Setting the Default Windows System Proxy in .NET

If you are building a client application that uses HTTP connections to interact with remote data, you've likely run into a scenario where you would like to examine what the content that you're sending and receiving actually looks like by hooking up an HTTP proxy like **Fiddler** or some other proxy tooling that can capture request and response data as it is sent. I find myself needing this all the time and I tend to use Fiddler for Windows (Classic) for this task.

Unfortunately, **using proxies in .NET is not automatic** and by default HTTP connections bypass the default Windows System proxy settings. Instead the proxy has to be set explicitly in code, or in an old school `app.config` (yes, this works even for .NET Core on Windows).

In this post I describe how you can get the default Windows proxy - which has changed for .NET Core and is not easy to discover - and how you can use it as part of your application interactively by allowing your users to optionally configure a proxy for debugging.

Getting or Creating a Proxy in .NET Code

If you rather have your application handle proxy settings directly via a configuration that is part of your application's configuration rather than obscure config file settings, you can do so with the following code that lets you either return the default Windows proxy or lets you explicitly set a proxy.

```
/// <summary>
/// Helper method that creates a proxy instance to store on the Proxy property
/// </summary>
/// <param name="proxyAddress">
/// Proxy Address to create or "default" for Windows default proxy.
/// Null or empty means no proxy is set
/// </param>
/// <param name="byPassonLocal">
/// Optional - bypass on local if you're specifying an explicit url
/// </param>
/// <param name="bypassList">
/// Optional list of root domain Urls that are bypassed
/// </param>
/// <returns>A Proxy Instance</returns>
public static IWebProxy CreateWebProxy(string proxyAddress = null,
                                       bool bypassonLocal = false,
                                       string[] bypassList = null)
{
    IWebProxy proxy = null;

    // no proxy - most efficient for HTTP requests
    if (string.IsNullOrEmpty(proxyAddress))
        return null;

    // use the default System Proxy
    if (proxyAddress.Equals("default", StringComparison.OrdinalIgnoreCase))
    {
        proxy = System.Net.WebRequest.GetSystemWebProxy();
    }
    else
    {
        proxy = new System.Net.WebProxy(proxyAddress, bypassonLocal, bypassList);
    }

    return proxy;
}
```

Most proxy tool like Fiddler automatically set the Windows Proxy settings, so you can use `GetSystemWebProxy()` to retrieve those settings using `"default"` as the parameter in the function. The default proxy also works to bypass everything if no proxy is configured in the system - so this setting effectively gives you the typically expected behavior where you don't use a proxy if not configured but you do use once it a proxy is added.

`WebRequest.GetSystemWebProxy()` is an old .NET function that uses the old `WebRequest` class that is generally deprecated in .NET Core, but this utility function still works.

> Note that if the underlying system proxy configuration is changed **while the application is running, that proxy change is not automatically detected by your process** by `GetSystemWebProxy()`. You have to restart the application to see the change.

Application Implementation

Inside of an application it's a good idea to store proxy configuration as part of application configuration. In West

Wind WebSurge, I allow for a proxy address to be supplied and that proxy is used for all HTTP requests. To make this easily accessible for all clients it's stored in the global configuration which holds two values:

- The user configurable ProxyAddress
- A provided Proxy instance that basically calls `CreateWebProxy()`

Here's what that looks like in my application's config class:

```
/// <summary>
/// Optionally specify an Http Proxy
/// null or empty - no Proxy
/// default - use the Windows Default System Proxy
/// url - explicit Proxy Url
/// </summary>
public string ProxyUrl { get; set; } = "default";


/// <summary>
/// Retrieve a full, ready to use WebProxy instance based on the entered
/// Proxy Url. If no proxy is specified returns null.
/// </summary>
[JsonIgnore]
public IWebProxy Proxy
{
    get
    {
        return HttpUtils.CreateWebProxy(ProxyUrl, false);
    }
}
```

ProxyUrl is the user configurable value that is stored in the configuration file, while `Proxy` is an actual instance that is created and provided as needed.

In the app I can then just get the proxy along with other config options. The most common place is to assign it to an Http handler instance as part of the HttpClient setup which looks something like this:

```
public HttpClient GetHttpClient(...)
{
    var socketsHandler = new SocketsHttpHandler
    {
       PooledConnectionLifetime = TimeSpan.FromMinutes(3),
       PooledConnectionIdleTimeout = TimeSpan.FromMinutes(2),
       UseCookies = false // we'll manually track cookies with CookieContainer
    };

    socketsHandler.MaxConnectionsPerServer = Options.ThreadCount;
    if (socketsHandler.MaxConnectionsPerServer > Options.MaxConnections)
       socketsHandler.MaxConnectionsPerServer = Options.MaxConnections;

    // ...  more handler configuration for credentials, cookies etc.

    if (!string.IsNullOrEmpty(wsApp.Configuration.ProxyUrl))
    {
       // uses default proxy unless explicitly overridden
       socketsHandler.UseProxy = true;

       // UseProxy uses default proxy - so let's not add overhead of discovering it again
       if (!wsApp.Configuration.ProxyUrl.Equals("default",
StringComparison.OrdinalIgnoreCase))
       {
           socketsHandler.Proxy = wsApp.Configuration.Proxy;
       }
    }
    else
    {
       socketsHandler.UseProxy = false;
    }

    var httpClient = new System.Net.Http.HttpClient(socketsHandler);
    httpClient.Timeout =
       TimeSpan.FromSeconds(Options.RequestTimeoutMs < 10 ? 10 : Options.RequestTimeoutMs);

    return httpClient;
}
```

Internally I then cache the `HttpClient` instances - one per 'session' so I can track users/sessions and cookies for multiple users simultaneously.

> This is a pain necessary due to the disconnected nature of HttpClient's handler/session model and it pisses me off everytime I use it. But alas this is what we got, and as much as it bugs me it does work well in terms of stability and performance which matters a lot in WebSurge.