**ПРОГРАММИРОВАНИЕ ИНТЕРНЕТ-СЕРВЕРОВ**

[1. Протокол HTTP: клиент-сервер; типы сообщений, структура запроса, структура ответа, статус (серии значений), методы, заголовки, параметры. Понятие stateless-протокола.](#_18zmb2ej9ii3) 3

[2. Протокол HTTPS: TLS, шифронаборы, сертификаты, процедура рукопожатия.](#_kdh8gox801dj) 5

[3. HTML. Структура HTML-страницы. Каскадные таблицы стилей (CSS). Модель DOM. Пример.](#_fcvh8jam35s) 7

[4. Протокол WebSockets: принципы работы и применения. Пример.](#_ll7xcdvw3as) 10

[5. JavaScript. Основные стандарты. Типы данных. Программные структуры. Принцип применения. Пример.](#_6ptiz0ghdnze) 12

[6. Методология Ajax. Структура Ajax-приложения, принципы разработки и применения. Пример.](#_v76zud6qemlp) 15

[7. Web-приложение. Архитектура web-приложения. Особенности реализации web-приложения. Web-сервер и web-клиент. Пример.](#_l6fmvac5xdxl) 19

[8. ASP.NET: публикация ASP.NET-приложения, структура и параметры узла IIS, реальный и виртуальный каталоги, процедура настройки web-узла.](#_d431gv5t3hgq) 26

[9. ASP.NET: http-обработчики, порядок разработки, http-обработчик для взаимодействия с клиентом по протоколу WebSockets. Пример.](#_bpdopvqhkezs) 28

[10. ASP.NET: ASMX-сервисы, WSDL, SOAP, прокси, порядок разработки, принципы применения. Пример.](#_t791xz9qnzzk) 30

[11. ASP.NET: MVC-приложение, структура MVC-приложения, назначение основных компонентов приложения, маршрутизация. Пример.](#_ovu8eitysegd) 34

[12. ASP.NET: MVC-приложение, маршрутизатор, принципы устройства и работы. Пример.](#_fteypfojrj5i) 36

[14. ASP.NET: MVC-приложение, котроллер, жизненный цикл контроллера, взаимодействие с моделью и представлениями. Пример.](#_muada8jhscti) 38

[15. ASP.NET: MVC-приложение, представление, Razor Engine, жизненный цикл представления. Пример.](#_qjwrmrmjxk69) 39

[16. ASP.NET: MVC-приложение, модель, жизненный цикл модели, репозиторий. Пример.](#_yxt07qrd943) 41

[17. ASP.NET: MVC-приложение, внедрение зависимостей. Пример.](#_y7m3je8rdtgn) 42

[18. ASP.NET: MVC Web API, структура Web API-приложения; назначение основных компонентов приложения, маршрутизация. Пример.](#_w9zdvnfl30s9) 43

[19. WCF-сервисы: WSDL, хост, прокси, модели взаимодействия клиента и сервера, порядок разработки, принципы применения. Пример.](#_gv62s07k3xkd) 44

[20. ASP.NET CORE: программная платформа, принципы работы, архитектура. Пример.](#_k96eakgocyhw) 47

[21. ASP.NET CORE: работа со статическими файлами, добавление заголовков, стартовые страницы, файлы для скачивания, вывод в журнал. Пример.](#_e56i6ecsr2gy) 48

[22. ASP.NET CORE: MVC, настройка MVC и маршрутизатора, применение атрибута Route для маршрутизации. Пример.](#_rtguoqdfyo25) 48

[Маршруты можно указывать явно с помощью [Route(“...”)] над IActionResult, таких можно указывать несколько.](#_zd0wzpapulcv) 48

[23. ASP.NET CORE: MVC-котроллер, действия (action) контроллера, контекст контроллера, поддержка сессии, результат работы действия, внедрение зависимостей. Пример.](#_jmq4cv8jy565) 49

[24. ASP.NET CORE: события OnAction, атрибуты HttpGet, HttpPost, …, AcceptVerb, принцип передачи параметров в метод действия.](#_lg6fjh6g0uu) 50

[25. ASP.NET CORE: Фильтры: Action Filter, Result Filter, Authorization Filter, Resource Filter, Exception Filter, пользовательские фильтры действий.](#_pewl9sbpj3tr) 51

[26. ASP.NET CORE: MVC-представление, обнаружение представления, жизненный цикл представления, методы рендеринга представления в web-страницу (методы View контроллера), способы передачи данных из контроллера в представление, строготипизированные представления, директива @model. Пример.](#_k6256mktnwpn) 52

[27. ASP.NET CORE: MVC-представление, директивы @using, @function, @inherits. Пример.](#_8fi73np54o7o) 56

[28. ASP.NET CORE: MVC-представление, директивы @addTagHelper, @removeTagHelper. Пример.](#_ut6mbnuwmbf6) 61

[29. ASP.NET CORE: MVC-представление, применение компоновки (Layout) представления, компоновка по умолчанию (\_ViewStart), применение секций @RenderSection, @RenderBody. Пример.](#_nyuttygpyekc) 62

[30. ASP.NET CORE: MVC-представление, частичные представления (partial view). Встроенные хелперы. Пример.](#_uqn332uqukv4) 65

[31. ASP.NET CORE: MVC-представление, вспомогательные методы представления (хелперы). Пример.](#_dfohn3nv5z6n) 67

[32. 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.](#_ifap3wcjvi2p) 70

***1ый вопрос***

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

**HTTP-протокол** – протокол передачи данных прикладного уровня, ассиметричный (сообщения от клиента к серверу и от сервера к клиенту разные). Протокол на каждый запрос должен получить один ответ, если на 1 запрос – 2 ответа – ошибка. Всегда подразумевает пару request/response. Относится к протоколу, который не помнит своего состояния. В запросе и ответе нет никаких ссылок на предыдущий и последующий ответ и запрос. Каждый запрос-ответ – новый жизненный цикл HTTP (stateless протокол).

Клиент-серверная архитектура – архитектура, которая подразумевает 2 компонента (клиент, являющийся инициатором соединения и сервер). В качестве клиента выступает веб-браузер.

Когда мы говорим о протоколе http между клиентом и сервером ходит 2 типа сообщения:

1. От клиента к серверу ходит запрос (request)

2. От сервера к клиенту ходит ответ (response)

**Request**

- метод (get/post/put/option и тд)

- URI (описывает месторасположение сервера)

- версия протокола (HTTP/1.1)

- заголовки (пары: имя/заголовок)

- параметры (пары: имя/значение)

- расширение(тело)

Все заголовки могут быть 2 основных типов:

- стандартные (прописаны в протоколе HTTP)

- пользовательские (рекомендация: они должны начинаться с буквы «x»)

**Response**

- Версия протокола

- Код состояния (1xx,2xx,3xx,4xx,5xx, статус ответа)

- Пояснение к коду состояния

- Расширение

- Заголовки (пары: имя/заголовок)

**Запрос (Request):** серверный объект, который образуется в результате обработки сервером http-запроса, поступающего от клиента и передается серверному программному коду для обработки. Обычно объект Request предоставляет возможность хранить данные в формате ключ/значение.

**Ответ (Response):** серверный объект, который автоматически формируется сервером, при получении http-запроса, заполняется данными серверными программным кодом, преобразуется в http-ответ и отправляется клиенту.

**Методы**

- options, get (с передачей параметров в адресной строке)

- head, post (параметры передаются в теле запроса)

- put, delete, trace, connect, extension-method

**Заголовки**

– General – общие заголовки, используются в запросах и ответах (cache-control, connection)

– Request – заголовки, используются только в запросах

– Response – заголовки, используются только в ответах

– Entity – заголовки, которые описывают сущность в ответах и запросах. Сущностью называется то, что находится в header.

**В ответе может быть код состояния:**

1. Информационные: 1xx

2. Успешный ответ: 2xx

3. Переадресация: 3xx

4. Ошибка клиента: 4xx

5. Ошибка сервера: 5xx

**Statless-протокол** – протокол, не сохраняющий состояния, то есть сервер не сохраняет никаких данных (состояние) между парами запросов-ответов.

##### **2. Протокол HTTPS: TLS, шифронаборы, сертификаты, процедура рукопожатия.**

**HTTPS (Hypertext Transport Protocol Secure)** – это протокол, который обеспечивает конфиденциальность обмена данными между сайтом и пользовательским устройством. HTTP – как транспорт. Протокол TLS – основан на ассиметричном криптовании, на этапе соединения они договариваются как криптуются. Сначала договариваются, а потом передают.

Безопасность информации обеспечивается за счет использования криптографических протоколов SSL/TLS, имеющих 3 уровня защиты:

1. Шифрование данных (Позволяет избежать их перехвата)

2. Сохранность данных (Любое изменение данных фиксируется)

3. Аутентификация (Защищает от перенаправления пользователя)

=================================================================================

**HTTPS** — расширение протокола HTTP для поддержки шифрования в целях повышения безопасности. Данные в протоколе HTTPS передаются поверх криптографических протоколов TLS. В отличие от HTTP с TCP-портом 80, для HTTPS по умолчанию используется TCP-порт 443.

**Протокол TLS** (transport layer security) основан на протоколе SSL (Secure Sockets Layer), для повышения безопасности в Интернете. Протокол SSL был реализован на application-уровне, непосредственно над TCP (Transmission Control Protocol).

TLS ставит своей целью создание между двумя узлами сети защищённого от прослушивания и подмены информации канала связи, пригодного для передачи произвольных данных в обоих направлениях, а также проверку того, что обмен данными происходит между именно теми узлами, для которых канал и планировался.

**Процедура рукопожатия:**

Перед тем, как начать обмен данными через TLS, клиент и сервер должны согласовать параметры соединения, а именно: версия используемого протокола, способ шифрования данных, а также проверить сертификаты, если это необходимо.

**Шифронабор** — это набор алгоритмов, определяющих параметры безопасного соединения.

В начале любого соединения самое первое взаимодействие, представляет собой список поддерживаемых шифронаборов. Сервер выбирает лучший, наиболее безопасный вариант, который поддерживается им и отвечает его требованиям. Вы можете посмотреть на шифронабор и выяснить все параметры рукопожатия и соединения

TLS обеспечивает отправку каждого сообщения с кодом **MAC** (Message Authentication Code), алгоритм создания которого – односторонняя криптографическая функция хеширования, ключи которой известны обоим участникам связи. Всякий раз при отправке сообщения, генерируется его MAC-значение, которое может сгенерировать и приёмник, это обеспечивает целостность информации и защиту от её подмены.

**Разберём каждый шаг процедуры рукпожатия:**

• Так как TLS работает над TCP, для начала между клиентом и сервером устанавливается TCP-соединение.

• После установки TCP, клиент посылает на сервер спецификацию в виде обычного текста (а именно версию протокола, которую он хочет использовать, поддерживаемые методы шифрования, etc).

• Сервер утверждает версию используемого протокола, выбирает способ шифрования из предоставленного списка, прикрепляет свой сертификат и отправляет ответ клиенту (при желании сервер может так же запросить клиентский сертификат).

• Версия протокола и способ шифрования на данном моменте считаются утверждёнными, клиент проверяет присланный сертификат и инициирует либо RSA, либо обмен ключами по Диффи-Хеллману, в зависимости от установленных параметров.

• Сервер обрабатывает присланное клиентом сообщение, сверяет MAC, и отправляет клиенту заключительное (‘Finished’) сообщение в зашифрованном виде.

• Клиент расшифровывает полученное сообщение, сверяет MAC, и если всё хорошо, то соединение считается установленным и начинается обмен данными приложений.

Также имеется дополнительное расширение процедуры Handshake, которое имеет название TLS False Start. Это расширение позволяет клиенту и серверу начать обмен зашифрованными данными сразу после установления метода шифрования, что сокращает установление соединения на одну итерацию сообщений. Об этом подробнее рассказано в пункте “TLS False Start”.

##### **3. HTML. Структура HTML-страницы. Каскадные таблицы стилей (CSS). Модель DOM. Пример.**

**HTML (от англ. Hypertext Markup Language — «язык разметки гипертекста»)** — это стандартный язык разметки документов во Всемирной паутине.

Элемент **<!DOCTYPE>** предназначен для указания типа текущего документа — DTD (document type definition, описание типа документа). Это необходимо, чтобы браузер понимал, как следует интерпретировать текущую веб-страницу, ведь HTML существует в нескольких версиях, кроме того, имеется XHTML (EXtensible HyperText Markup Language, расширенный язык разметки гипертекста), похожий на HTML, но различающийся с ним по синтаксису.

**Структура**



DOCTYPE – это тег, который сообщает браузеру версию HTML. Этот тег должен быть написан на первой строчке HTML документа.

Тег **<html>** определяет начало HTML-файла, внутри него хранится заголовок <head> и тело документа <body>.

Тег **<meta>** является универсальным и добавляет целый класс возможностей, с помощью метатегов, можно изменять кодировку страницы, добавлять ключевые слова, описание документа и многое другое.

Тег **<title>** определяет заголовок веб-страницы, это один из важных элементов предназначенный для решения множества задач. является обязательным и должен непременно присутствовать в коде документа.

Тело документа **<body>** предназначено для размещения тегов и содержательной части веб-страницы.

**CSS (англ. Cascading Style Sheets — каскадные таблицы стилей)** — формальный язык описания внешнего вида документа, написанного с использованием языка разметки. CSS из таблицы стилей имеет 2 основные части — селектор и блок объявлений. Подключаем так <link rel="stylesheet" href="style.css">.

Структура языка:

CSS можно охарактеризовать простыми словами как набор правил, описывающих, как должен выглядеть элемент. Правило состоит из селектора и блока объявлений.



В качестве селектора может выступать любой тег

Виды селекторов:

· Универсальный

· Селектор тегов

· Селектор классов

Селектор идентификаторов

**Подключение CSS**

CSS можно связать с HTML несколькими способами:

· внутри тега с помощью атрибута style. При этом нет нужды указывать селектор;

· добавить тег <style> с атрибутом type="text/css";

· подключить внешнюю таблицу стилей: <link rel="stylesheet" href="путь до style.css" type="text/css"/>.

Третий способ является наиболее популярным и рекомендуемым, потому что позволяет в полной мере пользоваться преимуществами разделения формы и содержания, обеспечиваемого с помощью CSS.

**DOM (Document Object Model — «объектная модель документа»)** — это не зависящий от платформы и языка программный интерфейс, позволяющий программам и скриптам получить доступ к содержимому HTML, XHTML и XML-документов, а также изменять содержимое, структуру и оформление таких документов.

**Дом – это правила, по которым формируется любая страница в интернете. Он нужен, чтобы гибко находить нужный элемент, изменять его, дополнять, добавлять и удалять элементы и т.д.**

**Любой документ с помощью DOM, может быть представлен в виде дерева узлов (объектов, например, элемент, атрибут). Узлы связаны между собой отношением – «родительский-дочерний»**

****

**Благодаря этой объектной модели JavaScript получает всю мощь, необходимую для создания динамического HTML. Например:**

**document.getElementsByTagName("P");**

##### 

##### **4. Протокол WebSockets: принципы работы и применения. Пример.**

**WebSocket —** протокол двунаправленной связи между веб-браузером и сервером в режиме реального времени.

Веб-сокеты определяются как двусторонняя связь между серверами и клиентами, что означает, что обе стороны обмениваются данными и обмениваются данными одновременно.

**Протокол состоит из двух частей —** установления соединения (hanshake) и передачи данных.

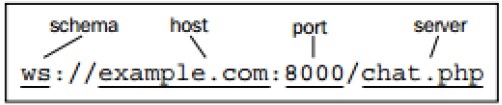
Когда сервер и клиент послали handshake запросы, и проверка пройдена, то начинается этап обмена данными.

**Основные особенности веб-сокетов следующие:**

* Протокол веб-сокета стандартизирован, что означает, что с помощью этого протокола возможна связь между веб-серверами и клиентами в режиме реального времени.
* Веб-сокеты трансформируются в кроссплатформенный стандарт для обмена данными между клиентом и сервером в режиме реального времени.
* Этот стандарт допускает новый вид приложений. С помощью этой технологии предприятия, работающие в режиме реального времени, могут ускорить работу.
* Самое большое преимущество Web Socket – это двусторонняя связь (полный дуплекс) по одному TCP-соединению.

Протокол веб-сокета также имеет аналогичную схему, определенную в его шаблоне URL.

**На следующем рисунке показан URL-адрес веб-сокета в токенах.**

****

Методы, которые используются для двусторонней связи между сервером и клиентом.

* Опрос - периодические запросы независимо от данных, которые существуют в передаче.
* Длинный опрос - как опрос. Клиент и сервер поддерживают соединение активным, пока не будут получены некоторые данные или не истечет время ожидания.
* Streaming - Сервер поддерживает соединение открытым и активным с клиентом до тех пор, пока не будут получены необходимые данные
* Постбэк и AJAX - основан на объекте **Jmlascript XmlHttpRequest** . Это сокращенная форма асинхронного JavaScript и XML.
* HTML5 - Основными компонентами являются API-интерфейсы **разметки, CSS3** и **Javascript** .

**Web Socket –** это независимый протокол на основе TCP, но он предназначен для поддержки любого другого протокола, который традиционно работает только поверх чистого TCP-соединения.

* Web Socket – это транспортный уровень, поверх которого может работать любой другой протокол. Web Socket API поддерживает возможность определения суб-протоколов: библиотек протоколов, которые могут интерпретировать определенные протоколы.
* Примеры таких протоколов включают XMPP, STOMP и AMQP. Разработчикам больше не нужно думать с точки зрения парадигмы HTTP запрос-ответ.
* Единственное требование на стороне браузера – это запуск библиотеки JavaScript, которая может интерпретировать рукопожатие Web Socket, устанавливать и поддерживать соединение Web Socket.
* На стороне сервера промышленным стандартом является использование существующих библиотек протоколов, работающих поверх TCP, и использование шлюза Web Socket.

Веб-сокеты могут быть дополнительным дополнением к существующей среде HTTP и могут предоставить необходимую инфраструктуру для добавления веб-функциональности. Он опирается на более совершенные полнодуплексные протоколы, которые позволяют передавать данные в обоих направлениях между клиентом и сервером.

**Шаги для установления соединения Web Socket следующие:**

* Клиент устанавливает соединение через процесс, известный как рукопожатие Web Socket.
* Процесс начинается с того, что клиент отправляет на сервер обычный HTTP-запрос.
* Требуется обновление заголовка. В этом запросе он сообщает серверу, что запрос на подключение к веб-сокету.
* URL-адреса веб-сокетов используют схему **ws** . Они также используются для безопасных соединений через веб-сокеты, которые эквивалентны HTTP.

**Простой пример начальных заголовков запроса:**

GET ws://websocket.example.com/ HTTP/1.1  
Origin: <http://example.com>  
Connection: Upgrade  
Host: websocket.example.com  
Upgrade: websocket

##### **5. JavaScript. Основные стандарты. Типы данных. Программные структуры. Принцип применения. Пример.**

Язык программирования, разработанный для записи "сценариев", последовательностей операций, которые пользователь может выполнять на компьютере. Скрипты – это программы которые не требует компиляции.

**JavaScript** — объектно-ориентированный скриптовый язык программирования. JavaScript обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.

JavaScript создавался как скриптовый язык для Netscape. После чего он был отправлен в ECMA International для стандартизации. Это привело к появлению нового языкового стандарта, известного как ECMAScript. Проще говоря, ECMAScript — стандарт, а JavaScript — самая популярная реализация этого стандарта.

Всего существует 8 версий (стандартов) ECMAScript.

Основная идея JavaScript состоит в возможности изменения значений атрибутов HTML-контейнеров и свойств среды отображения в процессе просмотра HTML- страницы пользователем. При этом перезагрузки страницы не происходит. На практике это выражается в том, что можно, например, изменить цвет фона страницы или интегрированную в документ картинку, открыть новое окно или выдать предупреждение.

С помощью JavaScript можно:

- Динамически изменять содержимое веб-страниц

- Привязывать к элементам обработчики событий

- Выполнять код через заданные промежутки времени

- Управлять поведением браузера (открывать новые окна, загружать указанные документы и т.д.)

- Создавать и считывать cookies

- Определять, какой браузер использует пользователь (также можно определить ОС, разрешение экрана, предыдущие страницы, которые посещал пользователь)

- Проверять данные форм перед отправкой их на сервер и многое другое

**Встраивание JS в html-код можно:**

- между парой тегов <script> и </script>;

- JavaScript в <head>

- JavaScript в <body>

<script type='text/javascript'> document.write ('текст'); </script>

- из внешнего файла, заданного атрибутом src тега <script>;

<script type="text/javascript" src="ex.js"></script>

- в обработчик события, заданный в качестве значения HTML атрибута, такого как onclick или onmouseover;

<button onClick="alert('П ривет')">нажми</button>

- как тело URL адреса, использующего специальный спецификатор псевдопротокола javascript:

<a href="JavaScript:new Date().toLocaleTimeString();">который час?</a>

JavaScript является слабо типизированным или динамическим языком. Это значит, что вам не нужно определять тип переменной заранее. Тип определится автоматически во время выполнения программы. Также это значит, что вы можете использовать одну переменную для хранения данных различных типов. Объявляется переменная при помощи слова var. Имя переменных в JavaScript не может (!) начинаться с цифр. Переменные с одинаковыми именами, написанными в разном регистре, будут являться разными переменными. Нельзя использовать для имен переменных зарезервированные слова, так как они используются самим языком var, class, return, export и др).

**Константа** – это переменная, значение которой никогда не меняется.

**Типы данных**

Число «**number**». Единый тип: используется как для целых, так и для дробных чисел.

Существуют специальные числовые значения **Infinity** (бесконечность) и **NaN** (ошибка вычислений). Infinity получается при делении на ноль. Ошибка вычислений NaN будет результатом некорректной математической операции, например, при умножении «нечисла» на число.

Строка «**string**». В JavaScript одинарные и двойные кавычки равноправны, можно использовать или те или другие.

Булевый (логический) тип «**boolean**». У него всего два значения: true (истина) и false (ложь). Как правило, такой тип используется для хранения значения типа да/нет.

**Специальное значение «null».** Значение null не относится ни к одному из типов выше, а образует свой отдельный тип, состоящий из единственного значения null. Это просто специальное значение, которое имеет смысл «ничего» или «значение неизвестно».

**Специальное значение «undefined»**. Значение undefined, как и null, образует свой собственный тип, состоящий из одного этого значения. Оно имеет смысл «значение не присвоено». Если переменная объявлена, но в неё ничего не записано, то её значение как раз и есть undefined.

**Символ «Symbol»** (в ECMAScript 6). Это новый примитивный тип данных, который служит для создания уникальных идентификаторов. let sym = Symbol(); Все символы уникальны. Символы с одинаковым именем не равны друг другу. Основная область использования символов – это системные свойства объектов, которые задают разные аспекты их поведения.

**Объекты «object».** Данный тип стоит особняком. Используется для коллекций данных и для объявления более сложных сущностей. Объявляются объекты при помощи фигурных скобок {...}, например: var user = { name: "Вася" };

Оператор typeof x позволяет выяснить, какой тип находится в x, возвращая его в виде строки.

При преобразовании в число null становится 0, а undefined становится NaN.

**Функции**

JS поддерживает функции. Есть встроенные функции: alert(message). А также можно писать свои собственные. Вначале идет ключевое слово function, после него имя функции, затем список параметров в скобках и тело функции – код, который выполняется при её вызове.

**Циклы**

Для многократного повторения одного участка кода – предусмотрены циклы.

while – проверка условия перед каждым выполнением. Пока условие верно – выполняется код из тела цикла: while (условие) { код, тело цикла}

do…while – проверка условия после каждого выполнения. do { тело цикла } while (условие); – сначала выполняет тело, а затем проверяет условие.

for – проверка условия перед каждым выполнением, а также дополнительные настройки. Применятся чаще всего. for (начало; условие; шаг) { тело цикла }

Прерывание цикла: break.

Директива continue прекращает выполнение текущей итерации цикла (прерывает только текущее выполнение его тела, как будто оно закончилось).

**Условные операторы**

if («если»). Получает условие, вычисляет его, и если результат – true, то выполняет команду. if (условие) { код }

else («иначе»). Необязательный блок выполняется, если условие неверно. if (условие) { код } else { код }

else if. Когда необходимо проверить несколько условий. else if (условие) { код } else if (условие) { код } else { код }

Оператор вопросительный знак '?'. Состоит из трех частей: условие ? значение1 : значение2. Проверяется условие, затем если оно верно – возвращается значение1, если неверно – значение2.

##### **6. Методология Ajax. Структура Ajax-приложения, принципы разработки и применения. Пример.**

**AJAX (Asynchronous JavaScript And XML)** — асинхронный js и xml. Это методология (механизм), с помощью которой мы можем выполнять асинхронные запросы к серверу. Это методология построения интерактивного пользовательского интерфейса web-приложения. Методология Ajax описывает способы разработки страниц сайта, которые могут динамически изменяться на основе данных поступающих с сервера, но без полной перезагрузки страницы.

*Структура Ajax-приложения, принципы разработки и применения*

**В основе методологии Ajax лежат следующие технологии:**

· язык HTML - язык гипертекстовой разметки. Интерпретируется браузером. В Ajax динамически изменяется содержимое html-документа.

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

· XML (eXtensible Markup Language) — расширяемый язык разметки данных. Предназначен для структуризации данных с целью хранения или/и передачи.

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

· протокол HTTP - HTTP (англ. HyperText Transfer Protocol — «протокол передачи гипертекста») — сетевой протокол передачи гипертекста. Используется для обмена данными между двумя приложениями (клиентом и сервером). В Ajax обмен данными между JavaScript-сценарием на клиенте и серверным приложением (например, сервлетом) осуществляется по правилам HTTP.

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

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

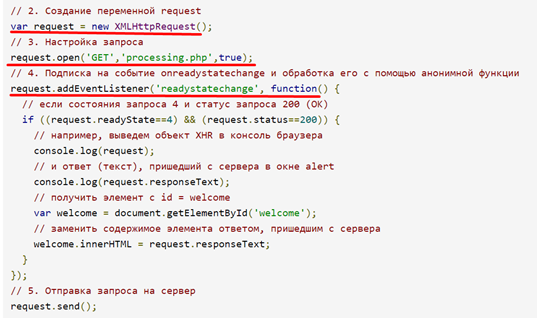
· Ф-я обратного вызова (callback) – функция, которая вызывается после того, как пришел ответ от сервера, для его обработки.

**Синхронный** — значит, поочередный, асинхронный – значит, очередности нет (это запрос без ожидания ответа от сервера (Интернет). Если мы хотим синхронизировать 2 процесса, это значит, что мы хотим установить порядок их работы. А если процессы асинхронные, то это значит, что мы не знаем, в каком порядке они будут выполняться.

**Асинхронный запрос** — запрос, который мы с вами выполняем и дальше начинаем делать что-то другое. А потом, когда приходит ответ, с помощью какого-то механизма мы обрабатываем ответ от сервера. Этот какой-то механизм — это объект JS, который живет в браузере и называется XMLHttpRequest. Насчет этого объекта долго ругались создатели браузеров и они договорились о том, что он будет себя одинаково вести во всех браузерах. С помощью JS мы можем создать экземпляр этого объекта и с помощью этого экземпляра можем делать асинхронные запросы. Т.е. если мы с одной страницы хотим выдать несколько асинхронных запросов, мы должны создать несколько экземпляров XMLHttpRequest и каждый их них будет делать свой асинхронный запрос.

Синхронный - блокирует поток до получения сообщения, асинхронный – нет.

*Структура AJAX*

**

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

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

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

**Методы объекта XmlHttpRequest**

· getAllResponseHeaders() — получить все заголовки ответа от сервера.

· getResponseHeader(«имя\_заголовка») — получить указаный заголовок.

· setRequestHeader(«имя\_заголовка»,«значение») — установить значения заголовка запроса.

· open(«тип\_запроса»,«URL»,«асинхронный»,«имя\_пользователя»,«пароль») — инициализация запроса к серверу, указание метода запроса. Тип запроса и URL — обязательные параметры. Третий аргумент — булево значение. Обычно всегда указывается true или не указывается вообще (по умолчанию — true). Четвертый и пятый аргументы используются для аутентификации (это очень небезопасно, хранить данные об аутентификации в скрипте, так как скрипт может посмотреть любой пользователь). (вызывается первым после создания XMLHttpRequest, вызов open – не открывает соединение. Он настраивает запрос.)

· send(«содержимое») — послать HTTP запрос на сервер и получить ответ. (открывает соединение и отправляет запрос на сервер.)

· abort() — отмена текущего запроса к серверу.

**Свойства объекта XmlHttpRequest**

· Status – http-код ответа.

· StatusText – текстовое описание http-кода.

· ResponseText – текст ответа сервера.

· onreadystatechange — одно из самых главных свойств объекта XMLHttpRequest. С помощью этого свойства задаётся обработчик, который вызывается всякий раз при смене статуса объекта.

· readyState — число, обозначающее статус объекта.

Ответ с сервера приходит в JSON.

5 состояния запроса:

· 0 – начальное состояние

· 1 – вызван open

· 2 – получены заголовки

· 3 – загружается тело

· 4 – запрос завершён

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

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

· UpdatePanel

· Trigger

· Timer

· UpdateProgress

##### 

##### **7. Web-приложение. Архитектура web-приложения. Особенности реализации web-приложения. Web-сервер и web-клиент. Пример.**

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

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

**Порт** — программа на сервере, которая прослушивает входящие сообщения.

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

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

На стороне сервера Web-приложение выполняется специальным программным обеспечением (Web-сервером), который и принимает запросы клиентов, обрабатывает их, формирует ответ в виде страницы, описанной на языке HTML, и передает его клиенту. Одним из таких Web-серверов является Internet Information Services (IIS) компании Microsoft. Это единственный Web-сервер, который способен выполнять Web-приложения, созданные с использованием технологии ASP.NET.

**Клиент-серверная архитектура** – это архитектура, которая подразумевает две компоненты: клиент и сервер. Клиент является инициатором соединения.

**Архитектура веб-приложения**

Есть клиент и есть сервер – две части одной и той же программы, которые взаимодействуют между собой по протоколу HTTP.

Бывают случаи, когда один клиент работает с несколькими серверами. Может быть, когда несколько клиентов работают с одним сервером. Может быть, когда сервер обращается к другому серверу и первый сервер выступает в качестве клиента по отношению к другому серверу.

Между клиентом и сервером ходит 2 типа сообщений:

· от клиента к серверу – **request**

· от сервера к клиенту – **response**

**Request**:

ü метод (get post put)

ü URI (описывает месторасположение сервера)

ü версия протокола (HTTP/1.1)

ü заголовки (пары: имя/значение)

ü параметры (пара: имя/значение)

ü расширение (тело). Может быть в post, редко бывает в get

Request – методы: options, get, head, post, put, delete, trace, connect.

Все заголовки могут быть 2 основных типов:

Ø стандартные (прописаны в протоколе HTTP)

Ø пользовательские (рекомендация: они должны начинаться с буквы «x»)

**Response**:

ü версия протокола (HTTP/1.1)

ü код состояния – статус ответа (1xx, 2xx, 3xx, 4xx, 5xx)

ü пояснение к коду состояния – статусу ответа

ü заголовки (пары: имя/значение)

ü расширение

Все стандартные заголовки, которые прописываются в протоколе HTTP, делятся на 4:

– general

– request (используются только в запросах)

– response

– entity (для сущности в ответах и запросах)

В ответе может быть код состояния.

Все коды делятся на 5 групп:

1. информационные 1xx

2. успешное выполнение запроса и ответ 2xx

3. переадресация 3xx

4. ошибка клиента 4xx

5. ошибка сервера 5xx

**URI (uniform resource identifier)** – унифицированный идентификатор ресурса (документ, изображение, файл, служба, электронная почта).

**URL** – унифицированный локатор ресурса - URI, содержащий местонахождение ресурса и способ обращения к ресурсу.

**URN** – унифицированное имя ресурса – URI, не содержащее в себе месторасположение и метод доступа к ресурсу.

**PURPL** (persistent uniform resource locator) – постоянный унифицированный локатор ресурса. Доступ к конечному ресурсу через redirect. Это некоторая база данных, содержащая месторасположение и способ доступа к ресурсу.

**HTTP-протокол** – протокол прикладного уровня, ассиметричный (сообщения от клиента к серверу и от сервера к клиенту разные). Протокол всегда на каждый запрос должен получить один ответ. Всегда подразумевает пару request/response. Относится к протоколу, который не помнит своего состояния. Жизненный цикл протокола – запрос-ответ. Каждый запрос-ответ – новый жизненный цикл HTTP (stat less протокол). HTTP по отношению к TCP представляет собой формат сообщений. Для HHTP TCP является транспортом.

**Клиент: веб-браузер**

Умеет интерпретировать программу, которую ему присылает сервер.

Веб-браузер умеет генерировать HTTP-запросы:

– если есть адресная строка, и он делает запрос

– есть ряд HTML-тегов, при интерпретации которых браузер делает запрос (form, a, img, script, link, audio, video)

– объект веб-браузера: XMLHTTPRequest

– JS API

Любой веб-браузер характеризуется моделью DOM. Браузер в себе содержит web engine (ядро браузера). На сегодняшний момент существует набор веб-движков, на базе которых работают браузеры. Стандарты веб-браузеров: HTML5, CSS3, SVG, JavaScript (ECMAScript5, ECMA-262-6, ECMA-262-7).

**Общие принципы построения веб-приложений**

– веб-ресурсы приложения

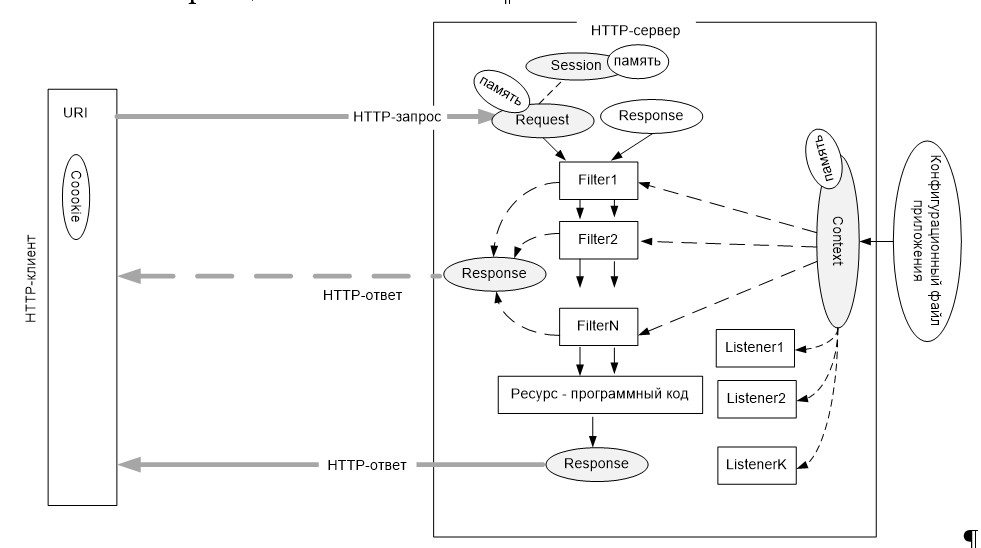
– запросы и ответы

– фильтры

– кэш (данных и вывода)

– слушатели событий

– принципы безопасности



**HTTP–сервер** – программа, которая слушает некоторый порт на своем компьютере.

**Порт** – номер приложения, которому адресован этот запрос. Понятие порт существует на уровне tsp-протокола. Порт слушает входящие сообщения.

**Веб-приложение** принимает заполненный request ответ, пустой ответ response, заполняет response и отдает его серверу.

**Сессия** – серверный объект, хранящий информацию о соединении с клиентом, создается при первом обращении. Время жизни: **timeout** (системный параметр, обычно равен 10-30 минут) – максимальное время между запросами клиента.

Если timeout превышен, то session разрушается и при последующем запросе создается новый экземпляр. Каждая сессия имеет собственный идентификатор (Session ID, 128 или больше бит (16 байт)). Каждый request принадлежит какой-то сессии. Сессия характеризуется двумя параметрами: timeout и session ID.

Обычно в request сервер записывает либо идентификатор этой сессии, либо просто программную ссылку на этот объект сессии.

*Если с одним и тем же сервером работает много клиентов, у каждого из них своя сессия.*

**Куки** – это порция информации, которая может быть сохранена на стороне клиента по инициативе сервера. Когда клиент делает первый запрос, сервер проверяет, есть ли у него заголовок с именем куки. Если этого заголовка нет, то он считает, что это первый запрос и для него создается сессия.

Клиент должен хранить информацию о том, в рамках какой сессии мы отправляем эти запросы. Браузер в себе сохраняет либо файл куки, но чаще сохраняет **localstorage** (поддерживается на стороне браузера).

**Протокол RFC 6265** – стандарт, описывающий механизм куки.

**Конфигурационный файл** (обычно xml) содержит в себе некоторые статические характеристики приложения. Он служит для создания контекста веб-приложения.

**Контекст веб-приложения** – системный объект общий для всех сессий. Предназначен для хранений информации об одном веб-приложений, общий для всех сессий. Как правило, формируется сразу при загрузке веб-сервера.

**Фильтр** – серверный объект, препроцессор запроса, предназначенный для предварительно обработки объекта request. К одному ресурсу может быть построена цепочка фильтров, последний в цепочке – ресурс. Фильтр может прервать цепочку и сам сформировать ответ клиенту.

Информация о цепочке фильтров и привязке этих фильтров к ресурсу хранится в контексте.

Фильтр может не пустить запрос к ресурсу. Он может сам ответить, не пуская его дальше: к следующему фильтру или ресурсу.

**Слушатели событий (Listener)** – серверные объекты для обработки событий жизненного цикла веб-приложения. С помощью Listener можно врезаться в ход выполнения приложения и что-то там изменит.

*Пример – событие создания контекста. Можно написать listener, который выполнится, когда создастся контекст. Это необходимо, например, когда надо записать динамические данные в контексте. Или может быть listener на создание сессии.*

**Кэш** – это системный объект, предназначенный для хранения данных в оперативной памяти с целью ускорения работы веб-приложения. Часто в кэше запоминаются response.

Для каждого запроса создается новый экземпляр приложения. Общим остается контекст. В рамках серии запросов остается сессия. На каждый новый запрос создается новый экземпляр приложения.

**Фильтры** загружены всегда, вместе с контекстом. Фильтры привязываются к URI.

**Постоянное соединение**. Использование одного TCP-соединения для многократных пар запрос-сервер вместо последовательного открытия новых соединений для каждой пары запрос-ответ. Клиент может запросить постоянное соединение с помощью заголовка *Connection: Keep-Alive*, сервер подтверждает заголовком Connection: Keep-Alive.

**Обычный запрос проходит несколько стадий**:

1. Открытие соединения.

2. Отправка запроса.

3. Получение ответа.

4. Закрытие соединения.

**Конвейерная обработка** (**HTTP pipelining**). Редко поддерживаются серверами.

**Пул соединение с базой данных**. Несколько предварительно и постоянно открытых соединений с сервером СУБД, которые используют приложения. Выбор подключения из пула по open, возврат в пул close. Если все подключения пула заняты, запрос на соединение ставиться в очередь.

**Пул соединение** – два постоянно открытых коннекта в базе данных. К этому пулу устанавливается очередь запросов. Можно регулировать количество этих соединений в зависимости от нагрузки.

Создается несколько соединений, к пулу отправляется sql-запрос, который пул перенаправляет к БД.

**Кэширование на стороне браузера**: управление заголовками. Кеширование на стороне браузера описано в протоколе 7234. Если запрос обыкновенный, без кэширования, то при каждом новом запросе эта картинка будет скачиваться. Если картинка кэшируется, то все сохраняется на стороне браузера и запрос получается легковесным.

**Краткий вариант объяснения**

У нас есть объекты **response**, **request**. Они всегда ходят парой. Время их жизни совпадает. У нас есть **ресурс** - это наш **программный код**, который генерирует нам ответ. В объекте request у нас есть **память**, которой программист может воспользоваться. Также у нас есть объект **session**. В нем тоже есть память. Есть **ID** сессии и **timeout** - время жизни. Сессия живет один сеанс. Очень важно, чтобы были **cookies**, иначе работать не будет. Сессия связана с нашим запросом.

У нас есть **конфигурационный файл веб-приложения**, который содержит все параметры приложения. На базе этого файла создается контекст приложения.

**Контекст веб-приложения** - это тоже системный объект (как сессия, запрос или ответ). Он общий для всех сессий! Он хранит информацию о веб-приложении. В нем соответственно своя память, которую программист может тоже использовать.

**Фильтр** - еще один системный объект, он нужен для предварительной обработки запросов и ответов. Он решает куда отдать твой запрос дальше. Может быть несколько фильтров и они будут друг другу их "перефутболивать", как говорил Смелов. Фильтр может вообще не отдать твой запрос на ресурс и, грубо говоря, сформировать ответ сам. Например, это используется для шифрования или расшифрования информации. Информация об этих всех фильтрах (**цепочке**) содержится в контексте.

**Слушатели событий** (листенер) - еще одни специальный объект на стороне сервера, используются для обработки событий жизненного цикла приложения. То есть, мы можем врезаться в ход выполнения нашего приложения. (например, мы это можем делать в файле global.asax). Жизненные циклы это: инит, лоад, прелоадер, анлоад, диспоуз.

**Кэш** - еще один объект. Есть специальная программа кэш-менеджер, которая управляет объектом кэш.

**Как это работает**

Мы отправляем http-запрос от клиента к серверу. Общий вид запроса - битовая последовательность. Http-сервер состоит из http-драйвера и ресурс-программный код. Драйвер первый принимает http-запрос. Он формирует 2 серверных объекта: реквест и респонс (пустой). Далее проверяется это новая сессия или нет. Дальше кэш-менеджер смотрит, надо ли отправлять на ресурс или нет. Допустим нет, тогда из кэша берет данные и назад их отсылает. А если в кэше нет данных, то идет на фильтр. Фильтр может отослать на другой фильтр (тогда будет цепочка) или на ресурс, а может решить, что не будет никуда отсылать.

Так же у нас есть контекст. Он хранит информацию. Используем его, если нужно что-то сохранить во всех сессиях! Он общий для всех сессий. Листенеры - они нужны, если нам нужная какая-то обработка дополнительная в жизненном цикле нашего приложения.

Дальше request + response переходят на программу http-сервера и в этой программе есть http-обработчики (хэндлеры) и эти обработчики принимают request и response. Из request программа, которая находится на сервере понимает, что именно клиент хочет получить от нее, эта программа заполняет response.

Обработчик формирует нам response и шлет обратно на http-драйвер. Http-драйвер преобразовывает в битовую последовательность и отправляет клиенту.

Клиент получает ответ в виде битовой последовательности, преобразовывает в ответ, который понятен для него - html-разметка. Действия повторяются несколько раз. На 1 запрос - 1 ответ. Так как много клиентов можно быть, а серверов меньше, то чтобы не перегружать сервер он удаляет все экземпляры, которые создаются при запросе.

**Таким образом у нас объекты**

1. Реквест

2. Респонс

3. Сессион

4. Куки с указателем на сессию на СЕРВЕРЕ

5. Контекст (кот. создаётся на базе конфигурационного файла)

6. Фильтр

7. Кэш с программой кэш менеджер

8. Http-драйвер

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

**IIS** – множество серверов.

**Публикация приложения** – разворачивание /\*перенос\*/ приложения на iis. Когда настраиваю узел в **IIS**, указываю **характеристики**: порт для узла, название самого узла, физический путь расположения самого приложения, протокол http/https, ip адрес (какой слушаем: конкретный или \* (на одном компе мб установлены несколько ip, несколько сетевых карт))

Мы должны знать, что любой http-сервер является tcp-севером. Http можно рассматривать как формат и правила передачи данных по tcp-соединении. Т.е. фактически устанавливается tcp-соединение.

Набираем адресную строку в браузере, нажимаем Enter и у нас отправляется запрос. Разрешается символическое имя - узнаем из uri (схема/протокол, символическое имя компьютера) ip компьютера. Несколько способов узнать по символическому имени компа его ip (dns (в сетевых настройках мы указываем ip dns), файл hosts, gethostbyname()). Идет запрос на сторону сервера, попадает на конкретный компьютер по ip-адресу. В рамках этого компа, запрос поступает на конкретный сервер, который находится в рамках iis (т.е. наше приложение – это сервер, который находится в рамках iis)

Он создает сокет, который принимает запрос, загружает dll в оперативную память, по web.config узнал handler, создает объект класса, вызывает ProcessRequest и передает туда контекст (IIS создал объекты Request, Response).

На IIS сбрасывать dll, web.config Http-handler можно увидеть во вкладке **Сопоставление обработчиков.**

**Пул приложений –** это окружение нашего приложения, здесь указывается версия .Net. Используется для управления потоками iis.

С помощью **IIS Manager** можно создавать **виртуальные каталоги** веб-приложений ASP.NET. Виртуальный каталог отображается в клиентских обозревателях так, как если бы он содержался в корневом каталоге веб-сервера, даже если он физически находится в другом месте. Этот подход позволяет публиковать веб-содержимое, которое не находится в корневой папке веб-сервера, например, содержимое, находящееся на удаленном компьютере. Это также удобный способ для настройки узла для локальной веб-разработки, поскольку она не требует уникального веб-узла для каждого виртуального каталога.

Необходимо создавать виртуальный каталог как часть существующего веб-узла IIS. Это может быть либо веб-узел по умолчанию, который создается при установке IIS, либо созданный пользователем веб-узел.

**Создание виртуального каталога:**

В IIS Manager откройте папку Веб-узлы узла «Локальный компьютер». Затем выберите веб-узел, для которого необходимо создать виртуальный каталог. Выполните следующие действия:

Щелкните правой кнопкой мыши узел или папку, в которой требуется создать виртуальный каталог, затем выберите команду добавить виртуальный каталог.

В диалоговом окне Добавление виртуального каталога укажите следующие сведения:

1. Псевдоним. Введите имя виртуального каталога.

2. Физический путь. Введите путь или перейдите к физическому каталогу, содержащему виртуальный каталог. Для размещения содержимого виртуального каталога можно выбрать существующую папку или создать новую.

Чтобы предоставить учетные данные для подключения к UNC-пути, нажмите кнопку Подключиться как. Нажмите кнопку ОК.

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

**Http-handle**r – простейшее приложение Asp.Net. Физически представляет собой dll. С точки зрения разработчика: класс имплементирующий интерфейс IHttpHandler.

Есть ещё следующие приложения: web-формы, mvc, web-api, http-handler

**Со́кет**— название программного интерфейса для обеспечения обмена данными между процессами.

**Http-handler** представляет собой класс, который реализует (имплементирует) интерфейс IHttpHandler. Реализует метод ProcessRequest (HttpContext – контекст http запроса содержит request (http version, method, uri, parameters (in uri or body), headers, body) и response (http version, status code, description of status, headers, body); его вызывает IIS) и свойство IsReusable ( = true, переиспользование, будет использоваться один и тот же экземпляр handler’а. = false, при каждом запросе создается новый экземпляр handler’а для обработки этого запроса)

Как проверить, что IsReusable = true работает, т.е. используется один и тот же объект обработчика? Создать переменную и в конструкторе увеличивать ее и проверять значение. ИЛИ в конструкторе выводит сообщение.

**Интерфейс** – это поименованный набор сигнатур

**ISReusable** – для каждого запроса используется один и тот же экземпляр httphandlerа (true).(false) новый экземпляр http-handler для каждого запроса.

**Websocket** – протокол передачи данных, который существует поверх tcp. Имеет дуплексный канал связи.

Tcp находится на транспортном уровне, а websocket на прикладном.

В модели tcp/ip под tcp находится ip, а под ip – Ethernet. Связь http и websocket – соединение осуществляется с помощью протокола http.

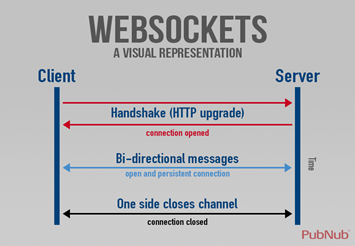
**Как происходит «рукопожатие»:**

- клиент посылает http-запрос серверу с заголовками Connection: Upgrade и Upgrade: websocket

- если сервер поддерживает протокол WebSocket, он согласится перейти на него и сообщит об этом в заголовке ответа Upgrade. Заголовки Connection: Upgrade и Upgrade: WebSocket

- после этого вызывается событие open в экземпляре WebSocket на клиенте

- теперь, после завершения фазы рукопожатия, исходное HTTP-соединение заменяется на WebSocket-соединение, которое использует то же самое базовое TCP/IP-соединение. В этот момент и клиент и сервер могут приступать к отправке данных.



ws – незашифрованное соединение

wss – зашифрованное

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

При поступлении сообщения от сервера вызывается обработчик события onmessage, который выполняет действия в рамках клиентской части. Т.е. с помощью протокола WS появляется возможность обрабатывать серверные события на стороне клиента, не выполняя запросов к серверу.

Свойство WebSocketRequest определяет является ли пришедший запрос запросом на соединение по протоколу WebSocets.

Метод ReciveAsync имеет 2 параметра – буфер для получаемых данных и маркер отмены.

**IIS** – множество серверов. Он создает сокет, который принимает запрос, загружает dll в оперативную память, по web.config узнал handler, создает объект класса, вызывает ProcessRequest и передает туда контекст (IIS создал объекты Request, Response).

В приложении не забыть!!!

1. Способ вызова http-обработчика описывается в конфигурационном файле

<handlers>

<add name="websockets" path="\*.websocket" verb="GET" type="IISHandler"/>

</handlers>

##### 

##### **10. ASP.NET: ASMX-сервисы, WSDL, SOAP, прокси, порядок разработки, принципы применения. Пример.**

**Web-сервис –** это приложение, предоставляющее открытый интерфейс, пригодный для использования другими приложениями в Web.

Сервисы можно разделить на *RPC и Rest.*

***RPC подход в веб-сервисам*** – предполагается, что интерфейс с веб-сервисом нам представляется в виде удаленного объекта, у которого есть доступные методы, которые мы можем вызывать.

***REST***– появилась раньше для доступа к ресурсам. ***REST-сервисы*** представляются как набор uri, к которым мы можем делать 4 вида запросов (get, post, put, delete), и правила взаимодействия с этим набором uri.

**ASMX-сервис**

Обеспечивает способ взаимодействия RPC. Когда говорим об этих сервисах, то всегда подразумеваем 4 технологии, на которых основывается этот веб-сервис *XML, SOAP, WSDL, UDDI*

**XML** – язык разметки, с помощью которого мы можем создавать свои собственные языки. Две других технологии полностью основываются на XML: SOAP, WSDL

**SOAP** (Simple Object Access Protocol) – протокол простого доступа к объекту. Протокол обмена информации между клиентом и сервером в формате XML, используется в ASMX-сервисах и WCF-сервисах. SOAP может перемещаться в разных протоколах (HTTP, TCP, NamedPipe). В сервисах SOAP использует HTTP в качестве транспорта – это значит, что SOAP-сообщение будет помещено в тело запроса и ответа, и будем использовать специальные заголовки **Content-Type: application/xml+soap**

**Структура SOAP запроса состоит из 3 основных компонент:**

· header(служебная информация)

· body(процедуры,параметры)

· внутри body может быть Fault(инфа об ошибках)

Предшественником SOAP является технология XML-RPC – это простая спецификация вызова удаленной RPC-процедуры, использующей XML в качестве кодирования и HTTP в качестве транспорта. Так выглядит SOAP в HTTP:

|  |
| --- |
| <!--пример запроса soap-документа -->  <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope"  xmlns:encodingStyle="http://schemas.xmlsoap.org/soap/encoding"  xmlns:smw ="http://smw2003.smw.net/namespace">  <!--encodingStyle = формат данных в документе -->  <soap:Header soap:actor ="http://smw2003.smw.net/actor">  <!--заголовочная информация -->  <!--информация специфическая для данного приложения -->  <!--soap:actor = получатель информации заголовка -->  <smw:language soap:mustUnderstand="0">ru</smw:language>  <smw:timedate soap:mustUnderstand="1">by</smw:timedate>  <!--soap:mustUnderstand= должен/нет получатель обрабатывать -->  </soap:Header >  <soap:Body xmlns:smw ="http://smw2003.smw.net/namespace">  <smw:GetFoto>  <smw:getfotoreq>2772</smw:getfotoreq>  </smw:GetFoto>  </soap:Body>  <soap:Fault>  <!--информация об ошибках приложения -->  <!--используется только в ответных сообщениях -->  </soap:Fault>  </soap:Envelope> |

Рис.Пример soap-запроса

**Что здесь подразумевается?** У нас на клиенте есть программа, которая вызывает удаленную процедуру с именем GetFoto и передает параметр 2772. На сервер приходит этот запрос, процедура отрабатывает и отправляет ответ, в котором она возвращает строку<http://smw2003.smw.net/Foto?ID=2772>. Это означает мб, что фото лежит по такой ссылке

|  |
| --- |
| <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope"  xmlns:encodingStyle="http://schemas.xmlsoap.org/soap/encoding"  xmlns:smw ="http://smw2003.smw.net/namespace">  <soap:Header soap:actor ="http://smw2003.smw.net/authentification">  <smw:key soap:mustUnderstand="1">ABCEGH2288</smw:key>  </soap:Header>  <soap:Body xmlns:smw ="http://smw2003.smw.net/namespace">  <smw:GetFoto>  <smw:getfotores>http://smw2003.smw.net/Foto?ID=2772</smw:getfotores>  </smw:GetFoto>  </soap:Body>  </soap:Envelope> |

Рис.Пример soap-ответа

**WSDL** (Web Services Description Language) – язык описания Web-сервисов. Помогает так описать веб-сервисы, чтобы можно было обратиться к сервису с запросами и знать, какого формата ответы мы получим.

WSDL – это реализация XML, имеет очень много комментов документации (для чего, кому принадлежит). Позволяет создать файлики на XML, которые описывают сервисы. Имея такой wsdl мы имеем исчерпывающую информацию о том, как работать с этим сервисом.

|  |
| --- |
| <definition>  <types>  <!-- оперделение типов, используемых web-сервисом -->  <!-- для пл.-независимого WSDL используется синтаксис XML Schema -->  </types>  <message>  <!-- сообщения, используемые web-сервисом -->  <!-- сообщений, может быть несколько -->  <!-- каждое сообщение может состоять из нескольких частей -->  </message>  <portType>  <!-- методы, предоставляемые web-сервисом -->  <!-- может быть несколько портов -->  <!-- определены операции web-сервиса и используемые сообщения -->  </portType>  <binding>  <!-- протоколы связи, используемые web-сервисом -->  <!-- форматы сообщений и детали протокола для каждого порта -->  </binding>  </definition> |

Рис. Структура WSDL-документа

***Definition*** - описание

***Types*** – знает фундаментальные (стринг и т.д), но можно создавать и комплексные (тип класс, состоящий из полей с фундаментальными типами) (тип данных параметра)

***Message*** – типы сообщений, которые ходят между клиентом и сервером, содержат описанные выше типы данных

***portType*** – описываются методы удаленного сервера, которые могут принимать сообщения, которые описаны выше и отправлять эти сообщения

***binding*** – та часть документа, которая связывает soap-протокол с протоколом транспорта (http)

|  |
| --- |
| <definition> <!-- ....................... -->  <message name ="GetDossierRequest">  <part name ="ID" type="xs:string" />  </message>  <message name ="GetDossierResponse">  <part name ="Dossier" type="xs:string" />  </message>  <portType name ="Dossier">  <operation name="GetDossier">  <input message = "GetDossierRequest"/>  <output message ="GetDossierResponse"/>  </operation>  </portType> <!-- ....................... -->  </definition> |

Рис. Фрагмент WSDL-документа

**UDDI** (Universal Description Discovery & Integration) стандарт хранения WSDL–файлов для поиска другими организациями для поиска и внедрения в свои организации. *Эта технология подразумевает создание стандартных хранилищ/каталогов wsdl-описаний. Предполагалось, что мы будем создавать каталоги, которые будут содержать описание сервисов.*

**ASMX-сервис** – это упрощенный веб-сервис, который основывается на SOAP протоколе.

***WCF*** (Windows Communication Foundation) – отдельная платформа, предназначенная для разработки приложений, имеющих сервис-ориентированную архитектуру. На этой платформе можно разрабатывать разного вида приложения/сервисы, в том числе точно такие же сервисы как ASMX.

Можно сказать, что сервисы ASMX являются *частным случаем* WCF-сервисов, НО эта технология довольно проста и удобна в использовании, поэтому Microsoft ее оставил отдельно.

Если мы разработаем ASMX-сервис, то он обязательно подлежит публикации на iis-сервере.

Отличительные черты ASMX-сервиса:

- используем soap-протокол

- http в качестве транспорта

- он должен исполняться с iis сервера

При создании сервиса в VS создается класс/сервис представляет собой класс, который наследуется от System.Web.Services.**WebService.** Также класс имеет аннотации:

- [WebService(Namespace = “http://tempuri.org/”)] – пространство имен, будет сопровождать данные в soap-запросах и ответах, чтобы мы знали, что они относятся к данному сервису

- [WebServiceBinding(ConformsTo = WsiProfilies.BasicProfile1\_1)] – спецификация совместимости web-служб. Связь между протоколами, на которых может работать этот сервис, осуществляется через привязки. Мы можем привязки создавать сами. Привязки – это просто набор характеристик и параметров протокола

- [System.ComponentModel.ToolboxItem(false)] – управление панелью элементов VS ToolBox, можно не указывать

Внутри класса есть методы. Помечаем их, что это методы сервиса (чтобы можно было вызывать их извне) *[WebMethod]*, можно указать описание [WebMethod(Description = “blaaaaaaaa”)]

Мы можем сразу запустить наш сервис. **Но что происходит, когда мы нажимаем на запуск?** Создается сервис – это dll, которая имеет свой web.config, также есть специальная страница asmx. И эта dll должна быть опубликована на iis. Запускается внутренний iis-сервер, на который публикуется наша dll и сервис готов для работы. После этого мы делаем запрос на localhost, автоматически создается узел. Сразу запускается браузер, который делает запрос к узлу, далее мы получаем страницу с инфой о сервисе. Доступность этой страницы конфигурируется web.config’ом.

Ответ приходит коротким, весь envelop (см. пример soap-ответа выше) не передается.

Сервис не сохраняет своего состояния по умолчанию, однако мы можем установить флаг для поддержания сессии [WebMethod(Description = “blaaaaaaaa”, **EnableSession = true**)]. **Сессия** – системный объект, который создается для соединений и хранит информацию о соединении.

! На клиенте обязательно должен быть установлен параметр в web.config, чтобы поддерживалась сессия.



Мы можем получить WSDL, добавив в адресной строке ?WSDL

**Как можем применять сервис?**

Мы можем применять в любом приложении, который поддерживает этот сервис. Можно разработать клиент на Windows Form приложении, который будет использовать RPC-сервис на основе soap протокола. Необходимо добавить ссылку на службу, где указываем адрес своей службы (служба должна быть запущена). Когда мы указали адрес и нажимаем кнопку перейти, то отправляется запрос на этот адрес с добавленным в конце **?WSDL**. VS обращается к службе и просит ее выслать файл wsdl, который описывает службу. Именно из этого wsdl-файла оно нам выбирает и показывает методы, которые отображаются в окне Добавления ссылки на службу. Также в этом окне мы указываем пространство имен, нажимаем ОК.

**Что происходит после нажатия на ОК?** VS на основе wsdl файла генерирует исходный код proxy-класса. **Proxy-класс** – это класс, эмитирует работу удаленного сервиса, имеет те же методы, что и удаленный объект. И мы с ним общаемся, как с обыкновенным локальным классом, мы создаем объект этого класса, вызываем метод этого класса, передаем параметры в метод и получаем результат.

**Что же делают эти методы?** Они принимают параметры, создают xml envelop’ы, туда заталкивают всю необходимую информацию для запроса, отправляют это на сторону службы, служба обрабатывает этот запрос и посылает обратно ответ со своим envelop’ом, в котором содержится результат выполнения. Наш прокси-класс парсит этот ответ и возвращает нам результат. Т.е. прокси-класс наш заместитель, который притворяется, что он сервис, и мы с ним работаем как с сервисом.

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

Чтобы создать asmx сервис в VS2019, выбираем *Веб-приложение ASP.NET (.NET Framework).* Далее выберите *Пустой шаблон* и добавьте ПКМ элемент *Веб-службы (ASMX).*

##### **11. ASP.NET: MVC-приложение, структура MVC-приложения, назначение основных компонентов приложения, маршрутизация. Пример.**

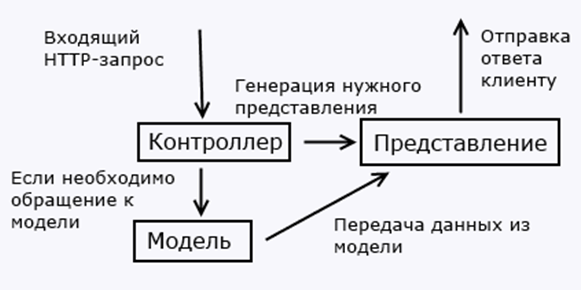
**MVC** (**Model View Controller**) — специальный тип приложений, которое должно состоять из 3 частей:

1. Модель

2. Представление

3. Котроллер

Но на самом деле, частей 4. Еще одна — **маршрутизатор**.

****

**Модель** – описывает логику всего приложения (то с чем мы будем работать).

**Представление** – то, что мы видит пользователь в виде html-разметки.

**Контроллер** – самый первый принимает данные от пользователя и обрабатывает их и открывает представление. все действия, которые мы будем делать с нашей веб-страницей прописаны в контроллере.

**Маршрутизатор** – прописываем наш путь для того чтобы открыть контроллер и стандартный вью (наше приложение стартует согласно тому, что написано в маршрутизаторе).

Если нам нужно написать приложение, которое будет состоять из 100 форм и которое в будущем будет дорабатываться и расширяться, то в этом случае нужно использовать MVC.

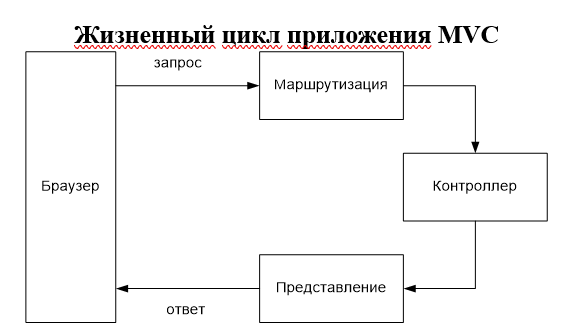
Этот шаблон заставляет нас писать строже, т.е. он нас ограничивает в некоторых возможностях, этим самым приложение получается более строже, легко-расширяемое и удобно-тестируемое.

В MVC нет элементов управления и нет view state — элемента, сохраняющего состояние. Это и преимущество, и недостаток.

Трафик между клиентом и сервером будет поменьше, потому что нет viewstate, а недостаток: нам придется самим заниматься вопросами сохранения состояния, т.е. на сервере что-то нужно написать, чтобы сохранять состояние.

**Жизненный цикл приложения MVC**

От клиента (браузер) поступает **запрос** на сторону сервера и его встречает **маршрутизатор**. Маршрутизатор анализирует **URI**, который у нас есть в запросе и на основе этого URI создает **объект Контроллера**, которому передает параметры, которые есть в этом запросе.



**Контроллер отрабатывает** и в большинстве случаев, результатом выполнения Контроллера является сформированное им **Представление** (View).

**Представление** — динамически формируемая разметка, которая идет на сторону браузера.

А где **модель**? Контроллер будет работать с моделью. Он будет работать, например, с **бд**, или чем-то другим и для этого у него будут использоваться некоторые объекты классов, которые будут представлять **данные**. Как частный случай, используется для этого **Entity Framework**.

Метод в терминологии MVC называется **акция** (action).

Какой контроллер выбрать и какую акцию для этого контроллера вызвать решает маршрутизатор, а решает он на основе uri, анализирует его и решает насчет контроллера, а потом насчет акции, передает необходимые параметры.

В файле **RouteConfig.cs** находится таблица маршрутизации. С помощью этой таблицы мы можем с вами повлиять на действие маршрутизатора, указав ему, как интерпретировать uri. Для этого в таблице маршрутизации создаются шаблоны (templates), которые говоря нам как интерпретировать uri.

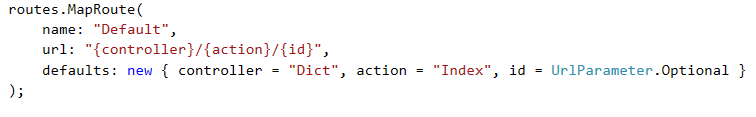
##### 

##### **12. ASP..NET: MVC-приложение, маршрутизатор, принципы устройства и работы. Пример.**

**Маршрутизатор** – прописываем наш путь для того, чтобы открыть контроллер и стандартный вью (наше приложение стартует согласно тому, что написано в маршрутизаторе).

От клиента (браузер) поступает запрос на сторону сервера и его встречает маршрутизатор. Маршрутизатор анализирует URI, который у нас есть в запросе и на основе этого URI создает объект Контроллера, которому передает параметры, которые есть в этом запросе.

В файле **RouteConfig.cs** находится таблица маршрутизации. С помощью этой таблицы мы можем с вами повлиять на действие маршрутизатора, указав ему, как интерпретировать uri. Для этого в таблице маршрутизации создаются шаблоны (templates), которые говорят нам как интерпретировать uri.

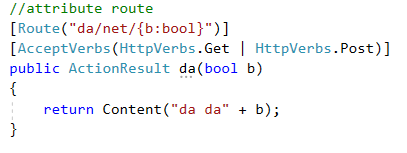


В url задается шаблон: в нем написано, что сразу за именем компьютера идет /, а после него уже идет имя контроллера, после второго / находится акция в этом контроллере. После третьего / находится id, он интерпретируется как параметр, который будет передавать в эту акцию.

В конце стоит default. В случае, если мы не указываем никакой контроллер, то будет стоять тот контроллер, который указан в этой строчке по дефолту.

Атрибуты маршрутизации позволяют более гибко маршрутизировать экшны и контроллеры в приложении. Для того, чтобы была возможность применять атрибуты маршрутизации в RegisterRoutes() указываем routes.MapMvcAttributwRoutes(); перед строками таблицы маршрутизации.

Примеры: [Route(“{x:int}/{name:alpha}”)], [RoutePrefix(“bstu”)]

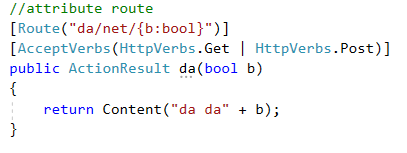


Можно создать пользовательский обработчик маршрутов, но для этого необходимо написать класс, который будет имплементировать интерфейс IRouteHandler. Реализовываем его с помощью метода GetHttpHandler, который должен возвращать хэндлер (обработчик, класс, реализующий интерфейс IHttpHandler (IsReusable, ProcessRequest()))

**13. ASP.NET: MVC-приложение, маршрутизация с помощью атрибутов, констрейны маршрутизации, принципы работы. Пример.**

Атрибуты маршрутизации позволяют более гибко маршрутизировать экшны и контроллеры в приложении. Для того, чтобы была возможность применять атрибуты маршрутизации в RegisterRoutes() указываем routes.MapMvcAttributwRoutes(); перед строками таблицы машрутизации.

Примеры: [Route(“{x:int}/{name:alpha}”)], [RoutePrefix(“bstu”)]



Можно создать пользовательский обработчик маршрутов, но для этого необходимо написать класс, который будет имплементировать интерфейс IRouteHandler. Реализовываем его с помощью метода GetHttpHandler, который должен возвращать хэндлер (обработчик, класс, реализующий интерфейс IHttpHandler (IsReusable, ProcessRequest()))

##### **14. ASP.NET: MVC-приложение, котроллер, жизненный цикл контроллера, взаимодействие с моделью и представлениями. Пример.**

**Контроллер** – класс, производный от базового класса Controller. Содержит открытые методы, называемые методами Action.

Если в акции контроллера не указан параметр, то вызывается то представление, которое имеет такое же название, что и акция. Акция Index вызовет представление Index и т.д. Так в 90% случаях.

**Представление** — разметка, которая находится в папке Views в подпапке, которая соответствует имени контроллера:

**Исключением является папка Shared.** Не допускается контроллер с именем Shared. Потому что в этой папке используется views, которые используются в нескольких контроллерах. Если в акции вызываем View, то это view будет описано следующим образом: сначала она будет описываться в папке, которая соответствует этому контроллеру, а потом, если он его не найдет, то будет искать это представление в папке Shared.

**Контроллер** из себя представляет набор экшнов (методов, которые в конце концов вызываются).

Контроллер в рамках своей работы взаимодействует с моделью.

Смелик любит задавать вопрос: **как контроллер** **взаимодействует с моделью?**

Он ждёт такой ответ: в рамках контроллера создаётся объект модели (объект ContextDB, если говорим про entity).

Модель создается для каждого запроса заново. То есть для каждого запроса создаётся новый экземпляр контроллера и контроллер создаёт новый экземпляр модели для обработки этого запроса. Наше приложение по большому счёту живет в рамках одного запроса.

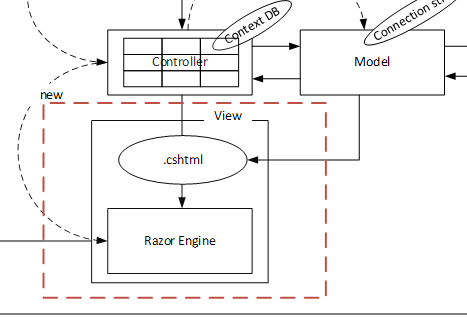
**То есть контроллер живет ровно один запрос!**

И работа контроллера, как правило, не всегда заканчивается тем, что создаётся представление.

##### 

##### **15. ASP.NET: MVC-приложение, представление, Razor Engine, жизненный цикл представления. Пример.**

**Представление** - с# объект, который создаётся с помощью view engine (специальный механизм, который превращает файл (cshtml) в объект с#, и это происходит один раз, при первом обращении к этому view, то есть при таких последующих обращениях уже view engine не нужен и сразу уже работает объект, который образовался на основе этого cshtml файла).



В нашем случае view engine - razor engine (он в данном случае и предназначен для генерации вот такого вот объекта). Основная задача этого объекта - сгенерировать response, который потом уходит на сторону клиента

**ViewData**

1. ViewData — это словарный объект, производный от TempViewDataDictionary;
2. Используется для передачи данных из контроллера в соответствующее представление;
3. Жизненный цикл ограничен текущим запросом;
4. Если происходит redirect, значение ViewData превращается в null;
5. Также необходимо осуществлять приведение типов и проверять на null, чтобы избежать ошибок;

**ViewBag**

1. ViewBag — динамическое свойство, представляет собой обертку вокруг ViewData , и также используется для передачи данных из контроллера в соответствующее представление;
2. Жизненный цикл также ограничивается текущий запросом;
3. При редиректе значение также оборачивается в null;
4. Нет необходимости в приведении типов для получения данных

Так же, ViewBag имеет преимущество в том, что можно использовать его свойства Strongly typed-методом, при этом корректность данных будет проверена еще на этапе компиляции/написания кода.

строготипизированные представления – представления в которых используется @model/Model

##### 

##### **16. ASP.NET: MVC-приложение, модель, жизненный цикл модели, репозиторий. Пример.**

**Модель** – описывает логику всего приложения (то с чем мы будем работать).

Контроллер будет работать с моделью. Он будет работать, например, с бд, или чем-то другим и для этого у него будут использоваться некоторые объекты классов, которые будут представлять данные. Как частный случай, используется для этого Entity Framework. Модель работающая с БД называется **репозиторием**

Объект модели создается каждый раз при вызове от контроллера, после умирает

##### **17. ASP.NET: MVC-приложение, внедрение зависимостей. Пример.**

Dependency Injection – внедрение зависимости.

* программный механизм, позволяющий в автоматическом режиме создавать программный объект, с заданными жизненным циклом (задаются события инстансирования и разрушения объекта), способом применения (в качестве параметра метода или конструктора, свойства или поля объекта) и областью действия.
* позволяет создавать слабосвязанные компоненты.
* повторное применение кода, упрощает внесение изменений, упрощает тестирование.
* чаще всего внедряется contextDB или репозиторий модели данных.
* IoC - Inversion of Control, принцип программирования, позволяющий снизить зависимость между компонентами программ; DI – один из способов реализации IoC.
* IoC-контейнеры: Ninject, AutoFac, Unity,….

// InTransientScope() - по умолчанию - новый на каждый вызов

// InSingletonScope() - паттерн Singleton - один на все вызовы

// InThreadScope() - новый экземпляр на каждый поток

// InRequestScope() - новый экземпляр на каждый запрос

##### 

##### **18. ASP.NET: MVC Web API, структура Web API-приложения; назначение основных компонентов приложения, маршрутизация. Пример.**

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

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

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

В общем случае MVC Web API приложение состоит из моделей, представлений и контроллеров типа API.

Модель предоставляет данные и реагирует на команды контроллера, изменяя своё состояние.

Представление отвечает за отображение данных модели пользователю, реагируя на изменения модели.

Контроллеры в Web API принимают запрос в виде объекта HttpRequestMessage, обрабатывают его с помощью одного из методов и посылают в ответ результат обработки в виде объекта HttpResponseMessage.

HttpRequestMessage с помощью своих свойств передает ряд данных о запросе:

Content: возвращает объект HttpContent, который представляет тело запроса

Headers: возвращает набор заголовков запроса в виде объектов HttpRequestHeader  
Method: тип запроса (Get/Post/Put/Delete)

Properties: словарь, который содержит объекты, предоставляемые хостинговой средой  
RequestUri: запрошенный ресурс URL

Version: используемая версия протокола HTTP

За сопоставление запросов с определенными маршрутами, как и в MVC, отвечает система маршрутизации. Ключевым классом для системы маршрутизации является класс HttpRoutingDispatcher, который обрабатывает запрос для получения данных маршрута и добавляет эти данные в коллекцию HttpRequestContext.RouteData.

Определение маршрутов по умолчанию производится в файле WebApiConfig.cs.

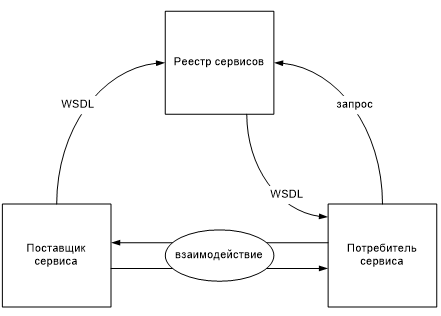
Шаблон позволяет сопоставить запрос с определенным маршрутом. В Web API шаблоны определяются также, как и в MVC. В частности, здесь определен шаблон "api/{controller}/{id}", где параметр id является необязательным. Поэтому данный маршрут будет соответствовать, например, запросу api/values/5, где "values" будет сопоставляться с названием контроллера, а число 5 - с параметром id. Поскольку Web API использует тип запроса HTTP (POST/PUT/GET) для сопоставления с методом контроллера, то мы можем опустить в шаблоне сегмент для метода.

##### **19. WCF-сервисы: WSDL, хост, прокси, модели взаимодействия клиента и сервера, порядок разработки, принципы применения. Пример.**



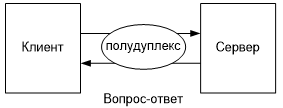
**WCF** (Windows Communication Foundation) – технология, предназначенная для разработки приложений, имеющих сервис-ориентированную архитектуру. На этой платформе можно разрабатывать разного вида приложения/сервисы, в том числе точно такие же сервисы как ASMX.

**SOA(Service Oriented Architecture)** – подход к проектированию распределенных приложений, при котором приложение строится из нескольких автономных сервисов, работающих совместно, обменивающихся сообщениями через границы сетевых машин или процессов с помощью четко определенных интерфейсов. (НАБОР СЕРВИСОВ КОТОРЫЕ НЕ ЗАВИСИМЫ МЕЖДУ СОБОЙ, НО ОБЩАЮТСЯ МЕЖДУ СОБОЙ)



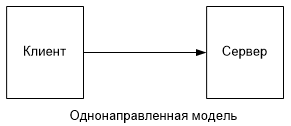
Про WSDL смотри в вопрос 10

WCF более общая технология, которая снимает ограничения ASMX-сервисов. Считается, что ASMX-сервисы – это частный случай WCF. Основное ограничение, которое есть у ASMX – это то, что он использует протокол HTTP и обеспечивает архитектуру, которая основывается на http в том смысле, что у нас всегда запрос-ответ и используется полудуплексный канал. В WCF эти ограничения преодолены и эти сервисы поддерживают следующий **набор моделей взаимодействия между клиентом и сервером**:



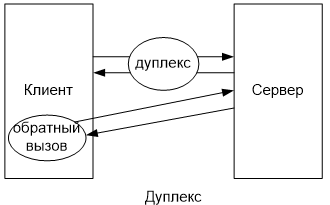
(как и у ASMX)

Если в каждый момент времени мы можем отправлять данные или получать ответы – это полудуплекс. Мы можем вызвать удаленную процедуру и получить ответ. HTTP протокол работает в режиме полудуплекса.



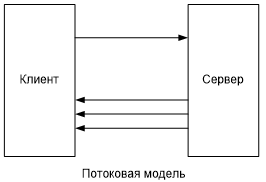
(отправили сообщение и не ждем результат)

Если мы в качестве транспорта используем http-протокол, то ответ в любом случае должен прийти, но мы не будем его ждать и не будем обрабатывать. Но вот если использовать TCP-протокол, то ответа не будет.



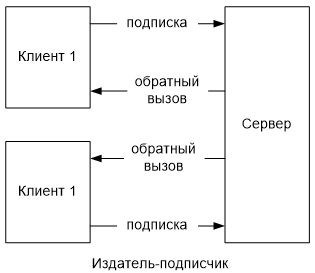
(можем иметь 2 канала связи, которые не зависят друг от друга и связывают клиент и сервер)

**Дуплекс** означает если у нас есть канал и по нему мы можем постоянно гнать вопросы/ответы. **В веб-сокетах** используются дуплексы (если *два канала*: для отправки запроса и получения и можно постоянно отправлять/получать). Два однонаправленных канала. Есть два канала, один от клиента к серверу, второй от сервера к клиенту, и они независимы друг от друга. С помощью полного дуплекса можно положить на клиенте специальную программу обратного вызова, которую вызовет сервер.



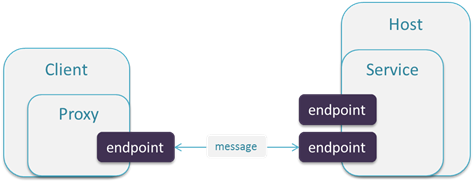
(клиент делает 1 запрос, а сервер посылает несколько ответов, когда работает с большими данными)

**Потоковая модель** означает возможность отправки одного сообщения, а в ответ получения множества сообщений. С помощью HTTP такое реализовать нельзя. Только на TCP. Если мы делаем запрос к БД, а данные у нас очень большие (нужно получить много разных картинок), то в этом случаем делаем запрос и получаем картинки по очереди (множеством ответов), используя потоковую модель.



**«Издатель-подписчик».** Клиент может подписаться на какое-то событие, которое есть на сервере (на сервере есть множество событий), и мы как клиент можем подписаться на какое-либо из событий. И как только событие произойдет будет вызвана функция клиента, на одно и тоже событие может подписаться множество клиентов (и как только событие произойдет, все клиенты будут оповещены сервисом). Например, смена курсы валют запускает событие на сервере, и клиент обновляет там у себя что-нибудь. То есть клиент ничего не спрашивает у сервиса. Сам сервис говорит подписчику.

**Архитектура WCF**



На стороне клиента мб сделан прокси-класс (можно посмотреть вопрос 10 про прокси), а сам сервис исполняется в рамках какого-то хоста. Хост – какое-то приложение, которое встраивает в себя сервис. Если говорить про ASMX, то встраивал в себя (был хостом) asmx-сервис iis. Здесь мы можем встроить wcf-сервис в обыкновенное консольное или WinForms приложение (в любое приложение, которое есть в .Net). Говорят, о том, что мы здесь **можем разрабатывать так называемое self-hosting приложение**, т.е приложение, которое мы запускаем, условно, через командную строку, оно стартует и обеспечивает работу сервиса, с которым может взаимодействовать клиент (без iis). Еще одна особенность, что сервис в **wcf может поддерживать несколько точек доступа** (ASMX может поддерживать только одну точку доступа). Т.е. мы можем обращаться к этому сервису по http протоколу, по tcp протоколу или еще какой-нибудь, это все будут различные точки доступа к сервису. ***Точки доступа обязаны отличаться протоколом доступа*** (**wcf-сервисы поддерживают различные протоколы доступа к этому сервису**). По разным точкам доступа мы можем обеспечить различный сервис, различный набор методов, который обеспечивается этим сервисом.

**Порядок разработки:++++++++++++++++++++**

1. разработка wcf-сервиса (создании dll- библиотеки сервиса wcf)

2. разработка хост

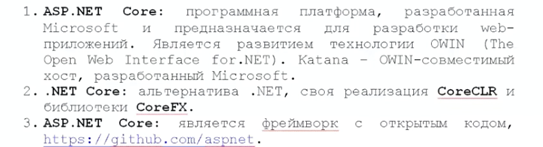
Первым делом создается проект. Сформировать в приложении ссылку на System.ServiceModel. Сформировать в приложении ссылку на dll-библиотеку сервиса WCF. Создать хост (объект типа ServiceHost) и выполнить метод Open. Создать файл конфигурации приложения (App.config). Содержимое App.config формируется на основе App.config сервиса.

3. разработка клиента (может быть любое .NET-приложение)

Запустите хост (сервер) на выполнение и запустите IE. В адресной строке укажите адрес. Создайте новый проект Windows Forms. Сформируйте proxy-ссылку на службу (хост должен быть активным). App.config клиента формируется автоматически.

***2ой вопрос***

##### **20. ASP.NET CORE: программная платформа, принципы работы, архитектура. Пример.**

****

****

**Имеет встроенный веб-сервер Kestrel.**

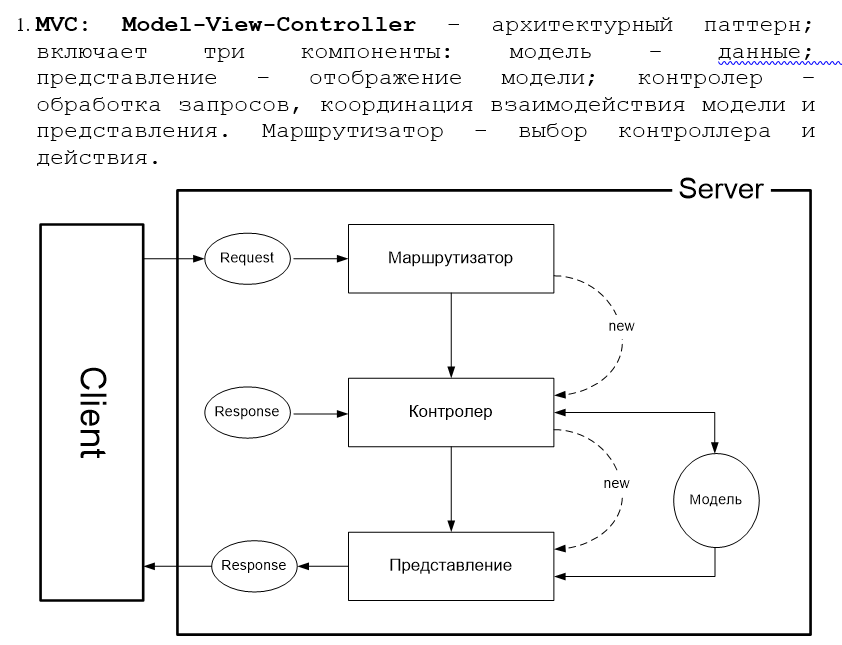
**wwwroot - директорий для хранения статического контекста.**

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

Это всё сделано в примере

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

Тоже много всего в примере, чекайте файл startup

****

##### **Маршруты можно указывать явно с помощью [Route(“...”)] над IActionResult, таких можно указывать несколько.**

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

**Результат действия** - это тот объект, который возвращается методом после обработки запроса.

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

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

2. MVC Controller: ControllerBase не поддерживает view

*reflection-поиск контролера, признаки*

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

*Action – действия: методы контроллера*

- нестатический публичный метод котроллера;

- не является generic;

- не помечен атрибутом [NonAction];

- перегруженные методы могут быть только отмеченные атрибутами [Action], [ActionName], [AcceptVerbs], [HttpGet], [HttpPost];

- ref и out–параметры в действиях не допустимы;

- возвращает IActionResult

**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.

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

***всё есть в примере***

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.

Метод OnActionExecuting вызывается перед вызовом метода действий. А метод OnActionExecuted - после.

Атрибут [HttpPost] указывает механизму маршрутизации отправлять любые POST-запросы этому методу действий одному методу над другим. Это тип перегрузки.

По умолчанию для метода является [HttpGet]. Из-за этого атрибут не требуется.

Параметры можем передать в виде параметров (?x=123) или типа в URI lala/2/2

##### **25. ASP.NET CORE: Фильтры: Action Filter, Result Filter, Authorization Filter, Resource Filter, Exception Filter, пользовательские фильтры действий.**

Фильтры выполняются в конвейере вызова действий ASP.NET Core, который иногда называют конвейером фильтров. Конвейер фильтров запускается после того, как платформа ASP.NET Core выбирает выполняемое действие.

**Фильтры** – механизм автоматического выполнения кода в рамках котроллера; фильтры: действий, результатов, исключений, авторизации. Фильтры могут быть в виде атрибутов акции (действуют для одной акции) или атрибутов контроллера (действуют для всех акций контроллера). Для однородных фильтров может быть установлен приоритет (чем больше, тем выше приоритет).

**Action Filter:** позволяет автоматически выделить методы класса до и после выполнения акции (действия) контроллера.

**Result Filter:** позволяет выполнить методы класса, до и после формирования объекта-результата (IActionResult), возвращаемого акцией.

**Authorization Filter:** проверяет наличие авторизации текущего пользователя.

**Resource Filter:** выполняется после фильтра авторизации и после всех остальных фильтров.

**Exception Filter:** обработка исключений в акциях

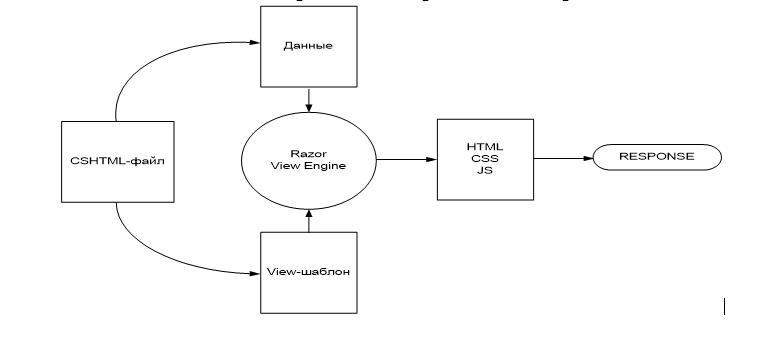
Можно создать свои собственные фильтры действий. Например, для реализации пользовательской системы аутентификации может потребоваться создать пользовательский фильтр действий. Или может потребоваться создать фильтр действия, который изменяет данные представления, возвращенные действием контроллера. Фильтр действия является атрибутом. Вы можете применить большинство фильтров действий либо к индивидуальному действию контроллера, либо ко всему контроллеру.

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

**View** - файл с расширением cshtml, содержащий HTML, CSS, JavaScript и Razor-конструкции.

**Razor** - View Engine – движок представления, компонент ASP.NET Core MVC-фреймворка, предназначенный для генерации содержимого Response на основе содержимого cshtml-файла. Впервые в ASP.NET MVC 3, 2010.

View: cshtml-файл компилируется в сборку, которая используется для генерации Response-содержимого.

****

Операторы Razor начинаются с символа "@". В этом случае оператор @model объявляет тип объекта модели, который будет передаваться представлению из метода действия. Это позволяет ссылаться на методы, поля и свойства объекта модели представления с помощью @Model

*Обнаружение представления*

Когда действие возвращает представление, происходит процесс, который называется обнаружением представления. Он служит для определения используемого файла представления на основе имени представления.

Метод View (return View();) по умолчанию возвращает представление с тем же именем, что и у метода действия, из которого он был вызван. Сначала среда выполнения ищет представление в папке Views/[имя\_контроллера]. Если подходящее представление в ней не найдено, поиск производится в папке Shared.

Не имеет значения, возвращается ли объект ViewResult неявно с помощью метода return View(); или имя представления явно передается в метод View с помощью return View("<ViewName>");. В обоих случаях обнаружение подходящего файла представления происходит в следующем порядке:

1. Views/ [ ControllerName]/ [ ViewName]. cshtml
2. Views/Shared/[ViewName].cshtml

Вместо имени файла можно предоставить путь к файлу представления. При использовании абсолютного пути, начинающегося с корня приложения (может начинаться с символов "/" или "~/"), необходимо указывать расширение

Обнаружение представлений предусматривает поиск файлов представлений по имени.

Существуют различные способы передачи данных из контроллера в представление:

* ViewData
* ViewBag
* Модель представления

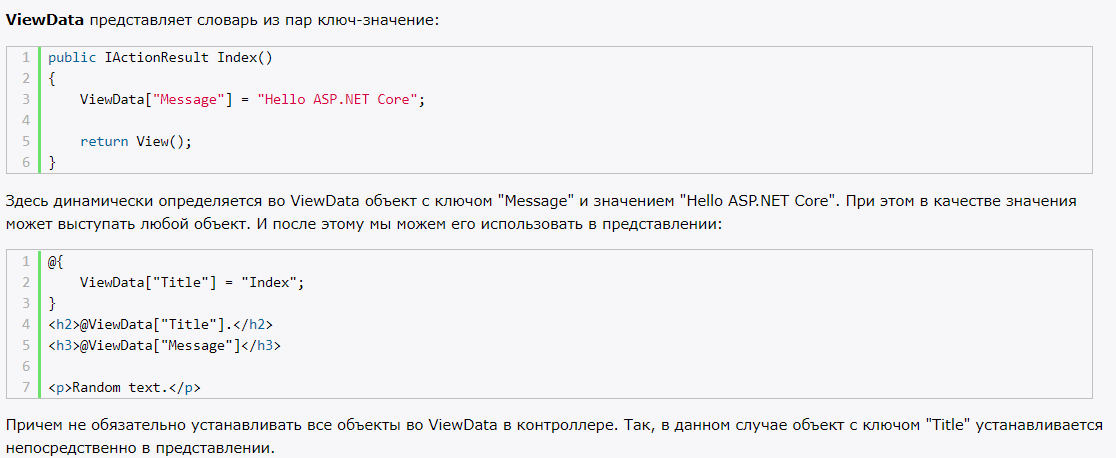
Есть ещё **строготипизированные вьюшки**

На самом деле имеется ввиду, что этим представлениям передаются некоторые параметры (return view(new A(3,5)) до этого просто было вью()

Тут вью принимает объект класса А

И мы получаем этот объект в виде модели [@model](https://vk.com/id220680) и пишем еще тип

(Смелов)



##### 

##### 

##### 

##### 

##### **27. ASP.NET CORE: MVC-представление, директивы @using, @function, @inherits. Пример.**

<https://docs.microsoft.com/ru-ru/aspnet/core/mvc/views/razor?view=aspnetcore-3.1>

*директива @using*

В C# оператор using позволяет обеспечить использование какого-то объекта. В Razor для создания вспомогательных функций HTML, содержащих дополнительное содержимое, используется тот же механизм.

В следующем коде вспомогательные функции HTML используют оператор @using для создания тега <form>:

@using (Html.BeginForm())

{

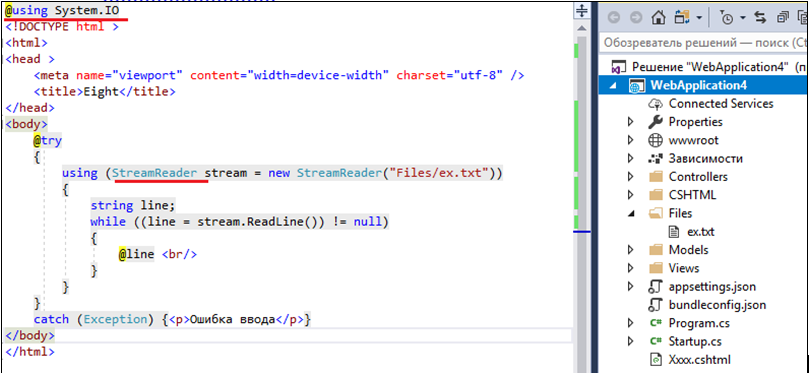
<div>

Email: <input type="email" id="Email" value="">

<button>Register</button>

</div>

}



*директива @functions*

Директива @functions позволяет добавлять элементы C# (поля, свойства и методы) в создаваемый класс:

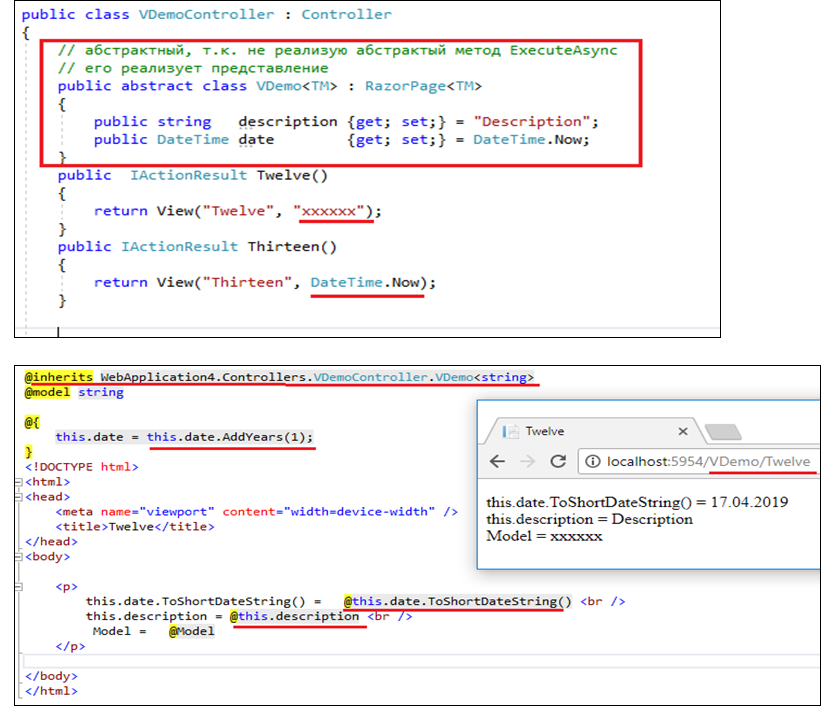


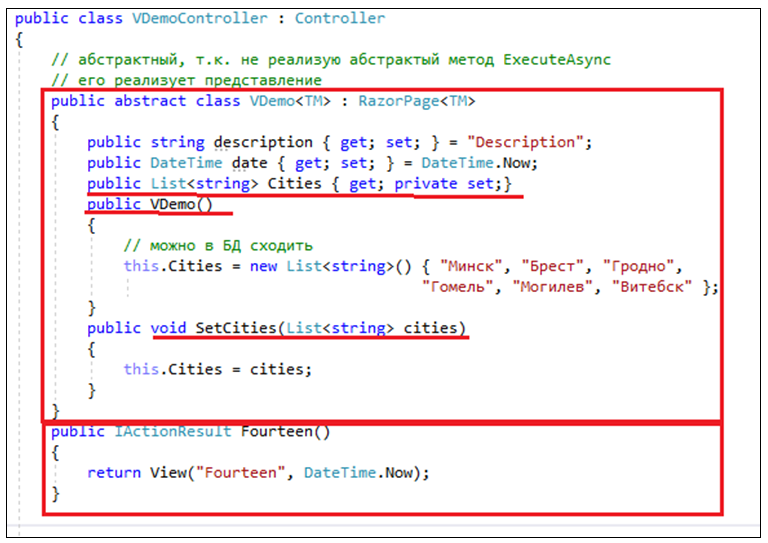
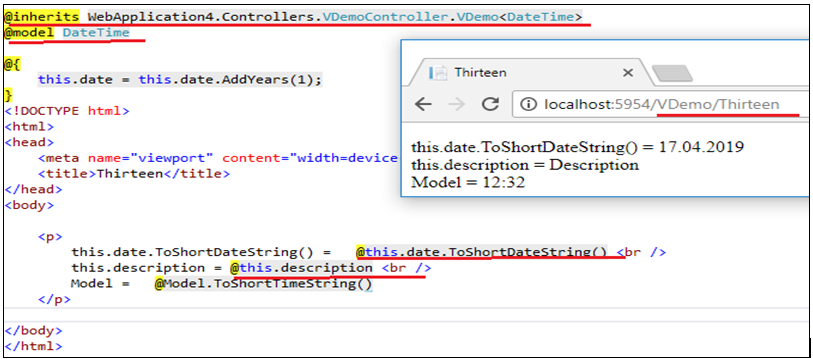
*директива @inherits*

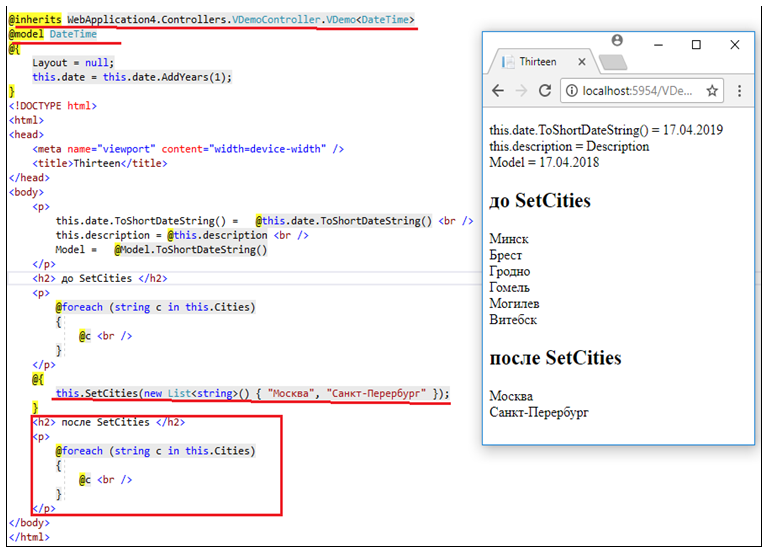
Позволяет создать собственный базовый класс для представления; базовый класс должен быть производным от RazorPage или RazorPage<ModelType>.

Директива @inherits позволяет полностью управлять классом, которому наследует представление:

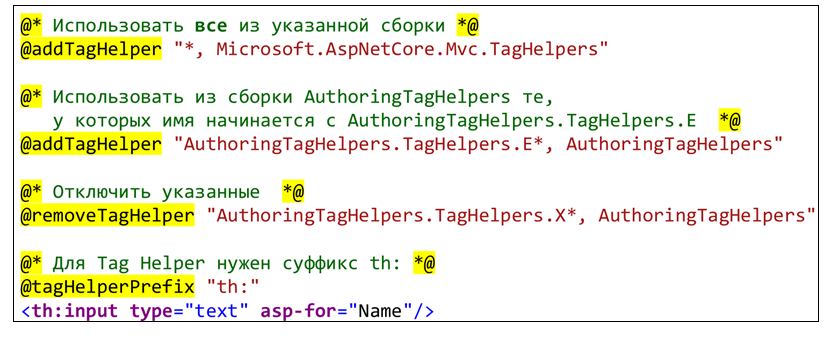
@inherits TypeNameOfClassToInheritFrom







##### **28. 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.

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

**В asp.net есть несколько View Engine: AspX, Razor**

**View** - это объект, который создаётся с помощью View Razor.

**Задача View Engine**: принимает на вход файл с разметкой (Razor Engine принимает файл cshtml), на выходе образуется c# класс, который и формирует респонс. Это происходит один раз за запуск приложения при первом обращении к view оно преобразуется объект.

**Создание/запуск вью инициируется контроллером (его экшном)**

Если вью без параметров, то подразумевается, что мы вызываем cshtml файл с именем, совпадающим с именем экшна и файл по умолчанию должен располагаться в Views/{имя контроллера}/{имя экшна}.cshtml

**Cshtml** файл содержит в себе статический html + razor разметку, которая дает возможность переключать контекст с помощью @{} с режима разметки в режим c#.

**Layout** – шаблон страницы/разметки (компоновщик, мастер страницы), общий шаблон для cshtml файлов (позволяет создать общую часть/структуру).

**@RenderBody()** – поступает разметка, которую мы никак не именуем, вставляет то, что находится вне секций

**@RenderSection(“head”, true)** – вставляет секции по имени (@section head{}), обязательность

**Свойство Layout** унаследовано классом (в который превращается вью) от базового класса, который предоставляется asp.net.

Базовый способ передачи параметров с контроллера на представление является коллекция ViewBag.

**ViewBag** – динамический объект, который позволяет динамически создавать его свойства.

Если во вьюбэг передаем экзмпляр класса, то во вью должны использовать нэймспейс этого класса через @using MVC01.Controllers

Строго-типизированные представления – передаем во вью параметр, представления, в которых используется @model

View(new A(3,5))

Внутри View получаем объект с помощью

@model MVC01.Controllers.VController.A

… @Model.Sum

**Существуют 3 модели:**

- модель данных (создаем при работе с EF)

- модель экшна (образуется тем набором параметров, которые передаются в экшн)

- модель вью (выше в примере, передаем во вью)

В cshtml, чтобы отобразить в разметке символ @ нужно продублировать @@

Мы можем создать вспомогательный метод helper

- внутренний (во вью, можем применять в рамках одного вью)

@helper MethodName(string[] params){}

…@MethodName(Model)

- внешний (статическим классом) можем использовать в нескольких вью

public static MvcHtmlString CityList (this HtmlHelper html, string[] cl)

{ return new MvcHtmlString(tag.ToString()); }

Во вью прописать @using MVC01.Helpers;

…@Html.CityList(Model)

**Стандартные вспомогательные методы**

Html.Action (возвращает string) – имя экшна, контроллер, параметры экшна, предполагается возврат разметки.

Html.RenderAction (пишет в response, @Html.Action = @{ Html.RenderAction})

Html.Partial (возвращает строку) – частичное представление для повторного применения

Html.RenderPartial (возвращает void, выводит (Write) в Response)

Html.ActionLink – генерирует ссылку, имя, экшн, контроллер

Работа с формой – 2 фазы: вывод формы гет запрос, нажатие на кнопку и пост запрос

Html.BeginForm, Html.EndForm

Html.TextBox, Html.RadioButton, Html.CheckBox

Html.Hidden, Html.Password, Html.TextArea

Html.Display, Html.DisplayText

С суффиксом For мы работаем с моделью, ее свойствами

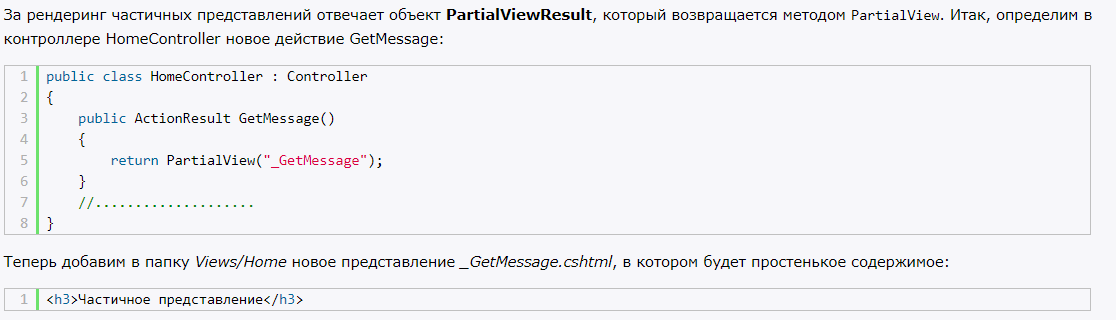
[Compare], [Range], [RegularExpression],[Required], [StringLenght], [Compare], ModelState (IsValid, AddModelError())

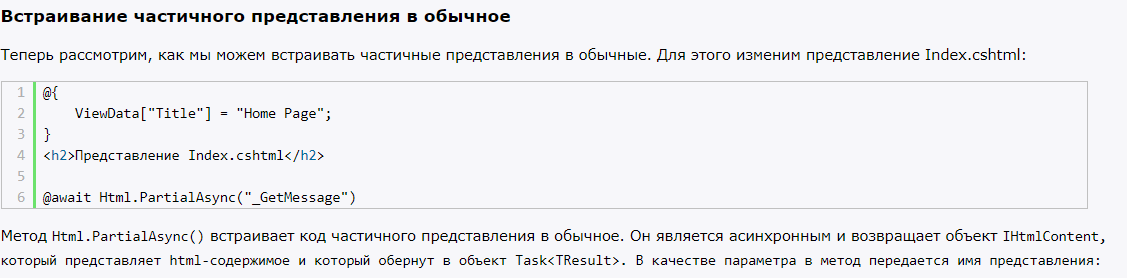
##### 

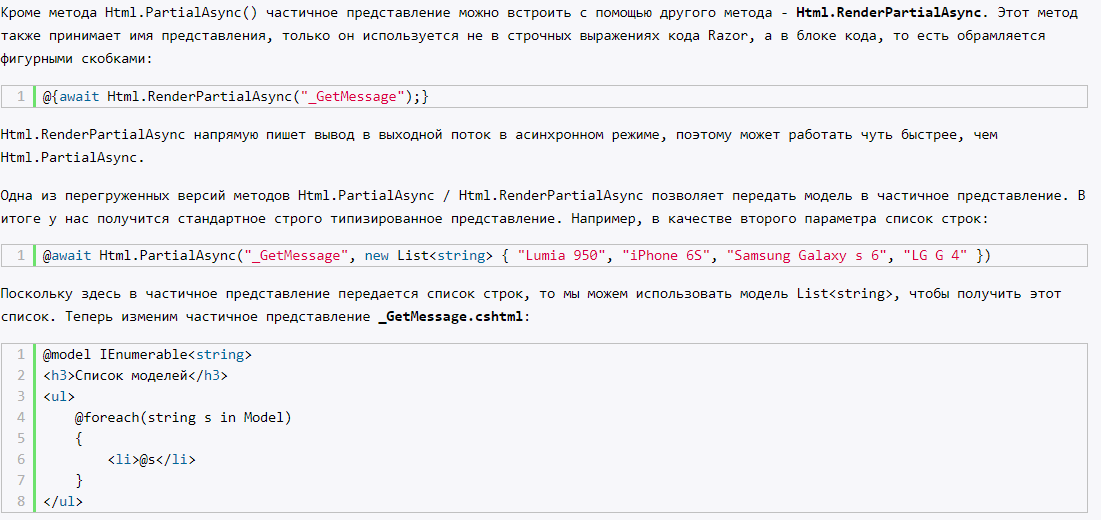
##### **30. ASP.NET CORE: MVC-представление, частичные представления (partial view). Встроенные хелперы. Пример.**

[**Частичные представления**](https://docs.microsoft.com/ru-ru/aspnet/core/mvc/views/partial?view=aspnetcore-3.1) сокращают дублирование кода, обеспечивая управление многократно используемыми частями представлений.

Их отличительной особенностью является то, что их можно встраивать в другие обычные представления. Частичные представления могут использоваться также как и обычные, однако наиболее удобной областью их использования является рендеринг результатов AJAX-запроса.

****

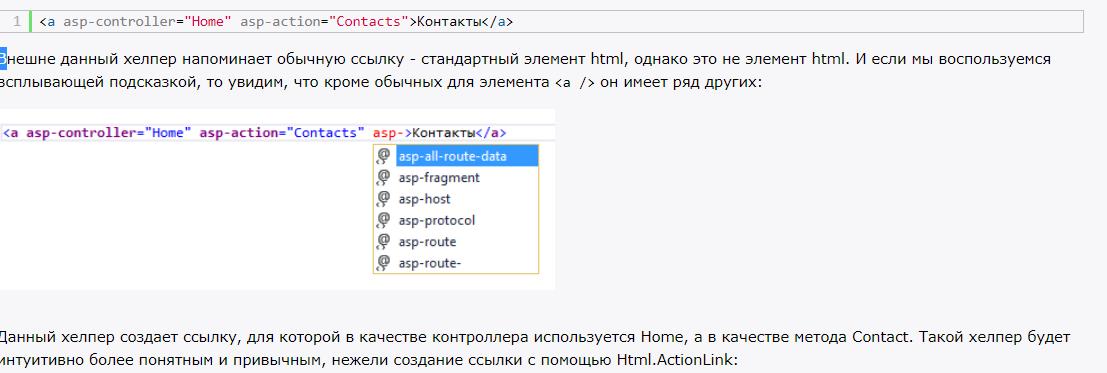
****

****

**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:

****

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.

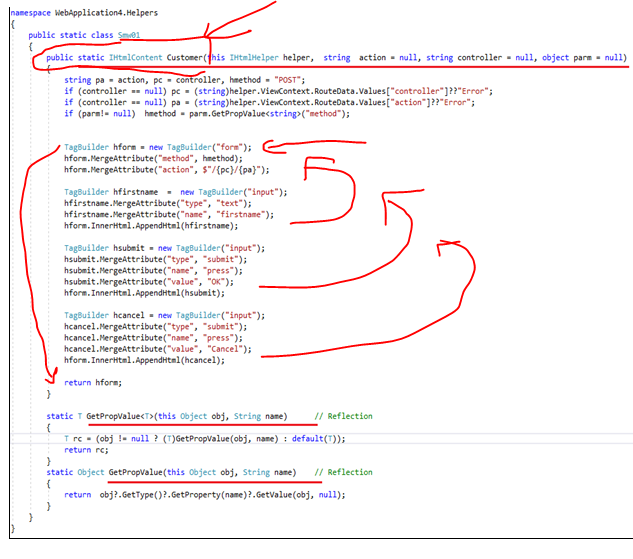
##### 

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

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

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

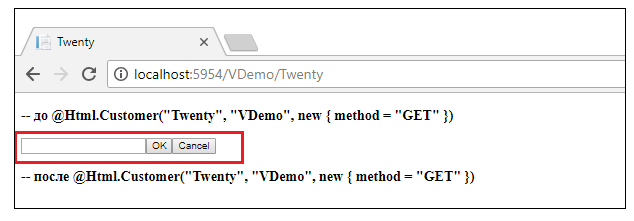
Парсятся приходящие параметры, создается тег **form**; Создаются теги **input**, которые затем встраиваются в тег form. И выводится form.



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):





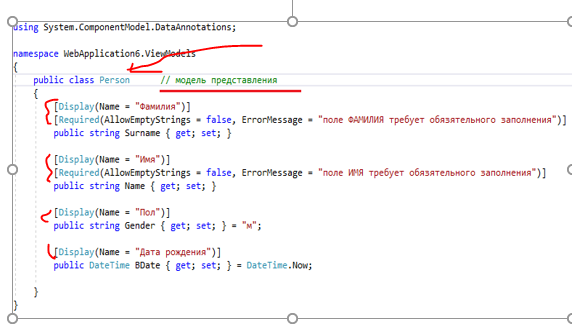
##### 

##### **32. 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.**

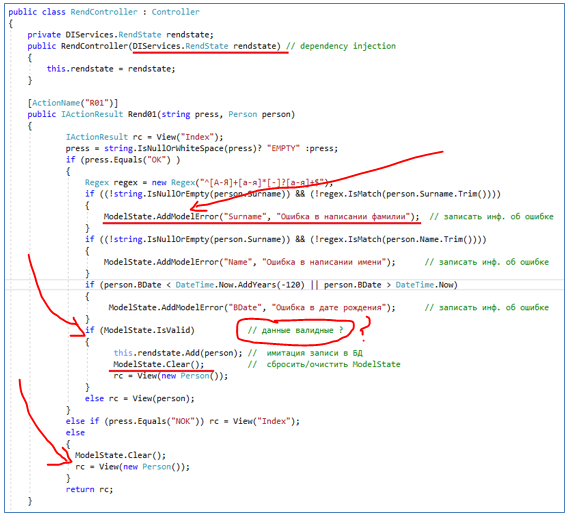
**ModelState:** объект, для хранения состояния представления. Заполнение ModelState при заполнении параметров action.

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

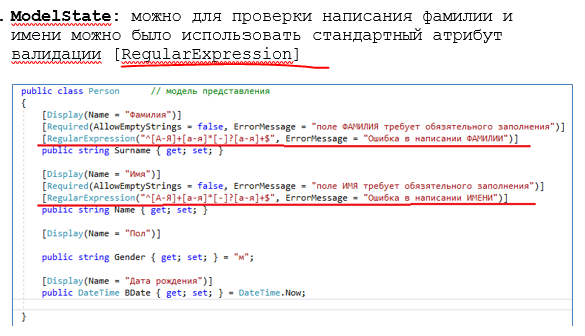
На рисунке снизу показана модель представления с валидацией (стандартные атрибуты c#, но можно и свой сделать)



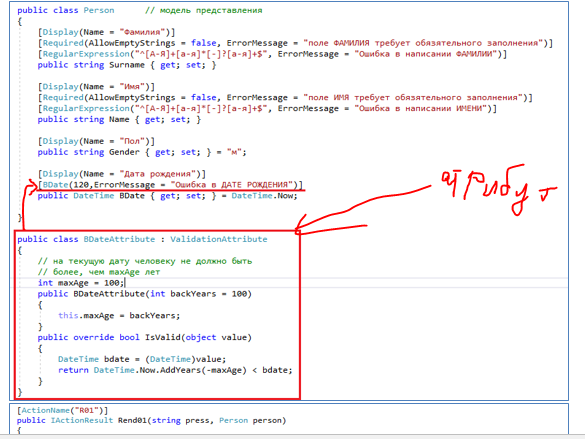
На скрине снизу показан фрагмент записи ошибок валидации в объект ModelState. В случае атрибутов (скрин сверху) они добавляются автоматически



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



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



**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.

**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 таблицы