Documentation For BestHTTP Pro And Basic

Introduction

You can read this documentation online too.

BestHTTP is a HTTP/1.1 implementation based on the RFC 2616, that supports almost all Unity mobile and standalone platforms (see Supported platforms).
My goal was to create an easy to use, but still powerful plugin to Unity to take advantage of the potential in HTTP/1.1.
This document is a quick guide, not all function and property can be found here.
You can also find some useful demo in the BestHTTP package:

- Load Image Test: A simple interactive demo to demonstrate downloading an image and using it as a texture.
- Rest API Test: An interactive demo to demonstrate how BestHTTP can be used with a Rest API service. This demo needs a json deserializer (LitJson included), but BestHTTP itself doesn't have any 3rd party dependency.
- Audio Streaming Test: A very basic example to demonstrate the streaming capabilities of BestHTTP.
- Stream Leaderboard Test: Another streaming example to demonstrate slow bandwidth(or slow server processing) and clever client side caching. This way we spare user and server bandwidth and processing power on the server.

For support, feature request or general questions you can email me at besthttp@gmail.com.

Installation

You need to move the *Plugins* folder from the *Best Http (Basic/Pro)* folder to your *Assets* folder.
Under Unity 3.5 you need to delete the WP8 folder inside the Plugins folder.

Getting Started Quickly

First, you should add a using statement to your source file after the regular usings:

```
using BestHTTP;
```

GET Requests

The simplest way to do a request to a web server is to create a HTTPRequest object providing the url and a callback function to it's constructor. After we constructed a new HTTPRequest object the only thing we need to do, is actually send the request with the Send() function. Let's see an example:

```
HTTPRequest request = new HTTPRequest(new Uri("https://google.com"), onRequestFinished);
request.Send();
```

The OnRequestFinished() function's implementation might be this:

```
void OnRequestFinished(HTTPRequest request, HTTPResponse response)
{
        Debug.Log("Request Finished! Text received: " + response.DataAsText);
}
```

As you can see the callback function always receives the original HTTPRequest object and an HTTPResponse object that holds the response from the server. The HTTPResponse object is null if there were an error and the request object has an Exception property that might carry extra information about the error if there were any.

While the requests are always processed on separate threads, calling the callback function is done on Unity's main thread, so we don't have to do any thread synchronization.

If we want to write more compact code we can use c#'s lambda expressions. In this example we don't even need a temporary variable:

```
new HTTPRequest(new Uri("https://google.com"), (request, response) =>
Debug.Log("Finished!")).Send();
```

POST Requests

The above examples were simple GET requests. If we doesn't specify the method, all requests will be GET requests by default. The constructor has another parameter that can be used to specify the method of the request:

```
HTTPRequest request = new HTTPRequest(new Uri("yourserver.com/posturi"),
HTTPMethods.Post,
                                      OnRequestFinished);
request.AddField("FieldName", "Field Value");
request.Send();
```

To POST any data without setting a field you can use the **RawData** property:

```
HTTPRequest request = new HTTPRequest(new Uri("yourserver.com/posturi"),
HTTPMethods.Post,
                                      OnRequestFinished);
request.RawData =  Encoding.UTF8.GetBytes("Field Value");
request.Send();
```

Beside **GET** and **POST** you can use the **HEAD**, **PUT**, **DELETE** and **PATCH** methods as well:

Head

```
HTTPRequest request = new HTTPRequest(new Uri("server.com/posturi"), HTTPMethods.Head,
                                      OnRequestFinished);
request.Send();
```

Put

```
HTTPRequest request = new HTTPRequest(new Uri("server.com/posturi"), HTTPMethods.Put,
                                      OnRequestFinished);
request.Send();
```

Delete

```
HTTPRequest request = new HTTPRequest(new Uri("server.com/posturi"),
HTTPMethods.Delete,
                                      OnRequestFinished);
request.Send();
```

Patch

```
HTTPRequest request = new HTTPRequest(new Uri("server.com/posturi"),
HTTPMethods.Patch,
                                      OnRequestFinished);
request.Send();
```

How To Access The Downloaded Data

Most of the time we use our requests to receive some data from a server. The raw bytes can be accessed from the HTTPResponse object's Data property. Let's see an example how to download an image:

```
new HTTPRequest(new Uri("http://yourserver.com/path/to/image.png"), (request, response) =>
{
        var tex = new Texture2D(0, 0);
        tex.LoadImage(response.Data);
        guiTexture.texture = tex;
}).Send();
```

Of course there is a more compact way to do this:

```
new HTTPRequest(new Uri("http://yourserver.com/path/to/image.png"), (request, response) =>
                guiTexture.texture = response.DataAsTexture2D).Send();
```

Beside of DataAsTexture2D there is a **DataAsText** property to decode the response as an Utf8 string. More data decoding properties may be added in the future. If you have an idea don't hesitate to mail me.

Warning: All examples in this document are without any error checking! In the production code make sure to add some null checks.

Switching from WWW

You can yield a HTTPRequest with the help of a StartCoroutine call:

```
HTTPRequest request = new HTTPRequest(new Uri("http://server.com"));
request.Send();
yield return StartCoroutine(request);
Debug.Log("Request finished! Downloaded Data:" + request.Response.DataAsText);
```

The Debug.Log will be called only when the request is done. This way you don't have to supply a callback function(however, you still can if you want to).

Advanced Topics

This section will cover some of the advanced usage that can be done with BestHTTP.

We can easily enable and disable some basic features with the help of the HTTPRequest class' constructor. These parameters are the following:

- **methodType**: What kind of request we will send to the server. The default methodType is ***HTTPMethods.Get***.
- **isKeepAlive**: Indicates to the server that we want the tcp connection to stay open, so consecutive http requests doesn't need to establish the connection again. If we leave it to the default true, it can save us a lot of time. If we know that we won't use requests that often we might set it to false. The default value is **true**.
- **disableCache**: Tells to the BestHTTP system to use or skip entirely the caching mechanism. If its value is true the system will not check the cache for a stored response and the response won't get saved neither. The default value is **false**.

Authentication

Best HTTP supports Basic and Digest authentication through the HTTPRequest's Credentials property:

```
using BestHTTP.Authentication;

var request = new HTTPRequest(new Uri("https://httpbin.org/digest-auth/auth-int/usr/paswd"), (req, resp)
=>
        {
                if (resp.StatusCode != 401)
                        Debug.Log("Authenticated");
                else
                        Debug.Log("NOT Authenticated");

                Debug.Log(resp.DataAsText);
```

```
    });
request.Credentials = new Credentials("usr", "paswd");
request.Send();
```

Streaming

By default the callback function we provide to the HTTPRequest's constructor will be called only once, when the server's answer is fully downloaded and processed. This way, if we'd like to download a bigger file we'd quickly run out of memory on a mobile device. Our app would crash, users would be mad at us and the app would get a lot of bad rating. And rightfully so.
To avoid this, BestHTTP is designed to handle this problem very easily: with only switching one flag to true, our callback function will be called every time when a predefined amount of data is downloaded.
Additionally if we didn't turn off caching, the downloaded response will be cached so next time we can stream the whole response from our local cache without any change to our code and without even touching the web server. *(Remarks: the server must send valid caching headers ("Expires" header: see the RFC) to allow this.)*

Lets see a quick example:

```
var request = new HTTPRequest(new Uri("http://yourserver.com/bigfile"), (req, resp) =>
        {
                List<byte[]> fragments = resp.GetStreamedFragments();

                // Write out the downloaded data to a file:
                using (FileStream fs = new FileStream("pathToSave", FileMode.Append))
                        foreach(byte[] data in fragments)
                                fs.Write(data, 0, data.Length);

                if (resp.IsStreamingFinished)
                        Debug.Log("Download finished!");
        });
request.UseStreaming = true;
request.StreamFragmentSize = 1 * 1024 * 1024; // 1 megabyte
request.DisableCache = true; // already saving to a file, so turn off caching
request.Send();
```

So what just happened above?

- We switched the flag - UseStreaming - to true, so our callback may be called more than one times.
- The StreamFragmentSize indicates the maximum amount of data we want to buffer up before our callback will be called.
- Our callback will be called every time when our StreamFragmentSize sized chunk is downloaded, and one more time when the IsStreamingFinished set to true.
- To get the downloaded data we have to use the *GetStreamedFragments()* function. We should save its result in a temporary variable, because the internal buffer is cleared in this call, so consecutive calls will give us null results.

- We disabled the cache in this example because we already saving the downloaded file and we don't want to take too much space.

Caching

Caching is based on the HTTP/1.1 RFC too. It's using the headers to store and validate the response. The caching mechanism is working behind the scenes, the only thing we have to do is to decide if we want to enable or disable it.
If the cached response has an 'Expires' header with a future date, BestHTTP will use the cached response without validating it with the server. This means that we don't have to initiate any tcp connection to the server. This can save us time, bandwidth and **works offline** as well.

Although caching is automatic we have some control over it, or we can gain some info using the public functions of the *HTTPCacheService* class:

- **BeginClear()**: It will start clearing the entire cache on a separate thread.
- **BeginMaintainence**(): With this function's help, we can delete cached entries based on the last access time. It deletes entries that's last access time is older than the specified time. We can also use this function to keep the cache size under control:

```
// Delete cache entries that weren't accessed in the last two weeks, then delete entries to keep the size of
// the cache under 50 megabytes, starting with the oldest.
HTTPCacheService.BeginMaintainence(new HTTPCacheMaintananceParams(TimeSpan.FromDays(14),
        50 * 1024 * 1024));
```

- **GetCacheSize()**: Will return the size of the cache in bytes.
- **GetCacheEntryCount()**: Will return the number of the entries stored in the cache. The average cache entry size can be computed with the *float avgSize = GetCacheSize() / (float)GetCacheEntryCount()* formula.

Cookies

Handling of cookie operations are transparent to the programmer. Setting up the request Cookie header and parsing and maintaining the response's Set-Cookie header are done automatically be the plugin.
However it can be controlled in various ways:

- It can be disabled per-request or globally by setting the HTTPRequest object's *IsCookiesEnabled* property or the *HTTPManager.IsCookiesEnabled* property.
- Cookies can be deleted from the Cookie Jar by calling the CookieJar.Clear() function.
- New cookies that are sent from the server are can be accessed through the response's Cookies property.
- There are numerous global setting regarding to cookies. See the Global Settings section for more information.

Proxy

A HTTPProxy object can be set to a HTTPRequest's Proxy property. This way the request will be go through the given proxy.

```
request.Proxy = new HTTPProxy(new Uri("http://localhost:3128"));
```

Download Progress Tracking

To track and display download progress you can use the OnProgress event of the HTTPRequest class. This event's parameters are the original HTTPRequest object, the downloaded bytes and the expected length of the downloaded content.

```
var request = new HTTPRequest(new Uri(address), OnFinished);
request.OnProgress = OnDownloadProgress;
request.Send();

void OnDownloadProgress(HTTPRequest request, int downloaded, int length) {
        float progressPercent = (downloaded / (length * 100.0f));
        Debug.Log("Downloaded: " + progressPercent.ToString("F2") + "%");
}
```

Aborting a Request

You can abort an *ongoing* request by calling the HTTPRequest object's Abort() function:

```
request = new HTTPRequest(new Uri("http://yourserver.com/bigfile"), (req, resp) => { … });
request.Send();

// And after some time:
request.Abort();
```

The callback function will be called and the response object will be null.

Timeouts

You can set two timeout for a request:

1. **ConnectTimeout**: With this property you can control how much time you want to wait for a connection to be made between your app and the remote server. It's default value is 20 seconds.

```
request = new HTTPRequest(new Uri("http://yourserver.com/"), (req, resp) => { … });
request.ConnectTimeout = TimeSpan.FromSeconds(2);
request.Send();
```

1. **Timeout**: With this property you can control how much time you want to wait for a request to be processed(sending the request, and downloading the response).

```
request = new HTTPRequest(new Uri("http://yourserver.com/"), (req, resp) => { … });
```

```
request.Timeout = TimeSpan.FromSeconds(10);
request.Send();
```

A more complete example:

```
string url ="http://besthttp.azurewebsites.net/api/LeaderboardTest?from=0&count=10";
HTTPRequest request = new HTTPRequest(new Uri(url), (req, resp) =>
  {
        switch (req.State)
        {
              // The request finished without any problem.
              case HTTPRequestStates.Finished:
                    Debug.Log("Request Finished Successfully!\n" + resp.DataAsText);
                    break;

              // The request finished with an unexpected error.
              // The request's Exception property may contain more information about the
              error.
              case HTTPRequestStates.Error:
                    Debug.LogError("Request Finished with Error! " +
                          (req.Exception != null ?
                                (req.Exception.Message + "\n" +
                          req.Exception.StackTrace) :
                                "No Exception"));
                    break;

              // The request aborted, initiated by the user.
              case HTTPRequestStates.Aborted:
                    Debug.LogWarning("Request Aborted!");
                    break;

              // Ceonnecting to the server timed out.
              case HTTPRequestStates.ConnectionTimedOut:
                    Debug.LogError("Connection Timed Out!");
                    break;

              // The request didn't finished in the given time.
              case HTTPRequestStates.TimedOut:
                    Debug.LogError("Processing the request Timed Out!");
                    break;
        }
  });

// Very little time, for testing purposes:
//request.ConnectTimeout = TimeSpan.FromMilliseconds(2);
request.Timeout = TimeSpan.FromSeconds(5);
request.DisableCache = true;
request.Send();
```

Request States

All request has a State property that contains it's internal state. The possible states are the following:

- **Initial**: Initial status of a request. No callback will be called with this status.
- **Queued**: Waiting in a queue to be processed. No callback will be called with this status.
- **Processing**: Processing of the request started. In this state the client will send the request, and parse the response. No callback will be called with this status.
- **Finished**: The request finished without problem. Parsing the response done, the result can be used. The user defined callback will be called with a valid response object. The request's Exception property will be null.
- **Error**: The request finished with an unexpected error in the plugin. The user defined callback will be called with a null response object. The request's Exception property may contain more info about the error, but it can be null.
- **Aborted**: The request aborted by the client(HTTPRequest's Abort() function). The user defined callback will be called with a null response. The request's Exception property will be null.
- **ConnectionTimedOut**: Connecting to the server timed out. The user defined callback will be called with a null response. The request's Exception property will be null.
- **TimedOut**: The request didn't finished in the given time. The user defined callback will be called with a null response. The request's Exception property will be null.

For a usage example see the previous section's example.

Global Settings

With the following properties we can change some defaults that otherwise should be specified in the HTTPRequest's constructor. So most of these properties are time saving shortcuts.
These changes will affect all request that created after their values changed.

Changing the defaults can be made through the static properties of the **HTTPManager** class:

- **MaxConnectionPerServer**: Number of connections allowed to a unique host. *http://example.org* and *https://example.org* are counted as two separate servers. The default value is **4**.
- **KeepAliveDefaultValue:** The default value of the HTTPRequest's **IsKeepAlive** property. If IsKeepAlive is false, the tcp connections to the server will be set up before every request and closed right after it. It should be changed to false if consecutive requests are rare. Values given to the HTTPRequest's constructor will override this value for only this request. The default value is **true**.
- **IsCachingDisabled:** With this property we can globally disable or enable the caching service. Values given to the HTTPRequest's constructor will override this value for only this request. The default value is **true**.

- **MaxConnectionIdleTime:** Specifies the idle time BestHTTP should wait before it destroys the connection after it's finished the last request. The default value is 2 minutes.
- **IsCookiesEnabled:** With this option all Cookie operation can be enabled or disabled. The default value is **true**.
- **CookieJarSize:** With this option the size of the Cookie store can be controlled. The default value is 10485760 (**10 MB**).
- **EnablePrivateBrowsing:** If this option is enabled no Cookie will be written to the disk. The default value is **false**.
- **ConnectTimeout**: With this option you can set the HTTPRequests' default ConnectTimeout value. The default value is 20 seconds.
- **RequestTimeout**: With this option you can set the HTTPRequests' default Timeout value. The default value is 60 seconds.
- **RootCacheFolderProvider**: By default the plugin will save all cache and cookie data under the path returned by Application.persistentDataPath. You can assign a function to this delegate to return a custom root path to define a new path. This delegate will be called on a non Unity thread!

Sample codes:

```
HTTPManager.MaxConnectionPerServer = 10;
HTTPManager.RequestTimeout = TimeSpan.FromSeconds(120);
```

Thread Safety

Because the plugin internally uses threads to process all requests parallelly, all shared resources(cache, cookies, etc) are designed and implemented thread safety in mind.
Some notes that good to know:

1. Calling the requests' callback functions, and all other callbacks (like the WebSocket's callbacks) are made on Unity's main thread(like Unity's events: awake, start, update, etc) so you don't have to do any thread synchronization.

Creating, sending requests on more than one thread are safe too, but you should call the *BestHTTP.HTTPManager.Setup();* function before sending any request from one of Unity's events(eg. awake, start).

WebSocket

We can use the WebSocket feature through the WebSocket class. We just need to pass the Uri of the sever to the WebSocket's constructor:

```
var webSocket = new WebSocket(new Uri("wss://html5labs-interop.cloudapp.net/echo"));
```

After this step we can register our event handlers to several events:

- **OnOpen** event: Called when connection to the server is established. After this event the WebSocket's IsOpen property will be True until we or the server closes the connection or if an error occurs.

```
webSocket.OnOpen += OnWebSocketOpen;
private void OnWebSocketOpen(WebSocket webSocket)
{
        Debug.Log("WebSocket Open!");
}
```

- **OnMessage** event: Called when a textual message received from the server.

```
webSocket.OnMessage += OnMessageReceived;
private void OnMessageReceived(WebSocket webSocket, string message)
{
        Debug.Log("Text Message received from server: " + message);
}
```

- **OnBinary** event: Called when a binary blob message received from the server.

```
webSocket.OnBinary += OnBinaryMessageReceived;
private void OnBinaryMessageReceived(WebSocket webSocket, byte[] message)
{
        Debug.Log("Binary Message received from server. Length: " + message.Length);
}
```

- **OnClosed** event: Called when the client or the server closes the connection, or en internal error occurs. When the client closes the connection through the Close function it can provide a Code and a Message that indicates a reason for closing. The server typically will echos our Code and Message.

```
webSocket.OnClosed += OnWebSocketClosed;
private void OnWebSocketClosed(WebSocket webSocket, UInt16 code, string message)
{
        Debug.Log("WebSocket Closed!");
}
```

- **OnError** event: Called when the we can't connect to the server, an internal error occurs or when the connection lost. The second parameter is an Exception object, but it can be null. In this case checking the InternalRequest of the WebSocket should tell more about the problem.

```
webSocket.OnError += OnError;
private void OnError(WebSocket ws, Exception exception)
{
        string errorMsg = string.Empty;
        if (ws.InternalRequest.Response != null)
                errorMsg = string.Format("Status Code from Server: {0} and Message: {1}",
                        ws.InternalRequest.Response.StatusCode,
                        ws.InternalRequest.Response.Message);

        Debug.Log("An error occured: " + (ex != null ? ex.Message : "Unknown Error " +
errorMsg);
}
```

- **OnIncompleteFrame** event: See Streaming at the Advanced Websocket topic.

After we registered to the event we can start open the connection:

```
webSocket.Open();
```

After this step we will receive an OnOpen event and we can start sending out messages to the server.

```
// Sending out text messages:
webSocket.Send("Message to the Server");

// Sending out binary messages:
byte[] buffer = new byte[length];
//fill up the buffer with data
webSocket.Send(buffer);
```

After all communication is done we should close the connection:

```
webSocket.Close();
```

Advanced WebSocket

- **Ping** messages: Its possible to start a new thread to send Ping messages to the server by setting the StartPingThread property to True before we receive an OnOpen event. This way Ping messages will be sent periodically to the server. The delay between two ping can be set in the PingFrequency property (it's default is 1000ms).

- **Pong** messages: All ping messages that received from the server the plugin will automatically generate a Pong answer.

- **Streaming**: Longer text or binary messages will get fragmented. These fragments are assembled by the plugin automatically by default. This mechanism can be overwritten if we register an event handler to the WebSocket's **OnIncompleteFrame** event. This event called every time the client receives an incomplete fragment. These fragments will be ignored by the plugin, it doesn't try to assemble these nor store them. This event can be used to achieve streaming experience.

Small Code-Samples

Add custom header

```
var request = new HTTPRequest(new Uri("http://server.com"), HTTPMethods.Post, onFinished);
request.SetHeader("Content-Type", "application/json; charset=UTF-8");
request.RawData = UTF8Encoding.GetBytes(ToJson(data));
request.Send();
```

Display download progress

```
var request = new HTTPRequest(
                new Uri("http://besthttp.azurewebsites.net/api/LeaderboardTest?from=0&count=10"),
                (req, resp) =>
        {
          Debug.Log("Finished!");
        });

request.OnProgress += (req, down, length) =>
                Debug.Log(string.Format("Progress: {0:P2}", down / (float)length));
request.Send();
```

Abort a request

```
var request = new HTTPRequest(new Uri(address), (req, resp) =>
        {
                // State should be HTTPRequestStates.Aborted if we call Abort() before
                //  it's finishes
                Debug.Log(req.State);
        });
request.Send();
```

```
request.Abort();
```

Range request for resumable download

First request is a Head request to get the server capabilities. When range requests are supported the DownloadCallback function will be called. In this function we will create a new real request to get chunks of the content with setting the callback function to this function too. The current download position saved to the PlayerPrefs, so the download can be resumed even after an application restart.

```
private const int ChunkSize = 1024 * 1024; // 1 MiB - should be bigger!
private string saveTo = "downloaded.bin";

void StartDownload(string url)
{
  // This is a HEAD request, to get some information from the server
  HTTPRequest headRequest = new HTTPRequest(new Uri(url),
                                            HTTPMethods.Head,
                                            (request, response) =>
        {
            if (response == null)
                Debug.LogError("Response null. Server unreachable? Try again later.");
            else
            {
                if (response.StatusCode == 416)
                    Debug.LogError("Requested range not satisfiable");
                else if (response.StatusCode == 200)
                    Debug.LogError("Partial content doesn't supported by the
    server, content can be downloaded as a whole.");
                else if (response.HasHeaderWithValue("accept-ranges", "none"))
                    Debug.LogError("Server doesn't supports the 'Range' header!
    The file can't be downloaded in parts.");
                else
                    DownloadCallback(request, response);
            }
        });

  // Range header for our head request
  int startPos = PlayerPrefs.GetInt("LastDownloadPosition", 0);
  headRequest.SetRangeHeader(startPos, startPos + ChunkSize);

  headRequest.DisableCache = true;
  headRequest.Send();
}

void DownloadCallback(HTTPRequest request, HTTPResponse response)
{
  if (response == null)
```

```
  {
        Debug.LogError("Response null. Server unreachable, or connection lost? Try again
later.");
        return;
  }

  var range = response.GetRange();

  if (range == null)
  {
        Debug.LogError("No 'Content-Range' header returned from the server!");
        return;
  }
  else if (!range.IsValid)
  {
        Debug.LogError("No valid 'Content-Range' header returned from the server!");
        return;
  }

  // Save(append) the downloaded data to our file.
  if (request.MethodType != HTTPMethods.Head)
  {
        using (FileStream fs = new FileStream(Path.Combine(Application.temporaryCachePath,
                                                           saveTo)
                                              , FileMode.Append))
              fs.Write(response.Data, 0, response.Data.Length);

        // Save our position
        PlayerPrefs.SetInt("LastDownloadPosition", range.LastBytePos);

        // Some debug output
        Debug.LogWarning(string.Format("Download Status: {0}-{1}/{2}",
                                       range.FirstBytePos,
                                       range.LastBytePos,
                                       range.ContentLength));

        // All data downloaded?
        if (range.LastBytePos == range.ContentLength - 1)
        {
              Debug.LogWarning("Download finished!");
              return;
        }
  }

  // Create the real GET request. The callback function is the function that we are currently in!
  var downloadRequest = new HTTPRequest(request.Uri,
                                        HTTPMethods.Get,
                                        /*isKeepAlive:*/ true,
                                        DownloadCallback);
```

```
    // Set the next range's position.
    int nextPos = 0;
    if (request.MethodType != HTTPMethods.Head)
            nextPos = range.LastBytePos + 1;
    else
            nextPos = PlayerPrefs.GetInt("LastDownloadPosition", 0);

    // Set up the Range header
    downloadRequest.SetRangeHeader(nextPos, nextPos + ChunkSize);

    downloadRequest.DisableCache = true;

    // Send our new request
    downloadRequest.Send();
}
```

This is just one implementation. The other would be that start a streamed download, save the chunks and when a failure occur try again with the starting range as the saved file's size.

Currently Supported HTTP/1.1 Features

These features are mostly hidden and automatically used.

- HTTPS
- Store responses in a cache
- Cache validation through the server
- Stream from cache
- Persistent connections
- gzip, deflate content encoding
- Raw and chunked transfer encoding
- Range requests
- Handle redirects
- POST forms and files
- WWW Authentication

Features That Proposed Or Are Under Development

If you want something on this list, feel free to write me a mail about it.

- Bandwidth throttling
- Cache statistics
- Cache tagging and managing
- ~~Abort requests~~ (done)
- ~~WebSockets~~ (done)
- ~~Proxy support~~ (done)
    - Authentication testing and fixes

    - Autodetection
- ~~Cookies~~ (done)
- ~~Authentication~~ (done)
- Http/2(Spdy) support
- ~~WinRT (Win8) support~~ (done)
- Request pipelining
- PlayMaker support
- More examples for various features

Supported Platforms

- iOS (with Unity iOS Pro licence)
- Android (with Unity Android Pro licence)
- Windows Phone 8.1
- Windows Store Apps 8.1
- Windows and Mac Standalone (Linux builds should work too)
- Web Player
    - Best HTTP Pro version only
    - Caching and Cookie persistence not supported
    - Please note, that you still have to run a ***Socket Policy Service*** on the server that you want to access to. For more details see the Unity's Security Sandbox manual's "Implications for use of Sockets" section.

Flash Player and Windows Store builds are not supported currently.

Known Bugs/Limitations

HTTPS

On Android, iOS and desktop platforms .net's Net SslStream are used for HTTPS. This can handle a wide range of certificates, however there are some that can fail with.
To give an alternate solution BouncyCastle are bundled in the plugin, you can use it by setting the UseAlternateSSL to true on your HTTPRequest object. But it can fail on some certifications too.
On Windows Phone 8 the native Ssl implementation is used but with the insecure Ssl protocol. On Windows Phone 8.1(and greater) and on WinRT(Windows Store Apps) a more secure Tls 1.2 protocol will handle the connection.

iOS

- Stripping is unsupported currently.

Android

- No platform specific bugs or limitations are known.

Windows Phone

- No platform specific bugs or limitations are known.

Windows Store Apps

- No platform specific bugs or limitations are known.

WebPlayer

- Only *Best HTTP (Pro)* version can be used.
- Caching and Cookie persistence is not supported.
- You have to run a ***Socket Policy Service*** on the server that you want to access to. For more details see the Unity's Security Sandbox manual's "*Implications for use of Sockets*" section.

FAQ

Q: I received the following error message: *Internal compiler error. See the console log for more information. output was:*
Unhandled Exception: ***System.Reflection.ReflectionTypeLoadException****: The classes in the module cannot be loaded.*
A: Check the Installation section.

Q: I received the following error message under Unity 3.5: *Internal compiler error. See the console log for more information. output was:*
Unhandled Exception: System.TypeLoadException: Could not load type ***'System.Runtime.Versioning.TargetFrameworkAttribute'*** *from assembly 'TcpClientImplementation'.*
A: Under Unity 3.5 you need to delete the Plugins/WP8 folder and restart your editor.

Q: What Stripping Level can be used with BestHTTP?
A: All stripping levels supported, except on iOS. There are some cases where you need a link.xml, for these cases a link.xml is included. You have to copy/merge this xml.