**DEX Sniper Pro WebSocket API Documentation**

**Overview**

DEX Sniper Pro implements a unified WebSocket system through a centralized hub architecture. The backend runs on localhost:8001 and provides real-time updates for autotrade operations, pair discovery, and system monitoring.

**Quick Reference**

| **Aspect** | **Details** |
| --- | --- |
| **Main Endpoint** | ws://localhost:8001/ws/{client\_id} |
| **Legacy Endpoint** | ws://localhost:8001/ws/autotrade |
| **Status Endpoint** | GET /ws/status |
| **Test Page** | GET /ws/test |
| **Channels** | autotrade, discovery, system, all |
| **Auth** | Client ID path parameter (no tokens required) |
| **Heartbeat** | 30-second intervals, 60-second timeout |

**1. Endpoints & URLs**

**Primary WebSocket Route**

ws://localhost:8001/ws/{client\_id}

* **Path Parameter**: client\_id (required) - Unique client identifier generated by frontend
* **Trailing Slash**: Not required
* **Development**: ws://localhost:8001/ws/{client\_id}
* **Production**: wss://yourdomain.com/ws/{client\_id}

**Legacy Routes (For Compatibility)**

ws://localhost:8001/ws/autotrade

**HTTP Endpoints**

* GET /ws/status - Hub status and connection stats
* GET /ws/test - HTML test page for development

**2. Authentication & Session**

**No traditional authentication required**. The WebSocket uses a simple client ID-based system:

* **Client ID**: Generated by frontend (e.g., dex\_${timestamp}\_${random})
* **Handshake**: Automatic on connection
* **Headers**: Standard WebSocket headers only
* **Cookies**: Not used
* **CSRF**: Not applicable

**Connection Example**

const clientId = `dex\_${Date.now()}\_${Math.random().toString(36).substr(2, 9)}`;

const ws = new WebSocket(`ws://localhost:8001/ws/${clientId}`);

**3. Message Contracts**

**Client → Server Messages**

interface ClientMessage {

id: string; // Unique message ID

type: MessageType; // Message type enum

channel: Channel; // Target channel

data: Record<string, any>; // Message payload

timestamp: string; // ISO timestamp

client\_id?: string; // Optional client identifier

}

**Server → Client Messages**

interface ServerMessage {

id: string; // Unique message ID

type: MessageType; // Message type enum

channel: Channel; // Source channel

data: Record<string, any>; // Message payload

timestamp: string; // ISO timestamp

client\_id?: string; // Target client (if applicable)

}

**Error Envelope**

interface ErrorMessage {

id: string;

type: "error";

channel: "system";

data: {

error\_code: string;

error\_message: string;

trace\_id?: string;

};

timestamp: string;

}

**4. Event Catalog**

**Message Types (Enum)**

**Autotrade Events**

* engine\_status - Engine state changes
* trade\_executed - Trade completion
* opportunity\_found - New trading opportunity
* risk\_alert - Risk management alerts

**Discovery Events**

* new\_pair - New pair detected
* risk\_update - Risk score changes
* discovery\_status - Discovery service status

**System Events**

* system\_health - Health status updates
* connection\_ack - Connection acknowledgment
* subscription\_ack - Channel subscription confirmation
* error - Error messages
* heartbeat - Keep-alive messages

**Channels (Enum)**

* autotrade - Trading engine updates
* discovery - Pair discovery feed
* system - System-wide notifications
* all - Special broadcast channel

**5. Connection Lifecycle**

**Connection Flow**

1. Client connects to ws://localhost:8001/ws/{client\_id}
2. Server accepts connection
3. Server sends connection\_ack with available channels
4. Client subscribes to desired channels
5. Server confirms with subscription\_ack
6. Real-time message exchange begins

**Close Codes**

* **1000**: Normal closure
* **1001**: Going away
* **1002**: Protocol error
* **1011**: Server error

**Heartbeat System**

* **Interval**: 30 seconds (server-initiated)
* **Timeout**: 60 seconds
* **Format**: { "type": "heartbeat", "channel": "system", "data": { "ping": true } }
* **Response**: Client should respond with pong (optional)

**6. Errors & Retries**

**Error Handling**

interface WebSocketError {

type: "error";

data: {

error\_code: "INVALID\_CHANNEL" | "SUBSCRIPTION\_FAILED" | "MESSAGE\_INVALID";

error\_message: string;

trace\_id?: string;

};

}

**Recommended Retry Strategy**

const reconnectDelays = [1000, 2000, 4000, 8000, 16000]; // Exponential backoff

let retryCount = 0;

const maxRetries = 5;

function reconnectWebSocket() {

if (retryCount >= maxRetries) {

console.error('Max reconnection attempts reached');

return;

}

setTimeout(() => {

connectWebSocket();

retryCount++;

}, reconnectDelays[Math.min(retryCount, reconnectDelays.length - 1)]);

}

**7. Concurrency & Limits**

* **Max Connections per Client ID**: 1 (previous connection replaced)
* **Message Rate Limit**: No explicit limit (reasonable use expected)
* **Message Size Limit**: No explicit limit (JSON messages expected to be < 1MB)
* **Channel Subscriptions**: Unlimited per client

**8. Ordering & Delivery**

* **Message Ordering**: Not guaranteed across channels
* **Delivery Semantics**: At-most-once (WebSocket reliability)
* **Deduplication**: Not implemented (client should handle if needed)

**9. Versioning**

* **Current Version**: v1 (implicit)
* **Breaking Changes**: No versioning strategy implemented yet
* **Backward Compatibility**: Legacy /ws/autotrade endpoint maintained

**10. Security & CORS**

**Development CORS**

// Allowed origins in backend

allow\_origins: ["http://localhost:3000", "http://localhost:5173"]

**Vite Proxy Configuration**

// frontend/vite.config.js

export default {

server: {

proxy: {

'/ws': {

target: 'http://localhost:8001',

ws: true,

changeOrigin: true

}

}

}

}

**Security Notes**

* No authentication tokens required
* Client ID prevents connection conflicts
* CORS configured for development ports

**11. Production Notes**

* **SSL Termination**: Use wss:// in production
* **Load Balancer**: WebSocket-aware load balancer required
* **Sticky Sessions**: Not required (stateless client identification)
* **Horizontal Scaling**: Hub is single-instance (no clustering implemented)

**12. Code Implementation**

**Backend Files**

* **Main Endpoint**: backend/app/api/websocket.py → websocket\_endpoint()
* **Hub Manager**: backend/app/ws/hub.py → class WebSocketHub
* **Message Types**: backend/app/ws/hub.py → class MessageType, class Channel
* **Broadcasting**: backend/app/api/websocket.py → broadcast\_\*\_message() functions

**Frontend Integration**

**Direct Connection (Port 8001)**

const clientId = `dex\_${Date.now()}\_${Math.random().toString(36).substr(2, 9)}`;

const ws = new WebSocket(`ws://localhost:8001/ws/${clientId}`);

**Proxied Connection (Through Vite at Port 3000)**

const clientId = `dex\_${Date.now()}\_${Math.random().toString(36).substr(2, 9)}`;

const ws = new WebSocket(`ws://localhost:3000/ws/${clientId}`);

**Complete Example: Connect → Subscribe → Receive**

class DEXWebSocketClient {

constructor() {

this.clientId = `dex\_${Date.now()}\_${Math.random().toString(36).substr(2, 9)}`;

this.ws = null;

this.reconnectCount = 0;

}

connect() {

// Use direct or proxied URL based on environment

const wsUrl = process.env.NODE\_ENV === 'development'

? `ws://localhost:3000/ws/${this.clientId}` // Proxied through Vite

: `wss://yourdomain.com/ws/${this.clientId}`; // Direct in production

this.ws = new WebSocket(wsUrl);

this.ws.onopen = (event) => {

console.log('✅ WebSocket connected');

this.reconnectCount = 0;

// Subscribe to autotrade channel

this.subscribe('autotrade');

};

this.ws.onmessage = (event) => {

try {

const message = JSON.parse(event.data);

this.handleMessage(message);

} catch (error) {

console.error('Failed to parse WebSocket message:', error);

}

};

this.ws.onclose = (event) => {

console.log(`❌ WebSocket closed: ${event.code} - ${event.reason}`);

if (event.code !== 1000) { // Not a normal closure

this.reconnect();

}

};

this.ws.onerror = (error) => {

console.error('🚨 WebSocket error:', error);

};

}

subscribe(channel) {

if (this.ws?.readyState !== WebSocket.OPEN) return;

const message = {

id: Date.now().toString(),

type: 'subscription\_ack',

channel: 'system',

data: {

action: 'subscribe',

channel: channel

},

timestamp: new Date().toISOString(),

client\_id: this.clientId

};

this.ws.send(JSON.stringify(message));

console.log(`📤 Subscribed to ${channel} channel`);

}

handleMessage(message) {

switch (message.type) {

case 'connection\_ack':

console.log('🤝 Connection acknowledged:', message.data);

break;

case 'subscription\_ack':

console.log('✅ Subscription confirmed:', message.data);

break;

case 'engine\_status':

console.log('⚙️ Engine status update:', message.data);

// Update UI with engine status

break;

case 'opportunity\_found':

console.log('💎 New opportunity:', message.data);

// Show opportunity notification

break;

case 'trade\_executed':

console.log('💰 Trade executed:', message.data);

// Update positions/portfolio

break;

case 'error':

console.error('❌ Server error:', message.data);

// Handle error appropriately

break;

case 'heartbeat':

// Optionally respond to heartbeat

if (message.data.ping) {

this.sendHeartbeat();

}

break;

default:

console.log('📨 Unhandled message:', message);

}

}

sendHeartbeat() {

if (this.ws?.readyState !== WebSocket.OPEN) return;

const response = {

id: Date.now().toString(),

type: 'heartbeat',

channel: 'system',

data: { pong: true },

timestamp: new Date().toISOString(),

client\_id: this.clientId

};

this.ws.send(JSON.stringify(response));

}

reconnect() {

const delays = [1000, 2000, 4000, 8000, 16000];

const delay = delays[Math.min(this.reconnectCount, delays.length - 1)];

console.log(`🔄 Reconnecting in ${delay}ms (attempt ${this.reconnectCount + 1})`);

setTimeout(() => {

this.reconnectCount++;

this.connect();

}, delay);

}

disconnect() {

if (this.ws) {

this.ws.close(1000, 'Client disconnect');

this.ws = null;

}

}

}

// Usage

const wsClient = new DEXWebSocketClient();

wsClient.connect();

// Cleanup on page unload

window.addEventListener('beforeunload', () => {

wsClient.disconnect();

});

**Helper: WebSocket URL Generator**

Create this in frontend/src/utils/websocket.js:

/\*\*

\* Generate appropriate WebSocket URL for current environment.

\*

\* @param {string} clientId - Unique client identifier

\* @returns {string} WebSocket URL

\*/

export function getWebSocketUrl(clientId) {

if (typeof window === 'undefined') {

throw new Error('WebSocket URLs can only be generated in browser environment');

}

// In development, use Vite proxy

if (import.meta.env.DEV) {

const protocol = window.location.protocol === 'https:' ? 'wss:' : 'ws:';

return `${protocol}//${window.location.host}/ws/${clientId}`;

}

// In production, use direct backend connection

const protocol = window.location.protocol === 'https:' ? 'wss:' : 'ws:';

const backendHost = import.meta.env.VITE\_BACKEND\_HOST || window.location.hostname;

const backendPort = import.meta.env.VITE\_BACKEND\_PORT || '8001';

return `${protocol}//${backendHost}:${backendPort}/ws/${clientId}`;

}

**Testing**

Use the built-in test page at http://localhost:8001/ws/test to verify WebSocket functionality during development.