**СМЕЛОВ**

****

**«Системная система систем систематизировала систему ой пизда короче учите»**

ВВЕДЕНИЕ

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



СТРУКТУРА

**Структура:** службы, организации и ресурсы(услуги).

**Internet-служба**: другое название Internet-сервис, один из видов Internet-ресурса, имеющий специальное назначение (DNS, WWW, E-mail, FTP, ICQ, Telnet).

**Службы (сервисы) Интернет**:

* DNS – система для преобразования доменных имен в IP-адрес,
* E-mail (STMP, POP3, IMAP);
* IRC (обмен сообщений в реальном времени);
* FTP;
* Telnet (управление удаленным компьютером в терминальном режиме);
* WWW.

ОРГАНИЗАЦИИ

**ISOC:** Internet Society – международная организация, занимающаяся развитием сети Internet. ISOC владелец RFC-стандартов. ISOC обеспечивает правовую поддержку и финансирует все другие организации, связанные с деятельностью Internet.

**IETF:** Internet Engineering Task Force - рабочая группа проектирования Internet. Публикует RFC (Request for). Задачи IETF описаны в RFC 4677.

**RFC:** Жизненный цикл описан в RFC 2026.

**Этапы ЖЦ:**

* Интернет-проект (Internet Draft): Предложение без официального статуса, удаляется через 6 месяцев после последнего изменения.
* Предложенный стандарт (Proposed Standard): Проект признан успешным, получает номер RFC. Наличие программной реализации желательно, но не обязательно.
* Проект стандарта (Draft Standard): Принят сообществом, существуют две независимые совместимые реализации. Считается стабильным и рекомендуемым для реализации, допускаются мелкие правки.
* Стандарт Интернета (Internet Standard): Высший уровень, с успешным опытом применения. Имеет собственную нумерацию STD, список представлен в документе STD 1 (RFC 5000). Достигли этого уровня только несколько десятков из более чем трех тысяч RFC.
* Исторические документы (Historic): Устаревшие RFC, замененные новыми версиями или вышедшие из употребления.

**IAB:** Internet Architecture Board - совет по архитектуре Internet, одна из комиссий IETF, имеет консультативный статус при ISOC.

**ICANN: Internet Corporation for Assigned Names and Numbers** – корпорация по управлению доменными именами и IP-адресами.

**IANA: Internet Assigned Numbers Authority –** Администрация адресного пространства Internet. Под контролем ICANN. Кроме того, регистрирует типы данных **MIME**.

**MIME**: **Multipurpose Internet Mail Extensions** - многоцелевые расширения Internet-почты. Используется и как стандарт кодирования Internet-сообщений.

**W3C: World Wide Web Consortium –** организация разрабатывающая и внедряющая web-стандарты.

**WHATWG:**Web Hypertext Application Technology Working Group – рабочая группа по развитию web, отделился от W3C, разработка HTML5 и Browser API.

ПРОГРАММИРОВАНИЕ В ИНТЕРНЕТ

**Программирование в Internet:** разработка распределенного (сетевого) приложения, представляющего собой один или несколько связанных Internet-ресурсов. Для связи между компонентами приложения применяются протоколы HTTP, WebSocket, SMTP, POP3, IMAP, WebDav.

**Технологии программирования в Internet:**

* LAMP (Linux, Apache, MySQL, PHP/Perl, кроссплатформенная технология);
* **Java EE (кроссплатформенная технология на основе Java);**
* **ASP.NET** (ASP.NET CORE – кроссплатформенная версия);
* **NODE.JS** (кроссплатформенная технология, на основе JavaScript);
* Ruby on Rails (кроссплатформенная технология на основе Ruby и фреймворка MVC для web-разработки);
* Python Django (кроссплатформенная технология на основе Python и фреймворка MVC для web-разработки).

WEB-ПРИЛОЖЕНИЕ

**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).

**Архитектура веб-приложения:** клиент и сервер.

**Клиент-серверная архитектура -** модель организации вычислительных систем, в которой задачи распределены между клиентами и серверами.

РЕЖИМЫ ПЕРЕДАЧИ ДАННЫХ

Симплексный режим: В симплексном режиме отправитель может отправлять данные, но не может их получать.

Дуплекс: Двусторонняя связь, клиент и сервер могут отправлять и получать данные одновременно.

Полудуплекс: Односторонняя связь, данные могут передаваться в одном направлении в каждый момент времени.

ПРОТОКОЛЫ

* HTTP (Hypertext Transfer Protocol): Протокол передачи гипертекста, используемый для передачи данных в вебе.
* HTTPS (Hypertext Transfer Protocol Secure): Защищенная версия HTTP с использованием шифрования SSL/TLS для безопасной передачи данных.
* FTP (File Transfer Protocol): Протокол передачи файлов, используемый для обмена файлами между компьютерами в сети.
* SMTP (Simple Mail Transfer Protocol): Протокол передачи почты, применяемый для отправки электронной почты между серверами электронной почты.
* POP3 (Post Office Protocol 3): Протокол получения почты, используемый для извлечения сообщений с почтового сервера.
* IMAP (Internet Message Access Protocol): Протокол доступа к почтовому ящику в Интернете, предоставляющий более расширенные функции, чем POP3.
* TCP (Transmission Control Protocol): Протокол управления передачей, обеспечивающий надежную и упорядоченную передачу данных в сети.
* UDP (User Datagram Protocol): Протокол пользовательских датаграмм, предоставляющий передачу данных без гарантии доставки или упорядочивания.
* DNS (Domain Name System): Протокол системы доменных имен, используемый для преобразования доменных имен в IP-адреса.
* SSH (Secure Shell): Протокол для безопасной удаленной работы с устройствами через зашифрованное соединение.
* Telnet: Протокол удаленного доступа, использующий небезопасное текстовое соединение.
* SNMP (Simple Network Management Protocol): Протокол для управления и мониторинга сетевых устройств.
* ICMP (Internet Control Message Protocol): Протокол для передачи сообщений об ошибках и управляющей информации в сети IP.
* WebSockets: Протокол для полнодуплексной связи между клиентом и сервером через веб.
* MQTT (Message Queuing Telemetry Transport): Протокол для передачи сообщений между устройствами в условиях низкой пропускной способности сети.

**TCP/IP (Transmission Control Protocol/Internet Protocol)** — это набор протоколов, обеспечивающий передачу данных в интернете и локальных сетях. Он состоит из нескольких уровней: IP отвечает за адресацию и маршрутизацию пакетов, TCP обеспечивает надежную, ориентированную на соединение передачу данных, а другие протоколы, такие как UDP, HTTP и FTP, выполняют специфические функции для различных приложений. TCP/IP масштабируем и универсален, что делает его основой современных сетевых коммуникаций.

HTTP

**HTTP (Hypertext Transfer Protocol):** Протокол передачи гипертекста, используемый для передачи данных в вебе.

**HTTPS** — это защищенная версия HTTP, использующая SSL/TLS для шифрования данных, что обеспечивает защиту конфиденциальности и целостности информации.

**Методы HTTP:**

* + - 1. **GET**: Запрос данных с сервера. Используется для получения информации.
      2. **POST**: Отправка данных на сервер. Обычно используется для создания новых ресурсов.
      3. **PUT**: Обновление существующих данных на сервере. Заменяет весь ресурс.
      4. **DELETE**: Удаление данных с сервера.
      5. **PATCH**: Частичное обновление существующего ресурса.
      6. **OPTIONS**: Запрос информации о доступных методах и параметрах для конкретного ресурса.
      7. **HEAD**: Запрос заголовков ответа без тела. Полезен для проверки доступности ресурса.
      8. **TRACE**: Отладочный метод, который возвращает запрос, полученный сервером, что позволяет проследить за маршрутом.
      9. **CONNECT:** используется для установления туннельного соединения с сервером через прокси.

СВОЙСТВА HTTP

* версии HTTP/1.1 – действующий (текстовый), HTTP/2 черновой (не распространен, бинарный);
* два типа абонентов: клиент и сервер;
* два типа сообщений: request и response;
* от клиента к серверу – request;
* от сервера к клиенту – response;
* на один request всегда один response, иначе ошибка;
* одному response всегда один request, иначе ошибка;
* TCP-порты: 80, 443;
* для адресации используется URI или URN;
* стейтлесс-протокол (взаимодействие не зависит от предыдущих);

ЗАПРОС

**Запрос:** серверный объект, который образуется в результате обработки сервером http-запроса, поступающего от клиента и передается серверному программному коду для обработки. Содержит: всю информацию из http-запроса: метод, коллекция заголовков, коллекция параметров, поток данных. Обычно объект Request предоставляет возможность хранить данные в формате ключ/значение.

**Структура запроса:**

1. метод;
2. URI;
3. версия протокола (HTTP/1.1);
4. заголовки (пары: имя/заголовок);
5. параметры (пары: имя/заголовок);
6. расширение.

ОТВЕТ

**Ответ:** серверныйобъект, который автоматически формируется сервером, при получении http-запроса (одновременно с объектом Request), заполняется данными серверными программным кодом, преобразуется в http-ответ и отправляется клиенту. Содержит: всю информацию, которая должна быть помещена в http-ответ: статус, коллекция заголовков, поток данных.

**Структура ответа:**

* версия протокола (HTTP/1.1);
* код состояния (1xx, 2xx, 3xx, 4xx, 5xx);
* пояснение к коду состояния;
* заголовки (пары: имя/заголовок);
* расширение.

ТИПЫ ЗАГОЛОВКОВ

* **General**: общие заголовки, используются в запросах и ответах;
* **Request**: используются только в запросах;
* **Response:** используются только в ответах;
* **Entity**: для сущности в ответах и запросах.
* **Пользовательский**

КОД СОСТОЯНИЯ

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

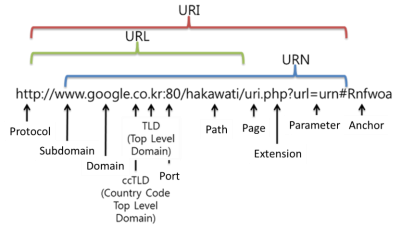
URI URL URN PURPL

**URI: Uniform Resource Identifier** – унифицированный идентификатор ресурса (документ, изображение, файл, служба, электронная почта).

**URL: Uniform Resource Location** - унифицированный локатор ресурса, содержащий местонахождение ресурса и способ обращения (протокол) к ресурса, описывает множество URI.

**URN: Uniform Resource Name** -унифицированное имя ресурса – URI, имя ресурса, не содержащее месторасположение и способ доступа к ресурсу. В будущем URN должен заменить URL (для решения проблем с перемещением ресурсов в Internet).

**URI, URL, URN** –рекомендуется использовать термин URI**.**



**PURPL**: **Persistent Uniform Resource Locator** – постоянный унифицированный локатор ресурса.

ПРОЦЕДУРА РУКОПОЖАТИЯ

**SSL/TLS Handshake** – «рукопожатие» между сервером и клиентом. Проще говоря – идентификация друг друга. Происходит во время HTTPS-соединения внутри зашифрованного туннеля SSL/TLS, который гарантирует безопасность как серверу, так и клиенту. После успешной идентификации генерируется секретный сеансовый ключ, который обеспечивает защищенную связь – он служит одновременно как для шифрования, так и для дешифрования передаваемых данных.

**Как происходит хэндшейк:**

Если представить SSL-рукопожатие как диалог между сервером и клиентом, то процесс будет выглядеть следующим образом:

1. Клиент обращается к серверу с просьбой установить безопасное соединение и предлагает набор шифров, которые «понимает», а также совместимую версию SSL/TLS.
2. Сервер проверяет присланный шифронабор, сравнивает со своим, и отсылает ответ клиенту с файлом сертификата и открытым ключом.
3. Клиент проверяет сертификат и, если всё в порядке, предлагает проверить закрытый ключ. Для этого он его генерирует и шифрует общий секретный ключ с помощью присланного ранее открытого ключа сервера.
4. Сервер принимает ключ, проверяет его своим закрытым ключом. Далее он создает главный секрет, который и будет использоваться для шифрования обмениваемой информации.

После этого клиент отправляет серверу тестовое сообщение, зашифрованное по продуманному ранее методу, а тот его расшифровывает и анализирует. На этом SSL/TLS Handshake завершается, и клиент с сервером могут спокойно обмениваться информацией дальше.

Если сеанс будет завершен, и через какое-то время клиент снова обратится к серверу, проходить заново процедуру «рукопожатия» не потребуется – все сгенерированные ранее данные и главный секрет сохранят свою актуальность. Весь этот процесс занимает считанные секунды и проходит абсолютно незаметно для пользователя.

СЕССИЯ ЖЦ КУКИС КЭШ ФИЛЬТР

**Сессия (Session):** серверный объект, хранящий информацию о соединении с клиентом, создается при первом обращении.

**ЖЦ сессии:** **timeout** (равен 10 – 30 минутам) – максимальное время между запросами клиента. Если **timeout** превышен, то Session разрушается и при следующем запросе создается новый экземпляр. Каждая сессия имеет собственный идентификатор (**Session ID**, 16 или более байт). Каждый Requestпринадлежит, какой-то сессии (имеет ссылку на объект Session или содержит Session ID). Обычно объект Session предоставляет приложению возможность хранить данные в формате ключ/значение.

**Cookie**: фрагмент данных, оправленный web-сервером и хранимый web-клиентом. Используется для аутентификации, хранения пользовательских предпочтений, статистики, **информации о сеансе** (обычно Session ID). Обычно имеет имя, содержащее URL, может иметь срок действия. Для создания и пересылки Cookie применяются заголовки.

**Контекст web-приложения:** серверный объект, предназначенный для хранения информации кодом о web-приложении, общий для всех файлов. Как правило, формируется сразу при загрузке web-сервера, основные данные (параметры приложения) копируются из конфигурационного файла приложения, общий для всех сессий приложения, обычно контекст предоставляет возможность хранить данные в формате ключ/значение.

**Кэш (Cache):** серверный объект, предназначенный для временного хранения данных с целью ускорения выполнения запроса. Кэширование – процессы записи и извлечения данных в/из Cache.

**Фильтр** — это серверный объект, который служит препроцессором запроса, предназначенный для предварительной обработки объекта Request.



**Слушатели событий (Lister):** серверные объекты – для обработки событий жизненного цикла web-приложения.

**Синхронный http-запрос:** запрос, при котором поток, выдавший http-запрос, блокируется до поступления запроса.

**Асинхронные запросы:** запрос, при котором поток, выдавший http-запрос, не блокируется до поступления запроса; для обработки ответа применяется функция обратного вызова.

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

MIDDLEWARE И КОНВЕЙЕР ОБРАБОТКИ ЗАПРОСОВ

Обработка запросов в ASP.NET Core устроена по **принципу конвейера.**

**Middleware** (промежуточное или связующее программное обеспечение) — это компонент (фрагмент кода в конвейере приложения), используемый для обработки запросов и ответов.

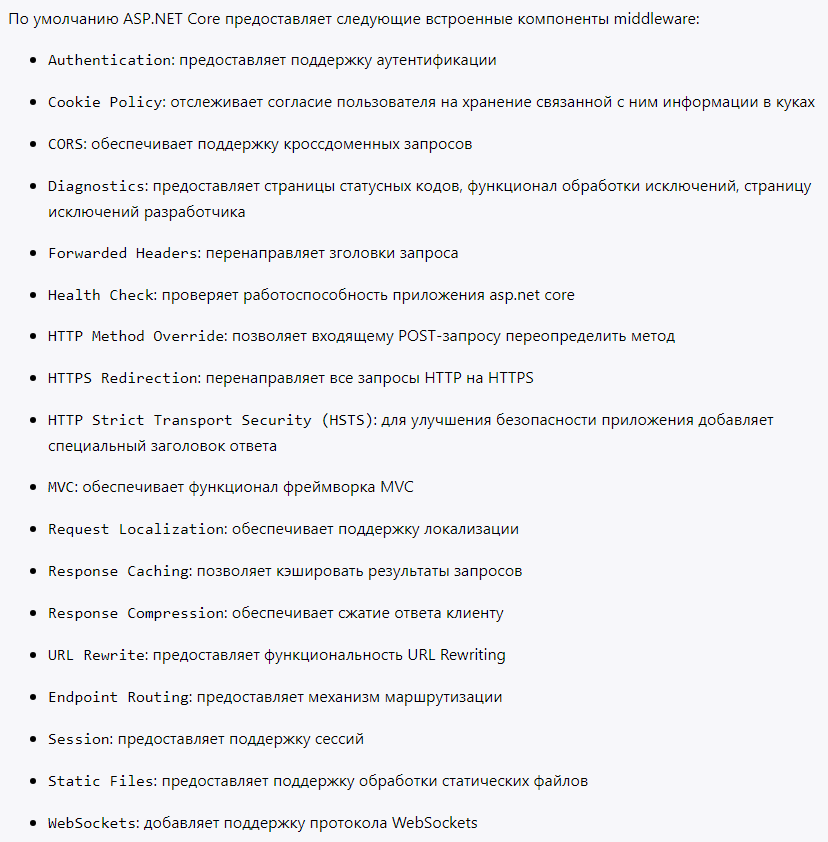
Компоненты middleware конфигурируются с помощью методов расширений Run, Map и Use объекта IApplicationBuilder, который передается в метод Configure() класса Startup.

Метод Configure выстраивает конвейер обработки запросов в ASP.NET Core приложении. Он состоит из последовательности делегатов запросов, вызываемых один за другим.

Цепочка у middleware следующая: обработать – передать следующему. Если не вызывать следующую часть, то произойдет замыкание, что позволяет избежать ненужной работы.

**app.Use()**

Этот метод используется для конфигурирования нескольких middleware. В отличие от app.Run(), мы можем включить в него параметр next, который вызывает следующий делегат запроса в конвейере. Мы также можем замкнуть (завершить) конвейер, не вызывая параметр next.



**app.Map()**

Этот метод расширения используются как условное обозначение для ветвления конвейера. Map разветвляет конвейер запросов на основе пути запроса. Если путь запроса начинается с указанного пути, ветвь выполняется.

**app.Run()**

Этот метод добавляет middleware-компонент в виде Run[Middleware], который выполнится в конце конвейера. Как правило, он действует как замыкающее middleware и добавляется в конце конвейера запросов, поскольку не может вызывать следующий middleware-компонент. Run — в соглашениях описывается так, что этим методом стоит пользоваться, только тогда, когда мы хотим добавить наш middleware в конец обработки запроса(pipeline), соответственно после Run ничего не будет вызвано.

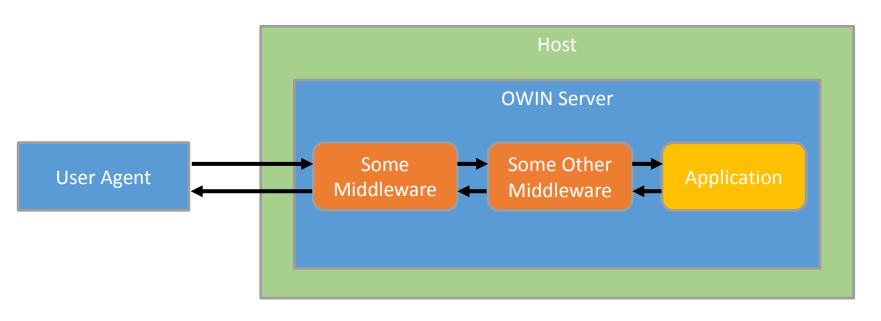
**ЖЦ:**

Метод Configure выполняется один раз при создании объекта класса Startup, и компоненты middleware создаются один раз и живут в течение всего жизненного цикла приложения. То есть для последующей обработки запросов используются одни и те же компоненты.

OWIN И KATANA

**По смелову:** OWIN – интерфейс между net-web-сервером и серверным приложением (обработчиками запросов). Основная цель OWIN отделить web-сервер от серверного приложения. Можно разрабатывать отдельно сервер и приложение.

**OWIN:** архитектура

****

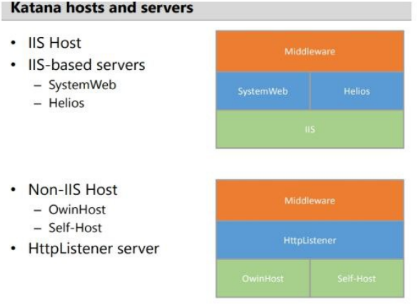
**Host** – приложение-процесс операционной системы, управляющий жизненным циклом OWIN Server.

**OWIN Server** – http-сервер, реализующий интерфейс OWIN.

**Middleware** – подключенные компоненты (модули), предназначенные для обработки запросов.

**Katana:** Microsoft-реализация OWIN сервера (говорят проект Katana). В качестве Host можно использовать IIS или self-hosting. Кроме того, позволять подключать модули middleware; предоставляет набор классов для работы с сервером и механизм подключения приложения к серверу.

**Виды конфигурации:**



**OwinMiddleware** — это базовый класс для создания middleware в Katana. Он предоставляет основные функции для работы с контекстом OWIN и управляет цепочкой вызовов между разными middleware. Конструктор класса принимает объект OwinMiddleware, который представляет следующий компонент в цепочке. Все middleware должны переопределить метод Invoke, который принимает IOwinContext. Этот метод выполняет основную логику middleware, а затем может вызывать Next.Invoke(context), чтобы передать управление следующему компоненту.

**По метаниту: OWIN или Open Web Interface for .NET** представляет собой спецификацию, определяющую взаимодействие между веб-приложением и веб-сервером.

**Katana** — это набор компонентов для создания и запуска веб-приложений на hosting abstraction, реализация спецификации owin. Katana позволяет создавать приложение, использующее различные веб-технологии, и запускать его, где угодно.

**Архитектура Katana:** приложение – промежуточное по(middleware) – сервер – хост.

ASP

БАЗА

**ASP.NET Core:** программная платформа, разработанная Microsoft и предназначается для разработки web-приложений. Кроссплатформенный, обновляется версия.

**ASP.NET Core: .NET Core = Core CLR + Core FX (BCL).**

**Core CLR**: Это среда выполнения, которая управляет выполнением приложений и предоставляет базовые функции, такие как сборка мусора и управление памятью.

**Core FX (BCL)**: Базовая библиотека классов, которая предоставляет основные функции, такие как работа с файлами, сетью и базами данных.

ASP.NET Core: .NET CLI

* .NET Command-Line Interface (CLI) — это инструмент командной строки для создания, сборки, тестирования и развертывания приложений на .NET.
* CLI позволяет разработчикам легко управлять проектами, устанавливать зависимости и выполнять приложения.

ASP.NET Core: Типы развертывания приложений

* **Portable (Framework-dependent deployment, FDD)**:
  + Приложение зависимо от установленной версии .NET на целевой машине.
  + Меньший размер приложения, так как общие библиотеки используют уже существующие на системе.
* **Standalone (Self-contained deployment, SDD)**:
  + Все необходимые библиотеки и среды выполнения включены в приложение.
  + Увеличенный размер, но позволяет запускать приложение независимо от установленной версии .NET.

Самое простое приложение в ASP.NET Core — это приложение, которое обрабатывает HTTP-запросы, например, возвращает текст или HTML-страницу.

HTTP-обработчик — это компонент, который обрабатывает входящие HTTP-запросы и генерирует HTTP-ответы.

Потоки (Threads) — это наименьшие единицы выполнения, которые могут быть запущены независимо. В ASP.NET Core управление потоками важно для обеспечения производительности и отзывчивости приложений. Работает на основе пула потоков, что позволяет эффективно использовать ресурсы сервера и уменьшает накладные расходы на создание новых потоков. ASP.NET Core использует асинхронное программирование, что позволяет обрабатывать множество запросов без блокировки потоков. Использование модификатора async и оператора await для выполнения длительных операций, таких как запросы к базе данных, не блокируя основной поток. Асинхронность позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы и при этом более экономно использовать потоки. Асинхронные методы выполняются в отдельных потоках. Слово async, которое указывается в определении метода, НЕ делает автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений await. Оператор await предваряет выполнение задачи, которая будет выполняться асинхронно.

Трассировка (Trace) — это процесс записи информации о выполнении приложения, который помогает в отладке и мониторинге.

WEBSOCKET

**WebSocket** - протокол (стандарт RFC 6455) для общения между клиентом и сервером, предоставляющий двухсторонне (дуплекс) постоянное соединение сверх протокола TCP.

Мы подключаем WS один раз, а затем сервер может отдавать нам ответы тогда, когда посчитает нужным. Он позволяет пересылать любые данные, на любой домен, безопасно и почти без лишнего сетевого трафика.

КАК РАБОТАЕТ

1. Обычный запрос TCP на сервер, ждем ответ (рукопожатие).
2. Обмен данными(один из клиентов отправил HTTP-запрос серверу и нужно отдать ответ не только одному клиенту, а целой сети, тогда сервер отдаст обычный ответ отправителю, а другим - пакеты по WebSocket-соединению с полезными данными).

КАК УСТАНАВЛИВАЕТСЯ СОЕДИНЕНИЯ

Протокол WebSocket работает над TCP.

При соединении браузер отправляет по HTTP специальные заголовки, спрашивая: «поддерживает ли сервер WebSocket?».

Если сервер в ответных заголовках отвечает «да, поддерживаю», то дальше HTTP прекращается и общение идёт на специальном протоколе WebSocket, который уже не имеет с HTTP ничего общего.

HTTP-ОБРАБОТЧИКИ

HTTP-обработчики (HTTP Handlers) используются для генерации ответа на HTTP-запросы. Они позволяют настраивать обработку запросов. HTTP-обработчик реализует интерфейс System.Web.IHttpHandler, который включает метод ProcessRequest(HttpContext context) для обработки запроса и свойство IsReusable, указывающее, может ли обработчик использоваться повторно.

ЧЕМ ОТЛИЧАЕТСЯ ОТ ОБЫЧНОГО ХТТП

HTTP-обработчик – это конечная точка, которая обслуживает запрос к ресурсу. Фактически, HTTP-обработчик содержит код, который запускается для генерации результата для данного типа запросов.

HTTP – просто запрос, он не обрабатывает входящие HTTP-запросы и не генерирует ответы. HTTP-обработчик имеет логику обработки запросов.

Свойство IsReusable позволяет определить поведение обработчика при обращении к нему. Метод ProcessRequest является входной точкой для обработки запроса. В нем содержится логика по генерации ответа клиенту.

КАК УСТАНАВЛИВАЕТСЯ СОЕДИНЕНИЕ ПО ВЕБ СОКЕТУ ПО ЗАГОЛОВКАМ

Установка соединения по веб-сокету начинается с клиента, который отправляет HTTP-запрос с заголовком Upgrade на сервер для перехода на протокол WebSocket. Сервер, поддерживающий WebSocket, отвечает статусом 101 Switching Protocols, и соединение устанавливается.

РУКОПОЖАТИЕ ЧЕРЕЗ ЗАГОЛОВКИ

Процесс установления соединения WebSocket известен как открывающее рукопожатие и состоит из обмена запросами/ответами HTTP/1.1 между клиентом и сервером. Клиент всегда инициирует рукопожатие; он отправляет запрос GET на сервер, указывая, что он хочет обновить соединение с HTTP на WebSockets. Сервер должен вернуть код ответа протокола коммутации HTTP 101 для установки соединения WebSocket. Как только это произойдет, соединение WebSocket можно будет использовать для постоянной двунаправленной полнодуплексной связи между сервером и клиентом.

ТАБЛИЦА С ЗАГОЛОВКАМИ ДЛЯ РУКОПОЖАТИЯ

|  |  |  |
| --- | --- | --- |
| Таблица 2.1: Открытие заголовков рукопожатия | | |
| ЗАГОЛОВОК | ОБЯЗАТЕ  ЛЬНОСТЬ | ОПИСАНИЕ |
| Host | Да | Имя хоста и, при необходимости, номер порта сервера, на который отправляется запрос. Если номер порта не указан, подразумевается значение по умолчанию (80 для ws или 433 для wss). |
| Connection | Да | Указывает, что клиент хочет согласовать изменение способа использования соединения. Значение должно быть Upgrade. Также возвращается сервером. |
| Upgrade | Да | Указывает, что клиент хочет обновить соединение до альтернативных средств связи. Значение должно быть websocket. Также возвращается сервером. |
| Sec-WebSocket-Version | Да | Единственное допустимое значение — 13. Любая другая версия, переданная в этом заголовке, недействительна. |
| Sec-WebSocket-Key | Да | Одноразовое случайное значение (nonce) в кодировке Base64, отправленное клиентом. Автоматически обрабатывается большинством библиотек WebSocket или с помощью класса WebSocket, предоставляемого в браузерах. Дополнительные сведения см. в разделах Sec-WebSocket-Key и Sec-WebSocket-Accept в этой главе. |
| Sec-WebSocket-Accept | Да | Хеш-значение SHA-1 в кодировке Base64, возвращаемое сервером как прямой ответ на Sec-WebSocket-Key. Указывает, что сервер готов инициировать соединение WebSocket. Дополнительные сведения см. в разделах Sec-WebSocket-Key и Sec-WebSocket-Accept в этой главе. |
| Sec-WebSocket-Protocol | Нет | Необязательное поле заголовка, содержащее список значений, указывающих, какие подпротоколы хочет использовать клиент, упорядоченные по предпочтениям. Серверу необходимо включить это поле вместе с одним из выбранных значений подпротокола (первым из списка, который он поддерживает) в ответ. Дополнительные сведения см. в разделе «Подпротоколы» далее в этой главе. |
| Sec-WebSocket-Extensions | Нет | Необязательное поле заголовка, первоначально отправляемое от клиента на сервер, а затем впоследствии отправляемое с сервера клиенту. Это помогает клиенту и серверу согласовать набор расширений уровня протокола, которые будут использоваться на протяжении всего соединения. Дополнительные сведения см. в разделе «Расширения» далее в этой главе. |
| Origin | Нет | Поле заголовка, отправляемое всеми браузерными клиентами (необязательно для небраузерных клиентов). Используется для защиты от несанкционированного использования сервера WebSocket из разных источников сценариями, использующими API WebSocket в веб-браузере. Соединение будет отклонено, если указанный Origin неприемлем для сервера. |

ЧТО ТАКОЕ ПРОТОКОЛ WEB SOCKETS API?

Интерфейс прикладного программирования (API) WebSocket, который расширяет возможности протокола WebSocket для веб-приложений. API WebSocket обеспечивает управляемую событиями связь через постоянное соединение. Это позволяет создавать веб-приложения, которые действительно работают в режиме реального времени и менее ресурсоемки как на клиенте, так и на сервере по сравнению с методами HTTP.

API WebSocket представляет собой технологию, позволяющую открыть постоянный двусторонний полнодуплексный канал связи между веб-клиентом и веб-сервером. Интерфейс WebSocket позволяет асинхронно отправлять сообщения на сервер и получать ответы на основе событий без необходимости опроса обновлений.

PING / PONG

В протокол встроена проверка связи при помощи управляющих фреймов типа PING и PONG.

Тот, кто хочет проверить соединение, отправляет фрейм PING с произвольным телом. Его получатель должен в разумное время ответить фреймом PONG с тем же телом.

Эта функциональность встроена в браузерную реализацию, так что браузер ответит на PING сервера, но управлять ей из JavaScript нельзя.

Иначе говоря, сервер всегда знает, жив ли посетитель или у него проблема с сетью.

WSS

Соединение WebSocket можно открывать как WS:// или как WSS://. Протокол WSS представляет собой WebSocket над HTTPS.

Кроме большей безопасности, у WSS есть важное преимущество перед обычным WS – большая вероятность соединения.

Дело в том, что HTTPS шифрует трафик от клиента к серверу, а HTTP – нет.

Если между клиентом и сервером есть прокси, то в случае с HTTP все WebSocket-заголовки и данные передаются через него. Прокси имеет к ним доступ, ведь они никак не шифруются, и может расценить происходящее как нарушение протокола HTTP, обрезать заголовки или оборвать передачу.

А в случае с WSS весь трафик сразу кодируется и через прокси проходит уже в закодированном виде. Поэтому заголовки гарантированно пройдут, и общая вероятность соединения через WSS выше, чем через WS.

MVC

БАЗА

**Model-View-Controller** – архитектурный паттерн; включает три компоненты: модель – данные; представление – отображение модели; контролер – обработка запросов, координация взаимодействия модели и представления. Маршрутизатор – выбор контроллера и действия.



CONTROLLER

**Контроллер (controller):** представляет центральный компонент MVC, который обеспечивает связь между пользователем и приложением, представлением и хранилищем данных. Он содержит логику обработки запроса пользователя. Контроллер получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления, наполненного данными моделей.

**Контроллер** — это компонент, который обеспечивает взаимосвязь между пользователем и приложением. Он обрабатывает входящие запросы и, либо, сразу генерирует данные для ответа (отправляет данные в **представление**), либо использует **модель** для манипуляции с данными и, затем, генерирует данные для представления.

1. **MVC Controller:** reflection-поиск контролера, признаки

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

1. **MVC Controller:** Action – действия: методы контроллера

* нестатический публичный метод котроллера;
* не является generic;
* не помечен атрибутом [NonAction];
* перегруженные методы могут быть только отмеченные атрибутами [Action], [ActionName], [AcceptVerbs], [HttpGet], [HttpPost];
* ref и out–параметры в действиях не допустимы;
* возвращает IActionResult;

1. **MVC Controller:** возвращаемые инфраструктурой значения

**IActionResult** –стандартное значение;

**Task или void** – заменяется на EmptyResult;

остальное – заменяется на ObjectResult.

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

VIEW

**Представление (view):** отвечают за визуальную часть или пользовательский интерфейс (html-страница). Может содержать логику, связанную с отображением данных. В то же время представление не должно содержать логику обработки запроса пользователя или управления данными.

**Представление (view)** — это та часть приложения, которая отвечает за отображение данных пользователю. Представление получает данные от контроллера и на основании этих данных формирует готовый ответ пользователю, например, генерирует html-страницу. Представление не должно содержать логику обработки запроса пользователя (за это отвечает контроллер) или управления данными (за это отвечает модель).

**Рендеринг**

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, в методе контроллера вызывается метод View.

Вызов метода View возвращает объект ViewResult. Затем уже ViewResult производит рендеринг определенного представления в ответ.

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

ViewData, ViewBag, TempData – механизмы передачи данных из контроллера в представление.

**ViewData**

ViewData представляет словарь из пар ключ-значение:

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

**ViewBag**

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

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

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

Layout компоновка (макет, планировка) страницы. Компоновка – это представление, предназначенное для макетирования страниц. Макетирование страниц осуществляется с помощью механизма секций. Любая страница в свою очередь тоже может служить компоновкой, т.е. допускается вложенность.

Определить там где Layout прописываем @section{ <p>НУ ХУЙ</p> }. @RenderSection(“head”) куда-нибудь в cshtml файл. В результате он будет на странице.

Обнаружение представления **ViewName**:

1)~/Views/ControllerName/ViewName

2)~/Views/Shared/ViewName

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

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

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

Валидация: [RegularExpression].

Валидация: собственный атрибут, унаследованный от ValidationAttribute.

Filter

**MVC Filters**: механизм автоматического выполнения кода в рамках контроллера; фильтры: действий, результатов, исключений, авторизации. Фильтры могут быть в форме атрибутов акции (действует для одну акцию) или атрибутов контроллера (действует для всех акций контроллера). Для однородны фильтров может быть установлен приоритет (Order, чем больше, тем приоритетней)

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

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

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

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

**Exception Filter:** обработка исключений в акциях

**MVC Filters**: общая схема выполнения фильтров. Для однородных фильтров может быть установлен приоритет (Order, чем больше, тем приоритетней). Для этого фильтр должен реализовывать IOrdersFilter (иначе Order = 0).

**Service Filters**: это механизм в ASP.NET Core, который позволяет внедрять зависимости в фильтры, используемые в контроллерах. Это особенно полезно, когда вам нужно использовать один и тот же фильтр в нескольких местах и при этом иметь доступ к сервисам из контейнера зависимостей.

Model

**Модель (model):** описывает используемые данные, а также логику этих данных. Как правило, объекты моделей хранятся в базе данных. Модель не должна содержать логику взаимодействия с пользователем и не должна определять механизм обработки запроса. Кроме того, модель не должна содержать логику отображения данных в представлении.

**Два типа моделей:**

*модели представлений* (используются представлениями для отображения и передачи данных),

*модели домена* (логика управления данными).

**Модель (model)** — это компонент, содержащий всю бизнес-логику приложения. Модель ничего не знает о контроллерах и представлениях в приложении, но знает всё о том, как манипулировать данными приложения — записывать данные в БД, удалять данные, производить сложные расчёты, анализ и т.д.

КАК ВЗАИМОДЕЙСТВУЮТ

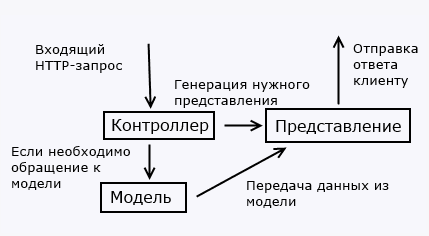
Сначала браузер отправляет запрос Контроллеру. Затем Контроллер взаимодействует с Моделью для отправки и получения данных.

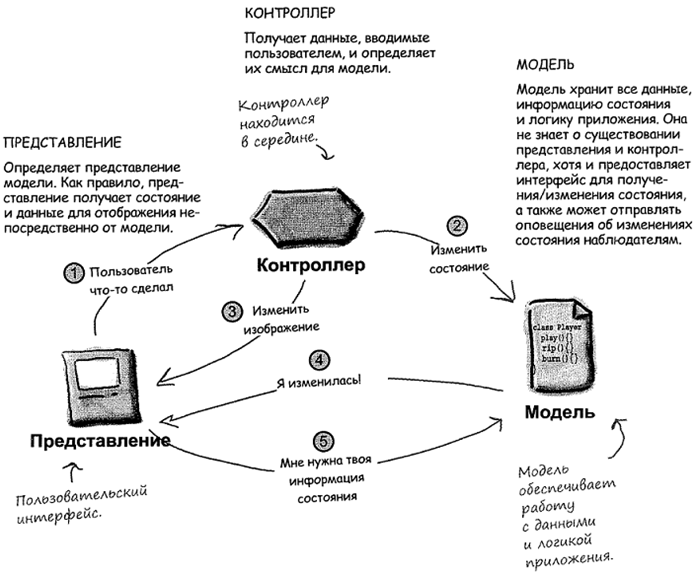
Потом Контроллер взаимодействует с Представлением для визуализации данных. Представление занимается только тем, как представить информацию, а не окончательным отображением. Отображать данные на основе того, что отправляет Контроллер, будет уже динамический HTML-файл.

Наконец, Представление отправит окончательный вид Контроллеру, а Контроллер передаст эти данные на вывод пользователям.

Важно то, что Представление и Модель никогда не взаимодействуют напрямую. Единственное взаимодействие между ними происходит через Контроллер.

Это означает, что логика приложения и интерфейс никогда не пересекаются.





ЖЦ

Фактически MVC определяется двумя жизненными циклами: жизненным циклом приложения и жизненным циклом запроса.

**Жизненный цикл приложения**, в котором процесс приложения запускает работающий сервер до момента его остановки. и он пометил два события в файле запуска вашего приложения. т. е. события запуска и завершения приложения.

Это отдельно от **жизненного цикла запроса**, который представляет собой последовательность событий или этапов, выполняемых каждый раз, когда HTTP-запрос обрабатывается приложением.

Отправной точкой для каждого приложения MVC является маршрутизация. После этого полученный запрос вычисляет и определяет, как его следует обрабатывать с помощью модуля маршрутизации URL. Модуль маршрутизации отвечает за сопоставление входящего URL с маршрутами, которые мы определили в нашем приложении.

У каждого маршрута есть связанный с ним обработчик маршрута. Если запрос соответствует маршруту, определенному в нашем приложении MVC, обработчик маршрута MVC извлекает экземпляр MVC HttpHandler.

Обработчик MVC Запускает процесс инициализации и выполнения контроллера. Платформа MVC обрабатывает преобразование данных маршрута в определенный контроллер, который может обрабатывать запросы.

Это достигается с помощью компонентов MVC, таких как фабрика контроллеров и Активаторы, которые отвечают за создание экземпляра класса Controller.

После создания контроллера компонент, называемый средством вызова действия, находит и выбирает соответствующий метод действия для вызова на нашем контроллере.

Привязка модели происходит перед вызовом метода, который сопоставляет данные из нашего HTTP-запроса с параметрами наших методов действий. Также называется фильтром действий до и после генерации результатов действий.

Теперь после того, как результат нашего действия подготовлен, запускается следующий этап - Выполнение результата.

Если результатом является тип представления, будет вызван механизм представления, который найдет представление и отобразит его.

Если не тип представления, то результат действия будет выполняться сам по себе. Выполнение этого результата есть не что иное, как генерация фактического ответа на этот исходный HTTP-запрос.

МАРШРУТИЗАЦИЯ

**Таблица маршрутов** — это структура данных, которая хранит все маршруты, определенные в приложении. Каждый маршрут может быть представлен в виде ключ-значение, где ключом является URI, а значением — соответствующий обработчик.

**Конечная точка (endpoint)** — это исполняемый кода приложения, обрабатывающий запрос.

Конечные точки определяются в приложении и настраиваются при его запуске. Процесс сопоставления конечных точек может извлекать значения из URL-адреса запроса и предоставлять эти значения для обработки запроса.

Шаблон маршрута, который сопоставляется с конечной точкой, может иметь параметры. Параметры имеют имя и определяются в шаблоне маршрута внутри фигурных скобок: {название\_параметра}

Система маршрутизации сопоставляет маршруты с запросом в том порядке, в котором они определены. Поэтому если мы определяем маршруты, которые могут пересекаться, то вначале надо определять те маршруты, которые имеют больший приоритет или более специфичные. А больше общие или стандартные маршруты должны определяться ближе к концу.

Сопоставление адреса URL или **URL matching** представляет процесс сопоставления запроса с конечной точкой. Данный процесс основывается на пути запроса и полученных в запросе заголовках. Данный процесс проходит ряд этапов:

1. Сначала выбираются все конечные точки, шаблон маршрута которых совпадает с путем запроса
2. Далее из полученного на предыдущем этапе набора конечных точек удаляются те, которые не соответствуют ограничениям маршрута
3. Затем из полученного на предыдущем этапе набора конечных точек удаляются те, которые не удовлетворяют политике объекта MatcherPolicy (вкратце: класс **MatcherPolicy** позволяет определить порядок сравнения конечных точек и адреса URL)
4. И в самом конце применяется объект **EndpointSelector** для выбора из полученного на предыдущем этапе списка конечной точки, которая в конечном счете и будет обрабатывать запрос

Приоритет конечных точек зависит от двух факторов:

* Порядок следования в наборе конечных точек
* Приоритетность шаблона маршрута

Приоритетность шаблонов маршрута зависит от специфичности шаблона. Специфичность шаблона определяется на основе следующих критериев:

* Шаблон маршрута с большим количеством сегментов более специфичен, чем шаблон меньшим количеством сегментов
* Сегмент с текстовым литералом (статический сегмент) более специфичен, чем сегмент с параметром маршрута
* Сегмент с параметром, к которому применяется ограничение маршрута, более специфичен, чем сегмент с параметром без ограничения
* Комплексный сегмент более специфичен, чем сегмент с параметром с ограничением
* Параметр catch-all (параметр, который соответствует неопределенному количеству сегментов) наименее специфичен

Если в конечном счете осталось две и более конечных точек, которые соответствуют запрошенному адресу, и соответственно система маршрутизации не может выбрать, какая из этих конечных точек должна обрабатывать маршрут, то генерируется исключение.

**UseRouting** добавляет соответствие маршрута в конвейер ПО промежуточного слоя. Это ПО промежуточного слоя обращается к набору конечных точек, определенных в приложении, и выбирает наиболее подходящее на основе запроса.

**UseEndpoints** добавляет выполнение конечной точки в конвейер ПО промежуточного слоя. Он запускает делегат, связанный с выбранной конечной точкой.

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.

В качестве параметра атрибут Route принимает шаблон URL, с которым будет сопоставляться запрошенный адрес.

Если в проекте планируется использовать только маршрутизацию на основе атрибутов, то в классе Startup мы можем не определять никаких маршрутов.

От всех параметров шаблона маршрутов в атрибутах отличаются два параметра controller и action, которые ссылаются соответственно на контроллер и его действие. При использовании их надо помещать в квадратные скобки, а не в фигурные, как другие параметры: [Route("[controller]/[action]/{id?}")].

С помощью атрибутов можно задать несколько маршрутов для одного метода.

PARTIAL VIEWS

Часто возникает необходимость в использовании одних и тех же фрагментов дескрипторов Razor и разметки HTML в разных местах приложения. Вместо дублирования содержимого можно применять **частичные представления**.

Их можно и нужно встраивать в другие обычные представления. Особенно удобно использовать частичные представления для отображения результатов AJAX-запросов, так как они динамически обновляют часть страницы без перезагрузки.

По действию похоже на обычное представление, но для него по умолчанию не определяется мастер-страница.

СПОСОБЫ ВНЕДРЕНИЯ

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

Чтобы встроить частичное представление в какое-то другое используется **хелпер HTML.Partial**.

@Html.Partial("Partial")

Html.RenderPartial - хелпер также принимает имя представления.

@{Html.RenderPartial("Partial");}

Он используется не в строчных выражениях кода Razor, а в блоке кода - то есть обрамляется фигурными скобками. Еще одно отличие между двумя способами заключается в том, что Html.RenderPartial напрямую пишет вывод в выходной поток, поэтому может работать чуть быстрее, чем Html.Partial.

Также как и в случае с обычными представлениями, мы можем создавать строго типизированные частичные представления, указав в шапке файла директиву @model.

ХЕЛПЕРЫ

HTML Helpers — это классы, которые помогают визуализировать HTML. Эти классы имеют методы, которые генерируют HTML во время выполнения. Мы также можем привязать объект модели к отдельному элементу HTML для отображения или извлечения значений.

Одно из основных различий между вызовом методов HtmlHelper и использованием тега HTML заключается в том, что методы HtmlHelper разработаны для упрощения привязки к данным представления или данным модели.

@Html используется для доступа к помощнику HTML, однако HTML является свойством HtmlHelpers, которое включено в базовый класс.

Виды: строчные, с#, tag.

Еще виды: HTML Helpers, Tag Helpers, Validation Helpers, Form Helpers, Custom Helpers.

ШАБЛОНИЗАТОР

Шаблонизатор — это программный компонент или библиотека, предназначенная для генерации текстового контента (обычно HTML) на основе шаблонов и данных. Он позволяет отделить логику представления от бизнес-логики приложения, обеспечивая механизм для динамического формирования выходных данных.

Razor, Mustache, Twig, DotLiquid, Scriban.

HTTP AUTENTICATION

**Идентификация** – заявление пользователя о себе.

**Аутентификация** – процедура проверки подлинности идентификации пользователя.

**Авторизация** - процедура проверки прав аутентифицированного пользователя.



TSL-аутентификация

Получив сертификат, клиент проверяет его на подлинность. Это чрезвычайно важный шаг. Чтобы соединение было безопасным, нужно не только зашифровать данные, нужно ещё убедиться, что они отправляются на правильный веб-сайт. Сертификаты SSL/TLS обеспечивают эту аутентификацию, а то, как они это делают, зависит от используемого шифронабора.

Все доверенные SSL-сертификаты выпускаются центром сертификации (ЦС). ЦС должен следовать строгим правилам выдачи и проверки сертификатов, чтобы ему доверяли. Вы можете считать ЦС кем-то вроде нотариуса — его подпись значит, что данные в сертификате реальны.

Во время аутентификационной части TLS-рукопожатия клиент выполняет несколько криптографически безопасных проверок с целью убедиться, что выданный сервером сертификат подлинный. Процесс включает в себя проверку цифровой подписи и того, выдан ли сертификат доверенным ЦС.

На этом этапе клиент косвенно проверяет, принадлежит ли серверу закрытый ключ, связанный с сертификатом.

В RSA, самой распространённой криптосистеме с открытым ключом, клиент с помощью открытого ключа шифрует случайные данные, которые будут использоваться для генерации сеансового ключа. Сервер сможет расшифровать и использовать эти данные, только если у него есть закрытый ключ, наличие которого обеспечивает подлинность стороны.

Если используется другая криптосистема, алгоритм может измениться, но проверка другой стороны на подлинность всё равно останется.

Обмен ключами

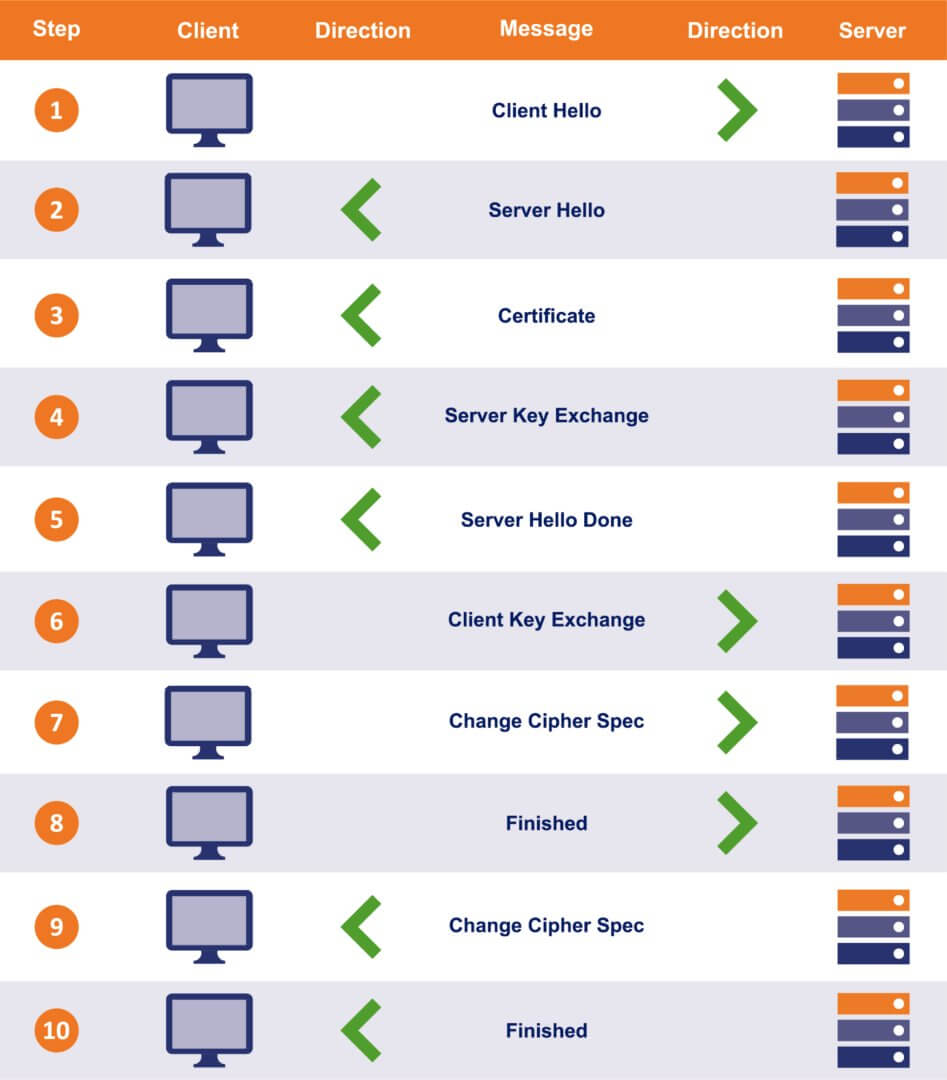
Последняя часть TLS-рукопожатия включает создание «сеансового ключа», который фактически будет использоваться для защищённой связи.

Сеансовые ключи являются «симметричными», то есть один и тот же ключ используется для шифрования и дешифрования.

Симметричное шифрование производительнее, чем асимметричное, что делает его более подходящим для отправки данных по HTTPS-соединению. Точный метод генерации ключа зависит от выбранного шифронабора, два самых распространённых из них — RSA и Диффи-Хеллман.

Чтобы завершить рукопожатие, каждая сторона сообщает другой, что она выполнила всю необходимую работу, а затем проверяет контрольные суммы, чтобы убедиться, что рукопожатие произошло без какого-либо вмешательства или повреждения.

Всё SSL-рукопожатие происходит за несколько сотен миллисекунд. Это первое, что произойдёт при HTTPS-соединении, даже до загрузки веб-страницы. После SSL-рукопожатия начинается зашифрованное и аутентифицированное HTTPS-соединение, и все данные, отправляемые и получаемые клиентом и сервером, защищены.



Сертификат X.509

Сертификат X.509 представляет собой стандартный формат для хранения и транспортировки атрибутов безопасности. Основным элементом сертификата является открытый ключ. Сертификаты выдают центры сертификации (Certificate Authority, CA), такие как Thawte и Verisign. В случае использования внутри локальной сети (intranet) можно применить Certificate Server Active Directory.

Содержимое сертификата X.509

Сертификат включает в себя следующие элементы:

* Имя держателя
* Адрес
* Серийный номер сертификата
* Даты проверки
* Открытый ключ держателя

Форматы файлов сертификатов X.509

.CER — сертификат, или набор сертификатов, закодированный по стандарту CER.

.DER — сертификат, закодированный по стандарту DER.

.PEM — сертификат, закодированный в формате DER и использующий Base64. Имеет вид: «-----BEGIN CERTIFICATE-----» ... «-----END CERTIFICATE -----»

.P7B, .P7C — формат PKCS#7, который содержит метаданные сертификатов и CRL.

P12 — PKCS#12, содержащий блок закрытого ключа и сертификат (в зашифрованном виде).

PFX - предшественник PKCS#12, тоже содержит блок.

Хранилище сертификатов

Сертификаты могут храниться в различных хранилищах, которые обеспечивают управление и доступ к ним. Это может быть как локальное хранилище на сервере, так и облачные решения, поддерживающие управление сертификатами.

Свойства: Интернет -> Сертификаты -> тык на сертификат -> Общие

TLS:

1) обеспечивает конфиденциальность;

2) обеспечивает целостность (обнаружение подмены);

3) аутентификация узлов (проверка подлинности источника сообщений);

4) последняя версия 1.2, в разработке 1.3 (значительные изменения от 1.2);

5) поверх потокового надежного соединения (для ненадежной передачи есть DTSL);

6) RFC 2246, новое название Transport Layer Security (TLS).

7) Последние обновления RFC 5246, 6176.

**Cipher Suites** – шифронаборы; в шифронабор входит: криптосистема для аутентификации сервера и сеансового секрета; шифр для защиты передаваемых данных; хеш-функция для кода аутентификации HMAC.

TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_GCM\_SHA256

ECDHE - Диффи-Хеллман на эллиптических кривых для вычисления общего секрета;

ECDSA - аутентификация данных на этапе установления соединения на основе цифровой подписи на эллиптических кривых;

AES\_128 – шифрование полезной нагрузки с помощью алгоритма AES с 128-битным ключом в режиме GCM;

SHA256 - для хеширования применяется алгоритм SHA с 256-битным ключом.

TLS: Схема работы

Клиент выдает запрос серверу (Client Hello).

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

Клиент проверяет сертификат в центре сертификации, которому доверяет.

Клиент сравнивает данные сертификата с информацией центра сертификации.

Клиент сообщает серверу, какие ключи шифрования он поддерживает.

Сервер выбирает подходящую длину ключа.

Клиент генерирует симметричный ключ, шифрует его открытым ключом.

Сервер получает симметричный ключ и расшифровывает его.

Token-аутентификация

**Token:** битовая последовательность, построенная по определенному принципу.

**Token-аутентификация:** аутентификация, использующая token для идентификации пользователя или авторизации операции пользователя; применяется, как правило, для реализации Single Sign-On в распределенных системах.

**Identity-provider** - сервер, генерирующий token.

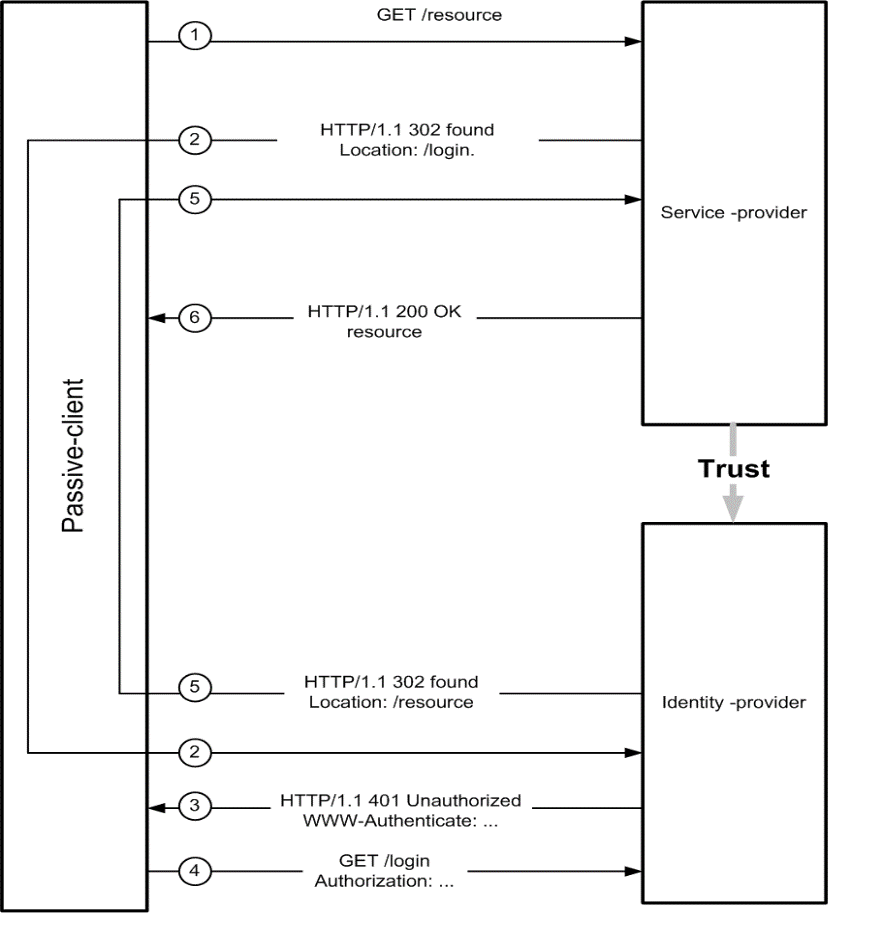
**Service-provider** – сервер, предоставляющий сервис клиенту.

**Service-provider и Identity-provider** должны иметь общий секретный ключ для шифрования/проверки token’а.Token обычно строится на базе Issuer(эмитент), Audience (аудитория), Expires On (время жизни), Claim (сведение о пользователе) & Statements (дополнительные утверждения о пользователе), HMAC (хеш).

**Активный клиент** – программный код, который может выполнять любые запросы и обрабатывать любые ответы. Обычно – это пользовательская программная реализация http-клиента.



**Пассивный клиент** – браузер.

****

OPENID

OpenID — открытый стандарт аутентификации, созданный Брэдом Фицпатриком (создателем LifeJournal) в 2006 году. Он позволяет реализовать Single Sign-On (технологию единого входа).

Существуют версии:

* OpenID 1.1
* OpenID 2.0
* OpenID Connect 1.0 (над протоколом OAuth)

Стандарт можно расширять. OpenID генерирует цифровой идентификатор для децентрализованного применения. Он использует только стандартные HTTP(S)-сообщения; cookie не требуется.

Идентификатор представлен в формате HTTP(S) URI или XRI (формат, разработанный OASIS). OpenID-провайдер — сервер, хранящий информацию об идентификаторе и подтверждающий его подлинность. Интернет-сервис — это приложение, которое хочет проверить подлинность OpenID-идентификатора.

IdentityAPI

В лекциях Смелова очень много скринов с кодом, тут их нет, просто теория.

ASP.NET Core Identity предоставляет API-интерфейсы, которые обрабатывают проверку подлинности, авторизацию и identity управление.

ORM/Entity Framework: Используется для работы с базой данных через объектно-реляционное отображение (ORM). Основные элементы:

DbContext: Контекст базы данных, который управляет сущностями и их состоянием.

Dependency Injection: Позволяет внедрять зависимости, что облегчает тестирование и поддержку кода.

**UserManager —** это конкретный класс, который управляет пользователем. Этот класс создает, обновляет и удаляет пользователей. У него есть методы поиска пользователя по идентификатору пользователя, имени пользователя и адресу электронной почты. UserManager также предоставляет функции для добавления утверждений, удаления утверждений, добавления и удаления ролей и т. д. Он также генерирует хеш паролей, проверяет пользователей и т. д.

Методы:

1. **Create**: Создание нового пользователя.
2. **Find**: Поиск пользователя по различным критериям.
3. **Delete**: Удаление пользователя.
4. **ChangePassword**: Изменение пароля пользователя.

**RoleManager**: Класс для управления ролями.

1. Методы:
2. **Create**: Создание новой роли.
3. **Delete**: Удаление роли.
4. **Find**: Поиск роли.
5. **Exists**: Проверка существования роли.