

# Conexión con API externas

## ¿Qué es el HttpClient?

Microsoft tiene una descripción buena y breve del HttpClient en C#:

Proporciona una clase para enviar solicitudes HTTP y recibir respuestas HTTP de un recurso identificado por un URI.

Y eso es exactamente lo que hace esta clase.

HTTP es el fundamento mismo de la comunicación de datos en Internet. Cada vez que visitas una página, subes una foto de un perro adorable a las redes sociales, revisas tu correo electrónico y mucho más, se requiere que los datos se envíen y reciban a través de la World Wide Web.

Si trabajas con C# y necesitas recibir información o datos de una fuente de datos externa en línea, necesitas enviar una solicitud HTTP y ser capaz de recibir la respuesta. Antes del .NET Framework 4.5, usábamos HttpRequest y HttpResponseMessage para manejar estas solicitudes. Desde la versión 4.5 tenemos la clase HttpClient.

Entonces, ¿por qué no usar HttpResponseMessage y HttpRequest? Hay algunas diferencias entre estos y el HttpClient. Aquí hay algunas:

- HttpClient tiene mejor soporte asíncrono.
- El HttpClient es más fácil de usar y leer.
- HttpClient maneja automáticamente la descompresión del contenido de respuesta.
- HttpClient reutiliza automáticamente la conexión para múltiples solicitudes para un mejor rendimiento. HttpRequest tiene la propiedad KeepAlive, pero está deshabilitada por defecto.
- HttpClient incluye automáticamente un conjunto de encabezados predeterminados en cada solicitud, como los encabezados User-Agent, Accept y Connection. HttpRequest no proporciona encabezados predeterminados.

El HttpRequest todavía es utilizable en la versión más reciente de .NET y podría ser una buena opción en varios casos. Las clases y funciones más nuevas no siempre significan que las más antiguas sean malas.

Realmente depende de la situación cuando necesitas realizar solicitudes HTTP. Aquí hay algunas pautas básicas

Usa HttpClient cuando:

- Necesitas una API más simple y moderna para enviar solicitudes HTTP y recibir respuestas HTTP.
- Quieres realizar operaciones asíncronas utilizando el patrón async/await.

- Necesitas manejo automático de descompresión, reutilización de conexiones y encabezados predeterminados.
- Quieres aprovechar el soporte de HTTP/2.

### Usa `HttpWebRequest` cuando:

- Necesitas más control sobre la solicitud y respuesta HTTP y estás dispuesto a trabajar con una API de nivel más bajo.
- Necesitas admitir versiones más antiguas de .NET Framework (anteriores a 4.5), ya que `HttpWebRequest` existe desde .NET Framework 1.1.
- Necesitas realizar operaciones HTTP más avanzadas, como enviar solicitudes fragmentadas o usar métodos HTTP personalizados.
- Estás trabajando con un sistema heredado que requiere el uso de `HttpWebRequest`.

## Configurando el `HttpClient` en C#

Hay algunos tipos de solicitudes HTTP que describen lo que podrías querer hacer. En este tutorial, explicaré cómo puedes realizar una solicitud GET, POST y DELETE. Comencemos con la más simple de todas: GET.

Antes de poder hacer una solicitud HTTP, necesitamos configurar el `HttpClient`. Creo una aplicación de consola para escribir y probar el código. También instalo el paquete NuGet `Newtonsoft.Json` y lo usaré en el siguiente código.

El uso y la configuración del `HttpClient` viene en diferentes capas. Necesitamos inicializar la clase `HttpClient`, luego enviar la solicitud y verificar si esa solicitud fue exitosa. Si esperamos que se envíen datos de vuelta, necesitamos capturar esos datos y transformarlos en nuestro código.

Si creamos código para implementarlo, se vería así:

```
1 using (HttpClient client = new())
2 {
3     HttpResponseMessage response = await client.GetAsync("https://official-joke-
    api.appspot.com/random_ten");
4     if (response.IsSuccessStatusCode)
5     {
6         string content = await response.Content.ReadAsStringAsync();
7         List<Joke>? jokes = JsonConvert.DeserializeObject<List<Joke>>(content);
8         if (jokes == null)
9             return;
10        foreach (Joke joke in jokes)
11        {
12            Console.WriteLine(joke.Setup);
13            Console.WriteLine(joke.Punchline);
14            Console.WriteLine();
15        }
16    }
17 }
18
19 public class Joke
20 {
21     public string Setup { get; set; }
22     public string Punchline { get; set; }
23 }
24
```

Vamos a analizarlo:

En la línea 1, se inicializa el `HttpClient`. Estoy usando el `using` para que la inicialización del `HttpClient` se elimine cuando ya no lo necesite, cerrando también las conexiones. Luego, en la línea 3, envío la solicitud GET a la URL. Este es un método asíncronico, así que uso el `await`. En la línea 5, verifico si la respuesta fue exitosa. Es decir, que obtengo un código de estado HTTP positivo. Si no es exitoso, podría lanzar una excepción o algo así.

Pero es exitoso, así que puedo recuperar el contenido de la respuesta en la línea 7. El `response.Content.ReadAsStringAsync()` recupera la respuesta como una cadena, haciéndola perfecta para convertirla a JSON en la línea 9.

Y luego hemos terminado con el `HttpClient` en C#. Recorro la lista de chistes y los imprimo en mi pantalla.

Bastante simple, ¿verdad? Todo lo que necesitas hacer es tener una URL, inicialización de la clase `HttpClient`, y listo.

## Requests POST

Mientras que una solicitud GET es bastante simple, un POST necesita un poco más. Una solicitud POST te permite enviar datos desde un cliente a la fuente, como una API. Por lo tanto, la solicitud POST necesita un cuerpo con datos.

Este cuerpo suele ser una situación de clave-valor. La clave es un nombre de propiedad y el valor es... bueno, el valor de esa propiedad. Aparte del cuerpo, no hay nada especial al respecto. Incluso comienza igual, pero no esperamos que se devuelvan datos, por lo que no tenemos que capturar y deserializar datos.

```
1 using (HttpClient client = new())
2 {
3     Joke newJoke = new Joke
4     {
5         Setup = "Why do bees hum?",
6         Punchline = "Because they don't know the words."
7     };
8     StringContent body = new(JsonConvert.SerializeObject(newJoke));
9     body.Headers.ContentType = new MediaTypeHeaderValue("application/json");
10    HttpResponseMessage response = await client.PostAsync("https://official-joke-
    api.appspot.com/random_ten", body);
11    if (!response.IsSuccessStatusCode)
12    {
13        var content = await response.Content.ReadAsStringAsync();
14        throw new Exception(content);
15    }
16 }
17
18 public class Joke
19 {
20     public string Setup { get; set; }
21     public string Punchline { get; set; }
22 }
23
```

De nuevo, inicializo el `HttpClient`. En las líneas 3 a 7, creo un nuevo chiste, nada especial aquí. Pero en la línea 9, creo una nueva variable del tipo `StringContent` y la inicializo con el `newJoke` que se está convirtiendo a una cadena JSON. Debido a que estoy usando JSON como el cuerpo de la solicitud, necesito establecer el `ContentType` del cuerpo como `application/json`. Hago esto en la línea 10.

La razón por la que necesito establecer el `ContentType` es para que el destino sepa qué tipo de datos le enviaré. También podría ser XML y el destino no puede recibir XML si envío JSON. La solicitud PUT funciona exactamente igual, pero en lugar de usar `client.PostAsync`, usas `client.PutAsync`.

Hay muchos otros encabezados que puedes configurar, pero te contaré más sobre eso más adelante.

En la línea 12 hago el POST real a la URL. Esta vez uso el `client.PostAsync`. El primer parámetro es la URL, ya lo hicimos con el GET, y el segundo es el cuerpo, que creamos en la línea 9.

Con una solicitud POST, o PUT y DELETE, no es necesario verificar si la respuesta es exitosa o no. Más bien lo contrario; queremos saber cuándo salió mal. Por lo tanto, verifico si la respuesta no es exitosa en la línea 13. Si esto es cierto, obtengo el contenido de la solicitud. La mayoría de las veces, este es el error del destino que me dice qué hice mal.

Pongo esta información en una excepción para que me avisen cuando algo sale mal al publicar información en el destino. Esta información podría ser que una propiedad requerida está vacía, el endpoint es incorrecto, etc.

## Requests DELETE

La solicitud DELETE se parece mucho a la solicitud POST, pero con una ligera diferencia: no esperamos recibir datos del destino. Si ya creaste el código para el POST, puedes simplemente copiarlo y pegarlo y cambiarlo un poco a esto:

```
1 using (HttpClient client = new())
2 {
3     HttpResponseMessage response = await
4 client.DeleteAsync($"https://api.com/random_ten/12");
5     {
6         var content = await response.Content.ReadAsStringAsync();
7         throw new Exception(content);
8     }
9 }
10
```

De nuevo, inicializo el `HttpClient` en la línea 1. En la línea 3 envío la solicitud DELETE a un destino. Las solicitudes DELETE generalmente necesitan un identificador en la URL, de ahí el '12' en este ejemplo de URL.

Solo queremos saber cuándo la solicitud ha fallado y envía una respuesta con un código de estado de error, así que solo queremos verificar si la respuesta no es exitosa en la línea 5. Para averiguar qué salió mal, podemos extraer el error leyendo el contenido de la respuesta y ponerlo en una excepción.

## Descargar una imagen

Puedes usar el `HttpClient` para descargar una imagen de la web y almacenarla en un array de bytes y luego guardarla como un archivo. Para esto, usamos el método `GetByteArrayAsync(uri)`.

```
1 using (HttpClient httpClient = new())
2 {
3     byte[] imageBytes = await httpClient.GetByteArrayAsync("https://i.imgur.com/OVFxdJy.jpg");
4
5     string desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
6
7     string localPath = Path.Combine(desktopPath, "this_makes_your_day.jpg");
8
9     File.WriteAllBytes(localPath, imageBytes);
10 }
```

## HttpClient Factory

### Definición y utilidad

¿Por qué evitar la creación manual de `HttpClient`?

En .NET, crear una instancia de `HttpClient` manualmente para cada solicitud, como se muestra a continuación, puede causar problemas:

```
public async Task<string> GetDataAsync()
{
    using var client = new HttpClient();
    var response = await client.GetAsync("https://api.example.com/data");
    return await response.Content.ReadAsStringAsync();
}
```

Problemas con el enfoque anterior:

- Agotamiento de Sockets: Las instancias de HttpClient usan un pool de conexiones subyacentes. Cuando creas una nueva instancia para cada solicitud y la eliminas de inmediato, las conexiones de socket subyacentes no se cierran al instante, lo que lleva a un agotamiento de sockets bajo una carga elevada.
- DNS obsoleto: Las instancias de HttpClient guardan en caché las entradas de DNS. Si reutilizas la misma instancia durante un período prolongado, los registros de DNS pueden volverse obsoletos, lo que podría hacer que las solicitudes fallen si la dirección IP del endpoint (punto final) cambia.
- Rendimiento: La frecuente creación y eliminación de instancias de HttpClient genera una sobrecarga innecesaria.

## ¿Por qué usar HttpClientFactory?

HttpClientFactory, introducido en ASP.NET Core 2.1, proporciona una forma centralizada de configurar y gestionar instancias de HttpClient. Ofrece:

- Agrupamiento y Reutilización: Reutilización gestionada de objetos HttpClientMessageHandler para evitar el agotamiento de sockets.
- Configuración y Personalización: Configuración centralizada de HttpClient para diferentes endpoints (puntos finales).
- Políticas de Resiliencia: Integración con Polly para reintentos, disyuntores (circuit breakers) y más."

# Implementación

La implementación de un HttpClientFactory consta de dos pasos

## 1. Registrarlo en el program.cs

Configurar la definición del HttpClient Factory. Las especificaciones que declaramos aca van a aplicar para todos los clientes instanciados con el HttpClient Factory

```
19
20  builder.Services.AddHttpClient(
21      "pokeHttpClient",
22      client =>
23      {
24          client.BaseAddress = new Uri("https://pokeapi.co/api/v2/");
25      });
26
```

## 2. Inyectar y usarlo en el servicio

Primero inyectamos la interfaz IHttpClientFactory (cambios marcados en amarillo)  
Luego instanciamos un HttpClient (definida como variable del servicio) usando el HttpClientFactory con el CreateClient

```
public class PokemonAPIService
{
    private readonly IHttpClientFactory httpClientFactory = null!;
    public HttpClient pokeClient { get; set; }

    public PokemonAPIService(IHttpClientFactory httpClientFactory)
    {
        _httpClientFactory = httpClientFactory;
        string? httpClientName = "pokeHttpClient";
        pokeClient = _httpClientFactory.CreateClient(httpClientName ?? "");
    }

    public async Task<GetPokeByIdResponse> GetBerryAsync(int id)
    {
        return await pokeClient.GetFromJsonAsync<GetPokeByIdResponse>($"berry/{id}");
    }
}
```

# Resilience

## Estrategias de Resiliencia con Polly en .Net

En el mundo interconectado de hoy, muchas aplicaciones dependen de APIs y servicios externos. Sin embargo, las llamadas de red pueden fallar por diversas razones, como problemas temporales en la red, sobrecargas del servidor o tiempo de inactividad por mantenimiento. Para construir aplicaciones robustas y tolerantes a fallos, es crucial implementar patrones de resiliencia. En este documento explicaremos cómo usar Polly, una biblioteca de resiliencia y manejo de fallos transitorios para .NET, para crear un cliente HTTP resiliente utilizando patrones de reintento (retry) y disyuntor (circuit breaker).

### Patrón Retry de Polly (Polly Retry Handler)

El patrón Retry es una forma simple pero efectiva de manejar fallos transitorios. Cuando una solicitud falla, en lugar de lanzar una excepción de inmediato, la aplicación espera un breve período y luego reintentar la solicitud. Esto a menudo puede resolver problemas causados por fallos temporales en la red o breves indisponibilidades del servicio.

El handler del Retry de Polly te permite especificar:

- El número de intentos del Retry.
- El retraso entre Retries (que puede ser fijo o aumentar con cada intento).
- Los tipos de excepciones o resultados de retorno que deben activar un Retry.

En nuestro ejemplo, estamos utilizando una estrategia de retroceso exponencial (exponential backoff), donde el retraso aumenta exponencialmente con cada intento de Retry.

### Circuit Breaker (Circuit Breaker Policy)

El patrón de Circuit Breaker evita que una aplicación intente repetidamente ejecutar una operación que probablemente fallará. Es como un disyuntor eléctrico: cuando se detecta un problema, se "dispara" y detiene el flujo para proteger el sistema.

El Circuit Breaker tiene tres estados:

- Cerrado (Closed): Cuando todo es normal, el circuito está cerrado y se permiten las solicitudes.
- Abierto (Open): Si el número de fallos supera un umbral, el circuito se abre y las solicitudes no se permiten durante un cierto período.

- Semi-abierto (Half-Open): Después del período de tiempo de espera, el circuito cambia a semi-abierto, permitiendo un número limitado de solicitudes para probar si el problema aún persiste.

Este patrón es particularmente útil para prevenir fallos en cascada en sistemas distribuidos.

## Implementación de clientes HTTP resilientes

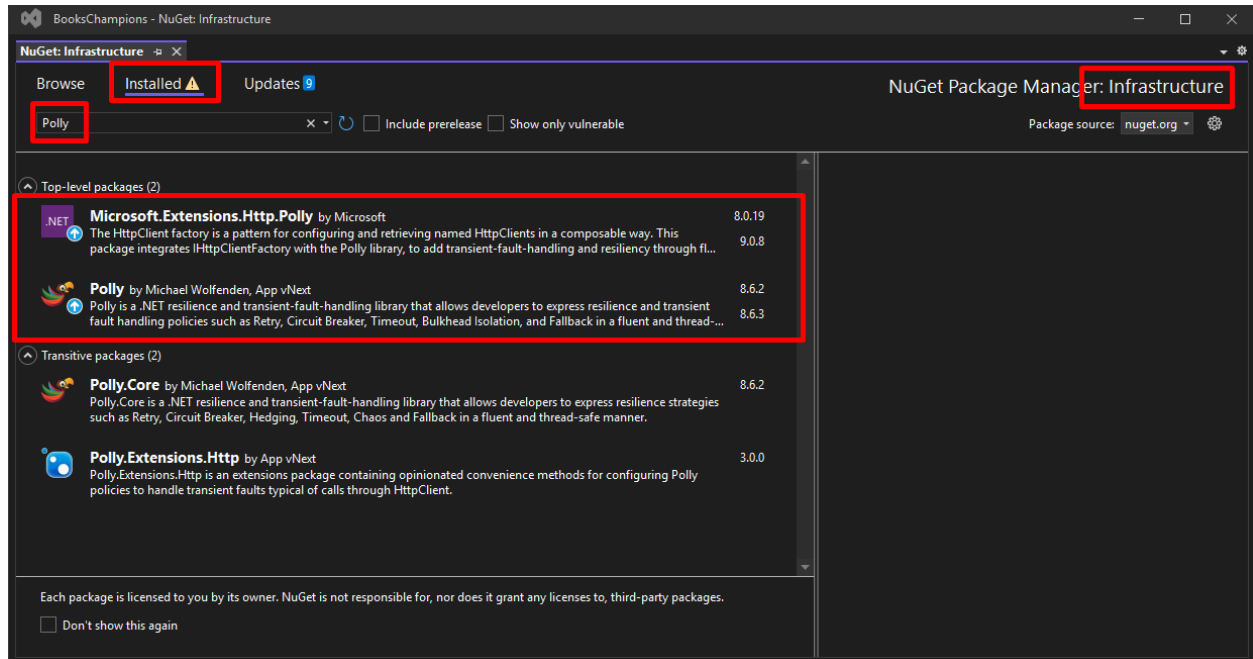
### Instalación de Paquetes

Necesitamos instalar los siguientes paquetes nuget:

```
dotnet add package Polly
dotnet add package Microsoft.Extensions.Http.Polly
```

También se pueden instalar por UI en Visual Studio.

Los paquetes se deben instalar en el proyecto en donde se definan las políticas. Si estamos siguiendo un patrón de clean Architecture sería en el proyecto de Infraestructura.



### Definición de Políticas

Las políticas las definimos en el proyecto de Infraestructura. Por lo general con una única clase en donde se agrupen todas las políticas definidas es más que suficiente pero si el escenario es muy complejo se puede optar por una clase por servicio externo del cual nuestro backend dependa.

En la siguiente captura se podrá ver dos métodos en donde en cada uno armamos una Policy para el patrón Retry and Wait y otra para el patrón Circuit Breaker:

El método que crea una política de Retry específica:

- Maneja errores HTTP transitorios (errores de red, códigos de estado 5xx y 400).
- Retries un número específico de veces.
- Utiliza retroceso exponencial para el retraso entre Retries.

Este método crea una política de Circuit Breaker establece:

- Maneja los mismos errores HTTP transitorios que la política de Retry.
- Abre el circuito después de un número específico de fallos consecutivos.
- Mantiene el circuito abierto durante una duración específica antes de permitir una solicitud de prueba.

La clase debe ser estática para poder ser usada en las configuraciones del program porque la vamos a necesitar antes de que la aplicación se termine de buildear por lo que la inyección no es una opción de esta manera la única opción es una clase estática

```
9 namespace Infrastructure
10 {
11     public static class PollyResiliencePolicies
12     {
13         public static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy(ApiClientConfiguration config)
14         {
15             return HttpPolicyExtensions
16                 .HandleTransientHttpError()
17                 .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.BadRequest)
18                 .WaitAndRetryAsync(
19                     config.RetryCount,
20                     retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt) * config.RetryAttemptInSeconds)
21                 );
22         }
23
24         public static IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy(ApiClientConfiguration config)
25         {
26             return HttpPolicyExtensions
27                 .HandleTransientHttpError()
28                 .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.BadRequest)
29                 .CircuitBreakerAsync(config.HandledEventsAllowedBeforeBreaking, TimeSpan.FromMinutes(config.DurationOfBreakInSeconds));
30         }
31     }
32 }
33
```

También definimos una clase de configuración con los parámetros que nos permitirá reutilizar los métodos para cada vendor cuando las usemos después

```

7  namespace Infrastructure
8  {
9      public class ApiClientConfiguration
10     {
11         public int RetryCount { get; set; }
12         public int RetryAttemptInSeconds { get; set; }
13         public int HandledEventsAllowedBeforeBreaking { get; set; }
14         public int DurationOfBreakInSeconds { get; set; }
15     }
16 }

```

## Implementación de las Políticas

La implementación se hace mediante la modificación de los HttpClient Factories explicados en la sección anterior (en el [Program.cs](#) en la capa de presentación). La única diferencia es que añadimos las políticas definidas en nuestra clase estática anterior.

Lo bueno de esta implementación es que podemos implementar diferentes políticas para diferentes httpClient factories lo que tiene mucha lógica ya que los HttpClient factories se crean para cada vendor o servicio externo y cada uno puede tener errores específicos, tiempos de recuperación y fallas comunes a los que es correcto responder con Políticas acordes a sus realidades.

De esta manera también instanciamos un objeto de configuración que definimos junto con la clase estática para asignarle los valores configurables de nuestras políticas específicas para el servicio en cuestión. Con este enfoque podríamos reutilizar nuestras políticas para diferentes servicios pero usando diferentes parámetros para cada uno lo que nos permitirá ajustarnos a la realidad de cada uno.

```

17  var builder = WebApplication.CreateBuilder(args);
18
19  #region HTTPClientFactories
20
21  ApiClientConfiguration pokeApiResilienceConfiguration = new()
22  {
23      RetryCount = 2,
24      RetryAttemptInSeconds = 3,
25      DurationOfBreakInSeconds = 120,
26      HandledEventsAllowedBeforeBreaking = 10
27  };
28
29  builder.Services.AddHttpClient(
30      "pokeHttpClient",
31      client =>
32      {
33          client.BaseAddress = new Uri("https://pokeapi.co/api/v2/");
34      })
35      .AddPolicyHandler(PollyResiliencePolicies.GetRetryPolicy(pokeApiResilienceConfiguration))
36      .AddPolicyHandler(PollyResiliencePolicies.GetCircuitBreakerPolicy(pokeApiResilienceConfiguration));
37
38  #endregion

```

## Conclusión

Al implementar los patrones de Retry y Circuit Breaker con Polly, podemos crear aplicaciones más resilientes que pueden manejar mejor los fallos transitorios y prevenir fallos en cascada en sistemas distribuidos. Este enfoque no solo mejora la fiabilidad de nuestras aplicaciones, sino que también optimiza la experiencia del usuario al reducir el impacto de problemas temporales.

Recuerda, aunque estos patrones pueden mejorar significativamente la resiliencia de tu aplicación, no son una solución mágica. Es importante combinarlos con otras buenas prácticas como el registro de eventos (logging), la monitorización y las alertas para crear sistemas verdaderamente robustos.

El ejemplo de esta implementación se encuentra en este repo y branch:

[gabrielgolzman/austral-book-champions-api at Feat-external-api](#)