WebSocket.au3

A Quick Start to the native Windows WebSocket UDF

`ws:// and wss:// - client and server - no third-party DLL`

What you get	Facts
Four layers	wsk (Winsock), tls (Schannel), wsp (websocket.dll), ws (the API you call). 210 public functions, 8 connections, 8 listeners.
Both directions	The same four layers dial and answer. wss works on both sides.
Both bitnesses	Every proof suite is replayed as x86 and as x64. The last full gate: 19 suites, 38 runs, 38 zero return codes.
Named failures	No silent minus one. Each step reports its own number, the Winsock or HRESULT code, and a text.
Nothing to install	Only ws2_32.dll, secur32.dll and websocket.dll, all part of Windows 8 and later.

This booklet is written for one reader: a scripter who has never opened the source and wants a working connection before lunch. Chapters 03 and 04 are complete programs. Chapter 12 is the page to keep next to the editor. Everything stated here is read from the four layer files or from a measured proof run, and when a number is a measurement the chapter names the probe that produced it.

Contents

Num	Chapter	Page
01	What this is, in three facts	01
02	Install and first check	02
03	First connection	03
04	First server	04
05	The calls you will use	05
06	Reading: pump, wait or call	06
07	Writing: arm, flush, round trip	06
08	Failures name themselves	07
09	TLS, wss and certificates	08
10	Limits, named	09
11	Recipes	09
12	Cheat sheet	10

01

What this is, in three facts

No callbacks, every failure names itself, four layers under one include.

Fact 1 - there is no callback, you pump

The transport moves forward only while one of the budgeted calls is running: _WS_Pump, _WS_WaitMessage, _WS_Call, _WS_UpgradeHead, _WS_Close. Nothing arrives in the background and nothing calls you back. That is a design choice, not an accident of the component: a script that also owns a GUI stays honest when it decides, on each of its own turns, how long it is willing to wait for bytes.

THE LOOP THIS IMPOSES - four lines, and they are the whole receive side

```
While 1
    Local $iGot = _WS_Pump($h, 20)               ; serve the queues, at most 20 ms
    For $m = 1 To $iGot
        ConsoleWrite(_WS_MessageText($h) & @CRLF) ; dequeue in arrival order
    Next
WEnd
```

Fact 2 - a failure always says which step and which code

Every entry point returns a negative or zero value on failure, sets @error to the step it reached, and puts the system code (Winsock or HRESULT) in @extended. Chapter 08 is that table in full. Two strings finish the story: _WS_ErrorText($h) is the last named error of that handle, and _WS_LastWalk() is the journal of every address the resolver returned, each with its start and end stamp.

Fact 3 - four layers, one include, and you only ever call ws

ws.au3 is the API. It sits on wsp.au3 (the Windows WebSocket component), which sits on tls.au3 (Schannel), which sits on wsk.au3 (Winsock). A normal program includes WebSocket.au3 and calls _WS_* functions. Reaching into a lower layer is a caller's choice, never a need, and the umbrella decides nothing about your policy - it only loads the four files in dependency order.

Layer	File	Wraps	Functions	You call it when
1	wsk.au3	ws2_32.dll (Winsock)	40	never, unless you want raw sockets
2	tls.au3	secur32.dll (Schannel/SSPI)	61	only for a TLS context outside a WebSocket
3	wsp.au3	websocket.dll (Windows component)	63	only for a component detail _WS_ does not expose
4	ws.au3	the three layers above	46	always - this is the API

> **Why that is worth knowing**
>
> Each of the four files is annotated with the probe that measured the size, offset or HRESULT it uses. When something in this booklet says measured, you can follow the name to the suite that produced it.

02 Install and first check

Five files, one include, nothing to download.

What to copy, and where

Copy the WebSocket folder next to your script. Five files make the UDF run - the umbrella and its four layers - and none of them is generated at run time.

THE FOLDER EXACTLY AS IT SHIPS

```
WebSocket\
  WebSocket.au3        <- the only file your script includes
  ws.au3               layer 4, the API (46 functions)
  wsp.au3              layer 3, websocket.dll (63)
  tls.au3              layer 2, Schannel (61)
  wsk.au3              layer 1, Winsock (40)
  LICENSE              MIT
  examples\
    example_server.au3   echo server, ws and wss, subprotocols, Origin policy
    example_client.au3   the matching client, text, binary, 102 400 bytes, close
    hello_server.au3     the server printed in chapter 04, run as printed
    hello_client.au3     the client printed in chapter 03, run as printed
```

Requirements, as measured

Item	Requirement
AutoIt	3.3.16.1 or later. Au3Check at warning level 1 reports zero errors on the five files.
Operating system	Windows 8 or later: websocket.dll is a system component since Windows 8. There is no fallback to a downloaded DLL.
Bitness	32 and 64 bit. The structures are sized per bitness, never assumed.
Elevation	none. Neither _WS_Listen nor _WS_Connect asks for it.
System libraries	ws2_32.dll, secur32.dll, crypt32.dll and websocket.dll, all already on the machine.

Sixty-second check that the copy is complete

CHECK.AU3 - RUN IT, IT ONLY TALKS TO WINDOWS

```
#include "WebSocket\WebSocket.au3"

If Not _WS_Startup() Then
    ConsoleWrite("startup failed: " & _WS_ErrorText(-1) & @CRLF)
    Exit 1
EndIf
ConsoleWrite("ready, slots=" & $g_iWspSlots & " listeners=" & $g_iWsListeners & _
    " x64=" & @AutoItX64 & @CRLF)
_WS_Cleanup()
```

Expected line: ready, slots=8 listeners=8 x64=0. If it is missing, the reason is in _WS_ErrorText(-1) and it is a system DLL that is not there, not a corrupted copy.

03

First connection

A complete client, sixteen lines, read one line at a time.

The smallest honest client: open, send one line, print the answer, close properly.

HELLO_CLIENT.AU3 - COMPLETE

```
#include "..\WebSocket.au3"

If Not _WS_Startup() Then Exit 1

Local $h = _WS_Connect("ws://127.0.0.1:8181/chat")
If $h < 0 Then
    ConsoleWrite("step " & @error & "/" & @extended & " :: " & _WS_ErrorText(-1) & @CRLF)
    ConsoleWrite("addresses: " & _WS_LastWalk() & @CRLF)
    _WS_Cleanup()
    Exit 1
EndIf

Local $b = _WS_Call($h, "hello", 8000)
If @error Then
    ConsoleWrite("round trip failed: " & _WS_ErrorText($h) & @CRLF)
Else
    ConsoleWrite("answer: " & BinaryToString($b, 8) & @CRLF)
EndIf

_WS_Close($h, 1000, "done", 2000)
_WS_Cleanup()
```

One line at a time

Line	What it really does
_WS_Startup()	Loads and starts the three system DLLs, allocates the 8 connection slots and the 8 listeners, returns 1. Without it every other call fails at step 7.
_WS_Connect(url)	Parses the URL, resolves the host, races every resolved address, opens TCP, then TLS if the scheme is wss, then asks for the upgrade and waits for the 101 - all inside one budget.
$h	A handle is a connection slot index, 0 to 7. It is not a socket, and a negative handle never touches a slot.
step @error/@extended	@error is the phase reached, @extended the Winsock or HRESULT code. Both are printed because one of them alone cannot be asserted on.

The one thing that trips people up

A text answer arrives as BINARY, never as a string. _WS_Call, _WS_WaitMessage and _WS_Message all return binary and put the frame type in @extended ($Wsp_UTF8_MESSAGE or $Wsp_BINARY_MESSAGE). _WS_MessageText is the only reader that decodes UTF-8 for you, and it deliberately returns an empty string for a binary frame rather than inventing a reading of bytes that are not text.

04

First server

Listen, accept, upgrade, echo - and where YOUR policy lives.

A server is the same API seen from the other end: a listener, a loop that accepts, an upgrade you answer, then the same read and write calls on each connection.

HELLO_SERVER.AU3 - COMPLETE

```
#include "..\WebSocket.au3"

If Not _WS_Startup() Then Exit 1
Local $hl = _WS_Listen(8181)                        ; 127.0.0.1, plain ws://
If $hl < 0 Then
    ConsoleWrite("cannot listen: " & _WS_ErrorText(-1) & @CRLF)
    _WS_Cleanup()
    Exit 1
EndIf
ConsoleWrite("listening on " & _WS_ListenPort($hl) & @CRLF)

While 1
    Local $h = _WS_Accept($hl)                      ; -1 = nobody is dialing now
    If $h < 0 Then
        Sleep(10)                                   ; _WS_Accept never waits, so the loop must
        ContinueLoop
    EndIf
    If _WS_UpgradeHead($h, 4000) < 0 Then           ; the request head first
        _WS_CloseRefused($h, "400 Bad Request", 2000)
        ContinueLoop
    EndIf
    ConsoleWrite("path=" & _WS_Path($h) & " origin=" & _WS_RequestHeader($h, "Origin") & @CRLF)
    If _WS_UpgradeAnswer($h, "", "", 4000) < 0 Then    ; accept, no subprotocol
        _WS_CloseHandle($h)
        ContinueLoop
    EndIf
    Local $b = _WS_WaitMessage($h, 8000)
    If @error = 0 Then _WS_Send($h, BinaryToString($b, 8))
    _WS_Flush($h, 3000)
    _WS_Close($h, 1000, "", 2000)
WEnd
; a real exit path also calls _WS_CloseListener($hl) and _WS_Cleanup()
```

What the server owns and what the UDF never decides

Decision	Where it lives
Which Origin is allowed	your code - pass the list as $sOriginAllow to _WS_UpgradeAnswer, or read the header yourself and refuse; the UDF only compares
Which subprotocol is kept	the order in $sSubprotocols you hand to _WS_UpgradeAnswer; the first match wins and _WS_SelectedSub($h) reports it
Which path means what	your code, read with _WS_Path($h) and _WS_RequestHeader($h,$sName) before you answer
Which HTTP status refuses	your code, in _WS_CloseRefused($h,"403 Forbidden",...)
Whether the peer is trusted	not this UDF at all - read the header you want and decide before answering

Both programs in this chapter are in the zip and were run

They ship as examples\hello_server.au3 and examples\hello_client.au3, inside the WebSocket folder, exactly as printed here - which is why their include reads ..\WebSocket.au3. Run on 2026-11-17 from a copy extracted out of the release zip: the server answers path=/chat and the client prints answer: hello with return code 0, on the 32-bit and the 64-bit interpreter.

The accept loop above is sequential on purpose

_WS_Accept takes one new peer per call and the loop serves one message per peer. That is right for a tool and for a proof. It is not a server for many simultaneous clients: to serve several, keep the accepted handles in an array and pump each one a little per turn. The shipped example_server.au3 has that shape, and spike10 proved two peers interleaved on one listener, six turns each, on both bitnesses.

05

The calls you will use

Everything else in the 210 functions is reachable from these.

Forty-six functions live in ws.au3. Twelve of them carry almost every program; the rest are accessors and policy helpers. The two tables are the working set, client first.

Client

Call	Returns	Use it for
_WS_Startup()	1 or 0	once, before anything else
_WS_Connect($sUrl,...)	handle or -1	opening ws:// or wss://
_WS_Call($h,$v,$iMs)	binary answer	a round trip - the simple case
_WS_Send($h,$v)	1 or 0	writing when a loop will serve the queue anyway
_WS_Flush($h,$iMs)	frames written	forcing armed frames out now
_WS_WaitMessage($h,$iMs)	binary message	waiting for one arrival, fragments reassembled
_WS_Pump($h,$iMs)	count delivered	one tick of a loop you own
_WS_MessageText($h)	decoded text	dequeuing after _WS_Pump said 1 or more
_WS_Close($h,$c,$r,$iMs)	1 or 0	closing the way RFC 6455 asks
_WS_Cleanup()	1	once, at the end

Server

Call	Returns	Use it for
_WS_Listen($iPort,$sAddr,$sPfx,$sPass,...)	listener or -1	a plain ws listener (no PFX) or a wss one
_WS_ListenPort($hl)	port bound	any caller that passed 0 and must know the real port
_WS_Accept($hl)	handle or -1	one new peer, non-blocking
_WS_UpgradeHead($h,$iMs)	1 or -1	reading the request head before deciding
_WS_UpgradeAnswer($h,$subs,$originList,$iMs)	1 or -1	accepting, with the policy you chose
_WS_CloseRefused($h,$sStatus,$iMs)	1 or -1	refusing with a named HTTP status
_WS_Path / _WS_Method / _WS_RequestHeader	text	routing and Origin checks
_WS_SelectedSub / _WS_OfferedSub	text	what was kept, what was offered
_WS_CloseInfo($h)	text	who closed, with which code and reason
_WS_CloseHandle($h)	1	releasing a handle that was never upgraded

06

Reading: pump, wait or call

Three ways to receive, and the one question that decides.

Three calls read, and choosing between them is one question: does your program already own a loop?

If	Use	Why
No loop yet, and you want one answer	_WS_Call($h,$v,$iMs)	it arms, flushes and waits - one call, one budget
No loop yet, but the peer speaks first	_WS_WaitMessage($h,$iMs)	reassembles fragments and returns the whole message
You own a loop (GUI, timer, several peers)	_WS_Pump($h,$iMs), then _WS_Message	a tick can deliver more than one message, so dequeue them in arrival order

Fragments are not your problem, but the queue is

A large message reaches the component as roughly $iRx-sized pieces plus a final frame (measured in spike3, replayed with 102 400 bytes in spike9 and spike11). _WS_WaitMessage and _WS_Call reassemble them and answer once, with the whole payload. _WS_Pump, the non-blocking path, hands over COMPLETE messages - and sometimes several per tick. That is why its return value is a count, and why reading exactly one message per tick quietly loses data.

THE SAFE DRAIN, EVERY TURN

```
Local $iGot = _WS_Pump($h, 20)
If @error = 3 Then ConsoleWrite("queue full, " & @extended & " lost" & @CRLF)
For $m = 1 To $iGot
    Local $b = _WS_Message($h)
    If @extended = $Wsp_UTF8_MESSAGE Then
        ConsoleWrite("text: " & BinaryToString($b, 8) & @CRLF)
    Else
        ConsoleWrite("binary: " & BinaryLen($b) & " bytes" & @CRLF)
    EndIf
Next
```

What a closed peer looks like

_WS_WaitMessage returns empty binary with @error 3 when the peer closed the WebSocket: @extended carries its close code and the returned value is its reason as bytes. _WS_Pump answers @error 1 when the TCP connection itself went away. A close is never queued as a message - you learn it from the type of the call that saw it, and _WS_CloseInfo($h) then holds the text of who closed, with what code and what reason.

07

Writing: arm, flush, round trip

The queue is real, so the word sent has three meanings.

Writing has three meanings, and the UDF keeps them apart instead of hiding the queue behind one word.

Call	Worked really means	Budget
_WS_Send($h,$v)	the frame is ARMED in the component; the bytes leave when something pumps	none, by design
_WS_Request($h,$v,$iMs)	the frame left AND the component confirmed it	$iMs for the flush
_WS_Call($h,$v,$iMs)	the answer has been read	$iMs for the flush plus $iMs for the answer

ARM THREE, FLUSH ONCE

```
For $i = 1 To 3
    _WS_Send($h, "mark" & $i)
Next
Local $iFrames = _WS_Flush($h, 5000)
If $iFrames < 3 Then ConsoleWrite("not all frames written: " & $iFrames & @CRLF)
; then drain the three echoes with _WS_Pump, one turn at a time
```

> **A spent budget is not a failed send**
>
> _WS_Request sets @error 3 when the budget ran out. The frame may still be in the send queue and may still leave afterwards, so it is reported as a budget note and never as a silent success or a silent loss. _WS_Call deliberately keeps waiting for the answer after that @error 3, because in practice the frame has gone out anyway.

Text, binary, and size

- _WS_Send picks the frame type from the argument: a string goes out as UTF-8 text, Binary(...) as a binary frame.
- _WS_SendBinary($h,$bData) forces a binary frame - that is what the echo server uses to keep the type it received.
- The largest message the UDF reassembles is $iMaxMsg, armed at connect or listen. The default 4 194 304 bytes is the same ceiling Windows IIS puts on a received message, which is a sensible number to leave alone.

08

Failures name themselves

Every step number, every system code, the address journal.

These are the @error values of _WS_Connect. Nothing else in the UDF invents a code: a lower layer's error is either mapped onto one of these steps or surfaced by _WS_ErrorText.

@error	Step reached	@extended holds	What to check first
1	the URL is not usable	1 scheme, 2 host, 3 port, 4 form	ws:// or wss://, host, optional :port, /path
2	name never resolved	Winsock code, 11001 = unknown host	DNS, or use the literal IP
3	no address of the list answered	last Winsock code: 10061 refused, 10060 budget	is something listening, and is the budget large enough
4	layer 2 unavailable	DllCall or Schannel package	wss only: secur32.dll and the TLS packages
5	TLS handshake never finished	layer 2 state	the certificate offered, the protocols the machine allows
6	peer chain REJECTED	verdict HRESULT	a trust problem, not a transport problem - chapter 09
7	no slot, or websocket.dll missing		did _WS_Startup run? there are only 8 slots
8	component refused the upgrade request	HRESULT	the path and headers the peer expects
9	request could not be written	named layer 1 or 2 error	the socket, or the TLS context
10	the 101 never arrived	bytes received	the peer answered 4xx or 200 - read _WS_ErrorText
11	answer invalid	component HRESULT	the endpoint is not a WebSocket
12	arming the receive refused		the buffer sizes given to _WS_Connect

The two strings that finish the story

PRINT EVERYTHING THE UDF KNOWS ABOUT A FAILED OPEN

```
Local $h = _WS_Connect("wss://127.0.0.1:8443/chat", 6000)
If $h < 0 Then
    ConsoleWrite("step " & @error & " / " & @extended & @CRLF)
    ConsoleWrite("named : " & _WS_ErrorText(-1) & @CRLF)
    ConsoleWrite("walk  : " & _WS_LastWalk() & @CRLF)
EndIf
```

A walk line from a real run, where the first record was IPv6 and the second won:

THE JOURNAL

```
#1 [0000:0000:0000:0000:0000:0000:0000:0001] start=1 end=517 abandoned
#2 127.0.0.1                                 start=515 end=517 CONNECTED
```

Read it as a race, not a walk: both records were in flight, the second answered first, the first was closed. On a host that resolves to several addresses this is what keeps an open to localhost at a few hundred milliseconds instead of the two

seconds a serial walk costs - probe_d16 measured 2 036 ms serial against 256 to 517 ms raced, which is rule V29 in the source.

Close codes, as this UDF hands them

Code	Meaning	In this UDF
1000	normal closure	the default of _WS_Close
1001	going away	used by the examples to show a reason travelling
1002 / 1003	protocol error / bad data type	written by the component or by your own refusal
1005 / 1006	no status / abnormal closure	never put on the wire: replaced by 1000. The field that tells the two cases apart is fromPeer= in _WS_CloseInfo
1007 / 1008 / 1009	bad payload / policy / too large	1009 is what RFC 6455 reserves for a message over $iMaxMsg; this suite has not exercised that path
1011	internal error	what the server sends when your code hits a bug
1015	TLS failure	a 1015 you write is rejected by the component - use 1011 with text

09 TLS, wss and certificates

What is verified, what the flag does not weaken, and the missing ALPN.

wss:// is not a second protocol here: it is the same WebSocket carried by layer 2, which is Schannel, the TLS already in Windows. Nothing was vendored, so nothing has to be updated by you.

Question	Answer
Who verifies the peer?	Schannel, against the Windows certificate store. The hostname you asked for is part of that check.
What does $fIgnoreRoot = True change?	It accepts a chain whose root is not trusted, which is what a self-signed test certificate produces. It does NOT weaken the name check.
What does a rejected chain look like?	_WS_Connect step 6, @extended the verdict HRESULT, text in _WS_ErrorText: 0x800B0109 untrusted root, 0x800B010F expired (both measured).
Which TLS versions?	whatever the machine allows, and this machine is Windows 10 19045. Measured on 2026-11-17 against ws.postman-echo.com, echo.websocket.org and echo-websocket.fly.dev with verification ON: all three settled on TLS 1.2 (ECDHE-RSA-AES256-GCM-SHA380 or AES128-GCM), at both bitnesses, open 596 to 2 885 ms. Schannel added TLS 1.3 in Windows 11 and Server 2022, so no session here - client or server - has ever reported 1.3.
Is ALPN used?	No. The Windows component does not offer it, so the subprotocol travels in the HTTP header alone - measured, and documented in wsp.au3.
Do I need a certificate to listen in wss?	Yes: a PFX path and its password, given to _WS_Listen. No PFX means the listener is plain ws://.

CLIENT AND SERVER - THE ONLY DIFFERENCES

```
; client: the scheme is the whole change, plus the flag for a test certificate
Local $h = _WS_Connect("wss://127.0.0.1:8443/chat", 15000, "", 65536, 4194304, True)

; server: a PFX turns the same listener into wss
Local $hl = _WS_Listen(8443, "127.0.0.1", "C:\keys\srv.pfx", "secret")
```

> **Do not ship True as your default**
>
> $fIgnoreRoot exists for the test certificate you generated yourself. A program that carries it as the default has no chain validation and will talk to whatever answers on the port. The name check stays active either way, which is why a certificate issued for the wrong host still fails.

Thirty seconds, no server of your own

TWELVE LINES AGAINST A REAL SERVER, VERIFICATION ON

```
#include "WebSocket\WebSocket.au3"

If Not _WS_Startup() Then Exit 1
Local $h = _WS_Connect("wss://ws.postman-echo.com/raw", 15000, "", 65536, 4194304, False)
If $h < 0 Then
    ConsoleWrite("step " & @error & "/" & @extended & " :: " & _WS_ErrorText(-1) & @CRLF)
    Exit 1
EndIf
ConsoleWrite("secure=" & _WS_IsSecure($h) & " answer=" & BinaryToString(_WS_Call($h, "ping"), 8) & @CRLF)
_WS_Close($h, 1000, "", 2000)
_WS_Cleanup()
```

Measured on 2026-11-16 with $fIgnoreRoot = False, so Schannel validated the chain against the Windows store and the host name: ws.postman-echo.com returned the word ping (open 1 008 ms, echo 193 ms), echo.websocket.org and echo-websocket.fly.dev answered their service identity (open 762 and 952 ms), and echo.websocket.events failed at step 2 with 11001 because that name no longer resolves - which is exactly the named failure chapter 08 promises. Any of those three lines is a test you can run before you trust a single line of this booklet.

10 Limits, named

Read this before you plan a feature, not after.

Every line below is a door that was tested and did not open, or a ceiling that was measured. Reading this page first is cheaper than discovering it in a design review.

Limit	Detail
8 connections, 8 listeners	$g_iWspSlots and $g_iWsListeners are constants, they do not grow. _WS_Connect fails at step 7 and _WS_Listen refuses beyond them.
No ping frame from your side	WebSocketSend with a ping payload answers E_INVALIDARG - keep-alive belongs to the component.
No callbacks	fact 1: nothing arrives unless a budgeted call is running.
Sequential accept	_WS_Accept takes one peer per call; interleaving peers is your loop, proved in spike10.
A close is not a message	it surfaces as the error of the pump that saw it, never in the delivery queue (rule V28 in wsp.au3).
Subprotocol by header only	no ALPN, so a peer that insists on ALPN will not negotiate through this stack.
4 194 304 bytes	the default receive ceiling, armed per connection at connect or listen.
TLS 1.3, either direction	never observed on this machine. Three public echo servers and local peers that offer 1.3 all settled on TLS 1.2 with verification ON, at both bitnesses, and the TLS 1.3 branch of the version reporter in tls.au3 has never fired. Schannel added TLS 1.3 in Windows 11 and Server 2022: run it there and tell me what you get.
No HTTP body on a refused upgrade	the 101 or the 4xx is the whole answer.
No DOM, no browser	this is a transport. Rendering a page is WebView2's job, a different UDF.

What is proved, and by which suite

spike3 for layer 3, spike4 and spike5 for the TLS client, spike7 for the layer 4 client, spike8 and spike9 for a TLS and wss server, spike10 for two simultaneous peers, spike11 for the layer 4 server, probe_d16, probe_d16c and probe_d16e for open latency, probe_pubtls for the TLS version negotiated with public servers. The guard gate replays 19 of those suites at both bitnesses - 38 runs - and the last recorded run finished with 38 zero return codes, with the failure counters read inside the logs and not just the exit codes.

11 Recipes

Subprotocols, Origin policy, GUI loop, large binary.

Offer subprotocols, and find out which one survived

CLIENT OFFERS, SERVER PICKS

```
; client: one header, and the component writes it for you
Local $h = _WS_Connect("ws://127.0.0.1:8181/chat", 15000, "chat,superchat")
ConsoleWrite("kept by the server: [" & _WS_SelectedSub($h) & "]" & @CRLF)

; server: your list is the preference order, the first match with the offer wins
_WS_UpgradeAnswer($h, "superchat,chat", "", 4000)
ConsoleWrite("we kept " & _WS_SelectedSub($h) & @CRLF)
```

Route by path, and refuse a foreign Origin by name

POLICY IN YOUR CODE, REFUSAL NAMED

```
Local $sPath = _WS_Path($h), $sOrigin = _WS_RequestHeader($h, "Origin")
If $sPath = "/admin" And $sOrigin <> "https://trusted.example" Then
    _WS_CloseRefused($h, "403 Forbidden", 2000)
Else
    _WS_UpgradeAnswer($h, "", "https://trusted.example", 4000)
EndIf
```

Handing the list to _WS_UpgradeAnswer is the shorter form: the UDF compares the header against the list and refuses with 403 itself, and the refusal appears in the journal as a named step. An empty list means no check at all. This is the cross-site check a WebSocket server has to do, because a page on any origin can open the socket.

A GUI that talks without freezing

GUIGETMSG LOOP, ONE SHORT TICK PER TURN

```
Local $h = _WS_Connect("ws://127.0.0.1:8181/chat")
While 1
    Local $msg = GUIGetMsg()
    If $msg = $GUI_EVENT_CLOSE Then ExitLoop
    Local $iGot = _WS_Pump($h, 20)          ; never longer than 20 ms per turn
    For $m = 1 To $iGot
        GUICtrlSetData($edit, _WS_MessageText($h))
    Next
WEnd
_WS_Close($h, 1000, "", 2000)
_WS_Cleanup()
```

Send a large binary and wait for the answer

BINARY, WITH THE CEILINGS ARMED LARGE ENOUGH

```
Local $h = _WS_Connect("ws://127.0.0.1:8181/blob", 15000, "", 65536, 8388608)
_WS_SendBinary($h, $bData)                    ; armed only
If _WS_Flush($h, 20000) < 1 Then ConsoleWrite("nothing written" & @CRLF)
Local $b = _WS_WaitMessage($h, 20000)          ; reassembles the fragments
```

102 400 bytes went through the API whole and were seen as fragments on the way, in spike9 and again in spike11. If an answer needs a longer budget than the payload, raise the budget rather than looping on _WS_Message and guessing.

12 Cheat sheet

One page to keep next to the editor.

Task	One line
Boot	_WS_Startup() ... _WS_Cleanup()
Open	_WS_Connect("ws://host:port/path", budgetMs, subs, rx, maxMsg, ignoreRoot)
Open secure	_WS_Connect("wss://host/path", 15000, "", 65536, 4194304, False)
Ask and answer	_WS_Call($h, "text", 8000) -> binary, frame type in @extended
Just send	_WS_Send($h, $v) then _WS_Flush($h, 5000)
Read once	_WS_WaitMessage($h, 5000)

Rules of the house

- Pump or wait; bytes never move on their own.
- Read the count _WS_Pump gives you, then dequeue that many messages.
- On failure print four things together: @error, @extended, _WS_ErrorText and _WS_LastWalk. One of them alone is a guess.
- Arm $iRx and $iMaxMsg at connect or listen; once the connection is open they belong to the peer.
- Close with _WS_Close rather than dropping the handle, or the peer sees 1006 instead of your reason.
- Do not call a lower layer to save one call - the counters of the four layers stay in sync only for _WS_* traffic.

Licence and author

WebSocket.au3 and its four layers are by Numeric1, under the MIT licence in the LICENSE file of the same folder. This booklet was typeset with PDFX, by the same author.