SGCWEBSOCKETS · TECHNICAL DOCUMENT

HTTP.sys API Server

TsgcWebSocketServer_HTTPAPI — kernel-mode (http.sys) HTTP and WebSocket server for the highest possible throughput on Windows.

eSeGeCe

WWW.ESEGECE.COM

Overview

The HTTP Server API enables applications to communicate over HTTP without using Microsoft Internet Information Server (IIS).

At a glance

COMPONENT CLASS

`TsgcWebSocketServer_HTTPAPI`

STANDARDS / SPEC

HTTP Server API — Microsoft Learn

TRANSPORTS

TCP, TLS, HTTP, HTTPS

PLATFORMS

Windows, macOS, Linux, iOS, Android

FRAMEWORKS

VCL, FireMonkey, Lazarus / FPC

EDITION

Standard / Professional / Enterprise

Features

- Native Delphi implementation with full ANSI/Unicode support.

Technical specification

Standards & specs	HTTP Server API — Microsoft Learn · WebSocket Protocol — RFC 6455
Component class	`TsgcWebSocketServer_HTTPAPI` (unit `sgcWebSocket_Server_HTTPAPI`)
Frameworks	VCL, FireMonkey, Lazarus / FPC
Platforms	Windows, macOS, Linux, iOS, Android

Main properties

The principal published / public properties used to configure and drive the component. Consult the online help for the full list.

`APIKeyManager`	Optional API-key manager component used to validate incoming API keys before accepting a connection.
`Authentication`	Enables and configures user/password authentication for incoming WebSocket and HTTP requests.
`Active`	Starts or stops the HTTP.sys listener, registering the configured URL with the Windows kernel driver.
`Host`	Hostname or IP address used to build the URL registered with the http.sys driver.
`Port`	TCP port that, combined with Host, forms the URL the http.sys driver reserves for the server.
`BindingOptions`	Fine-tunes how URL reservations and SSL certificates are registered with the http.sys driver at start-up.
`SSLOptions`	Identifies the Windows Certificate Store entry that http.sys binds to the listener when SSL is enabled.
`SecurityOptions`	Defines admission rules such as allowed origins for browser WebSocket handshakes.
`HeartBeat`	Sends periodic ping frames to keep idle WebSocket connections alive and detect dead peers.

`WatchDog`	Automatically restarts the server after an unexpected shutdown or listener failure.

Main methods

The principal public methods exposed by the component.

`Start()`	Starts the HTTP.sys server from a secondary thread so the calling thread is not blocked while URL groups and bindings are registered.
`Stop()`	Stops the HTTP.sys server from a secondary thread so the calling thread is not blocked while connections are closed and the request queue is released.
`ReStart()`	Stops and then restarts the HTTP.sys server from a secondary thread, useful after changing bindings, ports or SSL certificates at runtime.
`DisconnectAll()`	Disconnects every active WebSocket connection while keeping the HTTP.sys server listening for new connections.
`WriteData()`	Sends a message to a single client identified by its connection GUID.
`Ping()`	Sends a WebSocket ping frame to every connected client.
`ShareList()`	Acquires a shared (read-only) lock on the internal connection list and returns it for concurrent enumeration.
`UnShareList()`	Releases the shared (read-only) lock previously acquired by ShareList.
`Broadcast()`	Sends the same message to all connected clients, optionally filtered by channel, protocol, or connection GUID list.

Public events

The component exposes the following published events; consult the online help for full event-handler signatures.

`OnAfterForwardHTTP`	TsgcWebSocketServer_HTTPAPI › Events › OnAfterForwardHTTP
`OnAsynchronous`	Fires when an asynchronous send operation initiated by the HTTP API server has completed.
`OnAuthentication`	Fires when authentication is enabled so the application can check user and password and accept or reject the connection.

`OnBeforeBinding`	Fires before the server binds to the configured URL, so the list of bindings can be inspected or customized.
`OnBeforeForwardHTTP`	TsgcWebSocketServer_HTTPAPI › Events › OnBeforeForwardHTTP
`OnBeforeHeartBeat`	Fires before each HeartBeat ping so the application can implement a custom keep-alive.
`OnBinary`	Fires every time a client sends a binary message and it is received by the server.
`OnConnect`	Fires every time a WebSocket connection is established with a client.
`OnDisconnect`	Fires every time a WebSocket connection with a client is dropped.
`OnError`	Fires whenever a WebSocket protocol error occurs, such as a mal-formed handshake.
`OnException`	Fires whenever an unhandled exception is raised while processing a client connection.
`OnFragmented`	Fires when a fragment of a message is received (only when Options.FragmentedMessages is frgAll or frgOnlyFragmented).
`OnHTTPRequest`	Fires when the server receives an HTTP request so the application can build the response.
`OnHTTPUploadAfterSaveFile`	TsgcWebSocketServer_HTTPAPI › Events › OnHTTPUploadAfterSaveFile
`OnHTTPUploadBeforeCreatePostStream`	TsgcWebSocketServer_HTTPAPI › Events › OnHTTPUploadBeforeCreatePostStream
`OnHTTPUploadBeforeSaveFile`	TsgcWebSocketServer_HTTPAPI › Events › OnHTTPUploadBeforeSaveFile
`OnHTTPUploadReadInput`	TsgcWebSocketServer_HTTPAPI › Events › OnHTTPUploadReadInput
`OnHandshake`	Fires after the handshake is evaluated on the server side and before the response is sent.

`OnMessage`	Fires every time a client sends a text message and it is received by the server.
`OnShutdown`	Fires after the HTTP API server has stopped and no more requests are accepted.
`OnStartup`	Fires after the HTTP API server has started and is ready to accept connections.
`OnTCPConnect`	Fires after a client connects at TCP level and before the WebSocket handshake, so the connection can be accepted or rejected.
`OnUnknownProtocol`	Not currently supported by the HTTP API server; declared for API compatibility with TsgcWebSocketServer.

Quick Start

Drop the component on a form, configure the properties below and activate it. The snippet that follows shows the typical **TsgcWebSocketServer_HTTPAPI | Custom Headers** configuration sourced from the online help.

> **About this scenario.** You can customize the response of HTTP.SYS server using the CustomHeaders property of response object.

Delphi (VCL / FireMonkey)

```
procedure OnHTTPRequest(aConnection: TsgcWSConnection_HTTPAPI; const aRequestInfo: THttpServerRe
 var aResponseInfo: THttpServerResponse);
begin
aResponseInfo.ResponseNo := 200;
aResponseInfo.CustomHeaders := 'Access-Control-Allow-Origin: *' + #13#10 + 'Acces-Control-Allow-
   'GET, POST, OPTIONS, PUT, PATCH, DELETE';
end;
```

C++ Builder

```
private void OnHTTPRequest(TsgcWSConnection_HTTPAPI *aConnection, const THttpServerRequest *aReq
 ref THttpServerResponse *aResponseInfo)
{
aResponseInfo->ResponseNo = 200;
aResponseInfo->CustomHeaders = "Access-Control-Allow-Origin: *\r\n" + "Acces-Control-Allow-Metho
   "GET, POST, OPTIONS, PUT, PATCH, DELETE";
}
```

.NET (C#)

```
private void OnHTTPRequest(TsgcWSConnection_HTTPAPI aConnection, const THttpServerRequest aReque
 ref THttpServerResponse aResponseInfo)
{
aResponseInfo.ResponseNo = 200;
aResponseInfo.CustomHeaders = "Access-Control-Allow-Origin: *" + Environment.NewLine + "Acces-Co
    "GET, POST, OPTIONS, PUT, PATCH, DELETE";
}
```

Common scenarios

The following scenarios are lifted verbatim from the online help. Each shows the configuration and method calls needed to drive the component through a specific real-world flow.

1 · TsgcWebSocketServer_HTTPAPI | Send Text Response

Use the event OnHTTPRequest to handle the HTTP Requests.

Delphi (VCL / FireMonkey)

```
procedure OnHTTPRequest(aConnection: TsgcWSConnection_HTTPAPI;
        const aRequestInfo: THttpServerRequest;
        var aResponseInfo: THttpServerResponse);
begin
  if aRequestInfo.Method = 'GET' then
  begin
    if aRequestInfo.Document = '/test.html' then
        begin
          aResponseInfo.ResponseNo := 200;
          aResponseInfo.ContentText := 'OK';
          aResponseInfo.ContentType := 'text/html; charset=UTF-8';
        end
        else
          aResponseInfo.ResponseNo := 404;
  end
  else
    aResponseInfo.ResponseNo := 500;
end;
```

C++ Builder

```
void __fastcall OnHTTPRequest(TsgcWSConnection_HTTPAPI *aConnection,
        const THttpServerRequest *aRequestInfo,
        THttpServerResponse *aResponseInfo)
{
  if (aRequestInfo->Method == "GET")
  {
    if (aRequestInfo->Document == "/test.html")
        {
          aResponseInfo->ResponseNo = 200;
          aResponseInfo->ContentText = "OK";
          aResponseInfo->ContentType = "text/html; charset=UTF-8";
        }
        else
        {
          aResponseInfo->ResponseNo = 404;
        }
  }
  else
  {
    aResponseInfo->ResponseNo = 500;
  }
}
```

.NET (C#)

```
void OnHTTPRequest(TsgcWSConnection_HTTPAPI aConnection,
        THttpServerRequest aRequestInfo,
        ref THttpServerResponse aResponseInfo)
{
  if (aRequestInfo.Method == "GET")
  {
    if (aRequestInfo.Document == "/test.html")
        {
          aResponseInfo.ResponseNo = 200;
          aResponseInfo.ContentText = "OK";
          aResponseInfo.ContentType = "text/html; charset=UTF-8";
        }
        else
        {
          aResponseInfo.ResponseNo = 404;
        }
  }
  else
  {
    aResponseInfo.ResponseNo = 500;
  }
}
```

2 · NETSH Commands

Register an URL

```
Delphi (VCL / FireMonkey)
```

```
netsh http add urlacl url=http://example.com:80/ user=DOMAIN\user
```

```
C++ Builder
```

```
netsh http add urlacl url=http://example.com:80/ user=DOMAIN\user
```

```
.NET (C#)
```

```
netsh http add urlacl url=http://example.com:80/ user=DOMAIN\user
```

3 · Resumable Downloads

An HTTP 206 Partial Content response is used when a server is fulfilling a request for a specific portion (range) of a resource, instead of sending the entire file. This is commonly used for resumable downloads, media streaming, and large file transfers.

```
Delphi (VCL / FireMonkey)
```

```
procedure OnHTTPRequest(aConnection: TsgcWSConnection_HTTPAPI;
const aRequestInfo: THttpServerRequest; var aResponseInfo: THttpServerResponse);
var
  oStream: TFileStream;
  oRanges: TIdEntityRanges;
begin
  oStream := TFileStream.Create('test.pdf', fmOpenRead);
  oRanges := TIdEntityRanges.Create(nil);
  Try
    oRanges.Text := aRequestInfo.Range;
    aResponseInfo.ContentType := 'application/pdf';
    if oRanges.Count > 0 then
    begin
      aResponseInfo.ResponseNo := 206;
      aResponseInfo.AcceptRanges := 'bytes';
      aResponseInfo.ContentRangeStart := oRanges[0].StartPos;
      aResponseInfo.ContentRangeEnd := oRanges[0].EndPos;
      aResponseInfo.ContentRangeInstanceLength := oStream.Size;
      aResponseInfo.ContentStream := TIdHTTPRangeStream.Create(oStream,
        aResponseInfo.ContentRangeStart, aResponseInfo.ContentRangeEnd);
    end
    else
    begin
      aResponseInfo.ResponseNo := 200;
      aResponseInfo.ContentStream := oStream;
    end;
  Finally
    oRanges.Free;
  End;
end;
```

C++ Builder

```
void __fastcall OnHTTPRequest(TIdContext* AContext, TIdHTTPRequestInfo* ARequestInfo, TIdHTTPRes
{
  std::unique_ptr<TFileStream> oStream(new TFileStream("test.pdf", fmOpenRead | fmShareDenyWrite
  std::unique_ptr<TIdEntityRangeStrings> oRanges(new TIdEntityRangeStrings(nullptr));
  try {
    oRanges->Text = ARequestInfo->RawHeaders->Values["Range"];
    AResponseInfo->ContentType = "application/pdf";
    if (oRanges->Count > 0) {
      AResponseInfo->ResponseNo = 206; // Partial Content
      AResponseInfo->AcceptRanges = "bytes";
      AResponseInfo->ContentRangeStart = oRanges->Items[0]->StartPos;
      AResponseInfo->ContentRangeEnd = oRanges->Items[0]->EndPos;
      AResponseInfo->ContentRangeInstanceLength = oStream->Size;

      AResponseInfo->ContentStream = new TIdHTTPRangeStream(oStream.release(),
        AResponseInfo->ContentRangeStart, AResponseInfo->ContentRangeEnd);
    } else {
      AResponseInfo->ResponseNo = 200; // OK
      AResponseInfo->ContentStream = oStream.release();
    }
  } catch (...) {
    // Handle exceptions
  }
}
```

.NET (C#)

```
[HttpGet("download")]
public async Task<IActionResult> DownloadFile()
{
  string filePath = "test.pdf";
  if (!System.IO.File.Exists(filePath))
  {
    return NotFound();
  }
  FileInfo fileInfo = new FileInfo(filePath);
  long fileSize = fileInfo.Length;
  string contentType = "application/pdf";
  HttpContext.Request.Headers.TryGetValue("Range", out var rangeHeader);

  if (!string.IsNullOrEmpty(rangeHeader))
  {
    long start, end;
    string[] range = rangeHeader.ToString().Replace("bytes=", "").Split('-');
    start = string.IsNullOrEmpty(range[0]) ? 0 : long.Parse(range[0]);
    end = string.IsNullOrEmpty(range[1]) ? fileSize - 1 : long.Parse(range[1]);
    if (end ≥ fileSize)
      end = fileSize - 1;
    long contentLength = end - start + 1;

    Response.StatusCode = (int)HttpStatusCode.PartialContent;
    Response.Headers["Content-Range"] = $"bytes {start}-{end}/{fileSize}";
    Response.Headers["Accept-Ranges"] = "bytes";
    byte[] buffer = new byte[contentLength];
    using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read, FileShare.
    {
      fs.Seek(start, SeekOrigin.Begin);
      await fs.ReadAsync(buffer, 0, buffer.Length);
    }
    return File(buffer, contentType);
  }
  return File(System.IO.File.OpenRead(filePath), contentType);
}
```

4 · HTTPAPI FineTune

The following snippet configures an HTTP.sys server for a high-concurrency IoT backend: a large kernel queue to absorb reconnect storms, HighPerf dispatch with a widened pre-posted receive window, and inline-completion dispatch enabled.

```
Delphi (VCL / FireMonkey)
```

```
uses
  sgcWebSocket_Server_HTTPAPI,
  sgcWebSocket_HTTPAPI_Server;   // TsgcHTTPAPIOperatingMode
var
  oServer: TsgcWSServer_HTTPAPI;
begin
  oServer := TsgcWSServer_HTTPAPI.Create(nil);
  oServer.Host := '0.0.0.0';
  oServer.Port := 8080;
  // absorb 10,000-device reconnect bursts before kernel-level 503
  oServer.FineTune.QueueLength := 10000;
  // switch from single-acceptor to pre-posted IOCP workers
  oServer.FineTune.OperatingMode := ompHighPerf;
  // widen the per-worker pre-posted receive window (32 threads * 8 = 256)
  oServer.FineTune.HighPerfAcceptsPerWorker := 8;
  // dispatch inline on sync-success completions; skip the IOCP round-trip
  oServer.FineTune.SkipIOCPOnSuccess := True;
  oServer.Active := True;
end;
```

C++ Builder

```
#include "sgcWebSocket_Server_HTTPAPI.hpp"
#include "sgcWebSocket_HTTPAPI_Server.hpp"   // TsgcHTTPAPIOperatingMode
TsgcWSServer_HTTPAPI *oServer = new TsgcWSServer_HTTPAPI(NULL);
oServer->Host = "0.0.0.0";
oServer->Port = 8080;
// absorb 10,000-device reconnect bursts before kernel-level 503
oServer->FineTune->QueueLength = 10000;
// switch from single-acceptor to pre-posted IOCP workers
oServer->FineTune->OperatingMode = ompHighPerf;
// widen the per-worker pre-posted receive window (32 threads * 8 = 256)
oServer->FineTune->HighPerfAcceptsPerWorker = 8;
// dispatch inline on sync-success completions; skip the IOCP round-trip
oServer->FineTune->SkipIOCPOnSuccess = true;
oServer->Active = true;
```

Sources used to build this document

Every external claim links back to a primary source. The online-help references decode the canonical deep-link the company maintains for this component.

Primary standard / spec — HTTP Server API — Microsoft Learn	learn.microsoft.com/en-us/windows/win32/http/about-http-server-api
Primary standard / spec — WebSocket Protocol — RFC 6455	datatracker.ietf.org/doc/html/rfc6455
Online help — component page	www.esegece.com/help/sgcWebSockets/Components/TsgcWebSocketServer_HTTPAPI.htm
Delphi demo project (in the sgcWebSockets package)	`Demos\03.WebSocket_High_Performance_Server\01.HTTP_SYS_WebSocket_Server`
Component page	www.esegece.com/products/websockets/components/scaling/http-api-server/
Product page	www.esegece.com/products/websockets/

> **Document scope.** This document covers the publicly-documented surface of the HTTP.sys API Server component shipped with sgcWebSockets. For full property, method and event reference consult the online help linked above.