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

HTTP — протокол прикладного уровня передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных.

Основные свойства

- версии HTTP/1.1 – действующий (текстовый), HTTP/2 – черновой (бинарный);
- два типа абонентов: клиент и сервер;
- два типа сообщений: request и response;
- на один request всегда один response, и на оборот иначе ошибка;
- TCP-порты: 80, 443 (HTTPS);
- для адресации используется URI или URN.

Web-приложение: клиент-серверное приложение, применяющее для обмена данными протокол HTTP; может быть просто web-приложением (HTML+HTTP) или web-службой (API, HTTP-транспорт, формат XML, JSON).

Структура запроса Request: метод -> URI -> версия протокола (HTTP/1.1) -> заголовки (пары: имя/заголовок) -> расширение.

Структура ответа Response: версия протокола (HTTP/1.1) -> код состояния (1xx, - 5xx) -> пояснение к коду состояния -> заголовки (пары: имя/заголовок) -> расширение.

Response: Код состояния: 1xx: информационные сообщения; **2xx**: успешный ответ; **3xx**: переадресация; **4xx**: ошибка клиента; **5xx**: ошибка сервера.

Методы: **get** – только извлекать данные; **post** – отправки сущностей к ресурсу; **patch(или put)** – частичного изменения ресурса; **delete** – удаляет указанный ресурс; **connect** – устанавливает соединение к серверу, определенному ресурсу; **options** – исп для описания параметров соединения с ресурсом; **trace** – выполняет вызов возвращаемого тестового сообщения с ресурса.

Заголовки:

– **General:** общие заголовки, используются в запросах и ответах (transfer-encoding);

– **Request:** используются только в запросах (Accept, Accept-Encoding/Charset/Language);

– **Response:** используются только в ответах (Accept-Rangers);

– **Entity:** для сущности в ответах и запросах (Content-Encoding/Language/Length/Location)

Протокол без сохранения состояния (англ. *Stateless protocol*) — это протокол передачи данных, который относит каждый запрос к независимой транзакции, которая не связана с предыдущим запросом, то есть общение с сервером состоит из независимых пар запрос-ответ.

Протокол без сохранения состояния не нуждается в сохранении информации о сессии на сервере или статусе о каждом клиенте во время множественных запросов. В противовес этому, протокол, которому

необходим учёт о внутреннем состоянии сервера, называется протоколом с сохранением состояния (англ. *stateful*).

Примерами протоколов без сохранения состояния являются Internet Protocol (IP), и Hypertext Transfer Protocol (HTTP).

Пример протокола без сохранения состояния — HTTP означает, что каждое сообщение запроса может быть понято в изоляции от других запросов.

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

HTTPS –расширение протокола HTTP, который обеспечивает конфиденциальность обмена данными между сайтом и пользовательским устройством.

HTTPS (Hypertext Transport Protocol Secure) – расширение протокола HTTP, который обеспечивает конфиденциальность обмена данными между сайтом и пользовательским устройством. HTTP + SSL/TLS
SSL (Secure Sockets Layer) и TLS (Transport Level Security) — криптографические протоколы, обеспечивающие защищенную передачу данных в компьютерной сети.

Соединение, защищенное протоколом TLS, обладает одним или несколькими следующими свойствами:

– **Безопасность:** симметричное шифрование защищает передаваемую информацию от прочтения посторонними лицами.

– **Аутентификация**: "личность" участника соединения можно проверить с помощью асимметричного шифрования.

– **Целостность:** каждое сообщение содержит код (MAC), с помощью которого можно проверить, что данные не были изменены или потеряны в процессе передачи.

«Рукопожатие SSL/TLS» — это название этапа установки HTTPS-соединения. Большая часть работы,

связанной с протоколом SSL/TLS, выполняется на этом этапе.

TLS 1.2

1. Первое сообщение называется **«Client Hello»**. В сообщении перечислены возможности клиента, чтобы сервер выбрал **шифронабор**,который будет использовать для связи. Также сообщение включает в себя «случайное числом клиента».
2. Сервер отвечает сообщением **«Server Hello».** Там он сообщает клиенту, какие параметры соединения были выбраны, и возвращает своё «случайное число сервера». Если клиент и сервер не имеют общих шифронаборов, то соединение завершается неудачно.
3. В сообщении **«Certificate»** сервер отправляет клиенту свою цепочку SSL-сертификатов (листовой и промежуточные сертификаты). Получив их, клиент выполняет провероку для верификации сертификата. Клиент также должен убедиться, что сервер обладает закрытым ключом сертификата, что происходит в процессе обмена/генерации ключей.
4. Сообщение **«Server Hello Done»** уведомляет клиента, о конце передачи данных.
5. Затем клиент участвует в создании сеансового ключа. Особенности этого шага зависят от метода обмена ключами, который был выбран в исходных сообщениях.

6. Сообщение **«Change Cipher Spec»** информирует, что сеансовый ключ сгенерирован и можно переключиться на зашифрованное соединение.
7. Затем отправляется сообщение **«Finished»,** означающее, что на стороне клиента рукопожатие завершено. С этого момента соединение защищено сессионным ключом.
8. Теперь сервер расшифровывает pre-master secret и вычисляет сеансовый ключ. Затем отправляет сообщение **«Change Cipher Spec»**.
9. Сервер также отправляет сообщение **«Finished»**, используя только что сгенерированный симметричный сеансовый ключ, и проверяет контрольную сумму для проверки целостности всего рукопожатия.
После этих шагов SSL-рукопожатие завершено. У обеих сторон теперь есть сеансовый ключ, и они могут взаимодействовать через зашифрованное и аутентифицированное соединение.

TLS 1.3

1. Сообщение **«Client Hello»** запускает рукопожатие, но на этот раз оно содержит гораздо больше информации. TLS 1.3 сократил число поддерживаемых шифров с 37 до 5. Это значит, что клиент может угадать, какое соглашение о ключах или протокол обмена будет использоваться, поэтому в дополнение к сообщению

отправляет свою часть общего ключа из предполагаемого протокола.

2. Сервер ответит сообщением **«Server Hello».** Здесь отправляется сертификат. Если клиент правильно угадал протокол шифрования с присоединёнными данными и сервер на него согласился, последний отправляет свою часть общего ключа, вычисляет сеансовый ключ и завершает передачу сообщением **«Server Finished»**.

3. Теперь, когда у клиента есть вся необходимая информация, он верифицирует SSL-сертификат и использует два общих ключа для вычисления своей копии сеансового ключа. Когда это сделано, он отправляет сообщение **«Client Finished».**

SSL-сертификат – это уникальная цифровая подпись сайта. Такой сертификат нужен, сайтам, работающим с персональными данными, – для защиты транзакций и предотвращения несанкционированного доступа к информации.

SSL-сертификат содержит следующую информацию:

- доменное имя, на которое оформлен SSL-сертификат;
- юридическое лицо, которое владеет сертификатом;
- физическое местонахождение владельца сертификата (город, страна);
- срок действия сертификата;

- реквизиты компании-поставщика SSL-сертификата

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

HTML — стандартизированный язык разметки веб-страниц. Код HTML интерпретируется браузерами; полученная в результате интерпретации страница отображается на экране. Язык HTML до 5-й версии определялся как приложение SGML (стандартного обобщенного языка разметки по стандарту ISO 8879). Спецификации HTML5 формулируются в терминах DOM (объектной модели документа).

DOM «объектная модель документа» — это независящий 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.

Любой документ с помощью DOM может быть представлен в виде дерева узлов. Узлы связаны между собой отношениями «родительский-дочерний».

Изначально различные браузеры имели собственные модели документов (DOM), несовместимые с остальными. Для обеспечения взаимной и обратной совместимости специалисты международного консорциума W3C классифицировали эту модель по уровням, для каждого из которых была создана своя спецификация W3C DOM.

Структура DOM: <!DOCTYPE> предназначен для указания типа текущего документа. **<html>** определяет начало HTML-файла, внутри него хранится заголовок **<head>** и тело документа **<body>**. **<head>**, может содержать текст и теги, но содержимое этого раздела не показывается напрямую на странице, за исключением контейнера **<title>**. Тег **<meta>** является универсальным и добавляет целый класс возможностей, в частности, с помощью метатегов, можно изменять кодировку страницы, добавлять ключевые слова, описание документа и многое другое. **<body>** предназначено для размещения тегов и содержательной части веб-страницы.
CSS (каскадные таблицы стилей) — технология описания внешнего вида документа, оформленного языком разметки. Это стандарт, определяющий представление данных в браузере. **Селектор** — это часть CSS-правила, которая сообщает браузеру, к какому элементу веб-страницы будет применён стиль.
Стиль — это совокупность правил, применяемых к элементу гипертекста и определяющих способ его отображения.
Таблица стилей — это совокупность стилей, применимых к гипертекстовому документу.
Каскадирование — это порядок применения различных стилей к веб-странице. Браузер,

поддерживающий таблицы стилей, будет последовательно применять их в соответствии с приоритетом (от большего) по подключению: style в теге (внутренние), <style type="text/css"> (встроенные), <link> (внешние). Также есть каскадность по селектору (от большего): Inline-стиль, id, class, тег. Использование каскадных таблиц дает возможность разделить содержимое и его представление и гибко управлять отображением гипертекстовых документов путем изменения стилей.

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

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

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

Подключение:

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

```
GET/ demo HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Host: site.com
Origin: http://site.com
```

Если сервер поддерживает ВебСокеты, то он отвечает таким образом:

```
HTTP/1.1 101 Web Socket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
WebSocket-Origin: http://site.com
WebSocket-Location: ws://site.com/demo
```

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

Как только одна сторона хочет передать другой какую-то информацию, она отправляет дата-фрейм следующего вида: 0x00, <строка в кодировке Utf-8>, 0xFF. И все — никаких заголовков, метаданных! Можно отправлять XML или JSON, бинарные данные, картинки.

Скорость и эффективность:

Высокую скорость и эффективность передачи обеспечивает малый размер передаваемых данных. Так же соединение уже готово — не надо тратить время и трафик на его установление.

Время жизни канала:

Веб-сокеты не имеют ограничений на время жизни в неактивном состоянии. Соединение может висеть в неактивном виде и не требовать ресурсов.

Количество:

Открываете столько, сколько вам нужно. А сколько использовать — одно (и через него все

мультиплексировать) или же наоборот — на каждый блок свое соединение — решать вам. Исходите из удобства разработки, нагрузки на сервер и клиент.

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

JavaScript — это язык программирования, который даёт возможность реализовывать сложное **поведение веб-страницы.**

Основные стандарты: EcmaScript (ECMA – 262), ES-7/8/9 (распространённые), TypeScript, Стандарты внутри Java;

ECMAScript — это встраиваемый расширяемый не имеющий средств ввода-вывода язык программирования, используемый в качестве основы для построения других скриптовых языков

Поддерживается основных **7 типов данных**:

1. числовой (Number),
2. строковый (String),
3. логический (Boolean),
4. нулевой (Null),
5. неопределённый (Undefined)
6. function
7. symbol

Также имеется “составной” тип данных объектный (Object).

Программные структуры:

1. Функции взаимодействия с пользователем: alert, prompt, confirm

2. Условные операторы: if, '?'
3. Циклы while,do while, for
4. Конструкция switch
5. Функции

Применение:

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 используется в клиентской части веб-приложений: клиент-серверных программ, в котором клиентом является браузер, а сервером — веб-сервер, имеющих распределённую между сервером и клиентом логику. Обмен информацией в веб-приложениях происходит по сети. Одним из преимуществ такого подхода является тот факт, что клиенты не зависят от конкретной операционной системы пользователя, поэтому веб-приложения являются кроссплатформенными сервисами.

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

AJAX – асинхронный JavaScript and XML – методология (подход) построения динамических приложений, при которых не осуществляется полная перезагрузка html-страниц.

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

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

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

DOM – объектная модель, позволяющая сценариям JavaScript получить доступ к элементам html-документа. В Ajax ответ сервера —встраивается с помощью JavaScript-сценария в загруженную ранее браузером страницу. При этом доступ к элементам html-документа осуществляется в соответствии с моделью DOM.

HTTP

XML – расширяемый язык разметки данных.

JSON (JavaScript Object Notation) - текстовый формат обмена данными, применяемый обычно в сценариях JavaScript**.** Формат JSON основывается на функции **eval()** языка JavaScript.

XMLHttpRequest – специальный API (предопределенный объект), используемый в языке JavaScript для обмена данными. Данные могут получены в виде XML-документа и виде обыкновенного текста (в частном случае могут быть представлены в формате JSON).

значение	состояние	описание
0	Unsent	XMLHttpRequest создан, но open() не вызван
1	Opened	Open() вызван, можно поставить заголовки с помощью setRequestHeader(), можно вызывать метод send
2	Headers_Recived	Вызван send() заголовки и статус доступны
3	Loading	Загрузка тела ответа. Если responseType = “text” или пустой строке, значит responseText будет содержать частичные данные
4	Done	Операция завершена (успешна/не удалась)

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

– использование технологии динамического обращения к серверу «на лету», без перезагрузки всей страницы полностью, например с использованием XMLHttpRequest (основной объект);

- через динамическое создание дочерних фреймов;
- через динамическое создание тега <script>.
- через динамическое создание тега <img>

– использование DHTML для динамического изменения содержания страницы;

ПРИМЕНЕНИЕ:

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

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

Ускорение реакции интерфейса: поскольку загрузка изменившейся части значительно быстрее, то пользователь видит результат своих действий быстрее и

без мерцания страницы (возникающего при полной перезагрузке).

Возможности для интерактивной обработки

Мультимедиа не останавливается: Страница не перезагружается, плеер продолжает работать.

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

Веб-приложение — клиент-серверное приложение, в котором клиент взаимодействует с веб-сервером при помощи http протокола. Одним из преимуществ такого подхода является тот факт, что клиенты не зависят от конкретной операционной системы пользователя, поэтому веб-приложения являются межплатформенными службами.

Архитектура web-приложения(везде протокол http):

1. Клиент-веб-сервер
2. Один клиент и два веб-сервера(обращение либо к одному, либо к другому)
3. Два клиента и один веб-сервер
4. Клиент обращается к веб-серверу, а веб-сервер к другому веб-серверу

Особенности реализации web-приложения:

1. При проведении работ, которые требуют остановку веб сервера, необходимо иметь запасной, на который можно перенаправить трафик (**отказоустойчивость**)
2. Т.к. к веб-серверу происходит множество запросов от различных пользователей, то возможности ответа веб-сервера упираются в пропускную способность канала и, чтобы не загружать этот канал слишком сильно, используются максимально

короткие сеансы подключения и кэширование ответов на стороне клиента, а кэширование на стороне сервера позволяет быстрее сформировать ответ. (**Быстродействие**)
3. Использование https для устранения возможности злоумышленникам подделывать запросы/ответы и просматривать их. (**Шифрование**)

Web-сервер:

1. Программа, которая обрабатывает Http запросы от веб-клиента и отправляет ответ
2. Примером веб-сервера может служить IIS, Apache Tomcat

Web-клиент: можно рассматривать веб-браузер. Это программа, которая переносит большую часть функционала по обработке информации на сервер.

1. Для отображения контента используется HTML.
2. Веб-клиент инициирует соединение(отправляет запрос) с помощью протокола http
3. Использует DOM модель для представления html документа, а javascript для доступа к содержимому HTML документа
4. Использует CSS (Cascading Style Sheets) для добавления стилей к html документу
5. Web-engine - веб движок, который преобразует html страницу во внутреннее представление веб-браузера в соответствии с моделью DOM. В хроме это Blink, написан на C++

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

IIS (Internet Information Server) - набор серверов для Internet-служб компании Microsoft. Основным компонентом IIS является веб-сервер (служба www), который позволяет размещать в Интернете сайты. Поддерживает протоколы: HTTP, HTTPS, FTP

Публикация приложения – создание скомпилированного приложения. которое можно разместить на сервере.

Развертывание приложения – перемещение опубликованного приложения в систему размещения (на сервер).

Основные составные части публикуемого приложения:

– **статические ресурсы** (файлы стилей css, скрипты JS, файлы изображения);

– **сборки приложения** (в них все компилируются). После компиляции в папке bin окажется файл NameApp.dll (ключевой файл приложения) и нужные библиотеки.

– файлы, содержащие **настройки приложения** - Global.asax и Web.config.

– **представления**, содержащиеся в каталоге Views, включаются в пакет приложения как есть, в отличие от основной сборки приложения.

Процесс публикации заключается в переносе папок и файлов проекта web-приложения в папку сервера IIS. **Публикация** может быть выполнена несколькими способами. В простейшем случае - простым копированием папок и файлов. Для этого:

1. В IIS Manager создать новый сайт (web-узел). При этом необходимо указать имя сайта, выбрать пул приложений, физический путь для размещения папок и файлов web-приложения, а также установить привязки (тип протокола, номер TCP-порта и IP-адрес).

2. Скопировать весь билд проекта web-приложения в папку созданного на IIS сайта.

3. Проверить работоспособность сайта - на компьютере с сервером IIS запустить браузер и в адресной строке указать http://localhost:port/path/.

4. Удалить лишние файлы (исходные коды) из папки сайта.

Web-узел IIS — это любое web-приложение развернутое на сервере IIS из параметров самое основное: имя, порт, ip, тип (http), пул и физический путь

Реальный каталог - физическое расположение файлов приложения.

Виртуальный каталог — это псевдоним либо для физического каталога на жестком диске сервера, расположенного вне области реального каталога.

Локальный виртуальный каталог физически расположен на том же ПК, что и web-сервер, обеспечивающий работу web-узла. **Сетевой виртуальный каталог** физически расположен на любом сетевом компьютере, но не на компьютере с web-сервером.

Создание виртуального каталога. Происходит в настройке сайтов: добавить виртуальный каталог -> ввести псевдоним и физический путь

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

HTTP-обработчики используются для генерации содержимого ответа на HTTP-запрос.

Класс, реализующий интерфейс IHttpHandler:

- ProcessRequest - обрабатывает запрос и формирует ответ клиенту; в качестве параметра принимает контекст запроса HttpContext
- IsReusable - можно ли использовать экземпляр для повторного запроса

События, возникающие при запросе:

- *MapRequestHandler* происходит при выборе обработчика для ответа на запрос
- выбор обработчика запроса
- *PostMapRequestHandler* срабатывает после выбора обработчика
- *PreRequestHandlerExecute* происходит перед выполненем обработчика событий
- вызов метода *ProcessRequest* и генерация ответа
- *PostRequestHandlerExecute* возникает после генерации ответа

Порядок разработки:

- Создать класс обработчика, который наследует интерфейс System.Web.IHttpHandler

- Определить в классе обработчика метод ProcessRequest и свойство IsReusable
- Подключить обработчик к обработке запросов (через обработчик маршрутов RouterConfig или через web.config).
  - Класс RouterConfig: routes.Add(new Route(“handler”/{*path}, new CustomRouteHandler()));
  - web.config: `<handlers> <add name="Handler1" path="/handlers/handler1" verb="GET,POST,PUT" type="Lab1.Handlers.Lab1a.Task123" /></handler>`

Атрибуты обработчика в web.config:

- path: запрос URL, который будет обрабатываться обработчиком
- verb: тип запроса, GET. Мы также можем поставить любой тип запроса (*)
- type: полный тип класса обработчика

Http-обработчик для взаимодействия с клиентом по протоколу WebSocket.

С помощью свойства IsWebSocketRequest этого объекта можно определить является ли пришедший запрос, запросом на соединение по протоколу WebSocket.

Если это так, устанавливается соединение и управление передается методу *WebSocketRequest*.

- var socket = context.WebSocket
- Для отправки/получения: socket.ReceiveAsync/socket.SendAsync

В обработчике:

1. Рукопожатие (Receive -> Send)
2. while (socket.State == WebSocketState.Open)
3. Взаимодействие внутри цикла (например, отправка уведомлений…)

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

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

ASMX - веб-сервис на базе Dot.Net, предназначенный для передачи и получения сообщений, используя SOAP только через HTTP. ASMX простой, но во многих отношениях ограничен по сравнению с WCF. ASMX только на IIS (WCF где угодно например, консольное приложение, WinForms), ASMX только Http (WCF еще TCP, MSMQ, NamedPipes), у WCF расширенные настройки безопасности, у ASMX ограниченные, ASMX веб сервисы используют для сериализации класс XmlSerializer, в то время как WCF использует DataContractSerializer.

WSDL (англ. Web Services Description Language) — язык описания веб-сервисов и доступа к ним, основанный на языке XML. Разделен на части:

-определение типов данных (types) — определение вида отправляемых и получаемых сервисом XML-сообщений;

-элементы данных (message) — сообщения, используемые web-сервисом

-абстрактные операции (portType) — список операций, которые могут быть выполнены с сообщениями;

-связывание сервисов (binding) — способ, которым сообщение будет доставлено

SOAP протокол обмена структурированными сообщениями на базе XML. Не зависит от протокола прикладного уровня (Http, Smtp,...)

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/s
     oap-envelope/"
soap:encodingStyle="http://www.w3.org/2
     003/05/soap-encoding">
  <soap:Header> ... </soap:Header>
  <soap:Body> ...
    <soap:Fault> ... </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Порядок разработки:

Сервер

1. Веб Сервис (приложение типа веб-служба ASP.Net)
2. `[WebMehtod] public string AddTs(Ts ts) {}`
3. `[WebService(Namespace = “http://tempuri.org/”)] [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]`

```
[System.ComponentModel.ToolBoxItem
(false)]
public class
ASMX_Service:WebService()
```

Клиент

1. создать приложение любого типа, например WinForms
2. добавить в приложение ссылку на службу с помощью ПКМ->Добавить ссылку на службу
3. создать объект клиента и через клиента вызывать методы
4. если использовался параметр EnableSession = true в вебметоде, то на клиенте должен быть установлен параметр allowCookies=”true” в теге <binding>

Прокси

Для взаимодействия клиента с сервером может создаваться прокси-класс, который является обёрткой над HttpClient, и позволяет общаться с ASMX-сервером как с локальным сервисом. ПКМ->Добавить ссылку на службу или WSDL.exe

11. ASP.NET: MVC-приложение, структура MVC(R)-приложения, назначение основных компонентов приложения, маршрутизация. Пример.

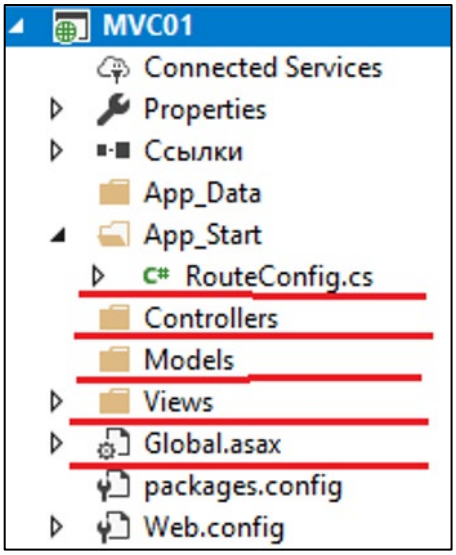
MVC(R): (Model-View-Controller(-Router)) – паттерн проектирования, в котором приложение состоит из трех(четырех) взаимодействующих компонентов: модель, представление, контроллер, маршрутизатор. В правильно разработанном MVC-приложений компоненты относительно независимы.

Каждый компонент имеет свою зону ответственности:

1. **модель** – данные и бизнес-логика;
2. **представление** – динамическое формирование разметки для отправки клиенту;
3. **контроллер** – обработка запроса, формирование экземпляра модели, вызов Razor Engine.
4. **маршрутизатор -** компонент по разбору запроса

Схема грубо говоря такая: Браузер делает запрос (http), маршрутизатор направляет его контроллеру (у него есть таблица и паттерыны что куда направлять исходя из строки запроса), котроллер сначала работает с моделью (а модель с источником данных) и после генерирует представление (разметку файл .cshtml) и возвращает браузеру в качестве ответа

Структура MVC(R) приложения:


MVC01
Connected Services
Properties
Ссылки
App_Data
App_Start
RouteConfig.cs
Controllers
Models
Views
Global.asax
packages.config
Web.config

12. ASP.NET: MVC(R)-приложение, маршрутизатор, принципы устройства и работы. Пример.

Маршрутизация — это процесс перенаправления HTTP-запроса на контроллер, а функциональность этой обработки реализована в System.Web.Routing.

В ASP.NET MVC 5 все определения маршрутов находятся в файле RouteConfig.cs, который располагается в проекте в папке App_Start.

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }
}
```

Первая строка **routes.IgnoreRoute("{resource}.axd/{*pathInfo}");** отключает обработку запросов для некоторых файлов, например с расширением *.axd (WebResource.axd).

Далее идет собственно определение маршрута по умолчанию.

Метод **routes.MapRoute** выполняет сопоставление маршрута запросу. В перегруженных версиях данного метода мы можем задать дополнительные параметры сопоставления. Разберем параметры метода.

– **name** - задает имя маршрута - Default.

– **url -** задает шаблон строки запроса или шаблон Url, с которым будет сопоставляться данный маршрут (может стоять все что угодно, хоть regex). При получении запроса механизм маршрутизации парсит строку URL и помещает значения маршрута в словарь в объект **RouteValueDictionary**, доступный через контекст приложения **RequestContext**. В качестве ключей в нем применяются имена параметров URL, а соответствующие сегменты URL выступают в качестве значений. Например, у нас есть следующий URL запроса: http://localhost/Home/Index/5, то в этом случае образуются следующие пары ключей и значений в словаре RouteValueDictionary: controller – home, action – index, id - 5

– **defaults -** определяет значения по умолчанию для маршрута. (если вдруг в строке запрос указаны не все параметры, а сам запрос, к примеру, идет по адресу http://localhost/, то система маршрутизации вызовет метод Index контроллера Home). Последний параметр объявлен как необязательный id = UrlParameter.Optional, поэтому, если он не указан в строке запроса, он не будет учитываться и передаваться в словарь параметров RouteValueDictionary.

Установкой маршрутов занимается статический метод RegisterRoutes. Вызов которого осуществляется в файле

Global.asax в методе Application_Start. RouteConfig.RegisterRoute(RouteTable.Routes)

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

Маршрутизация — это процесс перенаправления HTTP-запроса на контроллер, а функциональность этой обработки реализована в System.Web.Routing.

Фреймворк MVC позволяет использовать в приложении маршрутизацию на основе атрибутов. Такой тип маршрутизации еще называется **Attribute-Based Routing**. Атрибуты предоставляют более гибкий способ определения маршрутов. Маршруты, определенные с помощью атрибутов, имеют приоритет по сравнению с маршрутами, определенными в классе Startup.

Маршрутизация с помощью атрибутов по умолчанию отключена. Включается она посредством вызова метода **MapMvcAttributeRoutes()**, который вызывается на объекте **RouteCollection**, передаваемом в качестве аргумента статическому методу **RegisterRoutes()** в классе **RouterConfig**

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.MapMvcAttributeRoutes();
```

Вызов метода **MapMvcAttributeRoutes()** заставляет систему маршрутизации проинспектировать классы контроллеров в приложении в поисках атрибутов, конфигурирующих маршруты.

Атрибуты маршрутизации:

- атрибут маршрута: **Route** - может быть применен несколько раз к 1 методу. Значок “~” указывает на то, что префикс, указанный для контроллера, будет проигнорирован.
- атрибут префикса: **RoutePrefix**
- атрибуты запроса: **HttpGet**, **HttpPost** и т.д. или же **AcceptVerbs(“Post”, “Get”)**

Для определения маршрута, необходимо использовать атрибут **[Route]** в методе контроллера (акция/событие) или самом контроллере:
[Route("Home/{id:int}/{name}")] (запрос http://localhost:xxxx/Home/4/string),
[Route(“Home/Index”)](запрос http://localhost:xxxx/Home/Index)
В качестве параметра атрибут **Route** принимает шаблон URL, с которым будет сопоставляться запрошенный адрес. В неё могут входить такие параметры, как хардкодные строки, ограничения, а также названия контроллера и экшена (Route[“[controller]/[action]”]).
Route может быть применен к одной акции несколько раз. Это приведет к возможности вызова данной акции для запросов на несколько разных URL.
Route может быть применён как к акциям контроллера, так и к контроллеру. Применение атрибута к контроллеру позволяет автоматически добавить к

маршруту каждой акции определенный **обязательный** префикс.

При необходимости в параметре атрибута **Route** можно указывать определенные ограничения на входную строку (URL запроса). Пример:

[Route("Home/{id:int}/{name}")]

В этом случае на параметр **id** наложено ограничение **int**. Это значит, что он должен являться целым числом. На параметр **name** не наложено никаких ограничений, значит будет принято любое значение.

Список возможных ограничений:

- **alpha**: соответствует только алфавитным символам латинского алфавита. Например, {id:alpha}
- **decimal**,**double**,**float,int, dateTime, bool ….**
- **length, maxlength, minlength.** Например, {id:length(5)} или {id:length(5, 15)}
- **max, min.** Например, {id:max(99)}.
- **range**: указывает на диапазон, в пределах которого должно находиться значение сегмента. Например, {id:range(5, 20)}
- **regex**: соответствует регулярному выражению. Например, {id:regex(^\d{3}-\d{3}-\d{4}$)}

С помощью атрибута **RoutePrefix** можно определить общий префикс, который будет применяться ко всем маршрутам, заданным в контроллере. Однако при использовании именно этого атрибута указанный

префикс можно игнорировать (Значок “~” в атрибуте route в событии)

Атрибуты методов, такие как, например, **[HttpGet]**, **[HttpPost]** и т.д. необходимы для того, чтобы указать, что акция будет обрабатывать запрос не только с подходящим маршрутом, но и подходящим методом. При необходимости указать несколько допустимых методов запроса нужно использовать атрибут **AcceptVerbs.** Пример: [AcceptVerbs(“Get, Post”)]

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

MVC(R): (Model-View-Controller(-Router)) – паттерн проектирования, в котором приложение состоит из трех(четырех) взаимодействующих компонентов: модель, представление, контроллер, маршрутизатор. В правильно разработанном MVC-приложений компоненты относительно независимы.

Каждый компонент имеет свою зону ответственности:

1. **модель** – данные и бизнес-логика;
2. **представление** – динамическое формирование разметки для отправки клиенту;
3. **контроллер** – класс, обеспечивающий связь между пользователем и системой, представлением и хранилищем данных. Он получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления.
4. **маршрутизатор** - компонент по разбору запроса

Жизненный цикл Контроллера

Если вы используете фабрику контроллеров по умолчанию, для каждого запроса будет создаваться новый экземпляр, и так и должно быть. Контроллеры не

должны использоваться разными запросами. Однако вы могли бы написать собственную фабрику, которая управляет временем жизни контроллеров.

Взаимодействие с моделью и представлением

Контроллер обрабатывает введенные данные пользователем и при необходимости записывает полученные данные в модель, следовательно, при необходимости контроллер может подтягивать необходимые данные для вычислений из модели. После выполненных методов, их результаты передаются в представление, которые представлены страницами Razor Engine.

Пример контроллера

```
public class HomeController :
Controller
{
    BookContext db = new BookContext();
    public ActionResult Index()
    {
        IEnumerable<Book> books =
db.Books;
        ViewBag.Books = books;
        return View();
    }
    [HttpGet]
```

```
public ActionResult Buy(int id)
{
    ViewBag.BookId = id;
    return View();
}}
```

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

View - файл с расширением .cshtml - формирует вывод на страницу клиента, при этом используется разметка Razor.

Жизненный цикл

При **первом обращении** срабатывает Razor Engine, он генерирует объект **C# из файла .cshtml**. При **последующих обращениях** к странице будет использоваться этот объект. На основе объекта генерируется html страница без C# кода.

Способы передачи данных из контроллера:

1. HttpContext

2. ViewData - словарь ключ/объект

3. ViewBag - похож на ViewData, но вместо словаря используется тип dynamic. коллекция, хранит данные в рамках одного запроса. Эта коллекция создается в рамках одного запроса для взаимодействия контроллера и представления.

4. TempData - аналогично ViewData, но данные сохраняются на несколько запросов в рамках одного сеанса

5. С использованием модели - тут мы используем нотацию @model вверху страницы, это говорит нам о том, что подключаем модельку

```
public IActionResult Index()
```

```
{ return View(phones); // или
View("viewname", phones) }
```

Тип модели представления задается директивой @model в соответствующей .cshtml страниц

```
@model IEnumerable<Phone>
```

Синтаксис Razor

Стандартное представление очень похоже на обычную веб-страницу с html. Однако оно также имеет вставки кода на C#, которые предваряются знаком @.

Движок представлений

При вызове метода View контроллер не производит рендеринг представления и не генерирует разметку html. Контроллер только готовит данные и выбирает, какое представление надо возвратить в качестве объекта ViewResult. Затем уже объект ViewResult обращается к движку представления для рендеринга представления в выходной результат.

Ранее поддерживался также движок WebForms.

Razor - это не какой-то новый язык, это лишь способ рендеринга представлений, который имеет определенный синтаксис для перехода от разметки html к коду C#.

```
@model
IEnumerable<BookStore.Models.Book>
<div> <table>
```

```
@foreach (BookStore.Models.Book b in
Model)
{<tr> <td><p>@b.Name</p></td> </tr> }
</table></div>
```

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

MVC: (Model-View-Controller) – паттерн проектирования, в котором приложение состоит из трех взаимодействующих компонентов. В правильно разработанном MVC-приложений компоненты относительно независимы.

Каждый компонент имеет свою зону ответственности:

1. **модель** – данные и бизнес-логика;
2. **представление** – динамическое формирование разметки для отправки клиенту;
3. **контроллер** – класс, обеспечивающий связь между пользователем и системой, представлением и хранилищем данных.
4. **маршрутизатор** - компонент по разбору запроса

Модель - одна из главных составляющих компонент в шаблоне проектирования MVC. Она работает с данными и бизнес-логикой. Объект модели должен создаваться внутри контроллера. Жизненный цикл модели определяется также, как и контроллера - создается объект на каждый Request.

Если же говорить о модели с точки зрения данных, то существует **3 типа моделей:**

- Модель для представления (строго-типизированные представления);
- Модель для параметров в Action в контроллере;
- Модель для работы с БД;

Репозиторий - паттерн для взаимодействия приложения с базой данных. Его основная идея - абстрагироваться от жестких привязок к подключению к бд и сделать его более гибким и масштабируемым. Настройка:

- Задается строка подключения в файле web.config
- Создается класс контекста базы данных (в нем происходит конфигурация подключения с БД, настройка сущностей и тд);
- Создаем интерфейс с реализацией методов для работы с БД;
- Создаем класс, реализующий этот интерфейс;
- Вызываем в контроллере объект репозитория и работаем с БД;

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

Dependency Injection – внедрение зависимости, программный механизм, позволяющий в автоматическом режиме создавать программный объект, с заданными жизненным циклом (задаются события инстансирования и разрушения объекта), способом применения (в качестве параметра метода или конструктора, свойства или поля объекта) и областью действия.

Свойства:

– позволяет создавать слабосвязанные компоненты.

– повторное применение кода, упрощает внесение изменений, упрощает тестирование.

– чаще всего внедряется contextDB или репозиторий модели данных.

Inversion of Control (инверсия управления) — принцип программирования, позволяющий снизить зависимость между компонентами программ; DI – один из способов реализации IoC.

IoC-контейнеры: Ninject, AutoFac, Unity (не тот юнити, другой),….

Программная реализация DI:

– Для подключения Ninject в приложение ASP.NET нужно установить все пакеты Ninject.

– Далее необходимо создать интерфейс и класс, который будет его реализовать

– Далее нужно создать класс NIConfig. Этот класс реализует интерфейс NinjectModule и реализует метод `Load` в котором идет регистрация зависимости класса от интерфейса (binding интерфейса в конкретный класс) `Bing<InterfaceName>.To<ClassName>.Scope Name`

– В нужном контроллере создаем объект типа интерфейса `Ikernel kernel = new StandartKernel(new DI.NIConfig())` и потом создаем объект интерфейса `DI.InterfaceName = kernel.Get<DI.InterfaceName>`

– Глобальная регистрации зависимостей, в Global.asax необходимо инициализировать сопоставление зависимостей происходило при запуске приложения. В этом файле нужно прописать: `var kernel = new StandartKernel(new DI.NIConfig()); DependencyResolver.SetResolver(new NinjectDependencyResolver(kernel)).` Для подключения конфиграции Ninject

– **DI/Ninject: scope (область действия)**

Область	Метод связывания	Объяснение
Временный	`.InTransientScope()`	Объект класса будет создаваться по каждому требованию (метод по умолчанию).
Одиночка	`.InSingletonScope()`	Объект класса будет создан один раз и будет использоваться повторно.
Поток	`.InThreadScope()`	Один объект на поток.
Запрос	`.InRequestScope()`	Один объект будет на каждый web-запрос

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

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

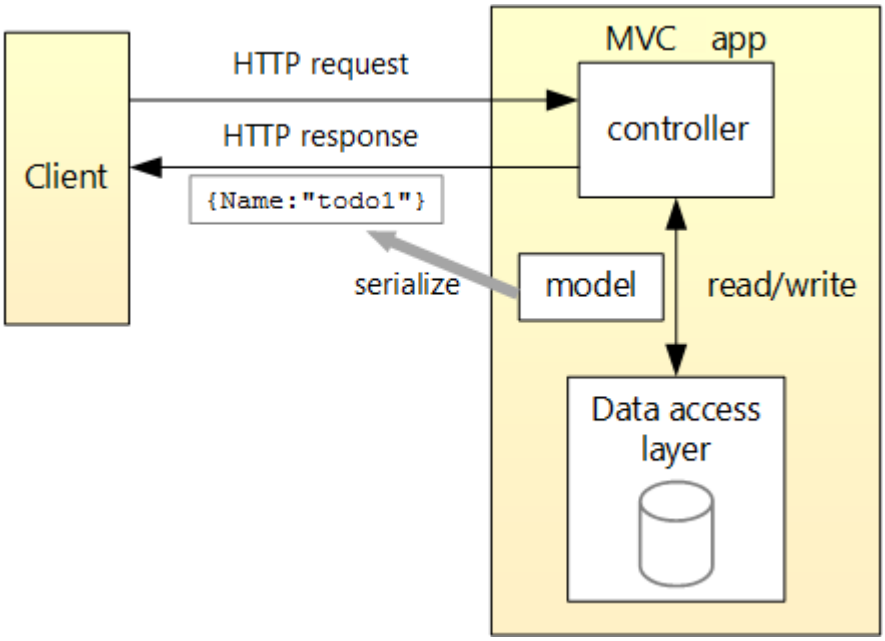
– Методы действий возвращают объекты моделей, а не объекты типа ActionResult.

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

Объекты моделей, возвращаемые методом действия контроллера API, кодируются в формате JSON и отправляются клиенту. Контроллеры API поэтому они не поддерживают представления, компоновки или любые другие средства, применяемые для генерации HTML-разметки.

Инфраструктура **ASP.NET MVC Framework** выполняет шаги, требуемые для доставки HTML-содержимого пользователю (включая аутентификацию, авторизацию, выбор и визуализацию представления). После того, как HTML-содержимое доставлено в браузер, запросы Ajax, генерируемые содержащимся внутри кодом JavaScript, будут обрабатываться контроллером Web API.

Структура WebAPI приложения (Ссылка на достоверный источник)



1. Слой доступа к данным управляет чтением и записью информации на носитель (БД, бинарный файл, XML и т. д.) **Назначение** - чтение и запись данных.
2. API контроллер отвечает на запросы клиента. Он сериализует модель в JSON формат и отправляет ее клиенту. **Назначение** - обработка запроса пользователя и отправка ему ответа.
3. Клиент в свою очередь отправляет HTTP запрос, который будет обработан контроллером.

Маршрутизация

Конфигурация маршрутов WebApi контроллеров производится в App_Start/WebApiConfig.cs

```
1 usage   Ilya Yushkevich
public static void Register(HttpConfiguration config)
{
    config.MapHttpAttributeRoutes();

    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = UrlParameter.Optional }
    );
}
```

WebApiConfig необходимо зарегистрировать в Global.asax.cs

```
public class MvcApplication : System.Web.HttpApplication
{
    Ilya Yushkevich
    protected void Application_Start()
    {
        var registrations = new RegisterNinject();
        var kernel = new StandardKernel(registrations);
        DependencyResolver.SetResolver(new NinjectDependencyResolver(kernel));

        var mvcResolver = new NinjectDependencyResolver(kernel);
        DependencyResolver.SetResolver(mvcResolver);

        var webApiResolver = new NinjectResolver(kernel);
        GlobalConfiguration.Configure( configurationCallback: WebApiConfig.RegisterWithResolver(webApiResolver));

        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

Пример WebAPI контроллера

```
[EnableCors(origins: "*", headers: "*", methods: "*")]
Ilya Yushkevich *
public class DictApiController: ApiController
{

    Ilya Yushkevich
    public async Task<IEnumerable<Node>> Get()
    {
        var nodes = await _nodeRepository.GetAll();
        var sortedNodes = nodes.OrderBy(n => n.Name).ToSafeReadOnlyCollection();

        return sortedNodes;
    }

    Ilya Yushkevich
    public async Task Post(Node node)
    {
        node.Id = Guid.NewGuid();
        await _nodeRepository.Save(node);
    }
```

Также возможна маршрутизация с помощью атрибутов

```
public class ProductsController : ApiController
{
    [HttpGet]
    public Product FindProduct(id) {}
}
```

```
public class ProductsController : ApiController
{
    [AcceptVerbs("GET", "HEAD")]
    public Product FindProduct(id) { }

    // WebDAV method
    [AcceptVerbs("MKCOL")]
    public void MakeCollection() { }
}
```

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

WCF (Windows Communication Foundation) - технология для построения SOA (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). Использование сервиса не зависит от платформы, на которой реализован сервис; использование сервиса не зависит от технологии его разработки. Логика сервиса и его реализация полностью отделена от его коммуникационной составляющей; способ взаимодействия с сервисом определяется конфигурационным файлом.

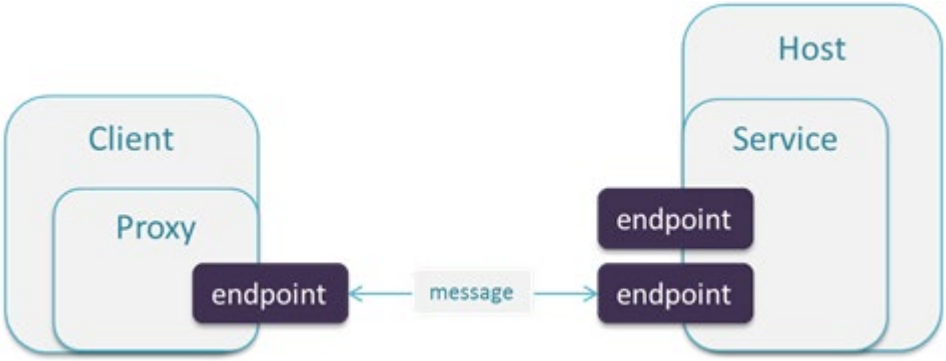
инфраструктура состоит из двух главных уровней:

1. сервисный уровень (Service Layer)- собственно сервис или клиент;
2. канальный уровень (Channel Layer), который определяет способ передачи (TCP, HTTP, Named Pipes и пр.).

Каждый из уровней может содержать несколько подуровней; в каждый подуровень может **вклиниться** код программиста.

WCF-служба, представляет собой класс; этот класс не может существовать самостоятельно, а должен находиться под управлением некоторого процесса Windows, называемого ***хостовым*** процессом. В качестве хоста может выступать консольное или графическое NET-приложение (автохостинг), Windows-служба (Windows Service), IIS, WAS.

WCF Архитектура



Service: dll-библиотека.

Host: программный модуль, содержащий в себе Service.

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

Message: сообщения для обмена данными между конечными точками

Proxy: КЛАСС, ЭМУЛИРУЮЩИЙ РАБОТУ СЕРВИСА НА МАШИНЕ. промежуточная dll-библиотека, эмулирующая работу с Service, как с локальным объектом.

WCF Endpoint состоит из**: Address:** сетевой адрес сервиса (Where). **Binding:** способ взаимодействия

клиента с сервисом (How). **Contract:** описание интерфейса сервиса (что может сервис) (What).

Контракт WCF – интерфейс описывающий необходимый функционал, который требуется реализовать в рамках endpoint

Типы контрактов WCF:

- **Data Contract** - описывает формат ваших данных и определяет, как данные должны быть сериализованы/десериализованы.
- **Сервисный контракт** - он описывает операции, выставленные службой. Он также может описывать шаблон обмена сообщениями.
- **Message Contract** - он дает вам контроль над SOAP-сообщением. Если вы используете только контракт с данными, все данные будут находиться в теле сообщения SOAP, но, если вам нужен элемент управления/доступ к заголовку сообщения SOAP, вы можете использовать MessageContract.
- **Fault Contract** — это специальный контракт, позволяющий клиенту знать, что что-то не так со стороны обслуживания. WCF обрабатывает исключение и передает сообщение об ошибке клиенту, используя SOAP Fault Contract.

WSDL-файл описывает веб-сервис. Он описывает местоположение сервиса и его методы, используя следующие элементы:

Элемент	Описание
<types>	Определяет типы данных (XML Schema), используемые веб-сервисом
<message>	Определяет элементы данных для каждой операции
<portType>	Описывает операции, которые могут быть выполнены и соответствующие сообщения
<binding>	Указывает протокол и тип данных для каждого типа портов

Отличия WSDL 2.0 от языка версии 1.1 (может спросить)

В язык WSDL добавлена дополнительная семантика, что явилось одной из причин, почему атрибут targetNamespace элемента definitions стал обязательным.

Удалены конструкции сообщений. Теперь они задаются в элементе types при помощи системы типов XML-схемы.

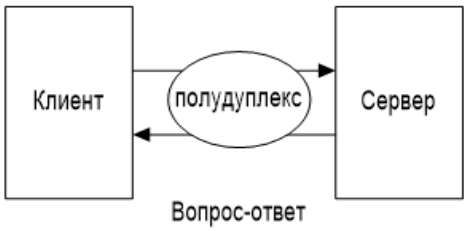
Отсутствует поддержка перегрузки операторов.

Элемент portType переименован как interface. Поддержка наследования элемента interface достигается благодаря использованию атрибута extends в элементе interface.

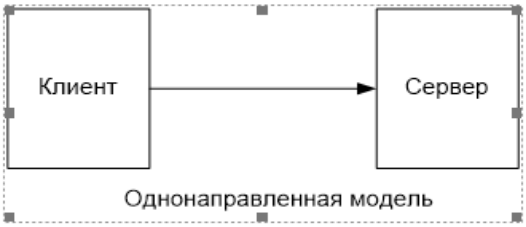
Элемент port переименован в endpoint.

Спецификация WSDL – это три документа: ***1) базовый язык; 2) шаблоны сообщений; 3) связывание.***

WCF: модели взаимодействия

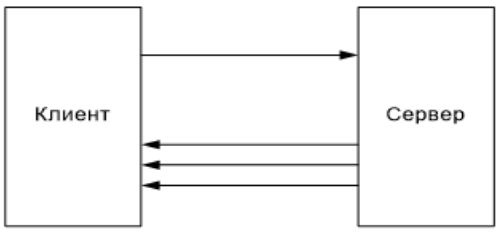


Вопрос-ответ



Однонаправленная модель

Дуплекс



Потоковая модель

Отличия ASMX от WCF

ASMX простой, но во многих отношениях ограничен по сравнению с WCF. ASMX только на IIS (WCF где угодно например, консольное приложение, WinForms), ASMX только Http (WCF еще TCP, MSMQ, NamedPipes), у WCF расширенные настройки безопасности, у ASMX ограниченные, ASMX веб сервисы используют для сериализации класс XmlSerializer, в то время как WCF использует DataContractSerializer.

Порядок разработки:

– создаём веб-сервис (приложение типа библиотека классов)

– создаём интерфейс, атрибут [ServiceContract]показывает, что интерфейс определяет

контракт службы в приложении WCF, [OperationContract] указывает, что метод определяет операцию, которая является частью контракта службы WCF

```
[ServiceContract]
1 reference
public interface ICatalog
{

    [OperationContract]
    1 reference
    List<User> GetContacts();
```

– создаем класс, который наследуется от интерфейса и прописываем методы сервиса

в файле app.config указываем имя сервиса, его конечные точки, адрес

– создаём хост (в данном случае консольное приложение)

```
0 references
static void Main(string[] args)
{
    ServiceHost host = new ServiceHost(typeof(WcfService.Catalog));
    host.Open();
    Console.Write("Success");
    Console.ReadLine();
}
```

– в файле App.config указываем имя сервиса, конечные точки и адрес хоста

```
<services>
  <service name="WcfService.Catalog" behaviorConfiguration="debug">
      <endpoint address="WcfService" binding="basicHttpBinding" contract="WcfService.ICatalog"/>
      <endpoint address="WcfService" binding="netTcpBinding" contract="WcfService.ICatalog"/>
      <endpoint contract="IMetadataExchange" binding="mexHttpBinding" address="mex"/>
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost:8089/"/>
          <add baseAddress="net.tcp://localhost:8090"/>
        </baseAddresses>
      </host>
  </service>
</services>
```

создаём клиент и используем прокси класс PhoneBookClient

```
public partial class Form1 : Form
{
    PhoneBookService.PhoneBookClient client = new PhoneBookService.PhoneBookClient("BasicHttpBinding_IPhoneBook");
    PhoneBookService.TS elementTS = new PhoneBookService.TS();
    DataTable table = new DataTable();
```

подключаем сервис к клиенту (появятся Connected Services и подключённый сервис в них)

В файле app.config указываем биндинг и конечные точки для клиента

```
<system.serviceModel>
    <bindings>
        <basicHttpBinding>
            <binding name="BasicHttpBinding_IPhoneBook" />
        </basicHttpBinding>
    </bindings>
    <client>
        <endpoint address="http://localhost:8080/PhoneBook" binding="basicHttpBinding"
            bindingConfiguration="BasicHttpBinding_IPhoneBook" contract="PhoneBookService.IPhoneBook"
            name="BasicHttpBinding_IPhoneBook" />
    </client>
</system.serviceModel>
```

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

ASP.NET Core: программная платформа, разработанная Microsoft и предназначается для разработки web-приложений. Является развитием технологии **OWIN** (The Open Web Interface for.NET) - обеспечивает интерфейсы между приложением и http-сервером, между http-сервером и Host, между http-сервером и middleware.

Структура: Приложение – Промежуточное ПО (Middleware) – Сервер – Хост (приложение процесс операционной системы управляющее жизненным циклом owin-server.)

Основные отличия Framework от Core

– Новый легковесный и модульный конвейер HTTP-запросов

– Возможность развертывать приложение как на IIS, так и в рамках своего собственного процесса

– Распространение пакетов платформы через NuGet

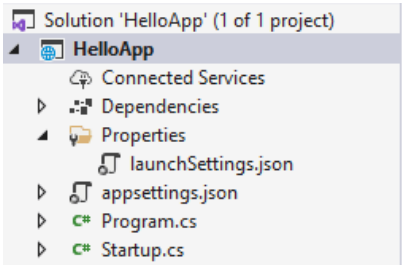
– Встроенная поддержка для внедрения зависимостей

– Расширяемость

– Кроссплатформенность (за счет clr для каждой платформы, которая вызывает свои системно-специфичные комманды)

– Развитие как open source, открытость к изменениям

Структура проекта ASP.NET Core



Connected Services: подключенные сервисы из Azure
Dependencies: все добавленные в проект пакеты и библиотеки/зависимости
Properties: узел, который содержит некоторые настройки проекта.
appsettings.json: файл конфигурации проекта в формате json
Program.cs: главный файл приложения, с которого и начинается его выполнение.
Startup.cs: файл, который определяет класс Startup и который содержит логику обработки входящих запросов. Класс **Startup** должен быть открытым и содержать следующие методы: **ConfigureServices** (необязательный, регистрирует сервисы используемые приложением) и **Configure** (обязательный, как приложение обрабатывает запрос), конструктор **Startup** (необязательный, начальная конфигурация приложения).

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

Статические файлы, такие как HTML, CSS, изображения и JavaScript, являются ресурсами, которые приложения ASP.NET Core предоставляют клиентам напрямую по умолчанию.

wwwroot – папка для статического контента (html, css, js,…), статический контент не отображается по умолчанию. Для отображения статического контента, необходимо подключить дополнительные nuget-пакеты; при создании Core-проекта автоматически подключен мегапакет Microsoft.AspNetCore.All в котором все есть. В Startup.cs app.UseStaticFiles(). С помощью специального метода расширения app.UseDefaultFiles() можно настроить отправку статических веб-страниц по умолчанию (**стартовые страницы**) без обращения к ним по полному. Будет искаться что-то на подобии index.html в wwwroot.

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

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

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        DefaultFilesOptions options = new DefaultFilesOptions();
        options.DefaultFileNames.Clear(); // удаляем имена файлов по умолчанию
        options.DefaultFileNames.Add("hello.html"); // добавляем новое имя файла
        app.UseDefaultFiles(options); // установка параметров

        app.UseStaticFiles();

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World");
        });
    }
}
```

Метод **UseDirectoryBrowser** позволяет пользователям просматривать содержимое каталогов на сайте. Перегрузка метода **UseStaticFiles**() позволяет сопоставить пути с определенными каталогами:

```
app.UseStaticFiles(new StaticFileOptions() // обрабатывает запросы к каталогу wwwroot/html
{
    FileProvider = new PhysicalFileProvider(
        Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\html")),
    RequestPath = new PathString("/pages")
});
```

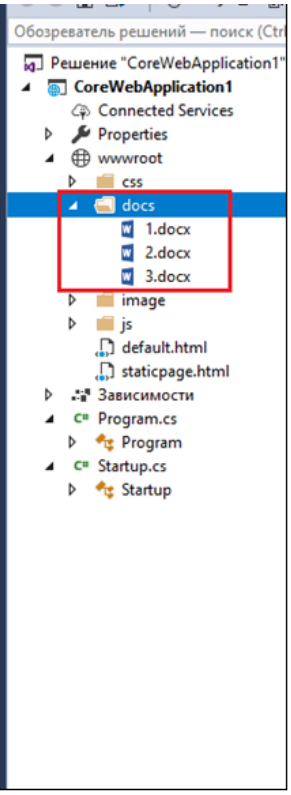
Метод **UseFileServer()** объединяет функциональность сразу всех трех вышеописанных методов UseStaticFiles, UseDefaultFiles и UseDirectoryBrowser.

Добавление заголовков:

```
app.UseStaticFiles(new StaticFileOptions
{
    OnPrepareResponse = ctx =>
    {
        ctx.Context.Response.Headers.Append("X-Smw60", "2018-01-25");
    }
});
app.UseStaticFiles();   //wwwroot
```

Файлы для скачивания будет отображаться в виде списка:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.FileProviders;
using System.IO;

namespace CoreWebApplication1
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            //services.AddDirectoryBrowser();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {

            DefaultFilesOptions options = new DefaultFilesOptions();
            options.DefaultFileNames.Clear();
            options.DefaultFileNames.Add("staticpage.html");
            app.UseDefaultFiles(options);

            app.UseStaticFiles();

            app.UseDirectoryBrowser(new DirectoryBrowserOptions
            {
                FileProvider = new PhysicalFileProvider(
                    Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "docs")),
                RequestPath = "/docs"
            });

        }
    }
}
```



Вывод в журнал:

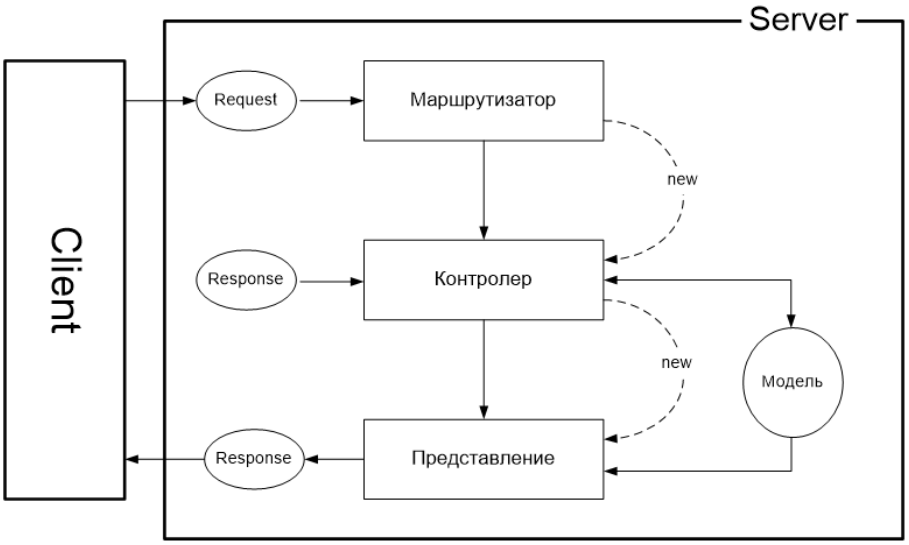
```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory logger)
{
    app.UseStaticFiles();

    logger.AddConsole();
```

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

```
public class HomeController : Controller
{
    private readonly ILogger logger;
    public HomeController(ILogger<HomeController> logger)
    {
        this.logger = logger;
    }

    public IActionResult Index()
    {
        return View();
    }
    public IActionResult One()
    {
        logger.LogInformation("Information:One ");
        return View();
    }
    public IActionResult Two()
    {
        return View();
    }
}
```

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

MVC (Model-View-Controller-Router) – архитектурный паттерн; включает четыре компонента: модель – данные; представление – отображение модели; контролер – обработка запросов, координация взаимодействия модели и представления. Маршрутизатор – выбор контроллера и действия.



НАСТРОЙКА MVC и МАРШРУТИЗАТОРА:

в ConfigureServices добавить поддержку MVC с помощью метода AddMvc()

В Configure определить маршрутизацию по умолчанию, указав имя контроллера,имя АКЦИИ и необязательный параметр id.

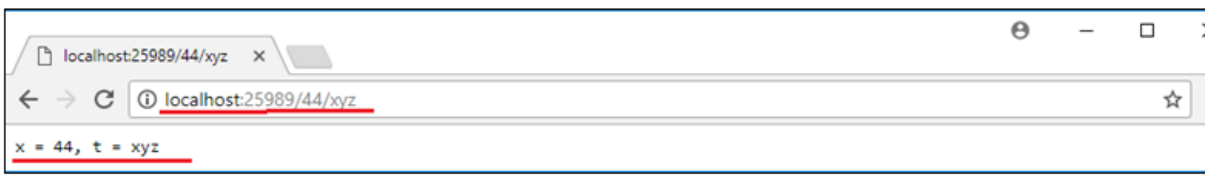
```
namespace CoreWebApplication2
{
    public class Startup
    {

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseStaticFiles();

            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });


        }
    }
}
```

применение атрибута Route для маршрутизации

```
public class Startup
{

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseMvc();
    }
}
```

```
public class HomeController : Controller
{
    [Route("")] //по умолчанию
    [Route("Default")]
    public IActionResult Default()
    {
        return Content("Application3:Home/Default");
    }
    [Route("Home/Index")]
    [Route("Home")]
    [Route("H/I")]
    [Route("0")]
    public IActionResult Index()
    {
        return Content("Application3:Home/Index");
    }
    [Route("One")]
    [Route("1")]
    public IActionResult One()
    {
        return Content("Application3:One");
    }
    [Route("Two")]
    [Route("2")]
    public IActionResult Two()
    {
        return Content("Application3:Two");
    }
}
```

```
[Route("{x:int}/{t:maxlength(3)}")]
public IActionResult Parm(int x, string t)
{
    return Content(String.Format("x = {0}, t = {1}", x, t));
}
```



```
public class ExerciseController : Controller
{
    [Route("square/{k:int}")]
    public IActionResult Square(int k)
    {
        ViewBag.Result = k * k;
        return View();
    }

    [Route("{n:minlength(1):maxlength(10)}/{y:int:min(1):max(150)}")]
    public IActionResult YearsOld(string n, int y)
    {
        return Content($"Your name is {n}, you're {y} years old");
    }
}
```

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

MVC-контроллер: контроллер получает ввод пользователя, обрабатывает его и посылает обратно результат обработки, например, в виде представления.

По соглашениям об именовании названия контроллеров должны оканчиваться на суффикс "Controller". Чтобы обратиться контроллеру из веб-браузера, надо в адресной строке набрать ***адрес_сайта/Имя_контроллера/Метод_контроллера*** **Действия(Акция) контроллера:** представляют такие методы, которые обрабатывают запросы по определенному URL. Так как запросы бывают разных типов, например, GET и POST (атрибуты [HttpGet], [HttpPost], [HttpDelete] или [HttpPut]). Метод контроллера обязательно должен быть public, но контроллер может включать и обычные методы, которые не являются методами действий. Методы действий могут принимать как простые типы данных(string, int), так и сложные(класс).

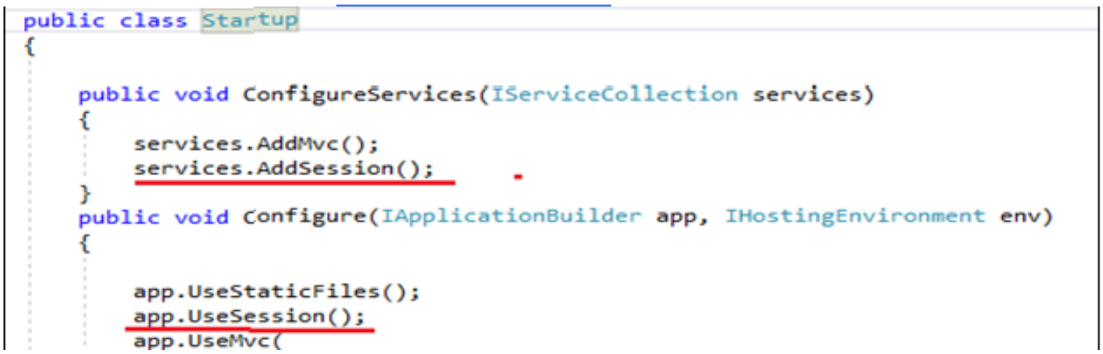
Контекст контроллера: Нам доступны следующие объекты контекста **ControllerContext,** он содержит свойства:

1. **HttpContext**: содержит информацию о контексте запроса
2. **ActionDescriptor**: возвращает дескриптор действия - объект **ActionDescriptor**, который описывает вызываемое действие **контроллера**
3. **ModelState**: возвращает словарь ModelStateDictionary, который используется для валидации данных, отправленных пользователем
4. **RouteData**: возвращает данные маршрута

Объект **HttpContext** инкапсулирует всю информацию о запросе. В частности, он определяет следующие свойства:

1. **Request**: содержит собственно информацию о текущем запросе.(содержит body, cookies, form, headers, path, query)
2. **Response**: управляет ответом (содержит body, cookies, contentType, headers, statusCode)
3. **User**: представляет текущего пользователя, который обращается к приложению
4. **Session**: объект для работы с сессиями

Сессия - несколько последовательных запросов, с общим идентификатором сессии, передается в куки.



```
public class Startup
{

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
        services.AddSession();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {

        app.UseStaticFiles();
        app.UseSession();
        app.UseMvc(
```

Объект **Session** определяет ряд свойств и методов, которые мы можем использовать:

1. **Keys**: свойство, представляющее список строк, который хранит все доступные ключи
2. **Clear**(): очищает сессию
3. **Get**(string key): получает по ключу key значение, которое представляет массив байтов
4. **GetInt32**(string key): получает по ключу key значение, которое представляет целочисленное значение
5. **GetString**(string key): получает по ключу key значение, которое представляет строку
6. **Set**(string key, byte[] value): устанавливает по ключу key значение, которое представляет массив байтов
7. **SetInt32**(string key, int value): устанавливает по ключу key значение, которое представляет целочисленное значение value
8. **SetString**(string key, string value): устанавливает по ключу key значение, которое представляет строку value
9. **Remove**(string key): удаляет значение по ключу

Для разграничения сессий для них устанавливается идентификатор. Каждая сессия имеет свой

идентификатор, который сохраняется в куках. По умолчанию эти куки имеют название ".AspNet.Session". И также по умолчанию куки имеют настройку CookieHttpOnly=true, поэтому они не доступны для клиентских скриптов из браузера. Но мы можем переопределить ряд настроек сессии с помощью свойств объекта **SessionOptions:**

1. **Cookie.Name**: имя куки
2. **Cookie.Domain**: домен, для которого устанавливаются куки
3. **Cookie.HttpOnly:** доступны ли куки только при передаче через HTTP-запрос
4. **Cookie.Path:** путь, который используется куками
5. **Cookie.Expiration:** время действия куки в виде объекта System.TimeSpan
6. **Cookie.IsEssential:** при значении true указывает, что куки критичны и необходимы для работы этого приложения
7. **IdleTimeout:** время действия сессии в виде объекта System.TimeSpan при неактивности пользователя. При каждом новом запросе таймаут сбрасывается. Этот параметр не зависит от Cookie.Expiration.

Сессии настраиваются в классе -> **ConfigureServices**(там же **session options**), а также в Configure (app.UseSession)**startup**

Результат работы действия(акции) – это тот объект, который возвращается методом после обработки запроса. Результатом действия может быть все, что угодно: страница, файл, строка, экземпляр класса, void и т.п.

IActionResult: содержит один метод **ExecuteResultAsync**, который принимает контекст (если надо будет создать свой класс)

Данный интерфейс реализован в таких классах:

1. **ContentResult**: пишет указанный контент напрямую в ответ в виде строки
2. **EmptyResult**: отправляет пустой ответ в виде статусного кода 200
3. **NoContentResult**: также отправляет пустой ответ, только в виде статусного кода 204
4. **FileResult**: является базовым классом для всех объектов, которые пишут набор байтов в выходной поток. Предназначен для отправки файлов
5. **FileStreamResult**: бинарный поток в выходной ответ
6. **JsonResult**: возвращает в качестве ответа объект или набор объектов в JSON
7. **ViewResult:** производит рендеринг представления и отправляет результаты рендеринга в виде html-страницы клиенту

Внедрение зависимостей: как и любой класс, контроллер может получать сервисы приложения через

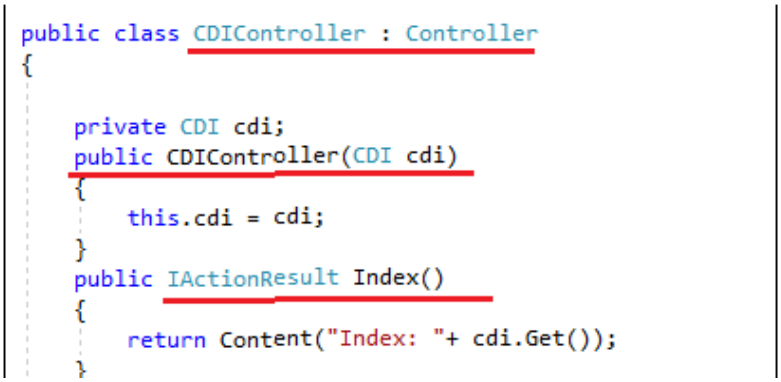
механизм dependency injection. В контроллере это можно сделать следующими способами:
1. Через конструктор
2. Через параметр метода, к которому **применяется атрибут FromServices**
3. Через свойство **HttpContext.RequestServices** в методах

Через конструктор

1. Приложение получает запрос к методу контроллера
2. Фреймворк MVC обращается к провайдеру сервисов для создания объекта контроллера
3. Провайдер сервисов смотрит на конструктор класса(контроллера) и видит, что там имеется зависимость от интерфейса
4. Провайдер сервисов среди зарегистрированных зависимостей ищет класс, который представляет реализацию интерфейса
5. Если нужная зависимость найдена, то провайдер сервисов создает объект класса, который реализует интерфейс
6. Затем провайдер сервисов создает объект контроллера, передавая в его конструктор ранее созданную реализацию
7. В конце провайдер сервисов возвращает созданный объект контроллера инфраструктуре MVC, которая использует контроллер для обработки запроса

В классе Startup необходимо прописать services.AddTransient<interfaceName>();

Потом:



```
public class CDIController : Controller
{

    private CDI cdi;
    public CDIController(CDI cdi)
    {
        this.cdi = cdi;
    }
    public IActionResult Index()
    {
        return Content("Index: "+ cdi.Get());
    }
```

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

Фильтры

Для создания фильтров необходимо унаследовать класс FilterAttribute и реализовать один из интерфейсов фильтра (IActionFilter, IAuthenticationFilter, IResultFilter, IAuthorizationFilter, IExceptionFilter)

IActionFilter

Методы интерфейса IActionFilter:

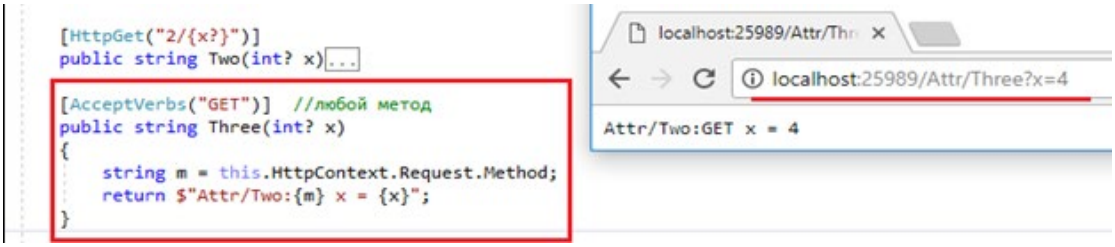
- **OnActionExecuting()** выполняется при вызове метода контроллера до его непосредственного выполнения;
- **OnActionExecuted()** выполняется после выполнения метода контроллера;
- **OnActionExecutionAsync()** представляет асинхронную версию метода OnActionExecuting().

Атрибуты методов контроллера

Для указания типа запроса HTTP нам надо применить к методу один из атрибутов: **[HttpGet], [HttpPost], [HttpPut], [HttpDelete]** и **[HttpHead]**. Если атрибут явным образом не указан, то метод может обрабатывать все типы запросов: GET, POST, PUT, DELETE и др.

[AcceptVerbs]

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



Принцип передачи параметров в метод действия

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

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

public string Sum(int a, int b) { return $”Sum {a} и {b}”; }

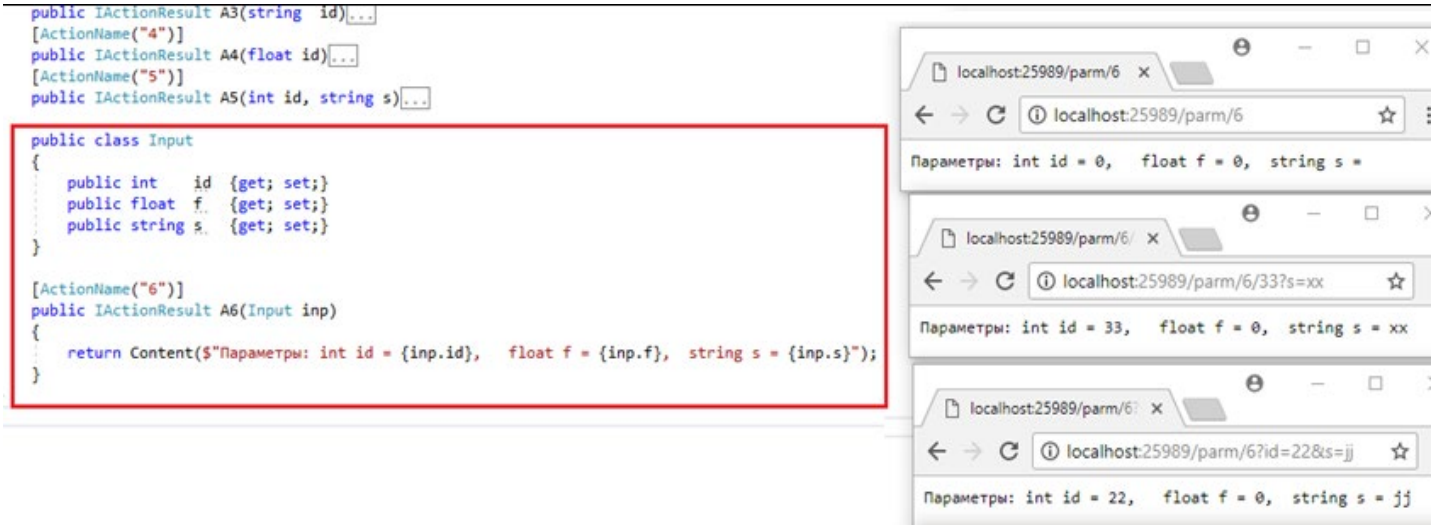
http://localhost:1111/Home/Sum/

public string Sum(int a = 2, int b = 4) { return $”Sum {a} и {b}”; }

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

Передача сложных объектов

Класс Input определяет 3 свойства. Теперь в контроллере метод A6 принимает параметр Input. Здесь параметры строки запроса должны соответствовать по имени свойствам объекта. Регистр названий при этом не учитывается.



Передача массивов

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

public string Sum(int[] nums){return ${nums.Sum()};}

Передача данных в запросе POST

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

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

```
<form method="post" action="~/Home/Area">
    <label>Высота:</label><br />
    <input type="number" name="height" /><br />
    <label>Основание:</label><br />
    <input type="number" name="altitude" /><br />
    <input type="submit" value="Отправить" />
</form>
```

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
[HttpPost]
public string Area(Geometry geometry)
{
    return $"Площадь треугольника с основанием {geometry.Altitude} и высотой {geometry.Height} равна {geometry.GetArea()}";
}
```

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

Фильтры позволяют вносить в приложение некоторую логику, которая должна отрабатывать до вызова действий контроллера. Например, пользователь вызывает определённое действие некоторого контроллера, и нам нужно проверить, авторизовался-ли данный пользователь, и после этого уже выполнять определённые действия. Для этого и нужны фильтры.

Фильтры реализованы как атрибуты, благодаря чему позволяют уменьшить объём кода в контроллере. Фильтры могут применяться как ко всему классу, так и к отдельным его методам, свойствам и полям.

Типы фильтров (фильтр каждого типа выполняется на определенном этапе конвейера фильтров):

– **Фильтры аутентификации** - IAuthenticationFilter (реализуемый интерфейс). Стандартная реализация отсутствует. Данный фильтр определяет аутентифицирован-ли клиент. Он запускается до выполнения любого другого фильтра или метода действий. Интерфейс представляет два метода: OnAuthentication() и OnAuthenticationChallenge().

– **Фильтр авторизации** - IAuthorizationFilter. Стандартная реализация - AuthorizeAttribute. Данный фильтр определяет, имеет-ли пользователь доступ к данному ресурсу. Запускается после фильтра,

аутентификации, но до любого другого фильтра или метода действия. Интерфейс представляет метод: OnAuthorization().

– **Фильтры ресурсов -** выполняются после фильтров авторизации. Его метод OnResourceExecuting() выполняется до всех остальных фильтров и до привязки

– **Фильтр действий** - IActionFilter. Стандартная реализация - ActionFilterAttribute. Фильтр, применяемый к действия. Может запускаться как до, так и после выполнения метода действий. Интерфейс представляет два метода: OnActionExecuting() и OnActionExecuted().

– **Фильтр исключений** - IExceptionFilter. Стандартная реализация - HandleErrorAttribute. Атрибут для обработки исключений, выбрасываемых методом действий и результатом действий.Интерфейс представляет метод: OnException().

– **Фильтр результата действий** - IResultFilter. Стандартная реализация - ActionFilterAttribute. Фильтр, применяемый к результатам действий. Может запускаться как до, так и после выполнения результата действий. Интерфейс представляет два метода: OnResultExecuting() и OnResultExecuted().

Вместе все эти типы фильтров образуют **конвейер фильтров (filter pipeline)**, который встроен в процесс обработки запроса в MVC и который начинает выполняется после того, как инфраструктура MVC выбрала метод контроллера для обработки запроса.

На рисунке ниже представлен **пользовательский фильтр действий.**

```
[AAFilter]
[ARFilter]
[AEFilter]
0 references
public ActionResult AE()
{
    throw new Exception("Exception in AE Action");
    return Content($"AE worked right\n");
}

2 references
public class AAFilter : FilterAttribute, IActionFilter
{
    0 references
    public void OnActionExecuting(ActionExecutingContext filterContext)
    {
        filterContext.HttpContext.Response.Write("AA:Вызов перед вызовом метода действия\n");
    }

    0 references
    public void OnActionExecuted(ActionExecutedContext filterContext)
    {
        filterContext.HttpContext.Response.Write("AA:Вызов после работы метода действия\n");
    }

}
```

Применение:

Используются посредством задания тегов. Могут иметь различную область действий: на метод, на класс контроллера, на класс страницы Razor Page, глобальную область действий на все методы всех контроллеров.

```
public class HomeController : Controller
{
    [SimpleResourceFilter]
    public IActionResult Index()
    {
        return View();
    }
}
```

Для определения фильтра как глобального нам надо изменить в методе ConfigureServices() класса Startup подключение соответствующих сервисов MVC

```
services.AddMvc(options =>
{
    options.Filters.Add(typeof(SimpleResourceFilter));
```

Для выхода из конвейера необходимо установить свойства Result переданного контекста.

```
public void OnResourceExecuting(ResourceExecutingContext context)
{
    context.Result = new ContentResult { Content = "Ресурс не найден" };
}
```

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

MVC-представление - это файл с расширением cshtml, содержащий HTML, CSS, JavaScript и Razor-конструкции. В простейшем случае cshtml-файл может содержать только html-разметку. В шаблоне MVC представление отвечает за отображение данных приложения и взаимодействие с пользователем.

Как правило, файлы представлений объединяются в папки с именами, соответствующими отдельным контроллерам приложения. Эти папки находятся в папке Views в корне приложения.

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

Метод View (return View();) по умолчанию возвращает представление с тем же именем, что и у метода действия, из которого он был вызван. Например, имя метода About ActionResult контроллера используется

для поиска файла представления с именем About.cshtml. Сначала среда выполнения ищет представление в папке Views/[имя_контроллера]. Если подходящее представление в ней не найдено, поиск производится в папке Shared.

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

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

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

```
return View("Views/Home/About.cshtml");
```

Для указания представлений в разных каталогах можно также использовать относительный путь без расширения CSHTML . Внутри HomeController можно вернуть представление Index из папки Manage с помощью следующего относительного пути:

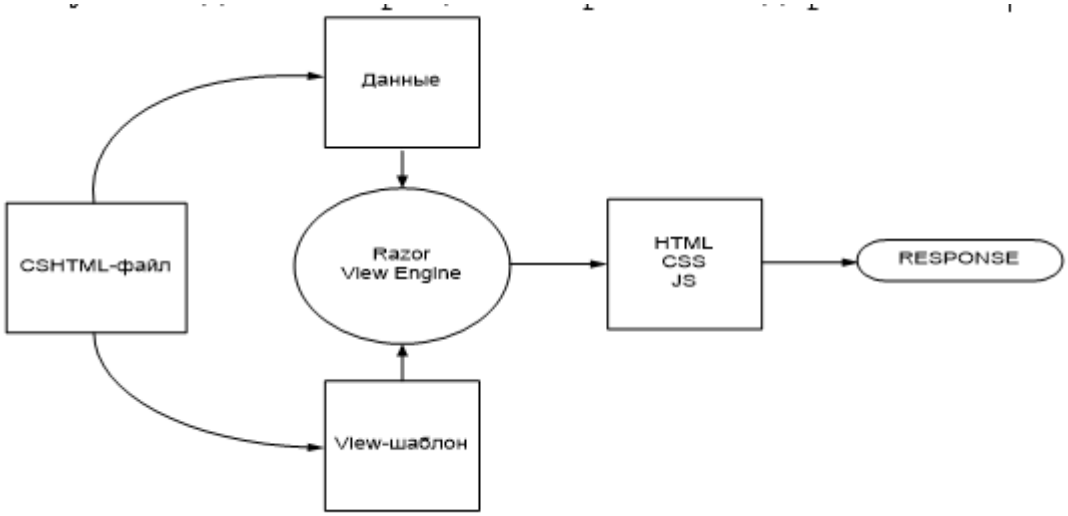
```
return View("../Manage/Index");
```

Частичные представления и компоненты представлений используют похожие (но не одинаковые) механизмы обнаружения **return PartialView().**

Настроить соглашение по умолчанию, определяющее способ поиска представлений в приложении, можно с помощью пользовательской реализации IViewLocationExpander.

Жизненный цикл представления.

Cshtml-файл компилируется в сборку, которая используется для генерации Response-содержимого.



Методы рендеринга представления в web-страницу.

За работу с представлениями отвечает объект ViewResult. Он производит рендеринг представления в веб-страницу и возвращает ее в виде ответа клиенту.

Чтобы возвратить объект ViewResult используется метод View (return View();)

Вызов метода View возвращает объект ViewResult. Затем уже ViewResult производит рендеринг определенного представления в ответ. По умолчанию контроллер производит поиск представления в проекте по следующим путям:

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

Метод View() имеет четыре перегруженных версии:

- View(): для генерации ответа используется представление, которое по имени совпадает с вызывающим методом
- View(string viewName): в метод передается имя представления, что позволяет переопределить используемое по умолчанию представление
- View(object model): передает в представление данные в виде объекта model
- View(string viewName, object model): переопределяет имя представления и передает в него данные в виде объекта model

Способы передачи данных из контроллера в представление:

1. Строго типизированные данные передаются с помощью viewmodel

Модель представления является во многих случаях более предпочтительным способом для передачи данных в представление. Для передачи данных в

представление используется одна из версий метода View:

```
public IActionResult Index()
{
    List<string> countries = new List<string> { "Бразилия", "Аргентина", "Уругвай", "Чили" };
    return View(countries);
}
```

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

```
@model List<string>
@{
    ViewBag.Title = "Index";
}

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

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

2. Слаботипизированные данные передаются с помощью ViewData и ViewBag

ViewData представляет словарь из пар ключ-значение:

```
public IActionResult Index()
{
    ViewData["Message"] = "Hello ASP.NET Core";

    return View();
}
```

Здесь динамически определяется во ViewData объект с ключом "Message" и значением "Hello ASP.NET Core". При этом в качестве значения может выступать любой объект. И после этому мы можем его использовать в представлении:

```
@{
    ViewData["Title"] = "Index";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>
```

Причем не обязательно устанавливать все объекты во ViewData в контроллере. Так, в данном случае объект с ключом "Title" устанавливается непосредственно в представлении.

ViewBag во многом подобен ViewData. Он позволяет определить различные свойства и присвоить им любое значение. И не важно, что изначально объект ViewBag не содержит **никакого свойства, оно определяется *динамически***. При этом свойства ViewBag могут содержать не только простые объекты типа string или int, но и сложные данные:

```
public IActionResult Index()
{
    ViewBag.Countries = new List<string> { "Бразилия", "Аргентина", "Уругвай", "Чили" };
    return View();
}
```

```
<h3>В списке @ViewBag.Countries.Count элемента:</h3>
```

Правда, чтобы выполнять различные операции, может потребоваться приведение типов, как в данном случае. Директива @model определяет модель представления, то есть позволяет связать представление с передаваемой моделью данных.

27. ASP.NET CORE: MVC-представление, директивы @using, @function, @inherits, #inject. Пример. (если что всё с microsoft documentation)

директива — это инструкции для указания необязательных настроек, таких как регистрация пользовательского элемента управления и языка страниц.

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

```
@using (Html.BeginForm())
{ <div><input type="email" id="Email"
value=""><button>Register</button> </div> }
```

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

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

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

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

CSHTML

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

@inject - директива для внедрения зависимости в представление

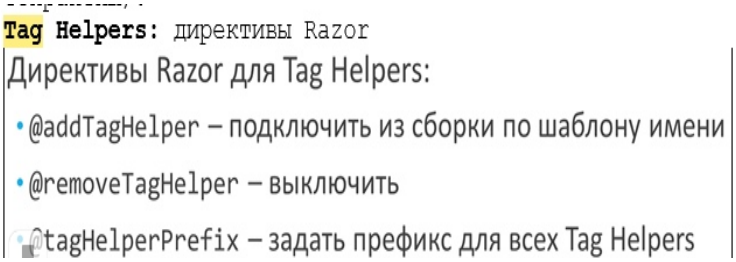
```
public class
VuiwInject{
 private string
txt=”empty”;
 public ViewInject()
{Console.WriteLine(“c
onstructor”);}

 public string Set(string
s)
{Console.WriteLine(thi
s.txt=s);
   retrun this.txt; }

 public string
Get(string s)
{Console.WriteLine(thi
s.txt);
   retrun this.txt; }
```

```
public void
ConfigureServices(IserviceColle
ction services){
services.addMvc();
services.AddSingleton
<Namespace.Inject.ViewInject>(
)
}
CSHTML
@inject
Namespace.Injects.ViewInject
VI
@{ Layout=null;}
<body>
<p>VI get = @VI.Get()</p>
@{VI.Set(“ok”);}
<p>VI get = @VI.Get()</p>
</body>
```

}	

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

Tag-хелперы представляют собой функциональность (классы), предназначенную для генерации HTML-разметки. Tag-хелперы используются в представлениях и выглядят как обычные html-элементы или атрибуты, однако при работе приложения они обрабатываются движком Razor на стороне сервера и в конечном счете преобразуются в стандартные html-элементы.




```
@helper BookList(IEnumerable<BookStore.Models.Book> books)
{
    <ul>
        @foreach (BookStore.Models.Book b in books)
        {
            <li>@b.Name</li>
        }
    </ul>
}
```

Данный хелпер мы можем определить в любом месте представления. И также в любом месте представления мы можем его использовать, передавая в него объект IEnumerable<BookStore.Models.Book>:

```
<h3>Список книг</h3>
@BookList(ViewBag.Books)
<!-- или если используется строго типизированное представление -->
@BookList(Model)
```

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

@addTagHelper - позволяет подключить Tag-хелпер из сборки по шаблону имени

```
@using AuthoringTagHelpers
@addTagHelper *,
Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

Первый параметр директивы указывает на tag-хелперы, которые будут **доступны во всех представлениях из папки Views, а** второй параметр определяет библиотеку хелперов (в которой находится вспомогательная функция тега). В данном случае директива использует синтаксис подстановок - знак звездочки ("*") означает, что все хелперы из библиотеки.

@removeTagHelper имеет те же 2 параметра, как и addTagHelper, но он удаляет Tag-хелперы, которые были ранее добавлены. Например, примененный к определенному представлению, он удаляет указанный Tag-хелпер из View.

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

В ASP.NET MVC Core **представления** – файлы с расширением cshtml, которые содержат код пользовательского интерфейса в основном на языке html, а также используют конструкции Razor – специального движка представлений, который позволяет использовать конструкции на языке C# в разметке html.

Кроме обычных представлений и мастер-страниц можно также использовать **частичные представления** или **partial views**. Особенность - их можно встраивать в другие обычные представления. Частичные представления могут использоваться так же, как и обычные, однако наиболее удобной областью их использования является рендеринг результатов AJAX-запроса. По своему действию частичные представления похожи на секции, только их код выносится в отдельные файлы.

Частичные представления полезны для создания различных панелей веб-страницы, например, панели меню, блока входа на сайт, каких-то других блоков, которые могут переиспользоваться много раз.

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

Пример:

```
public class HomeController : Controller
{public                    ActionResult
GetMessage()=>return
PartialView("_GetMessage");}
```

В папку Views/Home добавляется новое представление _GetMessage.cshtml, в котором будет содержаться razor разметка.

Можно также встроить частичное представление в обычное с помощью метода **Html.PartialAsync().** Он является асинхронным и возвращает объект IHtmlContent, который представляет html-содержимое и который обернут в объект Task<TResult>. В качестве параметра в метод передается имя представления. Обращения к методу GetMessage() в контроллере при этом не происходит.

```
<h2>Представление Index.cshtml</h2>
@await Html.PartialAsync("_GetMessage")
```

Также частичное представление можно встроить с помощью метода Html.RenderPartialAsync. Этот метод также принимает имя представления, только он используется не в строчных выражениях кода Razor, а в блоке кода, то есть обрамляется фигурными скобками.

```
@{await Html.RenderPartialAsync("_GetMessage");}
```

Html.RenderPartialAsync напрямую пишет вывод в выходной поток в асинхронном режиме, поэтому может работать чуть быстрее, чем Html.PartialAsync.

В методы Html.PartialAsync и Html.RenderPartialAsync можно в качестве второго параметра указать модель, тем самым передав ее в частичное представление. В итоге получится

стандартное строго типизированное представление. Пример:

```
@await Html.PartialAsync("_GetMessage", new List<string> { "Lumia 950", "iPhone 6S", "Samsung Galaxy s 6", "LG G 4" }
```

Встроенные хелперы - хелперы, которые предоставляются фреймворком MVC и которые позволяют генерировать ту или иную разметку (код элементов форм и т.д.).

Хелпер Html.BeginForm генерирует разметку для формы с отправкой данных определенному действию в определенный контроллер. 1-й параметр - имя действия, 2-й параметр - имя контроллера, 3-1 параметр - тип запроса. По сути генерирует <form>...</form>

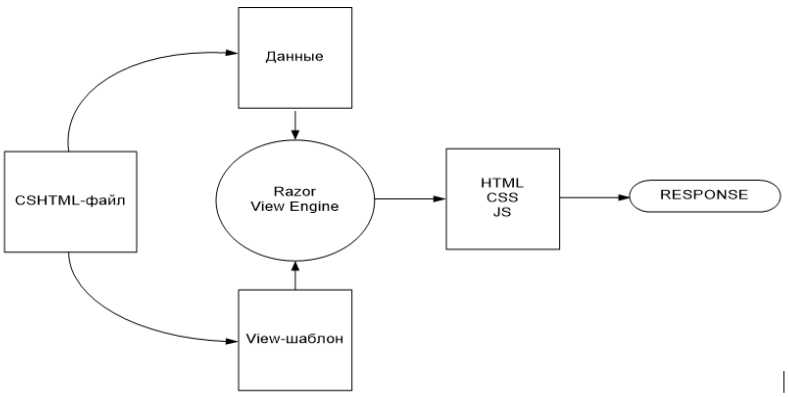
```
@using(Html.BeginForm("Create", "Home", FormMethod.Post))
{
    <p> <input type="text" name="name" /> </p>
    <p> <input type="submit" value="Отправить" /> </p>
}
```

Extension Method	Strongly Typed Method	Html Control
Html.ActionLink()	NA	<a>
Html.TextBox()	Html.TextBoxFor()	<input type="textbox">
Html.TextArea()	Html.TextAreaFor()	<input type="textarea">
Html.CheckBox()	Html.CheckBoxFor()	<input type="checkbox">
Html.RadioButton()	Html.RadioButtonFor()	<input type="radio">
Html.DropDownList()	Html.DropDownListFor()	<select> <option> </select>
Html.ListBox()	Html.ListBoxFor()	multi-select list box: <select>
Html.Hidden()	Html.HiddenFor()	<input type="hidden">
Html.Password()	Html.PasswordFor()	<input type="password">
Html.Display()	Html.DisplayFor()	HTML text: ""
Html.Label()	Html.LabelFor()	<label>
Html.Editor()	Html.EditorFor()	Generates Html controls based on data type of specified model property e.g. textbox for string property, numeric field for int, double or other numeric type.

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

Представление - файл с расширением cshtml, содержащий html, css, JavaScript и Razor-конструкции. cshtml-файл компилируется в сборку, которая используется для генерации Response-содержимого.

Razor View Engine – движок представления, компонент ASP.NET Core MVC-фреймворка, предназначенный для генерации содержимого Response на основе содержимого cshtml-файла. cshtml-файл компилируется в сборку, которая используется для генерации Response-содержимого.



Фактически html-хелперы представляют собой вспомогательные методы, цель которых - генерация html-разметки. Вспомогательные методы – методы расширения для IHtmlHelper, IHtmlHelper<TModel> (для типизированного представления) или IUriHelper.

```
public static class Helpers
{
    0 references
    public static MvcHtmlString HelperButton(this HtmlHelper html, string mission)
    {
        return new MvcHtmlString($"<input type='submit' value='{mission}' class='btn btn-default' />");
    }
```

```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        @Html.HelperButton("Create")
    </div>
</div>
```

Классы, реализующие IHtmlContent:

- HtmlString
- LocalizedHtmlString
- StringHtmlContent
- HelperResult
- TagBuilder

Вспомогательные методы сохраняют состояние соответственных связанных элементов в ModelState. Для создания html-тегов в хелпере можно использовать класс Microsoft.AspNetCore.Mvc.Rendering.TagBuilder.

```
public static class ListHelper
{       public static HtmlString CreateList(this IHtmlHelper html, string[] items)
    {
            TagBuilder ul = new TagBuilder("ul");
            foreach (string item in items)
            {
           TagBuilder li = new TagBuilder("li"); // добавляем текст в li
            li.InnerHtml.Append(item);  // добавляем li в ul
              ul.InnerHtml.AppendHtml(li);  }
        ul.Attributes.Add("class", "itemsList");
          var writer = new System.IO.StringWriter();
      ul.WriteTo(writer, HtmlEncoder.Default);
          return new HtmlString(writer.ToString());
```

```
}}
```

В конструктор TagBuilder передается элемент, для которого создается тег. TagBuilder имеет ряд свойств и методов, которые можно использовать:

- Свойство InnerHtml позволяет установить или получить содержимое тега в виде строки. Чтобы манипулировать этим свойством, можно вызвать один из методов:
  - Append(string text): добавление строки текста внутрь элемента
  - AppendHtml(IHtmlContent html): добавление в элемент кода html в виде объекта IHtmlContent - это может быть другой объект TagBuilder
  - Clear(): очистка элемента
  - SetContent(string text): установка текста элемента
  - SetHtmlContent(IHtmlContent html): установка внутреннего кода html в виде объекта IHtmlContent
- Свойство Attributes позволяет управлять атрибутами элемента
- Метод MergeAttribute() позволяет добавить к элементу один атрибут
- Метод AddCssClass() позволяет добавить к элементу класс css
- Метод WriteTo() позволяет создать из элемента и его внутреннего содержимого строку при помощью объектов TextWriter и HtmlEncoder.

```
@using HtmlHelpersApp.App_Code

<h3>Города</h3>
@Html.CreateList(cities)
<br />
<h3>Страны</h3>
<!-- или можно вызвать так -->
@ListHelper.CreateList(Html, countries)
```

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

Entity Framework - популярная ORM для взаимодействия с БД через объекты, имеется поддержка Linq EF упрощающая работу с БД.

Принцип Code подразумевает использование подхода Code First. Основные действия для работы с CF:

1. Создание моделей с автосвойствами, описывающими сущности БД;
2. Создание контекста базы данных (16 вопрос)
3. Создание миграции через .Net CLI: dotnet ef add migrations “Название”
4. Применение миграций для БД: dotnet ef database update

ModelState - объект валидации на стороне сервера. Для корректной валидации модель, использующаяся в параметрах Action должна обладать атрибутами валидации на свойствах из пространства имен DataAnnotations:

```
public class Person {
        [Required (ErrorMessage = "Игорь")]
        public string Name { get; set; }
```

[Required]
public string Password { get; set; }}

После этого в самом Action или фильтре можно проверять наличие ошибок в модели.
ModelState.IsValid - показывает, валидная ли модель.
ModelState["Name"].Errors - коллекция ошибок в модели
Встроенные атрибуты валидации:
[RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}", ErrorMessage = "Некорректный адрес")]
[Required] - обязательно должно быть [Range(1, 20)] - диапазон от 1 до 20
[Url/Email] - формат URL/почты
[Compare("Password")] - св-во сравнивается со свойством Passwords
Кастомные атрибуты валидации (жирным курсивом выделена обязательная реализация):

```
public class ClassicMovieAttribute : 
ValidationAttribute
{   public int Year { get; }
    public ClassicMovieAttribute(int 
year)
    { Year = year;  }
```

```
public string GetErrorMessage() =
$"must have year no later than
{Year}.";
protected override ValidationResult
IsValid(object value, ValidationContext
validationContext)
    {if (releaseYear > Year) {return
new
ValidationResult(GetErrorMessage());}
        return
ValidationResult.Success; }}
```

[ClassicMovie(2020)]

public int Year {get;set;}