

WebSocket Protocol Quick Reference

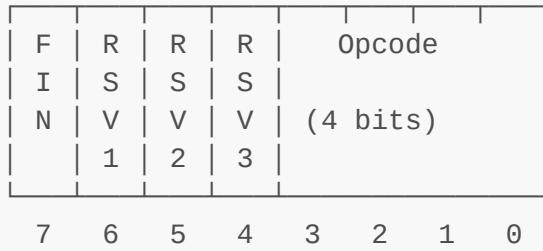
🎯 Quick Start

```
# Start server
node server.mjs

# Open client
open index.html
```

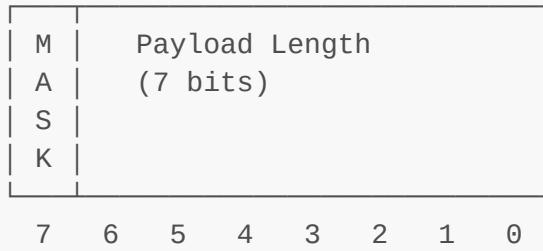
📦 Frame Structure

Byte 1: FIN + RSV + Opcode



- **FIN:** 1 = final fragment, 0 = more fragments coming
- **RSV1-3:** Reserved for extensions (must be 0)
- **Opcode:** Frame type

Byte 2: MASK + Payload Length



- **MASK:** 1 = masked (client → server), 0 = not masked (server → client)
- **Payload Length:**
 - 0-125: Actual length
 - 126: Next 2 bytes = length (uint16)
 - 127: Next 8 bytes = length (uint64)

Opcodes

Hex	Dec	Type	Description
0x0	0	CONTINUATION	Continuation of fragmented msg
0x1	1	TEXT	UTF-8 text message
0x2	2	BINARY	Binary data
0x8	8	CLOSE	Connection close
0x9	9	PING	Heartbeat request
0xA	10	PONG	Heartbeat response

🔒 Handshake

Client Request Headers

```
GET / HTTP/1.1
Host: localhost:1338
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNbXBsZSSub25jZQ==
Sec-WebSocket-Version: 13
```

Server Response Headers

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzhZRBK+xOo=
```

Accept Key Calculation

```
const accept = Base64(SHA1(clientKey + "258EAFA5-E914-47DA-95CA-
C5AB0DC85B11"));
```

🎭 Masking Algorithm

Client→Server messages MUST be masked:

```
// Masking (client)
for (let i = 0; i < payload.length; i++) {
  masked[i] = payload[i] ^ maskingKey[i % 4];
}
```

```
// Unmasking (server)
for (let i = 0; i < payload.length; i++) {
    unmasked[i] = masked[i] ^ maskingKey[i % 4];
}
```

Masking key: 4 random bytes chosen by client

Length Encoding

Small Message (<= 125 bytes)

Byte 2: [MASK bit][0-125]

Medium Message (126-65535 bytes)

Byte 2: [MASK bit][126]
Bytes 3-4: uint16 length (big-endian)

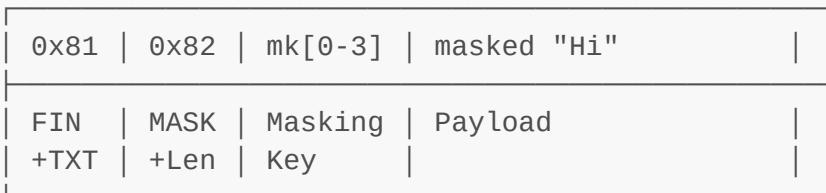
Large Message (> 65535 bytes)

Byte 2: [MASK bit][127]
Bytes 3-10: uint64 length (big-endian)

Common Frame Examples

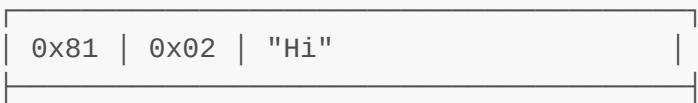
Text Frame (Client → Server)

Sending "Hi"



Text Frame (Server → Client)

Sending "Hi"



FIN	Len	Payload
+TXT		

Close Frame

0x88	0x02	0x03E8 (1000)
FIN	Len	Close Code
+CLS		(Normal)

Ping Frame

0x89	0x00	
FIN	Empty	
+PNG		

Pong Frame

0x8A	0x00	
FIN	Empty	
+PNG		

Bit Manipulation Cheat Sheet

1 2
3 4

```
// Check if bit is set
const isFIN = !(firstByte & 0x80); // Check bit 7
const isMasked = !(secondByte & 0x80); // Check bit 7

// Extract bits
const opcode = firstByte & 0x0f; // Get last 4 bits
const payloadLen = secondByte & 0x7f; // Get last 7 bits

// Set bits
const frame = 0x80 | opcode; // Set FIN + opcode
const lenByte = 0x80 | length; // Set MASK + length
```

```
// XOR operation
const unmasked = masked ^ maskKey; // Toggle bits
```

WebSocket States

```
WebSocket.CONNECTING; // 0: Connection not yet established
WebSocket.OPEN; // 1: Connection is open and ready
WebSocket.CLOSING; // 2: Connection is going through closing handshake
WebSocket.CLOSED; // 3: Connection is closed
```

Common Status Codes

Code	Meaning	Description
1000	Normal Closure	Successful operation / normal close
1001	Going Away	Endpoint going away (browser close)
1002	Protocol Error	Endpoint terminating due to error
1003	Unsupported Data	Data type cannot be accepted
1005	No Status Received	Reserved, must not be set
1006	Abnormal Closure	Reserved, must not be set
1007	Invalid Payload	Inconsistent data (e.g., invalid UTF-8)
1008	Policy Violation	Generic status code
1009	Message Too Big	Message too large to process
1010	Mandatory Extension	Client expected extension missing
1011	Internal Error	Server error
1015	TLS Handshake	Reserved, must not be set

Useful Calculations

Calculate Buffer Size

```
// Small message
const size = 2 + payloadLength;

// Medium message
const size = 2 + 2 + payloadLength;

// Large message
const size = 2 + 8 + payloadLength;
```

```
// With masking (client)
const size = baseSize + 4 + payloadLength;
```

Read Multi-byte Numbers

```
// 16-bit big-endian
const value = (buffer[0] << 8) | buffer[1];

// 32-bit big-endian
const value =
  (buffer[0] << 24) | (buffer[1] << 16) | (buffer[2] << 8) | buffer[3];

// Using Buffer methods
const value16 = buffer.readUInt16BE(0);
const value32 = buffer.readUInt32BE(0);
```

Debugging Tips

Log Frame Details

```
console.log("Frame:", {
  fin: !(firstByte & 0x80),
  rsv1: !(firstByte & 0x40),
  rsv2: !(firstByte & 0x20),
  rsv3: !(firstByte & 0x10),
  opcode: (firstByte & 0x0f).toString(16),
  masked: !(secondByte & 0x80),
  payloadLen: secondByte & 0x7f,
  maskKey: maskKey.toString("hex"),
  payload: payload.toString("utf8"),
});
```

Hex Dump

```
function hexDump(buffer) {
  return buffer
    .toString("hex")
    .match(/.{1,2}/g)
    .join(" ")
    .toUpperCase();
}
```

Binary Representation

```
function toBinary(byte) {
    return byte.toString(2).padStart(8, "0");
}
```

Magic Constants

```
// Protocol
const MAGIC_STRING = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
const VERSION = 13;

// Bits
const FIN_BIT = 0x80; // 10000000
const MASK_BIT = 0x80; // 10000000
const OPCODE_MASK = 0x0f; // 00001111
const LENGTH_MASK = 0x7f; // 01111111

// Length indicators
const LENGTH_7BIT = 125;
const LENGTH_16BIT = 126;
const LENGTH_64BIT = 127;

// Max payload sizes
const MAX_7BIT = 125; // 125 bytes
const MAX_16BIT = 65535; // 64 KB
const MAX_64BIT = 2 ** 53 - 1; // ~9 PB (JS safe integer)
```

Testing Commands

```
# Install wscat
npm install -g wscat

# Connect to server
wscat -c ws://localhost:1338

# Send message
> Hello WebSocket!
< Echo: Hello WebSocket!

# Binary message (hex)
wscat -c ws://localhost:1338 -x
> 48656c6c6f
< 4563686f3a2048656c6c6f
```