Unit test HttpClient with Polly

Asked 4 years, 8 months ago Modified 2 years, 2 months ago Viewed 10k times

4

I'm looking to unit test a `HttpClient` that has a `Polly` `RetryPolicy` and I'm trying to work out how to control what the `HTTP` response will be.

I have used a `HttpMessageHandler` on the client and then override the Send Async and this works great but when I add a Polly Retry Policy I have to create an instance of HTTP Client using the `IServiceCollection` and cant create a `HttpMessageHandler` for the client. I have tried using the `.AddHttpMessageHandler()` but this then blocks the Poll Retry Policy and it only fires off once.

This is how I set up my HTTP client in my test

```
IServiceCollection services = new ServiceCollection();

const string TestClient = "TestClient";

services.AddHttpClient(name: TestClient)
         .AddHttpMessageHandler()
         .SetHandlerLifetime(TimeSpan.FromMinutes(5))

.AddPolicyHandler(KYA_GroupService.ProductMessage.ProductMessageHandler.GetRetryPo

HttpClient configuredClient =
                services
                    .BuildServiceProvider()
                    .GetRequiredService<IHttpClientFactory>()
                    .CreateClient(TestClient);

public static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions
            .HandleTransientHttpError()
            .WaitAndRetryAsync(6,
                retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
retryAttempt)),
                onRetryAsync: OnRetryAsync);
}

private async static Task OnRetryAsync(DelegateResult<HttpResponseMessage>
outcome, TimeSpan timespan, int retryCount, Context context)
{
    //Log result
}
```

This will then fire the request when I call `_httpClient.SendAsync(httpRequestMessage)` but it actualy create a Http call to address and I need to intercept this some how and return a controlled response.

I would like to test that the policy is used to retry the request if the request fails and completes when it is a complete response.

The main restriction I have is I can't use Moq on MSTest.

c# unit-testing dotnet-httpclient polly retry-logic

Share

Improve this question

Follow

edited Oct 11, 2022 at 17:15
Peter Csala
22.5k ● 16 ● 47 ● 91

asked Apr 16, 2020 at 15:57
David Molyneux
304 ● 1 ● 2 ● 12

2 Answers

Sorted by: Highest score (default)

9

You don't want your `HttpClient` to be issuing real HTTP requests as part of a unit test - that would be an integration test. To avoid making real requests you need to provide a custom `HttpMessageHandler`. You've stipulated in your post that you don't want to use a mocking framework, so rather than mocking `HttpMessageHandler` you could provide a stub.

With heavy influence from this comment on an issue on Polly's GitHub page, I've adjusted your example to call a stubbed `HttpMessageHandler` which throws a 500 the first time it's called, and then returns a 200 on subsequent requests.

The test asserts that the retry handler is called, and that when execution steps past the call to `HttpClient.SendAsync` the resulting response has a status code of 200:

```
public class HttpClient_Polly_Test
{
    const string TestClient = "TestClient";
    private bool _isRetryCalled;

    [Fact]
    public async Task
Given_A_Retry_Policy_Has_Been_Registered_For_A_HttpClient_When_The_HttpRequest_F
    {
        // Arrange
        IServiceCollection services = new ServiceCollection();
        _isRetryCalled = false;

        services.AddHttpClient(TestClient)
            .AddPolicyHandler(GetRetryPolicy())
            .AddHttpMessageHandler(() => new StubDelegatingHandler());

        HttpClient configuredClient =
            services
                .BuildServiceProvider()
                .GetRequiredService<IHttpClientFactory>()
                .CreateClient(TestClient);

        // Act
        var result = await
configuredClient.GetAsync("https://www.stackoverflow.com");

        // Assert
        Assert.True(_isRetryCalled);
        Assert.Equal(HttpStatusCode.OK, result.StatusCode);
    }

    public IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
    {
        return HttpPolicyExtensions.HandleTransientHttpError()
            .WaitAndRetryAsync(
                6,
                retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
retryAttempt)),
                onRetryAsync: OnRetryAsync);
    }

    private async Task OnRetryAsync(DelegateResult<HttpResponseMessage>
outcome, TimeSpan timespan, int retryCount, Context context)
    {
        //Log result
        _isRetryCalled = true;
    }
}

public class StubDelegatingHandler : DelegatingHandler
{
    private int _count = 0;

    protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage
request,
        CancellationToken cancellationToken)
    {
        if (_count == 0)
        {
            _count++;
            return Task.FromResult(new
HttpResponseMessage(HttpStatusCode.InternalServerError));
        }

        return Task.FromResult(new HttpResponseMessage(HttpStatusCode.OK));
    }
}
```

Share

Improve this answer

Follow

edited Apr 17, 2020 at 7:10

answered Apr 16, 2020 at 18:36

simon-pearson

1,940 ● 10 ● 11

Thanks, I had tried something similer but it didn't work but it seems that the order you have AddPolicyHandler and AddHttpMessageHandler matter. AddHttpMessageHandler first and it will not run the retry policy, switch them round so AddHttpMessageHandler is the last call and it works as expected – David Molyneux Apr 17, 2020 at 8:24

This is almost ok but by adding "stackoverflow.com" you unit test alsi the connection tot that endpoint, which is an external dependency. What happen if your unit tests run on a machine which is not allowed to do http requests outside the intranet ? – cozmin-calin Jan 4, 2021 at 11:00

1

The answer above was very helpful in getting me on the right track. However I wanted to test that Policies had been added to a typed http client. This client is defined at application startup. So the challenge was how to add a stub delegating handler after the handlers specified in the typed client definition and it had been added to the Services collection.

I was able to leverage IHttpMessageHandlerBuilderFilter.Configure and added my stub handler as the last handler in the chain.

```
public sealed class HttpClientInterceptionFilter :
IHttpMessageHandlerBuilderFilter
{
    HandlerConfig handlerconfig { get; set; }

    public HttpClientInterceptionFilter(HandlerConfig calls)
    {
        handlerconfig = calls;
    }
    /// <inheritdoc/>
    public Action<HttpMessageHandlerBuilder>
Configure(Action<HttpMessageHandlerBuilder> next)
    {
        return (builder) =>
        {
            // Run any actions the application has configured for itself
            next(builder);

            // Add the interceptor as the last message handler
            builder.AdditionalHandlers.Add(new
StubDelegatingHandler(handlerconfig));
        };
    }
}
```

Register this class with the DI container in your unit test:

```
services.AddTransient<IHttpMessageHandlerBuilderFilter>(n => new
HttpClientInterceptionFilter(handlerConfig));
```

I needed pass in some parameters to the stub handler and to get data out of it and back to my unit test. I used this class to do so:

```
public class HandlerConfig
{
    public int CallCount { get; set; }
    public DateTime[] CallTimes { get; set; }
    public int BackOffSeconds { get; set; }
    public ErrorTypeEnum ErrorType { get; set; }
}

public enum ErrorTypeEnum
{
    Transient,
    TooManyRequests
}
```

My stub handler generates transient and too many request responses:

```
public class StubDelegatingHandler : DelegatingHandler
{
    private HandlerConfig _config;
    HttpStatusCode[] TransientErrors = new HttpStatusCode[] {
HttpStatusCode.RequestTimeout, HttpStatusCode.InternalServerError,
HttpStatusCode.OK };

    public StubDelegatingHandler(HandlerConfig config)
    {
        _config = config;
    }
    protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage
request,
        CancellationToken cancellationToken)
    {
        _config.CallTimes[_config.CallCount] = DateTime.Now;

        if (_config.ErrorType == ErrorTypeEnum.Transient)
        {
            var response = new
HttpResponseMessage(TransientErrors[_config.CallCount]);
            _config.CallCount++;
            return Task.FromResult(response);
        }

        HttpResponseMessage response429;
        if (_config.CallCount < 2)
        {
            //generate 429 errors
            response429 = new
HttpResponseMessage(HttpStatusCode.TooManyRequests);
            response429.Headers.Date = DateTime.UtcNow;

            DateTimeOffset dateTimeOffSet = DateTimeOffset.UtcNow.Add(new
TimeSpan(0, 0, 5));
            long resetDateTime = dateTimeOffSet.ToUnixTimeSeconds();
            response429.Headers.Add("x-rate-limit-reset",
resetDateTime.ToString());
        }
        else
        {
            response429 = new HttpResponseMessage(HttpStatusCode.OK);
        }

        _config.CallCount++;
        return Task.FromResult(response429);

    }
}
```

And finally the unit test:

```
[TestMethod]
public async Task
Given_A_429_Retry_Policy_Has_Been_Registered_For_A_HttpClient_When_429_Errors_Oc
    {
        // Arrange
        IServiceCollection services = new ServiceCollection();

        var handlerConfig = new HandlerConfig { ErrorType =
ErrorTypeEnum.TooManyRequests, BackOffSeconds = 5, CallTimes = new
System.DateTime[RetryCount] };

        // this registers a stub message handler that returns the desired error
codes
        services.AddTransient<IHttpMessageHandlerBuilderFilter>(n => new
HttpClientInterceptionFilter(handlerConfig));

        services.ConfigureAPIClient();  //this is an extension method that adds
a typed client to the services collection

        HttpClient configuredClient =
            services
                .BuildServiceProvider()
                .GetRequiredService<IHttpClientFactory>()
               .CreateClient("APIClient");  //Note this must be the same name
used in ConfigureAPIClient

        //  Act
        var result = await configuredClient.GetAsync("https://localhost/test");

        //   Assert
        Assert.AreEqual(3, handlerConfig.CallCount, "Expected number of  calls
made");
        Assert.AreEqual(HttpStatusCode.OK, result.StatusCode, "Verfiy status
code");

        var actualWaitTime = handlerConfig.CallTimes[1] -
handlerConfig.CallTimes[0];
        var expectedWaitTime = handlerConfig.BackOffSeconds + 1;
//ConfigureAPIClient adds one second to give a little buffer
        Assert.AreEqual(expectedWaitTime, actualWaitTime.Seconds);
    }
}
```

Share Improve this answer Follow

answered Dec 29, 2021 at 18:28

whitemtnelf
181 ● 2 ● 11

you seem to be using a named, not a typed http client. – more urgent jest Jul 18, 2022 at 23:27