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

**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:

ü стандартные (прописаны в протоколе HTTP)

ü 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**

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

ü head, post (параметры передаются в теле запроса)

ü 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.

**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)

# **2. HTML. Структура HTML-страницы. CSS. Модель DOM.**

**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.

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">.*

Тег **style** позволяет писать CSS прямо внутри HTML страницы. Данный тег должен размещаться внутри тега **<head>.**

**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.

# **3. JavaScript. Основные стандарты. Типы данных. Программные структуры. Принцип применения. Понятие DHTML.**

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:

ü Динамически изменять содержимое веб-страниц

ü Привязывать к элементам обработчики событий

ü Выполнять код через заданные промежутки времени

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

ü Создавать и считывать cookies

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

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

**Встраивание JS в html-код**

· 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.

Dynamic HTML или **-** — это способ создания интерактивного веб-сайта, использующий сочетание статичного языка разметки HTML, JavaScript, CSS и DOM. Он может быть использован для создания приложения в веб-браузере: например, для более простой навигации или для придания интерактивности форм. DHTML может быть использован для динамического перетаскивания элементов по экрану. Также он может служить как инструмент для создания основанных на браузере видеоигр.

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

**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.

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**

# 

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

**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

# **6. Технология ASP.NET: архитектура приложения, клиент, сервер, aspx-страница, серверные элементы управления, обратная отправка, модель обработки событий.**

## **Технология ASP.NET**

**ASP.NET** – технология создания web-приложений и web-сервисов компании Microsoft. Технология является составной частью Microsoft.NET. Является развитием более старой технологии Microsoft ASP. Концептуально является новой. Счет версий ASP.NET идет параллельно версиям .NET Framework. ASP .NET 1.0 2000г. для платформы .NET Framework 1.0. Последняя версия ASP .NET 4.0.

Шаблоны: WebForms, MVC, Asmx, WCF, WebApi, Handler.

¾ *ASP.NET компилируется также, как и любое приложение .NET.*

¾ *ASP.NET поддерживает три языка C#, VB.NET и MSIL.*

¾ *ASP.NET приложение исполняется CLR.*

¾ *ASP.NET является объектно-ориентированной технологией.*

¾ *ASP.NET поддерживает большинство браузеров.*

¾ *Компоненты ASP.NET: CLR 2.0, Механизм ASP.NET 2.0, FCL 3.5 (ADO.NET, LINQ, ASP.NET AJAX), C# 3.0, VB 9.0., J#.*

¾ *ASP.NET интегрируется с новой технологией Silverlight.*

**3 типа приложений**

**ASP.NET Web Forms** самый простой вид, предназначенный для быстрой разработки серверной (back-end) части приложения.

**ASP.NET MVC** исп. для разработки большихприложений. Правила разработки MVC загоняют вас в определенные рамки. Это более дисциплинирует, и приложение становится масштабируемым.

**ASP.NET Web-services** особый вид приложения, который исп. с SOAP протоколом, который не отменяет HTTP протокол.

|  |
| --- |
| Web Form |
| ASP.NET |
| FCL |
| CLR |
| Windows |

**ASP.NET** – технология, основанная на .NET, для разработки распределённых приложений. **Основа .NET**: Реализация .NET библиотеки IIS api и Набор обработчиков http, предназначенных для обработки запросов, пришедших на IIS. Исполняется приложение в **CLR** (Common Language Runtime) – общеязыковая среда исполнения, виртуальная машина на которой исполняются все приложения, работающие в среде .NET. Реализация CLI VES компанией Microsoft. Компилятор **JIT** (Just in Time). **FCL (Framework Class Library)** – реализация CLI BCL компанией Microsoft. Можно рассматривать, как API CRL.

## **Aрхитектура приложения**

Любой запрос встречает http-обработчик, после этого создаётся объект web-forms, наследник от Page. Есть объект application, он создаётся после инсталляции на IIS и запуска приложения. Общий объект для всех сессий, которые обращаются к приложению.

## **Клиент**

**Клиент** — аппаратный или программный компонент вычислительной системы, посылающий запросы серверу. В качестве клиента чаще всего выступает **браузер**, то есть браузер — это обыкновенный HTTP-клиент, который может слать запросы к серверу, получать ответы от сервера и их интерпретировать.

## **Сервер**

**Сервер** — программное обеспечение, принимающее запросы от клиентов. 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.

## **Aspx-страница**

Файлы разметки **aspx** содержат информацию, необходимую для прорисовки формы (информация об элементах управления). Здесь присутствуют html-теги. Файл может содержать статическую разметку, директиву (первая строка с @).

В директиве есть codeBehind, он указывает на файл cs, связанный с данным aspx-файлом. Другими словами, это связь между разметкой и её серверным представлением. Некоторые элементы разметки имеют атрибут **runat=”server”**, означающий, что этот элемент управления является серверным.

**Серверный элемент управление** – элемент управления, которому на сервере соответствует программный объект. Чтобы посмотреть серверные объекты, надо открыть файл designer.cs, где будут объявления этих серверных объектов. Этот файл лучше не менять вручную, поскольку он может потерять соответствие с файлом разметки, и можно загубить все приложение.

**Класс aspx.cs** является производным от **System.Web.UI.Page**. Также он является частичным, т.е. может быть описан в нескольких файлах. Если пометили, что объект управления является серверным, то VS автоматически объявит его в designer.cs.

Веб-приложение необходимо **опубликовать**. Сервер для публикации называется **IIS**. Он входит в состав Windows и слушает 80-й порт. Он нужен не только для расположения веб-приложений. В составе VS есть свой встроенный IIS, который называется **IIS Express**. В отладочном режиме мы пользуемся им. Но, чтобы приложение было доступно в сети, необходимо его опубликовать.

**Web-приложение** представляет собой страницу **aspx** с разметкой и именем файла, в котором находится **программный код** на c# предназначенный для обработки этой страницы. ASPX страница всегда является формой.

*Есть форма, она генерирует POST-запрос записанный в runat. Все что расположено внутри формы должно превратиться в INPUT теги в виде ПАРАМЕТРОВ (имя+значение).*

**3 режима просмотра ASPX разметки**: source, конструктор, split.

**Чтобы создать форму надо**: разработка формы, написать код, которые будет обрабатывать.

**PageLoad()** – осуществляется при каждой обратной отправке. Страница живет ровно один запрос.

## **Серверные элементы управления**

**Серверный элемент управления** – компонент веб-приложения, который умеет сам себя отображать на веб-странице, а также ему соответствует тег разметки aspx и программный объект на стороне сервера.

Все серверные элементы, которые используют в веб формах можно условно разбить на 2 большие части:

1. **базовый набор элементов** (разработан компанией Miсrosoft);

2. **пользовательский набор** **элементов** (существует специальных механизм, с помощью которого мы можем создавать собственные элементы).

Все элементы управления, которые входят в *базовый набор* мы можем также разбить на 2 части:

· **Общие** (серверные элементы управления HTML, элементы управления Web, полнофункциональные эл-ты управления Web);

· **Специализированные.**

У каждого из элементов есть своя особенность, но все они являются серверными (стоит *runat=”server”*), а это значит, что они умеют сохранять своё состояние во *viewstate*.

*Основные отличия серверных элементов управления в том, что это самые простые элементы управления, каждому их элементу управления соответствует один тег.*

**Полнофункциональные элементы управления Web** – сильно нагружены, им соответствуют много элементов разметки и имеют много свойств, например, календарь, wizard, меню, threeview.

## **Обратная отправка**

**AutoPostBack** – свойство, которое показывает, будет ли происходить отправка данных на сервер при изменении состояния объекта, у которого значение этого свойства установлено в *true*.

## **Модель обработки событий**

На веб-страницах ASP.NET события, связанные с серверными элементами управления, вызываются на клиенте, а обрабатываются на веб-сервере с помощью страницы ASP.NET. Модель событий элемента управления ASP.NET требует захвата сведений о событии, вызванном на клиенте, и передачи сообщения о событии серверу с помощью HTTP. Необходимо, чтобы страница интерпретировала обратный запрос и определила, какое событие произошло, а также вызвала бы в коде на сервере необходимый метод для обработки события. Основные серверные обработчики событий – **onserverclick**, **onserverchange**.

|  |  |
| --- | --- |
| **ServerClick** | HtmlAnchor, HtmlButton, HtmlInputButton,  HtmlInputSubmit, HtmlInputReset, HtmlInputImage |
| **ServerChange** | HtmlInputText, HtmlInputCheckBox, HtmlInputRadioButton, HtmlInputHidden, HtmlInputSelect, HtmlInputTextArea |

# **7. ASP.NET: web-форма, структура и жизненный цикл web-формы, основные события web-формы, состояние (viewstate) web-формы, автоматическая обратная отправка данных. (2 лаба)**

**ASP.NET WebForms** – это приложение, которое работает на Windows и использует .NET. Приложение WebForms состоит из 3 основных частей:

1. **файл aspx**, содержащий aspx-разметку;

2. связанные с этим файлом **файлы cs** (файлы c#);

3. файл конфигурации (**web.config**).

Файлы разметки в себе содержат информацию, необходимую для прорисовки формы (информация об элементах управления). Здесь присутствуют html-теги. Файл может содержать статическую разметку, директиву (первая строка с @).

В директиве есть codeBehind, он указывает на файл cs, связанный с данным aspx-файлом. Другими словами, это связь между разметкой и её серверным представлением. Некоторые элементы разметки имеют атрибут **runat=”server”**, означающий, что этот элемент управления является серверным.

**Серверный элемент управление** – элемент управления, которому на сервере соответствует программный объект. Чтобы посмотреть серверные объекты, надо открыть файл designer.cs, где будут объявления этих серверных объектов. Этот файл лучше не менять вручную, поскольку он может потерять соответствие с файлом разметки, и можно загубить все приложение.

**Класс aspx.cs** является производным от **System.Web.UI.Page**. Также он является частичным, т.е. может быть описан в нескольких файлах. Если пометили, что объект управления является серверным, то VS автоматически объявит его в designer.cs.

Веб-приложение необходимо **опубликовать**. Сервер для публикации называется **IIS**. Он входит в состав Windows и слушает 80-й порт. Он нужен не только для расположения веб-приложений. В составе VS есть свой встроенный IIS, который называется **IIS Express**. В отладочном режиме мы пользуемся им. Но, чтобы приложение было доступно в сети, необходимо его опубликовать.

*У нас всегда есть форма. Пока мы не нажмем submit, сервер не узнает, что мы сделали в форме. Сервер получает запрос только после submit. Можно написать собственный JS-код, который будет имитировать нажатие кнопки submit.*

**Жизненный цикл формы**

Любой запрос приводит к началу жизненного цикла формы. Весь жизненный цикл формы проходит за один запрос. Задача приложения – сформировать ответ. Форма в процессе формирования ответа проходит несколько этапов:

1. **Init**. Срабатывает только при первичной отправке. При повторном запросе он не выполняется.

2. **PageLoad**. Выполняется каждый раз при каждом запросе. Серверные объекты уже есть и можно с ними работать, их можно изменять.

3. **PagePrerender**. Это последнее место, где можно повлиять на разметку, которая уйдет на сторону браузера.

4. **PageUnload**. Серверные объекты ещё живы, но разметку изменить уже нельзя.

5. **PageDisposed**. Вызывается сборщиком мусора.

Каждый раз наше приложение создается заново и по очереди вызывает методы жизненного цикла. Объект не хранится, под каждый новый запрос создается новый ответ.

**Серверный элемент сохраняет свое состояние**. Если посмотреть разметку, которая приходит на браузер, то среди тегов можно увидеть тег *<input type=”hidden” name=”\_viewstate”>*, в value которого хранится изменение объекта от первоначального состояния. Именно он предназначен для сохранения данных серверных элементов.

**Viewatate** - объект, который хранит состояние элементов формы, которые отсылаются на сервер. У нас у каждого элемента есть какое-то начальное состояние: не нажатый баттн, пустой текстбокс. Когда мы что-то меняем на форме, то все это должно фиксироваться в viewstate и отпарвляться на сервер. У самой формы есть спрятанный объект viewstate, он hidden по умолчанию и нам этого не видно. Если открыть Ф12, то можно посмотреть его. Когда мы сделаем изменения на странице - все изменения будут зафиксированы в viewstate и отправятся на сервер. В сервере уже создан объект нашей формы и все объекты, которые у нас есть ан странице (для баттона и т.д.), у которых прописан рунат=сервер. Сервер читает форму и смотрит поле viewstate и если видет, что изменения произошли, то он накладывает на этот viewstate новое значение.

Если у нас произошло слишком много изменений, то значения будут накладываться и накладываться на вьюстэйт. Тогда будет слишком большое значние, которое надо будет делить на блоки. 1 блок = 64 бит данных. Будет много блоков. Это плохо, злоумышленники могут влезть и изменить что-то. Вьюстэйт не безопасно! В одном viewstate хранятся изменения каждого элемента.

**В какой момент уходят данные на сервер?**

По событию submit (нажатие на кнопку).

**Autopostback** – этой свойство, которое указывает, что данные формы должны передаваться на сервер автоматически. Например, если отметил чекбокс, то сразу идет запрос на сервер (автоматически генерируется js, в котором стоит onclick-обработчик, который заканчивается submit-ом).

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

**Публикация приложения** – разворачивание /\*перенос\*/ на iis.

**IIS** (**Internet Information Services -** информационные службы Интернета**)** – набор узлов. Для каждого узла указывается протокол, который используется для доступа к этому узлу, и номер порта, который обслуживает этот узел.

**Установка IIS**

Компонент IIS включен как часть установки Windows (как для сервера, так и для рабочих станций) и требует активизации и конфигурирования.

Каждая версия операционной системы Windows предлагает свою версию IIS - IIS 8 (в Windows 8), IIS 7.5 (в Windows 7) или IIS 7 (в Windows Vista). Во всех этих версиях Windows, IIS включен, но изначально не установлен. Чтобы установить его, необходимо выполнить следующие действия:

Откройте панель управления.

Выберите "Программы".

Нажмите кнопку "Включение или отключение компонентов Windows". Теперь вам нужно подождать, пока Windows исследует вашу систему.

Найдите элемент Internet Information Services (Службы IIS) в верхней части списка и нажмите на галочку чтобы включить его:

Windows позволяет включить множество компонентов IIS: поддержка FTP-сервера, дополнительные инструменты управления, службы обратной совместимости с IIS 6.

Убедитесь, что вы выбрали поддержку ASP.NET. Для этого раскройте узел Службы Интернета --> Компоненты разработки приложений --> ASP.NET (Internet Information Services --> World Wide Web Services --> Application Development Features --> ASP.NET):

Если вы хотите использовать поддержку IIS в Visual Studio, которая позволяет вам создавать виртуальные каталоги IIS непосредственно в диалоговом окне New Web Site, вам нужно выбрать пункт «Совместимость управления IIS 6» в разделе «Средства управления веб-сайтом» (Web Management Tools --> IIS 6 Management Compatibility).

Как только вы выбрали нужные параметры IIS, нажмите кнопку OK для завершения установки.

**Управление IIS**

При установке IIS, он автоматически создает каталог с именем **C:\inetpub\wwwroot**, который представляет ваш веб-сайт. Все файлы в этом каталоге будет отображаться, как будто они находятся в корневом каталоге вашего веб-сервера.

Чтобы добавить дополнительные страницы на ваш веб-сервер, можно скопировать файлы HTML, ASP или ASP.NET напрямую в каталог C:\Inetpub\wwwroot. Например, если добавить файл TestFile.html в этот каталог, вы можете запросить его в браузере через URL-адрес http://localhost/TestFile.html. Вы даже можете создавать вложенные папки для группирования связанных ресурсов. Например, вы можете получить доступ к C:\inetpub\wwwroot\MySite\MyFile.html через браузер, используя URL-адрес http://localhost/MySite/MyFile.html.

Каталог **wwwroot** удобен для запуска простых примеров и статичных страниц. Для правильного использования ASP.NET вы должны сделать свой собственный виртуальный каталог для каждого веб-приложения, которое вы создаете. Например, вы можете создать папку с любым именем на любом диске вашего компьютера и поместить ее в виртуальный каталог IIS как будто она расположена в каталоге C:\inetpub\wwwroot.

Прежде чем начать работу, вам нужно запустить диспетчер служб IIS. Его можно найти в меню Start (Пуск). Конкретное расположение может зависеть от используемой версии Windows (IIS --> Диспетчер служб IIS). Ярлык программы будет располагаться в разделе Programs (Программы) или Administrative Tools (Администрирование).

Если развернуть элемент сервера в древовидном представлении в левой части экрана, отобразится элемент Sites (Сайты), содержащий единственную запись Default Web Site (Веб-сайт по умолчанию). Сайт - это коллекция файлов и каталогов, образующих веб-сайт. На одном сервере IIS может поддерживать несколько сайтов, как правило, на различных портах TCP/IP (по умолчанию используется порт 80). Сочетание имени сервера и порта сайта образует первую часть URL-адреса. Например, при использовании сервера mywebserver с сайтом, подключенным к порту 80, URL-адрес выглядит следующим образом:

http://mywebserver:80

Каждый сайт может содержать множество файлов и каталогов. Каждый из них образует часть URL-адреса. Так, URL-адрес статической страницы mypage.html, расположенной в каталоге myfiles, будет следующим:

[http://mywebserver:80/myfiles/mypage.html](http://mywebserver/myfiles/mypage.html)

**Развертывание веб-сайта с помощью IIS**

После установки IIS мы можем создать собственный узел. В обозревателе серверов кликаем правой копкой мыши по пункту «Узлы» и в контекстном меню выбираем пункт «добавить веб-узел». В появившемся окне задаем:

1. имя узла

2. физический путь (место, в котором будут храниться каталоги и файлы разворачиваемого приложения)

3. тип (http или https)

4. IP-адрес для веб-узла. Если выбрать все (без значения), веб-узел доступен для всех интерфейсов и всех настроенных IP-адресов.

5. порт

6. поставить галочку «запустить узел немедленно»

**Копирование приложения**

После в физическом каталоге (который указали в пункте «физический путь») создаем папку с нашим приложением (можно и не создавать, а копировать файлы сразу в узел). В эту папку копируем необходимые файлы нашего приложения. IIS понадобится также указать, что развернутый сайт является приложением. Это не обязательно, но при развертывании приложений ASP.NET почти всегда будет желательным - активизируется состояние сеанса и другие функциональные средства ASP.NET. Щелкните ПКМ на папке FileCopy в области Connections (Подключения) и в контекстном меню выберите пункт Convert to Application (Преобразовать в приложение).

Откроется диалоговое окно Add Application (Добавление приложения). Используемый пул приложений можно изменить, щелкнув на кнопке Select (Выбрать). Настроить учетную запись пользователя, которую IIS будет применять для доступа к содержимому сайта, можно с помощью кнопок Connect as... (Подкл. как...) и Test Settings... (Тест настроек...). В текст-бокс Alias вводим значение, которое будет являться URL приложения (используется для доступа к приложению по URL). Также можно выбрать пул приложения.

**Пул приложений IIS** обслуживает сайты и веб-приложения, размещенные на вашем сервере. Выделенный пул приложений IIS обеспечивает вашим клиентам определенный уровень изоляции между сайтами. Так как каждый выделенный пул приложений работает независимо, ошибки, возникающие в пуле одного пользователя, не повлияют на приложения, запущенные в пулах других пользователей.

Пулы приложений позволяют для упрощения конфигурирования и управления группировать вместе аналогичные или связанные приложения. При этом приложения, которые включены в пулы приложений, изолируются, в результате чего проблемы, возникающие в одном пуле, не оказывают влияния на приложения из других пулов.

Не существует никаких жестких и кратких правил назначения приложений в пулы. Приложения можно группировать по принципу схожести профилей рабочих характеристик, принадлежности одному подразделению или по любому другому принципу, оправданному в конкретной среде. Одним из наиболее полезных средств является возможность наличия различных пулов приложений, которые используют различные версии .NET Framework.

IIS 8 автоматически создает набор пулов приложений, в том числе пул, используемый по умолчанию при создании нового приложения.

После задания всех параметров жмем «ок». Значок каталога нашего приложения должен измениться. Если этого не произошло, кликаем правой кнопкой мыши по нашему узлу, выбираем «управление веб-узлом» -> «перезапустить».

Копирование файлов приложения также может быть осуществлено с помощью VS.

Вам просто нужно выбрать в меню WebSite --> Copy Web Site. Откроется новое диалоговое окно Visual Studio.

Это окно состоит из двух списков файлов. Слева находятся файлы в текущем проекте (на локальном жестком диске). Справа находятся файлы на целевом месте (удаленный веб-сервер). При первом открытии этого окна вы ничего не увидите справа, потому что вы не указали цель. Вы должны нажать кнопку Connect вверху, чтобы предоставить эту информацию. При этом Visual Studio откроет новое диалоговое окно. В этом окне вы можете выбрать одно из мест, куда нужно копировать веб-сайт:

**File System**. Это самый простой выбор - вы копируете файлы сайта внутри текущей файловой системы.

**Local IIS**. Позволяет выбирать виртуальные каталоги локального сервера IIS. Это удобно если сайт нужно будет разместить в другом каталоге.

**FTP Site**. Обеспечивает доступ к удаленному сайту через FTP. Многие хостинг-провайдеры предлагают доступ к сайтам через FTP, так что благодаря этой настройке вы можете сразу разворачивать свой сайт на удаленном веб-сервере, не прибегая к услугам различных FTP-программ наподобие TotalCommander.

**Remote Web Server**. Эта опция предлагает подключение к сайту по URL-адресу с использованием расширения FrontPage Extensions.

Как только вы выберите соответствующий пункт назначения, нажмите кнопку Open. Visual Studio попытается подключиться к удаленному сайт и получить список файлов.

**Создание и настройка виртуальных каталогов**

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**

Чтобы создать виртуальный каталог с помощью диспетчера IIS:

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:

Псевдоним. Введите имя виртуального каталога. Выберите короткое имя, которое легко вводить, поскольку это имя вводится пользователями для доступа к веб-узлу.

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.

**Настройка безопасности виртуального каталога для уже существующей учетной записи**

В IIS Manager выберите настраиваемый виртуальный каталог, а затем в области **Действия** выберите пункт **Изменение разрешений**. Появится диалоговое окно **Свойства** виртуального каталога. Перейдите на вкладку **Безопасность**. Нажмите кнопку **Изменить**. Появится диалоговое окно **Разрешения**.

Выберите из списка **Имена пользователей или групп** существующую учетную запись. Установите флажки для необходимых разрешений в списке **Разрешения** для выбранной группы или пользователя в соответствии с требованиями безопасности виртуального каталога. Нажмите кнопку **Применить**, чтобы применить изменения.

Нажмите кнопку **ОК**, чтобы закрыть диалоговое окно **Разрешения**. В диалоговом окне **Свойства** нажмите кнопку **Применить**, чтобы применить изменения.

Нажмите кнопку **ОК**. Настройка безопасности виртуального каталога для новой учетной записи

В IIS Manager выберите настраиваемый виртуальный каталог, а затем в области **Действия** выберите пункт **Изменение разрешений**.

Появится диалоговое окно **Свойства** виртуального каталога.

Перейдите на вкладку **Безопасность**.

Нажмите кнопку **Изменить**.

Появится диалоговое окно **Разрешения**.

Нажмите кнопку **Добавить**, чтобы добавить учетную запись.

Появится диалоговое окно **Выбор: пользователи, компьютеры или группы**.

Выберите пользователя, компьютер или группу в соответствии с требованиями безопасности виртуального каталога.

Нажмите кнопку **ОК**, чтобы закрыть диалоговое окно **Выбор: пользователи, компьютеры или группы**.

Установите флажки для необходимых разрешений в списке **Разрешения** для нового пользователя, компьютера или группы в соответствии с требованиями безопасности виртуального каталога.

Нажмите кнопку **Применить**, чтобы применить изменения.

Нажмите кнопку **ОК**, чтобы закрыть диалоговое окно **Разрешения**.

В диалоговом окне **Свойства** нажмите кнопку **Применить**, чтобы применить изменения.

Нажмите кнопку **ОК**.

Настройка проверки подлинности для виртуального каталога

В IIS Manager выберите настраиваемый виртуальный каталог.

В меню **Просмотр возможностей** дважды щелкните элемент **Проверка подлинности**.

Выберите настраиваемый тип проверки подлинности. В области **Действия** убедитесь, что выбран параметр **Включить**, чтобы включить выбранный тип проверки подлинности.

Если выбранный тип проверки подлинности уже включен, в области **Действия** можно отключить его или изменить настройки проверки подлинности.

В области **Действия** выберите пункт **Изменить** или **Дополнительные параметры**, чтобы настроить дополнительные свойства для выбранного типа проверки подлинности.

Набор доступных действий в области **Действия** может отличаться в зависимости от выбранного типа проверки подлинности. Например, если установлен и включен модуль **Anonymous Authentication**, можно нажать кнопку **Изменить**, чтобы изменить дополнительные свойства анонимной проверки подлинности. Если установлен и включен модуль **Window Authentication** можно щелкнуть **Дополнительные параметры**, чтобы изменить свойства проверки подлинности Windows.

# **9. ASP.NET: Серверные элементы управления, html-элементы управления. (5 лаба)**

**Серверный элемент управления** – компонент веб-приложения, который умеет сам себя отображать на веб-странице, а также ему соответствует тег разметки **aspx** и программный объект на стороне сервера.

Все серверные элементы можно разбить на 2 части:

1. Базовый набор элементов

***a.*** ***Специализированные***

ü элементы управления проверки достоверности (проверка введённых данных в поле – textbox)

ü элементы управления данными (с БД, с xml)

ü элементы управления навигацией (переход между страницами)

ü элементы управления входом в систему

ü элементы управления web parts (используются для больших порталов, также связаны с безопасностью, дают пользователю работать с формой)

ü элементы управления asp.net ajax (с помощью них отправляется ajax-запрос)

ü мобильные элементы управления asp.net (для мобильных устройств, они позволяют делать специализированные сайты для просмотра их с мобильных телефонов)

***b.*** ***Общие***

ü серверные управления html (textbox, button, label…) отличие в том, что это самые простые элементы управления, каждому элементу соответствует один тег разметки html

ü элементы управления web

ü полнофункциональные элементы управления web (сильно нагружены, им соответствуют много элементов разметки и имеют много свойств , например, календарь, меню и т.д.)

2. Пользовательские элементы

· пользовательские элементы управления

· специализированные серверные пользовательские элементы управления

У каждого из элементов есть своя особенность, но все они являются серверными *(стоит runat=”server”),* а это значит, что они умеют сохранять своё состояние во viewstate. Серверные элементы управления используют ViewState для хранения большинства, если не всех, своих свойств. **ViewState** — контейнер, в котором содержится информация об измененных свойствах элементов управления на форме. **ViewState** служит для сохранения изменений, которые были произведены на форме после **postback** (запрос на сервер на изменение данных).

Viewstate делает следующее:

ü Сохраняет данные элементов управления по ключу, как хэш-таблица

ü Отслеживает изменения состояния ViewState'а

ü Сериализирует и десериализирует сохраненные данные в скрытое поле на клиенте

ü Автоматически восстанавливает данные на postback'ах

*Основные отличия серверных html-элементов управления в том, что это самые простые элементы управления, каждому их элементу управления соответствует один тег.*

**Серверные элементы управления HTML** – это те элементы управления ASP.NET, которые обладают 3 свойствами:

1. Он сохраняет своё состояние *(runat=”server”)*

2. Каждому этому элементу соответствует только 1 тег в HTML

3. Он обрабатывает одно из трёх событий *(1 – нет события, 2 – OnServerClick, 3 – OnServerChange)*

Данные элементы пишутся без префиксов и у них есть *runat=”server”*, присутствуют атрибуты **onserverchange** либо **onserverclick**. Но также присутствуют элементы управления, которые не генерируют никаких событий, это такие элементы как **InputReset** и **InputFile**.

**Обработка серверных событий элементов управления HTML**

|  |  |
| --- | --- |
| ServerClick | HtmlAnchor, HtmlButton, HtmlInputButton,  HtmlInputSubmit, HtmlInputReset, HtmlInputImage |
| ServerChange | HtmlInputText, HtmlInputCheckBox, HtmlInputRadioButton, HtmlInputHidden, HtmlInputSelect, HtmlInputTextArea |

# **10.ASP.NET: Серверные элементы управления, полнофункциональные элементы управления.**

**Серверный элемент** – компонент веб-приложения, который умеет сам себя отображать на веб-странице, ему соответствует тег разметки aspx и программный объект на стороне сервера. 3 свойства:

· сохраняет свое состояние

· каждому элементу соответствует только один тег разметки

· обрабатывает одно из трех событий (нет события, click, change)

Не обрабатывают никаких событий: InputReset, InputFile.

Серверные элементы управления – элементы HTML, которые включают атрибут runat=server. Серверные элементы имеют тот же вывод HTML и те же свойства, что и соответствующие HTML-теги. Кроме того, серверные элементы управления обеспечивают автоматические управление состояниями и событиями на сервере.

**Преимущества**:

- Отображаются один к одному с их соответствующими html-тегами

- При компиляции приложения asp.net серверные элементы управления с атрибутом runat=server компилируются в сборку

- Большинство элементов управления включают в себя обработчик события

- Html-теги, не реализованные в качестве отдельный серверных элементов управления, по-прежнему могут использовать на сервере, однако они добавляются в сборку как HtmlGenericControl

- При повторном размещении страницы asp.net серверные элементы управления сохраняют свои значения

Серверные элементы производные от класса HtmlControl

HtmlAnchor <a>, HtmlButton, HtmlForm, HtmlImage, HtmlInputButton и т.д.

Все серверные элементы можно разбить на 2 части:

Базовый набор элементов

***a.*** ***Специализированные***

ü элементы управления проверки достоверности (проверка введённых данных в поле – textbox)

ü элементы управления данными (с БД, с xml)

ü элементы управления навигацией (переход между страницами)

ü элементы управления входом в систему

ü элементы управления web parts

ü элементы управления asp.net ajax (отправляется ajax-запрос)

ü мобильные элементы управления asp.net

***b.*** ***Общие***

ü серверные управления html (textbox, button, label) отличие в том, что это самые простые элементы управления (1 элемент = 1 тег)

ü элементы управления web

ü полнофункциональные элементы управления web (календарь, меню)

Пользовательские элементы

· пользовательские элементы управления

· специализированные серверные пользовательские элементы управления

Некоторые элементы имеют префикс asp, а некоторые нет. С asp элементы, которые отличаются от стандартных. Кнопка есть и в html, и в web.

**Полнофункциональные элементы управления**

Полнофункциональные элементы управления – элементы управления Web, моделирующие сложные структуры пользовательского интерфейса

Table – элемент строится с помощью tr/td

Calendar – полнофукциональный

**Календарь**, **AdRotator**. Баннерная реклама, отображающая изображение из набора на основе предварительно заданного графика, сохраненного в XML-файле. Можно формировать группы баннеров, а также включать/выключать их. **Wizard** является более эффективной версией MultiView. Он поддерживает возможность разработки одновременно нескольких представлений в рамках одного элемента, имеет встроенные кнопки навигации и меню. **MultiView** задает группу представлений (объектов View), из которых может быть выбрано активным и показано клиенту в конкретный момент только одно. Каждый объект View имеет собственное "ID". Для показа конкретного элемента View достаточно установить свойство "ActiveViewIndex" элемента MultiView в значение, равное значению "ID" элемента View, который требуется показать. **TreeView** предназначен для показа иерархии страниц сайта, определенной либо в элементе управления, либо в XML-файле. **Menu** – элемент управления, который может быть вертикальным, горизонтальным, можно делать подсказки.

# **11. ASP.NET: Серверные элементы управления, элементы управления проверкой достоверности. (4 лаба)**

**Серверный элемент** – компонент веб-приложения, который умеет сам себя отображать на веб-странице, ему соответствует тег разметки aspx и программный объект на стороне сервера.

Все серверные элементы можно разбить на 2 части:

3. Базовый набор элементов

***a.*** ***Специализированные***

ü элементы управления проверки достоверности (проверка введённых данных в поле – textbox)

ü элементы управления данными (с БД, с xml)

ü элементы управления навигацией (переход между страницами)

ü элементы управления входом в систему

ü элементы управления web parts

ü элементы управления asp.net ajax (с помощью них отправляется ajax-запрос)

ü мобильные элементы управления asp.net

***b.*** ***Общие***

ü серверные управления html (textbox, button, label…) отличие в том, что это самые простые элементы управления, каждому элементу соответствует один тег разметки html

ü элементы управления web

ü полнофункциональные элементы управления web (календарь, меню и т.д.)

4. Пользовательские элементы

· пользовательские элементы управления

· специализированные серверные пользовательские элементы управления

Некоторые элементы имеют префикс asp, а некоторые нет. С asp элементы, которые отличаются от стандартных. Кнопка есть и в html, и в web.

**Серверный html элемент управления** – серверный элемент управления asp.net, который обладает 3 свойствами:

· сохраняет свое состояние

· каждому элементу соответствует только один тег разметки

· обрабатывает одно из трех событий (нет события, click, change)

Не обрабатывают никаких событий:

- InputReset

- InputFile

**Элементы управления проверкой достоверности**

**Элементы управления проверки достоверности** – это специальные элементы, с помощью которых мы можем проверить другие элементы управления, введённые в них значения. В основном проверяются textbox.

В результате генерируется 2 проверки: одна на стороне клиента – js код, другая – на стороне сервера. В общем случае проверка будет на клиенте и на сервере. Если на клиенте прошла проверка, то такая же будет проверка на сервере.

**RequiredFieldvalidator** проверяет, заполнено ли поле.

**RangeValidator** проверяет на диапазон значений. Диапазон может быть, число, дата или строка

**CompareValidator** сравнивает для введенных значения.

**RegularExpressValidator** основан на регулярном выражении. Задается регулярное выражение и проверяется значение введенного текста на соответствие этому регулярному выражению.

**CustomValidator** позволяет определить любую клиентскую или серверную операцию проверки.

Каждый валидатор указывает на те элементы, Которые он будет проверять. Валидирование осуществляется после нажатия submit. Можно поместить валидаторы и кнопку в одну ValidationGroup. Если нажимаем кнопку в пределах группы, срабатывают только те валидаторы, у которых точно такая же ValidationGroup.

**ValidstionSummary** – элемент, позволяющий собрать отчет по валидации в одном месте. Если его не использовать, сообщения об ошибках отображались бы после каждого элемента управления проверкой достоверности.

# **12. ASP.NET: Серверные элементы управления, пользовательские элементы управления.**

**Серверный элемент** – компонент веб-приложения, который умеет сам себя отображать на веб-странице, ему соответствует тег разметки aspx и программный объект на стороне сервера.

Все серверные элементы можно разбить на 2 части:

Базовый набор элементов

***a.*** ***Специализированные***

ü элементы управления проверки достоверности (проверка введённых данных в поле – textbox)

ü элементы управления данными (с БД, с xml)

ü элементы управления навигацией (переход между страницами)

ü элементы управления входом в систему

ü элементы управления web parts

ü элементы управления asp.net ajax (с помощью них отправляется ajax-запрос)

ü мобильные элементы управления asp.net

***b.*** ***Общие***

ü серверные управления html (textbox, button, label…) отличие в том, что это самые простые элементы управления, каждому элементу соответствует один тег разметки html

ü элементы управления web

ü полнофункциональные элементы управления web (календарь, меню и т.д.)

Пользовательские элементы

· пользовательские элементы управления

· специализированные серверные пользовательские элементы управления

**Серверный html элемент управления** – серверный элемент управления asp.net, который обладает 3 свойствами:

· сохраняет свое состояние

· каждому элементу соответствует только один тег разметки

· обрабатывает одно из трех событий (нет события, click, change)

Не обрабатывают никаких событий: InputReset, InputFile.

**Пользовательский элемент управления** – небольшая часть страницы, которая может содержать статический HTML-код и элементы управления веб-сервера.

*1.* *Пользовательские элементы управления*

*2.* *Пользовательские веб-элементы управления*

Используются, чтобы придать единообразие для форм, страниц и элементов.

Можно разработать свой кусочек разметки, а после его помещать на каждую страницу. Можно разработать свою мини-форму, которую вставлять в другие формы.

Создать новый пользовательский элемент управления значит:

1. Ему должен соответствовать какой-то тег

2. Должно быть имя

3. Могут быть обработчики событий

4. Должны быть свойства

5. Механизм перемещения из одного места в другое

Нужно создать файл с расширением **ascx**. Для такого расширения нет никаких обработчиков, поэтому мы его не можем запросить в URI когда делаем запрос к серверу, такой файл может быть только внедрен в другие страницы.

**Преимущество** заключается в том, что ПЭУ можно использовать на нескольких формах одного проекта или веб-приложения.

Пользовательские элементы управления начинаются с директивы **Control**, а не Page.

Пользовательские элементы управления используют расширение **.ascx**, а не .aspx, а их файлы отделенного кода наследуются от класса **System.Web.UI.UserControl**.

Фактически классы **UserControl** и Page унаследованы от одного и того же класса **TemplateControl**, что обусловливает общность многих их методов и свойств.

ПЭУ не могут запускаться самостоятельно, их нужно подключать обязательно в форме aspx.

**Свойства в ascx.cs** мы задаем для того, чтобы при подключении ascx формы можно было указывать значения по умолчанию.

Теперь, чтобы протестировать элемент управления, его нужно поместить в веб-форму. Вначале с помощью директивы **Register**, которую можно поместить непосредственно за директивой Page, странице ASP.NET необходимо указать, что мы планируем использовать этот пользовательский элемент управления.

Мы перетягиваем **MyControl.ascx** в нашу форму.

**CodeBehind** — страница связывается с кодом через эту директиву. В .сs файле находятся обработчики события, связанные с этой разметкой.

При перетаскивании в код страницы добавляется директива **Register** – регистрирует **Control** на этой странице (в ней указывается префикс (**TagPrefix**) нового контрола, тэг (**TagName**), и ссылка (**Src**) на сам файл контрола).

После того как перенесли, то появится такая строчка:

<uc1:MyControl ID="MyControl1" runat="server" InputButton1="Отправить информацию" InputButton2="Отказаться от отправки" Gender="мужской" Faculty="Информационных технологий"/>

Мы можем указать полям значения по умолчанию.

Чтобы преобразовать веб-форму в пользовательский элемент управления, выполните следующие действия:

1) Уберите все теги html, head body и form

2) Если отображается на странице директива @Page, измените на @Control

3) Включите атрибут className в директиву @Control, чтобы пользовательский элемент управления печатался строго при её создании

4) Присвойте элементу описательное имя и измените расширения файла с aspx на ascx

# **13. ASP.NET: кэширование в ASP.NET, назначение и виды кэширования, послекэшевая подстановка, политика устаревания данных. (8 лаба)**

**Кэширование** – техника хранения в памяти копий некоторой информации для повторного применения. Повышает производительность и масштабируемость приложения.

**Виды кэширования:**

- кэширование вывода – можно сохранить копию сгенерированной страницы отправленной клиенту; следующий запрос на такую же страницу не потребует её генерации – используется страница из кэша. Возможно кэширование фрагментов страницы.

- кэширование данных – программист может сохранить в кэше данные (например, в виде DataSet); страницы могут проверять наличие данных в кэше и использовать её; возможно кэширование источников данных (DataSource, ObjectDataSource и XmlDataSource). В этом случае управление кэшем осуществляется автоматически.

- кэширование ввода – outputCashe duration, varyByParam = none.



Если OutputCashe Duration=10, то не обязательно 10 сек будет страница храниться в кэше: если для IIS потребуется память, то он может освободить кэш.

Когда говорим про веб-приложения, то есть 3 вида кеширования:

**Кэширование вывода на стороне** сервера – делаем запрос на сервер. Он обрабатывает и формирует ответ. Он этот response запоминает и при следующем запросе не формирует, а сразу отдает. Можно делать 2 способами:

Ø С помощью http-протокола попросить закэшировать. И протокол может быть закэширует, а может и нет.

Ø С помощью механизмов asp.net.

**Кэширование вывода на стороне браузера (клиента)** – извлекаются из браузера. Полностью обеспечивается с помощью http-протокола.

**Кэширование данных**, если делаете запрос и для формирования ответа нужно залезть в БД и на основе этих данных сформировать ответ, то залазят заранее в БД, создадут объект и при запросе не лезет в БД, а берет с объекта.

Если ответ формируется на основе БД, и ответ положен в кэш для выдачи результатов пользователя, то в БД уже могут произойти новые изменения и тогда выйдет, что ответ немного запаздывает и даёт не полноценные данные. И только после повторного объявления выдача будет с полными данными из БД при условии, что не произошло новых изменений во время хранения кэша. В большинстве случаев это торможение выдачи информации является приемлемым для приложения и так делают, но могут быть случаи, когда так делать нельзя. Обычно кэш равен примерно 10 секунд, это время является наиболее оптимальным.

Для того, чтобы понять, как работает кэш, нужно разобраться, на каком этапе формирования приложения он делается и кем. Нужно знать, что такое **ключ кэширования**. Это ключ, с помощью которого данные записываются в кэш и извлекаются оттуда. Другими словами, те данные, которые кэшируются запоминаются в специальном объекте с названием кэш. И данные туда записываются как пара «ключ-значение». Важно понимать, как формируется ключ, по которому данные записываются в кэш и затем извлекаются. Также нужно понимать на каком этапе осуществляется кэширование данных, а на каком этапе осуществляется извлечение из кэша.

Предположим, у нас есть программа, обрабатывающая запросы, приходящие от клиента. Если мы хотим кэшировать запросы, то перед тем как отправлять запросы к БД, нужно проверить есть ли такой ответ уже в кэше. Если такой ответ уже в кэше существует, то тогда дать ответ из кэша, в противном случае – отправить запрос к БД.

Есть специальный компонент приложения Http-модуль (системный), который занимается кэшированием. Запрос, который идёт на сторону сервера, перехватывается на сервере Http-модулем. Http-модуль в объекте кэша проверяет, есть ли уже такой закэшированный ответ, далее он проверяет, не устарел ли кэш и при положительном ответе отправляет ответ. Если же ответ в кэше устарел или вообще не существует, то Http-модуль пропускает запрос к приложению (ресурсу).

Приложение работает. Какая его основная задача? Подготовить ответ на запрос. Приложение готовит ответ, записывает его в кэш и отправляет обратно клиенту. Далее снова клиент отправляет запрос, его перехватывает Http-модуль, смотрит, например, что такой ответ есть и сразу его отправляет, не пропуская запрос дальше. Когда время действия кэша истекает, Http-модуль пропускает запрос дальше к ресурсу. Записывает ответ в кэш и отправляет его клиенту.

Теперь осталось разобраться, по какому ключу записывает данные в кэш Http-модуль и по какому извлекает эти данные (они разные). Есть запрос, который описывается с помощью URI. Один и тот же запрос будет приходить на один и тот же URI. С помощью URI и можно формировать ключ.

Что является ключом кэширование – главное отличие различных типов кэширования.

**Кэширование вывода**

Кэширование по URI без параметров

*OutputCashe Duration VaryByParam = None*

Здесь ключ кэширования – только URI (!!! URI - чистая часть идущая до слэша, параметры здесь не учитываются).

Например, извлечь информацию о студентах и извлечь информацию о конкретном студенте (параметр) – будет является ОДИНАКОВЫМ запросом при таком кэшировании.

**Duration** *—*время хранения кэша. Пример, Duration= «10» означает, что в течение 10 секунд абсолютного времени запрос хранится в кэше. А по истечении 10 секунд устаревает.

Кэширование по URI с параметрами

Позволяет указать имена параметров, которые должны быть включены в ключ кэширования. Параметр

*ОutputCashe VaryByParam = “Parm1”*

Позволяет кэшировать страницы, вызываемые по запросу с различными значениями параметра Parm1. Остальные параметры в расчет не берутся. Реально заметить можно только для двух сессий. Теперь к URI присоединяется параметр и на каждый новый параметр будет создаваться новая запись в кэше. Если указано два параметра, то второй параметр будет игнорироваться, так как мы указали только один параметр.

*Например*, у нас есть первый параметр: курс и второй параметр: номер группы. Так вот в ключ кэширования включится только первый параметр (курс) и все запросы, в которых будет одинаковое значение первого параметра будет одинаковый ответ. То есть для всех групп на 2-ом курсе ответ будет одинаковым.

Запись двух параметров:

*http:/www.xxx/FFF?parm1=a&parm2=c*

*http:/www.xxx/FFF?parm1=b&parm2=d*

*Таким образом URI, которое считывается, – удлиняется.*

Самый неприятный момент кэширования. Кастомное кэширование.

Параметр OutputCashe VaryByCustom. Настраиваемое управление кэшем. **Custom-кэширование** означает написание своей собственной функции, которая формирует ключ кэширования. Этот ключ программист сам формирует из своих соображений так, как ему надо. Такая функция должна быть помещена в Global.asax файл (говорили раньше). Имя функции должно быть следующим: GetVaryByCustomString(…). Эта функция получает два параметра:

*HttpContext* – из него можно получить request и response

*string* – сюда передаётся информация, указанная в VaryByCustom, которая в свою очередь указана в директиве VaryByCustom.

Кто вызывает эту функцию? Пришёл запрос со стороны клиента, его встречает Http-модуль и видит, что для этого запроса предусмотрено Custom-кэширование, далее вызывает функцию GetVaryByCustomString, получает ответ в виде строки, идет в кэш и проверяет есть ли такой response с таким ключом кэширования и при наличии отправляет ответ, при отсутствии отправляет запрос к ресурсу, ресурс формирует response, вызывает функцию эту, получает строку-ключ и записывает в кэш по ключу кэширования. Для того, чтобы при следующем запросе Http-модуль снова вызвал ту функцию, получил ключ, залез в кэш, извлек ответ по ключу и отправил его.

GetVaryByCustomString оборачивает получаемую строку string таким образом, чтобы получить ключ кэширования.

**Зачем нужен параметр VaryByCustom?** При наличии различных страниц с одинаковой системой кэширования, этот параметр помогает идентифицировать страницу, с которой пришел вызов функции.

GetVaryByCustomString вызывается Http-модулем при запросу и ресурсом при формировании ответа, находится в Global.asax-файле и формирует строковый ключ.

Кэширование по заголовкам (хэдерам-headers).

**VaryByAcceptHeader**

Редко используется, так как мы в качестве клиента используем браузер и браузер сам формирует заголовки запроса и поэтому такое кэширование применяется в тех случаях, когда у вас есть собственный Http-client и вы сами управляете заголовками запроса (можете указать свой заголовок, значение которого будет являться ключом кэширования). Это возможно только в том случае, если пишешь сам клиента.

Параметр OutputCashe VaryByAcceptHeader (в кэше страницы с разными заголовками запроса).

**Кэширование части страницы:** фрагментарное кэширование (используется для пользовательских элементов управления), послекэшевая постановка (не может быть использована для пользовательских элементов управления)

**Послекэшевая постановка (Substitution)**

Бывает необходимо закэшировать всю страницу, за исключением какой-то небольшой части (дырочки в бублике, например). То есть, чтобы вся страница была в кэше за исключением некоторых элементов. *Например*, вся страница постоянная, а только поле с валютой обновляется. Для реализации такого примера нужно использовать специальный серверный элемент Substitution, в котором указано имя метода серверного объекта приложения, который должен вернуть строку вместо этого Substitution.



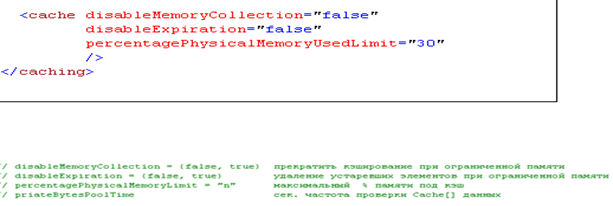
Как это происходит? Http-модуль извлекает данные из кэша по какому-то ключу и видит маленькую дырочку, которую необходимо обновить, тогда он вызывает метод, который прописан в коде и возвращаемое значение подставляет в Substitution.

*Пример.* Только строка со временем будет обновляться, а все остальное будет храниться в кэше.

Профиль кэша:



Конфигурация кэша:



Кэширование данных

Если в объект Session ничего не записано, то при новом запросе создаётся новая сессия.

ОТВЕТ НА ЭКЗАМЕН:

В кэшировании участвует **Http-модуль** и **ресурс**. **Http-модуль** извлекает из кэша response (ответ) и отправляет его клиенту, ресурс записывает его в кэш. И тот, и другой использует ключ кэширования, разные типы кэширования различаются ключом кэширования (по URI, по URI и параметру, кастомное (пишем функцию, которая возвращает ключ кэширования), по заголовку запроса (если сами пишем клиент, который записывает заголовок), кэширование указывается с помощью специальной директивы OutputCache, в которой мы указываем тип кэширования и Duration (промежуток времени, в течение которого мы хоти хранить данные в кэше). Кроме того, есть специальный механизм, который реализуется с помощью тега Substitution. Он делает послекэшевую подстановку. Мы можем извлечь данные из кэша и на них сверху наложить ту строку, которая заменит Substitution в окончательном выводе.

**Кэширование данных**

Кэширование данных осуществляется тоже с помощью объекта кэш (системный объект, который предназначен для записи данных). Этот объект имеет метод insert:

**Mykey** — ключ

**Mc** — ссылка на какой-то объект

**DateTime.MaxValue** — время хранения данных

**TimeSpan.FromSeconds(10)** — время скользящего устаревания

**TimeSpan.Zero** — означает политику абсолютного устаревания

**TimeSpan** — объект для представления интервала времени

С помощью этого метода response записывает туда данные, с помощью этого метода ресурс записывает данные (response) в кэш. Мы сами также можем с помощью этого метода записать туда данные и затем использовать этот кэш (*например*, запись трудно извлекаемых данных из БД, если требуется много ресурсов и времени для получения таких данных, но сама БД меняется нечасто). Тогда имеет смысла извлечь эти данные, поместить их в оперативную память и не лазить за ними постоянно, а брать из памяти. Объект кэш находится в оперативной памяти. Когда данные записываются в кэш через insert, то выходит, что данные записываются в оперативную память.

Каким образом можно записать? Когда данные записываются в память, имеет место такое понятие, как **политика устаревания данных**:

**Абсолютное устаревание.** Это значит, мы поместили данные в кэш и этим данным указали, что их время хранения 15 секунд. В таком случае ровно через 15 секунд эти данные устареют.

**Скользящее устаревание.** Если здесь указать 15 секунд в качестве времени хранения данных, но данные извлекутся раньше (через 3 секунды), то время ранения данных сдвинулось на 15 секунд, то есть является скользящим. Если очень часто обращаться к данным, то они могут никогда не устареть и, следовательно, никогда не обновиться.

От какой-либо зависимости.

Для кэширования вывода response используется абсолютное устаревание.

**Последовательность удаления кэша**

Кэш объект имеет замечательное свойство: если серверу нужна дополнительная память, когда собственной не хватило, то сервер удаляет кэш. При чем удаляет кэш не сразу, а по частям. Первым будет удален раздел *Low*, самый ненужный. Затем при необходимости новой памяти будет удалён *BelowNormal* и так далее, последним будет удалён блок кэша *High*.

Что можно сделать с помощью кэша данных?

Организовать кэширование с абсолютном кэшированием и скользящим, можем поместить кэш в различные области приоритета. Что означают раздел кэша? Раздел кэша отличается своим приоритетом при очистке кэша. В каком порядке будет происходить удаление.

**Зависимости кэша**

Есть специальный механизм, с помощью которого можно установить зависимость кэша (зависимость его устаревания), от каких-то внешних событий, например, от других кэшированных элементов, от файлов или папок, от запросов к БД. Что означает установить зависимость?

*Мы говорим: «Записываемые в мой кэш данные устаревают в том случае, если устарели другие данные, изменилась дата последнего изменения в таком-то файле, обновилась дата изменения любого файла в заданной папке, данные перезаписаны».*

И кроме того есть возможность установить агрегатную зависимость: если мы можем установить зависимость кэша от устаревания кэша и устаревания файла или устаревания нескольких кэшей, то есть мы можем агрегировать несколько событий и строить одну зависимость для кэша.

Есть кэш данных и при записи в кэш данных можно указать одну из политик устаревания (абсолютная, скользящая, от зависимости). Кроме того, можно указать раздел кэша, в который мы собираемся записать кэшируемые данные. Раздел отличается приоритетом при освобождении памяти путём очищения памяти. Необходимо знать про приоритеты.

Для кэширование данных используется объект Cache:

- Безопасный для потоков, если сами объекты являются безопасными (не требует блокирования)

- Элементы объекта удаляются автоматически

- Элементы объекта поддерживают зависимости

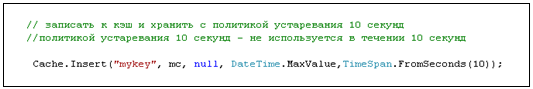
- Объект привязан к домену приложения (после перезапуска домена – новый кэш)

- Не разделяется в серверном кластере

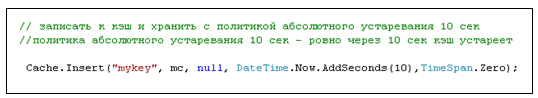
Session тоже сохраняется в Cache со скользящей политикой устаревания равной timeout.

Политика устаревания данных в кэше:

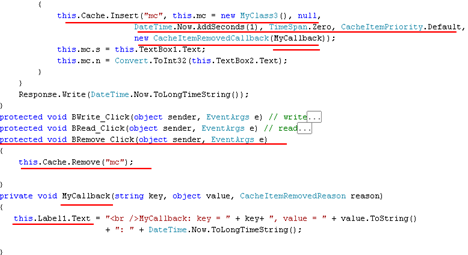
- политика скользящего устаревания – задаётся время неиспользования элемента хранения



- политика абсолютного устаревания



Приоритеты кэша и процедура обратного вызова.

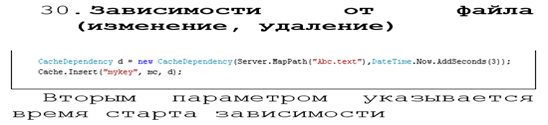


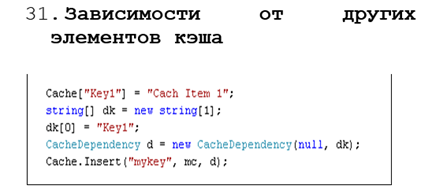
Зависимости кэша:

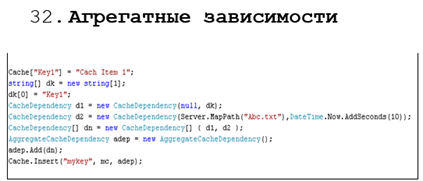
- От других кэшированных элементов

- От файлов или папок

- От запросов к бд







Кэширование на стороне клиента: управление заголовками

Last-modified/If-Modified-Since

Etag/If-None-Math

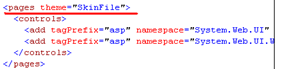
Expired

Cache-Control:max-age

# **14. ASP.NET: Темы (skin), назначение и принцип применения. (7 лаба)**

**Назначение**: придать стилевое единообразие страницам.

**Тема** — поименованная коллекция образцом элементов управления. Эти все образцы имеют свойства и их свойства могут быть образцовыми. Темы могут включать: css, графические элементы, skin-файлы. Темы основаны на серверных элементах управления, т.е. применяются на стороне сервера.



**Skin-файл** — файл, которые хранит в себе образцы элементов. Если мы с вами разрабатываем тему должны положить ее в папку **App\_Themes**, в которой должен лежать *skin-файл*. Когда мы затаскиваем стандартные элементы, к ним уже применяется тема, которая лежит глубоко. Когда мы разрабатываем новую тему, мы перекрываем ту тему, которая уже есть на элементах.

**Как создать skin-файл**

Нажимаем на 7 ПКМ – добавить – создать элемент – веб – веб-формы – **файл обложки веб форм**. Теперь мы можем добавлять коротким путем (2й раз): ПКМ 7 – добавить – файл обложки веб форм. Наш скин появится в папке App\_Themes.

Css-файлы реализуются бразуером (на стороне клиента), а skin-файлы (т.е. темы) применяются на сервере и клиенту(браузеру) отправляется вся визуализированная страница. Css стили создаются на основе html разметки. А skin создаются на основе элементов управления.

Если добавлю 2 кнопки, то будут 2 одинаково оформленные кнопки:

Можно отказаться от темы — добавить атрибут EnableTheming = false, она не будет применена:

Можно внутри скин-файла сделать несколько экземпляров одного и того же элемента. Для этого нужно указать SkinID:

Никто нам не мешает совмещать скины с таблицами стилей.

Динамически можно установить тему в программе:

Статически по умолчанию она указана в самом начале файла веб-страницы:

# **15. ASP.NET Мастер-страницы, назначение и принцип применения. (7 лаба)**

**Мастер-страница** – это пользовательский элемент управления. Нужен для повторного применения. Так, мы можем шаблонизировать страницу. Cтраницы, к которым применена одна и та же мастер-страница, будут иметь одну и ту же структуру (layout).

**Мастер-страница** – страница-шаблон для создания однотипных страниц.

включает в себя теги и контент-плейсхолдеры для данных из страниц, в которых данная мастер-страница будет применена.

*Мастер-страницы используются для создания единообразного, унифицированного вида сайта. По сути мастер-страницы – это те же самые представления, но позволяющие включать в себя другие представления. Например, можно определить на мастер-странице общие для всех остальных представлений элементы, а также подключить общие стили и скрипты. В итоге нам не придется на каждом отдельном представлении прописывать путь к файлам стилей. А используемые на мастер-страницах плейсхолдеры позволят вставить на их место другие представления.*

**Мастер страниц** предназначен для повторного использования шаблонов.

Для создания мастера страниц – 7 ПКМ – добавить – создать элемент – веб – веб-формы – **главная страница веб форм**. После этого, можно будет добавлять коротким путем.

Мы создаем обычную форму, а потом прописываем данный атрибут MasterPageFile="~/Site1.Master". так как в Site1.master уже прописано все это!

Нам не надо писать всякие head, body, мы должны вытянуть **content**.

Нам обязательно нужно указать **contentPlaceholder** и сюда будет вставлен текст, который мы прописали в aspx форме.

Таким образом, мастер страниц задает какой-то шаблон, который потом можно будет вставлять в разные aspx.

*Посмотрим на код мастер-страницы:*

Сверху у нее указан тег Master, что говорит о том, что это мастер страница.

В теле мастер страницы видим теги ContentPlaceHolder, их несколько, у каждого свой id. В эти теги будет вставляться контент из страниц, которые используют данную мастер-страницу.

*Теперь посмотрим на код веб-странички, которая использует мастер-страницу.*

Сверху есть тег Page, который показывает, что это обычная asmx-страница. В свойстве MasterPageFile мы указываем, какую мастер-страницу будем использовать. Теперь эта страница будет создана на основе шаблона указанной мастер-страницы.

Далее мы видим, что у этой странички нету секций head, body. Нам нет нужды их указывать, потому что мы используем мастер-страницу, где это все уже есть. Здесь нам нужно просто указать контент для заполнения тегов ContentPlaceHolder мастер-страницы.

Для этого используются теги Content. У тега Content есть свойство ContentPlaceHolderID, в котором мы можем сослаться на конкретный ContentPlaceHolder мастер-страницы.

В итоге при рендеринге данной страницы будет отображена её мастер-страница с контентом из данной страницы (из тегов Content).



****

# **16. ASP.NET: Основные системные объекты (request, response, session, application), системные http-обработчики.**

**Объект Application**

Начнём с объекта **Application**. Повторим каким образом обрабатывается запрос, который пришел на сервер. Мы в браузере набрали строчку и отправили запрос (**request**), который поступает в операционную систему windows. Там принимается сообщение. Этот запрос встречает специальный **http-драйвер**, который встроен в ядро ОС, и называется он **Http.SYS**. Он предназначен для того, чтобы принимать **http-запросы** и отправлять **http-ответы**. Здесь написано, что он находится в IIS, на самом деле он встроен в **ОС**. В рамках самого IIS сервера есть специальная программа – **WAS**, которая **Windows Activation Service** (**сервис активации**), он первый раз запускает экземпляр приложения. На каждый запрос запускается экземпляр приложения. Экземпляр приложения выполняется в рамках application pullа. **Application Pull** — это специальный механизм, позволяющий выполнять методы экземпляра приложения. Другими словами, это механизм потока. Мы, когда создаем приложение мы любое приложение приписываем к некоторому application pull’у. Если мы припишем 2 приложения к 1 пулу, то они одновременно работать не смогут, они будут по очереди выполняться. Если мы хотим работать быстрее, мы должны для каждого приложения создавать свой application pull. Кроме того, он позволяет указать версию .NET фреймворка в рамках которого будет выполняться приложение. Когда приходит наше приложение, оно приписывается к какому-то application pull, и WAS запускает наш экземпляр.

Пулы сами по себе работают параллельно, но в рамках одного пула приложения выполняются по очереди. Приложение для каждого запроса создается свой экземпляр – **экземпляр приложения**. Но есть часть нашей программы, которая является *общей* для всех экземпляров, и создается 1 раз при запуске приложения на узле IIS. Мы с вами написали приложение, установили его на IIS, и когда мы его там стартанули образовалось приложение (та часть, которая общая для всех), а при каждом запросе создается новый объект.

Т.е. у них есть общая часть и эта часть вся сосредоточена в объекте, который называется **application (или контекст)**. Если вы к одному и тому же приложению обращаетесь с разных браузеров, то они видят один и тот же общий application. Application перегружается (меняется) в следующих случаях:

*1.* *если мы меняем webconfig*

*2.* *если вносим изменения в global.asax (здесь мы создаем листенеры)*

*3.* *при недостатке памяти*

*4.* *при перезагрузке сервера*

С точки зрения программирования он нужен для использования, встроенного в него **словаря**. Мы можем в объект запомнить свои данные, т.е. это один из уровней памяти (общая память для всех приложений). Другими словами, если есть 1 компьютер и вы в Минске, у вас есть сервер с приложением в Москве. Если юзер, который из Минска запишет данные в application, который в Москве, то его увидит тот, что в Москве, потому что он общий. В нём находится словарь.

Таким образом application можно использовать как почтовый ящик. Мы можем в него записывать не только число, строку, но и ссылку на объект. Создать объект и записать туда ссылку, и тогда он не будет уничтожаться, потому что на него есть ссылка.

Весь жизненный цикл приложения мы можем проследить и врезаться с помощью специального файла, который называется global.asax. Там есть все события, которые связаны с жизненным циклом приложения. Вверху есть свойство application start, и вот когда оно стартует, вы можете записать какое-то свойство туда. Предположим, хотим, чтобы все видели 1 объект. Пишем в application создать объект. С помощью application можем перехватывать все-все запросы, не только идущие по вашему соединению, но и запросы по всем-всем соединениям. Это можно сделать в функциях Prerequest, postrequest.

**Объект Session**

Если application является общим для всех, то session является общим для всех потоков одного соединения. Этот объект соответствует соединению и хранит в себе всю информацию о соединении с пользователем. Если есть запрос мы говорим, что http протокол не помнит своего состояния. Он отправил запрос, получил запрос и всё забыл. И при следующем запросе всё заново. Вот сессия нужна для того, чтобы сохранять данные в рамках одного сеанса. При первом запросе в вашем запросе нет никакого заголовка куки, поэтому он считает, что это самый первый запрос, и тогда сервер создает для него сессию. И при следующем ответе он пишет заголовок set cookie и в нём указывает id сессии, и браузер запоминает этот id сессии у себя и запоминает под именем нашего uri и потом при всех запросах он сопровождает их этим заголовком с id сессии, и по этим cookie сервер узнает в какой сессии относиться этот запрос. *Таким образом все запросы будут выполняться в рамках 1 сессии (будут помечены одним и тем же id сессии).* Сессия живет время, которое называется **timeout\_session** это системный параметр. Обычно, время timeout\_session колеблется от 20 до 30 минут. Это максимальное время между 2 запросами. Если мы отправили запрос, а потом 20 минут ничего не делали, то сессия разрушается. Если мы хотим хранить данные в течение 1 сеанса, то мы должны хранить их в сессию. У нас образовалось 2 уровня памяти:

*1.* *Общее для всех*

*2.* *Общее для 1 соединения*

Если Вы сделаете 2 запроса с разных браузеров, то у вас будут разные сессии. Но нужно, чтобы cookie был включен, если отключите cookie, то потеряете сессию.

**Объекты Request и Response**

Итак, пришел к нам запрос. Сервер обработал битовую последовательность и превратил её в программный объект и в поля затолкал данные.

**Request**:

· метод (get/post/put/option и тд. Основные get и post)

· URI (описывает месторасположение сервера)

· версия протокола (HTTP/1.1)

· заголовки (пары: имя/заголовок)

· параметры (пары: имя/значение)

· расширение(тело)

**Response**:

· Версия протокола

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

· Пояснение к коду состояния

· Расширение

· Заголовки (пары: имя/заголовок)

Задача нашего приложения — это заполнить объект response и отдать его серверу, а сервер его преобразует в битовую последовательность, которая отправляется на сторону клиента. Таким образом, образом экземпляры request и response они живут вместе. Т.е. жизненный цикл их совпадает. Html-разметка, которую мы отправляем на сторону браузера, хранится в теле. Браузер это узнает по специальному заголовку (text/html).

Response: основные методы

Write;

Redirect;

Flush;

BinaryWrite;

WriteFile;

ClearHeaders;

ClearContent;

Close (закрывает соединение по данному сокету)

**В ответе может быть код состояния**:

1. Информационные: 1xx

2. Успешный ответ: 2xx

3. Переадресация: 3xx

4. Ошибка клиента: 4xx

5. Ошибка сервера: 5xx

**Фильтр** – это http-модуль

**Контекст** – application

**Листенер** прописываем в global.асах (позволяют врезаться в жизненный цикл приложения)

**Хэндлер** – это наш **http-драйвер** (**http-обработчик**) – это простейшее asp.net приложение, которое предназначено для обработки запроса и формирования ответа. Это класс, который реализует интерфейс **IHttpHandler**.

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

**Веб-сервисы (технология ASMX)**

**Веб-сервисы** – это особый вид веб-приложений, которые по-другому называются asmx-сервисы. **Веб-служба, или веб-сервис** — это удалённый программный объект, методами которого мы можем пользоваться.

У веб-приложений есть веб-интерфейс. Со стороны сервера разметка посылалась на клиента. Конечным пользователем такого интерфейса является человек.

Пользователем веб-сервиса является программа. Веб-сервис вызывается другой программой, которая будет использовать интерфейс веб-сервиса. По сути, другая программа вызывает методы веб-сервиса. Такая программа может быть написана на любом языке и на любой платформе: главное, чтобы она выполняла правила взаимодействия с веб-сервисом. Эти правила определяются технологиями SOAP, WSDL, XML, UDDI.

**SOAP (Single Object At This Protocol)** – протокол прикладного уровня, которому нужен какой-то «транспорт» для пересылки данных. В качестве транспорта может выступать HTTP-протокол, SMTP-протокол и др. SOAP представляет собой правила взаимодействия между клиентом и сервером. В качестве клиента выступает какая-то программа, а в качестве сервера – сервис.

**HTTP** — протокол, описывающий взаимодействие между клиентом и сервисом.

**Протокол** – правило взаимодействия двух абонентов, соединенных в сети.

SOAP встраивается в HTTP, т.е. сообщения в формате SOAP передаются в теле запроса и ответа HTTP.

Формат SOAP фактически является XML документом. SOAP построен на основе XML, т.е. по сути между клиентом и сервером перемещаются XML-документы определенного формата, называемые **envelope**.

**SOAP** – встроенный в HTTP протокол, который воздействует на тело и заголовки.

**WSDL (Web-Service Definition Language)** — формально описывает веб-сервис. Это XML-документ, который содержит в себе исчерпывающую информацию о сервисе. Если где-то в сети есть сервер, на котором установлен сервис, мы в большинстве случаев можем получить WSDL этого сервиса. **WSDL** – это тот документ, который содержит в себе исчерпывающее описание сервиса. **Исчерпывающее** – это такое описание, которое позволяет нам практически в автоматическом режиме связаться с этим сервисом (сделать запрос сервису).

**UDDI**. Предполагалось, что веб-сервисов будет много, и для того, чтобы создать реестр сервисов, было придумано специальное хранилище, и запросы к этому хранилищу можно отсылать с помощью протокола **UDDI**. Протокол не нашел широкого распространения.

**Сервис**, с точки зрения программирования, — это класс, который является производным от класса WebService.

В первую очередь необходимо сделать 2 вещи:

1. сделать аннотации для элементов этого класса;

2. прописать в web.config дополнительные параметры.

Когда мы прописываем аннотацию, мы пишем дополнительную информацию в мета-данные. Т.е. мы говорим: пометь этот класс результатом работы этого класса. Генерируется специальная информация, которая записывается потом в мета-данные. В результате можно прочитать эту **dll** или **exe** и увидеть, что этот класс помечен как веб-сервис.

**Namespace** – это идентификатор, который записывается в форме URI или URN. Он нас ни к чему не обязывает: мы не обязаны укладывать по этому URI какой-либо сервис. Все сообщения, которые будут ходить между клиентом и сервисом, будут помечаться этим идентификатором.

Второй параметр (**webServiceBinding**) определяет Binding. **Binding** — это часть правил протокола связи, в которой определяется, каким образом будет осуществляться связь между клиентом и сервисом. Для asmx протокола выбор небольшой, и мы можем выбрать один из стандартных способов связи – **basicProfile** — который говорит, что в качестве транспорта будет использоваться HTTP-протокол, что не будет организовываться сессия.

*Программист должен представлять веб-службу как удаленный программный объект, к методам которого мы можем получить доступ. У метода есть имя, есть URI и есть методы, которые мы можем вызывать. Для доступа к методам мы используем SOAP, а сам объект описывается с помощью WSDL.*

С помощью **webMethod** мы помечаем методы удаленного объекта, к которым можно получить доступ извне. Если не пометить, данный метод не отобразится в WSDL. У **webMethod** есть параметр description, в который записывается описание того, что метод делает. Мы можем проверить сервис, нажав ПКМ и выбрав пункт «просмотреть разметку». Стандартный хэндлер, обрабатывающий asmx-страницы, принимает все запросы и формирует ответы. Проверить сервис на локальном компьютере: ПКМ – «просмотр в обозревателе».

*Веб-сервис, как правило, работает в рамках какого-то веб-приложения. Сервис должен где-то «хоститься», у него должен быть хост. И в качестве хоста для asmx выступает веб-приложение.*

Когда проверяем на локальном компьютере, запускается веб-приложение, с браузера автоматически делается запрос и считывается WSDL.

На каждый запрос создается новый экземпляр сервиса. Т.е. сервис не хранит своего состояния. Чтобы сохранить состояние, нужно применять специальные меры:

1. нужно пометить методы параметром *EnableSession=”true”*, т.е. разрешить использование сессии. **Сессия** – системный объект, который создается для соединений и хранит информацию о соединении.

2. разрешить использование **cookie**. Это делается на клиенте: в **web.config** устанавливается *allowCookie=”true”*.

**Применение веб-сервиса**

Можно написать WinForms-приложение, в котором вызываются методы этого удаленного сервиса. Приложение необходимо соединить с веб-сервисом с помощью «**добавить ссылку на службу**».

Сгенерируется прокси-класс, который нужен для того, чтобы притвориться, что он и есть служба. Он из себя изображает локальный класс, который на самом деле является удаленным. Он создает иллюзию, что мы работаем не с удаленным, а с локальным классом. Он принимает параметры. Переправляет их по сети, делает запрос к сервису удаленному, сервис делает ответ, упаковывает в envelop-запрос, отправляет прокси-классу, тот его возвращает клиенту, распаковывает.

**В КРАТЦЕ**

**Веб-сервисы** – тип asp.net приложения. Особенность его в том, что в качестве пользователя выступает приложение (программа). База, на которой построен веб-сервис: Протокол SOAP, WSDL, XML.

WSDL генерируется автоматически по нашему описанию сервиса.

При разработке сервисов существует два подхода:

1. пишется код и по нему генерируется WSDL;

2. пишется WSDL и по нему генерируется (пишется разработчиком) код.

По сути **WSDL** – исчерпывающая спецификация сервиса.

В качестве транспорта для нашего сервиса для передачи информации используется **HTTP-протокол**, т.е. наша информация в формате **SOAP** передается в теле и заголовках HTTP-протокола.

Разработка самого сервиса сводится к разработке класса. Этот класс наследует определённый тип, сам класс сопровождаем специальными аннотациями, где помечаем, что класс является веб-сервисом, и помечаем методы, что они являются методами этого веб-сервиса. Именно эти аннотации позволяют VS потом сгенерировать на основе этого WSDL. Кроме того, метаинформацию можем сопроводить **description** (описаниями).

Написание приложения для веб-сервиса сводится к тому, что необходимо связать приложение с веб-службой и создать **прокси-класс**.

По умолчанию веб-служба не хранит своего состояния и сессия недоступна в методах. Для того, чтобы был доступен объект **session**, необходимо указать специальный параметр **enableSession** в атрибуте этого метода. Чтобы поддержать сеанс, необходимо обеспечить **cookie**.

При каждом запросе (вызове метода) будет создаваться новый экземпляр объекта. Можно сохранять данные с помощью **session** и **application**. **Application** создает глобальный объект для всех сессий.

У нас существует 2 типа интерфейсов: **RPC**, **REST**.

**RPC** - удаленный возов процедур. Сюда относится asmx-сервисы и wcf-сервисы.

**REST** служба описывается по определенным правилам, которые описаны для REST -интерфейсов. Сюда относится только Web Api.

**SOAP**, фактически – формат сообщений, отправляемых в теле и заголовках HTTP. Последняя версия SOAP – 1.2.

SOAP-сообщение состоит из трех частей: *envelope, header, body*.

WSDL используется для генерации прокси-файла. Если есть сервис, то по стандарту, если мы вызовем URI сервиса, поставим знак вопроса и напишем wsdl, то нам обязано выдать wsdl-файл.

# **18. ASP.NET: Http-обработчики ASP.NET, назначение, порядок разработки, принципы применения. (3 лаба)**

В первую очередь, это самое элементарное asp.net приложение. Когда мы устанавливаем asp.net, то устанавливается библиотека asp.iis.api и стандартные обработчики, которые уже написаны до нас и входят в состав .NET. Также мы можем сами писать HTTP-обработчики. Любое ASP.net приложение сводится к выполнению какого-либо обработчика.

**Задача HTTP-обработчика**: получить запрос http-request и отправить ответ http-response.

Каждому расширению файла соответствуюn некоторые обработчики.

**Каким образом работает IIS?**

IIS получает запрос, смотрит в конец URI, и смотрит, какое расширение стоит там. Если там есть .aspx, то футболит на объект, соответствующий этому расширению.

**HTTP-обработчик**

Он представляет собой класс, который реализует интерфейс **IHttpHandler**.

**Интерфейс** — поименованный набор сигнатур.

**Сигнатура** — это имя функции, что она возвращает и набор параметров.

Интерфейс содержит методы, но не их реализацию. А класс, который наследует данный интерфейс, wcfреализовать все его методы!

HTTPHandler содержит в себе 2 сигнатуры:

*1.* *ProccessRequest*

2. *IsReusable* (не метод, а свойство, которое работает только на чтение)

Если **IsReusable** возвращает **false**, то при каждом запросе создается новый экземпляр этого обработчика. Т.е. делается new GetHandler() и дальше вызываются соответствующие методы.

Пришел запрос, IIS посмотрел его расширение, понял, что этому расширению соответствует обработчик GetHandler, он делает new GetHandler(), и вызывает метод **ProccessRequest**. Он уверен, что он есть, потому что он соответствует интерфейсу. Если мы укажем здесь истину **return true**, то один и тот же экземпляр этого класса будет обрабатывать этот запрос. В этом случае, нам надо думать, как обрабатывать новые запросы, поэтому все делают return false, так проще и каждый обработчик обрабатывается новым экземпляром.

ProccessRequest, в основном, предназначен для получения 2 объектов: request–response. Request содержит в себе всю информацию, которая пришла в виде битовой последовательности и была преобразована в request. А response создается пустым. Когда наш обработчик обработает, сервер получит response, преобразует снова в битовую последовательность и отправит на сторону клиента.

Эти 2 объекта (request и response) инкапсулированы в 3-ий объект, который называется **HTTP context**. И из этого контекста можно получить ссылку на request и ссылку на response.

Для того, чтобы в тело response записать данные, можно использовать метод **write**. С помощью этого методы мы в тело ответа пишем данные. Теперь, завершил свою работу этот метод, сервер получил заполненный response и сервер отправляет его на сторону клиента.

Сам по себе обработчик можно написать несколькими способами.

Самый простой: в папку App\_Code могу поместить этот обработчик.

Как сказать IIS, чтобы он направлял запросы определенного типа с определенным расширением на мой обработчик? Например, как сказать ему, чтобы все запросы с расширением .sww (инициалы Смелова) отправил на мой обработчик?

*— Для этого есть web.config.*

**Указываю в Web.config связи между расширением, методом и типом, который является HTTP-обработчиками для данного типа запросов.**

Мы можем перенести папку App\_Code в исходном неоткомпилированном коде и перенести ее на сторону IIS. Сам IIS откомпилирует ее и приведет в полный порядок. IIS умеет компилировать содержимое папки App\_Code и сделать из нее нормальный загрузочный код.

Если мы установим на IIS наши обработчики и просмотрим содержимое папки, то увидим наши файлы- обработчики.

Можем написать http-клиент, который будет делать такой запрос. Я решил разработать asp.net веб-приложение, у которого при нажатии на кнопку будет вызываться обработчик этой кнопки и в конце концов будет формироваться запрос к нашему обработчику.

Я пишу серверный компонент (на сервере пишу программку), которая выступает в качестве клиента к какой-то другой программе. Т.е. она будет делать запрос к другой программе к нашему обработчику.

Вот я сделал 3 таких кнопки, для каждой из низ свой обработчик нажатия кнопки (onclick) и в них я буду формировать запросы к нашим http-обработчикам. Для этого я использую класс HttpWebRequest, прописываю путь URI. WWW ему до лампочки, ему важно расширение на конце. При нажатии на кнопку Get, я отправляю запрос, получаю ответ и вывожу некоторые данные у себя на странице. Для post и options buttons делаю тоже самое. Создастся объект HttpWebRequest и отправиться запрос по этому URI. А там его встретит обработчик.

На сервере пишу программу, которая выступают в качестве клиента, которая будет делать запрос к нашему обработчику. В них я буду формировать запросы к нашим http-обработчикам.

**Обработчики-генераторы ошибок**

Помимо стандартных обработчиков, которые реагируют на расширения есть еще обработчики, которые генерируют ответ об ошибке. Например, если мы хотим выдать forbidden для нашего сайта, мы можем повесить на get стандартный обработчик.

HttpForbiddenHandler (403)

HttpNotFoundHandler (404)

HttpMetodNotAllowedHandler (405)

HttpNotImplementationHandler (405)

Пример использования HttpForbiddenHandler (403):

Обработка 403 на стороне Web-http-клиента:

Я делаю request, в try перехватываю ошибку WebException и выводится ошибка.

# **19. ASP.NET: AJAX-приложение, AJAX-элементы ASP.NET (ScriptManager, UpdatePanel, Timer, Trigger, UpdateProgress).**

**AJAX** — асинхронный JavaScript и XML. Это механизм (методология), с помощью которого мы можем выполнять асинхронные запросы к серверу.

**Синхронный** — значит, поочередный, **асинхронный** – значит, очередности нет. Если мы хотим синхронизировать 2 процесса, то мы хотим установить порядок их работы. А если процессы асинхронные, мы не знаем, в каком порядке они будут выполняться.

Когда от сервера приходит ответ, он обрабатывается с помощью встроенного в браузер механизма **XMLHttpRequest**, представляющего собой объект JavaScript. Создатели браузеров договорились о том, что он будет себя одинаково вести во всех браузерах. С помощью JavaScript мы можем создать экземпляр этого объекта и с помощью этого экземпляра можем делать асинхронные запросы. Если мы с одной страницы хотим выдать несколько асинхронных запросов, мы должны создать несколько экземпляров **XMLHttpRequest** и каждый из них будет делать свой асинхронный запрос.

Ответ на запрос приходит в функцию обратного вызова **(callback)**. Любая асинхронная обработка всегда устроена так, что обработка ответов осуществляется с помощью данной функции.

Отправив запрос **Send()** объекта **XMLHttpRequest**, мы продолжаем дальше работать, и когда приходит ответ, выполняется **callback**. Причем эта функция вызывается 5 раз в разные этапы выполнения запроса и только на 4 этапе (нумерация этапов с 0) мы получаем статус ответа, где мы можем выяснить его статус (успешен или нет). Это то, что лежит в глубине AJAX.

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

Также и в ASP.net, существуют специальные серверные элементы, с помощью которых мы можем делать асинхронные запросы.

**ScriptManager**

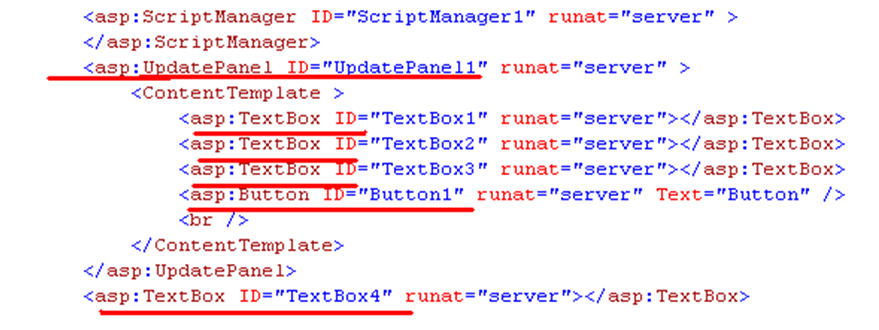
C помощью этого элемента мы подтягиваем JavaScript-код, который обеспечивает работу элементов AJAX.



**UpdatePanel**

Это элемент, между начальным и конечным тегом которого располагается тот фрагмент нашей страницы, который будет обновляться после получения ответа от сервера. Вся страница перезагружаться не будет, перезагрузится только та часть страницы, которая находится в **UpdatePanel.**

**UpdatePanel** требует внутри себя <**ContentTemplate>…</ContentTemplate>.** Внутри этих тегов располагаются те серверные элементы ASP, которые необходимо задействовать в AJAX-запросе.

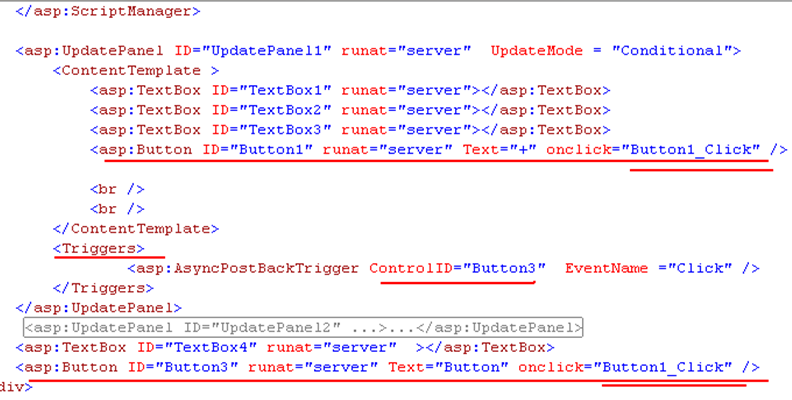


Можно создать несколько **UpdatePanel** и каждый из них будет независимый.

**Триггеры (Trigger)**

Бывает, что у нас есть элемент, который мы хотим включить в **UpdatePanel**, но сам он находится в другом конце страницы. Например, **UpdatePanel** находится в верхнем правом углу, а элемент, который мы хотим обновить в левом нижнем углу. Если нужно поместить серверный элемент, который находится за пределами **UpdatePanel**, в **UpdatePanel,** то это можно сделать с помощью **триггера.**

**Триггер** — связующий инструмент элемента с **UpdatePanel**. Внутри **UpdatePanel** помещаем **<Triggers></Triggers>.**

****

Элементы внутри **<Triggers>** будут работать так, как будто он находится внутри **UpdatePanel.**

**Таймер (Timer)**

Это AJAX-элемент, с помощью которого мы можем с некоторой периодичностью автоматически выполнять асинхронные запросы к серверу. Т.е. похоже на **UpdatePanel**, но не нужно ничего активизировать, он сам будет выполнять свою работу с некоторой периодичностью. Мы указываем периодичность Interval =”5000” – 5 сек. Это значит, что каждые 5 секунд будет выполняться запрос и данные будут передаваться на сервер.



Мы можем на одной и той же странице расположить несколько **таймеров**. И каждый **таймер** будет работать независимо друг от друга. Они будут слать запросы на сервер и получать свои ответы.

**UpdateProgress**

В случае, если запрос оказывается длительным, надо занять внимание клиента, чтобы показать, что его запрос выполняется. С помощью UpdateProgress мы можем показать какую-то картинку или gif во время выполнения запроса. Таким образом можем создать иллюзию, что мы контролируем процесс.

Если его вынести за пределы всех **UpdatePanel**, то он будет выполнять общий для всех выходных данных и триггеров. Если поместить внутрь **UpdatePanel**, то он будет действовать только в рамках этого **UpdatePanel.**

Подводя итоги, **AJAX** — это механизм, с помощью которого мы можем делать асинхронные запросы. Внутри механизма лежит **XMLHttpRequest** — специальный объект браузера, который умеет делать асинхронные запросы и получать ответы с помощью **callback**-функции. В AJAX в качестве формата передачи данные используется либо XML, либо чаще JSON.

AJAX в ASP.net представлен в виде:

· **UpdatePanel**

· **Trigger**

· **Timer**

· **UpdateProgress**

# **20. ASP.NET: MVC-приложение, структура MVC4-приложения, назначение основных компонентов приложения, маршрутизация.**

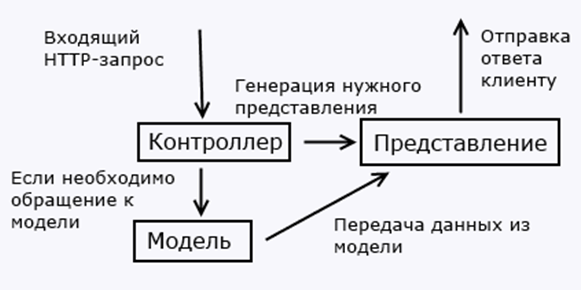
**MVC** (**Model View Controller**) — специальный тип приложений, которое должно состоять из 3 частей:

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.

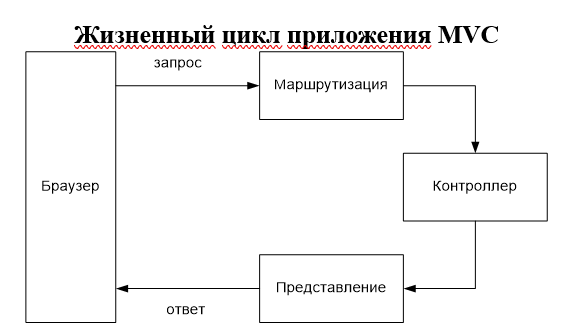
Этот шаблон заставляет нас писать строже, т.е. он нас ограничивает в некоторых возможностях, этим самым приложение получается более строже, легко-расширяемое и удобно-тестируемое.

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.

**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.

Контроллер — Home, акция — Index, id — опциональный параметр, который является необязательным. Таким образом, выдавая uri в запросе мы всегда адресуемся к какому-то контроллеру и к какой-то акции этого контроллера.

Как только маршрутизатор получает запрос с таким uri, он сразу же ищет, есть ли такой контроллер, создает объект этого контроллера и вызывает метод, который находится в этом контроллере.

**Структура приложения MVC**

Папка **Controllers**: папка, в которой находятся классы контроллеров.

Папка **Models**, в которой находятся классы моделей.

Папка **Views**, в которой находятся представления.

**Global.asax** всегда существует, независимо оттого, какое у нас приложение.

**Контроллер (Controller)**

HomeController — стандартный контроллер, который подсовывается вначале приложения. Имя контроллера всегда заканчивается на слово Controller. Внутри контроллера есть методы — акции.

**Представление (View)**

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.

В представлении находится HTML-разметка, хелперы разора (вспомогательные объекты разора) и c#-код.

**Разор** (**Razor**) — это интерпретатор этого view. Это такая программа, которая преобразует все, что находится в представлении в код, понятный браузеру (в html, в js и т.д.).

Чтобы написать C#-код, используется символ «@». После @ можно уже писать с#-код.

# **21. ASP.NET: MVC-приложение, маршрутизатор, принципы работы.**

**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.

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.

**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.

Контроллер — Home, акция — Index, id — опциональный параметр, который является необязательным. Таким образом, выдавая uri в запросе мы всегда адресуемся к какому-то контроллеру и к какой-то акции этого контроллера.

Как только маршрутизатор получает запрос с таким uri, он сразу же ищет, есть ли такой контроллер, создает объект этого контроллера и вызывает метод, который находится в этом контроллере.

# **22. ASP.NET: MVC-приложение, котроллер, жизненный цикл контроллера.**

(это все, что Смелик говорил на лекции)

Контроллер.

Получается, у нас маршрутизатор создаёт контроллер. Чтобы мы смогли это сделать, нам необходим рефлекшн. То есть он должен проверить наличие контроллера класса на уровне рефлекшн, должен его найти.

Класс лалалаcontroller должен наследовать класс httpcontroller.

Объект контроллера создаёт маршрутизатор на основе запроса, который он получил; на основе таблицы маршрутизации (которая прописывает шаблоны этих запросов).

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

(Короче, там ещё сказал, что маршрутизатор ещё передаёт параметры)

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).

И тут он задаёт «провокационный вопрос»: так что, у нас модель создаётся для каждого запроса заново??

Ответ: ДА, модель создаётся 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.

**(ЕЩЕ не закончено, я слушаю дальше)**

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

(Тоже все из лекций в тимсе)

ПРЕДСТАВЛЕНИЕ

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

**(ЕЩЕ не закончено, я слушаю дальше)**

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

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

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

**WCF-сервисы**

**Windows Communication Foundation** — технология для разработки приложений с сервис-ориентированной архитектурой (SOA). WCF является обобщением веб-сервисов, **один WCF сервис имеет много точек доступа**, веб-сервис имеет только одну.

**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.

**SOA**: основные принципы:

Ø границы сервиса определены явно: определяется контрактом и его адресом;

Ø сервисы автономны: сервисы независимы от версий, способа установки;

Ø детали реализации сервиса прозрачны для вызывающего клиента, клиенты и сервисы взаимодействуют исключительно через открытые интерфейсы;

Ø совместимость служб определяется политикой.

**RPC (Remote Procedure Call)** - удаленный вызов процедур. Сюда относится asmx-сервисы и wcf-сервисы.

**REST** (**Representational State Transfer** – передача состояния представления) служба описывается по определенным правилам, которые описаны для REST-интерфейсов. Сюда относится только Web Api.

**Web Api** – особый вид приложений asp.net, который построен на базе шаблона MVC, он предназначен для REST-службы.

**REST FULL API** – интерфейс службы, который поддерживает 4 типа запросов: get post put delete. Каждый из этих запросов интерпретируется определенным образом.

ü *get – аналог нашего select в бд*

ü *post – insert добавить в бд*

ü *put – update изменить*

ü *delete – delete*

Служба эта предназначена для работы с приложениями. Эти приложения будут генерировать запросы и интерпретировать ответы. Для передачи будут использоваться 2 формата: *xml, json*.

Также мы можем в обыкновенное приложение встроить web api.

Если мы это делаем, то у нас должны быть контроллеры, которые именно api.

Для web api свой маршрутизатор отдельный. Маршруты, которые должна идти к api контроллеру, должны идти туда, куда надо. Так как вначале обрабатываются таблицы для web api маршрутизации, а потом для mvc. Для этого отличия, для этого писали *api/{controller}/{action}/{id*}.

**WSDL**

**WSDL** (**Web Services Description Language**) – язык описания Web-сервисов. C помощью этого языка мы можем исчерпывающе описать сервис, чтобы с ним можно было связаться в автоматическом режиме. Последняя версия: WSDL Version 2.0 (не нашла широкого применения, находится в стадии рекомендации, поэтому все работают в версии 1.1). **WSDL** – это реализация XML. WSDL поддерживается консорциумом W3C. г. Спецификация WSDL – это три документа:

*1.* *базовый язык;*

*2.* *шаблоны сообщений;*

*3.* *связывание.*

Каждый WSDL-документ можно разбить на 3 логические части:

*1.* *определение типов данных, отправляемых и получаемых сервисом xml-сообщений;*

*2.* *абстрактные операции – список операций, которые могут выполнены с сообщениями;*

*3.* *связывание сервисов – способ, которым сообщение будет доставлено.*

**Структура WSDL-документа**

Описание типов данных **<types>** – это те типы данных, которые будут использоваться при пересылке информации между клиентом и сервисом. Если будет сложная структура, то её необходимо описать в этом блоке. Здесь находятся типы данных, которые будут использованы в качестве параметра при вызове удалённых процедур, либо в качестве возврата отработавшей удалённой процедуры.

Сообщения **<message>** – здесь записываются сообщения, которые будут ходить от клиента к сервису и наоборот. В каждом сообщении указывается, какую процедуру мы хотим вызвать на сервере и какие параметры мы должны передать.

Тип портов **<portType>.** Тут записаны методы удалённого сервиса, те, которые мы можем вызывать.

**<binding>** – здесь указываются протоколы связи, и информация как мы можем связаться с этим сервисом.

**<service>** – описывается сервис, его имя и порты (его методы) которые можно вызывать.

<!-- структура wsdl-документа -->

<definition>

<types>

<!-- определение типов, используемых web-сервисом -->

<!-- для пл.-независимого WSDL исп синтаксис XML Schema -->

</types>

<message>

<!-- сообщения, используемые web-сервисом -->

<!-- сообщений, может быть несколько -->

<!-- каждое сообщение может сост из нескольких частей -->

</message>

<portType>

<!-- методы, предоставляемые web-сервисом -->

<!-- может быть несколько портов -->

<!-- определены операции web-сервиса и исп сообщения -->

</portType>

<binding>

<!-- протоколы связи, используемые web-сервисом -->

<!-- форматы сообщ и детали протокола для каждого порта -->

</binding>

</definition>

**Пространство имён** – это указание парсеру, который проверяет правильность, какой алгоритм проверки использовать. Для каждого пространства имён есть свой собственный алгоритм проверки. Обычно пространства имён соответствуют XML-схемам (XML-схемы – это специальные документы, с помощью которых можно проверить любою XMLку).

**Хост**

**Хост** –программа, умеет работать с IP. Она умеет принимать сообщения по сети.

**Service**: dll-библиотека.

**Host**: программный модуль, содержащий в себе Service.

**Client**: приложение, использующее Service.

**Proxy**: промежуточная dll-библиотека, эмулирующая работу с Service, как с локальным объектом.

**Endpoint**: конечная точка – сетевой ресурс, которому можно отсылать message.

**Message**: сообщения для обмена данными между конечными точками (формат определяется контрактом).

**Прокси**

**Прокси** (посредник, заменитель) — это компонента, которая притворяется сервисом. Она имеет все те же возможности и такой же интерфейс, как сервис, но находится на стороне клиента. Она притворяется сервисом, но на самом деле она принимает какие-то параметры, отправляет их на сервис, потом получает данные от сервиса и возвращает их клиенту. Это нужно для облегчения работы сервиса и клиента. То есть так работать гораздо проще (всё есть в одном месте).

Сам сервис представляет собой отдельный класс и прокси тоже естественно. Класс не может жить сам по себе, он должен находиться где-то, в каком-то приложении. Приложение, в котором живет класс сервиса, называется **хост**.

**Модели взаимодействия клиента и сервера**

**Полудуплекс**

Режим полудуплексного канала: режим вопроса-ответа. Если в каждый момент времени мы можем отправлять данные и получать ответы – это **полудуплекс**. Мы можем вызвать удаленную процедуру и получить ответ. HTTP протокол работает в режиме полудуплекса.

**Однонаправленная модель** (мы сделали запрос и не ждем ответа), 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.

**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.

**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.

**Порядок разработки:**

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

2. разработка wcf-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 WCF-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.

**В КРАТЦЕ**

**Веб-сервисы** – это веб приложения, в качестве протокола используется HTTP, веб сервисы бывают двух стилей **REST** и **RPC**. Если мы говорим про ASMX, то это реализация RPC. В качестве протокола используется SOAP, который встраивается в транспорт HTTP. **SOAP** – это протокол, описывающий как отформатировать данные с помощью XML, которые будут перенаправляться с помощью HTTP между клиентом и сервером, причём верхний тег у них Envelop и есть такие теги как Header, Body и Fault, обязательным из которых является тег Body. В SOAP протоколе переносится служебная и полезная информация. Эта информация через Envelop ходит и от сервиса к клиенту и от клиента к сервису. Исчерпывающее описание сервиса описывается с помощью **WSDL** файла на языке WSDL, это описание в себя включает описание типов, данных, которые используются для передачи информации между клиентом и сервисом. Там же описываются типы сообщений, которые ходят от сервера к клиенту и от клиента к серверу. Присутствует также раздел «типы портов», в них описаны методы удалённого объекта и для каждого метода есть указания для входного и выходного сообщения. Есть **Binding** – описание как связаться с сервисом, там указан транспорт, для сервиса, который используется (HTTP) и конкретный URI по которому можно связаться. Последний – это описание самого сервиса, в котором обобщающая информация, в нём 2 подраздела, один из которых описывает описание связей по протоколу 1.1, а другой по протоколу 1.2.

Для чего используется WSDL в программе, мы же его не видим? Мы генерируем **прокси-класс на стороне клиента**, **вот он и генерировался на основе WSDL**, **который был получен со стороны сервиса**. То есть если у нас есть сервис, то по стандарту если мы вызовем и поставим у URI поставим знак вопроса и напишем WSDL он обязан выдавать WSDL файл, описывающий его, если сервис общего пользования. Если сервис закрытый, то он ничего выдавать не должен. Прокси будет генерировать на основе WSDL, который считывается со службы, то есть служба должна быть допущена, чтобы к ней можно было обратиться и получить от неё WSDL, потому что на основе этого WSDL Visual Studio создаёт наш прокси-класс, который мы используем на стороне клиента для связи с сервисом.

**SOAP** изначально разрабатывался для RPC-сервисов, но сейчас его обобщили и в версии 1.2 он может применятся и для REST-сервисов.

# **27. ASP.NET CORE: программная платформа, принципы работы, архитектура, типы приложений,**

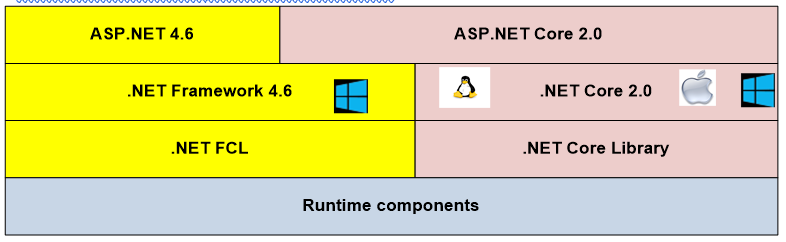
**.NET Core —** это модульная платформа для разработки программного обеспечения с открытым исходным кодом.

**Программная платформа**, разработанная Microsoft и предназначается для разработки web-приложения. Является развитием технологии **OWIN**. **Katana** – OWIN-совместный хост, разработанный Microsoft.

Альтернатива .NET, своя реализация CoreCLR и библиотеки CoreFX

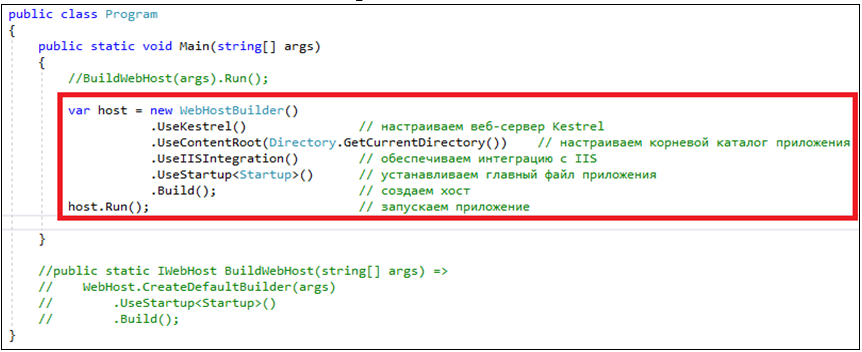
Фреймворк с открытым кодом.

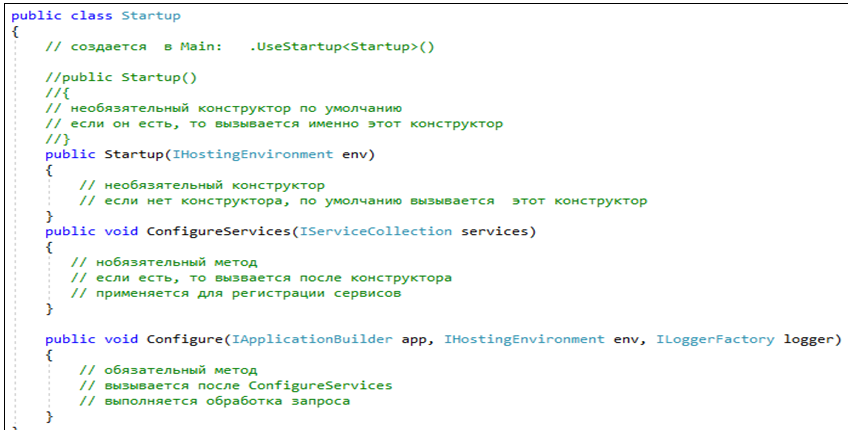
Архитектура – solution приложения



Типы приложений: MVC, WebApi, Web Pages. WebForms – не поддерживается.

Подробный метод Main:





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

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

нус

# **30. ASP.NET CORE: MVC-котроллер, действия (action) контроллера, контекст контроллера, поддержка сессии, результат работы действия, события OnAction, внедрение зависимостей в контроллер, динамическое внедрение зависимости, атрибуты HttpGet, HttpPost, …, AcceptVerb, принцип передачи параметров в метод действия. Фильтры: Action Filer, Result Filter, Authorization Filter, Resource Filter, Exception Filetr, пользовательские фильтры действий.**

# **31. ASP.NET CORE: MVC-представление, обнаружение представления, жизненный цикл представления, методы редеринга представления в web-страницу (методы View контроллера), способы передачи данных из котроллера в представление, строготипизированные представления, директивы @model, @using, @function, @inherits, #inject,@addTagHelper, @removeTagHelper. Применение компоновки (Layout) представления, компоновка по умолчанию (\_ViewStart), применение секций @RenderSection, @RenderBody. Применение представления \_ViewImport. Частичные представления (partial view). Вспомогательные методы представления (хелперы). Встроенные хелперы. Хелперы в строготипизированном представлении. Компоненты представления. Tag helpers представления. Встроенные Tag helpers представления.**

# **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.**

**ВОПРОСЫ**

**В какой зависимости WCF находится от веб-сервиса?**

*Веб-сервис является частным случаем WCF.*

**Какие у WCF есть особенности относительно веб-сервисов?**

Ø *WCF компилится в виде dll-ки и не может исполнятся самостоятельно – для этого ей необходимо приложение-хост. Хостом может быть что угодно, способное исполнять dll-ки – хоть WebForms приложение, хоть десктопное приложение, хоть вообще консольное.*

Ø *Веб-службы работают только по HTTP/HTTPS, в то время как WCF работают по куче протоколов – HTTP, TCP и всяким другим.*

**Сколько у WCF может быть эндпоинтов?**

*По одному на каждый протокол.*

**Как создается и подключается?**

*Сперва создаем саму службу. Создаем WCFLibrary проект, добавляем методы, компилим. Получаем DLL-ку, а еще сообщение от винды, что наша dll-ка хостится. Это с нами общается та самая служба, что я упоминал в первом пункте предыдущего вопроса.*

*Потом создаем хост-приложение. Через Refrence -> Add refrence добавляем сперва System.ServiceModel (лежит в пункте Assemble, помоему, если там нету – самый первый пункт смотрите), а потом через пункт Browse находим ту dll-ку, в которую скомпилился WCF-проект. Копируем из App.Config wcf-приложения все, кроме открывающего тега, вставляем в app.config хост-приложения. В коде Page\_Load, или, когда нам надо, создаем объект-клиент и вызываем его метод Open. Сохраняем, запускаем. Открываем браузер, в адресной строке вставляем адрес эндпоинта (берется из app.config сервиса/хоста). Любуемся красивой страничкой со списком методов. К клиенту служба подключается аналогично лабораторной 11.*

**Сколько точек доступа?**

*У веб-сервиса http, у WCF- http and tcp.*