

Forum:[Hack Forums](#)

Thread: [Asynchronous Download Manager \[Review Requested\]](#)

PavilionVI - April 25, 2016, 9:27 am

Hello HF,

I was trying to get the most out of the new .Net HttpClient and I decided to make a download manager that allow you to download multiple files asynchronously and control the number of simultaneous downloads.

There are plenty of tutorials on the HttpClient so I had no problem working with it, however, the tricky part was managing the simultaneous downloads. At first, I used Parallel.ForEach which worked fine and then I found this topic on Code Review.

Code

[codereview\[.\]stackexchange\[.\]com/questions/18519/real-world-async-and-await-code-example](#)

Sorry^^

The answer which was given there was exactly what I was looking for but I didn't quite understand the use of ISourceBlock.

I made few changes to that code and I came up with following:

PHP Code

```
<?php
class DManager
{
    public CookieContainer Cookies { get; set; }
    public int MaxConnections { get; set; }
    public string BaseDirectory { get; set; }

    public ISourceBlock<DownloadedFile> DownloadFiles(List<string> links)
    {
        var Block = new TransformManyBlock<string, DownloadedFile>(
            async url =>
            {
                HttpClientHandler myHandler = new HttpClientHandler()
                {
                    CookieContainer = Cookies
                };
                return await DownloadFile(url, myHandler);
            }, new ExecutionDataflowBlockOptions { MaxDegreeOfParallelism = MaxConnections });

        foreach (string link in links)
            Block.Post(link);
        Block.Complete();
        return Block;
    }

    private async Task<DownloadedFile[]> DownloadFile(string url, HttpClientHandler handler)
    {
```

```

DownloadedFile myFile = new DownloadedFile()
{
    FileName = Path.Combine(BaseDirectory, Path.GetFileName(url))
};

try
{
    using (HttpClient myClient = new HttpClient(handler))
    {
        myClient.DefaultRequestHeaders.Referrer = new Uri("YOUR LINK");
        myClient.DefaultRequestHeaders.UserAgent.TryParseAdd(
"Mozilla/5.0 (Windows NT 6.1)");

        using (HttpResponseMessage Response = await myClient.GetAsync(url))
        using (Stream httpStream = await Response.Content.ReadAsStreamAsync())
        using (FileStream filestream = new FileStream(myFile.FileName, FileMode.Create,
FileAccess.Write, FileShare.None, 4096, true))
            await httpStream.CopyToAsync(filestream, 4096);
    }
}
catch (Exception ex)
{
    Console.WriteLine("Error Downloading: {0}\r\nMessage: {1}", url, ex.Message);
    return new DownloadedFile[0];
}
return new[] { myFile };
}

internal class DownloadedFile
{
    public string FileName { get; set; }
}

```

Well the code works fine but to be honest, I'm just not comfortable with the way `ISourceBlock` is used here, I think there is a better way of doing it but I can't figure it out since I couldn't find any resources on the web so I need your help to understand this part.

There is a lot more work to do like handling different exceptions, the ability to cancel, pause/resume a download so if you have any thoughts on this please share with us.

0xDEAD10CC - April 25, 2016, 10:16 am

Why did you create a regular class for this? And a class for a string only? The functions don't really share any data, you could have put them into an abstract class. I don't see there being a reason as to why you would instantiate multiple download managers, there should just be ONE which handles all of the outbound connections. Also, the parameter could be more generic as an `IList` rather than a `List` I believe. There's no real good reason as to why you couldn't have just marked the function itself as `async` however and renamed it as `DownloadFilesAsync()`

Also, the points mentioned on SE were good:

Quote:2. You shouldn't read the response as a single byte array. Instead, you should use Streams where possible (it certainly is possible in your case), because they are more efficient.

3. Don't just ignore unknown exceptions. If there are some exceptions which you want to ignore

(e.g. when a website returns 404), specify them explicitly.

^ HTTP status codes are important.

I've proposed an edit to his post though. He has an async function that is NOT properly named with the "Async" suffix in terms of good naming conventions.

NOTICE - He subtly changed your function and added a static modifier as well; this is exactly what I'm talking about with the instantiation part.

Additionally, look into the SemaphoreSlim class.

Here was my proposed edit:

```
Code public ISourceBlock<DownloadedFile> DownloadFiles(IList<string> fileIds, string
securityCookieString, string securityCookieDomain)
{
    var urls = CreateUrls(fileIds);

    // we have to use TransformManyBlock here, because we want to be able to return 0 or 1 items
    var block = new TransformManyBlock<string, DownloadedFile>(
        async url =>
        {
            var httpClientHandler = new HttpClientHandler();
            if (string.IsNullOrEmpty(securityCookieString))
                return await DownloadFileAsync(url, httpClientHandler);

            var securityCookie = new Cookie(FormsAuthentication.FormsCookieName,
securityCookieString)
            {
                Domain = securityCookieDomain
            };
            httpClientHandler.CookieContainer.Add(securityCookie);

            return await DownloadFileAsync(url, httpClientHandler);
        }, new ExecutionDataflowBlockOptions { MaxDegreeOfParallelism =
DefaultMaxConcurrentDownloads });

    foreach (var url in urls)
        block.Post(url);

    block.Complete();

    return block;
}

private static async Task<DownloadedFile[]> DownloadFileAsync(string url, HttpClientHandler
clientHandler)
{
    var client = new HttpClient(clientHandler);
    var downloadedFile = new DownloadedFile();

    try
    {
```

```

var responseMessage = await client.GetAsync(url).ConfigureAwait(false);
if (!responseMessage.IsSuccessStatusCode)
{
    // TODO: handle
}

if (responseMessage.Content.Headers.ContentDisposition == null)
    return new DownloadedFile[0];

downloadedFile.FileName = Path.Combine(
    DefaultWorkingDirectory,
    responseMessage.Content.Headers.ContentDisposition.FileName);

if (!Directory.Exists(DefaultWorkingDirectory))
{
    Directory.CreateDirectory(DefaultWorkingDirectory);
}

const int bufferSize = 4096; // avoid magic numbers, explicit self-documenting variable name
using(var httpStream = await responseMessage.Content.ReadAsStreamAsync())
using(var filestream = new FileStream(
    downloadedFile.FileName, FileMode.Create, FileAccess.Write, FileShare.None, bufferSize,
true))
{
    await httpStream.CopyToAsync(filestream, bufferSize);
}
}
// TODO: improve
catch (Exception ex)
{
    return new DownloadedFile[0];
}

return new[] { downloadedFile };
}

```

Should be a good start. Note, that I also added in `ConfigureAwait(false)` for the async function call that is awaited to avoid the context switch.

PavilionVI - April 25, 2016, 1:34 pm

First of all, this is not my code I just found on SE and made few changes, in my code I used the `Parallel.ForEach` loop in a simple void to handle the simultaneous downloads and the reason I posted this code here is because I'm interested in the use of the `ISourceBlock` interface (maybe I'll ask a new question on SE).

The `DownloadedFile` class here is used to store some data about the file being downloaded like the file size for progress reporting, a unique id for cancellation or pause/resume... Well that's I plan to do later and I totally agree that there's no need for instantiating multiple download managers.

If you were to use this on a bot (Imgur or 4chan scraper) and you don't need to cancel or pause a download what would you use as a return value, would `public ISourceBlock<string> DownloadFiles()` work? Also, why did you use `IList<string>` instead of `List<string>`?

I would like to thank you for the tips, I didn't know it was bad to use magic numbers and I will look further into `ConfigureAwait()`.

0xDEAD10CC - April 25, 2016, 1:55 pm

>> Also, why did you use `IList<string>` instead of `List<string>`?

Because `IList<string>` allows me to pass an array without casting to a `List<T>` object. As I said, it's more generic. `List<T>` is a specific implementation of `IList<T>`. With a `List<T>` you cannot pass something like `new [] { "lkjsdf" }`, but you can with `IList<T>`. If the param type was `List<string>` you'd have to do `new List<string>(new [] { "lkjsdf" })` even if you didn't need to use `List<T>` functionality.

>> I didn't know it was bad to use magic numbers

Well... Think of what happens when you need to change that 4096 buffer to some other value and you forget to change it in all places of the code where that number needs to be the same value. It's much easier to remember to change it in one place than to have to read through your code and remember which numeric literals need to have the same value and change them. If you had code that was supposed to send X bytes and the expected size of the receive buffer was meant to be the same as the send buffer, your code may break if you change the send buffer to 8192 and the receive buffer is still 4096. Just an example, it may or may not break your code, but the point of re-usable code is to make it easy to change if necessary.

Additionally - a variable name (marked `const` to indicate that it's not meant to be changed) is already more self-documenting if you use a proper identifier for the variable as to what 4096 means. In his code he only used a redundant parameter name specifier once to indicate that it had something to do with the buffer.

If you don't need to do anything then why not just mark a function 'async' and have it return 'Task'? Call it as awaited from an 'async void' function, most likely a delegate assigned to some button event from your UI.

PavilionVI - April 25, 2016, 2:27 pm

(Apr 25, 2016 01:55 PM)0xDEAD10CC Wrote: [\[CITE\]](#) >> I didn't know it was bad to use magic numbers

Well... Think of what happens when you need to change that 4096 buffer to some other value and you forget to change it in all places of the code where that number needs to be the same value. It's much easier to remember to change it in one place than to have to read through your code and remember which numeric literals need to have the same value and change them. If you had code that was supposed to send X bytes and the expected size of the receive buffer was meant to be the same as the send buffer, your code may break if you change the send buffer to 8192 and the receive buffer is still 4096. Just an example, it may or may not break your code, but the point of re-usable code is to make it easy to change if necessary.

If you don't need to do anything then why not just mark a function 'async' and have it return 'Task'? Call it as awaited from an 'async void' function, most likely a delegate assigned to some button event from your UI.

You definitely have a point and your suggestion seems more acceptable to do it, however, any improvements for this code are welcome.

0xDEAD10CC - April 25, 2016, 3:10 pm

Also, I would not suggest the Semaphore class here as suggested. I suggested SemaphoreSlim for this reason: [https://msdn.microsoft.com/en-us/library/hh462723\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/hh462723(v=vs.110).aspx)