Documentation For BestHTTP Pro And Basic

Introduction

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 find an online demo here.

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

Installation

Please see the Upgrade Guide if you are upgrading to v1.9.x.

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("http://server.com/path"),
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("http://server.com/path"),
HTTPMethods.Post,
                            OnRequestFinished);
request.RawData =  Encoding.UTF8.GetBytes("Field Value");
request.Send();
```

For additional samples check out the Small Code-Samples section.

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

- ### Head

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

- ### Put

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

- **Delete**

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

- **Patch**

```
HTTPRequest request = new HTTPRequest(new Uri("http://server.com/path"),
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("http://yourserver.org/auth-path"), (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();
```

Download 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.

Upload Streaming[1]

You can set a Stream instance to upload to a HTTPRequest object through the *UploadStream* property. The plugin will use this stream to gather data to send to the server. When the upload finished and the *DisposeUploadStream* is true, then the plugin will call the Dispose() function on the stream.
If the stream's length is unknown, the *UseUploadStreamLength* property should be set to false. In this case, the plugin will send the data from the stream with chunked Transfer-Encoding.

```
var request = new HTTPRequest(new Uri(address), HTTPMethods.Post, OnUploadFinished);
request.UploadStream = new FileStream("File_To.Upload", FileMode.Open);
request.Send();
```

Upload Progress Tracking[2]

To track and display upload progress you can use the OnUploadProgress event of the HTTPRequest class. The OnUploadProgress can be used with RawData, forms(through AddField and AddBinaryData) and with UploadStream too.

```
var request = new HTTPRequest(new Uri(address), HTTPMethods.Post, OnFinished);
request.RawData =  Encoding.UTF8.GetBytes("Field Value");
request.OnUploadProgress = OnUploadProgress;
request.Send();

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

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

[1] First available in v1.7.4
[2] First available in v1.7.4

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 by 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.

Cookies can be added to a HTTPRequest by adding them to the Cookies list:

```
var request = new HTTPRequest(new Uri(address), OnFinished);
request.Cookies.Add(new Cookie("Name", "Value"));
request.Send();
```

These cookies will be merged with the server sent cookies. If *IsCookiesEnabled* is set to false on the request or in the HTTPManager, then only these user-set cookies will be sent.

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"));
```

You can set a global proxy too, so you don't have to set it to all request manually. See the Global Settings chapter.

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 / (float)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.

Request Priority

Request's priority can be change through the HTTPRequest's Priority property. Higher priority requests will be picked from the request queue sooner than lower priority requests.

```
var request = new HTTPRequest(new Uri("https://google.com"), ...);
request.Priority = -1;
request.Send();
```

Server Certificate Validation

Server sent certificates can be validated by implementing an ICertificateVerifyer interface and setting it to a HTTPRequest's CustomCertificateVerifyer:

```
using System;
using Org.BouncyCastle.Crypto.Tls;
using Org.BouncyCastle.Asn1.X509;

class CustomVerifier : ICertificateVerifyer
```

```
{
	public bool IsValid(Uri serverUri, X509CertificateStructure[] certs)
	{
		// TODO: Return false, if validation fails
		return true;
	}
}

var request = new HTTPRequest(new Uri("https://google.com"), ...);
request.CustomCertificateVerifyer = new CustomVerifier();
request.UseAlternateSSL = true;
request.Send();
```

Control Redirections

Redirection are handled automatically by the plugin, but sometimes we have to make changes before a new request is made to the uri that we redirected to. We can do these changes in the OnBeforeRedirection event handler of a HTTPRequest.
This event is called before the plugin will do a new request to the new uri. The return value of the function will control the redirection: if it's false the redirection is aborted.

This function is called on a thread other than the main Unity thread!

```
var request = new HTTPRequest(uri, HTTPMethods.Post);
request.AddField("field", "data");
request.OnBeforeRedirection += OnBeforeRedirect;
request.Send();

bool OnBeforeRedirect(HTTPRequest req, HTTPResponse resp, Uri redirectUri)
{
   if (req.MethodType == HTTPMethods.Post && resp.StatusCode == 302)
   {
        req.MethodType = HTTPMethods.Get;

        // Don't send more data than needed.
        // So we will delete our already processed form data.
        req.Clear();
   }
```

```
        return true;
    }
```

Statistics[3]

You can get some statistics about the underlying plugin using the HTTPManager.GetGeneralStatistics function:

```
GeneralStatistics stats = HTTPManager.GetGeneralStatistics(StatisticsQueryFlags.All);
Debug.Log(stats.ActiveConnections);
```

You can query for three type of statistics:

1. **Connections**: Connection based statistics will be returned. These are the following:
    - RequestsInQueue: Number of requests are waiting in the queue for a free connection.
    - Connections: The number of HTTPConnection instances that are tracked by the plugin. This is the sum of all of the following connections.
    - ActiveConnections: Number of active connections. These connections are currently processing a request.
    - FreeConnections: Number of free connections. These connections are finished with a request, and they are waiting for an another request or for recycling.
    - RecycledConnections: Number of recycled connections. These connections will be deleted as soon as possible.
1. **Cache**: Cache based statistics. These are the following:
    - CacheEntityCount: Number of cached responses.
    - CacheSize: Sum size of the cached responses.
1. **Cookie**: Cookie based statistics. These are the following:
    - CookieCount: Number of cookies in the Cookie Jar.
    - CookieJarSize: Sum size of the cookies in the Cookie Jar.

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.

[3] First available in v1.7.2.

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!
- **Proxy**: The global, default proxy for all HTTPRequests. The HTTPRequest's Proxy still can be changed per-request. Default value is null.
- **Logger**: An ILogger implementation to be able to control what informations will be logged about the plugin's internals, and how these will be logged.
- **DefaultCertificateVerifyer**: An ICertificateVerifyer implementation can be set to this property. All new requests created after this will use this verifier when a secure protocol is used and the request's UseAlternateSSL is true. An ICertificateVerifyer implementation can be used to implement server certificate validation.
- **UseAlternateSSLDefaultValue**: The default value of HTTPRequest's UseAlternateSSL can be changed through this property.

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 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 ex)
{
	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: " +
errorMsg));
}
```

- **OnErrorDesc** event: A more informative event then the OnError, as the later is called only with an Exception parameter. This event called after the OnError event, but it could provide a more detailed error report.

```
webSocket.OnErrorDesc += OnErrorDesc;

void OnErrorDesc(WebSocket ws, string error)
{
	Debug.Log("Error: " + error);
}
```

- **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.

Socket.IO[4]

The Socket.IO implementation uses features that the plugin already have. It will send *HTTPRequest*s to get the handshake data, sending and receiving packets when the polling transport is used with all of its features(cookies, connection reuse, etc.). And the *WebSocket* implementation is used for the WebSocket transport.

Brief feature list of this Socket.IO implementation:
- Easy to use and familiar api
- Compatible with the latest Socket.IO specification
- Seamless upgrade from polling transport to websocket transport
- Automatic reconnecting on disconnect
- Easy and efficient binary data sending and multiple ways of receiving
- Powerful tools to use it in an advanced mode(switch the default encoder, disable auto-decoding, etc.)

If you want to connect to a Socket.IO service you can do it using the BestHTTP.SocketIO.SocketManager class. First you have to create a SocketManager instance:

```
using System;
using BestHTTP;
using BestHTTP.SocketIO;

var manager = new SocketManager(new Uri("http://chat.socket.io/socket.io/"));
```

[4] First available in v1.7.0

The /socket.io/ path in the url is very important, by default the Socket.IO server will listen on this query. So don't forget to append it to your test url too!

Connecting to namespaces

By default the SocketManager will connect to the root("/") namespace while connecting to the server. You can access it through the SocketManager's Socket property:

```
Socket root = manager.Socket;
```

Non-default namespaces can be accessed through the GetSocket("/nspName") function or through the manager's indexer property:

```
Socket nsp = manager["/customNamespace"];
// the same as this methode:
Socket nsp = manager.GetSocket("/customNamespace");
```

First access to a namespace will start the internal connection process.

Subscribing and receiving events

You can subscribe to predefined and custom events. Predefined events are "connect", "connecting", "event", "disconnect", "reconnect", "reconnecting", "reconnect_attempt", "reconnect_failed", "error". Custom events are programmer defined events that your server will send to your client. You can subscribe to an event by calling a socket's On function:

```
manager.Socket.On("login", OnLogin);
manager.Socket.On("new message", OnNewMessage);
```

An event handler will look like this:

```
void OnLogin(Socket socket, Packet packet, params object[] args)
{
}
```

- The *socket* parameter will be the namespace-socket object that the server sent this event.
- The *packet* parameter contains the internal packet data of the event. The packet can be used to access binary data sent by the server, or to use a custom Json parser lib to decode the payload data. More on these later.
- The *args* parameter is a variable length array that contains the decoded objects from the packet's payload data. With the default Json encoder these parameters can be 'primitive' types(int, double, string) or list of objects(List<object>) or Dictionary<string, object> for objects.

  A message emitted on the server(node.js):

```
// send a message to the client
socket.emit('message', ‘MyNick’, ‘Msg to the client’);
```

can be caught by the client:

```
// subscribe to the "message" event
manager.Socket.On("message", OnMessage);

// event handler
void OnMessage(Socket socket, Packet packet, params object[] args)
{
	// args[0] is the nick of the sender
	// args[1] is the message
	Debug.Log(string.Format("Message from {0}: {1}", args[0], args[1]));
}
```

Predefined events

- **"connect"**: Sent when the namespace opens.
- **"connecting"**: Sent when the SocketManager start to connect to the socket.io server.
- **"event"**: Sent on custom (programmer defined) events.
- **"disconnect"**: Sent when the transport disconnects, SocketManager is closed, Socket is closed or when no Pong message received from the server in the given time specified in the handshake data.
- **"reconnect"**: Sent when the plugin successfully reconnected to the socket.io server.
- **"reconnecting"**: Sent when the plugin will try to reconnect to the socket.io server.
- **"reconnect_attempt"**: Sent when the plugin will try to reconnect to the socket.io server.
- **"reconnect_failed"**: Sent when a reconnect attempt fails to connect to the server and the ReconnectAttempt reaches the options’ ReconnectionAttempts’ value.
- **"error"**: Sent on server or internal plugin errors. The event’s only argument will be a `BestHTTP.SocketIO.Error` object.

Other event-related functions:

- **Once**: you can subscribe to an event that will be called only once.

```
// The event handler will be called only once
manager.Socket.Once("connect", OnConnected);
```

- **Off**: you can remove all event subscription, or just only one.

```
// Removes all event-handlers
manager.Socket.Off();

// Removes event-handlers from the "connect" event
```

```
manager.Socket.Off("connect");


// Removes the OnConnected event-handler from the "connect" event
manager.Socket.Off("connect", OnConnected);
```

Sending events

You can send an event with the Emit function. You have to pass the event name as the first parameter and optionally other parameters. These will be encoded to json and will be sent to the server. Optionally you can set a callback function that will be called when the server processes the event(you have to set up the server code properly to be able to send back a callback function. See the Socket.IO server side documentation for more information).

```
// Send a custom event to the server with two parameters
manager.Socket.Emit("message", "userName", "message");

// Send an event and define a callback function that will be called as an
// acknowledgement of this event
manager.Socket.Emit("custom event", OnAckCallback, "param 1", "param 2");

void OnAckCallback(Socket socket, Packet originalPacket, params object[] args)
{
	Debug.Log("OnAckCallback!");
}
```

Sending acknowledgement to the server

You can send back an acknowledgement to the server by calling the socket's EmitAck function. You have to pass the original packet and any optional data:

```
manager["/customNamespace"].On("customEvent", (socket, packet, args) =>
	{
		socket.EmitAck(packet, "Event", "Received", "Successfully");
	});
```

You can keep a reference to the packet, and call the EmitAck from somewhere else.

Sending binary data

There are two ways of sending binary(byte[]) data.

1. By passing to the Emit function the plugin will scan the parameters and if it founds one, it will convert it to a binary attachment(as introduced in Socket.IO 1.0). This is the most efficient way, because it will not convert the byte array to a Base64 encoded string on client side, and back to binary on server side.

```
byte[] data = new byte[10];
//...
manager.Socket.Emit("eventWithBinary", "textual param", data);
```

1. If the binary data is embedded in an object as a field or property the Json encoder must support the conversion. The default Json encoder can't convert the embedded binary data to Json, you have to use a more advanced Json parser library (like "JSON .NET For Unity" - http://u3d.as/5q2)

Receiving binary data

In the Socket.IO server when binary data sent to the client it will replace the data with a Json object(*{'_placeholder':true,'num':xyz}*) and will send the binary data in an other packet. On client side these packets will be collected and will be merged into one packet. The binary data will be in the packet's Attachments property.
Here you will have some options too to use this packet:

1. In your event-handler you can access all binary data through the packet's Attachments property.

```
Socket.On("frame", OnFrame);

void OnFrame(Socket socket, Packet packet, params object[] args)
{
	texture.LoadImage(packet.Attachments[0]);
}
```

1. The second option is almost the same as the previous, with a little improvement: we will not decode the sent Json string to c# objects. We can do it because we know that the server sent only the binary data, no other information came with this event. So we will let the plugin know that do not decode the payload:

```
// Subscribe to the "frame" event, and set the autoDecodePayload flag to false
Socket.On("frame", OnFrame, /*autoDecodePayload:*/ false);

void OnFrame(Socket socket, Packet packet, params object[] args)
{
	// Use the Attachments property as before
	texture.LoadImage(packet.Attachments[0]);
}
```

The *autoDecodePayload* parameter is true by default.

1. We can replace back the "*{'_placeholder':true,'num':xyz}*" string to the index of the attachment in the Attachments list.

```
Socket.On("frame", OnFrame, /*autoDecodePayload:*/ false);
```

```
void OnFrame(Socket socket, Packet packet, params object[] args)
{
	// Replace the Json object with the index
	packet.ReconstructAttachmentAsIndex();

	// now, decode the Payload to an object[]
	args = packet.Decode(socket.Manager.Encoder);

	// args now contains only an index number (probably 0)
	byte[] data = packet.Attachments[Convert.ToInt32(args[0])];

	texture.LoadImage(data);
}
```

1. We can replace the "*{'_placeholder':true,'num':xyz}*" string with the binary data from the Attachments converted to a Base64 encoded string. Advanced Json parsers can convert it to byte arrays when they have to set it to an object's field or property.

```
Socket.On("frame", OnFrame, /*autoDecodePayload:*/ false);

void OnFrame(Socket socket, Packet packet, params object[] args)
{
	// Replace the Json object with the Base64 encoded string
	packet.ReconstructAttachmentAsBase64();

	// now, decode the Payload to an object[]
	args = packet.Decode(socket.Manager.Encoder);

	// args now contains a Base64 encoded string
	byte[] data = Convert.FromBase64String(args[0] as string);

	texture.LoadImage(data);
}
```

Set the default Json encoder

You can change the default Json encoder by setting the SocketManager's static DefaultEncoder to a new encoder. After this step all newly created SocketManager will use this encoder.
Or you can set directly the SocketManager object's Encoder property to an encoder.

Writing a custom Json encoder

If you want to change the default Json encoder for various reasons, first you have to write a new one. To do so, you have to write a new class that implements the `IJsonEncoder` from the

BestHTTP.SocketIO.JsonEncoders namespace.
The stripped IJsonEncoder is very tiny, you have to implement only two functions:

```
public interface IJsonEncoder
{
	List<object> Decode(string json);
	string Encode(List<object> obj);
}
```

The Decode function must decode the given json string to a list of objects. Because of the nature of the Socket.IO protocol, the sent json is an array and the first element is the event's name.

The Encode function is used to encode the data that the client wants to send to the server. The structure of this list is the same as with the Decode: the first element of the list is the event's name, and any other elements are the user sent arguments.

And here comes a complete example using the LitJson library from the examples folder:

```
using LitJson;
public sealed class LitJsonEncoder : IJsonEncoder
{
	public List<object> Decode(string json)
	{
		JsonReader reader = new JsonReader(json);
		return JsonMapper.ToObject<List<object>>(reader);
	}

	public string Encode(List<object> obj)
	{
		JsonWriter writer = new JsonWriter();
		JsonMapper.ToJson(obj, writer);
		return writer.ToString();
	}
}
```

AutoDecodePayload

Already talked about AutoDecodePayload in "*Receiving binary data*", however you can set this value not just per-event, but per-socket too. The socket has an AutoDecodePayload property that used as the default value of event subscription. Its default value is true - all payload decoded and dispatched to the event subscriber. If you set to false no decoding will be done by the plugin, you will have to do it by yourself.

You don't want to cast the args every time: Sure! You can set the AutoDecodePayload on the Socket object, and you can use your favorite Json parser to decode the Packet's Payload to a strongly typed object. However keep in mind that the payload will contain the event's name and it's a json array. A sample

payload would look like this: `"[‘eventName’, {‘field’: ‘stringValue’}, {‘field’: 1.0}]"`.

Error handling

An `"error"` event emitted when a server side or client side error occurs. The first parameter of the event will be an Error object. This will contain an error code in the Code property and a string message in the Message property. The ToString() function int this class has been overridden, you can use this function to write out its contents.

```
Socket.On(SocketIOEventTypes.Error, OnError);

void OnError(Socket socket, Packet packet, params object[] args)
{
	Error error = args[0] as Error;

	switch (error.Code)
	{
		case SocketIOErrors.User:
			Debug.Log("Exception in an event handler!");
			break;
		case SocketIOErrors.Internal:
			Debug.Log("Internal error!");
			break;
		default:
			Debug.Log("Server error!");
			break;
	}

	Debug.Log(error.ToString());
}
```

Available options in the SocketOptions class

You can pass a SocketOptions instance to the SocketManager’s constructor. You can change the following options:

- Reconnection: Whether to reconnect automatically after a disconnect. Its default value is true.
- ReconnectionAttempts: Number of attempts before giving up. Its default value is Int.MaxValue.
- ReconnectionDelay: How long to initially wait before attempting a new reconnection. Affected by +/- RandomizationFactor. For example the default initial delay will be between 500ms to 1500ms. Its default value is 10000ms.
- ReconnectionDelayMax: Maximum amount of time to wait between reconnections. Each attempt increases the reconnection delay along with a randomization as above. Its default value is 5000ms.
- RandomizationFactor: It can be used to control the ReconnectionDelay range. Its default value is 0.5 and can be set between the 0..1 values inclusive.

- Timeout: Connection timeout before a "connect_error" and "connect_timeout" events are emitted. It's not the underlying tcp socket's connection timeout, it's for the socket.io protocol. Its default value is is 20000ms.
- AutoConnect: By setting this false, you have to call SocketManager's Open() whenever you decide it's appropriate.

When you create a new SocketOptions object its properties are set to theirs default values.

SignalR[5]

The SignalR implementation like the Socket.IO uses the plugin's base features. HTTPRequests and WebSockets are used to connect and communicate leveraging from connection pooling. Cookies are sent with the requests, and the logger is used to log informations about the protocol and errors.

Brief list of features of the SignalR implementation:

- Compatible with the latest SignalR server implementation
- Easy to use API
- Transport fallback
- Reconnect logic
- All Hub features are supported

The Connection class

The Connection class from the BestHTTP.SignalR namespace manages an abstract connection to the SignalR server. Connecting to a SignalR server start with the creation of a Connection object.
This class will keep track of the current state of the protocol and will fire events.

You can create a Connection object multiple ways:

```
using BestHTTP.SignalR;

Uri uri = new Uri("http://besthttpsignalr.azurewebsites.net/raw-connection/");
```

1. Create the connection, without Hubs by passing only the server's uri to the constructor.

```
Connection signalRConnection = new Connection(uri);
```

2. Create the connection, with Hubs by passing the hub names to the constructor too.

[5] Available in v1.8

```
Connection signalRConnection = new Connection(uri, "hub1", "hub2", "hubN");
```

3. Create the connection, with Hubs by passing Hub objects to the constructor.

```
Hub hub1 = new Hub("hub1");
Hub hub2 = new Hub("hub2");
Hub hubN = new Hub("hubN");
Connection signalRConnection = new Connection(uri, hub1, hub2, hubN);
```

You can't mix options 2 and 3.

After we created the Connection, we can start to connect to the server by calling the Open() function on it:

```
signalRConnection.Open();
```

Handling general events

The Connection class will allow you to subscribe to multiple events. These events are the following:

- **OnConnected**: This event is fired when the connection class successfully connected, and the SignalR protocol is up for communication.

```
signalRConnection.OnConnected += (con) =>
      Debug.Log("Connected to the SignalR server!");
```

- **OnClosed**: This event is fired when the SignalR protocol is closed, and no more further messages are sent or received.

```
signalRConnection.OnClosed += (con) =>
      Debug.Log("Connection Closed");
```

- **OnError**: Called when an error occurs. If the connection is already open, the plugin will try to reconnect, otherwise the connection will be closed.

```
signalRConnection.OnError += (conn, err) =>
      Debug.Log("Error: " + err);
```

- **OnReconnecting**: This event is fired, when a reconnect attempt is started. After this event an OnError or an OnReconnected event is called. Multiple OnReconnecting-OnError event pairs can be fired before an OnReconnected/OnClosed event, because the plugin will try to reconnect multiple

times in a given time.

```
signalRConnection.OnReconnecting += (con) =>
	Debug.Log("Reconnecting");
```

- **OnReconnected**: Fired when a reconnect attempt was successful.

```
signalRConnection.OnReconnecting += (con) =>
	Debug.Log("Reconnected");
```

- **OnStateChnaged**: Fired when the connection's State changed. The event handler will receive both the old state and the new state.

```
signalRConnection.OnStateChanged += (conn, oldState, newState) =>
	Debug.Log(string.Format("State Changed {0} -> {1}", oldState, newState));
```

- **OnNonHubMessage**: Fired when the server send a non-hub message to the client. The client should know what types of messages are expected from the server, and should cast the received object accordingly.

```
signalRConnection.OnNonHubMessage += (con, data) =>
	Debug.Log("Message from server: " + data.ToString());
```

- **RequestPreparator**: This delegate is called for every HTTPRequest that made and will be sent to the server. It can be used to further customize the requests.

```
signalRConnection.RequestPreparator = (con, req, type) =>
	req.Timeout = TimeSpan.FromSeconds(30);
```

Sending non-Hub messages

Sending non-hub messages to the server is easy as calling a function on the connection object:

```
signalRConnection.Send(new { Type = "Broadcast", Value = "Hello SignalR World!" });
```

This function will encode the given object to a Json string using the Connection's JsonEncoder, and sends it to the server.

Already encoded Json strings can be sent using the SendJson function:

```
signalRConnection.SendJson("{ Type: 'Broadcast', Value: 'Hello SignalR World!' }");
```

Hubs

In order to define methods on the client that a Hub can call from the server, and to invoke methods on a Hub at the server, Hubs must be added to the Connection object. This can be done by adding the hub names or hub instances to the Connection constructor, demonstrated in the *Connection Class* section.

Accessing hubs

Hub instances can be accessed through the Connection object by index, or by name.

```
Hub hub = signalRConnection[0];
Hub hub = signalRConnection["hubName"];
```

Register server callable methods

To handle server callable method calls, we have to call the On function of a hub:

```
// Register method implementation
signalRConnection["hubName"].On("joined", Joined);

// "Joined" method implementation on the client
void Joined(Hub hub, MethodCallMessage msg)
{
  Debug.log(string.Format("{0} joined at {1}",
       msg.Arguments[0], msg.Arguments[1]));
}
```

The MethodCallMessage is a server sent object that contains the following properties:

- **Hub**: A string containing the hub name that the method have to call on.
- **Method**: A string that contains the method name.
- **Arguments**: An array of objects that contains the arguments of the method call. It can be an empty array.
- **State**: A dictionary containing additional custom data.

The plugin will use the Hub and Method properties to route the message to the right hub and event handler. The function that handles the method call have to use only the Arguments and State properties.

Call server-side methods

Calling server-side methods can be done by call a Hub's Call function. The call function overloaded to be able to fulfill every needs. The Call functions are non-blocking functions, they will **not** block until the server sends back any message about the call.

The overloads are the following:

- **Call(string method, params object[] args)**: This can be used to call a server-side function in a fire-and-forget style. We will not receive back any messages about the method call's success or failure. This function can be called without any 'args' arguments, to call a parameterless method.

```
// Call a server-side function without any parameters
signalRConnection["hubName"].Call("Ping");

// Call a server-side function with two string parameters: "param1" and "param2"
signalRConnection["hubName"].Call("Message", "param1", "param2");
```

- **Call(string method, OnMethodResultDelegate onResult, params object[] args)**: This function can be used as the previous one, but a function can be passed as the second parameter that will be called when the server-side function successfully invoked.

```
signalRConnection["hubName"].Call("GetValue", OnGetValueDone);

void OnGetValueDone(Hub hub, ClientMessage originalMessage, ResultMessage result)
{
  Debug.Log("GetValue executed on the server. Return value of the function:" +
       result.ReturnValue.ToString());
}
```

This callback function receives the Hub that called this function, the original ClientMessage message that sent to the server and the ResultMessage instance sent by the server as a result of the method call. A ResultMessage object contains a ReturnValue and a State properties.

If the method's return type is void, the ReturnValue is null.

- **Call(string method, OnMethodResultDelegate onResult, OnMethodFailedDelegate onError, params object[] args)**: This function can be used to specifi a callback that will be called when the method fails to run on the server. Failures can be happen because of a non-found method, wrong parameters, or unhandled exceptions in the method call.

```
signalRConnection["hubName"].Call("GetValue", OnGetValueDone, OnGetValueFailed);
```

```
void OnGetValueFailed(Hub hub, ClientMessage originalMessage,
                              FailureMessage error)
{
  Debug.Log("GetValue failed. Error message from the server: " +
       error.ErrorMessage);
}
```

A FailureMessage contains the following properties:

- **IsHubError**: True if it is a Hub error.
- **ErrorMessage**: A brief message about the error itself.
- **StackTrace**: If detailed error reporting is turned on on the server then it contains the stack trace of the error.
- **AdditionalData**: If it's not null, then it contains additional informations about the error.

- **Call(string method, OnMethodResultDelegate onResult, OnMethodFailedDelegate onError, OnMethodProgressDelegate onProgress, params object[] args)**: This function can be used to add an additional progress message handler to the server-side method call. For long running jobs the server can send progress messages to the client.

```
signalRConnection["hubName"].Call("GetValue", OnGetValueDone, OnGetValueFailed,
OnGetValueProgress);

void OnGetValueProgress(Hub hub, ClientMessage originalMessage,
                                 ProgressMessage progress)
{
  Debug.Log(string.Format("GetValue progressed: {0}%", progress.Progress));
}
```

When a ResultMessage or FailureMessage received by the plugin, it will not serve the ProgressMessages that came after these messages.

Using the Hub class as a base class to inherit from

The Hub class can be used as a base class to encapsulate hub functionality.

```
class SampleHub : Hub
{
 // Default constructor. Every hubs have to have a valid name.
 public SampleHub()
```

```
    :base("SampleHub")
{
      // Register a server-callable function
      base.On("ClientFunction", ClientFunctionImplementation);
}

// Private function to implement server-callable function
private void ClientFunctionImplementation(Hub hub, MethodCallMessage msg)
{
  // TODO: implement
}

// Wrapper function to call a server-side function.
public void ServerFunction(string argument)
{
      base.Call("ServerFunction", argument);
}
}
```

This SampleHub can be instantiated and passed to the Connection's constructor:

```
SampleHub sampleHub = new SampleHub();
Connection signalRConnection = new Connection(Uri, sampleHub);
```

Authentication

The Connection class has an AuthenticationProvider property that can be set to an object that implements the IAuthenticationProvider interface.
The implementor has to implement the following property and functions:

- **bool IsPreAuthRequired**:Property that returns true, if the authentication must run before any request is made to the server by the Connection class. *Examples*: a cookie authenticator must return false, as it has to send user credentials and receive back a cookie that must sent with the requests.
- **StartAuthentication**: A function that required only if the *IsPreAuthRequired* is true. Otherwise it doesn't called.
- **PrepareRequest**: A function that called with a request and a request type enum. This function can be used to prepare requests before they are sent to the server.
- **OnAuthenticationSucceded**: An event that must be called when the IsPreAuthRequired is true and the authentication process succeeded.

- **OnAuthenticationFailed**: An event that must be called when the IsPreAuthRequired is true and the authentication process failed.

A very simple Header-based authenticator would look like this:

```
class HeaderAuthenticator : IAuthenticationProvider
{
   public string User { get; private set; }
   public string Roles { get; private set; }

   // No pre-auth step required for this type of authentication
   public bool IsPreAuthRequired { get { return false; } }

   // Not used event as IsPreAuthRequired is false
   public event OnAuthenticationSuccededDelegate OnAuthenticationSucceded;

   // Not used event as IsPreAuthRequired is false
   public event OnAuthenticationFailedDelegate OnAuthenticationFailed;

   // Constructor to initialise the authenticator with username and roles.
   public HeaderAuthenticator(string  user, string roles)
   {
        this.User = user;
        this.Roles = roles;
   }

   // Not used as IsPreAuthRequired is false
   public void StartAuthentication()
   { }

   // Prepares the request by adding two headers to it
   public void PrepareRequest(BestHTTP.HTTPRequest request, RequestTypes type)
   {
        request.SetHeader("username", this.User);
        request.SetHeader("roles", this.Roles);
   }
}
```

Writing custom Json encoders

Like for the Socket.IO's Manager class, the SignalR's Connection class has a JsonEncoder property, and

the static Connection.DefaultEncoder can be set too.
A JsonEncoder must implement the IJsonEncoder interface from the BestHTTP.SignalR.JsonEncoders namespace.
The package contains a sample LitJsonEncoder, that also used by some samples too.

Server-Sent Events (EventSource)[6]

The Server-Sent Events is a one-way string-based protocol. Data is came from the server, and there are no option to send anything to the server. It's implemented using the latest draft.
While the protocol's name is Server-Sent Events, the class itself is named EventSource.

When an error occurs, the plugin will try to reconnect once sending the LastEventId to let the server send any buffered message that we should receive.

The EventSource class

The EventSource class is located in the BestHTTP.ServerSentEvents namespace:

```
using BestHTTP.ServerSentEvents;

var sse = new EventSource(new Uri("http://server.com"));
```

Properties

These are the publicly exposed properties of the EventSource class:

- **Uri**: This is the endpoint where the protocol tries to connect to. It's set through the constructor.
- **State**: The current state of the EventSource object.
- **ReconnectionTime**: How many time to wait to try to do a reconnect attempt. It's default value is 2 sec.
- **LastEventId**: The last received event's id. It will be null, if no event id received at all.
- **InternalRequest**: The internal HTTPRequest object that will be sent out in the Open function.

Events

- **OnOpen**: It's called when the protocol is successfully upgraded.

```
eventSource.OnOpen += OnEventSourceOpened;
```

[6] First available in v1.8.1

```
void OnEventSourceOpened(EventSource source)
{
	Debug.log("EventSource Opened!");
}
```

- **OnMessage**: It's called when the client receives a new message from the server. This function will receive a Message object that contains the payload of the message in the Data property. This event is called every time the client receives a messages, even when the message has a valid Event name, and we assigned an event handler to this event!

```
eventSource.OnMessage += OnEventSourceMessage;

void OnEventSourceMessage(EventSource source, Message msg)
{
	Debug.log("Message: " + msg.Data);
}
```

- **OnError**: Called when an error encountered while connecting to the server, or while processing the data stream.

```
eventSource.OnError += OnEventSourceError;

void OnEventSourceError(EventSource source, string error)
{
	Debug.log("Error: " + error);
}
```

- **OnRetry**: This function is called before the plugin will try to reconnect to the server. If the function returns false, no attempt will be made and the EventSource will be closed.

```
eventSource.OnRetry += OnEventSourceRetry;

bool OnEventSourceRetry(EventSource source)
{
	// disable retry
	return false;
}
```

- **OnClosed**: This event will be called when the EventSource closed.

```
eventSource.OnClosed += OnEventSourceClosed;

void OnEventSourceClosed(EventSource source)
{
	Debug.log("EventSource Closed!");
}
```

- **OnStateChanged**: Called every time when the State property changes.

```
eventSource.OnStateChanged += OnEventSourceStateChanged;

void OnEventSourceStateChanged(EventSource source, States oldState, States newState)
{
	Debug.log(string.Format("State Changed {0} => {1}", oldSate, newState)));
}
```

Functions

These are the public functions of the EventSource object.

- **Open**: Calling this function the plugin will start to connect to the server and upgrade to the Server-Sent Events protocol.

```
EventSource eventSource = new EventSource(new Uri("http://server.com"));
eventSource.Open();
```

- **On**: Using this function clients can subscribe to events.

```
eventSource.On("userLogon", OnUserLoggedIn);

void OnUserLoggedIn(EventSource source, Message msg)
{
	Debug.log(msg.Data);
}
```

- **Off**: It can be used to unsubscribe from an event.

```
eventSource.Off("userLogon");
```

- **Close**: This function will start to close the EventSource object.

```
eventSource.Close();
```

The Message class

The Message class is a logical unit that contains all information that a server can send.

Properties

- **Id**: Id of the sent event. Can be null, if no id sent. It's used by the plugin.
- **Event**: Name of the event. Can be null, if no event name sent.
- **Data**: The actual payload of the message.
- **Retry**: A server sent time that the plugin should wait before a reconnect attempt. It's used by the plugin.

Logging[7]

To be able to dump out some important - and sometimes less important - functioning of the plugin a logger interface and implementation is introduced in v1.7.0. The default logger can be accessed through the HTTPManager.Logger property. The default loglevel is Warning for debug builds and Error for for others. The default logger implementation uses Unity's Debug.Log/LogWarning/LogError functions.
A new logger can be written by implementing the ILogger interface from the BestHTTP.Logger namespace.

Small Code-Samples

- **Upload a picture using forms**

```
var request = new HTTPRequest(new Uri("http://server.com"),
                              HTTPMethods.Post,
                              onFinished);
request.AddBinaryData("image", texture.EncodeToPNG(), "image.png");
request.Send();
```

- **Upload a picture without forms, sending only the raw data**

```
var request = new HTTPRequest(new Uri("http://server.com"),
                              HTTPMethods.Post,
                              onFinished);
request.Raw = texture.EncodeToPNG();
request.Send();
```

- **Add custom header**

```
var request = new HTTPRequest(new Uri("http://server.com"), HTTPMethods.Post,
```

[7] First available in v1.7.0

```
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://serveroflargefile.net/path"),
                              (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
  var 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)
    {
          string path = Path.Combine(Application.temporaryCachePath, saveTo);
          using (FileStream fs = new FileStream(path, 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
- Upload streams

How to disable features (Pro only)

Starting in v1.9.0 there are many defines that can be used to disable a feature. These defines can be combined, even all can be set. Disabled features will not compile, so build size can be reduced by disabling unused features.
These defines are the following:

- BESTHTTP_DISABLE_COOKIES: With this define all cookie releated code can be disabled. No cookie parsing, saving and sending will occur.
- BESTHTTP_DISABLE_CACHING: With this define all cache releated code can be disabled. No caching, or cache validation will be done.
- BESTHTTP_DISABLE_SERVERSENT_EVENTS: Server-Sent Events can be disabled with this. SignalR will not fallback to this.
- BESTHTTP_DISABLE_WEBSOCKET: Websocket can be disable with this. SignalR and Socket.IO will not use this protocol.
- BESTHTTP_DISABLE_SIGNALR: The entire SignalR implementation will be disabled.
- BESTHTTP_DISABLE_SOCKETIO: The entire Socket.IO implementation will be disabled.
- BESTHTTP_DISABLE_ALTERNATE_SSL: If you are not using HTTPS or WSS for WebSocket, or you are happy with the default implementation, you can disable the alretnate ssl handler.
- BESTHTTP_DISABLE_UNITY_FORM: You can remove the dependency on Unity's WWWForm.

Supported Platforms

- iOS (with Unity iOS Pro licence when used in Unity 4.6)
- Android (with Unity Android Pro licence when used in Unity 4.6)

- Windows Phone 8.1, 10
- WinRT/Metro/Windows Store Apps 8.1, 10
- Windows, Linux and Mac Standalone
- Web Player and Samsung Smart TV
    - 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 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 8.1(and greater) and on WinRT(Windows Store Apps) a secure, Tls 1.2 protocol will handle the connection.

iOS

- No platform specific bugs or limitations are known.

Android

- No platform specific bugs or limitations are known.

Windows Phone

- The native https handler can't connect to custom ports. If you want to connect to a custom port using https you have to use the plugin's alternate https handler by setting the request's UseAlternateSSL to true.

Windows Store Apps

- The native https handler can't connect to custom ports. If you want to connect to a custom port using https you have to use the plugin's alternate https handler by setting the request's UseAlternateSSL to true.

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.

Samsung Smart TV

- Only *Best HTTP (Pro)* version can be used.
- Caching and Cookie persistence is not supported.

WebGL

- In WebGL builds the plugin will use the underlying browser's XmlHTTPRequest implementation. Because of this, there are features that isn't available:
  - Cookies
  - Caching
  - Download and upload streaming
  - Proxy
  - Server Certificate Validation
  - Redirection Control

  These limitations may change in future update of the plugin.
- If you make requests to another server that your WebGL build is loaded from the remote server must set some headers to enable the requests. For more details you can start reading on the Wikipedia page of CORS.

Upgrade guide

Upgrading to v1.9.x: In v1.9.0 all DLL dependencies are removed, and some of the files from the source too. So a clean install is advised: Before you install v1.9.x you have to remove the /Assets/Best HTTP (Pro|Basic)/ folder and the BestHTTP.dll (it's in the Basic version only) and TcpClientImplementation.dll from the /Assets/Plugins/, /Assets/Plugins/Metro and /Assets/Plugins/WP8 folders.