REFACTORING HTTPCLIENT WITH ASP.NET CORE 3.1

Refactor a Legacy HttpClient into Reusable Components.

JONATHAN DANYLKO

Refactoring HttpClient with ASP.NET Core 3.1

Refactor a Legacy HttpClient Into Reusable Components with ASP.NET Core 3.1

By Jonathan Danylko

This book is self-published

Table of Contents

Introduction

"How is this happening using NetBIOS?"

This was the question I asked myself back in 1994 when I first experiencing the Internet. I was completely blown away by how the network was connecting across vast distances.

As we all know now, it was using TCP/IP, but I didn't realize it until I dug a little deeper into how everything was connected (excuse the pun). My research was using books back then, mind you (read my lips...No Internet).

Flash forward to 2020.

You can't look in any direction without seeing a connected device using TCP/IP. It is the life-blood flowing through the network's "arteries."

The world as we know it is completely comprised of TCP/IP connections and the world is now truly "the network as the operating system."

We now access the world every day using TCP/IP.

With the vast amount of data on the Internet, users only see the tip of the iceberg: the website. We don't see what's happening under the covers; providing access to websites through APIs

To make the world your oyster as a developer, you need to know where to look and how to access the data.

Consuming web APIs is how everything connects the dots.

The goal of this book is to show you how to use ASP.NET Core to harness the power of the HttpClient class to make a REST-based client API calls.

What kind of book is this?

I've been writing for over 10 years on my blog (https://www.danylkoweb.com) and felt it was time to dig into my first venture as a book writer.

Depending on how this book is received, I already have a number of topics I want to cover with future books.

I'm a technologist and developer at heart and, while I'm not a publisher by any stretch, I am a blogger. I love to learn new techniques and turn around and teach new and improved techniques through my writing.

With that said, this book is meant to accomplish two things:

1. Create small, digestible, and easy-to-read chapters
2. Make you a better developer

I want this to be a different kind of book for developers. It should break the tradition of what a technical book *should* look like.

This ebook:

- Is not meant to be an over-400-page book. I feel I would lose a lot of people with so many gigabytes.
- Should expand on ebooks in a different way and see this small book as a series of blog posts.

As I'm starting to move forward with this endeavor, I wanted to create a brand for these ebooks.

I'm calling series, *Pocket Projects*.

> ***Pocket Projects are ebooks longer than a blog post series and smaller than a full-blown ebook. They will also include repositories.***

These books are meant to be small in nature, but provide big value.

In developer-speak, *each ebook will be an extendable, proven subsystem for your applications* with a code repository of proven, real-world techniques.

What you'll learn

Pocket Projects will give you the knowledge to take existing server-side client API code and refactor it into something you can reuse in a different application.

The book is broken into two parts.

In the first part, we'll talk about the history of how server-side API calls were made and how to use (and not use) an HttpClient.

If you are experienced, you may want to skip the history and improper usage and pick up where you are "Creating a Reusable HttpClient" on page 16.

In the second half, we refactor the code into HttpClient components where we reduce the amount of time to writing future APIs to as little as 10 minutes.

In the end, you'll be able to build easier, maintainable, and quicker server-side APIs.

How to use this book

One approach I've seen used in best-selling technical books in the past was making the book as more of a reference manual instead of a novel.

Most developers are looking to solve a problem. They want to do *X* and they want to solve it as fast as possible.

While it is hard to update the paper-bound technical books, it's far more easier to update a simple ebook or find it on StackOverflow.com (am I right?).

The goal of Pocket Projects is ***to present developers with a collection of subtopics on a particular topic or subsystem.***

It may not be the entire wealth of knowledge on one topic in one book, but it will be considered a place for reference and code examples to point developers in the right direction.

Pocket Projects will allow the developer to open the book to a subtopic and examine the concepts found in the book.

Requirements

The technology used in this book is specifically ASP.NET Core 3.1, but does not mean other versions (including .NET Framework) could not use these techniques in older versions.

The core concept is to use HttpClient for consuming web APIs. As long as the HttpClient is available in .NET, a majority of these techniques will work.

The following editors recommended are:

- Visual Studio 2017/2019
- Visual Studio Code
- JetBrains' Rider
- Notepad++ (kidding, not kidding)

With ASP.NET Core, you have the ability to run applications at the command-line so **the preferred IDE for you to use is**...

...the one you feel comfortable using.

While Visual Studio 2017/2019 and Code (which is free) are Microsoft products, ASP.NET is available for most editors.

Code Repository

The finished code is located on Github.

```
https://www.danylkoweb.com/go/refactoredhttpclient
```

Thank You (with feedback)

With all of that out of the way, I want to say thank you for trying out the book and allowing me to take a “leap of faith” in writing my first book.

This will be my first step towards writing a collection of Pocket Projects *if* this book takes off.

I also want to let my readers know they can reach out to me with comments, complaints or future additions at any time regarding this book.

Thank you and I hope you enjoy the book.

Jonathan “JD” Danylko (*pronounced Dan-el-ko*)
Founder of DanylkoWeb.com
Twitter: @jdanylko
LinkedIn: jonathandanylko
Github: jdanylko

Crash-Course of API History

With the Internet taking off in the mid 90's, most developers (including myself) didn't understand too much about how everything was connecting except we could pull web pages using something called an HTTP protocol with a port of 80.

While server-side API calls were available, they weren't always straight-forward and required some knowledge of how to retrieve data from another server.

With that said, we need to take a step back and look at how server-side client API calls were made "back in the old days."

Of course, ASP.NET didn't exist yet. Microsoft developers were using Classic ASP.

Classic ASP

By the turn of the century, developers started learning how to build web services.

In Classic ASP, the code looked something like this:

```
<%
  Response.Buffer = True
  Dim objXMLHTTP, xml

  ' Create an xmlhttp object:
  Set xml = Server.CreateObject("Microsoft.XMLHTTP")
  ' Or, for version 3.0 of XMLHTTP, use:
  ' Set xml = Server.CreateObject("MSXML2.ServerXMLHTTP")

  ' Opens the connection to the remote server.
  xml.Open "GET", "http://www.4Guysfromrolla.com/", False

  ' Actually Sends the request and returns the data:
  xml.Send

  'Display the HTML both as HTML and as text
  Response.Write "<h1>The HTML text</h1><xmp>"
  Response.Write xml.responseText
  Response.Write "</xmp><p><hr><p><h1>The HTML Output</h1>"

  Response.Write xml.responseText
```

```
  Set xml = Nothing
%>
```

(ref: http://www.4guysfromrolla.com/webtech/110100-1.shtml)

We would make a web call to a web page using an ActiveX component and display it in the web browser.

Ahh..the good old days.

WebRequest

Then ASP.NET came along with true object-oriented programming concepts with C# and VB.NET.

In addition to OOP, additional web concepts became available to us like WebRequest. While this did modernize API calls,

```
using System;
using System.IO;
using System.Net;
using System.Text;

namespace Examples.System.Net
{
    public class WebRequestGetExample
    {
        public static void Main ()
        {
            // Create a request for the URL.
            WebRequest request = WebRequest.Create ("http://www.contoso.com/
default.html");
            // If required by the server, set the credentials.
            request.Credentials = CredentialCache.DefaultCredentials;
            // Get the response.
            HttpWebResponse response = (HttpWebResponse)request.GetResponse
();
            // Display the status.
            Console.WriteLine (response.StatusDescription);
            // Get the stream containing content returned by the server.
            Stream dataStream = response.GetResponseStream ();
            // Open the stream using a StreamReader for easy access.
            StreamReader reader = new StreamReader (dataStream);
            // Read the content.
            string responseFromServer = reader.ReadToEnd ();
            // Display the content.
            Console.WriteLine (responseFromServer);
            // Cleanup the streams and the response.
            reader.Close ();
```

```
            dataStream.Close ();
            response.Close ();
        }
    }
}
```

(ref: Microsoft WebRequest - .NET 2.0)

If you notice the WebRequest.Create, we aren't creating a new instance of an object. It was merely using a method to create the instance so we could use it later.

This is important to remember for later.

If you were a .NET developer, WebRequest was the class to use for API calls and the only time you saw clean code was on the site holding the documentation. Code back in the 2000s had a "wild west" feel to it.

Separation of concerns wasn't even thought of yet and wasn't even considered as a good practice.

Be honest. Back then, we all had our big ball of mud to play with.

In ASP.NET, making an API call was definitely easier than Classic ASP, but still lacked the flexibility of setting headers, authentication, and retrieving specific types of data. It only knew how to deliver HTML.

HttpClient

With the popularity of .NET rising, WebRequest was getting a little long in the tooth and required flexibility and a more modular design.

HttpClient was introduced in the .NET Framework 4.5 and was built to replace WebRequest in it's design.

With the HttpClient, developers were able to call web APIs relatively easy with the HttpWebRequest and HttpWebResponse classes.

The modular aspect of HttpClient provides simple calls with the capability of bolting on additional functionality like authentication and header manipulation.

While WebRequest is still available in the .NET Framework and .NET Core, Microsoft recommends using HttpClient instead of WebRequest.

HttpClientFactory

When ASP.NET Core was released, Microsoft decided to create a better way of making client-side API calls.

This fix was called HttpClientFactory.

There's a part of me that still thinks the HttpClientFactory was created based on bad usage of the HttpClient class, but after further investigation, I was right.

Most developers were not using the HttpClient properly.

We'll touch on these issues in the next section.

Using HttpClient (Badly)

Since introduced in 2012 with .NET Framework, HttpClient has been used in various ways to make API calls to other websites.

Most of the implementations were incorrect.

For example, you've probably seen how to build a simple API with the code below.

As a developer, you recognize it as a class, so it must need to be instantiated.

```
  public async Task<IActionResult> Index()
  {
      var client = new HttpClient();

      client.BaseAddress = new Uri("http://mywebapi.com/");
      client.DefaultRequestHeaders.Accept.Clear();
      client.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));

      Product product;
      HttpResponseMessage response = await client.GetAsync("api/products/1");
      if (response.IsSuccessStatusCode)
      {
          product = await response.Content.ReadAsAsync<Product>();
      }

      return View(product);
  }
```

This has been established as a bad practice.

A Better Approach?

From this point on, most developers decided to wrap the code with a using statement so the HttpClient would automatically dispose of itself.

The code would look like this:

```
  public async Task<IActionResult> Index()
  {
      using (var client = new HttpClient())
      {
          client.BaseAddress = new Uri("http://mywebapi.com/");
          client.DefaultRequestHeaders.Accept.Clear();
          client.DefaultRequestHeaders.Accept.Add(new
```

```
MediaTypeWithQualityHeaderValue("application/json"));

            Product product;
            HttpResponseMessage response = await client.GetAsync("api/products/1");
            if (response.IsSuccessStatusCode)
            {
                product = await response.Content.ReadAsAsync<Product>();
            }

            return View(product);
        }
    }
```

Remember in the last section when I said how the WebRequest object was created with a static Create method instead of using the new to create a new object?

When this action is called in an MVC project, we are creating another instance of an HttpClient object.

This is also considered a bad practice.

With the HttpClient class, this approach causes TCP port exhaustion when continually active. One of the most talked-about links is the ASP.NET Monsters example where this occurred (see Appendix A: Resources)

When using HttpClient, it's documented where ***only a single instance of an HttpClient object should be instantiated throughout the entire application***.

If you create multiple instances of an HttpClient in multiple locations, you are hindering your performance and scaling opportunities for your application.

When creating your HttpClient, it should follow a singleton pattern approach. If the object doesn't exist, create one and return it. If it was created and does exist, just return the instance.

Review

We reviewed some bad approaches developers use when creating HttpClients.

So what is the correct approach and how could we fix this issue to create a single instance of our HttpClient?

Creating a Reusable HttpClient

Applications usually have more than one API call making the code easy to locate and manage. However, if there are multiple API calls in your application, more than likely, they are decentralized throughout the application.

This section will take your decentralized API code and turn it into a maintainable and extendable REST-based code.

So let's get started with our existing code from above. How would we refactor this code?

Abstracting the Singleton Pattern

What is needed is a simple way to protect ourselves from too many HttpClient calls. Once we build the class, we can easily extend it for other opportunities.

To make our HttpClient easier to work with, we need a class to wrap around the HttpClient object.

```
public abstract class BaseHttpClient
{
   private static HttpClient _client = new HttpClient();
}
```

This simple class provides a lot of punch and protects us from a number of issues. So let's break it down a bit.

First, why an abstract class?

We want to create a single class to use only one HttpClient object. Keep in mind, we will be making multiple calls to a LOT of APIs (I'll explain later why this is important) and reusing some of the specific features of an HttpClient.

An abstract class to wrap the implementation details of an HttpClient makes sense. We can control when we use the HttpClient.

Speaking of the HttpClient, the next line creates a *static* instance of the HttpClient. This allows our HttpClient to always be available in memory.

Let's add some API characteristics to our class.

```
public abstract class BaseHttpClient
```

```
{
    private static HttpClient _client = new HttpClient();

    public Uri BaseAddress { get; set; }
    public string BasePath { get; set; }

    private HttpClient GetHttpClient()
    {
        return _client;
    }
}
```

BaseAddress and BasePath are absolutely required. BaseAddress would be the domain (www.myapi.com) and the BasePath property would contain full path of the API after the domain (/api/products/1).

We also create a private method called GetHttpClient() for our own purposes to confirm we don't create unnecessary HttpClients on the fly.

Using HttpClientFactory

With the release of ASP.NET Core, an additional class was introduced to help with managing HttpClient instances called the HttpClientFactory.

HttpClientFactory is basically a singleton by nature. When you ask for an HttpClient, it returns one for you. It may be a new instance or it could be the same one you used previously. All you need to know is you received one when you requested one.

We can easily replace the HttpClient in our code with an HttpClientFactory class making this singleton even simpler (a simpleton?).

Our code will now look like this.

```
    public abstract class BaseHttpClientWithFactory
    {
        private readonly IHttpClientFactory _factory;

        public Uri BaseAddress { get; set; }
        public string BasePath { get; set; }

        public BaseHttpClientWithFactory(IHttpClientFactory factory)
            => _factory = factory;

        private HttpClient GetHttpClient()
        {
            return _factory.CreateClient();
        }
    }
```

If you notice, we are passing in an IHttpClientFactory to our constructor. We are using dependency injection for this.

Out of the box, ASP.NET Core uses dependency injection and should always be implemented in the Startup.cs in the root of your project.

Lucky for us, all we need to do is add one line to make our code work.

Add a service.AddHttpClient() to your ConfigureServices method.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    services.AddHttpClient();
}
```

With this one line, it does two things for you. It adds the implementation of an HttpClient to your library and automatically sets up an IHttpClientFactory/HttpClientFactory pair through constructor injection.

In our BaseHttpClientWithFactory class, the ASP.NET Core will detect the IHttpClientFactory interface as a parameter in the constructor and automatically create a new instance of the HttpClientFactory and pass that instance through to our constructor.

All that with one line.

Review

We've taken an HttpClient call and isolated it into a reusable API class. Even though it is a simple class, it gives us a solid foundation moving forward.

Now we need to focus on how to build the request message.

A Better HttpRequestMessage

There are a number of methods you can use with an HttpClient. You can call the following on HttpClients:

- DeleteAsync()
- GetAsync()
- PostAsync()
- PutAsync()
- PatchAsync()
- SendAsync()

While all of these are most commonly-used API verbs, the one that should interest you is the last one, SendAsync().

SendAsync() is a general purpose method to accept an HttpRequestMessage as a parameter. It can be any one of these HTTP verbs from above by defining it through an HttpRequestMessage.

We need a way to define our HttpRequestMessage.

Creating HttpRequestBuilder

If you've ever deifned an HttpRequestMessage, you know it's a little tedious and requires the left-to-right assignment for all properties required for our message.

Why not make it like a programmatic sentence? Why not use a Builder design pattern to create our API message?

A builder design pattern takes a complex object and creates construction methods making it easier to define and build.

The code below is our first pass.

```
public class HttpRequestBuilder
{
    private readonly HttpRequestMessage _request;
    private Uri _uri;
```

```
    private string _baseAddress;

    public HttpRequestBuilder(string url)
    {
        _uri = new Uri(url);
        _baseAddress _uri.GetLeftPart();
    }

    public HttpRequestBuilder()
    {
        _request = new HttpRequestMessage();
    }

    public HttpRequestBuilder HttpMethod(HttpMethod httpMethod)
    {
        _request.Method = httpMethod;
        return this;
    }

    public HttpRequestBuilder Headers(
        Action<HttpRequestHeaders> funcOfHeaders)
    {
        funcOfHeaders(_request.Headers);
        return this;
    }

    public HttpRequestBuilder Headers(NameValueCollection headers)
    {
        _request.Headers.Clear();
        foreach (var item in headers.AllKeys)
        {
            _request.Headers.Add(item, headers[item]);
        }

        return this;
    }

    public HttpRequestBuilder Content(HttpContent content)
    {
        _request.Content = content;
        return this;
    }

    public HttpRequestBuilder BaseAddress(string address)
    {
        _baseAddress = address;
        return this;
    }

    public HttpRequestMessage GetHttpMessage()
    {
        return _request;
    }
}
```

When we instantiate our builder, we immediately create an internal HttpRequestMessage.

From this point forward, we expose certain methods to internally build the HttpRequestMessage, create methods to hide the implementation details of our internal object (_request), and return the instance of the builder which gives us our fluent syntax.

In our code example from before, our refactored request now looks like this:

```
var message = new HttpRequestBuilder("http://www.myapi.com/api/products/1")
    .HttpMethod(HttpMethod.Get)
    .Headers(headers =>
        headers.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json")))
    .GetHttpMessage();
```

This returns our constructed message we can pass into our API.

Review

We took the tedious process of performing a left-to-right assignment of an HttpRequestMessage and refactored it into a builder design pattern.

This gives us more flexibility to attach additional functionality as we need it moving forward.

Next, we need a way to manipulate the URL passed into the HttpRequestBuilder.

Url Management

In our last section, we passed in a URL to our HttpRequestBuilder, but there are a couple of issues with this approach.

- What if we have one base address with multiple paths available?
- How can we add more fragments onto the URL?

While the Uri class is helpful, it needs a little more. We can actually apply the same builder pattern to a Uri builder.

However, if you've done your homework, you know there is a UriBuilder already available.

Why not wrap the UriBuilder class with an ApiBuilder class so we can manage the Uri at an API level?

```
public class ApiBuilder
{
    private readonly string _fullUrl;
    private UriBuilder _builder;

    public ApiBuilder(string url)
    {
        _fullUrl = url;
        _builder = new UriBuilder(url);
    }

    public Uri GetUri() => _builder.Uri;

    public ApiBuilder Scheme(string scheme)
    {
        _builder.Scheme = scheme;
        return this;
    }

    public ApiBuilder Host(string host)
    {
        _builder.Host = host;
        return this;
    }

    public ApiBuilder Port(int port)
    {
```

```
    _builder.Port = port;
    return this;
}

public ApiBuilder AddToPath(string path)
{
    IncludePath(path);
    return this;
}

public ApiBuilder SetPath(string path)
{
    _builder.Path = path;
    return this;
}

public void IncludePath(string path)
{
    if (string.IsNullOrEmpty(_builder.Path)
        || _builder.Path == "/")
    {
        _builder.Path = path;
    }
    else
    {
        var newPath = $"{_builder.Path}/{path}";
        _builder.Path = newPath.Replace("//", "/");
    }
}

public ApiBuilder Fragment(string fragment)
{
    _builder.Fragment = fragment;
    return this;
}

public ApiBuilder SetSubdomain(string subdomain)
{
    _builder.Host = string.Concat(subdomain, ".", new Uri(_fullUrl).Host);
    return this;
}

public bool HasSubdomain()
{
    return _builder.Uri.HostNameType == UriHostNameType.Dns
           && _builder.Uri.Host.Split('.').Length > 2;
}

public ApiBuilder AddQueryString(string name, string value)
{
    var qsNv = HttpUtility.ParseQueryString(_builder.Query);
    qsNv[name] = string.IsNullOrEmpty(qsNv[name])
        ? value
        : string.Concat(qsNv[name], ",", value);

    _builder.Query = qsNv.ToString();
```

```
        return this;
    }

    public ApiBuilder QueryString(string queryString)
    {
        if (!string.IsNullOrEmpty(queryString))
        {
            _builder.Query = queryString;
        }
        return this;
    }

    public ApiBuilder UserName(string username)
    {
        _builder.UserName = username;
        return this;
    }

    public ApiBuilder Password(string password)
    {
        _builder.Password = password;
        return this;
    }

    public string GetLeftPart()
    {
        return _builder.Uri.GetLeftPart(UriPartial.Path);
    }
}
```

With this new ApiBuilder class, we can now modify our HttpRequestBuilder to make our Url management a little easier.

Here is our new HttpRequestBuilder code (changes in **bold**).

```
public class HttpRequestBuilder
{
    private readonly HttpRequestMessage _request;
    private string _baseAddress;
    private readonly ApiBuilder _apiBuilder;

    public HttpRequestBuilder(string uri): this(new ApiBuilder(uri)) { }
    public HttpRequestBuilder(ApiBuilder apiBuilder)
    {
        _request = new HttpRequestMessage();
        _apiBuilder = apiBuilder;
        _baseAddress = _apiBuilder.GetLeftPart();
    }

    public HttpRequestBuilder HttpMethod(HttpMethod httpMethod)
    {
        _request.Method = httpMethod;
        return this;
    }
```

```
public HttpRequestBuilder Headers(Action<HttpRequestHeaders> funcOfHeaders)
{
    funcOfHeaders(_request.Headers);
    return this;
}

public HttpRequestBuilder Headers(NameValueCollection headers)
{
    _request.Headers.Clear();
    foreach (var item in headers.AllKeys)
    {
        _request.Headers.Add(item, headers[item]);
    }

    return this;
}

public HttpRequestBuilder AddToPath(string path)
{
    _apiBuilder.AddToPath(path);
    _request.RequestUri = _apiBuilder.GetUri();

    return this;
}

public HttpRequestBuilder SetPath(string path)
{
    _apiBuilder.SetPath(path);
    _request.RequestUri = _apiBuilder.GetUri();

    return this;
}

public HttpRequestBuilder Content(HttpContent content)
{
    _request.Content = content;
    return this;
}

public HttpRequestBuilder RequestUri(Uri uri)
{
    _request.RequestUri = new ApiBuilder(uri.ToString()).GetUri();
    return this;
}

public HttpRequestBuilder RequestUri(string uri)
{
    return RequestUri(new Uri(uri));
}

public HttpRequestBuilder BaseAddress(string address)
{
    _baseAddress = address;
    return this;
}

public HttpRequestBuilder Subdomain(string subdomain)
```

```
    {
        _apiBuilder.SetSubdomain(subdomain);
        _request.RequestUri = _apiBuilder.GetUri();

        return this;
    }

    public HttpRequestBuilder AddQueryString(string name, string value)
    {
        _apiBuilder.AddQueryString(name, value);
        _request.RequestUri = _apiBuilder.GetUri();

        return this;
    }

    public HttpRequestBuilder SetQueryString(string qs)
    {
        _apiBuilder.QueryString(qs);
        _request.RequestUri = _apiBuilder.GetUri();

        return this;
    }

    public HttpRequestMessage GetHttpMessage()
    {
        return _request;
    }

    public ApiBuilder GetApiBuilder()
    {
        return new ApiBuilder(_request.RequestUri.ToString());
    }
}
```

Now we can dynamically add parameters to our APIs without worrying about breaking our Url.

Our HttpRequestBuilder is now able to create something a little easier to read.

```
var message = new HttpRequestBuilder("https://www.myapi.com/")
    .SetPath("/api/products/1")
    .HttpMethod(HttpMethod.Get)
    .Headers(headers =>
        headers.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json")))
    .GetHttpMessage();
```

This tells us we are making an API call to the myapi.com website, setting the path to /api/products/1, making it an HTTP GET, and returning back Json.

Review

We created an ApiBuilder to simplify the management of our Urls when building our HttpRequestMessage.

Now that we have our HttpRequestMessage, we need to send the actual request.

Sending the Request

With everything in place, we need to revisit our HttpRequestBuilder.

If you remember, the simplest way to send this message is to use the SendAsync() and let the message define the functionality of the HttpClient request.

However, every API call doesn't return the same class type every single time. We need a way to tell the API we are expecting a specific class type so we can work with the results.

This is where generics come into play. Generics allow various classes to be used from an abstract level.

The caller specifies the type to be returned from the API. The API makes the call and creates an instance of that type and returns it from the method. The ability to use generics in your code makes it easier to reuse components.

It's much easier to show what I mean in our BaseHttpClientFactory class than explain (changes in **bold**).

```
public abstract class BaseHttpClientWithFactory
{
    private readonly IHttpClientFactory _factory;

    public Uri BaseAddress { get; set; }
    public string BasePath { get; set; }

    public BaseHttpClientWithFactory(IHttpClientFactory factory)
        => _factory = factory;

    private HttpClient GetHttpClient()
    {
        return _factory.CreateClient();
    }

    public virtual async Task<T> SendRequest<T>(HttpRequestMessage request)
        where T : class
    {
        var client = GetHttpClient();

        var response = await client.SendAsync(request);

        T result = null;
```

```
            response.EnsureSuccessStatusCode();

            if (response.IsSuccessStatusCode)
            {
                result = await response.Content.ReadAsAsync<T>(GetFormatters());
            }

            return result;
        }

        protected virtual IEnumerable<MediaTypeFormatter> GetFormatters()
        {
            // Make JSON the default
            return new List<MediaTypeFormatter> { new JsonMediaTypeFormatter() };
        }

        public abstract HttpRequestBuilder GetHttpRequestBuilder(string path);
    }
```

The SendRequest method signature is loaded with a lot of details so let's down the method.

With every SendRequest called, we may want to perform some additional processing on returning the data, so making it virtual allows you to override the method for further processing.

Since async gives us the performance we look for in quick websites, this call is requires an async operator.

Next in line is the generic (<T>). In our caller, we can now add a class type on our SendRequest and the API will return an instance of our populated object.

```
var products = await base.SendRequest<List<Product>>(message);
```

This will return a list of Products from our API request. Using generics, we can make this a centralized, single point-of-entry where an API returns any type from this method.

The GetFormatters() method is also made override-able just in case you want to use XML instead of Json as your return type.

The GetHttpRequestBuilder() is a helper method allowing you to get an HttpRequestBuilder quicker from a descendant class with a simple path passed as a parameter.

We grab our *single instance* of an HttpClient from our factory and make a call to our API. We confirm our call was successful with the EnsureSuccessStatusCode() method. If it wasn't, it will return the appropriate error to the browser or calling program.

If our API call was was successful, we read the content based on the content format and return the deserialized object.

Review

We added a SendRequest method to our HttpRequestBuilder allowing us to retrieve results from all of our hard work.

With our foundation coded, you may be wondering how everything ties together. In the next section, we look at putting all of the pieces together to make a simple client-side API call.

Along with an example, we'll be moving a little faster throughout these sections bolting on additional features to expand on our existing codebase.

Building a Client-Side API Call

With the infrastructure in place and isolated, we can now create our first example of an API call with our code.

Let's take an example of an existing API out there: The Postman Echo service.

This call is merely to see if our client can call the service. The service is located at: `https://postman-echo.com/`

Create the Model

The first step is to get the model. Based on the documentation, the Json representation is below:


```
{
  "args": {
    "foo": "bar"
  },
  "headers": {
    "x-forwarded-proto": "https",
    "host": "postman-echo.com",
    "accept": "*/*",
    "accept-encoding": "gzip, deflate",
    "cache-control": "no-cache",
    "postman-token": "5c27cd7d-6b16-4e5a-a0ef-191c9a3a275f",
    "user-agent": "PostmanRuntime/7.6.1",
    "x-forwarded-port": "443"
  },
  "url": "https://postman-echo.com/get?foo1=bar1&foo2=bar2"
}
```

SECRET SPEED TIP

If you have an entire Json response for an API, you can use Visual Studio 2017/2019 to build your classes fast.

1. Copy the entire Json result of your API. It can usually be found in the documentation.
2. Create a new class in Visual Studio 2017/2019
3. Under Edit → Paste Special → Paste Json as classes
4. Rename your RootObject to xxxResponse.
5. Use this Response object as your result from the API.

Our newly created model looks like this.

```
public class PostmanEchoResponse
{
    public Args args { get; set; }
    public Headers headers { get; set; }
    public string url { get; set; }
}

public class Args
{
    public string foo { get; set; }
}

public class Headers
{
    public string xforwardedproto { get; set; }
    public string host { get; set; }
    public string accept { get; set; }
    public string acceptencoding { get; set; }
    public string cachecontrol { get; set; }
    public string postmantoken { get; set; }
    public string useragent { get; set; }
    public string xforwardedport { get; set; }
}
```

Create the Base Api

Since we now have our model, we can start building our base Postman Echo class.

This class will encapsulate all of the API calls to the postman-echo service specifically. If you have other APIs in your system, you would create another BasexxxxxApi for your calls.

Essentially, for every domain where you make an API call, you will

```
public class BasePostmanEchoApi : BaseHttpClientWithFactory
{
    private const string baseAddress = "https://postman-echo.com/";

    public BasePostmanEchoApi(IHttpClientFactory factory) : base(factory) { }

    public override HttpRequestBuilder GetHttpRequestBuilder(string path)
    {
        return new HttpRequestBuilder(baseAddress)
            .SetPath(path)
            .HttpMethod(HttpMethod.Get)
            .Headers(headers =>
                headers.Accept.Add(new MediaTypeWithQualityHeaderValue("application/
json")));
    }
}
```

Very small and very simple. This class should contain the basics for creating a simple API request to the postman-echo service and it returns an HttpRequestBuilder.

Look at it this way. When creating your API call with the HttpRequestBuilder, determine the most common settings you will make for each API call? This will include the base address, authorization, and header settings to name a few.

Create the Service

Next, we create the actual class for our call.

The Postman Echo Service gives us our top level access into our Postman API calls.

```
public class PostmanEchoApi : BasePostmanEchoApi, IPostmanEchoApi
{
    public PostmanEchoApi(IHttpClientFactory factory) : base(factory)
    {
        BasePath = "/get";
    }

    public async Task<PostmanEchoResponse> GetEchoAsync(string foo)
    {
        var message = GetHttpRequestBuilder(BasePath)
            .AddQueryString("foo", foo)
            .GetHttpMessage();

        return await base.SendRequest<PostmanEchoResponse>(message);
    }
}
```

Notice how we only have one method called GetEcho with a single parameter? This allows us to create multiple methods for the GET service. We'll cover this later with another API example.

Usage of our Example API

To use our PostmanEchoApi, we dependency-inject it into our controller and, just like a repository, we make the call to retrieve the data from the API.

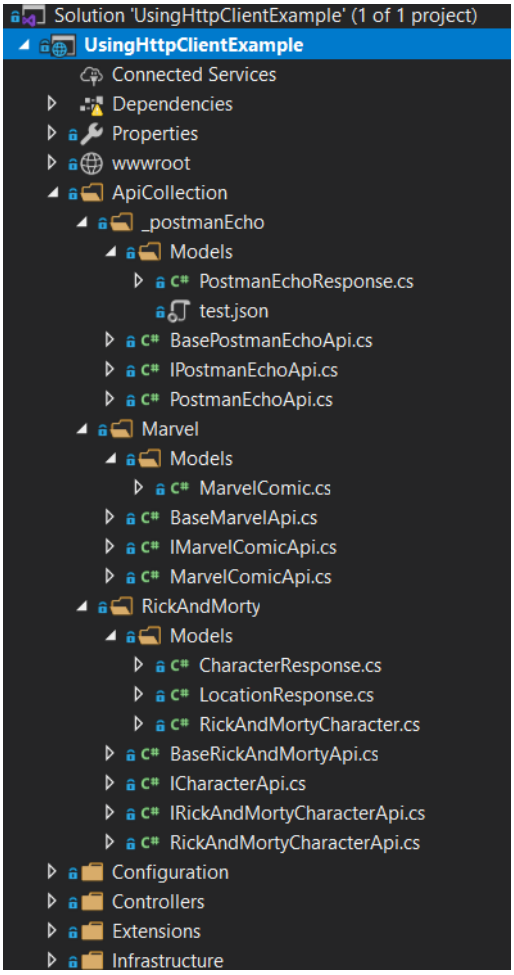
```
public class HomeController : Controller
{
    private readonly IPostmanEchoApi _postmanApi;

    public HomeController(IPostmanEchoApi postmanApi)
    {
        _postmanApi = postmanApi;
    }

    public async Task<IActionResult> Index()
    {
        var echo = await _postmanApi.GetEchoAsync("bar");

        return View(echo);
    }
}
```

Again, notice the IPostmanEchoApi as the only parameter? Don't forget to set up your dependency injection in your Startup.cs.

```
services.AddTransient<IPostmanEchoApi, PostmanEchoApi>();
```

You may be wondering what the folder structure looks like with this type of project. While the infrastructure piece is small and self-contained, it's suitable to place all of your API code in a separate folder.

Below is a possible folder structure for your projects.



The API folder (ApiCollection in this example) contains all of our APIs, Models, and implementations of our API Service calls.

Again, for every domain, you would create a new folder containing all of your response models and API code. This gives a nice centralized location for your API code.

Review

In our first example, we created a quick API to call the Postman Echo API to test our infrastructure.

As we look over the implementation, you can see it's very lightweight, can be extended further, and creates a simple, stable, and organized way to build your APIs.

Example: Rick and Morty API

With our new HttpClient able to handle a majority of APIs out there, there was one API I wanted to try out.

Since Rick and Morty are so popular, I was able to find a Rick and Morty API. This API doesn't require a token or key and is public.

We'll focus on the characters API for now.

Create the Model

Our model is based on the Json output located here: https://rickandmortyapi.com/documentation#character


```
{
  "info": {
    "count": 394,
    "pages": 20,
    "next": "https://rickandmortyapi.com/api/character/?page=20",
    "prev": "https://rickandmortyapi.com/api/character/?page=18"
  },
  "results": [
    {
      "id": 361,
      "name": "Toxic Rick",
      "status": "Dead",
      "species": "Humanoid",
      "type": "Rick's Toxic Side",
      "gender": "Male",
      "origin": {
        "name": "Alien Spa",
        "url": "https://rickandmortyapi.com/api/location/64"
      },
      "location": {
        "name": "Earth",
        "url": "https://rickandmortyapi.com/api/location/20"
      },
      "image": "https://rickandmortyapi.com/api/character/avatar/361.jpeg",
      "episode": [
        "https://rickandmortyapi.com/api/episode/27"
      ],
      "url": "https://rickandmortyapi.com/api/character/361",
      "created": "2018-01-10T18:20:41.703Z"
    }
```

```
  ]
}
```

Our generated model looks like this.

```
public class CharacterResponse
{
    public Info info { get; set; }
    public CharacterModel[] results { get; set; }
}

public class CharacterModel
{
    public int id { get; set; }
    public string name { get; set; }
    public string status { get; set; }
    public string species { get; set; }
    public string type { get; set; }
    public string gender { get; set; }
    public Origin origin { get; set; }
    public Location location { get; set; }
    public string image { get; set; }
    public string[] episode { get; set; }
    public string url { get; set; }
    public DateTime created { get; set; }
}

public class Origin
{
    public string name { get; set; }
    public string url { get; set; }
}

public class Location
{
    public string name { get; set; }
    public string url { get; set; }
}
```

Create the Base API

After analyzing the documentation, the base address adds an ‘api’ to the end.

```
public class BaseRickAndMortyApi : BaseHttpClientWithFactory
{
    private const string baseAddress = "https://rickandmortyapi.com/";

    public BaseRickAndMortyApi(IHttpClientFactory factory) : base(factory) { }

    public override HttpRequestBuilder GetHttpRequestBuilder(string path)
    {
        return new HttpRequestBuilder(baseAddress)
            .SetPath("api")
```

```
            .AddToPath(path)
            .HttpMethod(HttpMethod.Get)
            .Headers(headers =>
                headers.Accept.Add(
                    new MediaTypeWithQualityHeaderValue("application/json")));
    }
}
```

Create the Service

Finally, we complete the API with a service.

```
public class RickAndMortyCharacterApi : BaseRickAndMortyApi, IRickAndMortyCharacterApi
{
    public RickAndMortyCharacterApi(IHttpClientFactory factory) : base(factory)
    {
        BasePath = "/character";
    }

    public async Task<RickAndMortyCharacter> GetCharacters()
    {
        var message = GetHttpRequestBuilder(BasePath)
            .GetHttpMessage();

        return await base.SendRequest<RickAndMortyCharacter>(message);
    }

    public async Task<RickAndMortyCharacter> GetRickAndMorty()
    {
        var message = GetHttpRequestBuilder(BasePath)
            .AddToPath("1,2")
            .GetHttpMessage();

        return await base.SendRequest<RickAndMortyCharacter>(message);
    }
}
```

If we look at the documentation, they also mention how to get multiple characters using comma-delimited id numbers.

For example, to grab Rick and Morty data, they are 1 and 2 respectively. You add the id list to the end of the URL (see GetRickAndMorty)

Anything dealing with the /character API is an quick and easy method to create.

Review

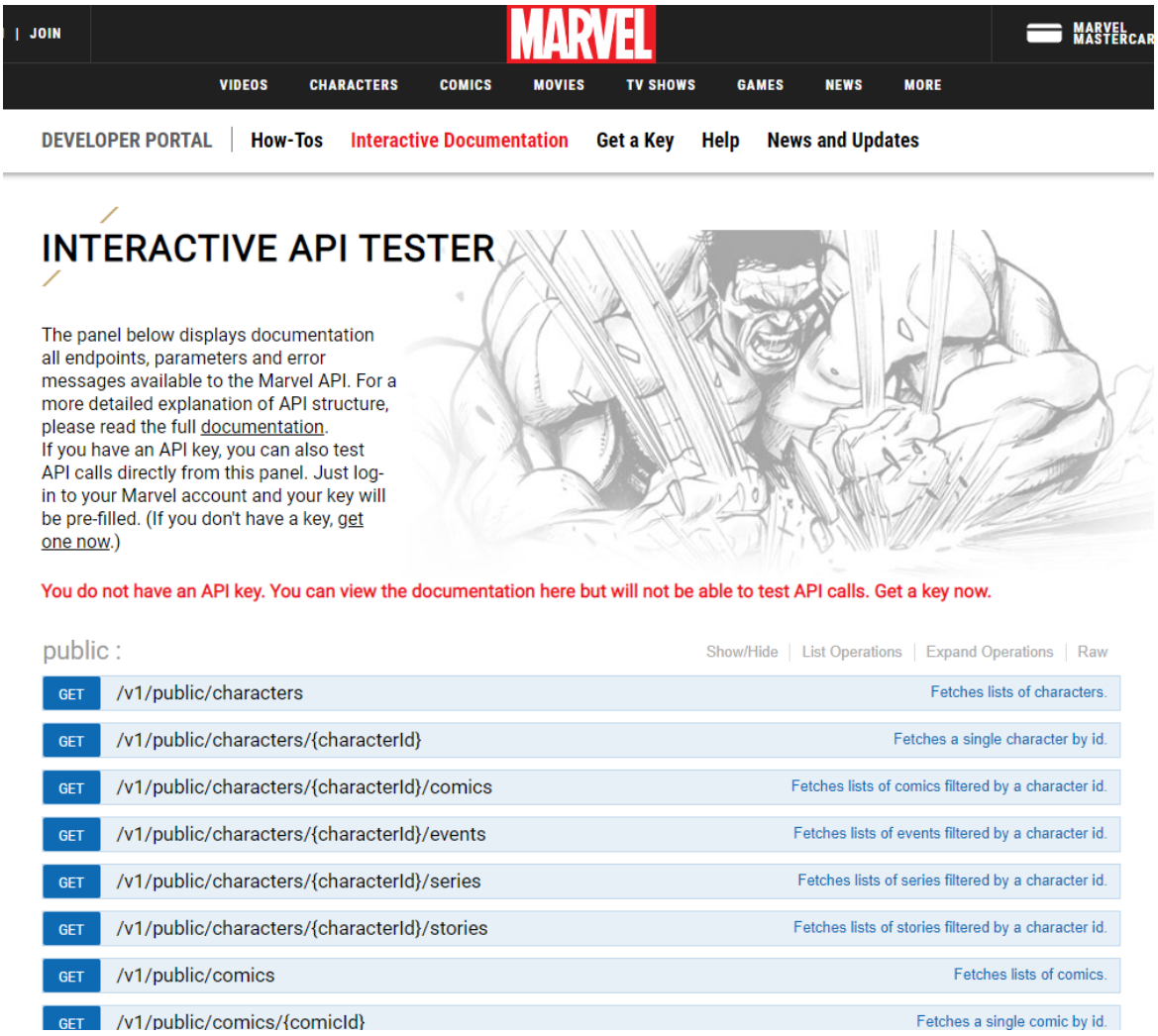
We took a simple REST-based API and made it easy to create additional methods.

We'll focus on one more example with a web token.

Example: Marvel API

Who doesn't love a good Marvel story, but did you know there is an API for everything Marvel?

The Marvel API is located at https://developer.marvel.com/docs and they have a lot of APIs available with varying calls.



While the Marvel API is REST-based, you can only make API calls by registering and receiving a private key.

Create the Model

Our Marvel model is a bit beefy compared to the previous ones.

```
public class MarvelComic
{
    public int code { get; set; }
    public string status { get; set; }
    public string copyright { get; set; }
    public string attributionText { get; set; }
    public string attributionHTML { get; set; }
    public string etag { get; set; }
    public Data data { get; set; }
}

public class Data
{
    public int offset { get; set; }
    public int limit { get; set; }
    public int total { get; set; }
    public int count { get; set; }
    public Result[] results { get; set; }
}

public class Result
{
    public int id { get; set; }
    public string name { get; set; }
    public string description { get; set; }

    public string modified { get; set; }
    public DateTime modifiedDate { get; set; }
    public Thumbnail thumbnail { get; set; }
    public string resourceURI { get; set; }
    public Comics comics { get; set; }
    public Series series { get; set; }
    public Stories stories { get; set; }
    public Events events { get; set; }
    public Url[] urls { get; set; }
}

public class Thumbnail
{
    public string path { get; set; }
    public string extension { get; set; }
}

public class Comics
{
    public int available { get; set; }
    public string collectionURI { get; set; }
    public Item[] items { get; set; }
    public int returned { get; set; }
}

public class Item
```

```
{
    public string resourceURI { get; set; }
    public string name { get; set; }
}

public class Series
{
    public int available { get; set; }
    public string collectionURI { get; set; }
    public Item1[] items { get; set; }
    public int returned { get; set; }
}

public class Item1
{
    public string resourceURI { get; set; }
    public string name { get; set; }
}

public class Stories
{
    public int available { get; set; }
    public string collectionURI { get; set; }
    public Item2[] items { get; set; }
    public int returned { get; set; }
}

public class Item2
{
    public string resourceURI { get; set; }
    public string name { get; set; }
    public string type { get; set; }
}

public class Events
{
    public int available { get; set; }
    public string collectionURI { get; set; }
    public Item3[] items { get; set; }
    public int returned { get; set; }
}

public class Item3
{
    public string resourceURI { get; set; }
    public string name { get; set; }
}

public class Url
{
    public string type { get; set; }
    public string url { get; set; }
}
```

Now that we have our model, we need to create our base API.

Create the Base API

Our BaseMarvelApi works a little bit different. Once you register, you get your private key.

Personally, I really like the way they provide access to their API through hashing.

When you register, they present you with a public and private key. When you make an API call, they require three pieces of data:

- Timestamp (ts)
- The Public Key
- A Hashed Key - Comprised of the Date/Time from above along with the publicKey and privateKey combined.

Since this is a repetitive task every time a Marvel API is called, we will place this into the BaseMarvelApi class.

```
public class BaseMarvelApi : BaseHttpClientWithFactory
{
    private const string baseAddress = "https://gateway.marvel.com/";
    private const string version = "v1";
    private const string publicKey = "public key goes here";
    private const string privateKey = "private key goes here";

    public BaseMarvelApi(IHttpClientFactory factory): base(factory) { }

    public override HttpRequestBuilder GetHttpRequestBuilder(string path)
    {
        var query = GetQueryString();

        return new HttpRequestBuilder(baseAddress)
            .SetPath(version)
            .AddToPath("public")
            .AddToPath(path)
            .SetQueryString(query.ToQueryString())
            .HttpMethod(HttpMethod.Get)
            .Headers(headers =>
                headers.Accept.Add(new MediaTypeWithQualityHeaderValue("application/
json")));
    }

    public NameValueCollection GetQueryString()
    {
        var current = DateTime.UtcNow.ToString("s");
        return new NameValueCollection
            {
                { "ts", current },
                { "apikey", publicKey }, // nope, can't show the public key.
                { "hash", GetApiKey(current) }
            };
    }
```

```
    public string GetApiKey(string ts)
    {
        using (MD5 md5 = MD5.Create())
        {
            return GetHash(md5, ts + privateKey + publicKey);
        }
    }

    private string GetHash(HashAlgorithm hashAlgorithm, string input)
    {
        var data = hashAlgorithm.ComputeHash(Encoding.UTF8.GetBytes(input));

        var sBuilder = new StringBuilder();

        // Loop through each byte of the hashed data
        // and format each one as a hexadecimal string.
        for (int i = 0; i < data.Length; i++)
        {
            sBuilder.Append(data[i].ToString("x2"));
        }

        // Return the hexadecimal string.
        return sBuilder.ToString();
    }
}

public static class NameValueCollectionExtensions
{
    public static string ToQueryString(this NameValueCollection nvc)
    {
        return string.Join("&",
            nvc.AllKeys.Select(
                key => string.Format("{0}={1}", key, nvc[key])));
    }
}
```

The GetQueryString() is the first thing to calculate based on the Date/Time and public and private key.

Once the collection is created, we can fill out all of the important base information for the API including what version of the API we want to hit.

The GetApiKey() only requires a timestamp and we receive a hash key to pass to Marvel to validate our request.

Create the Service

Since we finished the BaseMarvelApi, the hard work is done. All we need to do is pick an API and create the call.

Let's focus on searching for a particular comic.

```
public class MarvelComicApi : BaseMarvelApi, IMarvelComicApi
{
    private NameValueCollection SearchTerms { get; set; }

    public MarvelComicApi(IHttpClientFactory factory) : base(factory)
    {
        BasePath = "/comics";
    }

    public async Task<MarvelComic> GetComics()
    {
        var request = GetHttpRequestBuilder(BasePath);
        return await base.SendRequest<MarvelComic>(request.GetHttpMessage());
    }

    public async Task<MarvelComic> GetComicsByTitle(string title)
    {
        var request = GetHttpRequestBuilder(BasePath)
            .AddQueryString("title", title);

        return await base.SendRequest<MarvelComic>(request.GetHttpMessage());
    }
}
```

I added two methods described in the API documentation: GetComics() and GetComicsByTitle().

The first method allows us to return a lot of comics (it's paged and you have certain limitations on how much you can retrieve) whereas the second method allows you to search for comics.

Since the search is query-string based, it's easy to append it to the URL and retrieve the results.

With the Marvel API finished, we can search the entire Marvel catalog for Wolverine titles.

```
public async Task<IActionResult> Index()
{
    var comics = await _marvel.GetComicsByTitle("Wolverine");

    return View(comics);
}
```

With a vast amount of Marvel APIs available, all of them are querystring based. You could easily create a builder pattern for each Marvel API alone.

Conclusion

We've gone through the step-by-step process of refactoring an HttpClient into something reusable across an entire application. This simple concept was extended into builder patterns and allows the developer to write server-side API calls quicker while also making them readable.

We also created three, real-world samples to validate our workflow functions as expected.

With the amount of APIs available today, this infrastructure gives you more flexibility with various web services across the Internet.

I hope this book has made HttpClient classes easier to work with again.

Now, the world is truly your oyster.

The question is how will you evolve your application with all of the APIs available on the Internet?

Appendix A: Resources

Programmable Web - Catalog of Web APIs on the Internet
https://www.danylkoweb.com/go/pgrweb

Any API (any-api.com) - Great collection of over 1400 public APIs
https://www.danylkoweb.com/go/anyapi

ASP.NET Monsters - You're Using HttpClient Wrong And It Is Destabilizing Your Software
https://www.danylkoweb.com/go/HttpClientWrong

ASP.NET Core Diagnostic Scenarios (Github) - Async Guidance from David Fowler
https://www.danylkoweb.com/go/AsyncGuidance

Microsoft REST API Guidelines (Github)
https://www.danylkoweb.com/go/api-guidelines

You're (probably still) using HttpClient wrong and it is destabilizing your software (JosefOttosson.se)
https://www.danylkoweb.com/go/httpclientwrongstill

Use IHttpClientFactory to implement resilient Http requests (docs.microsoft.com) - HttpClientFactory *was* created to help developers with their improper implementations of the HttpClient class.
https://www.danylkoweb.com/go/httpclientfactory

Http Methods for RESTful services (restapitutorial.com)
https://www.danylkoweb.com/go/httpmethods