**Building Resilient HTTP Clients in C# with Polly**

Building integrations to 3rd party APIs at scale can be difficult. Of course it would be easy enough to just fire off an HttpRequestMessage using an HttpClient.

But what happens when the 3rd party is temporarily down or unresponsive?

What happens when there is a transient HTTP error?

We need to be able to resend the HTTP request again in the same context. We could try to write our own retry logic.

Instead of trying to reinvent the wheel, we will use the open source [.NET Polly](https://github.com/App-vNext/Polly) project and nuget package.

**Polly is a .NET resilience and transient-fault-handling library that allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, Rate-limiting and Fallback in a fluent and thread-safe manner.**

Additionally, we want to be able to make our methods that rely on Polly unit testable. We will use dependency injection to allow us to unit test our methods and make sure that Polly is performing as expected.

**Overview of the Project that We Will Build to Demonstrate HTTP Client Resiliency in C#**

**We will be using a**[**Test API specifically for Developers**](https://www.webdevtutor.net/blog/list-of-unique-test-apis-for-developers)**called [Reqres](https://reqres.in/)**.

We will create a new solution in Visual Studio with 2 projects, one for the library (or logic) code and 1 for our tests.

We will be using .NET 6 (the next generation of .NET Core) for this guide.

The most important goals for this guide are:

1. Build Resiliency in our HTTP Clients using Polly
2. Allow Our API Client to Be Unit Testable Using Dependency Injection

**Creating the Library and Test Projects in Visual Studio 2022**

If you don't already have Visual Studio 2022 installed on your computer, we recommend [Downloading Visual Studio 2022 from Microsoft](https://visualstudio.microsoft.com/vs/).

Select the Community edition, it's 100% free.

This project will still work even if you have an older version of Visual Studio.

We also recommend you download the [.NET 6 SDK](https://dotnet.microsoft.com/en-us/download/dotnet/6.0) if it isn't already on your computer

Let's create the first project. We want to make a C# class library on .NET 6.0 called ResilientAPIsWithPolly.

Next, let's add the Test Project to our solution. We will create a new XUnit .NET Core Unit Testing Project called: ResilientAPIsWithPolly.Tests.

## **Writing a Basic Class that Sends HTTP Requests using HTTP Client**

Before we get into adding resiliency to our HTTP Client using Polly, let's first make a basic class that can send HttpRequestMessage objects using the .NET HttpClient.

If you started from the beginning of the tutorial, the original class in our ResilientAPIsWithPolly library project has a class called Class1.cs.

### **Renaming the Class to ReqresAPIClient**

Let's rename that class to ReqresAPIClient. Remember we are going to be using the test API Reqres simply for an API that will give us basic responses back!

After we rename our class, the class will look like this:

**namespace** **ResilientAPIsWithPolly**

{

**public** **class** **ReqresApiClient**

{

}

}

### **Adding A Constructor That Accepts An HTTPClient and Adding the Base URL for the Reqres Test API**

It's a pretty empty class at this point, so let's add a constructor. We want our constructor to require an HttpClient instance (via Dependency Injection) so that we can Unit Test this class. Keep following along and we'll go into Unit Testing the class further down in the guide!

Reqres provides a test API endpoint that will return test data.

Here is what our class looks like after adding the constructor and the Base URL for the Test API:

**namespace** **ResilientAPIsWithPolly**

{

**public** **class** **ReqresApiClient**

{

**private** **const** string \_reqresAPIBaseUrl = "https://reqres.in/api";

**private** **readonly** HttpClient \_httpClient;

**public** **ReqresApiClient**(HttpClient httpClient)

{

\_httpClient = httpClient;

}

}

}

Our class is now ready to start sending API requests to one of the Reqres test endpoints!

### **Add a Method for Requesting a Single User From The Test API**

Next we'll add a new method that will allow us to send a request to the test API to retrieve data about a specific test user.

The endpoint that we will be calling is https://reqres.in/api/users/1.

This endpoint will give us a JSON request body that looks like this:

{

"data": {

"id": 1,

"email": "george.bluth@reqres.in",

"first\_name": "George",

"last\_name": "Bluth",

"avatar": "https://reqres.in/img/faces/1-image.jpg"

},

"support": {

"url": "https://reqres.in/#support-heading",

"text": "To keep ReqRes free, contributions towards server costs are appreciated!"

}

}

Let's add in the method for requesting an user, we'll create the method with a parameter that requires the caller to provide an User ID.

After we add the new method, our class looks something like this:

**namespace** **ResilientAPIsWithPolly**

{

**public** **class** **ReqresApiClient**

{

**private** **const** string \_reqresAPIBaseUrl = "https://reqres.in/api";

**private** **readonly** HttpClient \_httpClient;

**public** **ReqresApiClient**(HttpClient httpClient)

{

\_httpClient = httpClient;

}

**public** **async** Task<HttpResponseMessage> **GetUser**(int userId)

{

**var** userUrl = $"{\_reqresAPIBaseUrl}/users/{userId}";

**var** httpResponseMessage = **await** \_httpClient.GetAsync(userUrl);

**return** httpResponseMessage;

}

}

}

We now have the ability to retrieve a basic test user's information via the test API.

In the next section we will test to make sure that our current RegresApiClient actually gives us the expected response.

## **Testing Our ReqresApiClient Class to Confirm it Sends HTTP Requests using HTTP Client Successfully**

The easiest way to test that our new ReqresApiClient class is able to successfully retrieve an User via the Test API is to write an Integration Test against the GetUser method.

We'll temporarily move our focus away from our library project, and into our test project.

### **Adding a Reference to Our Library Project in Our Test Project**

In order to test our ReqresAPIClient class, we need to add a new reference to our ResilientAPIsWithPolly library project within our ResilientAPIsWithPolly.Tests project.

To do this, inside of the Solution Explorer expand the Dependencies section under our test project.

Then right click on Project and click Add Project Reference... then tick the checkbox for the ResilientAPIsWithPolly project and click OK.

We will now be able to call our library project from within our test project.

### **Renaming Our Test Class in The ResilientAPIsWithPolly.Tests Project**

Remember at the beginning of this guide when we created an XUnit .NET Core project to test our new ReqresAPIClient class?

It initially has one single class called UnitTest1.cs. Because we want to write Integration Tests against the class, let's rename that class to the class under test + "IntegrationTests", in this case the class will be called ReqresApiClientIntegrationTests.

Here is what our new integration test class will look like after we rename it:

**using** Xunit;

**namespace** **ResilientAPIsWithPolly.Tests**

{

**public** **class** **ReqresApiClientIntegrationTests**

{

[Fact]

**public** **void** **Test1**()

{

}

}

}

### **Writing an Integration Test to Confirm the API is Working Properly with HTTPClient**

Our new test class, ReqresApiClientIntegrationTests, is pretty empty. Let's add an integration test to make sure our HttpClient is working as expected.

We'll add a method called GetUserShouldReturnASuccessStatusCode.

We'll add a static HttpClient to the top of our test class and then inject the HttpClient instance into the constructor when we instantiate an instance of the ReqresApiClient class that we will be testing.

Here is what the integration test class will look like after adding the GetUserShouldReturnASuccessStatusCode method into our test class:

**using** System.Net.Http;

**using** System.Threading.Tasks;

**using** Xunit;

**namespace** **ResilientAPIsWithPolly.Tests**

{

**public** **class** **ReqresApiClientIntegrationTests**

{

**private** **static** HttpClient \_httpClient = **new** HttpClient();

[Fact]

**public** **async** Task **GetUserShouldReturnASuccessStatusCode**()

{

// Arrange

**var** userId = 1;

**var** resilientApisWithPolly = **new** ReqresApiClient(\_httpClient);

// Act

**var** httpResponseMessage = **await** resilientApisWithPolly.GetUser(userId);

// Assert

Assert.True(httpResponseMessage.IsSuccessStatusCode);

}

}

}

Right click on the test method (GetUserShouldReturnASuccessStatusCode) and click Run Tests.

The solution will build and the test will run. If the test succeeds we should see a green check mark right above the GetUserShouldReturnASuccessStatusCode method.

This proves that our current code can successfully call the API!

## **Adding Resiliency To Our API Client Using Polly**

If you've made it this far, congratulations! We now have a working API Client for our Test API.

Now the fun begins!

So our API Client is working properly and we've proven that using an Integration Test.

But what happens if the underlying HttpClient request fails?

What if there is a blip in the network?

What if the API server is overloaded and can't handle our request right now?

Now is the time to implement a retry policy with Polly!

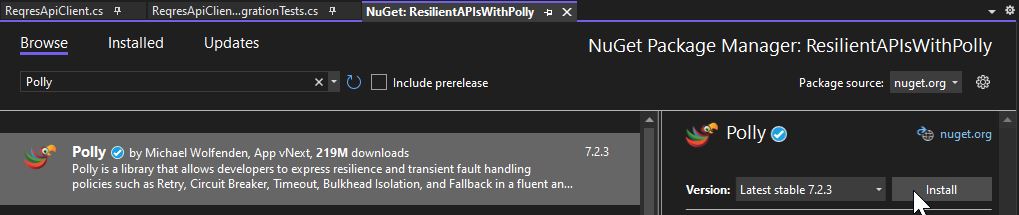
### **Adding The Polly Nuget package**

The first thing we need to do is add the [Polly Nuget Package](https://www.nuget.org/packages/polly/) to our library project.

In the Solution Explorer expand our ResilientAPIsWithPolly project and right-click on the Dependencies section. Then select Manage Nuget Packages.

The nuget package manager window will appear, select the Browse tab and search for Polly.

Select the latest version, I'm selecting Latest stable 7.2.3 and click Install.



### **Creating a Retry Policy with Polly**

For the initial addition of the retry policy, we won't worry about using Dependency Injection to make our API Client unit testable. Keep reading, we will add slight modifications to this code later on to allow for unit testing!

Our first retry policy will simply attempt to retry to send the HttpRequestMessage up to 3 times whenever a non-success code is returned in the response.

Here is our API Client with a basic retry policy, retrying up to 3 times when we don't receive a success message:

**using** Polly;

**using** Polly.Retry;

**namespace** **ResilientAPIsWithPolly**

{

**public** **class** **ReqresApiClient**

{

**private** **const** string \_reqresAPIBaseUrl = "https://reqres.in/api";

**private** **readonly** HttpClient \_httpClient;

**private** **const** int \_maxApiCallRetries = 3;

**private** **readonly** AsyncRetryPolicy<HttpResponseMessage> \_asyncRetryPolicy =

Policy.

HandleResult<HttpResponseMessage>(response => !response.IsSuccessStatusCode).

WaitAndRetryAsync(\_maxApiCallRetries, retryAttempt => TimeSpan.FromSeconds(retryAttempt));

**public** **ReqresApiClient**(HttpClient httpClient)

{

\_httpClient = httpClient;

}

**public** **async** Task<HttpResponseMessage> **GetUser**(int userId)

{

**var** userUrl = $"{\_reqresAPIBaseUrl}/users/{userId}";

**var** httpResponseMessage = **await** \_asyncRetryPolicy.ExecuteAsync(() => \_httpClient.GetAsync(userUrl));

**return** httpResponseMessage;

}

}

}

There is a little bit to unpack here.

First we created an AsyncRetryPolicy that will retry any async method that we want to execute up to 3 times.

The TimeSpan.FromSeconds(retryAttempt) portion of the code will increment the number of seconds to wait before retrying again.

For example, after the first failed attempt we will wait 1 second to retry. After the 2nd attempt we will wait to 2 seconds to retry and so on.

Then, we wrapped our original \_httpClient call with our policy.

We can run our integration test GetUserShouldReturnASuccessStatusCode again to make sure our happy path of successfully making the API call still works as expected.

This is a fairly straightforward example, but we can also add in some additional logic that runs before each retry. This gives us a great opportunity to write logs that contain information about the retry action.

### **Adding Logging on Retry to Our Retry Policy**

At this point we have set up our API Client to retry up to 3 times on a failure.

But we also want to be able to log a failure to have some type of history of each failure.

We will update our AsyncRetryPolicy to include a portion of code to write a log before each retry.

How you log your failures is beyond the scope of this guide, so the logging implementation has been left up to you in this example:

**using** Polly;

**using** Polly.Retry;

**namespace** **ResilientAPIsWithPolly**

{

**public** **class** **ReqresApiClient**

{

**private** **const** string \_reqresAPIBaseUrl = "https://reqres.in/api";

**private** **readonly** HttpClient \_httpClient;

**private** **const** int \_maxApiCallRetries = 3;

**private** **readonly** AsyncRetryPolicy<HttpResponseMessage> \_asyncRetryPolicy =

Policy.

HandleResult<HttpResponseMessage>(response => !response.IsSuccessStatusCode).

WaitAndRetryAsync(

\_maxApiCallRetries,

retryAttempt => TimeSpan.FromSeconds(retryAttempt),

(exception, timeSpan, retryCount, context) => {

// Logic that runs before each retry

// Write Logging Code Here

}

);

**public** **ReqresApiClient**(HttpClient httpClient)

{

\_httpClient = httpClient;

}

**public** **async** Task<HttpResponseMessage> **GetUser**(int userId)

{

**var** userUrl = $"{\_reqresAPIBaseUrl}/users/{userId}";

**var** httpResponseMessage = **await** \_asyncRetryPolicy.ExecuteAsync(() => \_httpClient.GetAsync(userUrl));

**return** httpResponseMessage;

}

}

}

### **Unit Testing Our Polly Retry Policy To Make Sure It's Working As Expected**

At this point we know we have a solid retry policy in place with logging.

Ok, so we have our policy in place.

But how do we know that the retry is actually going to work?

It's very difficult to have an external 3rd party give us a non-success result on command.

The answer is to write a unit test for GetUser. Inside of the unit test we will need to mock the HttpClient.

We won't go into depth on how mocking works, but in a brief answer - mocking allows us to control or simulate what values are returned by a specific method of a class.

This will allow us to simulate a failed response from the 3rd party test API via our HttpClient.

#### **Adding the Unit Test Class**

Similar to how we created a new ReqresApiClientIntegrationTests class, we now will create a Unit Test class.

Add a new test class by right clicking on our ResilientAPIsWithPolly.Tests project and clicking Add > Add New Item....

Select the item type Class and name it ReqresApiClientUnitTests.cs.

#### **Adding the Moq Nuget Package**

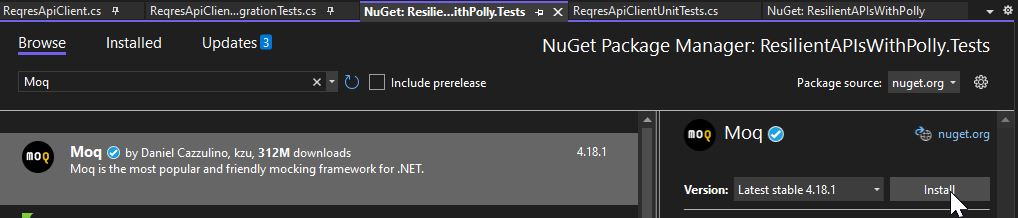
Before we write our Unit Test, we will need to add the Moq nuget package.

This nuget package will allow us to mock our HttpClient.

In the Solution Explorer expand our ResilientAPIsWithPolly.Tests project and right-click on the Dependencies section. Then select Manage Nuget Packages.

The nuget package manager window will appear, select the Browse tab and search for Polly.

Select the latest version, I'm selecting Latest stable 4.18.1 and click Install.



#### **Adding A Unit Test That Proves Our API Call Is Successful After An Initial Failure**

We will need to add functionality to confirm that our second API call is successful even after the initial call fails.

I won't go over how the mocking of the HTTP Client works here, but if you are interested in learning more please read [How to Test HttpClient with Moq in C#](https://dev.to/gautemeekolsen/how-to-test-httpclient-with-moq-in-c-2ldp).

Here is the unit test that confirms that even when our initial call fails, second call is still successful.

**using** Moq;

**using** Moq.Protected;

**using** System.Net;

**using** System.Net.Http;

**using** System.Threading;

**using** System.Threading.Tasks;

**using** Xunit;

**namespace** **ResilientAPIsWithPolly.Tests**

{

**public** **class** **ReqresApiClientUnitTests**

{

[Fact]

**public** **async** Task **GetUserShouldReturnASuccessStatusCodeAfterAnInitialUnsuccessfulRequest**()

{

// Arrange

**var** userId = 1;

Mock<HttpMessageHandler> handlerMock = BuildMockHttpHandlerFailOnFirstRequestThenSucceedOnSecond();

**var** httpClient = **new** HttpClient(handlerMock.Object);

**var** resilientApisWithPolly = **new** ReqresApiClient(httpClient);

// Act

**var** httpResponseMessage = **await** resilientApisWithPolly.GetUser(userId);

// Assert

Assert.True(httpResponseMessage.IsSuccessStatusCode);

}

**private** **static** Mock<HttpMessageHandler> **BuildMockHttpHandlerFailOnFirstRequestThenSucceedOnSecond**()

{

**var** handlerMock = **new** Mock<HttpMessageHandler>();

**var** initialFailureResponse = **new** HttpResponseMessage

{

StatusCode = HttpStatusCode.InternalServerError

};

**var** retrySecondRequestResponse = **new** HttpResponseMessage

{

StatusCode = HttpStatusCode.Accepted

};

handlerMock

.Protected()

.SetupSequence<Task<HttpResponseMessage>>(

"SendAsync",

ItExpr.IsAny<HttpRequestMessage>(),

ItExpr.IsAny<CancellationToken>())

.ReturnsAsync(initialFailureResponse)

.ReturnsAsync(retrySecondRequestResponse);

**return** handlerMock;

}

}

}

In a nutshell, this code allows us to control the underlying HttpClient using a sequence of test return values.

1. Return a failure on the first HTTP call
2. Return a success on the second HTTP call

This test proves that even though our original HTTP call fails, our API client is able to resend the request again and get a success on the second try.

This is all thanks to our underlying Polly Retry Policy that re-sends the request.

### **Resending an HttpRequestMessage Inside of the Polly Retry Policy**

So let's say, instead of simply calling \_httpClient.GetAsync, we want to be able to call \_httpClient.SendAsync and pass in our own HttpRequestMessage.

At first pass, we might write our code like this:

**using** Polly;

**using** Polly.Retry;

**namespace** **ResilientAPIsWithPolly**

{

**public** **class** **ReqresApiClient**

{

**private** **const** string \_reqresAPIBaseUrl = "https://reqres.in/api";

**private** **readonly** HttpClient \_httpClient;

**private** **const** int \_maxApiCallRetries = 3;

**private** **readonly** AsyncRetryPolicy<HttpResponseMessage> \_asyncRetryPolicy =

Policy.

HandleResult<HttpResponseMessage>(response => !response.IsSuccessStatusCode).

WaitAndRetryAsync(

\_maxApiCallRetries,

retryAttempt => TimeSpan.FromSeconds(retryAttempt),

(exception, timeSpan, retryCount, context) => {

// Logic that runs before each retry

// Write Logging Code Here

}

);

**public** **ReqresApiClient**(HttpClient httpClient)

{

\_httpClient = httpClient;

}

**public** **async** Task<HttpResponseMessage> **GetUser**(int userId)

{

**var** userUrl = $"{\_reqresAPIBaseUrl}/users/{userId}";

**var** httpRequestMessage = **new** HttpRequestMessage(HttpMethod.Get, userUrl);

**var** httpResponseMessage = **await** \_asyncRetryPolicy.ExecuteAsync(() => \_httpClient.SendAsync(httpRequestMessage));

**return** httpResponseMessage;

}

}

}

Now when we try to run our Unit Test GetUserShouldReturnASuccessStatusCodeAfterAnInitialUnsuccessfulRequest() the test will throw an exception.

That exception is: System.InvalidOperationException : The request message was already sent. Cannot send the same request message multiple times.

Why does this throw an exception?

**When using HttpClient you can't resend the same HttpRequestMessage twice!**

In order to fix this exception, we will need to wrap the creation of the HttpRequestMessage inside of the block of code that our Polly Retry Policy calls on each retry:

**using** Polly;

**using** Polly.Retry;

**namespace** **ResilientAPIsWithPolly**

{

**public** **class** **ReqresApiClient**

{

**private** **const** string \_reqresAPIBaseUrl = "https://reqres.in/api";

**private** **readonly** HttpClient \_httpClient;

**private** **const** int \_maxApiCallRetries = 3;

**private** **readonly** AsyncRetryPolicy<HttpResponseMessage> \_asyncRetryPolicy =

Policy.

HandleResult<HttpResponseMessage>(response => !response.IsSuccessStatusCode).

WaitAndRetryAsync(

\_maxApiCallRetries,

retryAttempt => TimeSpan.FromSeconds(retryAttempt),

(exception, timeSpan, retryCount, context) => {

// Logic that runs before each retry

// Write Logging Code Here

}

);

**public** **ReqresApiClient**(HttpClient httpClient)

{

\_httpClient = httpClient;

}

**public** **async** Task<HttpResponseMessage> **GetUser**(int userId)

{

**var** userUrl = $"{\_reqresAPIBaseUrl}/users/{userId}";

**var** httpResponseMessage = **await** \_asyncRetryPolicy.ExecuteAsync(

() =>

{

**var** httpRequestMessage = **new** HttpRequestMessage(HttpMethod.Get, userUrl);

**return** \_httpClient.SendAsync(httpRequestMessage);

});

**return** httpResponseMessage;

}

}

}

Now when we try to run our Unit Test GetUserShouldReturnASuccessStatusCodeAfterAnInitialUnsuccessfulRequest() the test passes.

This is because we create another instance of the same HttpRequestMessage every time our Polly Retry Policy runs!

### **Adding a Polly Retry Policy to Our API Client Using Dependency Injection**

This section is optional. but adding dependency injection will give the caller of the API client more control over how we want to retry HTTP calls.

In order to make our API client more customizable, we want to be able to allow an outside caller to define the retry policy.

We will use dependency injection here by adding a new AsyncRetryPolicy<HttpResponseMessage> parameter to our constructor.

Here is our API Client with support for dependency injection:

**using** Polly;

**using** Polly.Retry;

**namespace** **ResilientAPIsWithPolly**

{

**public** **class** **ReqresApiClient**

{

**private** **const** string \_reqresAPIBaseUrl = "https://reqres.in/api";

**private** **readonly** HttpClient \_httpClient;

**private** **const** int \_maxApiCallRetries = 3;

**private** **readonly** AsyncRetryPolicy<HttpResponseMessage> \_asyncRetryPolicy;

**public** **ReqresApiClient**(HttpClient httpClient, AsyncRetryPolicy<HttpResponseMessage> asyncRetryPolicy)

{

\_httpClient = httpClient;

\_asyncRetryPolicy = asyncRetryPolicy;

}

**public** **async** Task<HttpResponseMessage> **GetUser**(int userId)

{

**var** userUrl = $"{\_reqresAPIBaseUrl}/users/{userId}";

**var** httpResponseMessage = **await** \_asyncRetryPolicy.ExecuteAsync(() => \_httpClient.GetAsync(userUrl));

**return** httpResponseMessage;

}

}

}

This will allow any outside class that initializes our API Client to decide what the retry policy should be!

https://www.webdevtutor.net/blog/build-resilient-http-clients-in-c-sharp-on-dotnet-with-polly