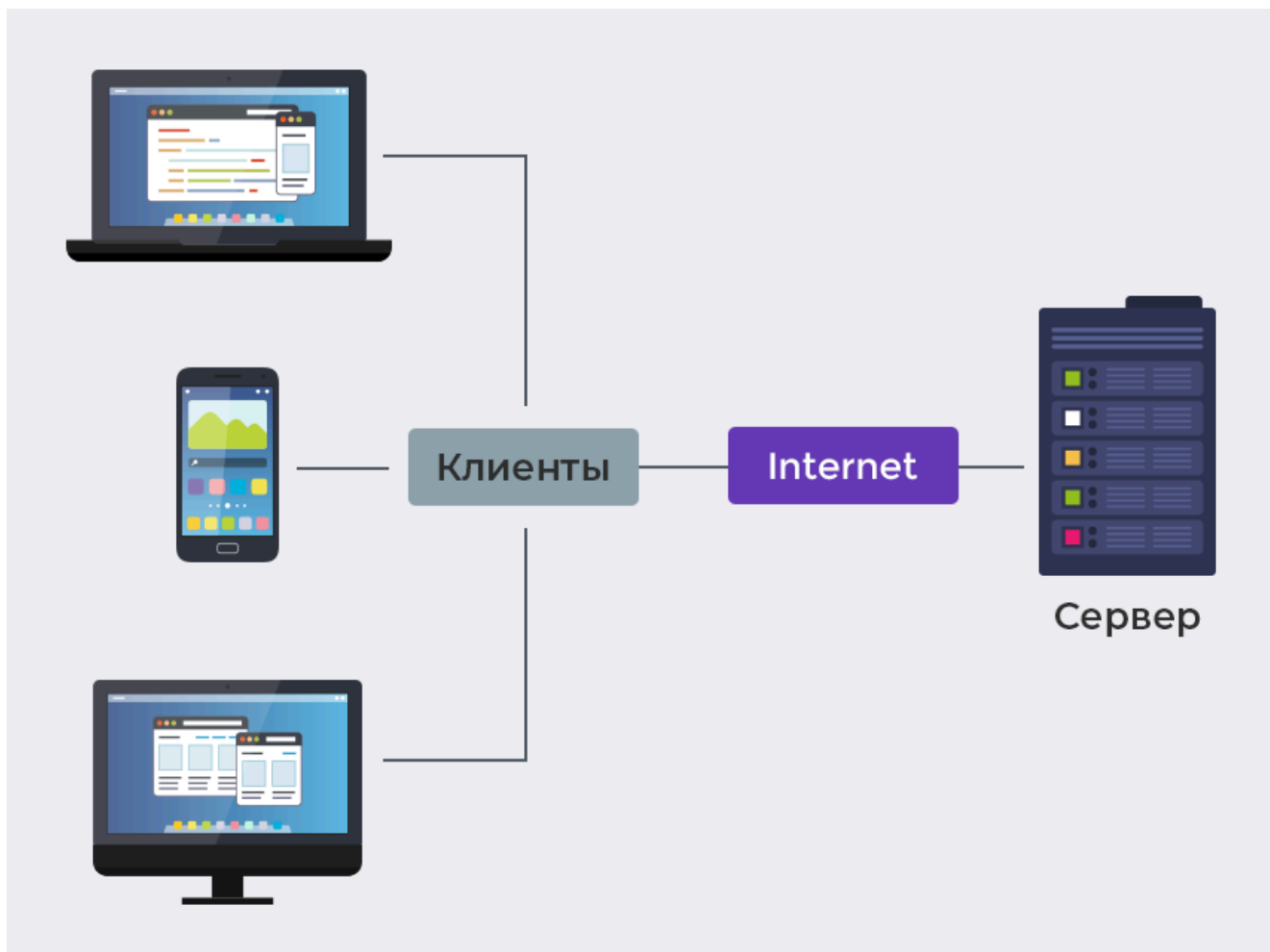


Лекция №8. Сетевое взаимодействие

Основы сетевого взаимодействия и протоколы

Сетевое взаимодействие строится на модели **клиент-сервер**, где:

- **Клиент** — приложение или устройство, инициирующее запросы (например, браузер, мобильное приложение).
- **Сервер** — приложение или устройство, обрабатывающее запросы и возвращающее результаты (например, веб-сервер, игровой сервер).



Для идентификации устройств и приложений используются:

- **IP-адрес:**
 - **IPv4:** 32-битный адрес в формате `192.168.0.1`.
 - **IPv6:** 128-битный адрес в формате `2001:0db8:85a3::8a2e:0370:7334`. Решает проблему нехватки IPv4.
- **Порт:**
 - Число от 0 до 65535, определяющее, какому приложению на устройстве предназначены данные. Например так мы подключимся к порту `8080` по

адресу 127.0.0.1 — 127.0.0.1:8080 .

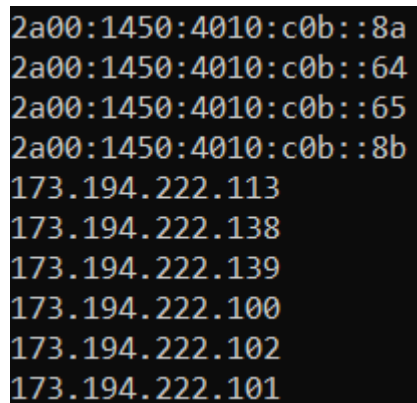
- **DNS (Domain Name System):**

- Система преобразования доменных имён (например, google.com) в IP-адреса.

С помощью этого кода, мы сможем получить список всех IP-адресов соответствующих доменному имени google.com .

```
IPHostEntry entry = Dns.GetHostEntry("google.com");
IPAddress[] addresses = entry.AddressList;
foreach(var address in addresses)
    Console.WriteLine(address);
```

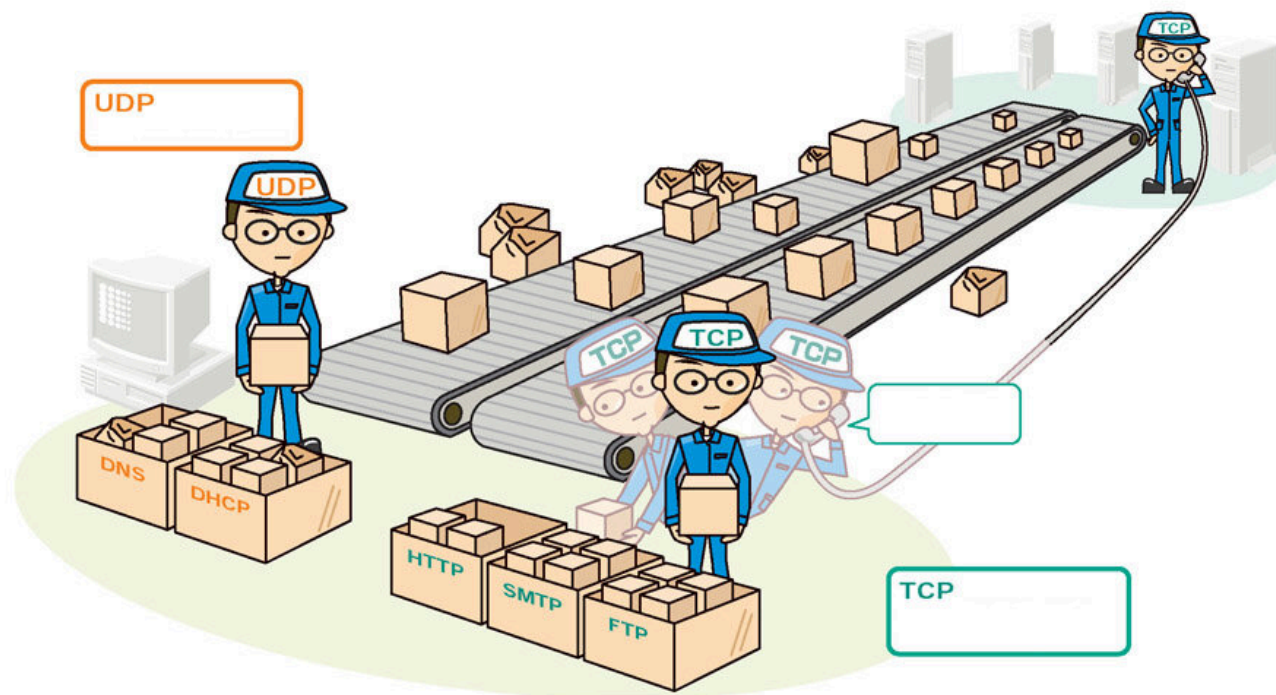
Результат в консоли.



```
2a00:1450:4010:c0b::8a
2a00:1450:4010:c0b::64
2a00:1450:4010:c0b::65
2a00:1450:4010:c0b::8b
173.194.222.113
173.194.222.138
173.194.222.139
173.194.222.100
173.194.222.102
173.194.222.101
```

При передачи данных по сети они разделяются на отдельные пакеты. Для их передачи используется 2 основных протокола — **TCP и UDP**:

- **TCP (Transmission Control Protocol)** — этот протокол гарантирует доставку данных и их порядок, он устанавливает соединение между источником и получателем данных и адаптирует скорость передачи под сетевые условия. TCP используется тогда, когда важна целостность данных (веб-страницы, файловые загрузки).
- **UDP (User Datagram Protocol)** — этот протокол не устанавливает соединение, данные отправляются сразу, пакеты обрабатываются независимо (может быть потеря пакетов или нарушение порядка). Он используется тогда, когда важна скорость, а не надёжность (стриминг, онлайн-игры).



На основе этих двух протоколов работают более высокоуровневые протоколы прикладного уровня. Передача данных по сети и иерархия протоколов описывается моделью **TCP/IP**:

1. **Прикладной уровень (HTTP, DNS и др.)** — интерфейсы для пользовательских приложений.
2. **Транспортный уровень (TCP, UDP)** — управление передачей данных между приложениями.
3. **Сетевой уровень (IP)** — маршрутизация пакетов между сетями.
4. **Канальный уровень (Ethernet, Wi-Fi)** — передача данных внутри одной сети.

Сокет — это программный интерфейс для взаимодействия с сетевыми протоколами. В C# класс `Socket` (пространство имён `System.Net.Sockets`) предоставляет низкоуровневый контроль над сетевыми операциями.

```
// TCP-сокет (поточковый)
Socket tcpSocket = new Socket(
    AddressFamily.InterNetwork, // IPv4
    SocketType.Stream,          // Поточковый тип (TCP)
    ProtocolType.Tcp
);

// UDP-сокет (датаграммный)
Socket udpSocket = new Socket(
    AddressFamily.InterNetwork,
```

```
        SocketType.Dgram,           // Датаграммный тип (UDP)
        ProtocolType.Udp
    );
```

Создадим TCP-сервер:

```
// Привязка к адресу и порту
IPAddress serverIP = IPAddress.Parse("127.0.0.1");
IPEndPoint serverEndPoint = new IPEndPoint(serverIP, 8080);
tcpSocket.Bind(serverEndPoint);
// Начало прослушивания
tcpSocket.Listen(10); // Максимум 10 подключений в очереди
while (true)
{
    // Принятие подключений
    using Socket clientSocket = tcpSocket.Accept();
    // Обработка клиента
    byte[] buffer = new byte[1024];
    // Чтение данных в буфер
    int bytesReceived = clientSocket.Receive(buffer);
    // Переводим данные в строку
    string message = Encoding.UTF8.GetString(buffer, 0, bytesReceived);
    Console.WriteLine($"Получено: {message}");

    // Ответ клиенту
    string response = "Сообщение получено!";
    // Переводим строку в массив байт и отправляем клиенту
    byte[] responseData = Encoding.UTF8.GetBytes(response);
    clientSocket.Send(responseData);

    // Закрываем клиентское подключение
    clientSocket.Shutdown(SocketShutdown.Both);
}
```

Напишем теперь код для TCP-клиента:

```
// Подключение к серверу
IPAddress serverIP = IPAddress.Parse("127.0.0.1");
IPEndPoint serverEndPoint = new IPEndPoint(serverIP, 8080);
tcpSocket.Connect(serverEndPoint);

// Отправка данных
string message = "Привет, сервер!";
byte[] data = Encoding.UTF8.GetBytes(message);
tcpSocket.Send(data);

// Получение ответа
byte[] buffer = new byte[1024];
```

```
int bytesReceived = tcpSocket.Receive(buffer);
string response = Encoding.UTF8.GetString(buffer, 0, bytesReceived);
Console.WriteLine($"Ответ сервера: {response}");
```

Теперь после того как мы запустим сервер, а потом клиент то получим:

- На сервере **Получено: Привет, сервер!**
- На клиенте **Ответ сервера: Сообщение получено!**

TcpClient, TcpListener, UdpClient

Классы `TcpClient`, `TcpListener` и `UdpClient` упрощают работу с сетевыми протоколами, инкапсулируя низкоуровневые операции сокетов. Их преимущества:

- Меньше шаблонного кода (не нужно вручную создавать сокет, биндить их и т.д.).
- Упрощённая обработка потоков данных (например, через `NetworkStream`).
- Более читаемый и поддерживаемый код.

Напишем TCP-сервер и TCP-клиент, аналогичный предыдущему, но с использованием `TcpClient`, `TcpListener`.

Сервер:

```
// Создаем
IPAddress serverIP = IPAddress.Parse("127.0.0.1");
using TcpListener server = new TcpListener(serverIP, 8080);
// Запускаем
server.Start();
while (true)
{
    // Примем подключение
    using TcpClient client = server.AcceptTcpClient();
    // Используем потоки для удобства
    using NetworkStream ns = client.GetStream();
    using StreamReader sr = new StreamReader(ns);
    using StreamWriter sw = new StreamWriter(ns);
    // Читаем
    string message = sr.ReadLine();
    Console.WriteLine($"Получено: {message}");

    // Отправка ответа
    sw.WriteLine("Сообщение получено!");
    sw.Flush();
}
```

Клиент:

```
// Создаем и подключаем к серверу
using TcpClient client = new TcpClient();
client.Connect("127.0.0.1", 8080);

using NetworkStream ns = client.GetStream();
using StreamReader sr = new StreamReader(ns);
using StreamWriter sw = new StreamWriter(ns);

sw.WriteLine("Привет, сервер!");
sw.Flush();

// Получение ответа
var response = sr.ReadLine();
Console.WriteLine($"Ответ сервера: {response}");
```

Теперь давайте научимся создавать UDP клиента и сервера.

Сервер:

```
using UdpClient udpServer = new UdpClient(8080); // Прослушивание порта 8080
while (true)
{
    UdpReceiveResult result = await udpServer.ReceiveAsync();
    IPEndPoint clientEndPoint = result.RemoteEndPoint;
    string message = Encoding.UTF8.GetString(result.Buffer);

    // Отправка ответа
    byte[] response = Encoding.UTF8.GetBytes("Ответ UDP");
    await udpServer.SendAsync(response, response.Length, clientEndPoint);
}
```

Клиент:

```
using UdpClient udpClient = new UdpClient();
IPEndPoint serverEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"),
8080);

// Отправка данных
byte[] data = Encoding.UTF8.GetBytes("Привет, UDP-сервер!");
await udpClient.SendAsync(data, data.Length, serverEndPoint);

// Получение ответа
UdpReceiveResult result = await udpClient.ReceiveAsync();
Console.WriteLine(Encoding.UTF8.GetString(result.Buffer));
```

HTTP-взаимодействие

HTTP (HyperText Transfer Protocol) — протокол прикладного уровня для передачи данных между клиентом и сервером. Работает на основе TCP.

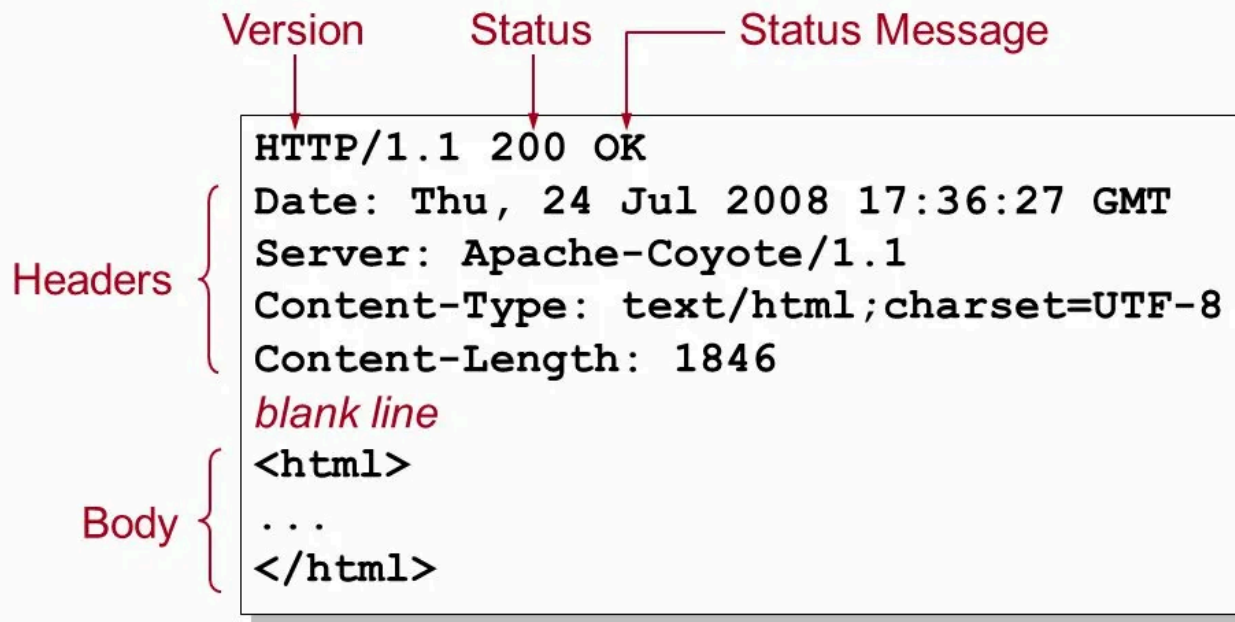
- **Ключевые особенности:**
 - **Статус-коды:** 200 (OK), 404 (Not Found), 500 (Server Error).
 - **Методы запросов:** GET (получение данных), POST (отправка данных), PUT, DELETE и др.
 - **Заголовки (Headers):** Содержат метаданные (тип контента, авторизация, кеширование).
 - **Тело (Body):** Содержит передаваемую информацию.
 - **HTTPS:** Зашифрованная версия HTTP с использованием SSL/TLS.
- **Используется для:**
 - Веб-сайтов, REST API, мобильных приложений, микросервисов.

Структура HTTP запроса:



Структура HTTP ответа:

HTTP Response



Заголовок **Content-Type** — указывает формат данных в теле запроса/ответа.

Основные типы:

- `application/json` : JSON-данные (стандарт для REST API).
- `application/xml` : XML-данные.
- `text/html` : HTML-страницы.
- `multipart/form-data` : Загрузка файлов через формы.
- `application/x-www-form-urlencoded` : Данные формы в виде `key=value`.

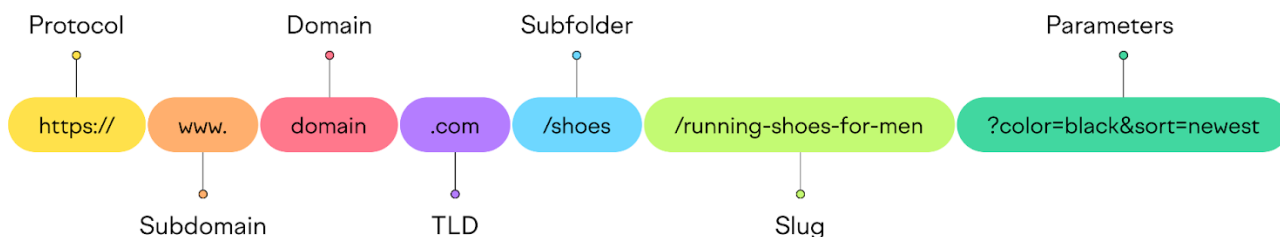
HTTP-запросы работают по разному, это можно представить в виде таблицы.

Метод	Назначение	Тело запроса	Идемпотентный	Безопасный
GET	Получение данных	Нет	Да	Да
POST	Создание ресурса	Да	Нет	Нет
PUT	Полное обновление ресурса	Да	Да	Нет
PATCH	Частичное обновление ресурса	Да	Нет	Нет
DELETE	Удаление ресурса	Нет	Да	Нет

Идемпотентность: Повторный вызов метода даёт тот же результат (например, PUT, DELETE).

Безопасность: Метод не изменяет состояние сервера (только GET).

Как вы могли заметить у методов GET и DELETE нет тела запроса. Как же они тогда передают данные? Это делается с помощью параметров в **URL (Uniform Resource Locator)** адресе.



В общем виде это можно записать так:

```
схема://домен:порт/путь?параметры#якорь
```

- **Схема:** `http`, `https`, `ftp`.
- **Домен:** Адрес сервера (`example.com`).
- **Путь:** Ресурс на сервере (`/api/users`).
- **Параметры:** Пары `ключ=значение` после `?`, разделённые `&` (`?id=5&sort=asc`).
- **Якорь:** Ссылка на часть страницы (не отправляется на сервер).

REST API

REST (Representational State Transfer) — архитектурный стиль для создания веб-сервисов, основанный на HTTP. Он позволяет упорядочить взаимодействие между клиентом и сервером. В качестве формата данных обычно используется JSON.

Ключевой абстракцией в REST является ресурс. **Ресурс** — это все, что вы хотите показать внешнему миру через ваше приложение. Например, если мы пишем приложение для управления задачами, экземпляры ресурсов будут следующие:

- Конкретный пользователь
- Конкретная задача
- Список задач

Для идентификации ресурса, ему назначается **URI (Uniform Resource Identifier)** адрес, по которому можно получить к нему доступ. Для каждого ресурса определяют список допустимых действий. Пара из HTTP-метода и URI адреса называется **эндпоинтом (endpoint)**. Пример REST-эндпоинтов:

- GET /api/users — список пользователей.
- POST /api/users — создать пользователя.
- GET /api/users/5 — получить пользователя с ID=5.

Обычно для каждого ресурса нужно определить 4 типа действий — **создание (Create)**, **чтение (Read)**, **обновление (Update)**, **удаление (Delete)**. Сокращенно это называют **CRUD**.

API (Application Programming Interface) — это набор правил, протоколов и инструментов, который позволяет различным программам взаимодействовать друг с другом. Если мы хотим сделать RESTfull API необходимо выполнить следующие шаги:

- Определить ресурсы, которые будут в нашей системе.
- Назначить для них URI адреса.
- Реализовать для них CRUD-операции.
- Определить формат передаваемых и возвращаемых данных (заголовки, структуру JSON файлов в body, параметры из URL строки и статус-коды).

URI адрес можно создать с помощью класса `UriBuilder`:

```
string url = "https://api.example.com/data";
var uriBuilder = new UriBuilder(url);

// Так можно добавить параметры запроса
var queryParams = new Dictionary<string, string>
{
    { "page", "1" },
    { "limit", "10" }
};
uriBuilder.Query = string.Join("&", queryParams.Select(kvp => $"{kvp.Key}={kvp.Value}"));
```

HttpClient

`HttpClient` (пространство имён `System.Net.Http`) — основной инструмент для работы с HTTP в современных приложениях.

Рассмотрим пример GET-запроса:

```
using HttpClient client = new HttpClient();

// Отправка GET-запроса
HttpResponseMessage response = await
client.GetAsync("https://api.example.com/data");

// Проверка статуса
```

```

if (response.IsSuccessStatusCode)
{
    // Чтение ответа как строки
    string content = await response.Content.ReadAsStringAsync();
    Console.WriteLine(content);
}

```

Пример POST-запроса:

```

using HttpClient client = new HttpClient();

// Создание JSON-данных
var data = new { Name = "Alice", Age = 30 };
string json = JsonSerializer.Serialize(data);
StringContent content = new StringContent(json, Encoding.UTF8,
"application/json");

// Отправка POST
HttpResponseMessage response = await
client.PostAsync("https://api.example.com/users", content);

if (response.IsSuccessStatusCode)
{
    // Чтение ответа как строки
    string content = await response.Content.ReadAsStringAsync();
    Console.WriteLine(content);
}

```

Запись данных в виде JSON файла называется **сериализацией**:

```

// Сериализация объекта в JSON
var user = new User { Id = 1, Name = "Bob" };
string json = JsonSerializer.Serialize(user);

```

```

// Десериализация JSON в объект
string jsonResponse = await response.Content.ReadAsStringAsync();
var user = JsonSerializer.Deserialize<User>(jsonResponse);

```

Можно изменять не только тело запроса но и заголовки:

```

client.DefaultRequestHeaders.Add("Authorization", "Bearer token123");
client.DefaultRequestHeaders.UserAgent.ParseAdd("MyApp/1.0");

```

И читать их:

```
IEnumerable<string> values = response.Headers.GetValues("X-RateLimit-Limit");
```

Рассмотрим пример взаимодействия с REST API сервера с данными о погоде:

```
public class WeatherClient
{
    private readonly HttpClient _client;
    private const string ApiKey = "ваш_ключ";

    public WeatherClient()
    {
        _client = new HttpClient();
        _client.BaseAddress = new Uri("https://api.weatherapi.com/v1/");
    }

    public async Task<WeatherData> GetWeatherAsync(string city)
    {
        string url = $"current.json?key={ApiKey}&q={city}";
        HttpResponseMessage response = await _client.GetAsync(url);
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadFromJsonAsync<WeatherData>();
    }
}
```

Рекомендации

- **Не создавайте избыточные экземпляры:**

Сокеты, HTTP-клиенты (`HttpClient` , `TcpClient`) и потоки (`NetworkStream`) — это ресурсоёмкие объекты.

```
// Не правильно: новый HttpClient для каждого запроса
using (var client = new HttpClient()) { ... }
```

```
// Правильно: используйте один экземпляр или IHttpClientFactory
private static readonly HttpClient _client = new HttpClient();
```

- **Всегда закрывайте соединения:**

Используйте `Dispose()` , `Close()` или `using` , чтобы избежать утечек ресурсов.

```
using (TcpClient client = new TcpClient())
{
    await client.ConnectAsync(...);
    // ...
} // Соединение закроется автоматически
```

- **Используйте `async / await` для всех I/O-операций:**

Синхронные вызовы блокируют потоки, что снижает производительность при высокой нагрузке.

```
// Не правильно: синхронный вызов
var data = client.GetString(url);

// Правильно: асинхронный подход
var data = await client.GetStringAsync(url);
```

- **Отмена долгих операций:**

Передавайте `CancellationToken` в асинхронные методы для контроля времени выполнения.

```
var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10));
await client.GetAsync(url, cts.Token);
```

- **Обязательно используйте `try-catch`:**

Сетевые операции ненадёжны (разрыв соединения, таймауты, некорректные данные).

```
try
{
    await client.ConnectAsync(...);
}
catch (SocketException ex)
{
    Console.WriteLine($"Ошибка подключения: {ex.SocketErrorCode}");
}
catch (HttpRequestException ex)
{
    Console.WriteLine($"HTTP-ошибка: {ex.StatusCode}");
}
```