Оглавление

[Оглавление 1](#_Toc188277658)

[Основы работы с сетями в C# и .NET 4](#_Toc188277659)

[Введение в сети и протоколы 4](#_Toc188277660)

[Адреса в .NET 4](#_Toc188277661)

[IPAddress 4](#_Toc188277662)

[Встроенные адреса 5](#_Toc188277663)

[Схема адресации 5](#_Toc188277664)

[Конечная точка IPEndPoint 6](#_Toc188277665)

[Адреса Uri 7](#_Toc188277666)

[System.Uri 7](#_Toc188277667)

[UriBuilder 8](#_Toc188277668)

[DNS 9](#_Toc188277669)

[Получение информации о сетевой конфигурации и сетевом трафике 10](#_Toc188277670)

[NetworkInterface и сетевые устройства 10](#_Toc188277671)

[*Получение информации о всех подключениях* 12](#_Toc188277672)

[Мониторинг трафика 14](#_Toc188277673)

[Класс Socket 16](#_Toc188277674)

[Определение сокета 16](#_Toc188277675)

[Свойства и методы сокета 18](#_Toc188277676)

[Закрытие сокета 19](#_Toc188277677)

[Протокол HTTP. Класс HttpClient и HttpListener 21](#_Toc188277678)

[Введение в протокол HTTP 21](#_Toc188277679)

[Методы HTTP 21](#_Toc188277680)

[Статусный код HTTP 22](#_Toc188277681)

[Формат запроса и ответа 23](#_Toc188277682)

[Создание HttpClient 24](#_Toc188277683)

[Создание HttpClient 24](#_Toc188277684)

[Исчерпание сокетов 24](#_Toc188277685)

[HttpClient как статический объект/синглтон 25](#_Toc188277686)

[Создание HttpClient с помощью IHttpClientFactory 25](#_Toc188277687)

[Отправка запросов с помощью HttpClient 26](#_Toc188277688)

[Отправка запроса методом SendAsync 27](#_Toc188277689)

[Получение ответа 27](#_Toc188277690)

[GetAsync() 28](#_Toc188277691)

[GetStringAsync / GetByteArrayAsync / GetStreamAsync 29](#_Toc188277692)

[Получение данных в формате JSON 29](#_Toc188277693)

[Определение сервера 29](#_Toc188277694)

[Определение клиента 29](#_Toc188277695)

[Метод ReadFromJsonAsync класса HttpContent 30](#_Toc188277696)

[Отправка и получение заголовков 31](#_Toc188277697)

[Отправка заголовков 32](#_Toc188277698)

[Получение заголовков 34](#_Toc188277699)

[Отправка данных в запросе 35](#_Toc188277700)

[Отправка текста 36](#_Toc188277701)

[Установка заголовков в POST и PUT-запросах 37](#_Toc188277702)

[Отправка json с помощью HttpClient 37](#_Toc188277703)

[PostAsJsonAsync 38](#_Toc188277704)

[Взаимодействие HttpClient с Web API 39](#_Toc188277705)

[Создание сервера 39](#_Toc188277706)

[Определение клиента 40](#_Toc188277707)

[Отправка форм и класс FormUrlEncodedContent 42](#_Toc188277708)

[Отправка потоков и массива байтов 43](#_Toc188277709)

[Отправка потоков. StreamContent 43](#_Toc188277710)

[Отправка массива байтов. ByteArrayContent 44](#_Toc188277711)

[Отправка файлов и класс MultipartFormDataContent 45](#_Toc188277712)

[Загрузка одного файла 45](#_Toc188277713)

[Множественная отправка файлов 47](#_Toc188277714)

[Отправка смешанного содержимого 47](#_Toc188277715)

[Отправка и получение куки с HttpClient 48](#_Toc188277716)

[Cookie 48](#_Toc188277717)

[CookieContainer 49](#_Toc188277718)

[Получение куки 51](#_Toc188277719)

[Отправка кук 52](#_Toc188277720)

[HttpListener. HTTP-сервер 53](#_Toc188277721)

[Запуск HttpListener 53](#_Toc188277722)

[Контекст 53](#_Toc188277723)

[Отправка ответа 54](#_Toc188277724)

[Получение информации о запросе 55](#_Toc188277725)

[Протокол TCP 58](#_Toc188277726)

[TCP-клиент на сокетах 58](#_Toc188277727)

[Подключение к хосту 58](#_Toc188277728)

[Информация о подключении 58](#_Toc188277729)

[Отключение от хоста 59](#_Toc188277730)

[Отправка данных 59](#_Toc188277731)

[Получение данных от хоста 60](#_Toc188277732)

[Свойство Available 61](#_Toc188277733)

[Рефакторинг подключения 61](#_Toc188277734)

[TCP-сервер на сокетах 63](#_Toc188277735)

[Привязка к конечной точке. Метод Bind 63](#_Toc188277736)

[Прослушивание подключений. Метод Listen 64](#_Toc188277737)

[Подключение клиента 64](#_Toc188277738)

[NetworkStream 65](#_Toc188277739)

[Методы NetworkStream 66](#_Toc188277740)

[Отправка данных 66](#_Toc188277741)

[Получение данных 67](#_Toc188277742)

[TCP-клиент. Класс TcpClient 68](#_Toc188277743)

[Создание TcpClient 68](#_Toc188277744)

[Свойства TcpClient 68](#_Toc188277745)

[Подключение к серверу 69](#_Toc188277746)

[Закрытие TcpClient 69](#_Toc188277747)

[Отправка и получение данных 70](#_Toc188277748)

[Отправка данных 70](#_Toc188277749)

[Получение данных 70](#_Toc188277750)

[Свойства TcpClient.Available и NetworkStream.DataAvailable и считывание данных 71](#_Toc188277751)

[Отправка и получение данных в TCP. Однонаправленная связь между сокетами 71](#_Toc188277752)

[Отправка данных клиенту 72](#_Toc188277753)

[Получение данных от клиента 73](#_Toc188277754)

[Использование маркера окончания ответа 74](#_Toc188277755)

[Установка размера сообщения 75](#_Toc188277756)

[Множественная отправка и получение 76](#_Toc188277757)

[Отправка и получение данных. Однонаправленная связь между TcpListener и TcpClient 78](#_Toc188277758)

[Отправка данных клиенту 78](#_Toc188277759)

[Получение данных от клиента 79](#_Toc188277760)

[Использование буфера фиксированной длины 80](#_Toc188277761)

[Использование маркера окончания ответа 80](#_Toc188277762)

[Установка размера сообщения 81](#_Toc188277763)

[Множественная отправка и получение 82](#_Toc188277764)

[Отправка и получение данных в TCP. Двунаправленная связь 83](#_Toc188277765)

[Пример с TcpListener и TcpClient 83](#_Toc188277766)

[Пример на сокетах 85](#_Toc188277767)

[Многопоточное клиент-серверное приложение TCP 86](#_Toc188277768)

[Пример с TcpListener и TcpClient 86](#_Toc188277769)

[Многопоточое клиент-серверное приложение на сокетах 88](#_Toc188277770)

[NetworkStream и текстовые потоки 90](#_Toc188277771)

[Клиент-серверное взаимодействие 91](#_Toc188277772)

[NetworkStream и бинарные потоки 92](#_Toc188277773)

[Консольный TCP-чат 94](#_Toc188277774)

[Протокол UDP 99](#_Toc188277775)

[Использование сокетов для работы с UDP 99](#_Toc188277776)

[Прослушивание входящих сообщений 99](#_Toc188277777)

[Получение данных 99](#_Toc188277778)

[Отправка данных 100](#_Toc188277779)

[UdpClient 100](#_Toc188277780)

[Получение данных 100](#_Toc188277781)

[Отправка данных 101](#_Toc188277782)

[Метод Connect 101](#_Toc188277783)

[Консольный UDP-чат 102](#_Toc188277784)

[Консольный чат на UDP-сокетах 102](#_Toc188277785)

[Консольный чат с помощью UdpClient 103](#_Toc188277786)

[Наложение полученного сообщения на ввод сообщения 104](#_Toc188277787)

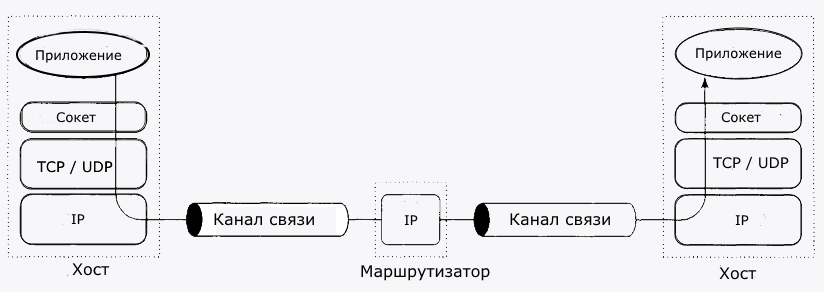
[Широковещательная рассылка 105](#_Toc188277788)

[Присоединение к группе. JoinMulticastGroup и DropMulticastGroup 105](#_Toc188277789)

[Получение и отправка сообщений в одном приложении 106](#_Toc188277790)

Основы работы с сетями в C# и .NET

## Введение в сети и протоколы



**Основная функциональность фреймворка .NET по работе с сетями содержится в пакете System.Net. Также есть дополнительные пакеты:**

* **System.Net.Http: содержит функциональность по работе с протоколом HTTP**
* **System.Net.NetworkInformation: предоставляет доступ к данным о сетевом трафике и сетевых адресах, а также к прочей информации о хостах сети. Также предоставляет функциональность ping**
* **System.Net.Security: предоставляет сетевые потоки для безопасной связи между хостами**
* **System.Net.Sockets: предоставляет доступ к функциональности сокетов операционной системы**
* **System.Net.WebSockets: предоставляет доступ к реализации инфтерфейса WebSocket**
* **System.Net.Quic: содержит типы, которые реализуют протокол QUIC в соответствии со спецификацией RFC 9000**

## Адреса в .NET

### IPAddress

**В системе классов .NET ip-адрес представлен классом IPAddress из пространства System.Net.**

***Конструкторы:***

**public IPAddress (byte[] address);**

**public IPAddress (long newAddress);**

#### Пример:

**using System.Net;**

**IPAddress ip = new IPAddress(new byte[] { 127, 0, 0, 1 });**

**// или**

**IPAddress ip = new IPAddress(0x0100007F);**

**// 127.0.0.1, где 1 - 01, 0 - 00, 0 - 00, 127 - 7F**

**// расположены наоборот**

**// или через Parse**

**IPAddress ip = IPAddress.Parse("127.0.0.1");**

**// или через TryParse**

**IPAddress.TryParse("127.0.0.1", out IPAddress? ip);**

**Console.WriteLine(ip); // 127.0.0.1**

### Встроенные адреса

**IPAddress предоставляет ряд адресов по умолчанию через ряд статических свойств:**

* **Статическое свойство Loopback: возвращает объект IPAddress для адреса 127.0.0.1**
* **Статическое свойство Any: возвращает объект IPAddress для адреса 0.0.0.0**
* **Статическое свойство Broadcast: возвращает объект IPAddress для адреса 255.255.255.255**

**IPAddress anyIp = IPAddress.Any;**

**IPAddress localIp = IPAddress.Loopback;**

**IPAddress broadcastIp = IPAddress.Broadcast;**

### Схема адресации

**Свойство класса IPAddress AddressFamily может принимать значения:**

* **AppleTalk: адрес AppleTalk**
* **Atm: адрес собственных служб ATM**
* **Banyan: адрес Banyan**
* **Ccitt: адреса протоколов CCITT, таких как протокол X25**
* **Chaos: адрес протоколов MIT CHAOS**
* **Cluster: адрес кластерных продуктов корпорации Майкрософт**
* **ControllerAreaNetwork: сетевой адрес области контроллера**
* **DataKit: адрес протоколов Datakit**
* **DataLink: адрес интерфейса прямого канала передачи данных**
* **DecNet: адрес DECnet**
* **Ecma: адрес ЕСМА (European Computer Manufacturers Association — европейская ассоциация производителей компьютеров)**
* **FireFox: адрес FireFox**
* **HyperChannel: адрес NSC Hyperchannel**
* **Ieee12844: адрес рабочей группы IEEE 12844**
* **ImpLink: адрес ARPANET IMP**
* **InterNetwork: IPv4-адрес**
* **InterNetworkV6: IPv6-адрес**
* **Ipx: IPX- или SPX-адрес**
* **Irda: IrDA-адрес**
* **Iso: адрес протоколов ISO**
* **Lat: LAT-адрес**
* **Max: MAX-адрес**
* **NetBios: адрес NetBios**
* **NetworkDesigners: адрес шлюзовых протоколов Network Designers OSI**
* **NS: адрес протоколов Xerox NS**
* **Osi: адрес протоколов OSI**
* **Packet: адрес пакета нижнего уровня**
* **Pup: адрес протоколов PUP**
* **Sna: адрес IBM SNA**
* **Unix: локальный адрес Unix для узла**
* **Unknown: семейство неизвестных адресов**
* **Unspecified: семейство неуказанных адресов**
* **VoiceView: адрес VoiceView**

#### Пример:

**IPAddress ip = new IPAddress(new byte[] { 127, 0, 0, 1 });**

**Console.WriteLine(ip.AddressFamily); // InterNetwork**

### Конечная точка IPEndPoint

**Конечная точка представляет объединение IP-адреса и порта и в .NET представляет класс IPEndPoint из пространства имен System.Net.**

**Конструкторы:**

**public IPEndPoint (long address, int port);**

**public IPEndPoint (IPAddress address, int port);**

***Пример:***

**using System.Net;**

**IPAddress ip = IPAddress.Parse("127.0.0.1");**

**IPEndPoint endpoint = new IPEndPoint(ip, 8080);**

**// или**

**IPEndPoint endPoint = IPEndPoint.Parse("127.0.0.1:8080");**

**// или**

**IPEndPoint.TryParse("127.0.0.1:8080", out IPEndPoint? endPoint);**

**Console.WriteLine(endpoint); // 127.0.0.1:8080**

**Свойства IPEndPoint предоставляют доступ к информации о конечной точке:**

* **Address: возвращает или устанавливает IP-адрес**
* **AddressFamily: возвращает схему адресации, которую применяет IP-адрес**
* **Port: возвращает или устанавливает номер порта**

**IPEndPoint endpoint = IPEndPoint.Parse("127.0.0.1:8080");**

**Console.WriteLine(endpoint.Address); // 127.0.0.1**

**Console.WriteLine(endpoint.Port); // 8080**

## Адреса Uri

**Uniform Resource Identifier (URI) - Единый идентификатор ресурса**

**Uniform Resource Locator (URL) - Единый локатор ресурсов - подвид URI**

**Общая форма адреса URL:**

**//[authority/]path[?query][#fragment]**

**https://user:password@www.somesite.com/home/index?q1=v1&q2=v2#fragmentName**

**authority включает домен и порт, а также возможные учетные данные пользователя как логин и пароль:**

**//[access\_credentials][@]host\_domain[:port]**

**Символ @ отделяет учетные данные от домена. Параметр access\_credentials, который описывает учетные данные, задается в виде**

**[user\_id][:][password]**

**Параметр user\_id предоставляет логин, а password - пароль для доступа к ресурсу. Они отделяются друг от друга двоеточием.**

**Компонент query имеет следующую форму:**

**?[parameter1=value2][(;|&)parameter2=value2]...**

**В конце после символа решетки # может указываться фрагмент - необязательная строка для идентификации компонентов внутри URL. Обычно применяется в веб-браузерах для навигации по частям веб-страницы.**

**При отправке сообщения по URI компьютер обращается к DNS (Domain Name System - система доменных имен), которая сопоставляет URI и IP.**

### System.Uri

**Uri uri = new("http://website.com/");**

**// или абсолютный адрес**

**Uri uri = new Uri("http://website.com/", UriKind.Absolute);**

**// относительный адрес**

**Uri uri = new Uri("sharp/net", UriKind.Relative);**

**// абсолютный или относительный**

**Uri uri = new Uri("http://website.com/sharp/net",**

**UriKind.RelativeOrAbsolute);**

**// или**

**Uri.TryCreate(string url,**

**UriCreationOptions options, // new UriCreationOptions()**

**out Uri? newUri);**

**Базовые свойства:**

* **AbsoluteUri: возвращает абсолютный адрес URI**

**https://user:password@www.somesite.com/home/index?q1=v1&q2=v2#fragmentName**

* **Authority: возвращает либо имя хоста в соответствии с системой доменных имен DNS, либо IP-адрес и порт сервера.**

**www.somesite.com**

* **Fragment: возвращает фрагмент адреса URI.**

**#fragmentName**

* **Host: возвращает хост.**

**www.somesite.com**

* **IsAbsoluteUri: возвращает true, если адрес абсолютный.**
* **IsDefaultPort: возвращает true, если адрес URI использует порт по умолчанию для своей схемы.**
* **IsFile: возвращает true, если адрес Uri представляет адрес файла.**
* **IsLoopback: возвращает true, если адрес Uri указывает на локальный хост.**
* **OriginalString: возвращает оригинальную строку адреса URI, которая передана в конструктор Uri.**
* **PathAndQuery: возвращает значения свойств AbsolutePath и Query, разделяя их вопросительным знаком (?).**

**/home/index?q1=v1&q2=v2**

* **Port: возвращает номер порта для текущего адреса URI.**

**443**

* **Query: возвращает строку запроса из текущего адреса URI.**

**?q1=v1&q2=v2**

* **Scheme: возвращает схему текущего адреса URI.**

**https**

* **Segments: возвращает массив сегментов пути для текущего адреса URI. Каждый сегмент представляет часть пути, которая ограничена слешами**

**["/", "home/", "index" ]**

* **UserInfo: возвращает имя и пароль пользователя.**

**user:password**

### UriBuilder

**Также для создания адреса можно использовать System.UriBuilder.**

**Свойства UriBuilder:**

* **Uri: возвращает созданный экземпляр Uri.**

**https://user:password@www.somesite.com/home/index?q1=v1&q2=v2#fragmentName**

* **Fragment: возвращает или задает фрагмент URI.**

**#fragmentName**

* **Host: возвращает или задает имя хоста или IP-адрес сервера.**

**www.somesite.com**

* **Password: возвращает или задает пароль.**

**password**

* **Path: возвращает или задает путь к ресурсу, на который ссылается URI.**

**/home/index**

* **Port: возвращает или задает номер порта URI.**

**443**

* **Query: возвращает или задает строку запроса.**

**?q1=v1&q2=v2**

* **Scheme: возвращает или задает схему URI.**

**https**

* **UserName: возвращает или задает имя пользователя.**

**user**

**public UriBuilder (string? scheme,**

**string? host,**

**int port,**

**string? path,**

**string? extraValue)**

## DNS

**Физичеcки dns - это сеть серверов, в которой размещен каталог, где доменные имена сопоставлены с IP-адресами. При отправке запроса по имени домена, например, при вводе в адресной строке браузера адреса "https://metanit.com" и отправке запроса, вначале идет запрос на один из этих dns-серверов. Однако, если текущий сервер не может разрешить доменное имя, он перенаправит его вверх по иерархии серверов имен в более общий родительский домен. Каждый сервер будет проверять данное доменное имя и искать домен в своем собственном каталоге, где сопоставляются домены и IP-адреса. Если сервер или один из подчиненных серверов смог сопоставить домен с одним из ip-адресов, то сервер просто заменяет доменное имя в запросе на IP-адрес, с которым он сопоставляется, и соответствующим образом пересылает запрос.**

**Для идентификации ip-адресов .NET предоставляет статический класс Dns, который располагается в пространстве имен System.Net.**

**Методы:**

* **GetHostAddresses (string hostNameOrAddress): запрашивает DNS-сервер и возвращает все ip-адреса для определенного имени хоста в виде массива System.Net.IPAddress[]. Данный метод имеет асинхронного двойника в виде метода GetHostAddressesAsync(string hostNameOrAddress)**
* **GetHostEntry (string hostNameOrAddress): запрашивает DNS-сервер и возвращает объект IPHostEntry для определенного имени хоста или ip-адреса. Данный метод имеет асинхронного двойника в виде метода GetHostEntryAsync(string hostNameOrAddress)**
* **GetHostName(): возвращает имя хоста для локального компьютера**

#### Пример:

**using System.Net;**

**var googleEntry = await Dns.GetHostEntryAsync("google.com");**

**// google.com**

**foreach (var ip in googleEntry.AddressList)**

**{**

**// 142.250.74.78**

**}**

**// или**

**var googleIps = await Dns.GetHostAddressesAsync("google.com");**

**foreach (var ip in googleIps)**

**{**

**// 142.250.74.78**

**}**

## Получение информации о сетевой конфигурации и сетевом трафике

### NetworkInterface и сетевые устройства

**Для получения информации о сетевых устройствах/интерфейсах на текущей машине можно использовать абстрактный класс NetworkInterface из пространства имен System.Net.NetworkInformation. Собственно класс NetworkInterface и представляет отдельный сетевой интерефейс компьютера.**

**Свойства NetworkInterface:**

* **Description: возвращает описание сетевого интерфейса**
* **Id: возвращает идентификатор сетевого адаптера**
* **Name: возвращает название сетевого адаптера**
* **NetworkInterfaceType: тип сетевого интерефейса в виде константы перечисления System.Net.NetworkInformation.NetworkInterfaceType**
* **OperationalStatus: возвращает текущий статус операций**
* **Speed: возвращает скорость сетевого адаптера в виде количества битов в секунду**

**Методы NetworkInterface:**

* **GetAllNetworkInterfaces(): возвращает массив объектов NetworkInterface, где каждый элемент представляет сетевой интерфейс на локальной машине (статический метод)**
* **GetIPProperties(): возвращает объект IPInterfaceProperties, который представляет все свойства сетевого интерфейса**
* **GetIPStatistics(): возвращает статистику для текущего сетевого интерфейса в виде объекта , который хранит статистику в свойствах:**
  + **BytesReceived: возвращает количество байтов, полученных интерфейсом.**
  + **BytesSent: возвращает количество байтов, отправленных интерфейсом.**
  + **IncomingPacketsDiscarded: возвращает количество входящих пакетов, которые были удалены.**
  + **IncomingPacketsWithErrors: возвращает количество входящих пакетов с ошибками.**
  + **IncomingUnknownProtocolPackets: возвращает количество входящих пакетов с неизвестным протоколом, которые были получены в интерфейсе.**
  + **NonUnicastPacketsReceived: возвращает количество неодноадресных пакетов, полученных интерфейсом.**
  + **NonUnicastPacketsSent: возвращает количество неодноадресных пакетов, отправленных интерфейсом.**
  + **OutgoingPacketsDiscarded: возвращает количество исходящих пакетов, которые были удалены.**
  + **OutgoingPacketsWithErrors: возвращает количество исходящих пакетов с ошибками.**
  + **OutputQueueLength: возвращает длину очереди вывода.**
  + **UnicastPacketsReceived: возвращает количество одноадресных пакетов, полученных интерфейсом.**
  + **UnicastPacketsSent: возвращает количество одноадресных пакетов, отправленных интерфейсом.**
* **GetIsNetworkAvailable(): возвращает true, если доступно какое-либо сетевое подключение (статический метод)**
* **GetPhysicalAddress(): возвращает физический адрес сетевого интерфейса**

#### Пример:

**using System.Net.NetworkInformation;**

**var adapters = NetworkInterface.GetAllNetworkInterfaces();**

**Console.WriteLine($"Обнаружено {adapters.Length} устройств");**

**foreach (NetworkInterface adapter in adapters)**

**{**

**Console.WriteLine("==================================================");**

**Console.WriteLine();**

**Console.WriteLine($"ID устройства: ------------- {adapter.Id}");**

**Console.WriteLine($"Имя устройства: ------------ {adapter.Name}");**

**Console.WriteLine($"Описание: ------------------ {adapter.Description}");**

**Console.WriteLine($"Тип интерфейса: ----{adapter.NetworkInterfaceType}");**

**Console.WriteLine($"Физический адрес:-- {adapter.GetPhysicalAddress()}");**

**Console.WriteLine($"Статус: ---------------{adapter.OperationalStatus}");**

**Console.WriteLine($"Скорость: ------------------ {adapter.Speed}");**

**IPInterfaceStatistics stats = adapter.GetIPStatistics();**

**Console.WriteLine($"Получено: ----------------- {stats.BytesReceived}");**

**Console.WriteLine($"Отправлено: --------------- {stats.BytesSent}");**

**}**

**/\***

**Output:**

**Обнаружено 6 устройств**

**=====================================================================**

**ID устройства: ------------- {D5CA0C59-7CDF-4B7C-9502-A7296BCF0158}**

**Имя устройства: ------------ vEthernet (WSL)**

**Описание: ------------------ Hyper-V Virtual Ethernet Adapter**

**Тип интерфейса: ------------ Ethernet**

**Физический адрес: ---------- 0D135B218A94**

**Статус: -------------------- Up**

**Скорость: ------------------ 10000000000**

**Получено: ----------------- 8403404**

**Отправлено: --------------- 515439337**

**=====================================================================**

**ID устройства: ------------- {E5786DCA-6A1D-15A1-B418-E19DA6C04076}**

**Имя устройства: ------------ Подключение по локальной сети\* 1**

**Описание: ------------------ Microsoft Wi-Fi Direct Virtual Adapter**

**Тип интерфейса: ------------ Wireless80211**

**Физический адрес: ---------- 0ACC270A7951**

**Статус: -------------------- Down**

**Скорость: ------------------ -1**

**Получено: ----------------- 0**

**Отправлено: --------------- 0**

**\*/**

## Получение информации о всех подключениях

**Класс IPGlobalProperties позволяет получить детальную информацию по сетевому трафику и его конфигурации.**

**Методы:**

* **GetActiveTcpConnections(): возвращает сведения о TCP-подключениях (массив TcpConnectionInformation[]). TcpConnectionInformation имеют 3 свойства:**
  + **LocalEndPoint: локальная конечная точка, через которую текущий компьютер установил TCP-подключение с удаленным хостом**
  + **RemoteEndPoint: адрес удаленного хоста, с которым установлено TCP-подключение**
  + **State: состояние TCP-подключения в виде одной из констант перечисления TcpState:**
    - **Closed: TCP-подключение закрыто**
    - **CloseWait: локальная конечная точка ТСР-подключения ожидает от локального пользователя запрос на разрыв подключения**
    - **Closing: локальная конечная точка ТСР-подключения ожидает подтверждение ранее отправленного запроса на разрыв подключения**
    - **DeleteTcb: удаляется буфер управления передачей (TCB) для ТСР-подключения**
    - **Established: TCP-подключение установлено.**
    - **FinWait1: локальная конечная точка ТСР-подключения ожидает от удаленной конечной точки запрос на разрыв подключения или подтверждение ранее отправленного запроса на разрыв подключения.**
    - **FinWait2: локальная конечная точка ТСР-подключения ожидает от удаленной конечной точки запрос на разрыв подключения.**
    - **LastAck: локальная конечная точка ТСР-подключения ожидает окончательное подтверждение ранее отправленного запроса на разрыв подключения.**
    - **Listen: локальная конечная точка ТСР-подключения прослушивает запросы на подключение**
    - **SynReceived: локальная конечная точка ТСР-подключения отправила и получила запрос на подключение, и ожидает подтверждения.**
    - **SynSent: локальная конечная точка ТСР-подключения отправила удаленной конечной точке заголовок сегмента с установленным управляющим битом синхронизации (SYN) и ожидает соответствующий запрос на подключение.**
    - **TimeWait: локальная конечная точка ТСР-подключения ожидает в течение достаточного времени, чтобы обеспечить получение удаленной точкой подтверждения ее запроса на разрыв подключения.**
    - **Unknown: неизвестное состояние ТСР-подключения .**
* **GetActiveTcpListeners(): возвращает массив адресов TCP-слушателей (массив IPEndPoint[])**
* **GetActiveUdpListeners(): возвращает массив адресов UDP-слушателей (массив IPEndPoint[]).**
* **GetIcmpV4Statistics(): возвращает статистику протокола ICMPv4 (объект IcmpV4Statistics)**
* **GetIcmpV6Statistics(): возвращает статистику протокола ICMPv6 (объект IcmpV6Statistics)**
* **GetIPv4GlobalStatistics(): возвращает статистику протокола IPv4 (объект IPGlobalStatistics)**
* **GetIPv6GlobalStatistics(): возвращает статистику протокола IPv6 (объект IPGlobalStatistics)**
* **GetIPGlobalProperties(): возвращает объект IPGlobalProperties, который предоставляет информацию по сетевой конфигурации и статистику трафика (статический метод)**
* **GetUnicastAddresses() / GetUnicastAddressesAsync(): возвращает таблицу IP-адресов одноадресной рассылки (объект UnicastIPAddressInformationCollection)**

#### Пример:

using System.Net.NetworkInformation;

var ipProps = IPGlobalProperties.GetIPGlobalProperties();

var tcpConnections = ipProps.GetActiveTcpConnections();

Console.WriteLine($"Всего {tcpConnections.Length} активных TCP-подключений");

Console.WriteLine();

foreach (var connection in tcpConnections)

{

Console.WriteLine("=============================================");

Console.WriteLine($"Локальный адрес: {connection.LocalEndPoint.Address}:{connection.LocalEndPoint.Port}");

Console.WriteLine($"Адрес удаленного хоста: {connection.RemoteEndPoint.Address}:{connection.RemoteEndPoint.Port}");

Console.WriteLine($"Состояние подключения: {connection.State}");

}

/\*

Output:

Всего 4 активных TCP-подключений

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

Локальный адрес: 192.168.0.112:52308

Адрес удаленного хоста: 13.69.68.64:443

Состояние подключения: TimeWait

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

Локальный адрес: 192.168.0.112:52310

Адрес удаленного хоста: 23.96.225.71:443

Состояние подключения: Established

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

Локальный адрес: 192.168.0.112:58623

Адрес удаленного хоста: 20.54.37.64:443

Состояние подключения: Established

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

Локальный адрес: 192.168.0.112:58631

Адрес удаленного хоста: 52.166.127.37:9354

Состояние подключения: Established

\*/

### Мониторинг трафика

Методы GetIPv4GlobalStatistics() и GetIPv6GlobalStatistics() класса IPGlobalProperties возвращают объект IPGlobalStatistics, который предоставляет доступ к статистике сетевого трафика с помощью следующих свойств:

* DefaultTtl: возвращает срок жизни (TTL) IP-пакетов.
* ForwardingEnabled: возвращает значение bool, которое указывает, разрешена ли переадресация IP-пакетов.
* NumberOfInterfaces: возвращает количество сетевых интерфейсов.
* NumberOfIPAddresses: возвращает количество IP-адресов, назначенных локальному компьютеру.
* NumberOfRoutes: возвращает количество маршрутов в таблице IP-маршрутизации.
* OutputPacketRequests: возвращает количество исходящих IP-пакетов.
* OutputPacketRoutingDiscards: возвращает количество маршрутов, удаленных из таблицы маршрутизации.
* OutputPacketsDiscarded: возвращает количество отправленных отброшенных IP-пакетов
* OutputPacketsWithNoRoute: возвращает количество IP-пакетов, для которых локальному компьютеру не удалось определить маршрут к адресу назначения.
* PacketFragmentFailures: возвращает количество IP-пакетов, которые не удалось фрагментировать.
* PacketReassembliesRequired: возвращает количество IP-пакетов, для которых требовалась восстановление.
* PacketReassemblyFailures: возвращает количество IP-пакетов, которые не были успешно восстановлены.
* PacketReassemblyTimeout: возвращает максимальное время, в течение которого должны поступить все фрагменты IP-пакета.
* PacketsFragmented: возвращает количество фрагментированных IP-пакетов.
* PacketsReassembled: возвращает количество собранных IP-пакетов.
* ReceivedPackets: возвращает количество полученных IP-пакетов.
* ReceivedPacketsDelivered: возвращает количество доставленных IP-пакетов.
* ReceivedPacketsDiscarded: возвращает количество отброшенных полученных IP-пакетов, которые были удалены.
* ReceivedPacketsForwarded: возвращает количество переадресованных IP-пакетов.
* ReceivedPacketsWithAddressErrors: возвращает количество полученных IP-пакетов с ошибками в адресе.
* ReceivedPacketsWithHeadersErrors: возвращает количество полученных IP-пакетов с ошибками в заголовке.
* ReceivedPacketsWithUnknownProtocol: возвращает количество IP-пакетов с неизвестным протоколом в заголовке, полученных локальным компьютером.

#### Пример:

using System.Net.NetworkInformation;

var ipProps = IPGlobalProperties.GetIPGlobalProperties();

var ipStats = ipProps.GetIPv4GlobalStatistics();

Console.WriteLine($"Входящие пакеты: {ipStats.ReceivedPackets}");

Console.WriteLine($"Исходящие пакеты: {ipStats.OutputPacketRequests}");

Console.WriteLine($"Отброшено входящих пакетов: {ipStats.ReceivedPacketsDiscarded}");

Console.WriteLine($"Отброшено исходящих пакетов: {ipStats.OutputPacketsDiscarded}");

Console.WriteLine($"Ошибки фрагментации: {ipStats.PacketFragmentFailures}");

Console.WriteLine($"Ошибки восстановления пакетов: {ipStats.PacketReassemblyFailures}");

/\*

Output:

Входящие пакеты: 10428590

Исходящие пакеты: 6745275

Отброшено входящих пакетов: 43737

Отброшено исходящих пакетов: 2

Ошибки фрагментации: 0

Ошибки восстановления пакетов: 0

\*/

## Класс Socket

**В основе межсетевых взаимодействий по сетевым протоколам TCP и UDP лежат сокеты. Сокет предоставляет интерфейс доступа к определенному порту определенного хоста. То есть через сокет один хост может обращаться к приложению на другом хосте. В .NET сокеты представлены классом Socket из пространства имен System.NET.Sockets, который предоставляет низкоуровневый интерфейс для приема и отправки сообщений по сети.**

### Определение сокета

**Конструкторы:**

* **Socket(AddressFamily, SocketType, ProtocolType): создает сокет, используя указанные семейство адресов, тип сокета и протокол.**
  + **AddressFamily - перечисление, которое задает схему адресации, которую может использовать сокет. Данное перечисление содержит 33 константы. Наиболее используемые:**
    - **InterNetwork: адрес по протоколу IPv4**
    - **InterNetworkV6: адрес по протоколу IPv6**
    - **Ipx: адрес IPX или SPX**
    - **NetBios: адрес NetBios**
  + **SocketType - перечисление, которое устанавливает тип сокета. Может принимать следующие значения:**
    - **Dgram: сокет будет получать и отправлять дейтаграммы по протоколу Udp. Данный тип сокета работает в связке с типом протокола - Udp и значением AddressFamily.InterNetwork**
    - **Raw: сокет имеет доступ к нижележащему протоколу транспортного уровня и может использовать для передачи сообщений такие протоколы, как ICMP и IGMP**
    - **Rdm: сокет может взаимодействовать с удаленными хостами без установки постоянного подключения. В случае, если отправленные сокетом сообщения невозможно доставить, то сокет получит об этом уведомление**
    - **Seqpacket: обеспечивает надежную двустороннюю передачу данных с установкой постоянного подключения**
    - **Stream: обеспечивает надежную двустороннюю передачу данных с установкой постоянного подключения. Для связи используется протокол TCP, поэтому этот тип сокета используется в паре с типом протокола Tcp и значением AddressFamily.InterNetwork**
    - **Unknown: адрес NetBios**
  + **ProtocolType - перечисление, которое устанавливает тип используемого протокола. Может принимать следующие значения:**
    - **Ggp**
    - **Icmp**
    - **IcmpV6**
    - **Idp**
    - **Igmp**
    - **IP**
    - **IPSecAuthenticationHeader (Заголовок IPv6 AH)**
    - **IPSecEncapsulatingSecurityPayload (Заголовок IPv6 ESP)**
    - **IPv4**
    - **IPv6**
    - **IPv6DestinationOptions (Заголовок IPv6 Destination Options)**
    - **IPv6FragmentHeader (Заголовок IPv6 Fragment)**
    - **IPv6HopByHopOptions (Заголовок IPv6 Hop by Hop Options)**
    - **IPv6NoNextHeader (Заголовок IPv6 No next)**
    - **IPv6RoutingHeader (Заголовок IPv6 Routing)**
    - **Ipx**
    - **ND**
    - **Pup**
    - **Raw**
    - **Spx**
    - **SpxII**
    - **Tcp**
    - **Udp**
    - **Unknown (неизвестный протокол)**
    - **Unspecified (неуказанный протокол)**

**Каждое значение представляет соответствующий протокол, но наиболее используемыми являются Tcp и Udp.**

* **Socket(SafeSocketHandle): создает сокет с помощью дескриптора сокета - объекта SafeSocketHandle.**
* **Socket(SocketInformation): создает сокет, используя структуру SocketInformation.**
* **Socket(SocketType, ProtocolType): создает сокет, используя указанные тип сокета и протокол.**

#### Пример:

**using System.Net.Sockets;**

**// Tcp-протокол**

**Socket socket = new Socket(AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**// Udp-протокол**

**Socket socket = new Socket(AddressFamily.InterNetwork,**

**SocketType.Dgram,**

**ProtocolType.Udp);**

### Свойства и методы сокета

**Свойства:**

* **AddressFamily: представляет схему адресации, используемую сокетом, в виде перечисления AddressFamily**
* **Available: возвращает объем данных, полученных от подключенного хоста и доступных для чтения**
* **Connected: возвращает true, если сокет подключен к удаленному хосту**
* **LocalEndPoint: возвращает локальную точку (объект типа EndPoint), по которой запущен сокет и по которой он принимает данные**
* **ProtocolType: возвращает тип протокола в виде значения перечисления ProtocolType**
* **RemoteEndPoint: возвращает адрес удаленного хоста, к которому подключен сокет (объект типа EndPoint)**
* **SocketType: возвращает тип сокета в виде значения перечисления SocketType**

**Методы:**

* **Accept() / AcceptAsync(): создает новый объект Socket для обработки входящего подключения**
* **Bind(): связывает объект Socket с локальной конечной точкой**
* **Close(): закрывает сокет**
* **Connect() / ConnectAsync: устанавливает соединение с удаленным хостом**
* **Listen(): начинает прослушивание входящих запросов**
* **Poll(): определяет состояние сокета**
* **Receive() / ReceiveAsync: получает данные**
* **ReceiveFrom() / ReceiveFromAsync(): получает данные и сохраняет конечную точку, от которой получены данные**
* **Send() / SendAsync(): отправляет данные**
* **SendTo() / SendToAsync(): отправляет данные на определенную конечную точку**
* **Shutdown(): блокирует на сокете прием и/или отправку данных. Принимает в качестве параметра значение из перечисления SocketShutdown:**
  + **Send: блокируется отправка данных**
  + **Receive: блокируется получение данных**
  + **Both: блокируются отправка и получение данных**

**В зависимости от применяемого протокола (TCP, UDP и т.д.) общий принцип работы с сокетами будет немного различаться.**

**При применении протокола, который требует установление соединения, например, TCP, сервер должен вызвать метод Bind для установки точки для прослушивания входящих подключений и затем запустить прослушивание подключений с помощью метода Listen. Далее с помощью метода Accept можно получить входящие запросы на подключение в виде объекта Socket, который используется для взаимодействия с удаленным узлом. У полученного объекта Socket вызываются методы Send и Receive соответственно для отправки и получения данных. Если необходимо подключиться к серверу, то вызывается метод Connect. Для обмена данными с сервером также применяются методы Send или Receive.**

**Если применяется протокол, для которого не требуется установление соединения, например, UDP, то после вызова метода Bind не надо вызывать метод Listen. И в этом случае для приема данных используется метод ReceiveFrom() / ReceiveFromAsync(), а для отправки данных - метод SendTo() / SendToAsync().**

### Закрытие сокета

**Socket socket = new Socket(AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**// работа с сокетом socket ...**

**socket.Close();**

**// или можно использовать конструкцию using**

**using (Socket socket = new Socket(AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp))**

**{**

**// работа с сокетом socket ...**

**}**

**При работе с протоколами, ориентированными на установку соединения, например, TCP, Microsoft рекомендует перед методом Close вызывать метод Shutdown, который блокирует прием и отправку данных для сокета и гарантирует, что все данные будут получены и отправлены перед закрытием сокета.**

#### Пример:

**Socket socket = new Socket(AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**// работа с сокетом socket ...**

**socket.Shutdown(SocketShutdown.Both);**

**}**

**catch(Exception e)**

**{**

**...**

**}**

**finally**

**{**

**socket.Close();**

**}**

Протокол HTTP. Класс HttpClient и HttpListener

## Введение в протокол HTTP

**HTTP (Hypertext Transfer Protocol) представляет протокол для запроса ресурсов с веб-сервера. В .NET основной функционал по работе с протоколом HTTP сосредоточен в простанстве имен System.Net.Http, среди основных классов в котором следует отметить класс HttpClient (класс, который позволяет отправлять запросы к веб-ресурсам) и класс HttpListener, который выполняет роль веб-сервера.**

### Методы HTTP

**Когда клиент собирается сделать запрос на сервер, данный клиент должен определить метод ответа сервера на запрос. Метод HTTP описывает действия сервера при обработке запроса. Основные из этих методов:**

* **OPTIONS: возвращает список остальных методов HTTP, которые поддерживает сервер для указанного адреса URL**
* **TRACE: служебный метод, который просто повторяет исходный запрос, полученный сервером. Полезен для идентификации любых изменений, внесенных в запрос объектами в сети, пока запрос находится в пути.**
* **CONNECT: устанавливают туннель TCP/IP между исходным хостом и удаленным хостом.**
* **GET: извлекает копию ресурса по URL-адресу, на который был отправлен HTTP-запрос.**
* **HEAD: так же как и GET, извлекает копию ресурса по URL, только ожидает получения одних заголовков без тела ответа**
* **POST: предназначен для отправки данных в теле запроса для их сохранения в виде нового ресурса на сервере**
* **PUT: предназначен для отправки данных в теле запроса для изменения уже имеющегося ресурса на сервере**
* **PATCH: предназначен для отправки данных в теле запроса для частичного изменения уже имеющегося ресурса на сервере**
* **DELETE: предназначен для удаления ресурса по указанному адресу URL**

**В .NET для представления метода HTTP в пространстве имен System.Net.Http определен класс HttpMethod. Для определения в приложении запроса определенного типа класс предоставляет ряд статических свойств, который возвращают объект HttpMethod, представляющий определенный тип запросов:**

* **HttpMethod.Delete**
* **HttpMethod.Get**
* **HttpMethod.Head**
* **HttpMethod.Options**
* **HttpMethod.Patch**
* **HttpMethod.Post**
* **HttpMethod.Put**
* **HttpMethod.Trace**

**Если используется HTTP-метод, который не покрывается этими свойствами, то можно использовать конструктор, в который передается название метода:**

**HttpMethod customMethod = new HttpMethod("CUSTOM");**

### Статусный код HTTP

**При ответе сервер устанавливает статусный код HTTP, который указывает на статус обработки запроса. По умолчанию это 3-цифровой код, где первая цифра указывает на общий характер ответа, а вторая и третья цифра конкретизируют статус. Есть пять групп статусных кодов:**

1. **1XX: информационные коды, которые указывают, что запрос получен, и его обработка продолжается**
2. **2XX: коды, которые указывают на успешную обработку запроса**
3. **3XX: коды переадресации**
4. **4XX: коды, которые указывают, что запрос содержит ошибки. То есть ошибка возникла на стороне клиента, который отправил запрос**
5. **5XX: коды, которые указывают, что в процессе обработки запроса возникла ошибка на сервере. То есть запрос от клиента корректен, а проблема на стороне сервера**

**В .NET за представление статусных кодов отвечает перечисление System.Net.HttpStatusCode, которое определяет для большинства статусных кодов соответствующие константы:**

* **Continue (100)**
* **SwitchingProtocols (101)**
* **Processing (102)**
* **EarlyHints (103)**
* **OK (200)**
* **Created (201)**
* **Accepted (202)**
* **NonAuthoritativeInformation (203)**
* **NoContent (204)**
* **ResetContent (205)**
* **PartialContent (206)**
* **MultiStatus (207)**
* **AlreadyReported (208)**
* **IMUsed (226)**
* **Ambiguous (300)**
* **MultipleChoices (300)**
* **Moved (301)**
* **MovedPermanently (301)**
* **Found (302)**
* **Redirect (302)**
* **SeeOther (303)**
* **RedirectMethod (303)**
* **NotModified (304)**
* **UseProxy (305)**
* **Unused (306)**
* **TemporaryRedirect (307)**
* **RedirectKeepVerb (307)**
* **PermanentRedirect (308)**
* **BadRequest (400)**
* **Unauthorized (401)**
* **PaymentRequired (402)**
* **Forbidden (403)**
* **NotFound (404)**
* **MethodNotAllowed (405)**
* **NotAcceptable (406)**
* **ProxyAuthenticationRequired (407)**
* **RequestTimeout (408)**
* **Conflict (409)**
* **Gone (410)**
* **LengthRequired (411)**
* **PreconditionFailed (412)**
* **RequestEntityTooLarge (413)**
* **RequestUriTooLong (414)**
* **UnsupportedMediaType (415)**
* **RequestedRangeNotSatisfiable (416)**
* **ExpectationFailed (417)**
* **MisdirectedRequest (421)**
* **UnprocessableEntity (422)**
* **Locked (423)**
* **FailedDependency (424)**
* **UpgradeRequired (426)**
* **PreconditionRequired (428)**
* **TooManyRequests (429)**
* **RequestHeaderFieldsTooLarge (431)**
* **UnavailableForLegalReasons (451)**
* **InternalServerError (500)**
* **NotImplemented (501)**
* **BadGateway (502)**
* **ServiceUnavailable (503)**
* **GatewayTimeout (504)**
* **HttpVersionNotSupported (505)**
* **VariantAlsoNegotiates (506)**
* **InsufficientStorage (507)**
* **LoopDetected (508)**
* **NotExtended (510)**
* **NetworkAuthenticationRequired (511)**

### Формат запроса и ответа

#### Пример запроса от клиента:

**GET /users/id/12 HTTP/1.1**

**// Метод, путь к ресурсу, протокол**

**Accept: application/json**

**Host: www.google.com\r\n**

**Connection: Close\r\n\r\n**

**// серия заголовков**

#### Пример ответа от сервера:

**HTTP/1.1 401 Bad Request**

**// протокол, статусный код**

**// далее идут заголовки с метаданными,**

**// которые указывают, как парсить ответ**

**// опционально тело ответа**

## Создание HttpClient

**Для отправки HTTP-запросов в .NET применяется класс HttpClient из пространства имен System.Net.Http.**

### Создание HttpClient

#### Конструкторы:

**public HttpClient (System.Net.Http.HttpMessageHandler handler);**

**HttpMessageHandler - абстрактный класс, реализующий метод Dispose().**

**public HttpClient (System.Net.Http.HttpMessageHandler handler, bool disposeHandler);**

**Если disposeHandler == true, HttpMessageHandler удаляется вместе с вызовом HttpClient.Dispose(). Если же мы хотим и после удаления HttpClient продолжать использовать объект HttpMessageHandler, то этому параметру следует передать значение false.**

**public HttpClient ();**

**// эквивалентно:**

**// HttpClient(new HttpClientHandler(), true),**

**// где HttpClientHandler - класс-наследник HttpMessageHandler**

### Исчерпание сокетов

**HttpClient нацелен на многоразовое использование в течение всей жизни приложения. Создание отдельного объекта для каждого запроса может привести к исчерпанию количества доступных сокетов и приведет к ошибкам SocketException.**

**Чтобы избежать проблемы нехватки сокетов, Microsoft рекомендует для определения HttpClient один из следующих подходов:**

* **Долговременные экземпляры HttpClient в виде статических объектов или синглтонов, которые существуют в течение всей жизни приложения**
* **Кратковременные экземпляры HttpClient, созданные с помощью фабрики IHttpClientFactory**

### HttpClient как статический объект/синглтон

#### Пример:

**class Program**

**{**

**static HttpClient client = new HttpClient();**

**static async Task Main(string[] args)**

**{**

**// использование HttpClient**

**}**

**}**

**Но в этом случае HttpClient устанавливает записи DNS только при создании подключения. Он не отслеживает срок жизни (TTL), указанный DNS-сервером для определенной dns-записи. Эта запись позволяет определить адрес хоста для создания запроса. Если эти записи DNS постоянно меняются, то клиент HttpClient не будет учитывать эти изменения.**

**Для решения этой проблемы рекомендуется устанавливать параметр SocketsHttpHandler (наследник HttpMessageHandler) со свойством PooledConnectionLifetime, чтобы при замене продключения производился поиск DNS.**

#### Пример:

**class Program**

**{**

**static HttpClient? httpClient;**

**static async Task Main(string[] args)**

**{**

**var socketsHandler = new SocketsHttpHandler**

**{**

**PooledConnectionLifetime = TimeSpan.FromMinutes(2)**

**// когда завершается интервал (2 минуты, в данном случае)**

**// текущее подключение закрывается, и открывается новое**

**};**

**httpClient = new HttpClient(socketsHandler);**

**// использование HttpClient**

**}**

**}**

### Создание HttpClient с помощью IHttpClientFactory

**Для создания объекта HttpClient у фабрики IHttpClientFactory вызывается метод CreateClient. Например, если у нас простое консольное или десктопное приложение, то нам надо добавить через nuget пакеты Microsoft.Extensions.DependencyInjection и Microsoft.Extensions.Http. Если у нас проект веб-приложения, то подобные пакеты уже по умолчанию установлены в проект.**

**Фабрика IHttpClientFactory управляет пулом объектов HttpMessageHandler. Если некоторый объект HttpMessageHandler продолжает существовать в этом пуле, то он может повторно использоваться для создания нового объекта HttpClient. Это позволяет снизить вероятность исчерпания сокетов.**

#### Пример:

**using Microsoft.Extensions.DependencyInjection;**

**// определяем коллекцию сервисов приложения**

**var services = new SeriviceCollection();**

**// используя встроенный механизм Dependency Injection**

**// внедряем сервисы, связанные с HttpClient, в том числе IHttpClientFactory**

**services.AddHttpClient();**

**// создаем провайдер сервисов**

**var serviceProvider = services.BuildServiceProvider();**

**// получаем сервис IHttpClientFactory**

**var httpClientFactory = serviceProvider.GetService<IHttpClientFactory>();**

**// создаем объект HttpClient**

**var httpClient = httpClientFactory?.CreateClient();**

**// использование HttpClient**

## Отправка запросов с помощью HttpClient

**Методы HttpClient для выполнения запросов:**

* **Send() / SendAsync(): отправляет запрос в виде объекта HttpRequestMessage и получает ответ сервера в виде объекта HttpResponseMessage**
* **GetAsync(): отправляет GET-запрос на указанный адрес и получает ответ сервера в виде объекта HttpResponseMessage**
* **GetByteArrayAsync(): отправляет GET-запрос на указанный адрес и возвращает ответ в виде массива байтов (byte[])**
* **GetStreamAsync(): отправляет GET-запрос на указанный адрес и возвращает ответ в виде объекта Stream**
* **GetStringAsync(): отправляет GET-запрос на указанный адрес и возвращает ответ в виде строки**
* **PostAsync(): отправляет POST-запрос на указанный адрес и получает ответ сервера в виде объекта HttpResponseMessage**
* **PutAsync(): отправляет PUT-запрос на указанный адрес и получает ответ сервера в виде объекта HttpResponseMessage**
* **PatchtAsync(): отправляет Patch-запрос на указанный адрес и получает ответ сервера в виде объекта HttpResponseMessage**
* **DeleteAsync(): отправляет DELETE-запрос на указанный адрес и получает ответ сервера в виде объекта HttpResponseMessage**

**С помощью ряда свойств можно настроить базовые настройки отправки запросов:**

* **BaseAddress: возвращает или устанавливает базовый адрес в виде объекта URI, который будет использоваться при отправке запросов.**
* **DefaultProxy: возвращает или устанавливает настройки прокси.**
* **DefaultRequestHeaders: возвращает или устанавливает коллекцию заголовков по умолчанию, которые отправляются на сервер при каждом запросе**
* **DefaultRequestVersion: возвращает или устанавливает версию протокола HTTP по умолчанию, которая применяется при отправке запросов**
* **DefaultVersionPolicy: возвращает или устанавливает политику версии по умолчанию**
* **MaxResponseContentBufferSize: возвращает или устанавливает максимальный размер буфера в байтах при считывании данных ответа**
* **Timeout: возвращает или устанавливает тайм-аут перед отправкой запроса.**

### Отправка запроса методом SendAsync

**public Task<System.Net.Http.HttpResponseMessage> SendAsync (System.Net.Http.HttpRequestMessage request);**

**Класс HttpRequestMessage позволяет настроить данные запроса с помощью следующих свойств:**

* **Content: возвращает или устанавливает содержимое запроса**
* **Method: возвращает или устанавливает метод запроса**
* **RequestUri: возвращает или устанавливает адрес запроса в виде объекта Uri**
* **Version: возвращает или устанавливает версию протокола HTTP**

**Установить метод и адрес запроса также можно с помощью одного из конструкторов класса.**

#### Пример:

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// определяем данные запроса**

**using HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get, "https://www.google.com");**

**// выполняем запрос**

**await httpClient.SendAsync(request);**

**}**

### Получение ответа

**После выполнения запроса метод SendAsync возвращает ответ в виде объекта HttpResponseMessage.**

**Свойства HttpRequestMessage:**

* **Content: возвращает или устанавливает содержимое HTTP-ответа. Это свойство представляет объект HttpContent, который позволяет считать данные запроса с помощью одного из методов:**
  + **ReadAsStringAsync(): возвращает ответ в виде строки**
  + **ReadAsByteArrayAsync(): возвращает ответ в виде массива байт**
  + **ReadAsStreemAsync(): возвращает ответ в виде потока - объекта Stream**
* **Headers: возвращает коллекцию заголовков HTTP-ответа (объект HttpResponseHeaders). Каждый заголовок в этой коллекции представлен парой ключ-значения. Для получения конкретного заголовка можно использовать метод GetValues(), в который передается название заголовка:**

**response.Headers.GetValues("Accept")**

* **IsSuccessStatusCode: возвращает true, если запрос HTTP прошел успешно**
* **ReasonPhrase: возвращает сообщение статуса**
* **RequestMessage: возвращает данные запроса, которые связаны с этим ответом**
* **StatusCode: возвращает статусный код ответа**
* **TrailingHeaders: возвращает дополнительные заголовки, включенные в запрос HTTP**
* **Version: возвращает версию использованного протокола HTTP**

#### Пример:

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// определяем данные запроса**

**using HttpRequestMessage request = new HttpRequestMessage(**

**HttpMethod.Get,**

**"https:www.google.com");**

**// получаем ответ**

**using HttpResponseMessage response = await httpClient.SendAsync(request);**

**// просматриваем данные ответа:**

**// статус**

**var status = response.StatusCode;**

**// заголовки**

**foreach (var header in response.Headers)**

**{**

**var headerKey = header.Key;**

**foreach (var headerValue in header.Value) ...**

**}**

**// содержимое ответа**

**string content = await response.Content.ReadAsStringAsync();**

**}**

### GetAsync()

**Аналогично GetAsync()/PostAsync()/PatchAsync()/DeleteAsync()**

#### Пример:

**// получаем ответ**

**using HttpResponseMessage response**

**= await httpClient.GetAsync("https://www.google.com");**

**// содержимое ответа в виде html-кода веб-страницы**

**string content = await response.Content.ReadAsStringAsync();**

### GetStringAsync / GetByteArrayAsync / GetStreamAsync

**Когда нам надо получить строку, то гораздо проще выполнить запрос методом GetStringAsync():**

**string content = await httpClient.GetStringAsync("https://www.google.com");**

**Если сервер отправляет двоичные данные, то их можно получить с помощью метода GetByteArrayAsync():**

**byte[] buffer = await httpClient.GetByteArrayAsync("https://www.google.com");**

**И также можно получить поток методом GetStreamAsync():**

**using Stream stream = await httpClient.GetStreamAsync("https://www.google.com");**

**StreamReader reader = new StreamReader(stream);**

**string content = await reader.ReadToEndAsync();// считываем поток в строку**

## Получение данных в формате JSON

### Определение сервера

#### Создаем проект ASP.NET. Пишем код для сервера:

**record Person(string Name, int age);**

**//для создания веб-приложения надо создать объект WebApplicationBuilder**

**WebApplicationBuilder builder = WebApplication.CreateBuilder();**

**// создаем объект WebApplication, который представляет приложение ASP.NET**

**WebApplication app = builder.Build();**

**// определяем конечную точку - обработчик запросов по определенному маршруту**

**app.MapGet("/", () => new Person("Tom", 38));**

**// в данном случае при обращении по адресу '/' (т.е. к корню веб-приложения)**

**// обработчик возвращает объект Person, который при отправке автоматически**

**// сериализуется в формат JSON**

**// запускаем приложение**

**app.Run();**

**В браузере по определенному адресу (например, https://localhost:7094) отобразится отправленный объект в формате JSON:**

**{"name":"Tom","age":38}**

### Определение клиента

**В пространстве имен System.Net.Http.Json определены методы расширения для HttpClient специально для работы с Json:**

* **GetFromJsonAsync(): отправляет запрос GET и возвращает десериализованные объекты из JSON**
* **DeleteFromJsonAsync(): отправляет запрос DELETE и возвращает десериализованные объекты из JSON**

#### Код для клиента:

**using System.Net.Http.Json;**

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// нетипизированная версия метода GetFromJsonAsync**

**object? data = await httpClient**

**.GetFromJsonAsync("http://localhost:7094/",**

**typeof(Person));**

**if (data is Person person) ... // person.name, person.age**

**// или типизированная версия**

**Person? person = await httpClient**

**.GetFromJsonAsync<Person>("http://localhost:7094/");**

**}**

**}**

### Метод ReadFromJsonAsync класса HttpContent

#### Изменим код для сервера:

**record Person(string Name, int age);**

**//для создания веб-приложения надо создать объект WebApplicationBuilder**

**WebApplicationBuilder builder = WebApplication.CreateBuilder();**

**// создаем объект WebApplication, который представляет приложение ASP.NET**

**WebApplication app = builder.Build();**

**// определяем конечную точку - обработчик запросов по определенному маршруту**

**app.MapGet("/{id?}", (int? id) =>**

**{**

**if (id is null)**

**return Results.BadRequest( // ошибка 400**

**new { Message = "Некорректные данные в запросе" });**

**else if (id != 1) // правильный id в нашем случае = 1**

**return Results.NotFound( // ошибка 404**

**new { Message = $"Объект с id={id} не существует" });**

**else**

**return Results.Json(new Person("Bob", 42));**

**});**

**// Теперь метод app.MapGet прикреплен к запросам, которые соответствуют**

**// шаблону "/{id?}". В данном случае пользователь после слеша может**

**// указать некоторое значение, и это значение будет передано в параметр**

**// id. Вопросительный знак ? после названия параметра id указывает,**

**// что этот параметр необязательный. То есть шаблон "/{id?}" будет**

**// соответствовать запросам "/" (в этом случае id равен null) или**

**// "/5" (в этом случае id равен 5). Причем если после слеша указано**

**// значение, оно должно представлять число.**

**// запускаем приложение**

**app.Run();**

#### Код для клиента:

**using System.Net;**

**using System.Net.Http.Json;**

**// для успешного ответа**

**record Person(string Name, int Age);**

**// для ошибок**

**record MyError(string Message);**

**class Program**

**{**

**static async Task Main()**

**{**

**using HttpMessageResponse response =**

**await httpClient.GetAsync("https://localhost:7094/1");**

**if (response.StatusCode == HttpStatusCode.BadRequest**

**|| response.StatusCode == HttpStatusCode.NotFound)**

**{**

**// получаем информацию об ошибке**

**MyError? error = await**

**response.Content.ReadFromJsonAsync<MyError>();**

**// response.StatusCode и error?.Message**

**}**

**else**

**{**

**// если запрос завершился успешного, получаем объект Person**

**Person? person = await**

**response.Content.ReadFromJsonAsync<Person>();**

**// person?.Name и person?.Age**

**}**

**}**

**}**

## Отправка и получение заголовков

**Тип System.Net.Http.Headers предоставляет рад методов для управления заголовками:**

* **Add(String headerName, IEnumerable<String> values): добавляет указанный заголовок и его значения в коллекцию HttpHeaders**
* **Add(String headerName, String headerValue): добавляет указанный заголовок и его значение в коллекцию HttpHeaders**
* **Clear(): удаляет все заголовки из коллекции HttpHeaders**
* **Contains(String headerName): возвращает true, если определенный заголовок присутствует в коллекции HttpHeaders**
* **Remove(String headerName): удаляет указанный заголовок из коллекции HttpHeaders**
* **TryGetValues(String headerName, IEnumerable<String> values): возвращает коллекцию значений для указанного заголовка, если он имеется в коллекции HttpHeaders**

**Этот класс абстрактный и реализуется классами HttpContentHeaders, HttpRequestHeaders и HttpResponseHeaders.**

### Отправка заголовков

##### Глобальная установка заголовков для HttpClient

#### Пример ASP.NET приложежния со стороны сервера:

**WebApplicationBuilder builder = WebApplication.CreateBuilder();**

**WebApplication app = builder.Build();**

**app.MagGet("/", (HttpContext context) =>**

**{**

**// пытаемся получить заголовок "User-Agent"**

**context.Request.Headers.TryGetValue("User-Agent", out var userAgent);**

**// пытаемся получить заголовок "SecretCode"**

**context.Request.Headers.TryGetValue("SecretCode", out var secreteCode);**

**// отправляем данные обратно клиенту**

**return $"User-Agent: {userAgent} SecretCode: {secreteCode}";**

**});**

**app.Run();**

#### Код со стороны клиента:

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// адрес сервера**

**string serverAddress = "https://localhost:7094/";**

**// устанавливаем оба заголовка по умолчанию**

**httpClient.DefaultRequestHeaders**

**.Add("User-Agent", "Mozilla Firefox 5.4");**

**httpClient.DefaultRequestHeaders**

**.Add("SecretCode", "Hello");**

**using var response = await httpClient.GetAsync(serverAddress);**

**var responseText = await response.Content.ReadAsStringAsync();**

**}**

**}**

**Для упрощения установки отдельных заголовков класс HttpRequestHeaders предоставляет ряд специальных свойств:**

* **Accept: возвращает значение заголовка Accept**
* **AcceptCharset: возвращает значение заголовка Accept-Charset**
* **AcceptEncoding: возвращает значение заголовка Accept-Encoding**
* **AcceptLanguage: возвращает значение заголовка Accept-Language**
* **Authorization: возвращает или задает значение заголовка Authorization**
* **CacheControl: возвращает или задает значение заголовка Cache-Control**
* **Connection: возвращает значение заголовка Connection**
* **Date: возвращает или задает значение заголовка Date**
* **Expect: возвращает значение заголовка Expect**
* **From: возвращает или задает значение заголовка From**
* **Host: возвращает или задает значение заголовка Host**
* **IfMatch: возвращает значение заголовка If-Match**
* **IfModifiedSince: возвращает или задает значение заголовка If-Modified-Since**
* **IfNoneMatch: возвращает значение заголовка If-None-Match**
* **IfRange: возвращает или задает значение заголовка If-Range**
* **IfUnmodifiedSince: возвращает или задает значение заголовка If-Unmodified-Since**
* **MaxForwards: возвращает или задает значение заголовка Max-Forwards**
* **Pragma: возвращает значение заголовка Pragma**
* **ProxyAuthorization: возвращает или задает значение заголовка Proxy-Authorization**
* **Range: возвращает или задает значение заголовка Range**
* **Referrer: возвращает или задает значение заголовка Referrer**
* **TE: возвращает значение заголовка TE**
* **Trailer: возвращает значение заголовка Trailer**
* **TransferEncoding: возвращает значение заголовка Transfer-Encoding**
* **Upgrade: возвращает значение заголовка Upgrade**
* **UserAgent: возвращает значение заголовка User-Agent**
* **Via: возвращает значение заголовка Via**
* **Warning: возвращает значение заголовка Warning**

#### Например:

**httpClient**

**.DefaultRequestHeaders**

**.UserAgent**

**.Add(new ProductInfoHeaderValue("User-Agent", "Mozilla Firefox 5.4");**

##### Установка заголовка для определенного запроса

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// адрес сервера**

**string serverAddress = "https://localhost:7094/";**

**using var request =**

**new HttpRequestMessage(HttpMethod.Get, serverAddress);**

**// устанавливаем оба заголовка**

**request.Headers.Add("User-Agent", "Mozilla Firefox 5.6");**

**request.Headers.Add("SecretCode", "Hello");**

**using var response = await httpClient.SendAsync(request);**

**var responseMessage = await response.Content.ReadAsStringAsync();**

**}**

**}**

### Получение заголовков

**Для получения заголовков, которые установил сервер, можно использовать свойство Headers класса HttpResponseMessage, которое представляет тип HttpResponseHeaders.**

#### Например, получим заголовок "Date":

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static string serverAddress = "https://localhost:7094/";**

**static async Task Main()**

**{**

**using var response = await httpClient.GetAsync(serverAddress);**

**var dateValues = response.Headers.GetValues("Date");**

**// или response.Headers.TryGetValues("date", out var dateValues);**

**// dateValues?.FirstOrDefault()**

**}**

**}**

**Для упрощения получения распространенных заголовков класс HttpResponseHeaders предоставляет ряд специальных методов:**

* **AcceptRanges: возвращает значение заголовка Accept-Ranges**
* **Age: возвращает или задает значение заголовка Age**
* **CacheControl: возвращает или задает значение заголовка Cache-Control**
* **Connection: возвращает значение заголовка Connection**
* **Date: возвращает или задает значение заголовка Date**
* **ETag: возвращает или задает значение заголовка ETag**
* **Location: возвращает или задает значение заголовка Location**
* **Pragma: возвращает значение заголовка Pragma**
* **ProxyAuthenticate: возвращает значение заголовка Proxy-Authenticate**
* **RetryAfter: возвращает или задает значение заголовка Retry-After**
* **Server: возвращает значение заголовка Server**
* **Trailer: возвращает значение заголовка Trailer**
* **TransferEncoding: возвращает значение заголовка Transfer-Encoding**
* **Upgrade: возвращает значение заголовка Upgrade**
* **Vary: возвращает значение заголовка Vary**
* **Via: возвращает значение заголовка Via**
* **Warning: возвращает значение заголовка Warning**
* **WwwAuthenticate: возвращает значение заголовка WWW-Authenticate для HTTP-ответа**

## Отправка данных в запросе

**Отправляемые на сервер данные представляют объект абстрактного класса System.Net.Http.HttpContent. В .NET есть следующие встроенные классы, унаследованные от HttpContent:**

* **System.Net.Http.ByteArrayContent: для отправки массива байтов**
* **System.Net.Http.FormUrlEncodedContent: для отправки данных, закодированных с помощью типа MIME "application/x-www-form-urlencoded". (унаследован от ByteArrayContent)**
* **System.Net.Http.StringContent: для отправки текста (унаследован от ByteArrayContent)**
* **System.Net.Http.Json.JsonContent: для отправки json**
* **System.Net.Http.MultipartContent: для отправки файлов**
* **System.Net.Http.MultipartFormDataContent: для отправки файлов (унаследован от MultipartContent)**
* **System.Net.Http.ReadOnlyMemoryContent: в качестве отправляемого содержимого используется ReadOnlyMemory<T>**
* **System.Net.Http.StreamContent: отправляет поток Stream**

**Обычно данные отправляются с помощью запросов типа Post/Put/Patch. У HttpClient для отправки данных можно применять метод SendAsync() - в качестве параметра он принимает объект HttpRequestMessage, у которого через свойство Content можно установить отправляемое содержимое.**

**HttpClient также предоставляет методы для каждого типа запроса:**

**PostAsync()**

**PutAsync()**

**PatchAsync()**

**Все эти методы принимают как минимум два параметра: адрес ресурса и отправляемые данные в виде объекта HttpContent?:**

**Task<HttpResponseMessage> PostAsync (string? requestUri, System.Net.Http.HttpContent? content);**

**Task<HttpResponseMessage> PutAsync (string? requestUri, System.Net.Http.HttpContent? content);**

**Task<HttpResponseMessage> PatchAsync (string? requestUri, System.Net.Http.HttpContent? content);**

### Отправка текста

##### Пример

**Определим сервер - приложение ASP.NET Core.Для этого создадим проект по типу ASP.NET Core Empty.**

#### В классе Program определим код:

**var builder = WebApplication.CreateBuilder();**

**var app = builder.Build();**

**// определяем конечную точку, которая будет обрабатывать POST-запросы по пути // "/data".**

**// Обработчик этой конечной точки в качестве параметра принимает контекст**

**// запроса - HttpContext**

**app.MapPost("/data", async (HttpContext httpContext) =>**

**{**

**using StreamReader reader = new StreamReader(httpContext.Request.Body);**

**string name = await reader.ReadToEndAsync();**

**return $"Получены данные: {name}";**

**});**

**app.Run();**

#### На стороне клиента:

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**StringContent content = new StringContent("Tom");**

**// определяем данные запроса**

**using var request = new HttpRequestMessage(HttpMethod.Post, "https://localhost:7094/data");**

**// установка отправляемого содержимого**

**request.Content = content;**

**// отправляем запрос**

**using var response = await httpClient.SendAsync(request);**

**// получаем ответ**

**string responseText = await response.Content.ReadAsStringAsync();**

**Console.WriteLine(responseText);**

**}**

**}**

#### или с методом PostAsync():

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**StringContent content = new StringContent("Tom");**

**using var response = await httpClient.PostAsync("https://localhost:7094/data", content);**

**string responseText = await response.Content.ReadAsStringAsync();**

**Console.WriteLine(responseText);**

**}**

**}**

### Установка заголовков в POST и PUT-запросах

#### Пример:

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**HttpContent content = new StringContent("Hello METANIT.COM");**

**// устанавливаем заголовок**

**content.Headers.Add("SecreteCode", "Anything");**

**using var response = await httpClient.PostAsync("https://localhost:7094/", content);**

**string responseText = await response.Content.ReadAsStringAsync();**

**Console.WriteLine(responseText);**

**}**

**}**

## Отправка json с помощью HttpClient

**Для отправки данных в формате JSON с помощью HttpClient проще использовать класс JsonContent, унаследованный от HttpContent. Для его создания нет публичного конструктора, но есть статический метод Create():**

**JsonContent content = JsonContent.Create(отправляемый\_объект);**

**отправляемый\_объект - это JSON-объект или объект С#, который будет автоматически сериализоваться в JSON.**

#### На стороне сервера ASP.NET Core:

**var builder = WebApplication.CreateBuilder();**

**var app = builder.Build();**

**app.MapPost("/create", (Person person) =>**

**{**

**// устанавливает id у объекта Person**

**person.Id = Guid.NewGuid().ToString();**

**// отправляем обратно объект Person**

**return person;**

**});**

**app.Run();**

**class Person**

**{**

**public string Id { get; set; } = "";**

**public string Name { get; set; } = "";**

**public int Age { get; set; }**

**}**

#### На стороне клиента:

**using System.Net.Http.Json;**

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// отправляемый объект**

**Person tom = new Person { Name = "Tom", Age = 38 };**

**// создаем JsonContent**

**JsonContent content = JsonContent.Create(tom);**

**// отправляем запрос**

**using var response = await httpClient**

**.PostAsync("https://localhost:7094/create", content);**

**Person? person = await response.Content.ReadFromJsonAsync<Person>();**

**Console.WriteLine($"{person?.Id} - {person?.Name}");**

**}**

**}**

**class Person**

**{**

**public string Id { get; set; } = "";**

**public string Name { get; set; } = "";**

**public int Age { get; set; }**

**}**

### PostAsJsonAsync

**Для упрощения отправки JSON для типа HttpClient в пространстве имен System.Net.Http.Json определены специальные методы расщирения:**

* **Task<HttpResponseMessage> PostAsync<T> (string? requestUri, T content);**
* **Task<HttpResponseMessage> PutAsync<T> (string? requestUri, T content);**
* **Task<HttpResponseMessage> PatchAsync<T> (string? requestUri,**

**T content);**

#### Пример:

**using System.Net.Http.Json;**

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// отправляемый объект**

**Person tom = new Person { Name = "Tom", Age = 38 };**

**// отправляем запрос**

**using var response = await httpClient**

**.PostAsJsonAsync("https://localhost:7094/create", tom);**

**// получаем ответ**

**Person? person = await response.Content.ReadFromJsonAsync<Person>();**

**Console.WriteLine($"{person?.Id} - {person?.Name}");**

**}**

**}**

**class Person**

**{**

**public string Id { get; set; } = "";**

**public string Name { get; set; } = "";**

**public int Age { get; set; }**

**}**

## Взаимодействие HttpClient с Web API

### Создание сервера

**Создадим проект по типу ASP.NET Core Empty.**

#### В файле Program.cs определим код:

**int id = 1; // для генерации id объектов**

**// начальные данные**

**List<Person> users = new List<Person>**

**{**

**new() { Id = id++, Name = "Tom", Age = 37 },**

**new() { Id = id++, Name = "Bob", Age = 41 },**

**new() { Id = id++, Name = "Sam", Age = 24 }**

**};**

**var builder = WebApplication.CreateBuilder();**

**var app = builder.Build();**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// ВОЗВРАЩАЕМ СПИСОК ОБЪЕКТОВ**

**app.MapGet("/api/users", () => users);**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// ВОЗВРАЩАЕМ ОБЪЕКТ**

**app.MapGet("/api/users/{id}", (int id) =>**

**{**

**// получаем пользователя по id**

**Person? user = users.FirstOrDefault(user => user.Id == id);**

**// если не найден, отправляем статусный код и сообщение об ошибке**

**if (user is null) return Results**

**.NotFound(new { message = "Пользователь не найден" });**

**// если пользователь найден, отправляем его**

**return Results.Json(user);**

**});**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// ДОБАВЛЯЕМ ОБЪЕКТ**

**app.MapPost("/api/users", (Person user) =>**

**{**

**// устанавливаем id для нового пользователя**

**user.Id = id++;**

**// добавляем пользователя в список**

**users.Add(user);**

**return user;**

**});**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// ИЗМЕНЯЕМ ОБЪЕКТ**

**app.MapPut("/api/users", (Person userData) =>**

**{**

**// получаем пользователя по id**

**var user = users.FirstOrDefault(u => u.Id == userData.Id);**

**// если не найден, отправляем статусный код и сообщение об ошибке**

**if (user is null) return Results**

**.NotFound(new { message = "Пользователь не найден" });**

**// если пользователь найден, изменяем его данные и отправляем обратно //клиенту**

**user.Age = userData.Age;**

**user.Name = userData.Name;**

**return Results.Json(user);**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// УДАЛЯЕМ ОБЪЕКТ**

**app.MapDelete("/api/users/{id}", (int id) =>**

**{**

**// получаем пользователя по id**

**Person? user = users.FirstOrDefault(u => u.Id == id);**

**// если не найден, отправляем статусный код и сообщение об ошибке**

**if (user is null) return Results**

**.NotFound(new { message = "Пользователь не найден" });**

**// если пользователь найден, удаляем его**

**users.Remove(user);**

**return Results.Json(user);**

**});**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**});**

**app.Run();**

**public class Person**

**{**

**public int Id { get; set; }**

**public string Name { get; set; } = "";**

**public int Age { get; set; }**

**}**

### Определение клиента

**using System.Net.Http.Json;**

**using System.Net;**

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**string url = "https://localhost:7094/api/users";**

**static async Task Main()**

**{**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// ПОЛУЧАЕМ СПИСОК ОБЪЕКТОВ**

**List<Person>? people = await httpClient**

**.GetFromJsonAsync<List<Person>>(url);**

**if (people != null)**

**{**

**foreach (var person in people)**

**{**

**// person - что-то делаем с каждым объектом из списка**

**}**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// ПОЛУЧАЕМ ОДИН ОБЪЕКТ**

**int id = 1; // получим объект с id = 1**

**// можем просто получить ответ:**

**Person? person = await httpClient**

**.GetFromJsonAsync<Person>($"{url}/{id}");**

**// либо вначале проверить, есть ли такой объект, и поступить //соответственно:**

**// вначале узнаем, есть ли такой объект на сервере**

**using var response = await httpClient.GetAsync($"{url}/{id}");**

**// если объект на сервере не найден, то есть статусный код равен 404**

**if (response.StatusCode == HttpStatusCode.NotFound)**

**{**

**Error? error = await response.Content.ReadFromJsonAsync<Error>();**

**// error?.Message - читаем сообщение ошибки**

**}**

**// если все в порядке, то считываем ответ**

**else if (response.StatusCode == HttpStatusCode.OK)**

**{**

**Person? person = await response**

**.Content**

**.ReadFromJsonAsync<Person>();**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// ДОБАВЛЯЕМ ОБЪЕКТ**

**// PostAsync() или PostAsJsonAsync()**

**// создаем отправляемый объект**

**Person tom = new Person { Name = "Tom", Age = 33 };**

**// отправляем объект**

**using var response = await httpClient.PostAsJsonAsync(url, tom);**

**// можем считать ответ**

**Person? person = await response.Content.ReadFromJsonAsync<Person>();**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// ИЗМЕНЯЕМ ОБЪЕКТ**

**// PutAsync() или PutAsJsonAsync()**

**// id изменяемого объекта**

**id = 1;**

**// создаем измененный объект**

**Person tom = new Person { Id = id, Name = "Tom", Age = 33 };**

**// отправляем объект**

**using var response = await httpClient.PutAsJsonAsync(url, tom);**

**// если объект на сервере не найден, то есть статусный код равен 404**

**if (response.StatusCode == HttpStatusCode.NotFound)**

**{**

**Error? error = await response.Content.ReadFromJsonAsync<Error>();**

**// error?.Message - читаем сообщение ошибки**

**}**

**// если все в порядке, то считываем ответ**

**else if (response.StatusCode == HttpStatusCode.OK)**

**{**

**Person? person = await response**

**.Content**

**.ReadFromJsonAsync<Person>();**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// УДАЛЯЕМ ОБЪЕКТ**

**// id удаляемого объекта**

**id = 1;**

**Person? person = await httpClient**

**.DeleteFromJsonAsync<Person>($"{url}/{id}");**

**// либо вначале проверим, есть ли вообще такой объект**

**using var response = await httpClient.DeleteAsync($"{url}/{id}");**

**if (response.StatusCode == HttpStatusCode.NotFound)**

**{**

**Error? error = await response.Content.ReadFromJsonAsync<Error>();**

**// error?.Message - читаем сообщение ошибки**

**}**

**// если все в порядке, то считываем ответ**

**else if (response.StatusCode == HttpStatusCode.OK)**

**{**

**Person? person = await response**

**.Content**

**.ReadFromJsonAsync<Person>();**

**}**

**}**

**}**

**public class Person**

**{**

**public int Id { get; set; }**

**public string Name { get; set; } = "";**

**public int Age { get; set; }**

**}**

## Отправка форм и класс FormUrlEncodedContent

#### На стороне сервера:

**var builder = WebApplication.CreateBuilder();**

**var app = builder.Build();**

**app.MapPost("/data", async(HttpContext httpContext) =>**

**{**

**// получаем данные формы**

**var form = httpContext.Request.Form;**

**string? name = form["name"];**

**string? email = form["email"];**

**string? age = form["age"];**

**// отправляем полученные значения одной строкой обратно клиенту**

**await httpContext**

**.Response**

**.WriteAsync($"Name: {name} Email:{email} Age: {age}");**

**});**

**app.Run();**

#### На стороне клиента:

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// данные для отправки должны быть в виде объекта**

**// IEnumerable<KeyValuePair<string, string>>**

**Dictionary<string, string> data = new Dictionary<string, string>**

**{**

**["name"]= "Tom",**

**["email"]= "tom@localhost.com",**

**["age"] = "38"**

**};**

**// создаем объект HttpContent**

**HttpContent contentForm = new FormUrlEncodedContent(data);**

**// отправляем запрос**

**using var response = await httpClient**

**.PostAsync("https://localhost:7094/data", contentForm);**

**// получаем ответ**

**string responseText = await response.Content.ReadAsStringAsync();**

**Console.WriteLine(responseText);**

**}**

**}**

## Отправка потоков и массива байтов

### Отправка потоков. StreamContent

#### На стороне сервера:

**var builder = WebApplication.CreateBuilder();**

**var app = builder.Build();**

**app.MapPost("/data", async(HttpContext httpContext) =>**

**{**

**// путь к папке, где будут храниться файлы**

**var uploadPath = $"{Directory.GetCurrentDirectory()}/uploads";**

**// создаем папку для хранения файлов**

**Directory.CreateDirectory(uploadPath);**

**// генерируем произвольное название файла с помощью guid**

**string fileName = Guid.NewGuid().ToString();**

**// получаем поток**

**using (var fileStream =**

**new FileStream($"{uploadPath}/{fileName}.jpg", FileMode.Create))**

**{**

**await httpContext.Request.Body.CopyToAsync(fileStream);**

**}**

**await httpContext.Response.WriteAsync("Данные сохранены");**

**});**

**app.Run();**

#### На стороне клиента:

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// отправляемые данные**

**string filePath = "D:\\forest.jpg";**

**using var fileStream = File.OpenRead(filePath);**

**// создаем объект HttpContent**

**StreamContent content = new StreamContent(fileStream);**

**// отправляем запрос**

**using var response = await**

**httpClient.PostAsync("https://localhost:7094/data", content);**

**// получаем ответ**

**string responseText = await response.Content.ReadAsStringAsync();**

**Console.WriteLine(responseText);**

**}**

**}**

### Отправка массива байтов. ByteArrayContent

#### На стороне сервера:

**var builder = WebApplication.CreateBuilder();**

**var app = builder.Build();**

**app.MapPost("/data", async(HttpContext httpContext) =>**

**{**

**// считываем полученные данные в строку**

**using StreamReader streamReader =**

**new StreamReader(httpContext.Request.Body);**

**string message = await streamReader.ReadToEndAsync();**

**await httpContext.Response.WriteAsync($"Отправлено сообщение: {message}");**

**});**

**app.Run();**

#### На стороне клиента:

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// отправляемые данные**

**string message = "Hello METANIT.COM";**

**// считываем строку в массив байтов**

**byte[] messageToBytes = System.Text.Encoding.UTF8.GetBytes(message);**

**// формируем отправляемое содержимое**

**var content = new ByteArrayContent(messageToBytes);**

**// отправляем запрос**

**using var response = await**

**httpClient.PostAsync("https://localhost:7094/data", content);**

**// получаем ответ**

**string responseText = await response.Content.ReadAsStringAsync();**

**Console.WriteLine(responseText);**

**}**

**}**

## Отправка файлов и класс MultipartFormDataContent

**Для отправки файлов на сервер HttpClient использует класс System.Net.Http.MultipartFormDataContent. Фактически этот класс выступает в качестве контейнера объектов HttpContent. А для добавления элементов в MultipartFormDataContent применяется метод Add():**

**public void Add(HttpContent content, string name, string fileName);**

**content - отправляемое содержимое (это может быть как файлы, так и любые другие данные);**

**name - определяет название данных в запросе, по которому мы можем получить файл на сервере;**

**fileName - устанавливает имя файла.**

### Загрузка одного файла

#### На стороне сервера:

**var builder = WebApplication.CreateBuilder();**

**var app = builder.Build();**

**app.MapPost("/upload", async (HttpContext context) =>**

**{**

**// получем коллецию загруженных файлов**

**IFormFileCollection files = context.Request.Form.Files;**

**// путь к папке, где будут храниться файлы**

**var uploadPath = $"{Directory.GetCurrentDirectory()}/uploads";**

**// создаем папку для хранения файлов**

**Directory.CreateDirectory(uploadPath);**

**// пробегаемся по всем файлам**

**foreach (var file in files)**

**{**

**// формируем путь к файлу в папке uploads**

**string fullPath = $"{uploadPath}/{file.FileName}";**

**// сохраняем файл в папку uploads**

**using (var fileStream = new FileStream(fullPath, FileMode.Create))**

**{**

**await file.CopyToAsync(fileStream);**

**}**

**}**

**await context.Response.WriteAsync("Файлы успешно загружены");**

**});**

**app.Run();**

#### На стороне клиента:

###### Отправка потоков. StreamContent

**using System.Net.Http.Headers;**

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// адрес сервера**

**var serverAddress = "https://localhost:7094/upload";**

**// путь к файлу**

**var filePath = @"D:\forest.jpg";**

**// создаем MultipartFormDataContent**

**using var multipartFormContent = new MultipartFormDataContent();**

**// Загружаем отправляемый файл**

**var fileStreamContent = new StreamContent(File.OpenRead(filePath));**

**// Устанавливаем заголовок Content-Type**

**fileStreamContent.Headers.ContentType =**

**new MediaTypeHeaderValue("image/jpeg");**

**// Добавляем загруженный файл в MultipartFormDataContent**

**multipartFormContent**

**.Add(fileStreamContent, name: "file", fileName: "forest.jpg");**

**// Отправляем файл**

**using var response = await httpClient**

**.PostAsync(serverAddress, multipartFormContent);**

**// считываем ответ**

**var responseText = await response.Content.ReadAsStringAsync();**

**Console.WriteLine(responseText);**

**}**

**}**

###### Отправка массива байтов. ByteArrayContent

**using System.Net.Http.Headers;**

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// адрес сервера**

**var serverAddress = "https://localhost:7094/upload";**

**// пусть к файлу**

**var filePath = @"D:\forest.jpg";**

**// создаем MultipartFormDataContent**

**using var multipartFormContent = new MultipartFormDataContent();**

**// считываем данные файла в массив байтов**

**byte[] fileToBytes = await File.ReadAllBytesAsync(filePath);**

**// формируем отправляемое содержимое**

**var content = new ByteArrayContent(fileToBytes);**

**// Устанавливаем заголовок Content-Type**

**content.Headers.ContentType = new MediaTypeHeaderValue("image/jpeg");**

**// Добавляем загруженный файл в MultipartFormDataContent**

**multipartFormContent**

**.Add(content, name: "file", fileName: "forest5.jpg");**

**// Отправляем файл**

**using var response = await httpClient**

**.PostAsync(serverAddress, multipartFormContent);**

**// считываем ответ**

**var responseText = await response.Content.ReadAsStringAsync();**

**Console.WriteLine(responseText);**

**}**

**}**

### Множественная отправка файлов

**using System.Net.Http.Headers;**

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// адрес для отправки**

**var serverAddress = "https://localhost:7094/upload";**

**// пути к файлам**

**var files = new string[] { "D:\forest.jpg", "D:\cats.jpg" };**

**using var multipartFormContent = new MultipartFormDataContent();**

**// в цикле добавляем все файлы в MultipartFormDataContent**

**foreach (var file in files)**

**{**

**// получаем краткое имя файла**

**var fileName = Path.GetFileName(file);**

**var fileStreamContent = new StreamContent(File.OpenRead(file));**

**fileStreamContent.Headers.ContentType =**

**new MediaTypeHeaderValue("image/jpeg");**

**multipartFormContent**

**.Add(fileStreamContent, name: "files", fileName: fileName);**

**}**

**// Отправляем файлы**

**using var response = await httpClient**

**.PostAsync(serverAddress, multipartFormContent);**

**// считываем ответ**

**var responseText = await response.Content.ReadAsStringAsync();**

**Console.WriteLine(responseText);**

**}**

**}**

### Отправка смешанного содержимого

##### На стороне сервера:

**using Microsoft.AspNetCore.Mvc;**

**var builder = WebApplication.CreateBuilder();**

**var app = builder.Build();**

**app.MapPost("/upload", async (HttpContext context) =>**

**{**

**var form = context.Request.Form;**

**// получаем отдельные данные**

**string? username = form["username"];**

**string? email = form["email"];**

**// получаем коллецию загруженных файлов**

**IFormFileCollection files = form.Files;**

**// путь к папке, где будут храниться файлы**

**var uploadPath = $"{Directory.GetCurrentDirectory()}/uploads";**

**// создаем папку для хранения файлов**

**Directory.CreateDirectory(uploadPath);**

**foreach (var file in files)**

**{**

**// путь к папке uploads**

**string fullPath = $"{uploadPath}/{file.FileName}";**

**// сохраняем файл в папку uploads**

**using (var fileStream = new FileStream(fullPath, FileMode.Create))**

**{**

**await file.CopyToAsync(fileStream);**

**}**

**}**

**return $"Данные пользователя {username} ({email}) успешно загружены";**

**});**

**app.Run();**

##### На стороне клиента:

**using System.Net.Http.Headers;**

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**using var multipartFormContent = new MultipartFormDataContent();**

**// добавляем обычные данные**

**multipartFormContent.Add(new StringContent("Tom"), name: "username");**

**multipartFormContent**

**.Add(new StringContent("tom@localhost.com"), name: "email");**

**// добавляем файл**

**var fileStreamContent =**

**new StreamContent(File.OpenRead("D:\logo.jpg"));**

**fileStreamContent.Headers.ContentType =**

**new MediaTypeHeaderValue("image/jpeg");**

**multipartFormContent**

**.Add(fileStreamContent, name: "files", fileName: "logo.jpg");**

**// Отправляем данные**

**using var response = await httpClient**

**.PostAsync("https://localhost:7094/upload", multipartFormContent);**

**// считываем ответ**

**var responseText = await response.Content.ReadAsStringAsync();**

**Console.WriteLine(responseText);**

**}**

**}**

## Отправка и получение куки с HttpClient

### Cookie

#### Конструкторы

**public Cookie ();**

**public Cookie (string name, string? value);**

**public Cookie (string name, string? value, string? path);**

**public Cookie (string name, string? value, string? path, string? domain);**

##### Параметры конструкторов:

* **name: название куки**
* **value: значение куки**
* **path: путь (по умолчанию "/", то есть корень веб-приложения)**
* **domain: домен, для которого созданы данные куки**

#### Свойства Cookie:

* **Comment: возвращает или задает комментарий, который сервер может добавлять в Cookie**
* **CommentUri: возвращает или задает комментарий для URI, который сервер может предоставлять с Cookie**
* **Discard: возвращает или задает флаг сброса, задаваемый сервером**
* **Domain: возвращает или задает URI, для которого Cookie является допустимым**
* **Expired: возвращает или задает текущее состояние Cookie**
* **Expires: возвращает или задает для Cookie дату и время окончания действия в виде DateTime**
* **HttpOnly: определяет, может ли получить доступ к файлу cookie скрипт страницы или другое активное содержимое**
* **Name: возвращает или задает имя Cookie**
* **Path: возвращает или задает идентификаторы URI, к которым применяется Cookie**
* **Port: возвращает или задает список TCP-портов, к которым применяется Cookie**
* **Secure: возвращает или задает уровень безопасности Cookie**
* **TimeStamp: возвращает время выпуска файла cookie в виде DateTime**
* **Value: возвращает или задает Value для объекта Cookie**
* **Version: возвращает или задает версию поддержки запоминания состояния HTTP, которой соответствует файл cookie**

#### Пример:

**Cookie nameCookie = new Cookie("name", "Tom");**

**Console.WriteLine(nameCookie.Name); // name**

**Console.WriteLine(nameCookie.Value); // Tom**

### CookieContainer

#### Свойства CookieContainer:

* **Capacity: получает или задает количество экземпляров Cookie, которое может храниться в CookieContainer. Значение по умолчанию - 300**
* **Count: возвращает количество экземпляров Cookie, которые имеются в текущий момент в CookieContainer.**
* **MaxCookieSize: представляет максимально допустимую длину Cookie в байтах. Значение по умолчанию - 4096 байт**
* **PerDomainCapacity: получает или задает количество экземпляров Cookie, которое может храниться в CookieContainer для каждого домена. Значение по умолчанию - 20**

#### Конструкторы:

**public CookieContainer (int capacity, int perDomainCapacity,**

**int maxCookieSize);**

**public CookieContainer (int capacity);**

**public CookieContainer ();**

#### Методы CookieContainer:

* **Add(Cookie): добавляет объект Cookie в CookieContainer. Метод использует свойство Domen из объекта Cookie для определения Uri, для которого добавляется Cookie.**
* **Add(Uri, Cookie): добавляет экземпляр Cookie в CookieContainer для определенного URI.**
* **GetAllCookies(): возвращает объект CookieCollection , который содержит все объекты Cookie контейнера.**
* **GetCookieHeader(Uri): возвращает заголовок HTTP-cookie, который включает данные всех объектов Cookie, связанных с определенным URI.**
* **GetCookies(Uri): возвращает коллекцию CookieCollection с объектами Cookie, которые связанны с указанным URI.**
* **SetCookies(Uri, String): добавляет объекты Cookie для определенного URI.**

#### Например, добавление кук:

**Uri uri = new Uri("http://metanit.com");**

**Cookie nameCookie = new Cookie("name", "Tom");**

**Cookie emailCookie = new Cookie("email", "tom@localhost.com");**

**CookieContainer cookieContainer = new CookieContainer();**

**// добавляем куки nameCookie для uri http://metanit.com**

**cookieContainer.Add(uri, nameCookie);**

**// добавляем куки emailCookie для uri http://metanit.com**

**cookieContainer.Add(uri, emailCookie);**

##### или:

**// установка кук**

**cookieContainer.SetCookies(uri, "name=Tom");**

**cookieContainer.SetCookies(uri, "email=tom@localhost.com");**

#### Получение кук из контейнера:

**// получаем все куки**

**var allCookies = cookieContainer.GetAllCookies();**

**foreach (Cookie cookie in allCookies) ... // cookie.Name, cookie.Value**

**// получаем куки только для uri http://metanit.com**

**var uriCookies = cookieContainer.GetCookies(uri);**

**foreach (Cookie cookie in uriCookies) ... // cookie.Name, cookie.Value**

**// получение заголовка кук**

**var cookieHeader = cookieContainer.GetCookieHeader(uri);**

**// name=Tom; email=tom@localhost.com**

### Получение куки

#### Отправка кук клиенту в ответ на Get-запрос. На стороне сервера:

**var builder = WebApplication.CreateBuilder();**

**var app = builder.Build();**

**app.MapGet("/", (HttpContext context) =>**

**{**

**// устанавливаем куки**

**context.Response.Cookies.Append("name", "Tom");**

**context.Response.Cookies.Append("email", "tom@localhost.com");**

**// с помощью метода context.Response.Cookies.Append() в ответ клиенту**

**// добавляются две куки: name и email**

**});**

**app.Run();**

#### На стороне клиента:

##### Получение кук из запроса через заголовок Set-Cookie

**using System.Net;**

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// адрес сервера**

**Uri uri = new Uri("https://localhost:7094/");**

**using var response = await httpClient.GetAsync(uri);**

**CookieContainer cookies = new CookieContainer();**

**// получаем из запроса все элементы с заголовком Set-Cookie**

**foreach (var cookieHeader in response**

**.Headers.GetValues("Set-Cookie"))**

**// добавляем заголовки кук в CookieContainer**

**cookies.SetCookies(uri, cookieHeader);**

**// получение всех куки**

**foreach (Cookie cookie in cookies.GetCookies(uri))**

**Console.WriteLine($"{cookie.Name}: {cookie.Value}");**

**// получение отдельных куки**

**// получаем куку "email"**

**Cookie? email = cookies.GetCookies(uri)**

**.FirstOrDefault(c => c.Name == "email");**

**Console.WriteLine($"Электронный адрес: {email?.Value}");**

**}**

**}**

##### Применение HttpClientHandler

**using System.Net;**

**class Program**

**{**

**static HttpClient? httpClient;**

**static async Task Main()**

**{**

**// адрес сервера**

**Uri uri = new Uri("https://localhost:7094/");**

**var cookies = new CookieContainer();**

**using var handler = new HttpClientHandler();**

**// HttpClientHandler имеет свойство CookieContainer,**

**// которое принимает объект типа CookieContainer**

**// и представляет контейнер куки для данного HttpClientHandler.**

**handler.CookieContainer = cookies;**

**// Объект HttpClient инициализируется определенным обработчиком.**

**// По умолчанию это объект типа HttpClientHandler.**

**httpClient = new HttpClient(handler);**

**using var response = await httpClient.GetAsync(uri);**

**// получение всех куки**

**foreach (Cookie cookie in cookies.GetCookies(uri))**

**Console.WriteLine($"{cookie.Name}: {cookie.Value}");**

**// получение отдельных куки**

**// получаем куку "email"**

**Cookie? email = cookies.GetCookies(uri)**

**.FirstOrDefault(c=>c.Name== "email");**

**Console.WriteLine($"Электронный адрес: {email?.Value}");**

**}**

**}**

**Если у объекта HttpClientHandler свойство UseCookies равно true (а это значение по умолчанию), то свойство CookieContainer будет использоваться для сохранения куки с сервера или их отправки на сервер.**

### Отправка кук

#### Получаем куки на сервере:

**var builder = WebApplication.CreateBuilder();**

**var app = builder.Build();**

**app.MapGet("/", (HttpContext context) =>**

**{**

**// получаем куки**

**context.Request.Cookies.TryGetValue("name", out string? name);**

**context.Request.Cookies.TryGetValue("email", out string? email);**

**// возвращаем значение кук клиенту**

**return $"Name: {name} Email:{email}";**

**});**

**app.Run();**

#### На стороне клиента:

**using System.Net;**

**class Program**

**{**

**static HttpClient httpClient = new HttpClient();**

**static async Task Main()**

**{**

**// адрес сервера**

**Uri uri = new Uri("https://localhost:7094/");**

**CookieContainer cookies = new CookieContainer();**

**// устанавливаем куки name и email**

**cookies.Add(uri, new Cookie("name", "Bob"));**

**cookies.Add(uri, new Cookie("email", "bob@localhost.com"));**

**// устанавливаем заголовок cookie**

**httpClient.DefaultRequestHeaders**

**.Add("cookie", cookies.GetCookieHeader(uri));**

**using var response = await httpClient.GetAsync(uri);**

**string responseText = await response.Content.ReadAsStringAsync();**

**Console.WriteLine(responseText);**

**}**

**}**

## HttpListener. HTTP-сервер

**Для прослушивания подключений по протоколу HTTP и ответа на HTTP-запросы предназначен класс HttpListener из пространства имен System.Net. Данный класс построен на базе библиотеки HTTP.sys, которая является слушателем режима ядра и обрабатывает весь трафик HTTP для Windows.**

#### Единственный конструктор:

**HttpListener server = new HttpListener();**

#### Свойства HttpListener:

* **AuthenticationSchemes: возвращает или задает схему аутентификации клиентов**
* **IsListening: указывает, был ли запущен объект HttpListener**
* **IsSupported: указывает, можно ли использовать прослушиватель HttpListener в текущей операционной системе.**
* **Prefixes: возвращает префиксы URI, обрабатываемые этим объектом HttpListener.**

#### Методы HttpListener:

* **Close(): завершает работу HttpListener.**
* **GetContext() / GetContextAsync(): ожидает входящие запросы**
* **Start(): запускает прослушивание входящих запросов**
* **Stop(): останавливает получение новых входящих запросов и завершает обработку всех текущих запросов.**

### Запуск HttpListener

**using System.Net;**

**HttpListener server = new HttpListener();**

**// установка адресов прослушки**

**// адрес должен заканчиваться на слеш '/'**

**server.Prefixes.Add("http://127.0.0.1:8888/connection/");**

**server.Start(); // начинаем прослушивать входящие подключения**

**// ............**

**server.Stop(); // останавливаем сервер**

**server.Close(); // закрываем HttpListener**

### Контекст

**С помощью метода GetContext()/GetContextAsync() можно получить контекст входящего подключения в виде объекта HttpListenerContext.**

#### Свойства HttpListenerContext:

* **Request: возвращает информацию о запросе в виде объекта HttpListenerRequest**
* **Response: устанавливаливает ответ сервера в виде объекта HttpListenerResponse**
* **User: представляет данные об аутентифицированном пользователе в виде объекта System.Security.Principal.IPrincipal?**

**// получаем контекст**

**var context = await server.GetContextAsync();**

**// получаем данные запроса**

**var request = context.Request;**

**// получаем объект для установки ответа**

**var response = context.Response;**

**// получаем данные пользователя**

**var user = context.User;**

### Отправка ответа

**Свойство Response объекта HttpListenerContext представляет объект HttpListenerResponse, который позволяет настроить и отправить клиенту некоторый ответ.**

#### Свойства HttpListenerResponse:

* **ContentEncoding: устанавливает кодировку ответа в виде объекта Encoding из заголовка Content-Type в виде объекта System.Text.Encoding.**
* **ContentLength64: устанавливает заголовок Content-Length (размер ответа).**
* **ContentType: устанавливает значение заголовка Content-Type.**
* **Cookies: устанавливает файлы cookie, которые отправляются клиенту в виде объекта System.Net.CookieCollection.**
* **Headers: устанавливает заголовки в виде коллекции WebHeaderCollection (аналоги словаря, где названия заголовков служат ключами).**
* **KeepAlive: устанавливает значение bool, которое указывает, требуется ли постоянное подключение.**
* **OutputStream: представляет поток ответа.**
* **ProtocolVersion: устанавливает версию протокола HTTP, которая используется в ответе.**
* **RedirectLocation: устанавливает адрес для переадресации.**
* **SendChunked: представляет значение bool, которое указывает, будет ли ответ разбиваться на чанки/части.**
* **StatusCode: устанавливает статусный код ответа.**
* **StatusDescription: устанавливает описание к статусному коду (строку статуса).**

#### Методы HttpListenerResponse:

* **void AddHeader (string name, string value): добавляет в ответ заголовок с именем name и значением value**
* **void AppendCookie (System.Net.Cookie cookie);: добавляет в ответ куки**
* **void AppendHeader (string name, string value): добавляет к заголовоку с именем name значение value**
* **void Redirect (string url): выполняет редирект на адрес url**
* **void SetCookie (System.Net.Cookie cookie): устанавливает куки**

#### Пример:

**using System.Net;**

**using System.Text;**

**HttpListener server = new HttpListener();**

**// установка адресов прослушки**

**server.Prefixes.Add("http://127.0.0.1:8888/connection/");**

**server.Start(); // начинаем прослушивать входящие подключения**

**// получаем контекст**

**var context = await server.GetContextAsync();**

**var response = context.Response;**

**// отправляемый в ответ код возвращает html**

**string responseText =**

**@"<!DOCTYPE html>**

**<html>**

**<head>**

**<meta charset='utf8'>**

**<title>METANIT.COM</title>**

**</head>**

**<body>**

**<h2>Hello METANIT.COM</h2>**

**</body>**

**</html>";**

**byte[] buffer = Encoding.UTF8.GetBytes(responseText);**

**// получаем поток ответа и пишем в него ответ**

**response.ContentLength64 = buffer.Length;**

**using Stream output = response.OutputStream;**

**// отправляем данные**

**await output.WriteAsync(buffer);**

**await output.FlushAsync();**

**Console.WriteLine("Запрос обработан");**

**server.Stop();**

### Получение информации о запросе

**Свойство Request возвращает информацию о запросе в виде объекта HttpListenerRequest.**

#### Свойства HttpListenerRequest:

* **AcceptTypes: возвращает значение заголовка Accept в виде массива string[].**
* **ClientCertificateError: возвращает код ошибки, связанной с сертификатом X509Certificate, предоставленным клиентом.**
* **ContentEncoding: возвращает значение кодировки из заголовка Content-Type в виде объекта System.Text.Encoding.**
* **ContentLength64: возвращает заголовок Content-Length (если он не установлен, то возвращается -1).**
* **ContentType: возвращает значение заголовка Content-Type.**
* **Cookies: возвращает файлы cookie, отправленные вместе с запросом в виде объекта System.Net.CookieCollection.**
* **HasEntityBody: возвращает true, если в запросе кроме заголовоков переданы какие-нибудь данные (тело запроса).**
* **Headers: возвращает все заголовки в виде коллекции System.Collections.Specialized.NameValueCollection (аналоги словаря, где названия заголовков служат ключами).**
* **HttpMethod: возвращает метод HTTP, использованный для отправки запроса клиентом.**
* **InputStream: возвращает поток, которые содержит данные тела запроса.**
* **IsAuthenticated: возвращает значение bool, которое указывает, аутентифицирован ли клиент, отправивший этот запрос.**
* **IsLocal: возвращает значение bool, которое указывает, был ли запрос отправлен с локального компьютера.**
* **IsSecureConnection: возвращает значение bool, которое указывает, применяется ли для запроса протокол SSL.**
* **IsWebSocketRequest: возвращает значение bool, которое указывает, использовался ли при запросе WebSocket.**
* **KeepAlive: возвращает значение bool, которое указывает, требует ли клиент постоянного подключения.**
* **LocalEndPoint: возвращает IP-адрес сервера и номер порта, на который будет перенаправлен запрос.**
* **ProtocolVersion: возвращает версию протокола HTTP, которая используется в запросе.**
* **QueryString: возвращает строку запроса.**
* **RawUrl: возвращает URL-адрес запроса (без узла и порта).**
* **RemoteEndPoint: возвращает IP-адрес клиента, который отправил запрос.**
* **RequestTraceIdentifier: возвращает идентификатор HTTP-запроса.**
* **Url: возвращает адрес запроса в виде объекта Uri.**
* **UrlReferrer: возвращает URI ресурса, который перенаправляет клиент на сервер.**
* **UserAgent: возвращает заголовок User-Agent.**
* **UserHostAddress: возвращает IP-адрес, на который будет перенаправлен запрос.**
* **UserHostName: возвращает DNS-хост, указанный клиентом.**
* **UserLanguages: возвращает значение заголовка AcceptLanguage (при его отсутствии возвращается null).**

#### Пример:

**using System.Net;**

**using System.Text;**

**HttpListener server = new HttpListener();**

**// установка адресов прослушки**

**server.Prefixes.Add("http://127.0.0.1:8888/connection/");**

**server.Start(); // начинаем прослушивать входящие подключения**

**Console.WriteLine("Сервер запущен. Ожидание подключений...");**

**// получаем контекст**

**var context = await server.GetContextAsync();**

**var request = context.Request; // получаем данные запроса**

**Console.WriteLine($"адрес приложения: {request.LocalEndPoint}");**

**Console.WriteLine($"адрес клиента: {request.RemoteEndPoint}");**

**Console.WriteLine(request.RawUrl);**

**Console.WriteLine($"Запрошен адрес: {request.Url}");**

**Console.WriteLine("Заголовки запроса:");**

**foreach(string item in request.Headers.Keys)**

**{**

**Console.WriteLine($"{item}:{request.Headers[item]}");**

**}**

**var response = context.Response; // получаем объект для установки ответа**

**byte[] buffer = Encoding.UTF8.GetBytes("Hello METANIT");**

**// получаем поток ответа и пишем в него ответ**

**response.ContentLength64 = buffer.Length;**

**using Stream output = response.OutputStream;**

**// отправляем данные**

**await output.WriteAsync(buffer);**

**await output.FlushAsync();**

**server.Stop(); // останавливаем сервер**

Протокол TCP

**Для работы с протоколом TCP в .NET предназначены классы TcpClient и TcpListener. Эти классы строятся поверх класса System.Net.Sockets.Socket.**

## TCP-клиент на сокетах

**Для определения сокета, который использует протокол TCP, необходимо для сокета указать в качестве типа протокола Tcp, а в качестве типа сокета - Stream:**

**Socket tcpSocket = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

### Подключение к хосту

**Для подключения к удаленному хосту применяется метод Connect()/ConnectAsync():**

**public Task ConnectAsync (string host, int port);**

**public Task ConnectAsync (IPAddress address, int port);**

**// есть и другие перегрузки**

#### Пример:

**using System.Net.Sockets;**

**var port = 80;**

**var url = "www.google.com";**

**using var socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);**

**try**

**{**

**// пытаемся подключиться используя URL-адрес и порт**

**await socket.ConnectAsync(url, port);**

**Console.WriteLine($"Подключение к {url} установлено");**

**}**

**// при неудаче будет сгенерировано исключение SocketException**

**catch (SocketException)**

**{**

**Console.WriteLine($"Не удалось установить подключение к {url}");**

**}**

### Информация о подключении

**С помощью свойства RemoteEndPoint можно получить ip-адрес+порт хоста.**

**С помощью свойства LocalEndPoint можно получить адрес самого сокета.**

#### Пример:

**using System.Net.Sockets;**

**var port = 80;**

**var url = "www.google.com";**

**using var socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);**

**try**

**{**

**// пытаемся подключиться используя URL-адрес и порт**

**await socket.ConnectAsync(url, port);**

**Console.WriteLine($"Подключение к {url} установлено");**

**Console.WriteLine($"Адрес подключения {socket.RemoteEndPoint}");**

**Console.WriteLine($"Адрес приложения {socket.LocalEndPoint}");**

**}**

**catch (SocketException)**

**{**

**Console.WriteLine($"Не удалось установить подключение к {url}");**

**}**

### Отключение от хоста

**Если мы завершили взаимодействие с хостом, но планируем продолжать использовать сокет, чтобы соединение с удаленным хостом не висело, мы можем отключиться с помощью метода Disconnect() / DisconnectAsync(). Данный метод в качестве параметра принимает значение bool - если оно равно true, то после отключения можно заново использовать сокет для новых подключений:**

#### Пример:

**using System.Net.Sockets;**

**var port = 80;**

**var url = "www.google.com";**

**using var socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);**

**try**

**{**

**await socket.ConnectAsync(url, port);**

**await socket.DisconnectAsync(true); // отключаемся**

**}**

**catch (SocketException ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

**Чтобы гарантировать, что все данные отправлены и получены перед закрытием подключения, перед вызовом метода Disconnect/DisconnectAsync Microsoft рекомендует вызывать метод Shutdown.**

### Отправка данных

**Для отправки данных применяется метод Send()/SendAsync().**

**public Task<int> SendAsync (ArraySegment<byte> buffer);**

**public Task<int> SendAsync (**

**ArraySegment<byte> buffer,**

**SocketFlags socketFlags);**

**// есть и другие перегрузки**

**В качестве результата метод SendAsync() возвращает количество отправленных данных.**

#### Пример:

**using System.Net.Sockets;**

**using System.Text;**

**var port = 80;**

**var url = "www.google.com";**

**using var socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);**

**try**

**{**

**await socket.ConnectAsync(url, port);**

**// определяем отправляемые данные**

**var message = $"GET / HTTP/1.1\r\nHost: {url}\r\nConnection: close\r\n\r\n";**

**// конвертируем данные в массив байтов**

**var messageBytes = Encoding.UTF8.GetBytes(message);**

**int bytesSent = await socket.SendAsync(messageBytes);**

**// массив байт автоматически конвертируется в ArraySegment<byte>**

**Console.WriteLine($"на адрес {url} отправлено {bytesSent} байт(а)");**

**}**

**catch (SocketException ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

### Получение данных от хоста

**Task<int> ReceiveAsync (ArraySegment<byte> buffer);**

**int Receive (byte[] buffer);**

**// есть и другие перегрузки**

**Результатом обоих методов является количество считанных байтов.**

#### Пример:

**using System.Net.Sockets;**

**using System.Text;**

**var port = 80;**

**var url = "www.google.com";**

**using var socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);**

**try**

**{**

**await socket.ConnectAsync(url, port);**

**// определяем отправляемые данные**

**var message = $"GET / HTTP/1.1\r\nHost: {url}\r\nConnection: close\r\n\r\n";**

**// конвертируем данные в массив байтов**

**var messageBytes = Encoding.UTF8.GetBytes(message);**

**// отправляем данные**

**await socket.SendAsync(messageBytes);**

**// явным образом отключаем отправку данных на сокете**

**await socket.Shutdown(SocketShutdown.Send);**

**// буфер для получения данных**

**var responseBytes = new byte[512];**

**// объект StringBuilder для склеивания ответа в строку**

**var builder = new StringBuilder();**

**int bytes;**

**// в цикле получаем данные**

**do**

**{**

**// получаем данные**

**bytes = await socket.ReceiveAsync(responseBytes);**

**// массив байт автоматически конвертируется в ArraySegment<byte>**

**// преобразуем полученные данные в строку**

**string responsePart = Encoding.UTF8.GetString(responseBytes, 0, bytes);**

**// добавляем в StringBuilder**

**builder.Append(responsePart);**

**}**

**while (bytes > 0); // повторяем цикл, пока сервер отправляет более 0 байтов**

**// выводим ответ на консоль**

**Console.WriteLine(builder);**

**}**

**catch (SocketException ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

**ReceiveAsync завершает выполнение и возвращает количество байт ТОЛЬКО В ТОМ СЛУЧАЕ, когда буфер заполнен или когда одна из сторон завершает соединение. Заголовок "Connection: Close" предписывает гуглу закрыть подключение после завершения текущей транзакции. Поэтому выполнение не зависает на строке "bytes = await socket.ReceiveAsync(responseBytes);". Мы можем отключить получение и/или отправку данных на сокете с помощью метода Shutdown().**

### Свойство Available

**Свойство Available хранит количество доступных для чтения байтов.**

**Почему бы не использовать свойство Available для выхода из цикла:**

**while (socket.Available > 0);**

**Природа TCP такова, что какой-то пакет может прийти быстрее, какой-то задержится, какой-то будет потерян, и потребуется переотправка. Поэтому может возникнуть ситуация, что сервер отправил данные, часть данных пришла. В какой-то момент свойство Available у сокета возратило 0, соответственно произошел выход из цикла. И в итоге мы получим неполные данные. Поэтому использование свойства Available в данном случае не лучший вариант.**

**Обычно при получении данных используют одну из следующих стратегий, которые позволяют определить завершение получения данных:**

* **Использование буфера фиксированной длины, когда мы точно знаем, какой именно объем данных будет послан**
* **Отправка в ответе информации о размере ответа, получив которую, нам будет проще считать нужное количество байтов**
* **Использование маркера окончания ответа, получив который, мы завершим считывание данных**

**Выбор и реализация конкретной стратегии всецело зависит от сервера, который получает запрос и отправляет ответ.**

### Рефакторинг подключения

**Для упрощения работы с сокетом мы можем вынести код подключения и код отправки-получения данных в отдельные методы:**

#### Пример:

**using System.Net.Sockets;**

**using System.Text;**

**var port = 80;**

**var url = "www.google.coms";**

**var response = await SocketSendReceiveAsync(url, port);**

**Console.WriteLine(response);**

**async Task<Socket?> ConnectSocketAsync(string url, int port)**

**{**

**Socket tempSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);**

**try**

**{**

**await tempSocket.ConnectAsync(url, port);**

**return tempSocket;**

**}**

**catch(SocketException ex)**

**{**

**Console.WriteLine(ex.Message);**

**tempSocket.Close();**

**}**

**return null;**

**}**

**async Task<string> SocketSendReceiveAsync(string url, int port)**

**{**

**using Socket? socket = await ConnectSocketAsync(url, port);**

**if (socket is null)**

**return $"Не удалось установить соединение с {url}";**

**// отправляем данные**

**var message = $"GET / HTTP/1.1\r\nHost: {url}\r\nConnection: Close\r\n\r\n";**

**var messageBytes = Encoding.UTF8.GetBytes(message);**

**await socket.SendAsync(messageBytes);**

**// получаем данные**

**int bytes;**

**// буфер для получения данных**

**var responseBytes = new byte[512];**

**var builder = new StringBuilder();**

**do**

**{**

**bytes = await socket.ReceiveAsync(responseBytes);**

**string responsePart = Encoding.UTF8.GetString(responseBytes, 0, bytes);**

**builder.Append(responsePart);**

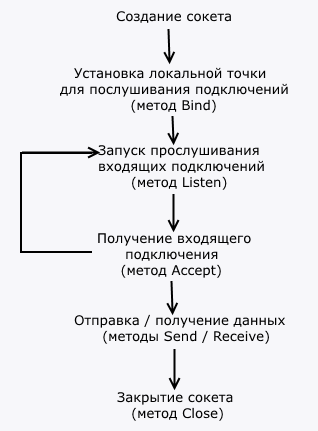
**}**

**while (bytes > 0);**

**return builder.ToString();**

**}**

## TCP-сервер на сокетах



### Привязка к конечной точке. Метод Bind

**Вначале серверный сокет с помощью метода Bind связывается с локальной точкой. В качестве параметра этот метод принимает локальную точку EndPoint, на которой сокет будет принимать подключения от клиентов:**

**public void Bind (EndPoint localEndPoint);**

**Если не имеет значения, на каком именно локальном адресе сервер будет запущен, то можно в качестве адреса использовать значение IPAddress.Any. Тогда серверу будет назначен наиболее подходящий сетевой адрес (при наличии нескольких сетевых интерфейсов). Кроме того, если номер порта не имеет значения, то в качестве порта можно указать число 0. Тогда серверу будет предоставлен один из доступных портов. При использовании такого подхода точный адрес и порт затем можно будет получить через свойство LocalEndpoint.**

#### Пример:

**using System;**

**using System.Net.Sockets;**

**using System.Net;**

**public class Program**

**{**

**public static void Main()**

**{**

**IPEndPoint ipPoint = new IPEndPoint(IPAddress.Any, 8888);**

**using Socket socket = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**socket.Bind(ipPoint); // связываем с локальной точкой ipPoint**

**// получаем конечную точку, с которой связан сокет**

**Console.WriteLine(socket.LocalEndPoint); // 0.0.0.0:8888**

**}**

**}**

### Прослушивание подключений. Метод Listen

**Для запуска прослушивания подключений на выбранной локальной конечной точке применяется метод Listen:**

**public void Listen ();**

**public void Listen (int backlog);**

**При обращении к серверу входящие подключения помещаются в очередь для последующей обработки. По умолчанию эта очередь допускает 2147483647 подключений. Вторая версия метода Listen через параметр позволяет переопределить длину очереди.**

#### Пример:

**using System.Net.Sockets;**

**using System.Net;**

**public class Program**

**{**

**public static void Main()**

**{**

**IPEndPoint ipPoint = new IPEndPoint(IPAddress.Any, 8888);**

**using Socket socket = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**socket.Bind(ipPoint); // связываем с локальной точкой ipPoint**

**socket.Listen(1000); // запускаем прослушивание**

**// количество входящий подключений,**

**// которые можно поместить в очередь**

**// равно 1000**

**}**

**}**

### Подключение клиента

**Для приема подключений сервером применяются методы Accept()/AcceptAsync().**

**public Task<Socket> AcceptAsync ();**

**Все версии методов Accept()/AcceptAsync() в качестве результа возвращают объект Socket, который инкапсулирует входящее подключение, то есть по сути представляет подключенного клиента.**

#### На стороне сервера:

**using System;**

**using System.Net.Sockets;**

**using System.Net;**

**using System.Threading.Tasks;**

**public class Program**

**{**

**public static async Task Main()**

**{**

**IPEndPoint ipPoint = new IPEndPoint(IPAddress.Any, 8888);**

**using Socket socket = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**socket.Bind(ipPoint);**

**socket.Listen();**

**Console.WriteLine("Сервер запущен. Ожидание подключений...");**

**// получаем входящее подключение**

**using Socket client = await socket.AcceptAsync();**

**// получаем адрес клиента**

**Console.WriteLine($"Адрес подключенного клиента: {client.RemoteEndPoint}");**

**}**

**}**

#### На стороне клиента:

**using System.Net.Sockets;**

**using var socket = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**await socket.ConnectAsync("127.0.0.1", 8888);**

**Console.WriteLine($"Подключение к {socket.RemoteEndPoint} установлено");**

**}**

**catch (SocketException)**

**{**

**Console.WriteLine(**

**$"Не удалось установить подключение к {socket.RemoteEndPoint}");**

**}**

## NetworkStream

**System.Net.Sockets class NetworkStream : Stream**

**NetworkStream отличается от других классов Stream тем, что он не является буферизованным и не поддерживает перемещение в произвольную позицию с помощью метода Seek. Также при записи в поток не надо использовать метод Flush для сброса в поток всех данных.**

#### Конструктор:

**public NetworkStream (System.Net.Sockets.Socket socket);**

**// при этом сокет должен иметь SocketType.Stream, ProtocolType.Tcp**

**Мы можем создать поток NetworkStream лишь после подключения сокета к серверу.**

#### Пример:

**using System.Net.Sockets;**

**using var mySocket = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**// подключемся к удаленному серверу**

**mySocket.Connect("www.google.com", 80)**

**// создаем сетевой поток**

**using var stream = new NetworkStream(mySocket);**

**// либо используем конструкцию using,**

**// либо после завершения его необходимо закрыть stream.Close()**

**// получаем локальный адрес**

**Console.WriteLine($"Локальный адрес: {stream.Socket.LocalEndPoint}");**

**// получаем адрес сервера**

**Console.WriteLine($"Адрес сервера: {stream.Socket.RemoteEndPoint}");**

### Методы NetworkStream

* **Read()/ReadAsync(): считывает данные из потока в массив байтов и возвращает количество считанных байтов**
* **ReadByte(): считывает один байт из потока и возвращает его**
* **ReadAtLeast()/ReadAtLeastAsync(): считывает из потока как минимум определенное количество байтов (или больше) и возвращает количество считанных байтов**
* **ReadExactly()/ReadExactlyAsync(): считывает из потока точное количество байтов**
* **Write()/WriteAsync(): отправляет в поток данные в виде массива байтов**
* **WriteByte(): отправляет в поток один байт**

### Отправка данных

**public ValueTask WriteAsync (**

**ReadOnlyMemory<byte> buffer,**

**CancellationToken cancellationToken = default);**

**public Task WriteAsync (byte[] buffer, int offset, int count);**

**Отправляемые данные передаются в виде массива байтов, либо в виде объекта ReadOnlyMemory<byte>, который опять же инкапсулирует массив байтов.**

#### Пример:

**using System.Net.Sockets;**

**using System.Text;**

**using var mySocket =**

**new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);**

**var server = "www.google.com";**

**mySocket.Connect(server, 80);**

**using var stream = new NetworkStream(mySocket);**

**// отправляем сообщение для отправки**

**var message = $"GET / HTTP/1.1\r\nHost: {server}\r\nConnection: Close\r\n\r\n";**

**// кодируем его в массив байт**

**var data = Encoding.UTF8.GetBytes(message);**

**// отправляем массив байт на сервер**

**await stream.WriteAsync(data);**

**Console.WriteLine($"Данные отправлены на сервер {server}");**

### Получение данных

**public ValueTask<int> ReadAsync (**

**Memory<byte> buffer,**

**CancellationToken cancellationToken = default);**

**public Task<int> ReadAsync (**

**byte[] buffer,**

**int offset,**

**int count);**

**public Task<int> ReadAsync (**

**byte[] buffer,**

**int offset,**

**int count,**

**CancellationToken cancellationToken);**

**В качестве первого параметра метод получает буфер чтения в виде массива байт или объекта Memory<byte>, в которые считываются данные из потока. Дополнительно можно установить смещение в массиве и количество считываемых байтов.**

**В качестве результата метод неявно возвращает количество реально считанных байтов.**

#### Пример:

**using System.Net.Sockets;**

**using System.Text;**

**using var mySocket = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**var server = "www.google.com";**

**mySocket.Connect(server, 80);**

**using var stream = new NetworkStream(mySocket);**

**// отправляем сообщение для отправки**

**var message = $"GET / HTTP/1.1\r\nHost: {server}\r\nConnection: Close\r\n\r\n";**

**// кодируем его в массив байт**

**var data = Encoding.UTF8.GetBytes(message);**

**// отправляем массив байт на сервер**

**await stream.WriteAsync(data);**

**// буфер для получения данных**

**var responseData = new byte[512];**

**// StringBuilder для склеивания полученных данных в одну строку**

**var response = new StringBuilder();**

**int bytes; // количество полученных байтов**

**do**

**{**

**// получаем данные**

**bytes = await stream.ReadAsync(responseData);**

**// преобразуем в строку и добавляем ее в StringBuilder**

**response.Append(Encoding.UTF8.GetString(responseData, 0, bytes));**

**}**

**while (bytes > 0); // пока данные есть в потоке**

**// выводим данные на консоль**

**Console.WriteLine(response);**

##### ReadAtLeastAsync

**public ValueTask<int> ReadAtLeastAsync (**

**Memory<byte> buffer, // буфер для считывания данных**

**int minimumBytes, // минимум байт, которые надо считать**

**bool throwOnEndOfStream = true,**

**CancellationToken cancellationToken = default);**

#### Пример:

**// буфер ддя получения данных**

**var responseData = new byte[512];**

**// получаем данные**

**int bytes = await stream.ReadAtLeastAsync(responseData, 16);**

**// преобразуем в строку**

**var response = Encoding.UTF8.GetString(responseData, 0, bytes);**

**// выводим данные на консоль**

**Console.WriteLine(response);**

##### ReadExactly

**public ValueTask ReadExactlyAsync (**

**Memory<byte> buffer, // буфер для считывания данных**

**CancellationToken cancellationToken = default);**

**// количество байт, которые надо считать, берется из размера buffer**

**public ValueTask ReadExactlyAsync (**

**byte[] buffer, // буфер для считывания данных**

**int offset,**

**int count, // количество байт, которые надо считать**

**CancellationToken cancellationToken = default);**

#### Пример:

**int bytes = 16; // сколько надо получить данных**

**// буфер ддя получения данных**

**var responseData = new byte[bytes];**

**// получаем данные**

**await stream.ReadExactlyAsync(responseData);**

**// преобразуем в строку**

**var response = Encoding.UTF8.GetString(responseData);**

**// выводим данные на консоль**

**Console.WriteLine(response);**

## TCP-клиент. Класс TcpClient

Для создания tcp-клиента платформа .NET предоставляет класс класс **TcpClient**, который построен поверх сокетов и опять же использует сокет для отправки и получения данных, но при этом упрощает написание некоторых вещей.

### Создание TcpClient

**public TcpClient ();**

**public TcpClient (System.Net.Sockets.AddressFamily family);**

**public TcpClient (string hostname, int port);**

**public TcpClient (System.Net.IPEndPoint localEP);**

### Свойства TcpClient

* **Available: возвращает количество байтов данных, полученных из сети и доступных для чтения.**
* **Client: возвращает или задает объект Socket, который используется объектом TcpClient.**
* **Connected: возвращает true, если TcpClient подключен к удаленному узлу.**
* **LingerState: возвращает или устанавливает, доступен ли порт только одному клиенту.**
* **NoDelay: указывает, применяется ли задержка, когда буферы отправки и получения не заполнены. Если равно false, то TcpClient отправляет пакеты по сети только тогда, когда наберется достаточное количество данных. Это сделано, потому что отправка небольших кусочков данных может быть неэффективной и может привести к перегрузке. В то же время могут быть ситуации, когда необходимо отправлять небольшие объемы данных или когда необходимо побыстрее получить ответ.**
* **ReceiveBufferSize: возвращает или задает размер буфера приема (по умолчанию равно 65536).**
* **ReceiveTimeout: возвращает или задает длительность интервала, в течение которого объект TcpClient будет ожидать получение данных после начала операции чтения (по умолчанию равно 0).**
* **SendBufferSize: возвращает или задает размер буфера отправки (по умолчанию равно 65536).**
* **SendTimeout: возвращает или задает длительность интервала, в течение которого объект TcpClient будет ожидать успешное завершение отправки данных (по умолчанию равно 0).**

### Подключение к серверу

**public Task ConnectAsync (System.Net.IPEndPoint remoteEP);**

**public Task ConnectAsync (string host, int port);**

**public Task ConnectAsync (System.Net.IPAddress address, int port);**

#### Пример:

**using System.Net.Sockets;**

**using TcpClient tcpClient = new TcpClient();**

**try**

**{**

**// подключение к www.google.com**

**await tcpClient.ConnectAsync("www.google.com", 80);**

**Console.WriteLine("Подключение установлено");**

**}**

**catch(SocketException ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

### Закрытие TcpClient

**После окончания работы с TcpClient его надо закрыть методом Close() (или можно использовать конструкцию using)**

**tcpClient.Close(); // закрываем подключение**

### Отправка и получение данных

**Для отправки и получения данных TcpClient в общем случае использует рассмотренный в прошлой теме класс NetworkStream. Для получения объекта NetworkStream после подключения к серверу у TcpClient можно вызвать метод GetStream().**

#### Пример:

**using System.Net.Sockets;**

**using TcpClient tcpClient = new TcpClient();**

**await tcpClient.ConnectAsync("www.google.com", 80);**

**// получаем поток для взаимодействия с сервером**

**NetworkStream stream = tcpClient.GetStream();**

**NetworkStream будет использовать для отправки и получения данных тот же объект Socket, что и TcpClient. Если закрывается объект TcpClient, то вместе с ним автоматически закрывается и связанный с ним объект NetworkStream (а также освобождается связанный с ними сокет).**

### Отправка данных

**public ValueTask NetworkStream.WriteAsync (**

**ReadOnlyMemory<byte> buffer,**

**CancellationToken cancellationToken = default);**

**public Task NetworkStream.WriteAsync (byte[] buffer, int offset, int count);**

**Отправляемые данные передаются в виде массива байтов, либо в виде объекта ReadOnlyMemory<byte>, который опять же инкапсулирует массив байтов.**

**using System.Net.Sockets;**

**using System.Text;**

**using TcpClient tcpClient = new TcpClient();**

**await tcpClient.ConnectAsync("www.google.com", 80);**

**// получаем поток для взаимодействия с сервером**

**NetworkStream stream = tcpClient.GetStream();**

**// определяем отправляемые данные**

**var requestMessage = $"GET / HTTP/1.1\r\nHost: www.google.com\r\nConnection: Close\r\n\r\n";**

**// конвертируем данные в массив байтов**

**var requestData = Encoding.UTF8.GetBytes(requestMessage);**

**// отправляем данные серверу**

**await stream.WriteAsync(requestData);**

### Получение данных

**Для получения данных к NetworkStream применяется методы Read()/ReadAsync().**

**using System.Net.Sockets;**

**using System.Text;**

**using System;**

**using TcpClient tcpClient = new TcpClient();**

**await tcpClient.ConnectAsync("www.google.com", 80);**

**// получаем поток для взаимодействия с сервером**

**NetworkStream stream = tcpClient.GetStream();**

**/\* ОТПРАВКА ДАННЫХ \*/**

**// определяем отправляемые данные**

**var requestMessage = $"GET / HTTP/1.1\r\nHost: www.google.com\r\nConnection: Close\r\n\r\n";**

**// конвертируем данные в массив байтов**

**var requestData = Encoding.UTF8.GetBytes(requestMessage);**

**// отправляем данные серверу**

**await stream.WriteAsync(requestData);**

**/\* ПОЛУЧЕНИЕ ДАННЫХ \*/**

**// буфер для получения данных**

**var responseData = new byte[512];**

**// StringBuilder для склеивания полученных данных в одну строку**

**var response = new StringBuilder();**

**int bytes; // количество полученных байтов**

**do**

**{**

**// получаем данные**

**bytes = await stream.ReadAsync(responseData);**

**// преобразуем в строку и добавляем ее в StringBuilder**

**response.Append(Encoding.UTF8.GetString(responseData, 0, bytes));**

**}**

**while (bytes > 0); // пока есть данные в потоке**

**// выводим данные на консоль**

**Console.WriteLine(response);**

### Свойства TcpClient.Available и NetworkStream.DataAvailable и считывание данных

**В реальности оба этих свойства (TcpClient.Available и NetworkStream.DataAvailable) смотрят на значение свойства Available используемого объекта Socket, которое возвращает количество доступных для чтения байтов. И соответственно свойство Available будет иметь ненулевое значение, если в текущий момент есть доступные данные. Но природа протокола TCP такова, что крупные наборы данных отправляются отдельными пакетами. Какой-то пакет может прийти быстрее, какой-то задержится, какой-то будет потерян, и потребуется переотправка. Поэтому может возникнуть ситуация, что сервер отправил данные, часть данных пришла. В какой-то момент свойство Available у сокета возратило 0, соответственно произошел выход из цикла.**

## Отправка и получение данных в TCP. Однонаправленная связь между сокетами

**Однонаправленная связь между сокетом-клиентом и сокетом-сервером: либо сервер посылает данные, а клиент получает, либо клиент отправляет данные, а сервер получает.**

### Отправка данных клиенту

#### На стороне сервера:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**IPEndPoint endPoint = new IPEndPoint(IPAddress.Any, 8888);**

**using Socket server = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**server.Bind(endPoint);**

**server.Listen(); // запускаем сервер**

**while (true)**

**{**

**// получаем входящее подключение**

**using Socket client = await server.AcceptAsync();**

**// определяем данные для отправки - текущее время**

**byte[] data = Encoding.UTF8.GetBytes(DataTime.Now.ToLongTimeString());**

**// отправляем данные**

**await client.SendAsync(data);**

**Console.WriteLine($"Данные отправлены клиенту” +**

**“{client.RemoteEndPoint}");**

**}**

**}**

**catch(Exception ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

#### На стороне клиента:

**using System.Net.Sockets;**

**using System.Text;**

**using var tcpClient = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**await tcpClient.ConnectAsync("127.0.0.1", 8888);**

**// буфер для считывания данных**

**byte[] data = new byte[512];**

**// получаем данные из потока**

**int bytes = await tcpClient.ReceiveAsync(data);**

**// получаем отправленное время**

**string time = Encoding.UTF8.GetString(data, 0, bytes);**

**Console.WriteLine($"Текущее время: {time}");**

**}**

**catch (Exception ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

### Получение данных от клиента

#### На стороне сервера:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**IPEndPoint ipPoint = new IPEndPoint(IPAddress.Any, 8888);**

**using Socket server = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**server.Bind(ipPoint);**

**server.Listen(); // запускаем сервер**

**Console.WriteLine("Сервер запущен. Ожидание подключений... ");**

**while (true)**

**{**

**// получаем подключение в виде Socket**

**using Socket client = await server.AcceptAsync();**

**// определяем буфер для получения данных**

**List<byte> response = [];**

**byte[] buffer = new byte[512];**

**int bytes = 0; // количество считанных байтов**

**// считываем данные**

**do**

**{**

**bytes = await client.ReceiveAsync(buffer);**

**// добавляем полученные байты в список**

**response.AddRange(buffer.Take(bytes));**

**}**

**while (bytes > 0);**

**// выводим отправленные клиентом данные**

**var responseText = Encoding.UTF8.GetString(response.ToArray());**

**Console.WriteLine(responseText);**

**}**

**}**

**catch(Exception ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

#### На стороне клиента:

**using System.Net.Sockets;**

**using System.Text;**

**using var tcpClient = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**await tcpClient.ConnectAsync("127.0.0.1", 8888);**

**// сообщение для отправки**

**var message = "Hello METANIT.COM";**

**// считыванием строку в массив байт**

**byte[] requestData = Encoding.UTF8.GetBytes(message);**

**// отправляем данные**

**await tcpClient.SendAsync(requestData);**

**Console.WriteLine("Сообщение отправлено");**

**}**

**catch (Exception ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

#### Стратегии получения данных

* **Использование буфера фиксированной длины, когда мы точно знаем, какой именно объем данных будет послан**
* **Отправка в ответе информации о размере ответа, получив которую, нам будет проще считать нужное количество байтов**
* **Использование маркера окончания ответа, получив который, мы завершим считывание данных**

### Использование маркера окончания ответа

#### На стороне сервера:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**using Socket server = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**server.Bind(new IPEndPoint(IPAddress.Any, 8888));**

**server.Listen(); // запускаем сервер**

**Console.WriteLine("Сервер запущен. Ожидание подключений... ");**

**while (true)**

**{**

**// получаем подключение в виде Socket**

**using Socket client = await server.AcceptAsync();**

**// буфер для накопления входящих данных**

**var buffer = new List<byte>();**

**// буфер для считывания одного байта**

**var bytesRead = new byte[1];**

**// считываем данные до конечного символа**

**while (true)**

**{**

**var count = await client.ReceiveAsync(bytesRead);**

**// смотрим, если считанный байт представляет конечный символ, выходим**

**if (count == 0 || bytesRead[0] == '\n') break;**

**// иначе добавляем в буфер**

**buffer.Add(bytesRead[0]);**

**}**

**var message = Encoding.UTF8.GetString(buffer.ToArray());**

**Console.WriteLine($"Получено сообщение: {message}");**

**}**

**}**

**catch(Exception ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

#### На стороне клиента:

**using System.Net.Sockets;**

**using System.Text;**

**using var tcpClient = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**await tcpClient.ConnectAsync("127.0.0.1", 8888);**

**// сообщение для отправки**

**// сообщение завершается конечным символом - \n,**

**// который символизирует окончание сообщения**

**var message = "Hello METANIT.COM\n";**

**// считыванием строку в массив байт**

**byte[] requestData = Encoding.UTF8.GetBytes(message);**

**// отправляем данные**

**await tcpClient.SendAsync(requestData);**

**Console.WriteLine("Сообщение отправлено");**

**}**

**catch (Exception ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

### Установка размера сообщения

#### На стороне сервера:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**using Socket tcpListener = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**tcpListener.Bind(new IPEndPoint(IPAddress.Any, 8888));**

**tcpListener.Listen(); // запускаем сервер**

**Console.WriteLine("Сервер запущен. Ожидание подключений... ");**

**while (true)**

**{**

**// получаем подключение в виде Socket**

**using Socket client = await tcpListener.AcceptAsync();**

**// буфер для считывания размера данных**

**byte[] sizeBuffer = new byte[4];**

**// сначала считываем размер данных**

**await client.ReceiveAsync(sizeBuffer);**

**// узнаем размер и создаем соответствующий буфер**

**int size = BitConverter.ToInt32(sizeBuffer, 0);**

**// создаем соответствующий буфер**

**byte[] data = new byte[size];**

**// считываем собственно данные**

**int bytes = await client.ReceiveAsync(data);**

**var message = Encoding.UTF8.GetString(data, 0, bytes);**

**Console.WriteLine($"Размер сообщения: {size} байтов");**

**Console.WriteLine($"Сообщение: {message}");**

**}**

**}**

**catch(Exception ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

#### На стороне клиента:

**using System.Net.Sockets;**

**using System.Text;**

**using var tcpClient = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**await tcpClient.ConnectAsync("127.0.0.1", 8888);**

**// сообщение для отправки**

**var message = "Hello METANIT.COM";**

**// считыванием строку в массив байт**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**// определяем размер данных**

**byte[] size = BitConverter.GetBytes(data.Length);**

**// отправляем размер данных**

**await tcpClient.SendAsync(size);**

**// отправляем данные**

**await tcpClient.SendAsync(data);**

**Console.WriteLine("Сообщение отправлено");**

**}**

**catch (Exception ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

### Множественная отправка и получение

#### На стороне сервера:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**IPEndPoint ipPoint = new IPEndPoint(IPAddress.Any, 8888);**

**using Socket server = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**server.Bind(ipPoint);**

**server.Listen();**

**Console.WriteLine("Сервер запущен. Ожидание подключений...");**

**while (true)**

**{**

**// получаем подключение в виде Socket**

**using Socket client = await server.AcceptAsync();**

**// List для накопления входящих данных**

**var list = new List<byte>();**

**// буфер для считывания одного байта**

**var buffer = new byte[1];**

**// считываем данные до конечного символа**

**while (true)**

**{**

**int count = await server.ReceiveAsync(buffer);**

**// смотрим, если считанный байт представляет конечный символ, выходим**

**if (count == 0 || buffer[0] == '\n') break;**

**// иначе добавляем в List**

**list.Add(buffer[0]);**

**}**

**string message = Encoding.UTF8.GetString(list.ToArray());**

**// если прислан маркер окончания взаимодействия,**

**// выходим из цикла и завершаем взаимодействие с клиентом**

**if (message == "END") break;**

**Console.WriteLine($"Получено сообщение: {message}");**

**list.Clear();**

**}**

**}**

**catch (Exception ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

#### На стороне клиента:

**using var client = new Socket**

**(AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**await client.ConnectAsync("127.0.0.1", 8888);**

**// сообщения для отправки**

**// сообщение завершается конечным символом - \n,**

**// который символизирует окончание сообщения**

**var messages = new string[]**

**{**

**"Hello METANIT.COM\n",**

**"Hello TcpListener\n",**

**"Bye METANIT.COM\n",**

**"END\n"**

**};**

**foreach (var message in messages)**

**{**

**// считыванием строку в массив байт**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**// отправляем данные**

**await client.SendAsync(data);**

**}**

**Console.WriteLine("Все сообщения отправлены");**

**}**

**catch (SocketException ex)**

**{**

**Console.WriteLine(ex.Message);**

**}**

## Отправка и получение данных. Однонаправленная связь между TcpListener и TcpClient

### Отправка данных клиенту

#### На стороне сервера:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**var tcpListener = new TcpListener(IPAddress.Any, 8888);**

**try**

**{**

**tcpListener.Start(); // запускаем сервер**

**Console.WriteLine("Сервер запущен. Ожидание подключений... ");**

**while (true)**

**{**

**// получаем подключение в виде TcpClient**

**using var tcpClient = await tcpListener.AcceptTcpClientAsync();**

**// получаем объект NetworkStream для взаимодействия с клиентом**

**var stream = tcpClient.GetStream();**

**// определяем данные для отправки - отправляем текущее время**

**byte[] data =**

**Encoding.UTF8.GetBytes(DateTime.Now.ToLongTimeString());**

**// отправляем данные**

**await stream.WriteAsync(data);**

**Console.WriteLine($"Клиенту {tcpClient.Client.RemoteEndPoint}" +**

**"отправлены данные");**

**}**

**}**

**finally**

**{**

**tcpListener.Stop();**

**}**

#### На стороне клиента:

**using System.Net.Sockets;**

**using System.Text;**

**using TcpClient tcpClient = new TcpClient();**

**await tcpClient.ConnectAsync("127.0.0.1", 8888);**

**// буфер для считывания данных**

**byte[] data = new byte[512];**

**// получаем NetworkStream для взаимодействия с сервером**

**var stream = tcpClient.GetStream();**

**// получаем данные из потока**

**int bytes = await stream.ReadAsync(data);**

**// получаем отправленное время**

**string time = Encoding.UTF8.GetString(data, 0, bytes);**

**Console.WriteLine($"Текущее время: {time}");**

### Получение данных от клиента

#### На стороне сервера:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**var tcpListener = new TcpListener(IPAddress.Any, 8888);**

**try**

**{**

**tcpListener.Start(); // запускаем сервер**

**Console.WriteLine("Сервер запущен. Ожидание подключений... ");**

**while (true)**

**{**

**// получаем подключение в виде TcpClient**

**using var tcpClient = await tcpListener.AcceptTcpClientAsync();**

**// получаем объект NetworkStream для взаимодействия с клиентом**

**var stream = tcpClient.GetStream();**

**// определяем буфер для получения данных**

**byte[] data = new byte[1024];**

**int bytes = 0; // количество считанных байтов**

**var builder = new StringBuilder(); // для склеивания данных в строку**

**// считываем данные**

**do**

**{**

**bytes = await stream.ReadAsync(data);**

**builder.Append(Encoding.UTF8.GetString(data, 0, bytes));**

**}**

**while(bytes > 0);**

**// выводим отправленные клиентом данные**

**Console.WriteLine(builder);**

**}**

**}**

**finally**

**{**

**tcpListener.Stop(); // останавливаем сервер**

**}**

#### На стороне клиента:

**using System.Net.Sockets;**

**using System.Text;**

**using TcpClient tcpClient = new TcpClient();**

**await tcpClient.ConnectAsync("127.0.0.1", 8888);**

**// сообщение для отправки**

**var message = "Hello METANIT.COM";**

**// считыванием строку в массив байт**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**// получаем NetworkStream для взаимодействия с сервером**

**var stream = tcpClient.GetStream();**

**// отправляем данные**

**await stream.WriteAsync(data);**

**Console.WriteLine("Сообщение отправлено");**

#### Стратегии получения данных

* **Использование буфера фиксированной длины, когда мы точно знаем, какой именно объем данных будет послан**
* **Отправка в ответе информации о размере ответа, получив которую, нам будет проще считать нужное количество байтов**
* **Использование маркера окончания ответа, получив который, мы завершим считывание данных**

### Использование буфера фиксированной длины

**Эта стратегия очевидная - используем массив фиксированной известной длины, в который запишем полученный ответ.**

### Использование маркера окончания ответа

#### На стороне сервера:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**var tcpListener = new TcpListener(IPAddress.Any, 8888);**

**try**

**{**

**tcpListener.Start(); // запускаем сервер**

**Console.WriteLine("Сервер запущен. Ожидание подключений... ");**

**while (true)**

**{**

**// получаем подключение в виде TcpClient**

**using var tcpClient = await tcpListener.AcceptTcpClientAsync();**

**// получаем объект NetworkStream для взаимодействия с клиентом**

**var stream = tcpClient.GetStream();**

**// буфер для входящих данных**

**var buffer = new List<byte>();**

**int byteRead = '\n';**

**// считываем данные до конечного символа**

**while((byteRead = stream.ReadByte())!='\n')**

**{**

**// добавляем в буфер**

**buffer.Add((byte)byteRead);**

**}**

**var message = Encoding.UTF8.GetString(buffer.ToArray());**

**Console.WriteLine($"Получено сообщение: {message}");**

**}**

**}**

**finally**

**{**

**tcpListener.Stop(); // останавливаем сервер**

**}**

#### На стороне клиента:

**using System.Net.Sockets;**

**using System.Text;**

**using TcpClient tcpClient = new TcpClient();**

**await tcpClient.ConnectAsync("127.0.0.1", 8888);**

**// сообщение для отправки**

**// сообщение завершается конечным символом - \n,**

**// который символизирует окончание сообщения**

**var message = "Hello METANIT.COM\n";**

**// получаем NetworkStream для взаимодействия с сервером**

**var stream = tcpClient.GetStream();**

**// считыванием строку в массив байт**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**// отправляем данные**

**await stream.WriteAsync(data);**

**Console.WriteLine("Сообщение отправлено");**

### Установка размера сообщения

#### На стороне сервера:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**var tcpListener = new TcpListener(IPAddress.Any, 8888);**

**try**

**{**

**tcpListener.Start(); // запускаем сервер**

**Console.WriteLine("Сервер запущен. Ожидание подключений... ");**

**while (true)**

**{**

**// получаем подключение в виде TcpClient**

**using var tcpClient = await tcpListener.AcceptTcpClientAsync();**

**// получаем объект NetworkStream для взаимодействия с клиентом**

**var stream = tcpClient.GetStream();**

**// буфер для считывания размера данных**

**byte[] buffer = new byte[4];**

**// сначала считываем размер данных**

**await stream.ReadExactlyAsync(buffer, 0, buffer.Length);**

**// узнаем размер и создаем соответствующий буфер**

**int size = BitConverter.ToInt32(buffer, 0);**

**// создаем соответствующий буфер**

**byte[] data = new byte[size];**

**// считываем собственно данные**

**int bytes = await stream.ReadAsync(data);**

**var message = Encoding.UTF8.GetString(data, 0, bytes);**

**Console.WriteLine($"Размер сообщения: {size} байтов");**

**Console.WriteLine($"Сообщение: {message}");**

**}**

**}**

**finally**

**{**

**tcpListener.Stop(); // останавливаем сервер**

**}**

#### На стороне клиента:

**using System.Net.Sockets;**

**using System.Text;**

**using TcpClient tcpClient = new TcpClient();**

**await tcpClient.ConnectAsync("127.0.0.1", 8888);**

**// сообщение для отправки**

**var message = "Hello METANIT.COM";**

**// получаем NetworkStream для взаимодействия с сервером**

**var stream = tcpClient.GetStream();**

**// считыванием строку в массив байт**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**// определяем размер данных**

**byte[] size = BitConverter.GetBytes(data.Length);**

**// отправляем размер данных**

**await stream.WriteAsync(size);**

**// отправляем данные**

**await stream.WriteAsync(data);**

**Console.WriteLine("Сообщение отправлено");**

### Множественная отправка и получение

#### На стороне сервера:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**var tcpListener = new TcpListener(IPAddress.Any, 8888);**

**try**

**{**

**tcpListener.Start();**

**Console.WriteLine("Сервер запущен. Ожидание подключений... ");**

**while (true)**

**{**

**// получаем подключение в виде TcpClient**

**using var tcpClient = await tcpListener.AcceptTcpClientAsync();**

**// получаем объект NetworkStream для взаимодействия с клиентом**

**var stream = tcpClient.GetStream();**

**// буфер для входящих данных**

**var buffer = new List<byte> ();**

**int bytesRead = 10;**

**while(true)**

**{**

**// считываем данные до конечного символа**

**while ((bytesRead = stream.ReadByte()) != '\n')**

**{**

**// добавляем в буфер**

**buffer.Add((byte)bytesRead);**

**}**

**var message = Encoding.UTF8.GetString(buffer.ToArray());**

**// если прислан маркер окончания взаимодействия,**

**// выходим из цикла и завершаем взаимодействие с клиентом**

**if (message == "END") break;**

**Console.WriteLine($"Получено сообщение: {message}");**

**buffer.Clear();**

**}**

**}**

**}**

**finally**

**{**

**tcpListener.Stop();**

**}**

#### На стороне клиента:

**using System.Net.Sockets;**

**using System.Text;**

**using TcpClient tcpClient = new TcpClient();**

**await tcpClient.ConnectAsync("127.0.0.1", 8888);**

**// сообщения для отправки**

**// сообщение завершается конечным символом - \n,**

**// который символизирует окончание сообщения**

**var messages = new string[]**

**{**

**"Hello METANIT.COM\n",**

**"Hello Tcplistener\n",**

**"Bye METANIT.COM\n",**

**"END\n"**

**};**

**// получаем NetworkStream для взаимодействия с сервером**

**var stream = tcpClient.GetStream();**

**foreach (var message in messages)**

**{**

**// считыванием строку в массив байт**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**// отправляем данные**

**await stream.WriteAsync(data);**

**}**

**Console.WriteLine("Все сообщения отправлены");**

## Отправка и получение данных в TCP. Двунаправленная связь

### Пример с TcpListener и TcpClient

#### Со стороны сервера:

**var server = new TcpListener(IPAddress.Any, 8888); // создаем сервер**

**var words = new Dictionary<string, string>**

**{**

**{ "red", "красный" },**

**{ "green", "зеленый" },**

**{ "blue", "синий" }**

**};**

**try**

**{**

**server.Start(); // запускаем сервер**

**Console.WriteLine("Server: Сервер запущен. Ожидание подключений...");**

**// ожидаем очередного клиента**

**while (true)**

**{**

**// получаем подключение в виде TcpClient**

**using TcpClient client = await server.AcceptTcpClientAsync();**

**// получаем объект NetworkStream для взаимодействия с клиентом**

**NetworkStream stream = client.GetStream();**

**// буфер для входящих данных**

**var responseList = new List<byte>();**

**int byteRead = 10; // для считывания байтов из потока**

**// получили клиента, запускаем цикл для считывания данных из потока**

**while (true)**

**{**

**// считываем строку (до '\n')**

**while ((byteRead = stream.ReadByte()) != '\n')**

**{**

**// добавляем в буфер по одному байту**

**responseList.Add((byte)byteRead);**

**}**

**string word = Encoding.UTF8.GetString(responseList.ToArray());**

**// если прислан маркер окончания взаимодействия,**

**// выходим из цикла и завершаем взаимодействие с клиентом**

**if (word == "END") break;**

**Console.WriteLine($"Server: Запрошен перевод слова {word}");**

**// находим слово в словаре и отправляем ответ клиенту**

**if (!words.TryGetValue(word, out var translation))**

**translation = "Не найдено в словаре";**

**// добавляем символ окончания сообщения**

**translation += '\n';**

**// отправляем перевод слова**

**await stream.WriteAsync(Encoding.UTF8.GetBytes(translation));**

**responseList.Clear();**

**}**

**}**

**}**

**finally { server.Stop(); }**

#### Со стороны клиента:

**using TcpClient client = new TcpClient();**

**await client.ConnectAsync("127.0.0.1", 8888);**

**// слова для отправки и для получения перевода**

**var words = new string[] { "red", "yellow", "blue" };**

**// получаем NetworkStream для взаимодействия с сервером**

**var stream = client.GetStream();**

**// буфер для входящих данных**

**var responseList = new List<byte>();**

**int byteRead = 10; // для считывания байтов из потока**

**foreach (var word in words)**

**{**

**// считываем строку в массив байт**

**// при отправке добавляем маркер '\n' завершения сообщения**

**byte[] data = Encoding.UTF8.GetBytes(word + '\n');**

**// отправляем данные**

**await stream.WriteAsync(data);**

**// считываем данные до конечного символа**

**while ((byteRead = stream.ReadByte()) != '\n')**

**{**

**// добавляем в буфер**

**responseList.Add((byte)byteRead);**

**}**

**var translation = Encoding.UTF8.GetString(responseList.ToArray());**

**Console.WriteLine($"Client: Слово {word}: {translation}");**

**responseList.Clear();**

**}**

**// отправляем маркер завершения подключения "END"**

**await stream.WriteAsync(Encoding.UTF8.GetBytes("END\n"));**

**Console.WriteLine("Client: Все сообщения отправлены");**

### Пример на сокетах

#### Со стороны сервера:

**using Socket server = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**var words = new Dictionary<string, string>**

**{**

**{ "red", "красный" },**

**{ "blue", "синий" },**

**{ "green", "зеленый" }**

**};**

**try**

**{**

**server.Bind(new IPEndPoint(IPAddress.Any, 8888));**

**server.Listen(); // запускаем сервер**

**Console.WriteLine("Server: Сервер запущен. Ожидание подключений...");**

**while (true)**

**{**

**// получаем подключение в виде TcpClient**

**using var client = await server.AcceptAsync();**

**// буфер для накопления входящих данных**

**var responseList = new List<byte>();**

**// буфер для считывания одного байта**

**var bytesRead = new byte[1];**

**while (true)**

**{**

**// считываем данные до конечного символа**

**while (true)**

**{**

**int count = client.Receive(bytesRead);**

**// смотрим, представляет ли считанный байт конечный символ**

**if (count == 0 || bytesRead[0] == '\n') break;**

**// если не конечный символ, то добавляем в буфер**

**responseList.Add(bytesRead[0]);**

**}**

**var word = Encoding.UTF8.GetString(responseList.ToArray());**

**// если прислан маркер окончания взаимодействия,**

**// выходим из цикла и завершаем взаимодействие с клиентом**

**if (word == "END") break;**

**Console.WriteLine($"Server: Запрошен перевод слова {word}");**

**// находим слово в словаре и отправляем клиенту ответ**

**if (!words.TryGetValue(word, out var translation))**

**translation = "Не найдено в словаре";**

**// добавляем символ окончания сообщения**

**translation += '\n';**

**// отправляем перевод слова**

**await client.SendAsync(Encoding.UTF8.GetBytes(translation));**

**responseList.Clear();**

**}**

**}**

**}**

**catch(Exception ex) { Console.WriteLine(ex.Message); }**

#### Со стороны клиента:

**{**

**// слова для отправки и получения перевода**

**var words = new string[] { "red", "yellow", "blue" };**

**using var client = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**await client.ConnectAsync("127.0.0.1", 8888);**

**// буфер для входящих данных**

**var responseList = new List<byte>();**

**foreach (var word in words)**

**{**

**// считываем строку в массив байт**

**// при отправке добавляем маркер завершения сообщения '\n'**

**byte[] data = Encoding.UTF8.GetBytes(word + '\n');**

**// отправляем данные**

**await client.SendAsync(data);**

**// буфер для считывания одного байта**

**var bytesRead = new byte[1];**

**// считываем данные до конечного символа**

**while (true)**

**{**

**int count = client.Receive(bytesRead);**

**// смотрим, представляет ли считанный байт конечный символ '\n'**

**if (count == 0 || bytesRead[0] == '\n') break;**

**// если не конечный символ, то добавляем в буфер**

**responseList.Add(bytesRead[0]);**

**}**

**var translation = Encoding.UTF8.GetString(responseList.ToArray());**

**Console.WriteLine($"Client: Слово {word}: {translation}");**

**responseList.Clear();**

**}**

**// отправляем маркер завершения подключения "END"**

**await client.SendAsync(Encoding.UTF8.GetBytes("END\n"));**

**Console.WriteLine("Client: Все сообщения отправлены");**

**}**

**catch (Exception ex) { Console.WriteLine(ex.Message); }**

## Многопоточное клиент-серверное приложение TCP

### Пример с TcpListener и TcpClient

#### На стороне сервера:

**var server = new TcpListener(IPAddress.Any, 8888);**

**try**

**{**

**server.Start(); // запускаем сервер**

**Console.WriteLine("SERVER: Сервер запущен. Ожидание подключений...");**

**while (true)**

**{**

**// получаем подключение в виде TcpClient**

**TcpClient client = await server.AcceptTcpClientAsync();**

**// создаем новую задачу для обслуживания нового клиента**

**Task.Run(async () => await ProcessClientAsync(client));**

**// либо вместо задач можно использовать стандартный Thread**

**//new Thread(async () => await ProcessClientAsync(client)).Start();**

**}**

**}**

**finally { server.Stop(); }**

***, где метод ProcessClientAsync:***

**// обрабатываем клиента**

**async static Task ProcessClientAsync(TcpClient client)**

**{**

**// условный словарь**

**var words = new Dictionary<string, string>**

**{**

**{ "red", "красный" },**

**{ "green", "зеленый" },**

**{ "blue", "синий" }**

**};**

**NetworkStream stream = client.GetStream();**

**// буфер для входящий данных**

**var responseList = new List<byte>();**

**int byteRead = 10;**

**while (true)**

**{**

**// считываем данные до конечного символа**

**while ((byteRead = stream.ReadByte()) != '\n')**

**{**

**// добавляем в буфер**

**responseList.Add((byte)byteRead);**

**}**

**string word = Encoding.UTF8.GetString(responseList.ToArray());**

**// если прислан маркер окончания взаимодействия,**

**// выходим из цикла и завершаем взаимодействие с клиентом**

**if (word == "END") break;**

**Console.WriteLine($"SERVER: Клиент {client.Client.RemoteEndPoint}"**

**+ $" запросил перевод слова {word}");**

**// находим слово в словаре и отправляем обратно клиенту**

**if (!words.TryGetValue(word, out string translation))**

**translation = "не найдено в словаре";**

**// добавляем символ окончания сообщения**

**translation += '\n';**

**// отправляем перевод слова из словаря**

**await stream.WriteAsync(Encoding.UTF8.GetBytes(translation));**

**responseList.Clear();**

**}**

**}**

#### На стороне клиента:

**// слова для отправки и получения перевода**

**var words = new string[] { "red", "yellow", "blue", "green" };**

**using TcpClient client = new TcpClient();**

**await client.ConnectAsync("127.0.0.1", 8888);**

**NetworkStream stream = client.GetStream();**

**// буфер для входящих данных**

**var responseList = new List<byte>();**

**int byteRead = 10; // для считывания байтов из потока**

**foreach (var word in words)**

**{**

**// считываем строку в массив байтов**

**// при отправке добавляем маркер завершения сообщения '\n'**

**byte[] data = Encoding.UTF8.GetBytes(word + '\n');**

**// отправляем данные**

**await stream.WriteAsync(data);**

**// получаем данные - считываем до конечного символа**

**while ((byteRead = stream.ReadByte()) != '\n')**

**{**

**// добавляем в буфер**

**responseList.Add((byte)byteRead);**

**}**

**string translation = Encoding.UTF8.GetString(responseList.ToArray());**

**Console.WriteLine($"CLIENT: Слово {word}: {translation}");**

**responseList.Clear();**

**// имитируем долговременную работу,**

**// чтобы несколько клиентов обрабатывались параллельно**

**await Task.Delay(3000);**

**}**

**// отправляем маркер завершения подключения "END"**

**await stream.WriteAsync(Encoding.UTF8.GetBytes("END\n"));**

### Многопоточое клиент-серверное приложение на сокетах

#### На стороне сервера:

**using Socket server = new Socket**

**(AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**server.Bind(new IPEndPoint(IPAddress.Any, 8888));**

**server.Listen(); // запускаем сервер**

**Console.WriteLine("SERVER: Сервер запущен. Ожидание подключений...");**

**while (true)**

**{**

**// получаем подключение в виде Socket**

**Socket client = await server.AcceptAsync();**

**// создаем новую задачу для обслуживания нового клиента**

**Task.Run(async () => await ProcessClientAsync(client));**

**// вместо задач можно использовать стандартный Thread**

**//new Thread(async () => await ProcessClientAsync(client)).Start();**

**}**

**}**

**catch (Exception ex) { Console.WriteLine(ex.Message); }**

***, где метод ProcessClientAsync:***

**// обрабатываем клиента**

**async static Task ProcessClientAsync(Socket client)**

**{**

**// условный словарь**

**var words = new Dictionary<string, string>**

**{**

**{ "red", "красный" },**

**{ "green", "зеленый" },**

**{ "blue", "синий" }**

**};**

**// буфер для накопления входящих данных**

**var responseList = new List<byte>();**

**// буфер для считывания одного байта**

**var byteRead = new byte[1];**

**while (true)**

**{**

**// считываем данные до конечного символа**

**while (true)**

**{**

**int count = await client.ReceiveAsync(byteRead);**

**// смотрим - если считанный байт - '\n', выходим из цикла**

**if (count == 0 || byteRead[0] == '\n') break;**

**// иначе добавляем в буфер**

**responseList.Add(byteRead[0]);**

**}**

**string word = Encoding.UTF8.GetString(responseList.ToArray());**

**// если прислан маркер окончания взаимодействия "END",**

**// выходим из цикла и завершаем взаимодействие с клиентом**

**if (word == "END") break;**

**Console.WriteLine($"SERVER: Запрошен перевод слова {word}");**

**// находим слово в словаре и отправляем обратно клиенту**

**if (!words.TryGetValue(word, out string translation))**

**translation = "не найдено в словаре";**

**// добавляем символ окончания сообщения**

**translation += '\n';**

**// отправляем перевод слова из словаря**

**await client.SendAsync(Encoding.UTF8.GetBytes(translation));**

**responseList.Clear();**

**}**

**client.Shutdown(SocketShutdown.Both);**

**client.Close();**

**}**

#### На стороне клиента:

**// слова для отправки и получения перевода**

**var words = new string[] { "red", "yellow", "blue" };**

**using var client = new Socket**

**(AddressFamily.InterNetwork,**

**SocketType.Stream,**

**ProtocolType.Tcp);**

**try**

**{**

**await client.ConnectAsync("127.0.0.1", 8888);**

**// буфер для входящих данных**

**var responseList = new List<byte>();**

**foreach (var word in words)**

**{**

**// конвертируем строку в массив байт**

**// при отправке добавляем маркер завершения сообщения '\n'**

**byte[] data = Encoding.UTF8.GetBytes(word + '\n');**

**// отправляем данные**

**await client.SendAsync(data);**

**// буфер для считывания одного байта**

**var byteRead = new byte[1];**

**// считываем данные до конечного символа**

**while (true)**

**{**

**int count = await client.ReceiveAsync(byteRead);**

**// если считанный байт представляет конечный символ '\n',**

**// выходим из цикла**

**if (count == 0 || byteRead[0] == '\n') break;**

**// иначе добавляем в буфер**

**responseList.Add(byteRead[0]);**

**}**

**string translation = Encoding.UTF8.GetString(responseList.ToArray());**

**Console.WriteLine($"CLIENT: Слово {word}: {translation}");**

**responseList.Clear();**

**// имитируем долговременную работу,**

**// чтобы несколько клиентов обрабатывались параллельно**

**await Task.Delay(2000);**

**}**

**// отправляем маркер завершения подключения "END"**

**await client.SendAsync(Encoding.UTF8.GetBytes("END\n"));**

**}**

**catch (Exception ex) { Console.WriteLine(ex.Message); }**

## NetworkStream и текстовые потоки

**using System; // Console**

**using System.IO; // StreamWriter**

**using System.Net.Sockets; // TcpClient**

**// создаем клиента и соединяем его с сервером google**

**using TcpClient tcpClient = new TcpClient();**

**var server = "www.google.com";**

**await tcpClient.ConnectAsync(server, 80);**

**// получаем поток**

**var stream = tcpClient.GetStream();**

**var message = $"GET / HTTP/1.1\r\n" +**

**"Host: {server}\r\nConnection: Close\r\n\r\n";**

**// создаем объект класса текстового потока StreamWriter**

**using var writer = new StreamWriter(stream);**

**// отправляем сообщение**

**await writer.WriteAsync(message);**

**await writer.FlushAsync();**

**// создаем объект класса текстового потока StreamReader**

**using var reader = new StreamReader(stream);**

**// считываем первую строку**

**var response = await reader.ReadToEndAsync();**

**// или считываем ответ до конца, но только, если будет конец.**

**// Т.е. в данном случае сервер google закроет подключение**

**// по окончании сообщения, потому что в нашем сообщении**

**// указан запрос на это: Connection: Close**

**var fullResponse = await reader.ReadToEndAsync();**

**Console.WriteLine(response); // HTTP/1.1 200 OK**

### Клиент-серверное взаимодействие

#### На стороне сервера:

**using System; // Console**

**using System.IO; // StreamWriter, StreamReader**

**using System.Collections.Generic; // Dictionary**

**using System.Net; // IPAddress**

**using System.Net.Sockets; // TcpListener**

**// создаем сервер**

**var server = new TcpListener(IPAddress.Any, 8888);**

**var words = new Dictionary<string, string>**

**{**

**{ "red", "красный" },**

**{ "blue", "синий" },**

**{ "green", "зеленый" }**

**};**

**try**

**{**

**server.Start(); // запускаем сервер**

**Console.WriteLine("Сервер запущен. Ожидание подключений...");**

**while (true)**

**{**

**// получаем подключение в виде TcpClient**

**using var client = await server.AcceptTcpClientAsync();**

**// получаем объект NetworkStream для взаимодействия с клиентом**

**var stream = client.GetStream();**

**// создаем StreamReader для чтения данных**

**using var streamReader = new StreamReader(stream);**

**// создаем StreamWriter для отправки данных**

**using var streamWriter = new StreamWriter(stream);**

**while (true)**

**{**

**// считываем запрошенное слово**

**var word = await streamReader.ReadLineAsync();**

**// если получен маркер окончания подключения "END",**

**// завершаем соединение с клиентом**

**if (word == "END") break;**

**Console.WriteLine($"Запрошен перевод слова {word}");**

**// находим слово в словаре и отправляем перевод обратно клиенту**

**if (word is null**

**|| !words.TryGetValue(word, out var translation))**

**translation = "не найдено в словаре";**

**// отправляем перевод слова**

**await streamWriter.WriteLineAsync(translation);**

**await streamWriter.FlushAsync();**

**}**

**}**

**}**

**finally { server.Stop(); }**

#### На стороне клиента:

**using System; // Console**

**using System.IO; // StreamWriter, StreamReader**

**using System.Net.Sockets; // TcpClient**

**// слова для отправки и получения перевода**

**string[] words = [ "red", "yellow", "blue" ];**

**// создаем клиента**

**using TcpClient client = new TcpClient();**

**// подключаемся к серверу**

**await client.ConnectAsync("127.0.0.1", 8888);**

**// получаем NetworkStream для взаимодействия с сервером**

**var stream = client.GetStream();**

**// создаем StreamReader для чтения данных**

**using var streamReader = new StreamReader(stream);**

**// создаем StreamWriter для отправки данных**

**using var streamWriter = new StreamWriter(stream);**

**foreach(var word in words)**

**{**

**// отправляем слово на сервер**

**await streamWriter.WriteLineAsync(word);**

**await streamWriter.FlushAsync();**

**// получаем перевод от сервера**

**var translation = await streamReader.ReadLineAsync();**

**Console.WriteLine($"CLIENT: {word} - {translation}");**

**}**

**// посылаем маркер окончания подключения**

**await streamWriter.WriteLineAsync("END");**

**await streamWriter.FlushAsync();**

## NetworkStream и бинарные потоки

**Для отправки бинарных данных в поток объект NetworkStream обертывается в объект класса BinaryWriter, который для отправки данных предоставляет метод Write().**

**Для чтения бинарных данных из потока объект NetworkStream обертывается в объект класса BinaryReader, который для чтения данных предоставляет ряд методов:**

* **ReadBoolean()**
* **ReadByte()**
* **ReadBytes()**
* **ReadChar()**
* **ReadChars()**
* **ReadDecimal()**
* **ReadDouble()**
* **ReadHalf()**
* **ReadInt16()**
* **ReadInt32()**
* **ReadInt64()**
* **ReadSByte()**
* **ReadSingle()**
* **ReadString()**
* **ReadUInt16()**
* **ReadUInt32()**
* **ReadUInt64()**

#### На стороне сервера:

**// создаем сервер**

**var server = new TcpListener(IPAddress.Any, 8888);**

**try**

**{**

**server.Start(); // запупскаем сервер**

**Console.WriteLine("Сервер запущен. Ожидание подключений...");**

**while (true)**

**{**

**// получаем подключение в виде TcpClient**

**using var client = await server.AcceptTcpClientAsync();**

**// получаем объект NetworkStream для взамодействия с клиентом**

**var stream = client.GetStream();**

**// создаем BinaryReader для чтения данных**

**using var reader = new BinaryReader(stream);**

**// создаем BinaryWriter для отправки данных**

**using var writer = new BinaryWriter(stream);**

**var name = reader.ReadString();**

**var company = reader.ReadString();**

**var count = reader.ReadInt32();**

**var price = reader.ReadDecimal();**

**// Guid.NewGuid() создает уникальный идентификатор**

**string id = Guid.NewGuid().ToString();**

**// создаем новый продукт с полученными данными и**

**// сгенерированным id**

**var product = new Product(id, name, company, count, price);**

**Console.WriteLine($"SERVER: Добавлен новый товар: {product}");**

**// отправляем клиенту сгенерированный id**

**writer.Write(product.Id);**

**writer.Flush();**

**}**

**}**

**finally { server.Stop(); }**

**record Product(**

**string Id,**

**string Name,**

**string Company,**

**int Count,**

**decimal Price);**

#### На стороне клиента:

**// данные для отправки**

**string name = "iPhone 14";**

**string company = "Apple";**

**int count = 2;**

**decimal price = 146890.34m;**

**// создаем клиента**

**using TcpClient client = new TcpClient();**

**// подключаемся к серверу**

**await client.ConnectAsync("127.0.0.1", 8888);**

**// получаем NetworkStream для взаимодействия с сервером**

**var stream = client.GetStream();**

**// создаем BinaryRider для чтения данных**

**using var reader = new BinaryReader(stream);**

**// создаем BinaryWriter для отправки данных**

**using var writer = new BinaryWriter(stream);**

**// отправляем данные товара на сервер**

**writer.Write(name);**

**writer.Write(company);**

**writer.Write(count);**

**writer.Write(price);**

**writer.Flush();**

**// считываем сгенерированный на сервере id товара**

**var id = reader.ReadString();**

**Console.WriteLine($"CLIENT: Id товара: {id}");**

## Консольный TCP-чат

#### Определение сервера

**using System; // Console**

**using System.IO; // StreamWriter**

**using System.Net; // IPAddress**

**using System.Collections.Generic; // List**

**using System.Linq; // List.FirstOrDefault()**

**using System.Net.Sockets; // TcpClient**

**using System.Threading.Tasks; // Task**

**ServerObject server = new ServerObject(); // создаем сервер**

**await server.ListenAsync(); // запускаем сервер**

**class ServerObject**

**{**

**// создаем сервер для прослушивания**

**TcpListener server = new TcpListener(IPAddress.Any, 8888);**

**List<ClientObject> clients = new List<ClientObject>(); // все подключения**

**// закрытие ресурсов клиента по id**

**protected internal void RemoveConnection(string id)**

**{**

**// получаем закрытое подключение по id**

**ClientObject? client = clients.FirstOrDefault(c => c.Id == id);**

**// и удаляем его из списка подключений**

**if (client is not null) clients.Remove(client);**

**client?.Close();**

**}**

**// прослушивание входящих подключений**

**protected internal async Task ListenAsync()**

**{**

**try**

**{**

**server.Start(); // запускаем сервер**

**Console.WriteLine("Сервер запущен. Ожидание подключений...");**

**while (true)**

**{**

**// получаем клиентов по очереди**

**TcpClient client = await server.AcceptTcpClientAsync();**

**// создаем из них ClientObject для прослушивания**

**// сообщений от клиентов**

**var clientObject = new ClientObject(client, this);**

**// добавляем его в список клиентов**

**clients.Add(clientObject);**

**// запускаем метод ClientObject.ProcessAsync параллельно**

**// для работы с клиентом - получения от него сообщений**

**Task.Run(clientObject.ProcessAsync);**

**// запускается параллельно, т.к. нет 'await'**

**}**

**}**

**catch (Exception e) { Console.WriteLine(e.Message); }**

**finally { Disconnect(); }**

**}**

**// трансляция сообщения всем подключенным клиентам**

**protected internal async Task BroadcastMessageAsync(string message, string id)**

**{**

**foreach (var client in clients)**

**{**

**if (client.Id != id) // если id клиента не равно id отправителя**

**{**

**// передача данных**

**await client.Writer.WriteLineAsync(message);**

**await client.Writer.FlushAsync();**

**}**

**}**

**}**

**// отключение всех клиентов**

**protected internal void Disconnect()**

**{**

**foreach (var client in clients)**

**client.Close(); // отключение клиентов**

**server.Stop(); // остановка сервера**

**}**

**}**

**// ClientObject представляет подключение отдельного клиента**

**class ClientObject**

**{**

**protected internal string Id { get; } = Guid.NewGuid().ToString();**

**protected internal StreamWriter Writer { get; }**

**protected internal StreamReader Reader { get; }**

**TcpClient client;**

**ServerObject server; // объект сервера**

**public ClientObject(TcpClient tcpClient, ServerObject serverObject)**

**{**

**client = tcpClient;**

**server = serverObject;**

**// получаем NetworkStream для взаимодействия с сервером**

**var stream = client.GetStream();**

**// создаем StreamReader для чтения данных**

**Reader = new StreamReader(stream);**

**// создаем StreamWriter для чтения данных**

**Writer = new StreamWriter(stream);**

**}**

**// метод для работы с клиентом:**

**// ClientObject читает сообщения от клиента, переданного в конструктор**

**// и публикует их в чат на сервер, также переданный в конструктор.**

**// Все сообщения отображаются в консоли**

**public async Task ProcessAsync()**

**{**

**try**

**{**

**// получаем имя пользователя**

**string? userName = await Reader.ReadLineAsync();**

**string? message = $"{userName} вошел в чат";**

**// посылаем сообщение о входе в чат всем**

**// подключенным пользователям**

**await server.BroadcastMessageAsync(message, Id);**

**Console.WriteLine(message);**

**// в бесконечном цикле получаем сообщения от клиента**

**while (true)**

**{**

**try**

**{**

**message = await Reader.ReadLineAsync();**

**if (message is null) continue;**

**message = $"{userName}: {message}";**

**Console.WriteLine(message);**

**// посылаем сообщение клиента в чат всем**

**// подключенным пользователям**

**await server.BroadcastMessageAsync(message, Id);**

**}**

**catch**

**{**

**// если соединение с клиентом было разорвано**

**// и StreamReader не смог прочитать сообщение**

**// от клиента, то он выбросит исключение.**

**message = $"{userName} покинул чат";**

**Console.WriteLine(message);**

**// посылаем всем подключенным пользователям в чат**

**// сообщение о том, что клиент покинул чат**

**await server.BroadcastMessageAsync(message, Id);**

**break;**

**}**

**}**

**}**

**catch (Exception e) { Console.WriteLine(e.Message); }**

**// в случае выхода из цикла закрываем ресурсы**

**finally { server.RemoveConnection(Id); }**

**}**

**protected internal void Close()**

**{**

**Writer.Close();**

**Reader.Close();**

**client.Close();**

**}**

**}**

#### Создание клиента

**string host = "127.0.0.1";**

**int port = 8888;**

**using TcpClient client = new TcpClient();**

**Console.Write("CLIENT: Введите свое имя: ");**

**string? userName = Console.ReadLine();**

**Console.WriteLine($"CLIENT: Добро пожаловать, {userName}");**

**StreamReader? Reader = null;**

**StreamWriter? Writer = null;**

**try**

**{**

**client.Connect(host, port); // подключение клиента**

**Reader = new StreamReader(client.GetStream());**

**Writer = new StreamWriter(client.GetStream());**

**if (Writer is null || Reader is null) return;**

**// запускаем новый поток для получения данных**

**Task.Run(() => ReceiveMessageAsync(Reader));**

**// запускаем ввод сообщений**

**await SendMessageAsync(Writer, userName);**

**}**

**catch (Exception e) { Console.WriteLine($"CLIENT: {e.Message}"); }**

**Writer?.Close();**

**Reader?.Close();**

**// отправка сообщений**

**async Task SendMessageAsync(StreamWriter writer, string userName)**

**{**

**// сначала отправляем имя**

**await writer.WriteLineAsync(userName);**

**await writer.FlushAsync();**

**Console.WriteLine("CLIENT: Для отправки сообщений введите сообщение" +**

**" и нажмите Enter");**

**while (true)**

**{**

**string? message = Console.ReadLine();**

**await writer.WriteLineAsync(message);**

**await writer.FlushAsync();**

**}**

**}**

**// получение сообщений**

**async Task ReceiveMessageAsync(StreamReader reader)**

**{**

**while (true)**

**{**

**try**

**{**

**// считываем ответ в виде строки**

**string? message = await reader.ReadLineAsync();**

**// если ответ пустой, ничего не выводим на консоль**

**if (string.IsNullOrEmpty(message)) continue;**

**Print(message); // вывод сообщения**

**}**

**catch { break; }**

**}**

**}**

**// чтобы полученное сообщение не накладывалось на ввод нового сообщения**

**void Print(string message)**

**{**

**if (OperatingSystem.IsWindows()) // если ОС Windows**

**{**

**// получаем текущую позицию курсора**

**var position = Console.GetCursorPosition();**

**// смещение в символах относительно левого края**

**int left = position.Left;**

**// смещение в строках относительно верха**

**int top = position.Top;**

**// копируем ранее введенные в строку символы на следующую строку**

**Console.MoveBufferArea(0, top, left, 1, 0, top + 1);**

**// Устанавливаем курсор в начало текущей строки**

**Console.SetCursorPosition(0, top);**

**// в текущей строке выводим полученное сообщение**

**Console.WriteLine(message);**

**// переносим курсор на следующую строку**

**// и пользователь продолжит ввод уже на следующей строке**

**Console.SetCursorPosition(left, top + 1);**

**}**

**else Console.WriteLine(message);**

**}**

Протокол UDP

## Использование сокетов для работы с UDP

### Прослушивание входящих сообщений

**using System.Net;**

**using System.Net.Sockets;**

**// создаем сокет для UDP**

**using var udpSocket = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Dgram,**

**ProtocolType.Udp);**

**// привязываем его к локальной точке 127.0.0.1:5555**

**var localIP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 5555);**

**// и начинаем прослушивание входящих сообщений**

**udpSocket.Bind(localIP);**

### Получение данных

**// .NET 7**

**public Task<SocketReceiveFromResult> ReceiveFromAsync(**

**ArraySegment<byte> buffer, // сюда получаем данные**

**EndPoint remoteEndPoint); // адрес клиента, отправившего данные**

**// .NET 7, 6**

**public Task<SocketReceiveFromResult> ReceiveFromAsync(**

**ArraySegment<byte> buffer, // сюда получаем данные**

**SocketFlags socketFlags, // настройка параметров получения данных**

**EndPoint remoteEndPoint); // адрес клиента, отправившего данные**

**SocketReceiveFromResult - структура, которая содержит свойства:**

* **ReceivedBytes - количество полученных байтов (в случае недуачного выполнения операции, возвращается 0)**
* **RemoteEndPoint - объект EndPoint или адрес клиента, который отправил данные**

#### Пример сервера:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**using var socket = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Dgram,**

**ProtocolType.Udp);**

**var localIp = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 5555);**

**// начинаем прослушивание входящих сообщений**

**socket.Bind(localIp);**

**Console.WriteLine("UDP-сервер запущен...");**

**byte[] data = new byte[256]; // буфер для получаемых данных**

**// 65535 - максимально возможный размер дейтаграммы UDP**

**// адрес, с которого придут данные - любой**

**EndPoint remoteIp = new IPEndPoint(IPAddress.Any, 0);**

**// получаем данные в массив data**

**var result = await socket.ReceiveFromAsync(data, remoteIp);**

**var message = Encoding.UTF8.GetString(data, 0, result.ReceivedBytes);**

**Console.WriteLine($"Получено {result.ReceivedBytes} байт");**

**Console.WriteLine($"Удаленный адрес: {result.RemoteEndPoint}");**

**Console.WriteLine(message); // выводим полученное сообщение**

### Отправка данных

**// .NET 7+**

**public Task<int> SendToAsync(**

**ArraySegment<byte> buffer, // отправляемые данные**

**EndPoint remoteEndPoint); // адрес получателя**

**// .NET 6+**

**public Task<int> SendToAsync(**

**ArraySegment<byte> buffer, // отправляемые данные**

**SocketFlags socketFlags, // настройка параметров отправления данных**

**EndPoint remoteEndPoint); // адрес получателя**

**Результат - количество отправленных байтов.**

#### Пример клиента:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**using var socket = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Dgram,**

**ProtocolType.Udp);**

**// для отравки данных необязательно вызывать метод Bind**

**string message = "Hello World!";**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**EndPoint remoteEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 5555);**

**int bytes = await socket.SendToAsync(data, remoteEndPoint);**

**Console.WriteLine($"Отправлено {bytes} байт");**

## UdpClient

**public UdpClient ();**

**public UdpClient (AddressFamily family);**

**public UdpClient (int port);**

**public UdpClient (IPEndPoint localEP);**

**public UdpClient (int port, AddressFamily family);**

**public UdpClient (string hostname, int port);**

**Используемый сокет можно получить из свойства Client.**

### Получение данных

**public byte[] Receive(**

**ref System.Net.IPEndPoint? remoteEP); // адрес отправителя данных**

**public Task<System.Net.Sockets.UdpReceiveResult> ReceiveAsync();**

**У UdpReceiveResult есть свойства:**

* **Buffer - с помощью этого свойства можно получить массив пришедших данных.**
* **RemoteEndPoint - IPEndPoint отправителя**

#### Пример сервера:

**using System.Net.Sockets;**

**using System.Text;**

**using var server = new UdpClient(5555);**

**// сервер будет прослушивать порт 5555**

**Console.WriteLine("SERVER: UDP-сервер запущен...");**

**// получаем данные**

**UdpReceiveResult result = await server.ReceiveAsync();**

**// предположим, что отправлена строка**

**// преобразуем байты в строку**

**var message = Encoding.UTF8.GetString(result.Buffer);**

**Console.WriteLine($"SERVER: Получено {result.Buffer.Length} байт");**

**Console.WriteLine($"SERVER: Удаленный адрес: {result.RemoteEndPoint}");**

**Console.WriteLine($"SERVER: {message}");**

### Отправка данных

**public int Send(**

**ReadOnlySpan<byte> datagram, // отправляемые данные**

**IPEndPoint? endPoint); // адрес отправки**

**public ValueTask<int> SendAsync(**

**ReadOnlyMemory<byte> datagram, // отправляемые данные**

**System.Net.IPEndPoint? endPoint, // адрес отправки**

**CancellationToken cancellationToken = default);**

**Результат - количество отправленных байтов.**

#### Пример клиента:

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**using var client = new UdpClient();**

**// отправляемые данные**

**string message = "Hello World!";**

**// преобразуем в массив байтов**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**// определяем конечную точку для отправки данных**

**IPEndPoint remotePoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 5555);**

**// отправляем данные**

**int bytes = await client.SendAsync(data, remotePoint);**

**Console.WriteLine($"CLIENT: Отправлено {bytes} байт");**

### Метод Connect

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**using var client = new UdpClient();**

**// данные будут отправлятся только на 127.0.0.1:5555**

**client.Connect("127.0.0.1", 5555);**

**string message = "Hello METANIT";**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**int bytes = await client.SendAsync(data);**

**Console.WriteLine($"Отправлено {bytes} байт");**

## Консольный UDP-чат

### Консольный чат на UDP-сокетах

**using System;**

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**using System.Threading.Tasks;**

**Console.Write("Введите свое имя: ");**

**string? username = Console.ReadLine();**

**Console.WriteLine("Введите IP-адрес для приема сообщений: ");**

**string localIpAddress = Console.ReadLine();**

**IPAddress localAddress = IPAddress.Parse(localIpAddress);**

**Console.Write("Введите порт для приема сообщений: ");**

**if (!int.TryParse(Console.ReadLine(), out var localPort)) return;**

**Console.WriteLine("Введите адрес для отправки сообщений: ");**

**string remoteIpAddress = Console.ReadLine();**

**IPAddress remoteAddress = IPAddress.Parse(remoteIpAddress);**

**Console.Write("Введите порт для отправки сообщений: ");**

**if (!int.TryParse(Console.ReadLine(), out var remotePort)) return;**

**Console.WriteLine();**

**Task.WaitAll(SendMessageAsync(), ReceiveMessageAsync());**

**// отправка сообщений в группу**

**async Task SendMessageAsync()**

**{**

**using Socket sender = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Dgram,**

**ProtocolType.Udp);**

**Console.WriteLine("Для отправки сообщений введите сообщение "**

**+ "и нажмите Enter");**

**// отправляем сообщения**

**while (true)**

**{**

**var message = Console.ReadLine(); // сообщение для отправки**

**// если введена пустая строка,**

**// выходим из цикла и завершаем ввод сообщений**

**if (string.IsNullOrWhiteSpace(message)) break;**

**// иначе добавляем к сообщению имя пользователя**

**message = $"{username}: {message}";**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**// и отправляем на remoteAddress:remotePort**

**await sender**

**.SendToAsync(data, new IPEndPoint(remoteAddress, remotePort));**

**}**

**}**

**// получение сообщений**

**async Task ReceiveMessageAsync()**

**{**

**byte[] data = new byte[65535]; // буфер для получаемых данных**

**// сокет для прослушки сообщений**

**using Socket receiver = new Socket(**

**AddressFamily.InterNetwork,**

**SocketType.Dgram,**

**ProtocolType.Udp);**

**// запускаем прослушивание сообщений по адресу localAddress:localPort**

**receiver.Bind(new IPEndPoint(localAddress, localPort));**

**while (true)**

**{**

**// получаем данные в массив data**

**var result = await receiver.ReceiveFromAsync(**

**data,**

**new IPEndPoint(IPAddress.Any, 0));**

**var message = Encoding.UTF8.GetString(data, 0, result.ReceivedBytes);**

**// выводим сообщение**

**Console.WriteLine(message);**

**}**

**}**

### Консольный чат с помощью UdpClient

**using System;**

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**using System.Threading.Tasks;**

**Console.Write("Введите свое имя: ");**

**string? username = Console.ReadLine();**

**Console.WriteLine("Введите IP-адрес для приема сообщений: ");**

**string localIpAddress = Console.ReadLine();**

**IPAddress localAddress = IPAddress.Parse(localIpAddress);**

**Console.Write("Введите порт для приема сообщений: ");**

**if (!int.TryParse(Console.ReadLine(), out var localPort)) return;**

**Console.WriteLine("Введите адрес для отправки сообщений: ");**

**string remoteIpAddress = Console.ReadLine();**

**IPAddress remoteAddress = IPAddress.Parse(remoteIpAddress);**

**Console.Write("Введите порт для отправки сообщений: ");**

**if (!int.TryParse(Console.ReadLine(), out var remotePort)) return;**

**Console.WriteLine();**

**Task.WaitAll(SendMessageAsync(), ReceiveMessageAsync());**

**// отправка сообщений в группу**

**async Task SendMessageAsync()**

**{**

**using UdpClient sender = new UdpClient();**

**Console.WriteLine("Для отправки сообщений введите сообщение "**

**+ "и нажмите Enter");**

**// отправляем сообщения**

**while (true)**

**{**

**var message = Console.ReadLine(); // сообщение для отправки**

**// если введена пустая строка, выходим из цикла**

**// и завершаем ввод сообщений**

**if (string.IsNullOrWhiteSpace(message)) break;**

**// иначе добавляем к сообщению имя пользователя**

**message = $"{username}: {message}";**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**// и отправляем на remoteAddress: remotePort**

**await sender**

**.SendAsync(data, new IPEndPoint(remoteAddress, remotePort));**

**}**

**}**

**// получение сообщений**

**async Task ReceiveMessageAsync()**

**{**

**using UdpClient receiver = new UdpClient(localPort);**

**while (true)**

**{**

**// получаем данные**

**var result = await receiver.ReceiveAsync();**

**var message = Encoding.UTF8.GetString(result.Buffer);**

**// выводим сообщение**

**Console.WriteLine(message);**

**}**

**}**

### Наложение полученного сообщения на ввод сообщения

**Чтобы сообщения не накладывались одно на другое, когда пользователь пишет свое сообщение, а собеседник уже отправил ему свое, для ОS Windows можно изменить метод ReceiveMessageAsync:**

**async Task ReceiveMessageAsync()**

**{**

**using UdpClient receiver = new UdpClient(localPort);**

**while (true)**

**{**

**// получаем данные**

**var result = await receiver.ReceiveAsync();**

**var message = Encoding.UTF8.GetString(result.Buffer);**

**// выводим сообщение**

**Print(message);**

**}**

**}**

**void Print(string message)**

**{**

**if (OperatingSystem.IsWindows()) // если OS Windows**

**{**

**// получаем текущую позицию курсора**

**var position = Console.GetCursorPosition();**

**// смещение в символах относительно левого края**

**int left = position.Left;**

**// смещение в строках относительно верха**

**int top = position.Top;**

**// копируем ранее введенные символы в строке на следующую строку**

**Console.MoveBufferArea(0, top, left, 1, 0, top + 1);**

**// устанавливаем курсор в начало текущей строки**

**Console.SetCursorPosition(0, top);**

**// в текущей строке выводим полученное сообщение**

**Console.WriteLine(message);**

**// переносим курсор на следующую строку**

**// и пользователь продолжает ввод уже на следующей строке**

**Console.SetCursorPosition(left, top + 1);**

**}**

**else Console.WriteLine(message);**

**}**

## Широковещательная рассылка

**При использовании широковещательной передачи надо учитывать, что передача не идет дальше локальных сетей, так как маршрутизаторы подобные рассылки не пропускают.**

### Присоединение к группе. JoinMulticastGroup и DropMulticastGroup

**Широковещательная групповая рассылка доступна только для группы. Чтобы добавить клиент UdpClient в группу, у него вызывается метод JoinMulticastGroup(). Одна из его версий:**

**public void JoinMulticastGroup(System.Net.IPAddress multicastAddr);**

**public void JoinMulticastGroup(**

**// адрес группы: 224.0.0.0 - 239.255.255.255**

**System.Net.IPAddress multicastAddr,**

**// число маршрутизаторов,**

**// через которое дейтаграмма может пройти**

**int timeToLive);**

**Стоит отметить, что для присоединения к группе UdpClient должен быть привязан к определенному порту, по которому его сокет будет прослушивать входящиее сообщения. Для этого в конструктор UdpClient можно передать номер порта. Кроме того, присоединение к группе необходимо только для получения сообщений, которые предназначаются группе. А для отправки сообщений в группу необязательно принадлежать группе.**

**Для удаления из группы у UdpClient вызывается метод DropMulticastGroup() с тем же адресом группы:**

**public void DropMulticastGroup (System.Net.IPAddress multicastAddr);**

#### Пример клиента-получателя:

**using System.Net.Sockets;**

**using System.Net;**

**using System.Text;**

**using var client = new UdpClient(8001);**

**// хост для получения данных**

**var broadcastAddress = IPAddress.Parse("235.5.5.11");**

**// присоединяемся к группе**

**client.JoinMulticastGroup(broadcastAddress);**

**Console.WriteLine("Начало прослушивания сообщений");**

**while (true)**

**{**

**var result = await client.ReceiveAsync();**

**string message = Encoding.UTF8.GetString(result.Buffer);**

**if (message == "END") break;**

**Console.WriteLine(message);**

**}**

**// отсоединяемся от группы**

**client.DropMulticastGroup(broadcastAddress);**

**Console.WriteLine("UDP-клиент завершил свою работу");**

#### Пример клиента-отправителя:

**using System.Net.Sockets;**

**using System.Net;**

**using System.Text;**

**var messages = new string[]**

**{**

**"Hello World!",**

**"Hello METANIT.COM",**

**"Hollo work",**

**"END"**

**};**

**// хост для отправки данных**

**var broadcastAddress = IPAddress.Parse("235.5.5.11");**

**using var sender = new UdpClient();**

**Console.WriteLine("Начало отправки сообщений...");**

**// отправляем сообщения**

**foreach(var message in messages)**

**{**

**Console.WriteLine($"Отправляется сообщение: {message}");**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**await sender.SentAsync(data, new IPEndPoint(broadcastAddress, 8001));**

**await Task.Delay(1000);**

**}**

### Получение и отправка сообщений в одном приложении

**using System.Net;**

**using System.Net.Sockets;**

**using System.Text;**

**int localPort = 8001;**

**IPAddress broadcastAddress = IPAddress.Parse("235.5.5.11");**

**Console.Write("Введите свое имя: ");**

**string? username = Console.ReadLine();**

**Task.WaitAll(ReceiveMessagAsync(), SendMessageAsync());**

**// отправка сообщений в группу**

**async Task SendMessageAsync()**

**{**

**// создаем UdpClient для отправки**

**using var sender = new UdpClient();**

**// отправляем сообщения**

**while (true)**

**{**

**// сообщение для отправки**

**string? message = Console.ReadLine();**

**// если введена пустая строка,**

**// выходим из цикла и завершаем ввод сообщений**

**if (string.IsNullOrWhiteSpace(message)) break;**

**// иначе добавляем к сообщению имя пользователя**

**message = $"{username}: {message}";**

**byte[] data = Encoding.UTF8.GetBytes(message);**

**// и отправляем в группу**

**await sender**

**.SendAsync(data, new IPEndPoint(broadcastAddress, localPort));**

**}**

**}**

**// получение сообщений из группы**

**async Task ReceiveMessagAsync()**

**{**

**// UdpClient для получения данных**

**using var receiver = new UdpClient(localPort);**

**receiver.JoinMulticastGroup(broadcastAddress);**

**// отключаем получение своих же сообщений**

**receiver.MulticastLoopback = false;**

**while (true)**

**{**

**var result = await receiver.ReceiveAsync();**

**string message = Encoding.UTF8.GetString(result.Buffer);**

**Console.WriteLine(message);**

**}**

**}**