

Клиентский HTTP в .NET: от WebRequest до SocketsHttpHandler

Евгений Пешков
JetBrains
@epeshk

О чём будем говорить

Hypertext Transfer Protocol (HTTP) - сетевой протокол для клиент-серверного взаимодействия с request-response семантикой

Применение:

- Загрузка Web-страниц (headless браузеры, Selenium)
- Межсервисное взаимодействие через API
- Транспорт для других протоколов (gRPC-over-HTTP2)

Стандартный доклад про HTTP в .NET

- Есть класс `HttpClient`
- У него есть метод `.Dispose()`

```
using (var httpClient = new HttpClient())  
{  
    var stream = await httpClient.GetStreamAsync("https://dot.net");  
    ...  
}
```

- Так нельзя, это не даёт переиспользовать TCP-соединения!
- Нужно переиспользовать объекты `HttpClient`

Рассмотрим этот пример подробнее

```
using (var httpClient = new HttpClient())  
{  
    var stream = await httpClient.GetStreamAsync("https://dot.net");  
    ...  
}
```

Рассмотрим этот пример подробнее

```
using (var httpClient = new HttpClient())  
{  
    var stream = await httpClient.GetStreamAsync("https://dot.net");  
    ...  
}
```

1. HttpClient появился только в .NET Framework 4.5 (*4.0 – NuGet)

Рассмотрим этот пример подробнее

```
using (var httpClient = new HttpClient(new HttpClientHandler()))  
{  
    var stream = await httpClient.GetStreamAsync("https://dot.net");  
    ...  
}
```

1. HttpClient появился только в .NET Framework 4.5 (*4.0 – NuGet)
2. Какой механизм отправки запросов будет использован?

Рассмотрим этот пример подробнее

```
using (var httpClient = new HttpClient(new HttpClientHandler()))  
{  
    public abstract class HttpResponseMessage : IDisposable  
    {  
        protected internal abstract Task<HttpResponseMessage>  
        SendAsync(  
            HttpRequestMessage request,  
            CancellationToken cancellationToken);  
    }  
}
```

1.

2.

Рассмотрим этот пример подробнее

```
using (var httpClient = new HttpClient(new HttpClientHandler()))
```

```
{  
    public abstract class HttpResponseMessage : IDisposable  
    {  
        protected internal abstract Task<HttpResponseMessage>  
            SendAsync(  
                HttpRequestMessage request,  
                CancellationToken cancellationToken);  
    }  
}
```

1.

```
2. public class HttpResponseMessageInvoker : IDisposable  
    {  
        new HttpResponseMessageInvoker(HttpMessageHandler handler) { }  
  
        public Task<HttpResponseMessage> SendAsync(..., ...);  
    }
```


Рассмотрим этот пример подробнее

```
using (var httpClient = new HttpClient(new HttpClientHandler()))  
{  
    var stream = await httpClient.GetStreamAsync("https://dot.net");  
    ...  
}
```

1. HttpClient появился только в .NET Framework 4.5 (*4.0 – NuGet)
2. Какой механизм отправки запросов будет использован?

Рассмотрим этот пример подробнее

```
using (var httpClient = new HttpClient(new HttpClientHandler()))
{
    var stream = await httpClient.GetStreamAsync("https://dot.net");
    ...
}
```

1. `HttpClient` появился только в .NET Framework 4.5 (*4.0 – NuGet)
2. Какой механизм отправки запросов будет использован?
3. Как устроены Request/Response body? Как с ними работать?

Рассмотрим этот пример подробнее

```
using (var httpClient = new HttpClient(new HttpClientHandler()))
{
    var stream = await httpClient.GetStreamAsync("https://dot.net");
    ...
}
```

1. `HttpClient` появился только в .NET Framework 4.5 (*4.0 – NuGet)
2. Какой механизм отправки запросов будет использован?
3. Как устроены Request/Response body? Как с ними работать?
4. Как устанавливается соединение с сервером? (DNS, версия HTTP...)

Рассмотрим этот пример подробнее

```
using (var httpClient = new HttpClient(new HttpClientHandler()))
{
    var stream = await httpClient.GetStreamAsync("https://dot.net");
    ...
}
```

1. `HttpClient` появился только в .NET Framework 4.5 (*4.0 – NuGet)
2. Какой механизм отправки запросов будет использован?
3. Как устроены Request/Response body? Как с ними работать?
4. Как устанавливается соединение с сервером? (DNS, версия HTTP...)
5. Особенности защищённого соединения, проверка сертификата

Рассмотрим этот пример подробнее

```
using (var httpClient = new HttpClient(new HttpClientHandler()))
{
    var stream = await httpClient.GetStreamAsync("https://dot.net");
    ...
}
```

1. `HttpClient` появился только в .NET Framework 4.5 (*4.0 – NuGet)
2. Какой механизм отправки запросов будет использован?
3. Как устроены Request/Response body? Как с ними работать?
4. Как устанавливается соединение с сервером? (DNS, версия HTTP...)
5. Особенности защищённого соединения, проверка сертификата
6. Ситуации, когда запросов несколько, переиспользование соединений

Основная проблема

- Работа с клиентским HTTP зависит от реализации .NET
- В интернете представлена противоречивая информация о клиентском HTTP

HTTP клиенты в .NET Framework

На .NET Framework:

- WebRequest – основное API
 - Основан на Managed Socket + winapi
- WebClient – устаревшая обёртка над ним
- HttpClientHandler – тоже обёртка над WebRequest

HTTP клиенты в .NET Core/.NET

- HttpClient – основное API
- WebRequest – неудачная обёртка над HttpClient
- WebClient – неудачная обёртка над WebRequest

Ну а что вы хотели, всё ради быстрого перехода на кор

Почему неудачные – будет рассказано позже

HttpMessageHandler's: .NET Core/.NET

Рассмотрим, какие HttpMessageHandler используются в современном .NET

Мир современного .NET в плане клиентского HTTP делится на:

- До .NET Core 2.1
- .NET Core 2.1 и следующие

.NET Core до 2.1: нативные хэндлеры

- WinHttpHandler (`http.sys`)
 - Входит в .NET Core как `internal` (в .NET 5 уже нет)
 - Доступен из NuGet как `public`
- CurlHandler (`libcurl`) – `internal only`

Недостатки:

- Неконсистентная работа на разных ОС
- Сложная обработка ошибок

.NET Core 2.1 and later

SocketsHttpHandler

- Managed-реализация (на основе класса Socket)
- Consistent behavior across all .NET platforms (*Docs)
- Поддержка HTTP/2 с .NET Core 3.0
- Множество настроек, делающих хаки ненужными
- Единственная развивающаяся реализация HTTP в .NET

.NET Framework: alternative handlers

- Стандартный – обёртка над WebRequest
- WinHttpHandler (NuGet)
 - Возможное применение – HTTP/2.0
- SocketsHttpHandler ??? **его здесь НЕТ**
 - На .NET 5-* переезжайте
 - Backport на .NET Standard 2 (**на свой страх и риск**)
 - <https://github.com/TalAloni/StandardSocketsHttpHandler/tree/3.1>

Mono

- Do 6.0: custom WebRequest + HttpClientHandler
- C 6.0:
 - SocketsHttpHandler (default)
 - MonoWebRequestHandler

"Imperial Red": Bring HttpClient from CoreFx

<https://github.com/mono/mono/pull/11906>

3. Вред обёрток

Повторим, что:

- HttpClientHandler в .NET Framework – обёртка над WebRequest
- WebRequest в .NET Core/.NET – обёртка над HttpClient

Покажем один из недостатков этих обёрток

Big streams

Отправим большой стрим в качестве Request Body

```
new HttpRequestMessage(HttpMethod.Post, ...) {  
    Content = new StreamContent(bigStream)  
});
```

SocketsHttpHandler: OK

.NET Framework: контент стрима копируется в память,
если Stream – non-seekable

Big streams: HttpClient B Framework

```
var client = new HttpClient(new HttpClientHandler
{
    MaxRequestContentBufferSize = 4096
});
```

System.Net.Http.HttpRequestException:
Cannot write more bytes to the buffer than
the configured maximum buffer size: 4096.

Big streams: HttpClient в Framework

```
new HttpRequestMessage(HttpMethod.Post, ...) {  
    Content = new StreamContent(bigStream) {  
        Headers = { ContentLength = bigStream.Length }  
    }  
});
```

```
new HttpRequestMessage(HttpMethod.Post, ...) {  
    Content = new StreamContent(bigStream),  
    Headers = { TransferEncodingChunked = true }  
});
```

// OK: хорошая практика так делать всегда

Big streams: WebRequest на .NET Core

- Всё ещё хуже: буферизуется любой стрим

`System.Net.RequestStream`

```
// Cache the request stream into a MemoryStream. This is the  
// default behavior of Desktop HttpWebRequest.AllowWriteStreamBuffering (true).  
// Unfortunately, this property is not exposed in .NET Core, so it can't be changed  
// This will result in inefficient memory usage when sending (POST'ing) large  
// amounts of data to the server such as from a file stream.
```

- Упомянутый метод на самом деле в .NET Core есть
- Решений нет
- ИМХО, реальная причина – никому не нужен легаси

Какое API из стандартных выбрать?

- HttpClient stack
 - Основное API для современного .NET (с SocketsHttpHandler)
 - Good enough на .NET Framework
 - Универсален для использования в библиотеках
- WebRequest на .NET Framework
 - Если вы чётко понимаете, для чего вам это нужно

Рекомендация по работе с HttpClient

- Никогда не используйте конструктор HttpClient без параметров
- Всегда создавайте HttpClientHandler по умолчанию или конкретную его реализацию и сконфигурируйте всё, что вам нужно

Альтернативные API (нугет либы)

Как же всё сложно, пойду скачаю либу, где уже всё сделали за меня

- RestSharp (98M downloads, 25000/day)
- Flurl.Http (15M downloads, 5700/day)

RestSharp

- 98 миллионов загрузок, 25 тысяч в день
- До сих пор использует legacy HttpRequest

Issues:

- [RestSharp on .NET Core loads entire file into memory when uploading #1481](#)
- [Out of Memory issue when uploading large files in .NET Core #1441](#)

RestSharp

Но у меня нет больших стримов, это меня не коснётся!!!

- Это лишь одна из возможных проблем
- Нельзя предсказать новые требования к сервису
- Использование legacy – бомба замедленного действия

Swagger Codegen

Генератор API-клиентов по спецификации

- Проблема:
 - Генерирует RestSharp-based клиент
 - В итоге сгенерированный клиент плох для .NET Core/.NET
- Решение:
 - Использовать другой генератор (NSwag)

Flurl

- Использует System.Net.Http
- Даёт возможность тюнинга хэндлера
- Но всё по прежнему вручную
- Ничего криминального не диагностировано
- Но ничего и не гарантируется

HTTP/2.0 & QUIC

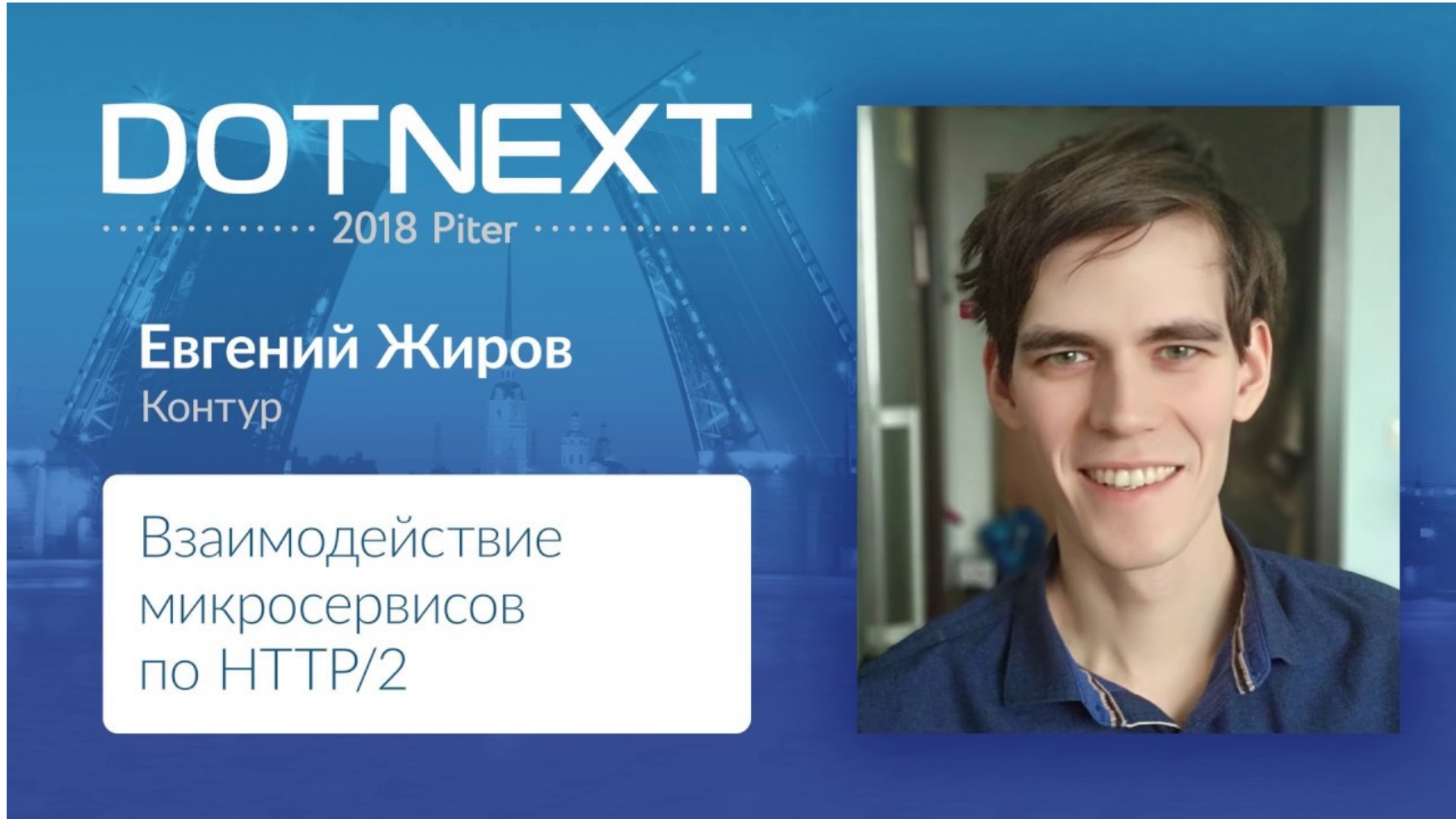
Версии HTTP

- 1991 год: HTTP/0.9 – первая документированная версия
- 1996 год: HTTP/1.0 – почти современный
- 1997 год: HTTP/1.1 – pipelining, keep-alive, Chunked Transfer Encoding
- 2015 год: HTTP/2.0 – эффективное использование TCP-соединений
- 2020 год: HTTP/3.0 (QUIC) – переход на UDP

HTTP/2.0 в .NET

- .NET Framework
 - WinHttpHandler
 - Порт SocketsHttpHandler (на свой страх и риск)
- .NET Core/.NET
 - Нативные хэндлеры (WinHttpHandler/CurlHandler)
 - Изничтожены с .NET 5
 - SocketsHttpHandler (с .NET Core 3.0)
 - Да, в .NET Core 2.1 и 2.2 по умолчанию HTTP/2 не работал

Native handlers: HTTP/2.0



The banner for DOTNEXT 2018 Piter features a blue background with a cityscape and cranes. The text 'DOTNEXT' is prominently displayed in white, followed by '..... 2018 Piter'. Below this, the name 'Евгений Жиров' and the company 'Контур' are listed. A white box contains the title of the presentation: 'Взаимодействие микросервисов по HTTP/2'. To the right is a portrait of Evgeny Zhironov, a young man with dark hair, smiling and wearing a blue shirt.

DOTNEXT
..... 2018 Piter

Евгений Жиров
Контур

Взаимодействие
микросервисов
по HTTP/2

<https://www.youtube.com/watch?v=QgK6-8zCnQM>

<https://2018.dotnext-piter.ru/2018/spb/people/I5snp8o1iqiw4awwyaego/>

[https://assets.ctfassets.net/9n3x4rtjlya6/2D1uCQKQ4QoaE4YCEq4sey/0467c6c44911a6bc6530793307b8b7fc/Evgeny_Zhironov - Microservice interaction with HTTP2.pdf](https://assets.ctfassets.net/9n3x4rtjlya6/2D1uCQKQ4QoaE4YCEq4sey/0467c6c44911a6bc6530793307b8b7fc/Evgeny_Zhironov_-_Microservice_interaction_with_HTTP2.pdf)

Native handlers: HTTP/2.0

Одна из проблем:

- WinHTTPHandler поддерживает только HTTP2-over-TLS
- В use-case из доклада это приводило к большому потреблению unmanaged памяти

Plain text HTTP/2.0 (h2c)

- Изначально HTTP/2.0 требовал шифрование
- Но к релизу стандарта здравый смысл победил паранойю
- Но обязательный TLS успел стать частью реализаций

SocketsHttpHandler: h2c

```
HttpClient httpClient = new(new SocketsHttpHandler());

HttpRequestMessage message = new (HttpMethod.Get, "http://nghttp2.org/")
{
    Version = new Version(2, 0),
};

HttpResponseMessage response = await httpClient.SendAsync(message);
Version usedVersion = response.Version;
Console.WriteLine(usedVersion);

// 1.1
```


SocketsHttpHandler: h2c

.NET Core 3.x

- `AppContext.SetSwitch("System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport", true);`
- `DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_HTTP2UNENCRYPTEDSUPPORT`

.NET 5.0 - *

```
new HttpRequestMessage(...)
{
    Version = new Version(2, 0),
    VersionPolicy = HttpVersionPolicy.RequestVersionExact
};                                     // .RequestVersionOrHigher
                                     // (exactly same for plaintext!)
```

HTTP/3.0 (QUIC)

- В разработке
- Сейчас используется нативная реализация MsQuic

Low level & Performance

Request timeout

- `.Timeout` в `WebRequest` работает только для синхронных запросов
- Замена – `CancellationToken` с таймаутом
- Таймауты бывают разные (DNS, headers, body)

DNS Cache

DNS: IP-адрес сервиса может измениться.

WebRequest (Framework):

A) `ServicePointManager.DnsRefreshTimeout = 0;`
`request.ServicePoint.ConnectionLeaseTimeout = ...;`

B) `dnsapi.dll` calls for Windows DNS cache (`ipconfig /flushdns`)

DNS Cache

DNS: IP-адрес сервиса может измениться.

SocketsHttpHandler (modern .NET):

- A) `handler.PooledConnectionLifetime = ...;`
- B) `HttpClientFactory` (require Microsoft DI)
- C) your OS DNS cache

Другие сетевые хаки

- ARP cache (IP-to-MAC)
- TCP Keep-Alive
- Остаются для самостоятельного рассмотрения

S for Security and Suffering

- Аналогично – конфигурируется в разных местах
 - Certificate validation callback
- Mono: custom cert store

Много запросов

WebRequest – ограничение тредпула

Exception: "There were not enough free threads in the ThreadPool to complete the operation."

Много запросов

WebRequest – ограничение тредпула

```
internal static bool IsThreadPoolLow() {  
    int workerThreads, completionPortThreads;  
    ThreadPool.GetAvailableThreads(out workers, out iocp);  
  
    return workers < 2;  
}
```

- Решение: сконфигурировать тредпул – увеличить количество потоков, создаваемых без задержки (SetMinThreads)

6. Много запросов

WebRequest – ограничение числа соединений к одному endpoint

Для localhost – бесконечно

Для внешних хостов – по умолчанию 2

Тестируйте ваш сетевой код не только на LocalHost

```
request.ServicePoint.ConnectionLimit = ...;
```

```
httpClientHandler.MaxConnectionsPerSever = ...;
```

6. Много запросов: троттлинг

Для самостоятельного изучения:

- Nagle Algorithm
- Expect: 100-continue

Выводы

- Устройство HTTP-клиента зависит от платформы
- Используйте современный .NET и SocketsHttpHandler
- Даже популярные либы могут подвести
- Делаете библиотеку – предусмотрите extension point для конфигурации клиента
- Тестируйте код на всех платформах, где он будет работать
- Тестируйте код в условиях, приближенных к боевым

Links

- Подводные камни HTTP в .NET ([@YuriIvon](#))
<https://habr.com/ru/post/424873/>
- Reusing HttpClient didn't solve all my problems ([@rahulbhuwal](#))
(О тюнинге настроек WebRequest на Framework, на самом деле)
<https://itnext.io/reusing-httpclient-didnt-solve-all-my-problems-142a32a5b4d8>
- Vostok.ClusterClient.Transport
(Используемая в Контуре обёртка над различными HTTP API со множеством хаков)
<https://github.com/vostok/clusterclient.transport>
- Sources
<https://github.com/dotnet/runtime/blob/main/src/libraries/System.Net.Http/src/>

Вопросы

Евгений Пешков
JetBrains
@epeshk