

# HttpClient

прошлое, настоящее, будущее

Риваль Абдрахманов

Positive Technologies

SpbDotNet, 2019

- 1 HttpClient - базовая информация
- 2 Неочевидные проблемы
- 3 Интерфейс IHttpConnectionFactory
- 4 Дополнительные улучшения в .NET Core 2.1
- 5 HttpRequestMessage и HttpResponseMessage
- 6 Новое в .NET Core 3.0

# HttpClient - базовая информация

# HttpClient Class

- Provides a base class for sending HTTP requests and receiving HTTP responses from a resource identified by a URI;

- Provides a base class for sending HTTP requests and receiving HTTP responses from a resource identified by a URI;
- *GetAsync(...)*, *PostAsync(...)*, *SendAsync(...)* и др.;

- Provides a base class for sending HTTP requests and receiving HTTP responses from a resource identified by a URI;
- *GetAsync(...)*, *PostAsync(...)*, *SendAsync(...)* и др.;
- *HttpClient* реализует *IDisposable*.

# IDisposable Interface

- Provides a mechanism for releasing unmanaged resources;

# IDisposable Interface

- Provides a mechanism for releasing unmanaged resources;
- *public void Dispose();*



# IDisposable Interface

- Provides a mechanism for releasing unmanaged resources;
- *public void Dispose();*
- Конструкция *using(...);*

# IDisposable Interface

- Provides a mechanism for releasing unmanaged resources;
- *public void Dispose();*
- Конструкция *using(...);*
- Диспозиться → диспозь

# IDisposable Interface

- Provides a mechanism for releasing unmanaged resources;
- *public void Dispose()*;
- Конструкция *using(...)*;
- Дислопозитесья → диспозь
- Дислопозитесья → внимательней

# Disposable HttpClient

```
using(var client = new HttpClient())  
{  
    var response = await client.GetStringAsync(...);  
}
```

## Неочевидные проблемы

# Проблема socket exhaustion

<https://aspnetmonsters.com/2016/08/2016-08-27-httpclientwrong/>

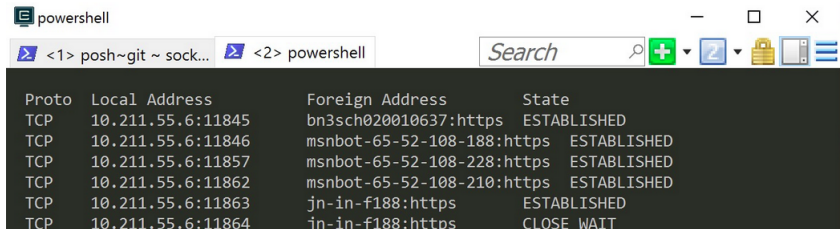
## YOU'RE USING HTTPCLIENT WRONG AND IT IS DESTABILIZING YOUR SOFTWARE

BY : SIMON TIMMS

2016-08-28

COMMENTS

I've been using HttpClient wrong for years and it finally came back to bite me. My site was unstable and my clients furious, with a simple fix performance improved greatly and the instability disappeared.



The screenshot shows a Windows PowerShell window with the title "powershell". The command prompt shows two tabs: "<1> posh~git ~ sock..." and "<2> powershell". The command "netstat -an" has been executed, displaying a list of active TCP connections. The output is as follows:

Proto	Local Address	Foreign Address	State
TCP	10.211.55.6:11845	bn3sch020010637:https	ESTABLISHED
TCP	10.211.55.6:11846	msnbot-65-52-108-188:https	ESTABLISHED
TCP	10.211.55.6:11857	msnbot-65-52-108-228:https	ESTABLISHED
TCP	10.211.55.6:11862	msnbot-65-52-108-210:https	ESTABLISHED
TCP	10.211.55.6:11863	jn-in-f188:https	ESTABLISHED
TCP	10.211.55.6:11864	jn-in-f188:https	CLOSE_WAIT

# Проблема socket exhaustion

```
for(int i = 0; i < 10; i++)  
{  
    using (var client = new HttpClient())  
    {  
        await client.GetStringAsync("https://google.com");  
    }  
}
```

# Проблема socket exhaustion

Проверяем через netstat:

```
tcp      0      0 rafaelldi-Latitud:41757 lq-in-f147.1e100.:https TIME_WAIT
tcp      0      0 rafaelldi-Latitud:44779 lq-in-f147.1e100.:https TIME_WAIT
tcp      0      0 rafaelldi-Latitud:37367 lq-in-f147.1e100.:https TIME_WAIT
tcp      0      0 rafaelldi-Latitud:32979 lq-in-f147.1e100.:https TIME_WAIT
tcp      0      0 rafaelldi-Latitud:38399 lq-in-f104.1e100.:https TIME_WAIT
tcp      0      0 rafaelldi-Latitud:44257 lq-in-f147.1e100.:https TIME_WAIT
tcp      0      0 rafaelldi-Latitud:46173 lq-in-f147.1e100.:https TIME_WAIT
tcp      0      0 rafaelldi-Latitud:44449 lq-in-f147.1e100.:https TIME_WAIT
tcp      0      0 rafaelldi-Latitud:35151 lq-in-f147.1e100.:https TIME_WAIT
tcp      0      0 rafaelldi-Latitud:46791 lq-in-f147.1e100.:https TIME_WAIT
```



# Проблема socket exhaustion

- 10 сокетов в состоянии *TIME WAIT*;
- Соединение закрыто с одной стороны, но мы всё ещё ждём входящие пакеты, чтобы правильно их обработать;
- Приводит к *SocketException*.

“HttpClient is intended to be instantiated once and re-used throughout the life of an application. Instantiating an HttpClient class for every request will exhaust the number of sockets available under heavy loads.”

*<https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient>*

# Проблема socket exhaustion

Решение проблемы - переиспользование клиента:

```
private static HttpClient Client = new HttpClient();
```

<https://byterot.blogspot.com/2016/07/singleton-httpclient-dns.html>

Wednesday, 20 July 2016

## Singleton HttpClient? Beware of this serious behaviour and how to fix it

If you are consuming a Web API in your server-side code (or .NET client-side app), you are very likely to be using an HttpClient.

HttpClient is a very nice and clean implementation that came as part of Web API and replaced its clunky predecessor WebClient (although only in its HTTP functionality, WebClient can do more than just HTTP).

HttpClient is usually meant to be used with more than just a single request. It conveniently allows for default headers to be set and applied to all requests. Also you can plug in a CookieContainer to allow for all sessions.

# Проблема кеширования DNS

- Не учитываются изменения DNS;
- Соединение держится до закрытия сокета.

Решение для .NET Framework:

- Класс *ServicePointManager*;
- *ServicePointManager.DnsRefreshTimeout* – время, которое будет закеширован полученный IP адрес для каждого доменного имени, по умолчанию 2 минуты;
- *ServicePoint.ConnectionLeaseTimeout* – время, которое соединение будет удерживаться открытым, по умолчанию не ограничено;
- *ServicePoint.MaxIdleTime* – время бездействия, после которого соединение будет закрыто, по умолчанию 100 секунд.

# Проблема кеширования DNS

Пример:

```
ServicePointManager.DnsRefreshTimeout = 60000;

var sp = ServicePointManager.FindServicePoint(
    new Uri("https://google.com"));
sp.ConnectionLeaseTimeout = 60000;
sp.MaxIdleTime = 60000;
```

<https://habr.com/ru/post/424873/>



YuriyIvon October 1, 2018 at 08:36 AM

## Подводные камни HttpClient в .NET

.NET

Продолжая серию статей о «подводных камнях» не могу обойти стороной System.Net.HttpClient, который очень часто используется на практике, но при этом имеет несколько серьезных проблем, которые могут быть сразу не видны.

Достаточно частая проблема в программировании — то, что разработчики сфокусированы только на функциональных возможностях того или иного компонента, при этом совершенно не учитывают очень важную нефункциональную составляющую, которая может влиять на производительность, масштабируемость, легкость восстановления в случае сбоев, безопасность и т.д. Например, тот же HttpClient — вроде бы и элементарный компонент, но есть несколько вопросов: сколько он создает параллельных соединений к серверу, как долго они живут, как он себя поведет, если DNS имя, к которому обращался ранее, будет переключено на другой IP адрес? Попробуем ответить на эти вопросы в статье.



# Лимит одновременных соединений с сервером

- Лимит одновременных соединений с сервером по умолчанию равен 2;
- *ServicePointManager.DefaultConnectionLimit*;
- Для *localhost* по умолчанию равен *int.MaxValue*;
- Только для .NET Framework.

## Интерфейс IHttpConnectionFactory

## IHttpClientFactory

An IHttpClientFactory can be registered and used to configure and create HttpClient instances in an app.

Интерфейс был добавлен в ASP.NET Core 2.1.

Для консольного приложения придётся добавить *Microsoft.Extensions.Hosting* и *Microsoft.Extensions.Http*.

- Регистрация через метод расширения *IServiceCollection*:

```
services.AddHttpClient();
```

# IHttpClientFactory - базовое использование

- Регистрация через метод расширения *IServiceCollection*:

```
services.AddHttpClient();
```

- Добавление в конструктор с помощью DI:

```
public SomeService(IHttpClientFactory  
    clientFactory)  
{  
    _clientFactory = clientFactory;  
}
```

# IHttpClientFactory - базовое использование

- Регистрация через метод расширения *IServiceCollection*:

```
services.AddHttpClient();
```

- Добавление в конструктор с помощью DI:

```
public SomeService(IHttpClientFactory  
    clientFactory)  
{  
    _clientFactory = clientFactory;  
}
```

- Создание клиента (без *using*):

```
var client = _clientFactory.CreateClient();  
var response = await client.SendAsync(request);
```

# Named clients

- Регистрация через метод расширения *IServiceCollection*:

```
services.AddHttpClient("some-site", c =>
{
    c.BaseAddress =
        new Uri("https://some-site.com/");
    c.DefaultRequestHeaders
        .Add("Accept", "application/json");
});
```

# Named clients

- Регистрация через метод расширения *IServiceCollection*:

```
services.AddHttpClient("some-site", c =>
{
    c.BaseAddress =
        new Uri("https://some-site.com/");
    c.DefaultRequestHeaders
        .Add("Accept", "application/json");
});
```

- Добавление в конструктор с помощью DI:

```
public SomeService(IHttpClientFactory
    clientFactory)
{
    _clientFactory = clientFactory;
}
```



# Named clients

- Регистрация через метод расширения *IServiceCollection*:

```
services.AddHttpClient("some-site", c =>
{
    c.BaseAddress =
        new Uri("https://some-site.com/");
    c.DefaultRequestHeaders
        .Add("Accept", "application/json");
});
```

- Добавление в конструктор с помощью DI:

```
public SomeService(IHttpClientFactory
    clientFactory)
{
    _clientFactory = clientFactory;
}
```

- Создание клиента:

```
var client =
    _clientFactory.CreateClient("some-site");
```

# Typed clients

Класс типизированного клиента:

```
public class SomeSiteClient
{
    private readonly HttpClient _client;

    public SomeSiteClient(HttpClient client)
    {
        client.BaseAddress =
            new Uri("https://some-site.com/");
        client.DefaultRequestHeaders
            .Add("Accept", "application/json");

        _client = client;
    }

    ...
}
```

# Typed clients

Класс типизированного клиента:

```
public class SomeSiteClient
{
    ...

    public async Task<SomeData> GetSomeData()
    {
        var response =
            await _client.GetAsync("/get-some-data");

        ...

        return result;
    }
}
```

- Регистрация типизированного клиента:

```
services.AddHttpClient<SomeSiteClient>();
```

- Регистрация типизированного клиента:

```
services.AddHttpClient<SomeSiteClient>();
```

- Добавление в конструктор через DI:

```
public SomeService(SomeSiteClient someSiteClient)
{
    _someSiteClient = someSiteClient;
}
```

Библиотека Refit для REST API (<https://github.com/reactiveui/refit>)

```
public interface ISomeSiteClient
{
    [Get("/get-some-data")]
    Task<SomeData> GetSomeData();
}

public class SomeData
{
    public string Message { get; set; }
}
```

- Регистрация клиента:

```
services
    .AddRefitClient<ISomeSiteClient>()
    .ConfigureHttpClient(c => c.BaseAddress =
        new Uri("https://some-site.com"));
```

- Регистрация клиента:

```
services
    .AddRefitClient<ISomeSiteClient>()
    .ConfigureHttpClient(c => c.BaseAddress =
        new Uri("https://some-site.com"));
```

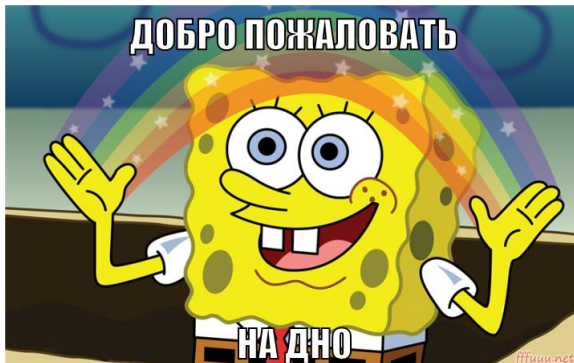
- Добавление в конструктор через DI:

```
public SomeService(ISomeSiteClient someSiteClient)
{
    _someSiteClient = someSiteClient;
}
```



# Создание HttpClient

Посмотрим глубже, как происходит создание *HttpClient*



# Создание HttpClient

Посмотрим на наследование:

```
public class HttpClient : HttpMessageInvoker  
  
public class HttpMessageInvoker : IDisposable
```

# Конструкторы `HttpMessageInvoker`

У *`HttpMessageInvoker`* два конструктора:

```
public HttpMessageInvoker(HttpMessageHandler handler,
    bool disposeHandler)
{
    ...
    this._handler = handler;
    this._disposeHandler = disposeHandler;
    ...
}

public HttpMessageInvoker(HttpMessageHandler handler)
    : this(handler, true)
{
}
```

# Конструкторы HttpClient

У *HttpClient* три конструктора:

```
public HttpClient(HttpMessageHandler handler,
    bool disposeHandler) : base(handler, disposeHandler)
{
    ...
}
```

```
public HttpClient(HttpMessageHandler handler)
    : this(handler, true)
{
}
```

```
public HttpClient()
    : this((HttpMessageHandler) new HttpClientHandler())
{
}
```

# Конструкторы HttpClient

- 3 конструктора;
- Есть возможность передать *HttpMessageHandler*;
- В стандартном конструкторе *disposeHandler* равен *true*.

# Dispose HttpClient

```
protected override void Dispose(bool disposing)
{
    if (disposing && !this._disposed)
    {
        this._disposed = true;
        this._pendingRequestsCts.Cancel();
        this._pendingRequestsCts.Dispose();
    }
    base.Dispose(disposing);
}
```

# Dispose HttpResponseMessage

```
public void Dispose()
{
    this.Dispose(true);
    GC.SuppressFinalize((object) this);
}

protected virtual void Dispose(bool disposing)
{
    if (!disposing || this._disposed)
        return;
    this._disposed = true;
    if (!this._disposeHandler)
        return;
    this._handler.Dispose();
}
```

# Dispose HttpClient

- Отменяются все повисшие запросы  $\Rightarrow$  если переиспользовать клиент, могут отменяться чужие запросы;
- Флаг *disposed* выставляется в *true*;
- *Dispose* вызывается у *HttpMessageHandler* только в случае *disposeHandler = true*.



# Создание с помощью HttpClientFactory

```
public static HttpClient CreateClient(this
    IHttpHttpClientFactory factory)
{
    if (factory == null)
        throw new ArgumentNullException(nameof (factory));

    return factory
        .CreateClient(Microsoft.Extensions
            .Options.Options.DefaultName);
}

...

public static readonly string DefaultName =
    string.Empty;
```

# Создание с помощью DefaultHttpClientFactory

```
public HttpClient CreateClient(string name)
{
    ...
    HttpClient httpClient =
        new HttpClient(this.CreateHandler(name), false);
    ...
    return httpClient;
}

public HttpResponseMessage CreateHandler(string name)
{
    ActiveHandlerTrackingEntry entry =
        this._activeHandlers.GetOrAdd(name,
            this._entryFactory).Value;
    this.StartHandlerEntryTimer(entry);
    return (HttpMessageHandler) entry.Handler;
}
```

# Создание с помощью HttpClientFactory

- При создании через *HttpClientFactory* в параметр *disposeHandler* передаётся *false*  $\Rightarrow$  при *Dispose* ничего не произойдёт;
- *HttpClientFactory* при создании проверяет наличие *Handler* с соответствующим именем и по возможности использует его;
- *HttpClientFactory* выставляет таймер для *Handler*.

## Дополнительные улучшения в .NET Core 2.1

- *DelegatingHandler* позволяют создать цепочку обработки исходящих запросов;
- Схоже с middleware в ASP.NET Core;
- Функциональность уже была, но с *IHttpClientFactory* стало проще использовать.

# Создание DelegatingHandler

```
public class SomeHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage>
        SendAsync(HttpRequestMessage request,
            CancellationToken cancellationToken)
    {
        ...

        var response = await base.SendAsync(request,
            cancellationToken);

        ...

        return response;
    }
}
```

# Регистрация DelegatingHandler

```
services.AddHttpClient("some-site")  
    //first  
    .AddHttpMessageHandler<OutsideHandler>()  
    //second  
    .AddHttpMessageHandler<InsideHandler>()
```

- Polly - популярная библиотека обработки ошибок;
- Содержит различные политики: Retry, Circuit Breaker, Timeout, ...
- Необходимо установить *Microsoft.Extensions.Http.Polly*;
- Подходит не только для *HttpClient*.





Обработка всех ответов со статус кодами 5xx и 408.

```
services.AddHttpClient("some-site")
    .AddTransientHttpErrorPolicy(p => p.RetryAsync(3))
    .AddTransientHttpErrorPolicy(
        p => p.CircuitBreakerAsync(5,
            TimeSpan.FromSeconds(30)));
```

# Настройка внутреннего HttpResponseMessage

```
services.AddHttpClient("some-site")
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        return new SocketsHttpHandler()
        {
            AutomaticDecompression =
                DecompressionMethods.GZip
        };
    });
```

# Настройка времени жизни `HttpMessageHandler`

- Для каждого именованного клиента есть свой *`HttpMessageHandler`*;
- *`IHttpClientFactory`* при создании нового *`HttpClient`* будет переиспользовать *`HttpMessageHandler`*, если его время жизни не вышло;
- Время жизни по умолчанию 2 минуты.

```
services.AddHttpClient("some-site")  
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

## HttpRequestMessage и HttpResponseMessage

# HttpRequestMessage

Represents a HTTP request message.

Содержит в себе:

- *HttpMethod method*;
- *Uri requestUri*;
- *HttpRequestHeaders headers*;
- *Version version*, значение по умолчанию *Version(2, 0)*;
- *HttpContent content*, который является *IDisposable*;

# Диспозить или нет?

Зависит от *HttpContent*. Чаще всего это *StringContent*  $\Rightarrow$  можно не диспозить.

Represents a HTTP response message including the status code and data.

Содержит в себе:

- *HttpStatusCode statusCode* (есть проверка *value > 0* и *value < 999*);
- *HttpResponseHeaders headers*;
- *string reasonPhrase*
- *HttpRequestMessage requestMessage*
- *Version version*, значение по умолчанию *Version(1, 1)*;
- *HttpContent content*, который является *IDisposable*;

# Диспозить или нет?

*HttpClient* в методах *GetAsync* и *SendAsync* принимает параметр *HttpCompletionOption*: *ResponseContentRead*, *ResponseHeadersRead* (первый вариант по умолчанию).

- Если *ResponseContentRead*, то данные сохраняются в *MemoryStream*  $\Rightarrow$  можно без диспоза;
- Иначе стоит диспозить.



# Антипаттерн чтения в строку

Получаем данные, сохраняем в строку и десериализуем.

```
var response = await client
    .GetAsync("/get-some-data");
var str = await response.Content
    .ReadAsStringAsync();
var someData = JsonConvert
    .DeserializeObject<SomeData>(str);
return someData;
```

## Десериализуем из stream

```
var serializer = new JsonSerializer();  
var response = await client  
    .GetAsync("/get-some-data");  
var stream = await response.Content  
    .ReadAsStreamAsync();  
using (var sr = new StreamReader(stream))  
using (var jsonReader = new JsonTextReader(sr))  
{  
    var someData = serializer  
        .Deserialize<SomeData>(jsonReader);  
    return someData;  
}
```

# Десериализуем из stream в .net core 3.0

```
var response = await client
    .GetAsync("/get-some-data");
var stream = await response.Content
    .ReadAsStreamAsync();
var someDate = await JsonSerializer
    .DeserializeAsync<SomeData>(stream);
return someDate;
```

## Новое в .NET Core 3.0

Два способа:

```
using (var request =  
    new HttpRequestMessage(HttpMethod.Get, "/")  
    { Version = new Version(2, 0) })  
  
var client = new HttpClient()  
{  
    BaseAddress = new Uri("https://localhost:5001"),  
    DefaultRequestVersion = new Version(2, 0)  
};
```

Схожий шаблон с *HttpClient*:

```
services
    .AddGrpcClient<GreeterClient>(options =>
    {
        options.BaseAddress =
            new Uri("https://localhost:5001");
    });
```

- Слайды и примеры  
<https://github.com/rafaelldi/SpbDotNet-HttpClient>;
- HttpClient Class;
- IDisposable Interface;
- You're using HttpClient wrong and it is destabilizing your software;
- Singleton HttpClient? Beware of this serious behaviour and how to fix it;
- Beware of the .NET HttpClient;
- Подводные камни HttpClient в .NET;
- Make HTTP requests using IHttpClientFactory in ASP.NET Core
- Refit;
- Polly;
- Новость про HTTP/2;
- Новость про gRPC.