх.его вып-ния

-select \* from faculty where fac\_name := name;

Статич.запрос – устан.во время проектир.и не сод.пар-ров и арг-тов

-select \* from faculty where fac\_name = ‘ИТ’;

*Итеративный язык* – парадигма проги, кот.опис проц вычисления в виде инструкций, кот.измен.сост.д-х:

Говорим как вып-ть задачу, опис.алгорит, даем интсрукции

*Декларативн.язык (SQL)* – парадигма проги, кот.опис.специф.решения задачи

Описывается что представл собой проблема и ожид.рез-т

**Группы sql-оп-ций:**

\*ddl –create/drop/alter

\*dml – insert/update/delete/select

\*tcl – commit/rollback, begin tran

\*dcl – grant, revoke, deny

Сходства HTTP и запросы SQL:

\* stateless – отсут. состояние

\* любой запрос http не связан ни с послед., ни с пред; в SQL тоже нельзя связать запросы

\* каждый новый запрос начинает новую жизнь серверу

Sql – доступ осущ.с пом.языка sequel;

Варианты соед.с С:

1. Исп-ние пула соед
2. Открыть С и при запуске им.пост.соед
3. Connect=> запрос к д-м => отключ.соед

*Пул соединений:*

хотим сделать запрос как можно более независимым от остальных (вносим в него измен. – никак не влияет на др. части). Сам себе открываю и закрываю соединение.

Есть min и max кол-во соед. При старте соп. пула, он авто- открывает min соед-й. Когда они все занят, пул откр еще одно… до max. Никогда не закр соед!

*Пул соед* – набор соед. с БД, кот. по мере нагрузки увел. с min->max + очередь

*Прогр. интерфейсы для работы с MSSQL:*

ADO, jdbc, odbc(c++), оледиби

*odbc* – первооснова всех остальных, ост. – это вариации или похожие/одинак

Логику лучше возлагать на SQL-сервер (процедуры, триггеры, ф-и, констрейны…) – более верный подход. Логику двигать подальше от UI, от клиента, прятать глубже.

|  |  |  |
| --- | --- | --- |
| OSI | TCP/IP | |
| 7 Прикладной | HTTP, FTP, [Telnet](https://ru.wikipedia.org/wiki/Telnet), SMTP, DNS ([RIP](https://ru.wikipedia.org/wiki/RIP_(%D1%81%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB)), работающий поверх [UDP](https://ru.wikipedia.org/wiki/UDP), и [BGP](https://ru.wikipedia.org/wiki/Border_Gateway_Protocol), работающий поверх [TCP](https://ru.wikipedia.org/wiki/TCP), являются частью сетевого уровня), [LDAP](https://ru.wikipedia.org/wiki/LDAP), RTP | Прикладной |
| 6 Представления |
| 5 Сеансовый |
| 4 Транспортный | TCP, UDP, [SCTP](https://ru.wikipedia.org/wiki/SCTP), [DCCP](https://ru.wikipedia.org/wiki/DCCP) (протоколы маршрутизации, подобные [OSPF](https://ru.wikipedia.org/wiki/OSPF), что работают поверх [IP](https://ru.wikipedia.org/wiki/IP), являются частью сетевого уровня) | Транспортный |
| 3 Сетевой | IP (вспомогательные протоколы, вроде [ICMP](https://ru.wikipedia.org/wiki/ICMP) и [IGMP](https://ru.wikipedia.org/wiki/IGMP), но являются частью сетевого уровня; [ARP](https://ru.wikipedia.org/wiki/ARP) не работает поверх IP) | Сетевой |
| 2 Канальный | Ethernet, Token Ring, и подобные | Канальный |
| 1 Физический |

Свойства http - Stateless, Синхронность, Клиент-сервер

**Функции,** обеспечивающие взаимодействие открытых систем в модели ISO/OSI распределены по следующим уровням:

1. Физический – определяет свойства среды передачи д-х и способы ее соединения с сетевыми адаптерами.
2. Канальный – подготовить блок данных для сетевого уровня

\*управление доступом к среде передачи данных

(опред.методы совместного исп-ния сетевыми адаптерами среды перед д-х)

\*управление логическим каналом

(опред.понятие канала между 2 сетевыми адаптерами + способы обнаруж.и исправления ошибок передачи данных)

1. Сетевой – опред методы адресации и маршрутизации ПК в сети.
2. Транспортный – подготовка и доставка пакетов данных между конечными точками без ошибок и в правильной последовательности.
3. Сеансовый – опред способы установка и разрыва соединений 2 приложений, работающих в сети.
4. Представительский – опред формат д-х, используемых прилож
5. Прикладной – опред способы взаимодействия пользователей с системой (определить интерфейс)

**Задача каждого уровня** – предоставление услуг вышестоящему уровню, таким образом, чтобы детали реализации этих услуг были скрыты.

Данные проходят от прикладного уровня одной системы до прикладного уровня другой системы через все нижние уровни системы. При чем по мере своего движения от отправителя к получателю данные подвергаются необходимому преобразованию: при движении от прикладного к физическому данные преобразовываются в формат, позволяющий передать данные по физическому каналу; при движении от физического уровня до прикладного происходит обратное преобразование данных.

1. Кто может назначать узлам IP адреса и прочие параметры

Администратор

1. С помощью чего клиент сможет получить от сервера все необходимые параметры

Запросы, DHCP

1. Как узел, не имеющий IP адреса, может общаться с сервером? От какого IP адреса узел, пока не имеющий Ip адреса сможет отправлять пакеты?

Широковещательный запрос.

От кого – пишем все 0.

1. Откуда клиент будет знать IP адрес сервера перед тем, как начинать взаимодействие?

Со стороны клиента рассылаем широковещательный запрос, сервер получает и отсылает клиенту свой IP

1. Как клиет сможет получить от сервера ответ, если он еще не имеет IP адреса?

Сервер отправляет широковещательный запрос всем клиентам.

Клиент взаимодействует с первым сервером, приславшим запрос.

Предполагается, что тот кто прислал первым, находится ближе, а значит лучше обмен.

1. Откуда DHCP сервер узнает, какие адреса и дополнительные конфигурационные параметры предлагать клиенту

Клиент при отправке запроса на получение IP адреса может попросить и другие параметры (маску, время аренды IP-адреса и т.д.).

**Граница между аппаратным и программнм обеспечением –** какльный уровень между подур-ми.

**Уровни модели TCP/IP:**

|  |  |  |
| --- | --- | --- |
| Прикладной | Содержит службы TCP/IP + прикладные сист.юзера | HTTP, TELNET, POPS |
| Транспортн | Обесп.сквозную доставку д-х произв.размера по сети, между прикл.процессами, запущ.на узлах сети | TCP, UPD |
| Межсетевой | Осущ.перенос между сетями разл типов адресной инфы в унифиц.форме. | IP, ICMP, ARP |
| Ур.доступа к сети | Исп.протоколы, кот.обесп.создание лок.сетей или соед.с глобальными сетями | Ethernet, Token Ring |

# DHCP

Служба DHCP занимает порт UDP 67.

Отметим, что DHCP клиент пользуется портом UDP 68.

Служба находится на прикладном уровне, а реализация протоколов на сетевом уровне. (ОТЛИЧИЕ)

## Взаимодействие клиента и сервера

* **DHCPDISCOVER** – отправляется от клиента к серверу. в качестве отправителя все 0000. в качестве получателя 255.255.255.255. клиент, не зная что за сеть, может выдвинуть требования серверу для установки IP адреса с параметрами (маска, время и т.д.). этот пакет рассылается по сети. Содержит в себе ключ, который генерируется. Сервера обязаны отреагировать ответом DCHPOFFER.
* **DHCPOFFER** – ответ сервера; посылается клиенту с ip адресом; отсылается ответ «я сервер, со мной можно общаться по такому-то адресу. я могу выдать вам такой-то ip». этот пакет также отсылается всем в сети. Вставляет также ключ, который был сформирован клиентом. Клиент смотрит его ключ или нет, и в зависимости от этого или принимает, или не принимает этот IP.
* **DHCPREQUEST** – возможность дополнительно установить какие-то параметры. Отсылается от клиента серверам широковещательно. IP адрес сервера уже известен. Указываем параметры какие хотим получить.
* **DHCPPACK** – получает DHCPREQUEST и смотрит был ли какой-то клиент, который быстрее запросил что-то, чем мы. Если такого не было, то записываем клиента (нас) в таблицу с выданными параметрами. И выдаем это подтверждение клиенту. Клиент выставляет параметры и пользуется выделенным IP адресом.
* **DHCPNACK** – если кто-то успел забрать IP-адрес быстрее нас, то он отправляет сообщение клиенту, что данный IP адрес уже занят. Тогда клиент заново посылает широковещательный запрос на получение IP адреса и делает это до тех пор, пока не получит его.
* **DHCPDECLINE** – клиент отправляет запрос серверу, если в сети уже есть хост с таким же IP адресом и параметрами. Т.е. если клиент получил и пришел хост с уже установленными параметрами. Клиент в таком случае просит новый IPадрес и параметры.

**FTP** расшифровывается как File Transfer Protocol — протокол передачи файлов. Он отличается от других протоколов тем, что если в процессе передачи возникает какая-то ошибка, то процесс останавливается и выводится сообщение для пользователя. Если ошибок не было, значит, пользователь получил именно тот файл, который нужен, в целости и без недостающих элементов.