HttpClient Best Practices

Michael Schneider · Follow
4 min read · Aug 9

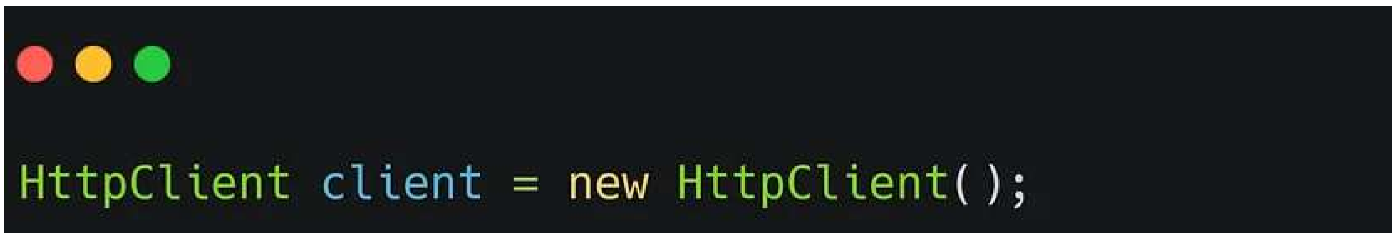
Listen Share More

The **HttpClient** class provides a base class for sending HTTP requests and receiving responses from a resource identified by a URI. This is often the go-to choice when calling RESTful APIs in .NET.

Support me by buying me a cup of coffee. Donate Coffee ☕ .

However, using HttpClient isn't as straightforward as it may seem.

You might be tempted to instantiate a new HttpClient for each request like so:



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

Unfortunately, this approach has significant downsides:

Socket Exhaustion:

Each HttpClient instance has its pool of sockets. Creating new instances for each request could exhaust the available sockets.

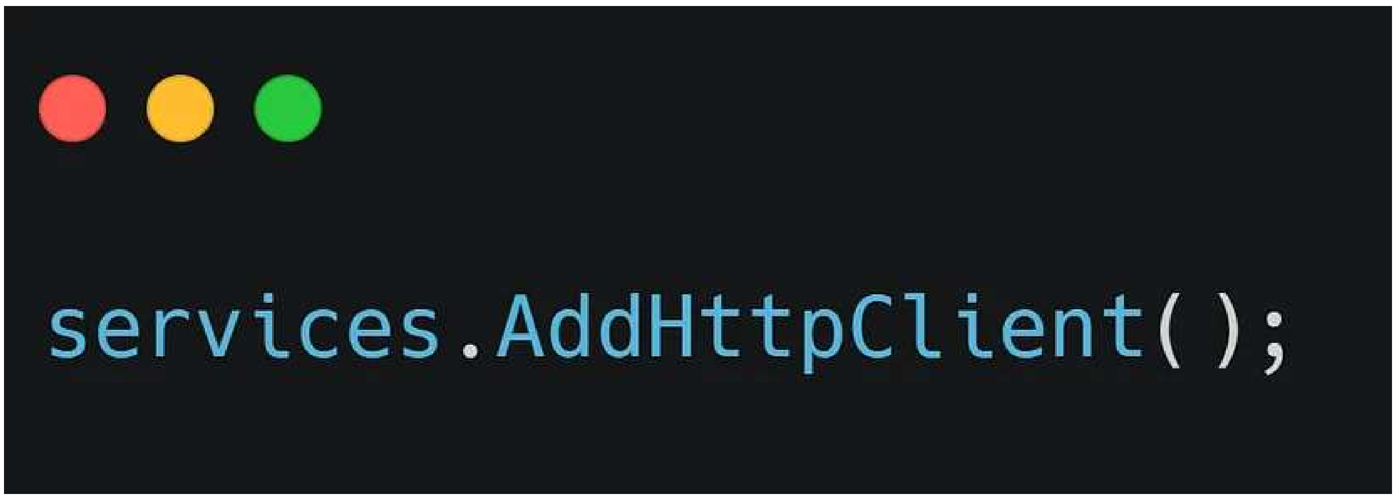
Stale DNS Entries:

HttpClient instances cache DNS entries. If the IP address of a service changes, HttpClient might still point to the old address.

HttpClientFactory: The Preferred Approach

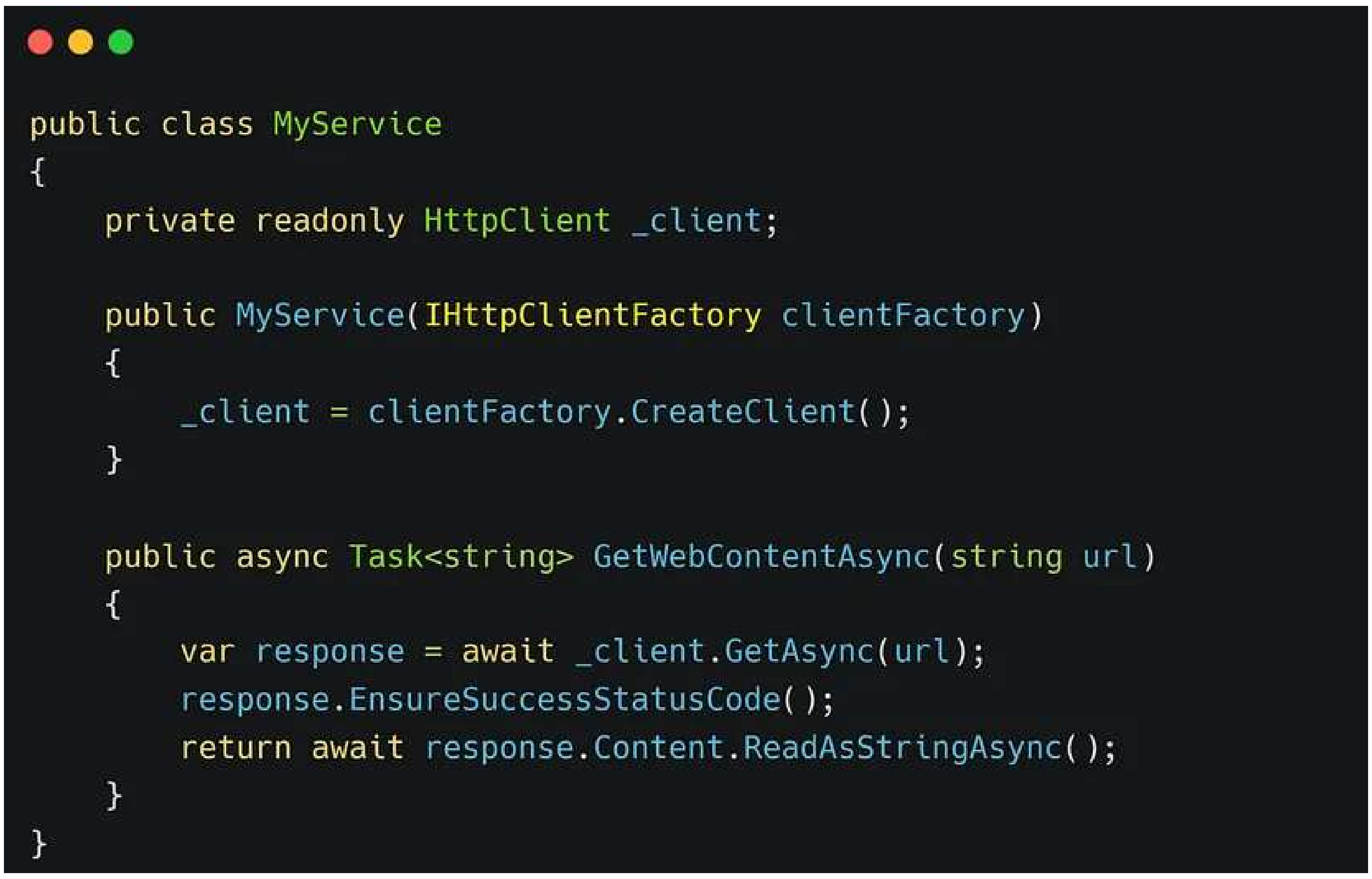
HttpClientFactory is designed to manage HttpClient instances, solving problems of socket exhaustion and providing better resource management.

By default, it creates HttpClient instances as Transient, but with a key difference: it reuses the underlying HttpMessageHandler, which manages the HTTP connections, effectively giving you **the benefits of a Singleton HttpClient** (connection pooling) without its drawbacks (DNS changes not respected).



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

Then you would inject **IHttpClientFactory** into the constructor of the class where you want to use HttpClient.

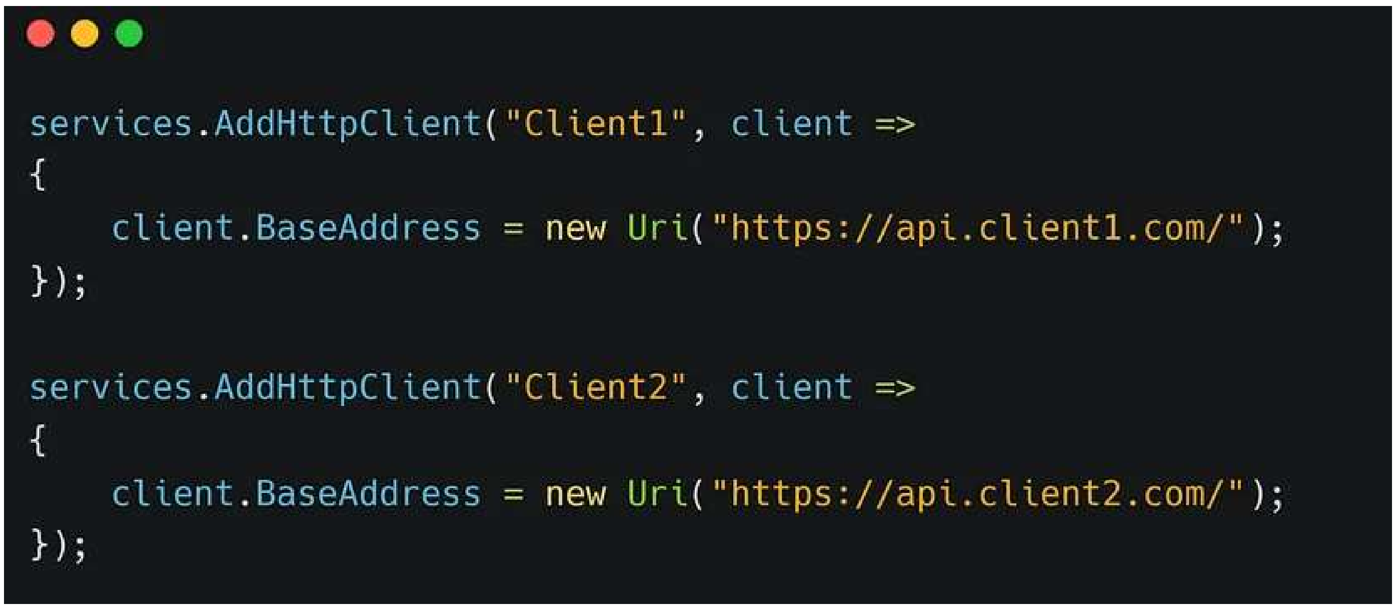


```
public class MyService
{
    private readonly HttpClient _client;

    public MyService(IHttpClientFactory clientFactory)
    {
        _client = clientFactory.CreateClient();
    }

    public async Task<string> GetWebContentAsync(string url)
    {
        var response = await _client.GetAsync(url);
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

This setup ensures that HttpClient instances are properly managed and reused, avoiding common pitfalls such as socket exhaustion and stale DNS entries.

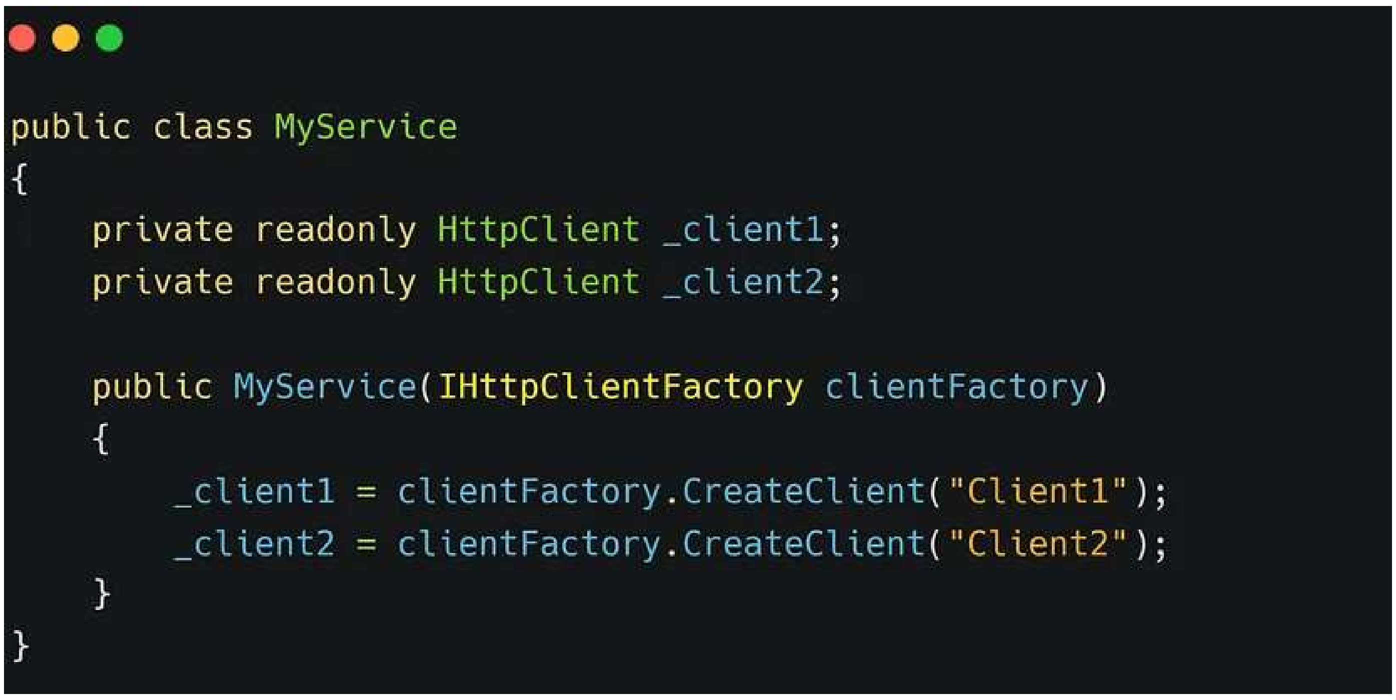
Named Clients: For Distinct Configurations

In situations where your application communicates with **multiple external services**, you can use “**Named Clients**” — these are pre-configured HttpClient instances associated with a string-based name. This allows for separate configurations (e.g., base address, default headers) for different services.



```
services.AddHttpClient("Client1", client =>
{
    client.BaseAddress = new Uri("https://api.client1.com/");
});

services.AddHttpClient("Client2", client =>
{
    client.BaseAddress = new Uri("https://api.client2.com/");
});
```

Using:



```
public class MyService
{
    private readonly HttpClient _client1;
    private readonly HttpClient _client2;

    public MyService(IHttpClientFactory clientFactory)
    {
        _client1 = clientFactory.CreateClient("Client1");
        _client2 = clientFactory.CreateClient("Client2");
    }
}
```

Advantages of using Named Clients:

1. Multiple Configurations:

Named clients allow for separate configurations for different HttpClients. For example, if your application communicates with different APIs that need different base addresses, headers, or other settings, named clients are the perfect solution.

2. Improved Readability:

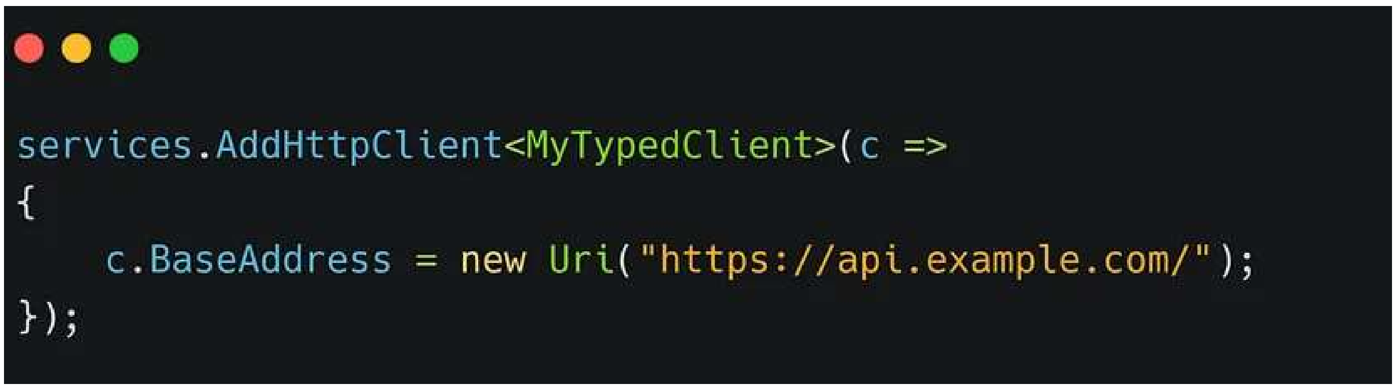
By naming the different configurations, the code becomes more readable, and it's easier to understand which configuration is being used.

When to avoid Named Clients:

If your application only communicates with a single endpoint, or all your HttpClient instances share the same configuration, named clients won't provide any advantages and will just add unnecessary complexity.

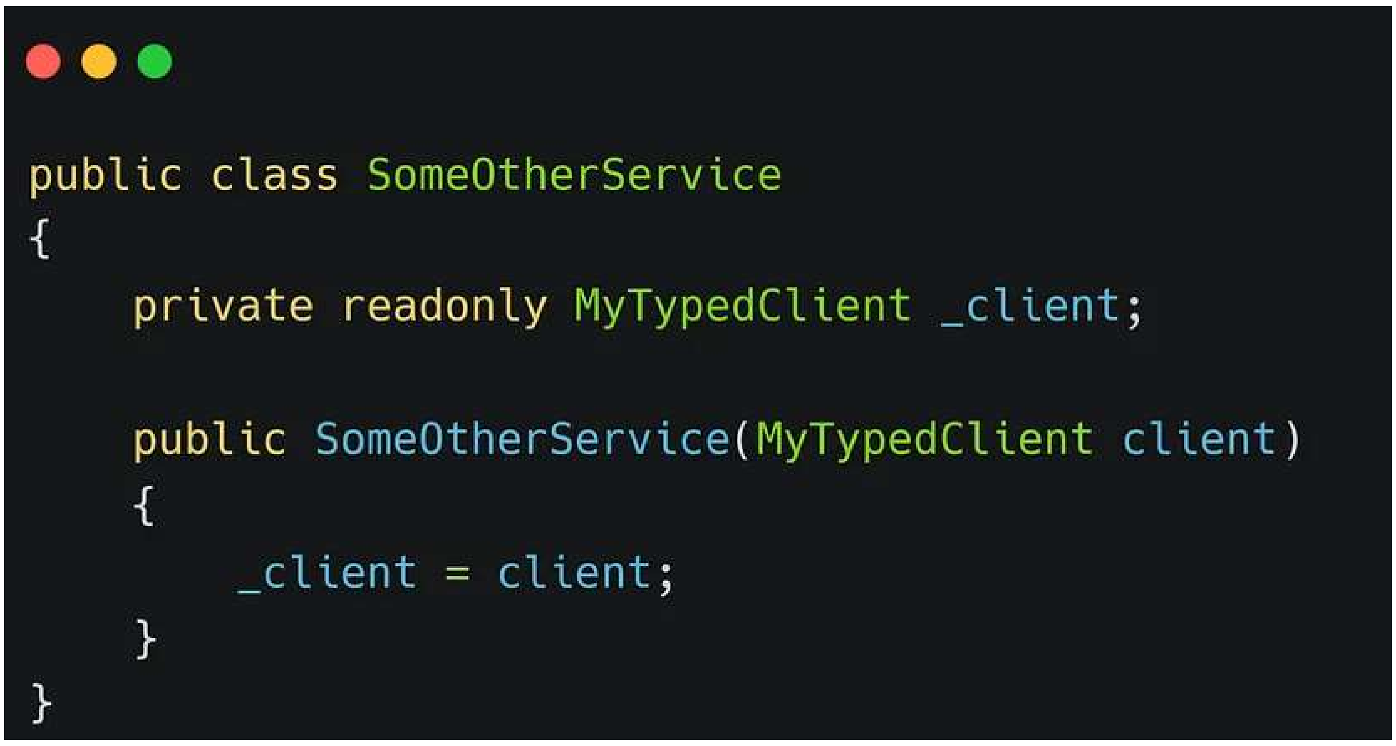
Typed Clients: For Encapsulation and Testability

Typed clients offer another level of abstraction. They encapsulate all interactions with the HttpClient within a class. This class is then injected wherever you need to perform HTTP requests, simplifying the code and making it easier to test.



```
services.AddHttpClient<MyTypedClient>(c =>
{
    c.BaseAddress = new Uri("https://api.example.com/");
});
```

Using:



```
public class SomeOtherService
{
    private readonly MyTypedClient _client;

    public SomeOtherService(MyTypedClient client)
    {
        _client = client;
    }
}
```

Advantages of using Typed Clients:

1. Encapsulation:

By wrapping all interactions with HttpClient in a class, you can encapsulate all the complexity and just expose the methods that your application requires.

2. Easier Testing:

By encapsulating the HttpClient usage in a class, it's easier to mock that class in your unit tests, improving testability.

When to avoid Typed Clients:

If your HttpClient usage is straightforward and doesn't involve complex logic, or if you prefer to have the HttpClient code directly in your services or controllers rather than in a separate class, you may not need to use typed clients.

Key Takeaways

1. Avoid creating new HttpClient instances manually for each request.
2. Leverage HttpClientFactory for managing HttpClient instances.
3. Use dependency injection to handle the lifecycle of HttpClient instances.
4. Consider using named clients when dealing with multiple services with differing configurations.
5. Consider typed clients to encapsulate HttpClient interactions, improving code readability and testability.
6. Always handle exceptions that HttpClient can throw.
7. Use Polly for Resilience

Choosing the right approach always depends on your specific use case, and each approach has its pros and cons. Always aim for maintainability, readability, and robustness when making your decision.

What next?

HttpClient provides a powerful interface for sending HTTP requests and receiving responses. Using it with dependency injection allows for better control over its lifetime and usage. HttpClientFactory provides a flexible and optimized way to handle HttpClient instances, mitigating common problems such as socket exhaustion.

That's all from me today. Shout-out to **Stefan.**

Support me by buying me a cup of coffee. **Donate Coffee** ☕ .

Stay Blessedd, Happy Coding -:)

Http Client Https Http Status Code Aspnetcore API

Open in app ↗

Search Medium

Follow

Written by Michael Schneider

102 Followers

I'm an indie full-stack developer and content creator building my version of the digital world one step at a time