# libwsv5

Generated by Doxygen 1.9.4

# Chapter 1

# Data Structure Index

## 1.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 2

# File Index

## 2.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 3

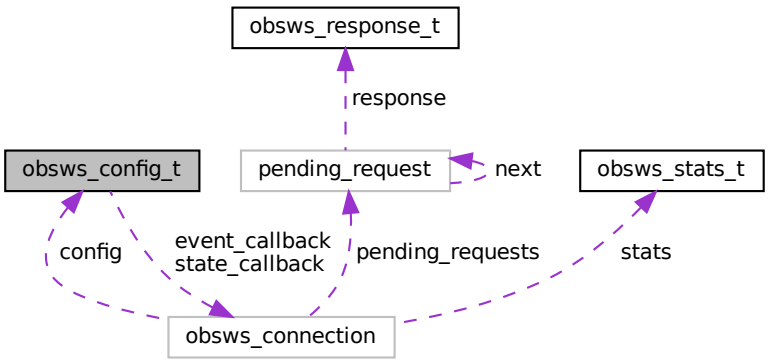
# Data Structure Documentation

## 3.1 obsws_config_t Struct Reference

Connection configuration structure.

`#include <libwsv5.h>`

Collaboration diagram for obsws_config_t:




### Data Fields

- const char * host
- uint16_t port
- const char * password
- bool use_ssl
- uint32_t connect_timeout_ms
- uint32_t recv_timeout_ms
- uint32_t send_timeout_ms
- uint32_t ping_interval_ms
- uint32_t ping_timeout_ms

- bool auto_reconnect
- uint32_t reconnect_delay_ms
- uint32_t max_reconnect_delay_ms
- uint32_t max_reconnect_attempts
- obsws_log_callback_t log_callback
- obsws_event_callback_t event_callback
- obsws_state_callback_t state_callback
- void * user_data

### 3.1.1 Detailed Description

Connection configuration structure.

This structure holds all the settings for connecting to OBS. You should fill this out with your specific needs, then pass it to obsws_connect(). A good starting point is to call obsws_config_init() which fills it with reasonable defaults, then only change the fields you care about (usually just host, port, and password).

Design note: We use a config struct instead of many function parameters because it's more flexible - adding new configuration options doesn't break existing code. It also makes it clear what options are available.

Definition at line 242 of file libwsv5.h.

### 3.1.2 Field Documentation

#### 3.1.2.1 auto_reconnect

```
bool obsws_config_t::auto_reconnect
```

Definition at line 266 of file libwsv5.h.

#### 3.1.2.2 connect_timeout_ms

```
uint32_t obsws_config_t::connect_timeout_ms
```

Definition at line 252 of file libwsv5.h.

#### 3.1.2.3 event_callback

```
obsws_event_callback_t obsws_config_t::event_callback
```

Definition at line 275 of file libwsv5.h.

#### 3.1.2.4 host

```
const char* obsws_config_t::host
```

Definition at line 244 of file libwsv5.h.

#### 3.1.2.5 log_callback

```
obsws_log_callback_t obsws_config_t::log_callback
```

Definition at line 274 of file libwsv5.h.

#### 3.1.2.6 max_reconnect_attempts

```
uint32_t obsws_config_t::max_reconnect_attempts
```

Definition at line 269 of file libwsv5.h.

#### 3.1.2.7 max_reconnect_delay_ms

```
uint32_t obsws_config_t::max_reconnect_delay_ms
```

Definition at line 268 of file libwsv5.h.

#### 3.1.2.8 password

```
const char* obsws_config_t::password
```

Definition at line 246 of file libwsv5.h.

#### 3.1.2.9 ping_interval_ms

```
uint32_t obsws_config_t::ping_interval_ms
```

Definition at line 259 of file libwsv5.h.

#### 3.1.2.10 ping_timeout_ms

```
uint32_t obsws_config_t::ping_timeout_ms
```

Definition at line 260 of file libwsv5.h.

#### 3.1.2.11 port

```
uint16_t obsws_config_t::port
```

Definition at line 245 of file libwsv5.h.

#### 3.1.2.12 reconnect_delay_ms

```
uint32_t obsws_config_t::reconnect_delay_ms
```

Definition at line 267 of file libwsv5.h.

#### 3.1.2.13 recv_timeout_ms

```
uint32_t obsws_config_t::recv_timeout_ms
```

Definition at line 253 of file libwsv5.h.

#### 3.1.2.14 send_timeout_ms

```
uint32_t obsws_config_t::send_timeout_ms
```

Definition at line 254 of file libwsv5.h.

#### 3.1.2.15 state_callback

```
obsws_state_callback_t obsws_config_t::state_callback
```

Definition at line 276 of file libwsv5.h.

#### 3.1.2.16 use_ssl

```
bool obsws_config_t::use_ssl
```

Definition at line 247 of file libwsv5.h.

#### 3.1.2.17 user_data

```
void* obsws_config_t::user_data
```

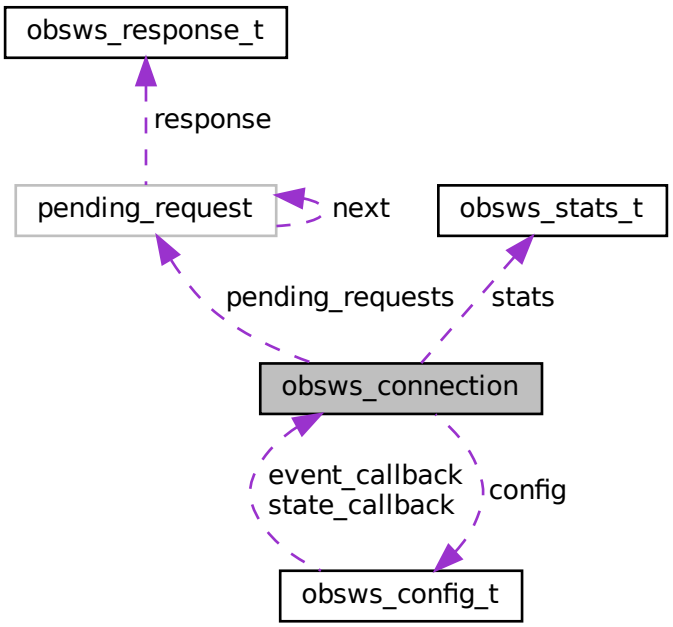
Definition at line 277 of file libwsv5.h.

The documentation for this struct was generated from the following file:

- libwsv5.h

## 3.2 obsws_connection Struct Reference

Collaboration diagram for obsws_connection:



## Data Fields

- obsws_config_t config
- obsws_state_t state
- pthread_mutex_t state_mutex
- struct lws_context * lws_context
- struct lws * wsi
- char * recv_buffer
- size_t recv_buffer_size
- size_t recv_buffer_used
- char * send_buffer
- size_t send_buffer_size
- pthread_t event_thread
- pthread_mutex_t send_mutex
- bool thread_running
- bool should_exit
- pending_request_t * pending_requests
- pthread_mutex_t requests_mutex
- obsws_stats_t stats
- pthread_mutex_t stats_mutex
- time_t last_ping_sent
- time_t last_pong_received
- uint32_t reconnect_attempts
- uint32_t current_reconnect_delay
- bool auth_required
- char * challenge
- char * salt
- char * current_scene
- pthread_mutex_t scene_mutex

### 3.2.1 Detailed Description

Definition at line 184 of file libwsv5.c.

### 3.2.2 Field Documentation

#### 3.2.2.1 auth_required

```
bool obsws_connection::auth_required
```

Definition at line 239 of file libwsv5.c.

#### 3.2.2.2 challenge

```
char* obsws_connection::challenge
```

Definition at line 240 of file libwsv5.c.

#### 3.2.2.3 config

```
obsws_config_t obsws_connection::config
```

Definition at line 186 of file libwsv5.c.

#### 3.2.2.4 current_reconnect_delay

```
uint32_t obsws_connection::current_reconnect_delay
```

Definition at line 234 of file libwsv5.c.

#### 3.2.2.5 current_scene

```
char* obsws_connection::current_scene
```

Definition at line 246 of file libwsv5.c.

#### 3.2.2.6 event_thread

```
pthread_t obsws_connection::event_thread
```

Definition at line 209 of file libwsv5.c.

#### 3.2.2.7 last_ping_sent

```
time_t obsws_connection::last_ping_sent
```

Definition at line 227 of file libwsv5.c.

#### 3.2.2.8 last_pong_received

```
time_t obsws_connection::last_pong_received
```

Definition at line 228 of file libwsv5.c.

#### 3.2.2.9 lws_context

```
struct lws_context* obsws_connection::lws_context
```

Definition at line 193 of file libwsv5.c.

#### 3.2.2.10 pending_requests

```
pending_request_t* obsws_connection::pending_requests
```

Definition at line 217 of file libwsv5.c.

#### 3.2.2.11 reconnect_attempts

```
uint32_t obsws_connection::reconnect_attempts
```

Definition at line 233 of file libwsv5.c.

#### 3.2.2.12 recv_buffer

```
char* obsws_connection::recv_buffer
```

Definition at line 199 of file libwsv5.c.

#### 3.2.2.13 recv_buffer_size

```
size_t obsws_connection::recv_buffer_size
```

Definition at line 200 of file libwsv5.c.

#### 3.2.2.14 recv_buffer_used

```
size_t obsws_connection::recv_buffer_used
```

Definition at line 201 of file libwsv5.c.

#### 3.2.2.15 requests_mutex

```
pthread_mutex_t obsws_connection::requests_mutex
```

Definition at line 218 of file libwsv5.c.

#### 3.2.2.16 salt

```
char* obsws_connection::salt
```

Definition at line 241 of file libwsv5.c.

#### 3.2.2.17 scene_mutex

```
pthread_mutex_t obsws_connection::scene_mutex
```

Definition at line 247 of file libwsv5.c.

#### 3.2.2.18 send_buffer

```
char* obsws_connection::send_buffer
```

Definition at line 203 of file libwsv5.c.

#### 3.2.2.19 send_buffer_size

```
size_t obsws_connection::send_buffer_size
```

Definition at line 204 of file libwsv5.c.

#### 3.2.2.20 send_mutex

```
pthread_mutex_t obsws_connection::send_mutex
```

Definition at line 210 of file libwsv5.c.

#### 3.2.2.21 should_exit

```
bool obsws_connection::should_exit
```

Definition at line 212 of file libwsv5.c.

#### 3.2.2.22 state

```
obsws_state_t obsws_connection::state
```

Definition at line 189 of file libwsv5.c.

#### 3.2.2.23 state_mutex

```
pthread_mutex_t obsws_connection::state_mutex
```

Definition at line 190 of file libwsv5.c.

#### 3.2.2.24 stats

```
obsws_stats_t obsws_connection::stats
```

Definition at line 221 of file libwsv5.c.

#### 3.2.2.25 stats_mutex

```
pthread_mutex_t obsws_connection::stats_mutex
```

Definition at line 222 of file libwsv5.c.

#### 3.2.2.26 thread_running

```
bool obsws_connection::thread_running
```

Definition at line 211 of file libwsv5.c.

#### 3.2.2.27 wsi

```
struct lws* obsws_connection::wsi
```

Definition at line 194 of file libwsv5.c.

The documentation for this struct was generated from the following file:

- libwsv5.c

## 3.3 obsws_log_context_t Struct Reference

### Data Fields

- bool enabled
- char log_directory [PATH_MAX]
- FILE * current_file
- char current_filename [PATH_MAX]
- time_t current_file_date
- time_t last_rotation_check
- int rotation_hour
- size_t max_file_size
- int color_mode
- bool use_timestamps
- bool is_tty
- pthread_mutex_t mutex

### 3.3.1 Detailed Description

Definition at line 276 of file libwsv5.c.

### 3.3.2 Field Documentation

#### 3.3.2.1 color_mode

```
int obsws_log_context_t::color_mode
```

Definition at line 287 of file libwsv5.c.

#### 3.3.2.2 current_file

```
FILE* obsws_log_context_t::current_file
```

Definition at line 279 of file libwsv5.c.

#### 3.3.2.3 current_file_date

```
time_t obsws_log_context_t::current_file_date
```

Definition at line 281 of file libwsv5.c.

#### 3.3.2.4 current_filename

```
char obsws_log_context_t::current_filename[PATH_MAX]
```

Definition at line 280 of file libwsv5.c.

#### 3.3.2.5 enabled

```
bool obsws_log_context_t::enabled
```

Definition at line 277 of file libwsv5.c.

#### 3.3.2.6 is_tty

```
bool obsws_log_context_t::is_tty
```

Definition at line 289 of file libwsv5.c.

#### 3.3.2.7 last_rotation_check

```
time_t obsws_log_context_t::last_rotation_check
```

Definition at line 282 of file libwsv5.c.

#### 3.3.2.8 log_directory

```
char obsws_log_context_t::log_directory[PATH_MAX]
```

Definition at line 278 of file libwsv5.c.

#### 3.3.2.9 max_file_size

```
size_t obsws_log_context_t::max_file_size
```

Definition at line 285 of file libwsv5.c.

#### 3.3.2.10 mutex

```
pthread_mutex_t obsws_log_context_t::mutex
```

Definition at line 291 of file libwsv5.c.

#### 3.3.2.11 rotation_hour

```
int obsws_log_context_t::rotation_hour
```

Definition at line 284 of file libwsv5.c.

#### 3.3.2.12 use_timestamps

```
bool obsws_log_context_t::use_timestamps
```

Definition at line 288 of file libwsv5.c.

The documentation for this struct was generated from the following file:

- libwsv5.c

## 3.4 obsws_response_t Struct Reference

Response structure for requests to OBS.

`#include <libwsv5.h>`

### Data Fields

- bool success
- int status_code
- char * error_message
- char * response_data

### 3.4.1 Detailed Description

Response structure for requests to OBS.

When you send a request like obsws_set_current_scene(), you can get back a response with the result. The response tells you if it succeeded, and if not, why it failed. It might also contain response data from OBS - for example, obsws_get_current_scene() puts the scene name in response_data as JSON.

If you don't care about the response, you can pass NULL and not get one back. Otherwise you must free it with obsws_response_free() when done.

Design note: Responses are returned as strings instead of parsed JSON to save CPU - different callers care about different response fields, so we let them parse what they need. This also avoids dependency bloat.

Definition at line 314 of file libwsv5.h.

### 3.4.2 Field Documentation

#### 3.4.2.1 error_message

`char* obsws_response_t::error_message`

Definition at line 317 of file libwsv5.h.

#### 3.4.2.2 response_data

`char* obsws_response_t::response_data`

Definition at line 318 of file libwsv5.h.

#### 3.4.2.3 status_code

```
int obsws_response_t::status_code
```

Definition at line 316 of file libwsv5.h.

#### 3.4.2.4 success

```
bool obsws_response_t::success
```

Definition at line 315 of file libwsv5.h.

The documentation for this struct was generated from the following file:

- libwsv5.h

## 3.5 obsws_stats_t Struct Reference

Connection statistics - useful for monitoring and debugging connection health.

```
#include <libwsv5.h>
```

### Data Fields

- uint64_t messages_sent
- uint64_t messages_received
- uint64_t bytes_sent
- uint64_t bytes_received
- uint64_t reconnect_count
- uint64_t error_count
- uint64_t last_ping_ms
- time_t connected_since

### 3.5.1 Detailed Description

Connection statistics - useful for monitoring and debugging connection health.

These stats let you see what's happening on the connection - how many messages have been sent/received, error counts, latency, etc. Useful for monitoring the connection quality, detecting if something is wrong, or just being curious about protocol activity. You get these by calling obsws_get_stats().

Definition at line 288 of file libwsv5.h.

### 3.5.2 Field Documentation

#### 3.5.2.1 bytes_received

```
uint64_t obsws_stats_t::bytes_received
```

Definition at line 292 of file libwsv5.h.

#### 3.5.2.2 bytes_sent

```
uint64_t obsws_stats_t::bytes_sent
```

Definition at line 291 of file libwsv5.h.

#### 3.5.2.3 connected_since

```
time_t obsws_stats_t::connected_since
```

Definition at line 296 of file libwsv5.h.

#### 3.5.2.4 error_count

```
uint64_t obsws_stats_t::error_count
```

Definition at line 294 of file libwsv5.h.

#### 3.5.2.5 last_ping_ms

```
uint64_t obsws_stats_t::last_ping_ms
```

Definition at line 295 of file libwsv5.h.

#### 3.5.2.6 messages_received

```
uint64_t obsws_stats_t::messages_received
```

Definition at line 290 of file libwsv5.h.

#### 3.5.2.7 messages_sent

```
uint64_t obsws_stats_t::messages_sent
```

Definition at line 289 of file libwsv5.h.

#### 3.5.2.8 reconnect_count

```
uint64_t obsws_stats_t::reconnect_count
```

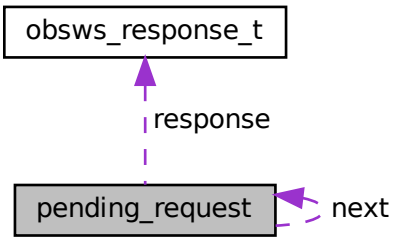
Definition at line 293 of file libwsv5.h.

The documentation for this struct was generated from the following file:

- libwsv5.h

## 3.6 pending_request Struct Reference

Collaboration diagram for pending_request:




### Data Fields

- char request_id [OBSWS_UUID_LENGTH]
- obsws_response_t * response
- bool completed
- pthread_mutex_t mutex
- pthread_cond_t cond
- time_t timestamp
- struct pending_request * next

### 3.6.1 Detailed Description

Definition at line 142 of file libwsv5.c.

### 3.6.2 Field Documentation

#### 3.6.2.1 completed

```
bool pending_request::completed
```

Definition at line 145 of file libwsv5.c.

#### 3.6.2.2 cond

```
pthread_cond_t pending_request::cond
```

Definition at line 147 of file libwsv5.c.

#### 3.6.2.3 mutex

```
pthread_mutex_t pending_request::mutex
```

Definition at line 146 of file libwsv5.c.

#### 3.6.2.4 next

```
struct pending_request* pending_request::next
```

Definition at line 149 of file libwsv5.c.

#### 3.6.2.5 request_id

```
char pending_request::request_id[OBSWS_UUID_LENGTH]
```

Definition at line 143 of file libwsv5.c.

#### 3.6.2.6 response

```
obsws_response_t* pending_request::response
```

Definition at line 144 of file libwsv5.c.

#### 3.6.2.7 timestamp

```
time_t pending_request::timestamp
```

Definition at line 148 of file libwsv5.c.

The documentation for this struct was generated from the following file:

- libwsv5.c

# Chapter 4

# File Documentation

## 4.1 libwsv5.c

```
00001 /*
00002 * libwsv5.c - OBS WebSocket v5 Protocol Library Implementation
00003 *
00004 * A robust C library for managing OBS connections via the WebSocket v5 protocol.
00005 * Supports multiple concurrent connections, automatic reconnection, and thread-safe
00006 * operations for streaming, recording, scene control, and live production setups.
00007 *
00008 * Author:  Aidan A. Bradley
00009 * Maintainer:  Aidan A. Bradley
00010 * License:  MIT
00011 */
00012
00013 #define _POSIX_C_SOURCE 200809L
00014
00015 #include "libwsv5.h"
00016
00017 #include <stdio.h>
00018 #include <stdlib.h>
00019 #include <string.h>
00020 #include <time.h>
00021 #include <pthread.h>
00022 #include <unistd.h>
00023 #include <stdarg.h>
00024 #include <sys/socket.h>
00025 #include <sys/select.h>
00026 #include <sys/stat.h>
00027 #include <sys/types.h>
00028 #include <netinet/in.h>
00029 #include <arpa/inet.h>
00030 #include <netdb.h>
00031 #include <errno.h>
00032 #include <poll.h>
00033 #include <limits.h>
00034
00035 /* Third-party dependencies */
00036 #include <libwebsockets.h>
00037 #include <openssl/sha.h>
00038 #include <openssl/evp.h>
00039 #include <openssl/bio.h>
00040 #include <openssl/buffer.h>
00041 #include <cjson/cJSON.h>
00042
00043 /* ============================================================================
00044 * Constants and Macros
00045 * ============================================================================ */
00046
00047 #define OBSWS_VERSION "1.0.0"                   /* Library version string */
00048 #define OBSWS_PROTOCOL_VERSION 1                /* OBS WebSocket protocol version (v5 uses RPC version
      1) */
00049
00050 /* Buffer sizing:  64KB is large enough for most OBS messages.  Larger messages
00051 (like scene lists with many scenes) might need bigger buffers, but this is
00052 a reasonable default.  We could make it dynamic, but that adds complexity.
00053 The protocol itself doesn't define a max message size, so we have to choose.  */
00054 #define OBSWS_DEFAULT_BUFFER_SIZE 65536         /* 64KB buffer for WebSocket messages */
00055
00056 /* Pending requests tracking:  We use a linked list to track requests waiting for
00057 responses.  256 is a reasonable limit - you can have up to 256 requests in-flight
```

```
00058 at once.  In practice, most apps will have way fewer.  We chose a limit to prevent
00059 unbounded memory growth if something goes wrong and requests never complete.  */
00060 #define OBSWS_MAX_PENDING_REQUESTS 256
00061
00062 /* UUIDs are exactly 36 characters (8-4-4-4-12 hex digits with dashes) plus null terminator.
00063 We use these to match requests with their responses in the asynchronous protocol.  */
00064 #define OBSWS_UUID_LENGTH 37
00065
00066 /* OBS WebSocket v5 OpCodes - message type identifiers in the protocol.
00067
00068 The OBS WebSocket v5 protocol uses opcodes to identify message types.  The protocol
00069 is based on a request-response model layered on top of WebSocket.  Here's the flow:
00070
00071 1.  Server sends HELLO (opcode 0) with auth challenge and salt
00072 2.  Client sends IDENTIFY (opcode 1) with auth response and client info
00073 3.  Server sends IDENTIFIED (opcode 2) if auth succeeded
00074 4.  Client can send REQUEST messages (opcode 6)
00075 5.  Server responds with REQUEST_RESPONSE (opcode 7)
00076 6.  Server sends EVENT messages (opcode 5) for things happening in OBS
00077
00078 Batch operations (opcodes 8-9) let you send multiple requests in one message,
00079 but we don't use them in this library - each request is sent individually.
00080 REIDENTIFY (opcode 3) is used if we lose connection and reconnect.
00081 */
00082
00083 #define OBSWS_OPCODE_HELLO 0                    /* Server:  Initial greeting with auth info */
00084 #define OBSWS_OPCODE_IDENTIFY 1                 /* Client:  Authentication and protocol agreement */
00085 #define OBSWS_OPCODE_IDENTIFIED 2               /* Server:  Auth successful, ready for commands */
00086 #define OBSWS_OPCODE_REIDENTIFY 3               /* Client:  Re-authenticate after reconnect */
00087 #define OBSWS_OPCODE_EVENT 5                    /* Server:  Something happened in OBS */
00088 #define OBSWS_OPCODE_REQUEST 6                  /* Client:  Execute an operation in OBS */
00089 #define OBSWS_OPCODE_REQUEST_RESPONSE 7         /* Server:  Result of a client request */
00090 #define OBSWS_OPCODE_REQUEST_BATCH 8            /* Client:  Multiple requests at once (unused) */
00091 #define OBSWS_OPCODE_REQUEST_BATCH_RESPONSE 9   /* Server:  Responses to batch (unused) */
00092
00093 /* Event subscription flags - bitmask for which OBS event categories we subscribe to.
00094
00095 The OBS WebSocket protocol lets you specify which events you want to receive.  This
00096 avoids bandwidth waste - if you don't care about media playback events, don't subscribe.
00097 We subscribe to most categories by default (using a bitmask), but you could modify
00098 this to be more selective if needed.
00099
00100 We chose a bitmask (0x7FF for all) rather than subscribing/unsubscribing individually
00101 because it's more efficient - one subscription message at connect-time instead of
00102 many individual subscribe/unsubscribe messages.
00103 */
00104
00105 #define OBSWS_EVENT_GENERAL (1 << 0)        /* General OBS events (startup, shutdown) */
00106 #define OBSWS_EVENT_CONFIG (1 << 1)         /* Configuration change events */
00107 #define OBSWS_EVENT_SCENES (1 << 2)         /* Scene-related events (scene switched, etc) */
00108 #define OBSWS_EVENT_INPUTS (1 << 3)         /* Input source events (muted, volume changed) */
00109 #define OBSWS_EVENT_TRANSITIONS (1 << 4)    /* Transition events (transition started) */
00110 #define OBSWS_EVENT_FILTERS (1 << 5)        /* Filter events (filter added, removed) */
00111 #define OBSWS_EVENT_OUTPUTS (1 << 6)        /* Output events (recording started, streaming stopped) */
00112 #define OBSWS_EVENT_SCENE_ITEMS (1 << 7)    /* Scene item events (source added to scene) */
00113 #define OBSWS_EVENT_MEDIA_INPUTS (1 << 8)   /* Media playback events (media finished) */
00114 #define OBSWS_EVENT_VENDORS (1 << 9)        /* Vendor-specific extensions */
00115 #define OBSWS_EVENT_UI (1 << 10)            /* UI events (Studio Mode toggled) */
00116 #define OBSWS_EVENT_ALL 0x7FF                /* Subscribe to all event types */
00117
00118 /* ============================================================================
00119 * Internal Structures
00120 * ============================================================================ */
00121
00122 /* Pending request tracking - manages asynchronous request/response pairs.
00123
00124 The OBS WebSocket protocol is asynchronous - when you send a request, you don't
00125 wait for the response before continuing.  Instead, responses come back later with
00126 a request ID matching them to the original request.
00127
00128 This struct tracks one in-flight request.  We keep a linked list of these, one for
00129 each request waiting for a response.  When a response arrives, we find the matching
00130 pending_request by ID, populate the response field, and set completed=true.  The
00131 thread that sent the request is waiting on the condition variable, so it wakes up
00132 and gets the response.
00133
00134 Why use a condition variable instead of polling?  Because polling wastes CPU. A
00135 thread waiting on a condition variable goes to sleep until the response arrives,
00136 at which point it's woken up.  Much more efficient.
00137
00138 Why use a timestamp?  For timeout detection.  If a response never arrives (OBS crashed,
00139 network died, etc.), we detect it by checking if the request is older than the timeout.
00140 */
00141
00142 typedef struct pending_request {
00143     char request_id[OBSWS_UUID_LENGTH];     /* Unique ID matching request to response */
00144     obsws_response_t *response;             /* Response data populated when received */
```

```
00145     bool completed;                         /* Flag indicating response received */
00146     pthread_mutex_t mutex;                  /* Protects the response/completed fields */
00147     pthread_cond_t cond;                    /* Waiting thread sleeps here until response arrives */
00148     time_t timestamp;                       /* When request was created - used for timeout detection
    */
00149     struct pending_request *next;           /* Linked list pointer to next pending request */
00150 } pending_request_t;
00151
00152 /* Main connection structure - holds all state for an OBS WebSocket connection.
00153
00154 This is the main opaque type that users interact with.  It holds everything needed
00155 to manage one connection to OBS:
00156 - Configuration (where to connect, timeouts, callbacks)
00157 - WebSocket instance (from libwebsockets)
00158 - Threading state (the event thread runs in the background)
00159 - Buffers for sending/receiving messages
00160 - Pending request tracking (for async request/response)
00161 - Statistics (for monitoring)
00162 - Authentication state (challenge/salt for password auth)
00163
00164 Why is it opaque (hidden in the .c file)?  So we can change the internal structure
00165 without breaking the API. Callers just use the pointer, they don't know what's inside.
00166
00167 Threading model:  Each connection has one background thread (event_thread) that
00168 processes WebSocket events, calls callbacks, etc.  The main application thread sends
00169 requests and gets responses.  This avoids the app freezing while waiting for responses.
00170
00171 Synchronization:  We use many mutexes because different parts of the connection
00172 are accessed from different threads:
00173 - state_mutex protects the connection state (so both threads see consistent state)
00174 - send_mutex protects sending (prevents two threads from sending at the same time)
00175 - requests_mutex protects the pending requests list
00176 - stats_mutex protects the statistics counters
00177 - scene_mutex protects the cached current scene name
00178
00179 The scene cache is an optimization - some operations need to know the current
00180 scene.  Instead of querying OBS every time, we cache it and update when we get
00181 SceneChanged events.
00182 */
00183
00184 struct obsws_connection {
00185     /* === Configuration and Setup === */
00186     obsws_config_t config;                  /* User-provided config (copied at construction) */
00187
00188     /* === Connection State === */
00189     obsws_state_t state;                    /* Current state (CONNECTED, CONNECTING, etc) */
00190     pthread_mutex_t state_mutex;            /* Protects state from concurrent access */
00191
00192     /* === WebSocket Layer === */
00193     struct lws_context *lws_context;        /* libwebsockets context (manages the WebSocket) */
00194     struct lws *wsi;                        /* WebSocket instance - the actual connection */
00195
00196     /* === Message Buffers ===
00197 We keep persistent buffers instead of allocating for every message because
00198 it's more efficient and avoids memory fragmentation.  */
00199     char *recv_buffer;                      /* Buffer for incoming messages from OBS */
00200     size_t recv_buffer_size;                /* Total capacity of receive buffer */
00201     size_t recv_buffer_used;                /* How many bytes are currently in the buffer */
00202
00203     char *send_buffer;                      /* Buffer for outgoing messages to OBS */
00204     size_t send_buffer_size;                /* Total capacity of send buffer */
00205
00206     /* === Background Thread ===
00207 The event thread continuously processes WebSocket events.  This allows the
00208 connection to receive messages and call callbacks without blocking the app.  */
00209     pthread_t event_thread;                 /* ID of the background thread */
00210     pthread_mutex_t send_mutex;             /* Prevents two threads from sending simultaneously */
00211     bool thread_running;                    /* Is the thread currently running?  */
00212     bool should_exit;                       /* Signal to thread:  time to stop */
00213
00214     /* === Async Request/Response Handling ===
00215 When you send a request, it returns immediately with a request ID. When the
00216 response comes back, we find the pending_request by ID and notify the waiter.  */
00217     pending_request_t *pending_requests;    /* Linked list of in-flight requests */
00218     pthread_mutex_t requests_mutex;         /* Protects the linked list */
00219
00220     /* === Performance Monitoring === */
00221     obsws_stats_t stats;                    /* Message counts, errors, latency, etc */
00222     pthread_mutex_t stats_mutex;            /* Protects stats from concurrent access */
00223
00224     /* === Keep-Alive / Health Monitoring ===
00225 We send periodic pings to detect when the connection dies.  If we don't get
00226 a pong back within the timeout, we know something is wrong.  */
00227     time_t last_ping_sent;                  /* When we last sent a ping */
00228     time_t last_pong_received;              /* When we last got a pong back */
00229
00230     /* === Reconnection === */
```

```
00231 If the connection drops and auto_reconnect is enabled, we try to reconnect.
00232 We use exponential backoff - each attempt waits longer, up to a maximum.  */
00233     uint32_t reconnect_attempts;            /* How many times have we tried reconnecting */
00234     uint32_t current_reconnect_delay;       /* How long we're waiting before next attempt */
00235
00236     /* === Authentication State ===
00237 OBS uses a challenge-response authentication scheme.  The server sends a
00238 challenge and salt, we compute a response using SHA256, and send it back.  */
00239     bool auth_required;                     /* Does OBS need authentication?  */
00240     char *challenge;                        /* Challenge string from OBS HELLO */
00241     char *salt;                             /* Salt string from OBS HELLO */
00242
00243     /* === Optimization Cache ===
00244 We cache the current scene to avoid querying OBS unnecessarily.  When we get
00245 a SceneChanged event, we update the cache.  */
00246     char *current_scene;                    /* Cached name of active scene */
00247     pthread_mutex_t scene_mutex;            /* Protects the cache */
00248 };
00249
00250 /* ============================================================================
00251 * Global State
00252 * ============================================================================ */
00253
00254 /* Global initialization flag - tracks whether obsws_init() has been called.
00255
00256 Why have global state at all?  Some underlying libraries (like libwebsockets
00257 and OpenSSL) need one-time initialization.  We do that in obsws_init() and
00258 make sure it only happens once, even if called multiple times.  This flag
00259 tracks whether we've done it.
00260
00261 We use a mutex to protect the flag because someone might call obsws_init()
00262 from multiple threads simultaneously.  The mutex ensures only one thread does
00263 the initialization.
00264 */
00265
00266 static bool g_library_initialized = false;      /* Have we called the init code yet?  */
00267 static obsws_log_level_t g_log_level = OBSWS_LOG_INFO;  /* Global filtering level */
00268 static obsws_debug_level_t g_debug_level = OBSWS_DEBUG_NONE;  /* Global debug verbosity */
00269 static pthread_mutex_t g_init_mutex = PTHREAD_MUTEX_INITIALIZER;  /* Thread-safe initialization */
00270
00271 /* ============================================================================
00272 * Advanced Logging System - Global State
00273 * ============================================================================ */
00274
00275 /* Logging configuration context - manages file handles, rotation, colors, timestamps */
00276 typedef struct {
00277     bool enabled;                           /* Is file logging currently enabled?  */
00278     char log_directory[PATH_MAX];           /* Directory where log files are written */
00279     FILE *current_file;                     /* Current open log file handle */
00280     char current_filename[PATH_MAX];        /* Full path to current log file */
00281     time_t current_file_date;               /* Date when current file was created */
00282     time_t last_rotation_check;             /* Last time we checked for rotation */
00283
00284     int rotation_hour;                      /* Hour (0-23) when daily rotation occurs, -1 to disable
    */
00285     size_t max_file_size;                   /* Max size in bytes before rotation, 0 to disable */
00286
00287     int color_mode;                         /* 0=force_off, 1=force_on, 2=auto_detect */
00288     bool use_timestamps;                    /* Include timestamps in output?  */
00289     bool is_tty;                            /* Is output a TTY? (cached for efficiency) */
00290
00291     pthread_mutex_t mutex;                  /* Protects all fields above */
00292 } obsws_log_context_t;
00293
00294 static obsws_log_context_t g_log_ctx = {
00295     .enabled = false,
00296     .log_directory = {0},
00297     .current_file = NULL,
00298     .current_filename = {0},
00299     .current_file_date = 0,
00300     .last_rotation_check = 0,
00301     .rotation_hour = 0,                     /* Default:  rotate at midnight */
00302     .max_file_size = 0,                     /* Default:  disabled */
00303     .color_mode = 2,                        /* Default:  auto-detect */
00304     .use_timestamps = true,                 /* Default:  enabled */
00305     .is_tty = false,
00306     .mutex = PTHREAD_MUTEX_INITIALIZER
00307 };
00308
00309 /* ============================================================================
00310 * Logging Implementation
00311 * ============================================================================ */
00312
00313 /* Forward declarations for logging helper functions */
00314 static void obsws_log_rotate_if_needed(void);
00315 static int obsws_log_should_use_colors(void);
00316 static void obsws_log_get_timestamp(char *buf, size_t size);
```

```
00317
00322 static obsws_error_t obsws_log_create_directory(const char *path) {
00323     if (mkdir(path, 0700) == 0) {
00324         return OBSWS_OK;  /* Created successfully */
00325     }
00326
00327     if (errno == EEXIST) {
00328         return OBSWS_OK;  /* Already exists */
00329     }
00330
00331     /* Failed to create */
00332     return OBSWS_ERROR_CONNECTION_FAILED;
00333 }
00334
00338 static void obsws_log_get_default_directory(char *buf, size_t size) {
00339     const char *home = getenv("HOME");
00340     if (!home) {
00341         home = "/tmp";  /* Fallback if HOME not set */
00342     }
00343
00344     snprintf(buf, size, "%s/.config/libwsv5/logs", home);
00345 }
00346
00350 static void obsws_log_get_timestamp(char *buf, size_t size) {
00351     struct timespec ts;
00352     clock_gettime(CLOCK_REALTIME, &ts);
00353
00354     struct tm *tm_info = localtime(&ts.tv_sec);
00355     strftime(buf, size - 5, "[%Y-%m-%d %H:%M:%S", tm_info);
00356
00357     /* Add milliseconds */
00358     int ms = ts.tv_nsec / 1000000;
00359     size_t len = strlen(buf);
00360     snprintf(buf + len, size - len, ".%03d]", ms);
00361 }
00362
00366 static void obsws_log_get_date_str(char *buf, size_t size, time_t *date_out) {
00367     time_t now;
00368     time(&now);
00369     *date_out = now;
00370
00371     struct tm *tm_info = localtime(&now);
00372     strftime(buf, size, "%Y-%m-%d", tm_info);
00373 }
00374
00378 static int obsws_log_should_use_colors(void) {
00379     if (g_log_ctx.color_mode == 0) {
00380         return 0;  /* Force off */
00381     }
00382     if (g_log_ctx.color_mode == 1) {
00383         return 1;  /* Force on */
00384     }
00385     /* Auto-detect (mode == 2) */
00386     return g_log_ctx.is_tty;
00387 }
00388
00392 static FILE* obsws_log_open_file(void) {
00393     char date_str[11];  /* "YYYY-MM-DD" + null */
00394     time_t file_date;
00395     obsws_log_get_date_str(date_str, sizeof(date_str), &file_date);
00396
00397     /* Build filename:  libwsv5_YYYY-MM-DD.log */
00398     snprintf(g_log_ctx.current_filename, sizeof(g_log_ctx.current_filename),
00399              "%s/libwsv5_%s.log", g_log_ctx.log_directory, date_str);
00400
00401     /* Open file in append mode, create if doesn't exist */
00402     FILE *f = fopen(g_log_ctx.current_filename, "a");
00403     if (!f) {
00404         fprintf(stderr, "[OBSWS-ERROR] Failed to open log file:  %s (errno:  %d)\n",
00405                 g_log_ctx.current_filename, errno);
00406         return NULL;
00407     }
00408
00409     /* Set to line-buffered mode for better performance */
00410     setvbuf(f, NULL, _IOLBF, 0);
00411
00412     g_log_ctx.current_file_date = file_date;
00413     return f;
00414 }
00415
00419 static void obsws_log_rotate_if_needed(void) {
00420     if (!g_log_ctx.enabled || !g_log_ctx.current_file) {
00421         return;
00422     }
00423
00424     time_t now;
00425     time(&now);
```

```
00426
00427     /* Check daily rotation */
00428     int should_rotate = 0;
00429     if (g_log_ctx.rotation_hour >= 0) {
00430         struct tm *now_tm = localtime(&now);
00431         struct tm *file_tm = localtime(&g_log_ctx.current_file_date);
00432
00433         /* Rotate if:  different day, OR same day but we crossed rotation hour */
00434         if (now_tm->tm_yday != file_tm->tm_yday) {
00435             should_rotate = 1;
00436         }
00437     }
00438
00439     /* Check size-based rotation */
00440     if (!should_rotate && g_log_ctx.max_file_size > 0) {
00441         struct stat st;
00442         if (fstat(fileno(g_log_ctx.current_file), &st) == 0) {
00443             if ((size_t)st.st_size > g_log_ctx.max_file_size) {
00444                 should_rotate = 1;
00445             }
00446         }
00447     }
00448
00449     if (should_rotate) {
00450         fclose(g_log_ctx.current_file);
00451         g_log_ctx.current_file = obsws_log_open_file();
00452     }
00453 }
00454
00458 static void obsws_log_format_message(char *output, size_t out_size,
00459                                      obsws_log_level_t level,
00460                                      const char *message) {
00461     output[0] = '\0';
00462
00463     int use_colors = obsws_log_should_use_colors();
00464     const char *level_str[] = {"NONE", "ERROR", "WARN", "INFO", "DEBUG"};
00465     const char *color_codes[] = {
00466         "\x1b[0m",      /* NONE: reset */
00467         "\x1b[31m",     /* ERROR: red */
00468         "\x1b[33m",     /* WARNING: yellow */
00469         "\x1b[32m",     /* INFO: green */
00470         "\x1b[36m"      /* DEBUG: cyan */
00471     };
00472
00473     size_t pos = 0;
00474
00475     /* Add timestamp if enabled */
00476     if (g_log_ctx.use_timestamps) {
00477         char ts[32];
00478         obsws_log_get_timestamp(ts, sizeof(ts));
00479         pos += snprintf(output + pos, out_size - pos, "%s ", ts);
00480     }
00481
00482     /* Add colored level indicator */
00483     if (use_colors) {
00484         pos += snprintf(output + pos, out_size - pos, "%s[%s]\x1b[0m ",
00485                         color_codes[level], level_str[level]);
00486     } else {
00487         pos += snprintf(output + pos, out_size - pos, "[%s] ", level_str[level]);
00488     }
00489
00490     /* Add message */
00491     snprintf(output + pos, out_size - pos, "%s", message);
00492 }
00493
00494 /* ============================================================================
00495 * Public Logging Configuration API
00496 * ============================================================================ */
00497
00498 obsws_error_t obsws_enable_log_file(const char *directory) {
00499     pthread_mutex_lock(&g_log_ctx.mutex);
00500
00501     /* Use default directory if none provided */
00502     if (!directory) {
00503         obsws_log_get_default_directory(g_log_ctx.log_directory,
00504                                         sizeof(g_log_ctx.log_directory));
00505     } else {
00506         strncpy(g_log_ctx.log_directory, directory,
00507                 sizeof(g_log_ctx.log_directory) - 1);
00508         g_log_ctx.log_directory[sizeof(g_log_ctx.log_directory) - 1] = '\0';
00509     }
00510
00511     /* Create directory if it doesn't exist */
00512     obsws_error_t err = obsws_log_create_directory(g_log_ctx.log_directory);
00513     if (err != OBSWS_OK) {
00514         pthread_mutex_unlock(&g_log_ctx.mutex);
00515         return err;
```

```
00516     }
00517
00518     /* Close existing file if open */
00519     if (g_log_ctx.current_file) {
00520         fclose(g_log_ctx.current_file);
00521     }
00522
00523     /* Open new file */
00524     g_log_ctx.current_file = obsws_log_open_file();
00525     if (!g_log_ctx.current_file) {
00526         pthread_mutex_unlock(&g_log_ctx.mutex);
00527         return OBSWS_ERROR_CONNECTION_FAILED;
00528     }
00529
00530     g_log_ctx.enabled = true;
00531     pthread_mutex_unlock(&g_log_ctx.mutex);
00532
00533     return OBSWS_OK;
00534 }
00535
00536 obsws_error_t obsws_disable_log_file(void) {
00537     pthread_mutex_lock(&g_log_ctx.mutex);
00538
00539     if (g_log_ctx.current_file) {
00540         fflush(g_log_ctx.current_file);
00541         fclose(g_log_ctx.current_file);
00542         g_log_ctx.current_file = NULL;
00543     }
00544
00545     g_log_ctx.enabled = false;
00546     pthread_mutex_unlock(&g_log_ctx.mutex);
00547
00548     return OBSWS_OK;
00549 }
00550
00551 obsws_error_t obsws_set_log_rotation_hour(int hour) {
00552     if (hour < -1 || hour > 23) {
00553         return OBSWS_ERROR_INVALID_PARAM;
00554     }
00555
00556     pthread_mutex_lock(&g_log_ctx.mutex);
00557     g_log_ctx.rotation_hour = hour;
00558     pthread_mutex_unlock(&g_log_ctx.mutex);
00559
00560     return OBSWS_OK;
00561 }
00562
00563 obsws_error_t obsws_set_log_rotation_size(size_t max_size_bytes) {
00564     pthread_mutex_lock(&g_log_ctx.mutex);
00565     g_log_ctx.max_file_size = max_size_bytes;
00566     pthread_mutex_unlock(&g_log_ctx.mutex);
00567
00568     return OBSWS_OK;
00569 }
00570
00571 obsws_error_t obsws_set_log_colors(int mode) {
00572     if (mode < 0 || mode > 2) {
00573         return OBSWS_ERROR_INVALID_PARAM;
00574     }
00575
00576     pthread_mutex_lock(&g_log_ctx.mutex);
00577     g_log_ctx.color_mode = mode;
00578     pthread_mutex_unlock(&g_log_ctx.mutex);
00579
00580     return OBSWS_OK;
00581 }
00582
00583 obsws_error_t obsws_set_log_timestamps(bool enabled) {
00584     pthread_mutex_lock(&g_log_ctx.mutex);
00585     g_log_ctx.use_timestamps = enabled;
00586     pthread_mutex_unlock(&g_log_ctx.mutex);
00587
00588     return OBSWS_OK;
00589 }
00590
00591 const char* obsws_get_log_file_directory(void) {
00592     if (!g_log_ctx.enabled || g_log_ctx.log_directory[0] == '\0') {
00593         return NULL;
00594     }
00595     return g_log_ctx.log_directory;
00596 }
00597
00598 /* ============================================================================
00599 * Logging
00600 * ============================================================================ */
00601
00602 /* Internal logging function - core logging infrastructure.
```

```
00603
00604 Design:  We filter by log level (higher level = more verbose).  If the message
00605 is below the current level, we don't even format it (saves CPU). If there's a
00606 user-provided callback, we use it; otherwise we print to stderr.
00607
00608 Why two parameters (conn and format)?  So we can log from both the main thread
00609 (with a connection object) and the global initialization code (without one).
00610 */
00611
00612 static void obsws_log(obsws_connection_t *conn, obsws_log_level_t level, const char *format, ...)  {
00613     /* Early exit if this message is too verbose */
00614     if (level > g_log_level) {
00615         return;
00616     }
00617
00618     /* Format the message using printf-style arguments */
00619     char message[1024];
00620     va_list args;
00621     va_start(args, format);
00622     vsnprintf(message, sizeof(message), format, args);
00623     va_end(args);
00624
00625     /* Route to user callback first (if provided) */
00626     if (conn && conn->config.log_callback) {
00627         conn->config.log_callback(level, message, conn->config.user_data);
00628     }
00629
00630     /* Also handle advanced logging system (file, timestamps, colors, etc.)  */
00631     pthread_mutex_lock(&g_log_ctx.mutex);
00632
00633     /* Format message with advanced features */
00634     char formatted[2048];
00635     obsws_log_format_message(formatted, sizeof(formatted), level, message);
00636
00637     /* Write to file if enabled */
00638     if (g_log_ctx.enabled && g_log_ctx.current_file) {
00639         obsws_log_rotate_if_needed();
00640         fprintf(g_log_ctx.current_file, "%s\n", formatted);
00641         fflush(g_log_ctx.current_file);
00642     }
00643
00644     /* Write to console if no user callback (backward compat) */
00645     if (!conn || !conn->config.log_callback) {
00646         fprintf(stderr, "%s\n", formatted);
00647     }
00648
00649     pthread_mutex_unlock(&g_log_ctx.mutex);
00650 }
00651
00652 /* Debug logging - finer control for protocol-level troubleshooting.
00653
00654 Separate from regular logging because debug messages are very verbose and
00655 developers typically only enable them when debugging specific issues.  The
00656 debug level goes 0-3, with higher levels including all output from lower levels.
00657
00658 We use a larger buffer (4KB) because debug messages can include JSON payloads.
00659 */
00660
00661 static void obsws_debug(obsws_connection_t *conn, obsws_debug_level_t min_level, const char *format,
      ...)  {
00662     /* Only output if global debug level is at or above the minimum for this message */
00663     if (g_debug_level < min_level) {
00664         return;
00665     }
00666
00667     /* Format with a larger buffer for JSON and other verbose output */
00668     char message[4096];
00669     va_list args;
00670     va_start(args, format);
00671     vsnprintf(message, sizeof(message), format, args);
00672     va_end(args);
00673
00674     /* Route through the callback as DEBUG-level logs */
00675     if (conn && conn->config.log_callback) {
00676         conn->config.log_callback(OBSWS_LOG_DEBUG, message, conn->config.user_data);
00677     }
00678
00679     /* Also handle advanced logging system for debug messages */
00680     pthread_mutex_lock(&g_log_ctx.mutex);
00681
00682     const char *debug_level_str[] = {"NONE", "LOW", "MED", "HIGH"};
00683     char debug_msg[2048];
00684     snprintf(debug_msg, sizeof(debug_msg), "[DEBUG-%s] %s", debug_level_str[min_level], message);
00685
00686     char formatted[2560];
00687     obsws_log_format_message(formatted, sizeof(formatted), OBSWS_LOG_DEBUG, debug_msg);
00688
```

```
00689     /* Write to file if enabled */
00690     if (g_log_ctx.enabled && g_log_ctx.current_file) {
00691         obsws_log_rotate_if_needed();
00692         fprintf(g_log_ctx.current_file, "%s\n", formatted);
00693         fflush(g_log_ctx.current_file);
00694     }
00695 
00696     /* Write to console if no user callback (backward compat) */
00697     if (!conn || !conn->config.log_callback) {
00698         fprintf(stderr, "%s\n", formatted);
00699     }
00700 
00701     pthread_mutex_unlock(&g_log_ctx.mutex);
00702 }
00703 
00704 /* ============================================================================
00705 * Utility Functions
00706 * ============================================================================ */
00707 
00708 /* Generate a UUID v4 for request identification.
00709 
00710 UUIDs uniquely identify each request, so when a response comes back, we can match
00711 it to the original request.  We use UUID v4 (random) because it's simple and the
00712 uniqueness probability is astronomically high.
00713 
00714 Note:  This implementation uses rand() for simplicity.  A production system might
00715 use /dev/urandom for better randomness, but the current approach is fine for
00716 most use cases.  The protocol doesn't require cryptographically secure randomness.
00717 
00718 Format:  8-4-4-4-12 hex digits with dashes, exactly 36 characters.
00719 Example:  550e8400-e29b-41d4-a716-446655440000
00720 
00721 The version bits (0x4) and variant bits (0x8-b) mark this as a v4 UUID.
00722 */
00723 
00724 static void generate_uuid(char *uuid_out) {
00725     unsigned int r1 = rand();
00726     unsigned int r2 = rand();
00727     unsigned int r3 = rand();
00728     unsigned int r4 = rand();
00729 
00730     sprintf(uuid_out, "%08x-%04x-%04x-%04x-%04x%08x",
00731             r1,                          /* 8 hex digits */
00732             r2 & 0xFFFF,                 /* 4 hex digits */
00733             (r3 & 0x0FFF) | 0x4000,      /* 4 hex digits (set version 4) */
00734             (r4 & 0x3FFF) | 0x8000,      /* 4 hex digits (set variant bits) */
00735             rand() & 0xFFFF,             /* 4 hex digits */
00736             (unsigned int)rand());       /* 8 hex digits */
00737 }
00738 
00739 /* Base64 encode binary data using OpenSSL.
00740 
00741 Why base64 and not hex?  Hex would be twice as large.  Base64 is a standard
00742 encoding for binary data in text contexts (like WebSocket JSON messages).
00743 
00744 We use OpenSSL's BIO (Basic I/O) interface for encoding because it's robust
00745 and well-tested.  The BIO_FLAGS_BASE64_NO_NL flag removes newlines that OpenSSL
00746 normally adds for readability - we don't want those in JSON.
00747 */
00748 
00749 static char* base64_encode(const unsigned char *input, size_t length) {
00750     BIO *bio, *b64;
00751     BUF_MEM *buffer_ptr;
00752 
00753     /* Set up OpenSSL base64 encoder:  b64 filter pushing to memory buffer */
00754     b64 = BIO_new(BIO_f_base64());
00755     bio = BIO_new(BIO_s_mem());
00756     bio = BIO_push(b64, bio);
00757 
00758     /* Disable newlines in output (OpenSSL adds them by default for readability) */
00759     BIO_set_flags(bio, BIO_FLAGS_BASE64_NO_NL);
00760 
00761     /* Encode the data */
00762     BIO_write(bio, input, length);
00763     BIO_flush(bio);
00764     BIO_get_mem_ptr(bio, &buffer_ptr);
00765 
00766     /* Copy result to our own allocated buffer and null-terminate */
00767     char *result = malloc(buffer_ptr->length + 1);
00768     memcpy(result, buffer_ptr->data, buffer_ptr->length);
00769     result[buffer_ptr->length] = '\0';
00770 
00771     BIO_free_all(bio);
00772     return result;
00773 }
00774 
00775 /* Compute SHA256 hash of a null-terminated string.
```

```
00776
00777 SHA256 is a cryptographic hash function.  It's deterministic (same input always
00778 produces same output) and has an avalanche property (changing one bit in the
00779 input completely changes the output).  This makes it perfect for authentication
00780 protocols.
00781
00782 Why SHA256 instead of SHA1 or MD5?  SHA256 is current-best-practice.  SHA1 has
00783 known collisions, and MD5 is even more broken.  SHA256 is secure for the
00784 foreseeable future.
00785
00786 Why use EVP (Envelope) API instead of raw SHA256 functions?  EVP is higher-level
00787 and more flexible - if we ever need to support different hash algorithms, we
00788 just change one line.
00789 */
00790
00791 static void sha256_hash(const char *input, unsigned char *output) {
00792     EVP_MD_CTX *ctx = EVP_MD_CTX_new();
00793     EVP_DigestInit_ex(ctx, EVP_sha256(), NULL);
00794     EVP_DigestUpdate(ctx, input, strlen(input));
00795     EVP_DigestFinal_ex(ctx, output, NULL);
00796     EVP_MD_CTX_free(ctx);
00797 }
00798
00799 /* Generate OBS WebSocket v5 authentication response using challenge-response protocol.
00800
00801 OBS WebSocket v5 uses a two-step authentication protocol:
00802 1.  Server sends challenge + salt
00803 2.  Client computes:  secret = base64(sha256(password + salt))
00804 3.  Client computes:  response = base64(sha256(secret + challenge))
00805 4.  Client sends response
00806 5.  Server verifies by computing the same thing
00807
00808 Why this design?  The password never travels over the network.  Instead, a hash
00809 derived from the password (the secret) is combined with a fresh challenge each
00810 time, preventing replay attacks.  This is similar to HTTP Digest Authentication.
00811
00812 Why not use the password directly?  That would be incredibly insecure.  The
00813 two-step approach means an eavesdropper who sees the response can't use it
00814 again - the challenge was random and won't repeat.
00815 */
00816
00817 static char* generate_auth_response(const char *password, const char *salt, const char *challenge) {
00818     unsigned char secret_hash[SHA256_DIGEST_LENGTH];
00819     unsigned char auth_hash[SHA256_DIGEST_LENGTH];
00820
00821     /* Step 1:  Compute secret = base64(sha256(password + salt)) */
00822     char *password_salt = malloc(strlen(password) + strlen(salt) + 1);
00823     sprintf(password_salt, "%s%s", password, salt);
00824     sha256_hash(password_salt, secret_hash);
00825     free(password_salt);
00826
00827     char *secret = base64_encode(secret_hash, SHA256_DIGEST_LENGTH);
00828
00829     /* Step 2:  Compute auth response = base64(sha256(secret + challenge)) */
00830     char *secret_challenge = malloc(strlen(secret) + strlen(challenge) + 1);
00831     sprintf(secret_challenge, "%s%s", secret, challenge);
00832     sha256_hash(secret_challenge, auth_hash);
00833     free(secret_challenge);
00834     free(secret);
00835
00836     /* Return the final response, base64-encoded */
00837     char *auth_response = base64_encode(auth_hash, SHA256_DIGEST_LENGTH);
00838     return auth_response;
00839 }
00840
00841 /* ============================================================================
00842 * State Management
00843 * ============================================================================ */
00844
00845 /* Update connection state and notify callback if state changed.
00846
00847 This function is responsible for state transitions and notifying the user.
00848 We lock the mutex, make the change, unlock it, then call the callback without
00849 holding the lock.  Why release the lock before calling the callback?  Because
00850 the callback might take a long time, and we don't want to hold a lock during
00851 that time - it would prevent other threads from checking the state.
00852
00853 We only call the callback if the state actually changed.  This prevents spurious
00854 notifications if something tries to set the same state again.
00855 */
00856
00857 static void set_connection_state(obsws_connection_t *conn, obsws_state_t new_state) {
00858     /* Acquire lock, save old state, set new state, release lock */
00859     pthread_mutex_lock(&conn->state_mutex);
00860     obsws_state_t old_state = conn->state;
00861     conn->state = new_state;
00862     pthread_mutex_unlock(&conn->state_mutex);
```

```
00863
00864     /* Call callback only if state actually changed (not a duplicate) */
00865     if (old_state != new_state && conn->config.state_callback) {
00866         conn->config.state_callback(conn, old_state, new_state, conn->config.user_data);
00867     }
00868
00869     /* Log the transition for debugging/monitoring */
00870     obsws_log(conn, OBSWS_LOG_INFO, "State changed:  %s -> %s",
00871               obsws_state_string(old_state), obsws_state_string(new_state));
00872 }
00873
00874 /* ============================================================================
00875 * Request Management
00876 * ============================================================================ */
00877
00878 /* Create a new pending request and add it to the tracking list.
00879
00880 When we send a request to OBS, we need to track it so we can match the response
00881 when it arrives.  This function creates a pending_request_t struct and adds it
00882 to the linked list.  The request is initialized with the ID, a condition variable
00883 for waiting, and a current timestamp for timeout detection.
00884 */
00885
00886 static pending_request_t* create_pending_request(obsws_connection_t *conn, const char *request_id) {
00887     pending_request_t *req = calloc(1, sizeof(pending_request_t));
00888     if (!req) return NULL;
00889
00890     /* Copy request ID and ensure null termination */
00891     strncpy(req->request_id, request_id, OBSWS_UUID_LENGTH - 1);
00892     req->request_id[OBSWS_UUID_LENGTH - 1] = '\0';
00893
00894     /* Initialize request structure */
00895     req->response = calloc(1, sizeof(obsws_response_t));
00896     req->completed = false;
00897     req->timestamp = time(NULL);
00898     pthread_mutex_init(&req->mutex, NULL);
00899     pthread_cond_init(&req->cond, NULL);
00900
00901     /* Add to linked list of pending requests */
00902     pthread_mutex_lock(&conn->requests_mutex);
00903     req->next = conn->pending_requests;
00904     conn->pending_requests = req;
00905     pthread_mutex_unlock(&conn->requests_mutex);
00906
00907     return req;
00908 }
00909
00910 /* Find a pending request by its UUID */
00911 static pending_request_t* find_pending_request(obsws_connection_t *conn, const char *request_id) {
00912     pthread_mutex_lock(&conn->requests_mutex);
00913     pending_request_t *req = conn->pending_requests;
00914
00915     /* Search linked list for matching request ID */
00916     while (req) {
00917         if (strcmp(req->request_id, request_id) == 0) {
00918             pthread_mutex_unlock(&conn->requests_mutex);
00919             return req;
00920         }
00921         req = req->next;
00922     }
00923
00924     pthread_mutex_unlock(&conn->requests_mutex);
00925     return NULL;
00926 }
00927
00928 /* Remove a pending request from the tracking list and free it */
00929 static void remove_pending_request(obsws_connection_t *conn, pending_request_t *target) {
00930     pthread_mutex_lock(&conn->requests_mutex);
00931     pending_request_t **req = &conn->pending_requests;
00932
00933     /* Find and remove from linked list */
00934     while (*req) {
00935         if (*req == target) {
00936             *req = target->next;
00937             pthread_mutex_unlock(&conn->requests_mutex);
00938
00939             /* Clean up request resources */
00940             pthread_mutex_destroy(&target->mutex);
00941             pthread_cond_destroy(&target->cond);
00942             free(target);
00943             return;
00944         }
00945         req = &(*req)->next;
00946     }
00947
00948     pthread_mutex_unlock(&conn->requests_mutex);
00949 }
```

```
00950
00951 /* Clean up requests that have exceeded the timeout period */
00952 static void cleanup_old_requests(obsws_connection_t *conn) {
00953     time_t now = time(NULL);
00954     pthread_mutex_lock(&conn->requests_mutex);
00955
00956     pending_request_t **req = &conn->pending_requests;
00957     while (*req) {
00958         /* Check if request has timed out (30 seconds) */
00959         if (now - (*req)->timestamp > 30) {
00960             pending_request_t *old = *req;
00961             *req = old->next;
00962
00963             /* Mark as completed with timeout error */
00964             pthread_mutex_lock(&old->mutex);
00965             old->completed = true;
00966             old->response->success = false;
00967             old->response->error_message = strdup("Request timeout");
00968             pthread_cond_broadcast(&old->cond);  /* Wake waiting threads */
00969             pthread_mutex_unlock(&old->mutex);
00970         } else {
00971             req = &(*req)->next;
00972         }
00973     }
00974
00975     pthread_mutex_unlock(&conn->requests_mutex);
00976 }
00977
00978 /* ============================================================================
00979 * WebSocket Protocol Handling
00980 * ============================================================================ */
00981
01016 static int handle_hello_message(obsws_connection_t *conn, cJSON *data) {
01017     /* DEBUG_LOW: Basic connection event */
01018     obsws_debug(conn, OBSWS_DEBUG_LOW, "Received Hello message from OBS");
01019
01020     cJSON *auth = cJSON_GetObjectItem(data, "authentication");
01021     if (auth) {
01022         conn->auth_required = true;
01023
01024         cJSON *challenge = cJSON_GetObjectItem(auth, "challenge");
01025         cJSON *salt = cJSON_GetObjectItem(auth, "salt");
01026
01027         if (challenge && salt) {
01028             conn->challenge = strdup(challenge->valuestring);
01029             conn->salt = strdup(salt->valuestring);
01030             /* DEBUG_MEDIUM: Show auth parameters */
01031             obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Authentication required - salt:  %s, challenge:
      %s",
01032                         conn->salt, conn->challenge);
01033         }
01034     } else {
01035         conn->auth_required = false;
01036         obsws_debug(conn, OBSWS_DEBUG_LOW, "No authentication required");
01037     }
01038
01039     /* Send Identify message */
01040     set_connection_state(conn, OBSWS_STATE_AUTHENTICATING);
01041
01042     cJSON *identify = cJSON_CreateObject();
01043     cJSON_AddNumberToObject(identify, "op", OBSWS_OPCODE_IDENTIFY);
01044
01045     cJSON *identify_data = cJSON_CreateObject();
01046     cJSON_AddNumberToObject(identify_data, "rpcVersion", OBSWS_PROTOCOL_VERSION);
01047     cJSON_AddNumberToObject(identify_data, "eventSubscriptions", OBSWS_EVENT_ALL);
01048
01049     if (conn->auth_required && conn->config.password) {
01050         /* DEBUG_HIGH: Show password being used */
01051         obsws_debug(conn, OBSWS_DEBUG_HIGH, "Generating auth response with password:  '%s'",
      conn->config.password);
01052         char *auth_response = generate_auth_response(conn->config.password, conn->salt,
      conn->challenge);
01053         /* DEBUG_MEDIUM: Show generated auth string */
01054         obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Generated auth response:  '%s'", auth_response);
01055         cJSON_AddStringToObject(identify_data, "authentication", auth_response);
01056         free(auth_response);
01057     } else {
01058         if (conn->auth_required) {
01059             obsws_log(conn, OBSWS_LOG_ERROR, "Authentication required but no password provided!");
01060         }
01061     }
01062
01063     cJSON_AddItemToObject(identify, "d", identify_data);
01064
01065     char *message = cJSON_PrintUnformatted(identify);
01066     cJSON_Delete(identify);
01067
```

```
01068     /* DEBUG_HIGH: Show full Identify message */
01069     obsws_debug(conn, OBSWS_DEBUG_HIGH, "Sending Identify message:  %s", message);
01070
01071     pthread_mutex_lock(&conn->send_mutex);
01072     size_t len = strlen(message);
01073     if (len < conn->send_buffer_size - LWS_PRE) {
01074         memcpy(conn->send_buffer + LWS_PRE, message, len);
01075         int written = lws_write(conn->wsi, (unsigned char *)(conn->send_buffer + LWS_PRE), len,
      LWS_WRITE_TEXT);
01076         /* DEBUG_HIGH: Show bytes sent */
01077         obsws_debug(conn, OBSWS_DEBUG_HIGH, "Sent %d bytes (requested %zu)", written, len);
01078     } else {
01079         obsws_log(conn, OBSWS_LOG_ERROR, "Message too large for send buffer:  %zu bytes", len);
01080     }
01081     pthread_mutex_unlock(&conn->send_mutex);
01082
01083     free(message);
01084     return 0;
01085 }
01086
01114 static int handle_identified_message(obsws_connection_t *conn, cJSON *data) {
01115     (void)data;  /* Unused parameter */
01116     obsws_log(conn, OBSWS_LOG_INFO, "Successfully authenticated with OBS");
01117     /* DEBUG_LOW: Authentication success */
01118     obsws_debug(conn, OBSWS_DEBUG_LOW, "Identified message received - authentication successful");
01119     set_connection_state(conn, OBSWS_STATE_CONNECTED);
01120
01121     pthread_mutex_lock(&conn->stats_mutex);
01122     conn->stats.connected_since = time(NULL);
01123     conn->stats.reconnect_count = conn->reconnect_attempts;
01124     pthread_mutex_unlock(&conn->stats_mutex);
01125
01126     conn->reconnect_attempts = 0;
01127     conn->current_reconnect_delay = conn->config.reconnect_delay_ms;
01128
01129     return 0;
01130 }
01131
01166 static int handle_event_message(obsws_connection_t *conn, cJSON *data) {
01167     cJSON *event_type = cJSON_GetObjectItem(data, "eventType");
01168     cJSON *event_data = cJSON_GetObjectItem(data, "eventData");
01169
01170     /* DEBUG_MEDIUM: Show event type */
01171     if (event_type) {
01172         obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Event received:  %s", event_type->valuestring);
01173     }
01174
01175     if (event_type && conn->config.event_callback) {
01176         char *event_data_str = event_data ?  cJSON_PrintUnformatted(event_data) :  NULL;
01177         /* DEBUG_HIGH: Show full event data */
01178         if (event_data_str) {
01179             obsws_debug(conn, OBSWS_DEBUG_HIGH, "Event data:  %s", event_data_str);
01180         }
01181         conn->config.event_callback(conn, event_type->valuestring, event_data_str,
      conn->config.user_data);
01182         if (event_data_str) free(event_data_str);
01183     }
01184
01185     /* Update current scene cache if scene changed */
01186     if (event_type && strcmp(event_type->valuestring, "CurrentProgramSceneChanged") == 0) {
01187         cJSON *scene_name = cJSON_GetObjectItem(event_data, "sceneName");
01188         if (scene_name) {
01189             pthread_mutex_lock(&conn->scene_mutex);
01190             free(conn->current_scene);
01191             conn->current_scene = strdup(scene_name->valuestring);
01192             pthread_mutex_unlock(&conn->scene_mutex);
01193             /* DEBUG_LOW: Scene changes are important */
01194             obsws_debug(conn, OBSWS_DEBUG_LOW, "Scene changed to:  %s", scene_name->valuestring);
01195         }
01196     }
01197
01198     return 0;
01199 }
01200
01239 static int handle_request_response_message(obsws_connection_t *conn, cJSON *data) {
01240     cJSON *request_id = cJSON_GetObjectItem(data, "requestId");
01241     if (!request_id) return -1;
01242
01243     /* DEBUG_MEDIUM: Show request ID being processed */
01244     obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Response received for request:  %s",
      request_id->valuestring);
01245
01246     pending_request_t *req = find_pending_request(conn, request_id->valuestring);
01247     if (!req) {
01248         obsws_log(conn, OBSWS_LOG_WARNING, "Received response for unknown request:  %s",
      request_id->valuestring);
01249         return -1;
```

```
01250     }
01251
01252     pthread_mutex_lock(&req->mutex);
01253
01254     cJSON *request_status = cJSON_GetObjectItem(data, "requestStatus");
01255     if (request_status) {
01256         cJSON *result = cJSON_GetObjectItem(request_status, "result");
01257         cJSON *code = cJSON_GetObjectItem(request_status, "code");
01258         cJSON *comment = cJSON_GetObjectItem(request_status, "comment");
01259
01260         req->response->success = result ?  result->valueint :  false;
01261         req->response->status_code = code ?  code->valueint :  -1;
01262
01263         if (comment) {
01264             req->response->error_message = strdup(comment->valuestring);
01265         }
01266     }
01267
01268     cJSON *response_data = cJSON_GetObjectItem(data, "responseData");
01269     if (response_data) {
01270         req->response->response_data = cJSON_PrintUnformatted(response_data);
01271     }
01272
01273     req->completed = true;
01274     pthread_cond_broadcast(&req->cond);
01275     pthread_mutex_unlock(&req->mutex);
01276
01277     return 0;
01278 }
01279
01312 static int handle_websocket_message(obsws_connection_t *conn, const char *message, size_t len) {
01313     /* DEBUG_HIGH: Show full message content */
01314     obsws_debug(conn, OBSWS_DEBUG_HIGH, "Received message (%zu bytes):  %.*s", len, (int)len,
      message);
01315
01316     cJSON *json = cJSON_ParseWithLength(message, len);
01317     if (!json) {
01318         obsws_log(conn, OBSWS_LOG_ERROR, "Failed to parse JSON message");
01319         return -1;
01320     }
01321
01322     cJSON *op = cJSON_GetObjectItem(json, "op");
01323     cJSON *data = cJSON_GetObjectItem(json, "d");
01324
01325     if (!op) {
01326         obsws_log(conn, OBSWS_LOG_ERROR, "Message missing 'op' field");
01327         cJSON_Delete(json);
01328         return -1;
01329     }
01330
01331     /* DEBUG_MEDIUM: Show opcode being processed */
01332     obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Processing opcode:  %d", op->valueint);
01333
01334     int result = 0;
01335     switch (op->valueint) {
01336         case OBSWS_OPCODE_HELLO:
01337             result = handle_hello_message(conn, data);
01338             break;
01339         case OBSWS_OPCODE_IDENTIFIED:
01340             result = handle_identified_message(conn, data);
01341             break;
01342         case OBSWS_OPCODE_EVENT:
01343             result = handle_event_message(conn, data);
01344             break;
01345         case OBSWS_OPCODE_REQUEST_RESPONSE:
01346             result = handle_request_response_message(conn, data);
01347             break;
01348         default:
01349             obsws_log(conn, OBSWS_LOG_DEBUG, "Unhandled opcode:  %d", op->valueint);
01350             break;
01351     }
01352
01353     cJSON_Delete(json);
01354
01355     pthread_mutex_lock(&conn->stats_mutex);
01356     conn->stats.messages_received++;
01357     conn->stats.bytes_received += len;
01358     pthread_mutex_unlock(&conn->stats_mutex);
01359
01360     return result;
01361 }
01362
01363 /* ============================================================================
01364 * libwebsockets Callbacks
01365 * ============================================================================ */
01366
01402 static int lws_callback(struct lws *wsi, enum lws_callback_reasons reason,
```

```
01403                       void *user, void *in, size_t len) {
01404     obsws_connection_t *conn = (obsws_connection_t *)user;
01405 
01406     switch (reason) {
01407         case LWS_CALLBACK_CLIENT_ESTABLISHED:
01408             obsws_log(conn, OBSWS_LOG_INFO, "WebSocket connection established");
01409             set_connection_state(conn, OBSWS_STATE_CONNECTING);
01410             break;
01411 
01412         case LWS_CALLBACK_CLIENT_RECEIVE:
01413             if (conn->recv_buffer_used + len < conn->recv_buffer_size) {
01414                 memcpy(conn->recv_buffer + conn->recv_buffer_used, in, len);
01415                 conn->recv_buffer_used += len;
01416 
01417                 if (lws_is_final_fragment(wsi)) {
01418                     handle_websocket_message(conn, conn->recv_buffer, conn->recv_buffer_used);
01419                     conn->recv_buffer_used = 0;
01420                 }
01421             } else {
01422                 obsws_log(conn, OBSWS_LOG_ERROR, "Receive buffer overflow");
01423                 conn->recv_buffer_used = 0;
01424             }
01425             break;
01426 
01427         case LWS_CALLBACK_CLIENT_WRITEABLE:
01428             /* Handle queued sends if needed */
01429             break;
01430 
01431         case LWS_CALLBACK_CLIENT_CONNECTION_ERROR:
01432             obsws_log(conn, OBSWS_LOG_ERROR, "Connection error: %s", in ? (char *)in : "unknown");
01433             set_connection_state(conn, OBSWS_STATE_ERROR);
01434             break;
01435 
01436         case LWS_CALLBACK_CLIENT_CLOSED:
01437             obsws_log(conn, OBSWS_LOG_INFO, "WebSocket connection closed (reason in 'in' param)");
01438             if (in && len > 0) {
01439                 obsws_log(conn, OBSWS_LOG_INFO, "Close reason: %.*s", (int)len, (char*)in);
01440             }
01441             set_connection_state(conn, OBSWS_STATE_DISCONNECTED);
01442             break;
01443 
01444         case LWS_CALLBACK_WSI_DESTROY:
01445             conn->wsi = NULL;
01446             break;
01447 
01448         default:
01449             break;
01450     }
01451 
01452     return 0;
01453 }
01454 
01455 static const struct lws_protocols protocols[] = {
01456     {
01457         "obs-websocket",
01458         lws_callback,
01459         0,
01460         OBSWS_DEFAULT_BUFFER_SIZE,
01461         0, /* id */
01462         NULL, /* user */
01463         0 /* tx_packet_size */
01464     },
01465     { NULL, NULL, 0, 0, 0, NULL, 0 }
01466 };
01467 
01468 /* ============================================================================
01469 * Event Thread
01470 * ============================================================================ */
01471 
01511 static void* event_thread_func(void *arg) {
01512     obsws_connection_t *conn = (obsws_connection_t *)arg;
01513 
01514     bool should_continue = true;
01515     while (should_continue) {
01516         /* Check exit flag with mutex protection */
01517         pthread_mutex_lock(&conn->state_mutex);
01518         should_continue = !conn->should_exit;
01519         pthread_mutex_unlock(&conn->state_mutex);
01520 
01521         if (!should_continue) break;
01522 
01523         if (conn->lws_context) {
01524             lws_service(conn->lws_context, 50);
01525 
01526             /* Cleanup old requests periodically */
01527             cleanup_old_requests(conn);
01528 
```

```
01529             /* Handle keep-alive pings */
01530             if (conn->config.ping_interval_ms > 0 && conn->state == OBSWS_STATE_CONNECTED) {
01531                 time_t now = time(NULL);
01532                 if (now - conn->last_ping_sent >= conn->config.ping_interval_ms / 1000) {
01533                     if (conn->wsi) {
01534                         lws_callback_on_writable(conn->wsi);
01535                     }
01536                     conn->last_ping_sent = now;
01537                 }
01538             }
01539         } else {
01540             struct timespec ts = {0, 50000000};  /* 50ms = 50,000,000 nanoseconds */
01541             nanosleep(&ts, NULL);
01542         }
01543     }
01544
01545     return NULL;
01546 }
01547
01548 /* ============================================================================
01549 * Public API Implementation
01550 * ============================================================================ */
01551
01576 obsws_error_t obsws_init(void) {
01577     pthread_mutex_lock(&g_init_mutex);
01578
01579     if (g_library_initialized) {
01580         pthread_mutex_unlock(&g_init_mutex);
01581         return OBSWS_OK;
01582     }
01583
01584     /* Initialize OpenSSL */
01585     OpenSSL_add_all_algorithms();
01586
01587     /* Seed random number generator */
01588     srand(time(NULL));
01589
01590     /* Detect if stderr is a TTY for color output auto-detection */
01591     g_log_ctx.is_tty = isatty(STDERR_FILENO) == 1;
01592
01593     g_library_initialized = true;
01594     pthread_mutex_unlock(&g_init_mutex);
01595
01596     return OBSWS_OK;
01597 }
01598
01620 void obsws_cleanup(void) {
01621     pthread_mutex_lock(&g_init_mutex);
01622
01623     if (!g_library_initialized) {
01624         pthread_mutex_unlock(&g_init_mutex);
01625         return;
01626     }
01627
01628     EVP_cleanup();
01629     g_library_initialized = false;
01630
01631     pthread_mutex_unlock(&g_init_mutex);
01632 }
01633
01642 const char* obsws_version(void) {
01643     return OBSWS_VERSION;
01644 }
01645
01668 void obsws_set_log_level(obsws_log_level_t level) {
01669     g_log_level = level;
01670 }
01671
01696 void obsws_set_debug_level(obsws_debug_level_t level) {
01697     g_debug_level = level;
01698 }
01699
01716 obsws_debug_level_t obsws_get_debug_level(void) {
01717     return g_debug_level;
01718 }
01719
01750 void obsws_config_init(obsws_config_t *config) {
01751     memset(config, 0, sizeof(obsws_config_t));
01752
01753     config->port = 4455;
01754     config->use_ssl = false;
01755     config->connect_timeout_ms = 5000;
01756     config->recv_timeout_ms = 5000;
01757     config->send_timeout_ms = 5000;
01758     config->ping_interval_ms = 10000;
01759     config->ping_timeout_ms = 5000;
01760     config->auto_reconnect = true;
```

```
01761     config->reconnect_delay_ms = 1000;
01762     config->max_reconnect_delay_ms = 30000;
01763     config->max_reconnect_attempts = 0; /* Infinite */
01764 }
01765
01808 obsws_connection_t* obsws_connect(const obsws_config_t *config) {
01809     if (!g_library_initialized) {
01810         obsws_init();
01811     }
01812
01813     if (!config || !config->host) {
01814         return NULL;
01815     }
01816
01817     obsws_connection_t *conn = calloc(1, sizeof(obsws_connection_t));
01818     if (!conn) return NULL;
01819
01820     /* Copy configuration */
01821     memcpy(&conn->config, config, sizeof(obsws_config_t));
01822     if (config->host) conn->config.host = strdup(config->host);
01823     if (config->password) conn->config.password = strdup(config->password);
01824
01825     /* Initialize mutexes */
01826     pthread_mutex_init(&conn->state_mutex, NULL);
01827     pthread_mutex_init(&conn->send_mutex, NULL);
01828     pthread_mutex_init(&conn->requests_mutex, NULL);
01829     pthread_mutex_init(&conn->stats_mutex, NULL);
01830     pthread_mutex_init(&conn->scene_mutex, NULL);
01831
01832     /* Allocate buffers */
01833     conn->recv_buffer_size = OBSWS_DEFAULT_BUFFER_SIZE;
01834     conn->recv_buffer = malloc(conn->recv_buffer_size);
01835     conn->send_buffer_size = OBSWS_DEFAULT_BUFFER_SIZE;
01836     conn->send_buffer = malloc(conn->send_buffer_size);
01837
01838     conn->state = OBSWS_STATE_DISCONNECTED;
01839     conn->current_reconnect_delay = config->reconnect_delay_ms;
01840
01841     /* Create libwebsockets context */
01842     struct lws_context_creation_info info;
01843     memset(&info, 0, sizeof(info));
01844
01845     info.port = CONTEXT_PORT_NO_LISTEN;
01846     info.protocols = protocols;
01847     info.gid = -1;
01848     info.uid = -1;
01849     info.options = LWS_SERVER_OPTION_DO_SSL_GLOBAL_INIT;
01850
01851     conn->lws_context = lws_create_context(&info);
01852     if (!conn->lws_context) {
01853         obsws_log(conn, OBSWS_LOG_ERROR, "Failed to create libwebsockets context");
01854         free(conn->recv_buffer);
01855         free(conn->send_buffer);
01856         free(conn);
01857         return NULL;
01858     }
01859
01860     /* Connect to OBS */
01861     struct lws_client_connect_info ccinfo;
01862     memset(&ccinfo, 0, sizeof(ccinfo));
01863
01864     ccinfo.context = conn->lws_context;
01865     ccinfo.address = conn->config.host;
01866     ccinfo.port = conn->config.port;
01867     ccinfo.path = "/";
01868     ccinfo.host = ccinfo.address;
01869     ccinfo.origin = ccinfo.address;
01870     ccinfo.protocol = protocols[0].name;
01871     ccinfo.userdata = conn;
01872
01873     if (config->use_ssl) {
01874         ccinfo.ssl_connection = LCCSCF_USE_SSL;
01875     }
01876
01877     conn->wsi = lws_client_connect_via_info(&ccinfo);
01878     if (!conn->wsi) {
01879         obsws_log(conn, OBSWS_LOG_ERROR, "Failed to initiate connection");
01880         lws_context_destroy(conn->lws_context);
01881         free(conn->recv_buffer);
01882         free(conn->send_buffer);
01883         free(conn);
01884         return NULL;
01885     }
01886
01887     /* Start event thread - protect flags with mutex */
01888     pthread_mutex_lock(&conn->state_mutex);
01889     conn->thread_running = true;
```

```
01890     conn->should_exit = false;
01891     pthread_mutex_unlock(&conn->state_mutex);
01892
01893     pthread_create(&conn->event_thread, NULL, event_thread_func, conn);
01894
01895     obsws_log(conn, OBSWS_LOG_INFO, "Connecting to OBS at %s:%d", config->host, config->port);
01896
01897     return conn;
01898 }
01899
01932 void obsws_disconnect(obsws_connection_t *conn) {
01933     if (!conn) return;
01934
01935     obsws_log(conn, OBSWS_LOG_INFO, "Disconnecting from OBS");
01936
01937     /* Stop event thread - protect flag with mutex */
01938     pthread_mutex_lock(&conn->state_mutex);
01939     conn->should_exit = true;
01940     bool thread_was_running = conn->thread_running;
01941     pthread_mutex_unlock(&conn->state_mutex);
01942
01943     if (thread_was_running) {
01944         pthread_join(conn->event_thread, NULL);
01945     }
01946
01947     /* Close WebSocket - only if connected */
01948     if (conn->wsi && conn->state == OBSWS_STATE_CONNECTED) {
01949         lws_close_reason(conn->wsi, LWS_CLOSE_STATUS_NORMAL, NULL, 0);
01950     }
01951
01952     /* Cleanup libwebsockets */
01953     if (conn->lws_context) {
01954         lws_context_destroy(conn->lws_context);
01955     }
01956
01957     /* Free pending requests */
01958     pthread_mutex_lock(&conn->requests_mutex);
01959     pending_request_t *req = conn->pending_requests;
01960     while (req) {
01961         pending_request_t *next = req->next;
01962         if (req->response) {
01963             obsws_response_free(req->response);
01964         }
01965         pthread_mutex_destroy(&req->mutex);
01966         pthread_cond_destroy(&req->cond);
01967         free(req);
01968         req = next;
01969     }
01970     pthread_mutex_unlock(&conn->requests_mutex);
01971
01972     /* Free resources */
01973     free(conn->recv_buffer);
01974     free(conn->send_buffer);
01975     free((char *)conn->config.host);
01976     free((char *)conn->config.password);
01977     free(conn->challenge);
01978     free(conn->salt);
01979     free(conn->current_scene);
01980
01981     /* Destroy mutexes */
01982     pthread_mutex_destroy(&conn->state_mutex);
01983     pthread_mutex_destroy(&conn->send_mutex);
01984     pthread_mutex_destroy(&conn->requests_mutex);
01985     pthread_mutex_destroy(&conn->stats_mutex);
01986     pthread_mutex_destroy(&conn->scene_mutex);
01987
01988     free(conn);
01989 }
01990
02008 bool obsws_is_connected(const obsws_connection_t *conn) {
02009     if (!conn) return false;
02010
02011     /* Thread-safe state check */
02012     pthread_mutex_lock((pthread_mutex_t *)&conn->state_mutex);
02013     bool connected = (conn->state == OBSWS_STATE_CONNECTED);
02014     pthread_mutex_unlock((pthread_mutex_t *)&conn->state_mutex);
02015
02016     return connected;
02017 }
02018
02046 obsws_state_t obsws_get_state(const obsws_connection_t *conn) {
02047     if (!conn) return OBSWS_STATE_DISCONNECTED;
02048
02049     pthread_mutex_lock((pthread_mutex_t *)&conn->state_mutex);
02050     obsws_state_t state = conn->state;
02051     pthread_mutex_unlock((pthread_mutex_t *)&conn->state_mutex);
02052
```

```
02053     return state;
02054 }
02055
02089 obsws_error_t obsws_get_stats(const obsws_connection_t *conn, obsws_stats_t *stats) {
02090     if (!conn || !stats) return OBSWS_ERROR_INVALID_PARAM;
02091
02092     pthread_mutex_lock((pthread_mutex_t *)&conn->stats_mutex);
02093     memcpy(stats, &conn->stats, sizeof(obsws_stats_t));
02094     pthread_mutex_unlock((pthread_mutex_t *)&conn->stats_mutex);
02095
02096     return OBSWS_OK;
02097 }
02098
02159 obsws_error_t obsws_send_request(obsws_connection_t *conn, const char *request_type,
02160                                  const char *request_data, obsws_response_t **response, uint32_t
      timeout_ms) {
02161     if (!conn || !request_type || !response) {
02162         return OBSWS_ERROR_INVALID_PARAM;
02163     }
02164
02165     if (conn->state != OBSWS_STATE_CONNECTED) {
02166         return OBSWS_ERROR_NOT_CONNECTED;
02167     }
02168
02169     /* Generate request ID */
02170     char request_id[OBSWS_UUID_LENGTH];
02171     generate_uuid(request_id);
02172
02173     /* Create pending request */
02174     pending_request_t *req = create_pending_request(conn, request_id);
02175     if (!req) {
02176         return OBSWS_ERROR_OUT_OF_MEMORY;
02177     }
02178
02179     /* Build request JSON */
02180     cJSON *request = cJSON_CreateObject();
02181     cJSON_AddNumberToObject(request, "op", OBSWS_OPCODE_REQUEST);
02182
02183     cJSON *d = cJSON_CreateObject();
02184     cJSON_AddStringToObject(d, "requestType", request_type);
02185     cJSON_AddStringToObject(d, "requestId", request_id);
02186
02187     if (request_data) {
02188         cJSON *data = cJSON_Parse(request_data);
02189         if (data) {
02190             cJSON_AddItemToObject(d, "requestData", data);
02191         }
02192     }
02193
02194     cJSON_AddItemToObject(request, "d", d);
02195
02196     char *message = cJSON_PrintUnformatted(request);
02197     cJSON_Delete(request);
02198
02199     /* DEBUG_HIGH: Show request being sent */
02200     obsws_debug(conn, OBSWS_DEBUG_HIGH, "Sending request (ID: %s):  %s", request_id, message);
02201
02202     /* Send request */
02203     pthread_mutex_lock(&conn->send_mutex);
02204     size_t len = strlen(message);
02205     obsws_error_t result = OBSWS_OK;
02206
02207     if (len < conn->send_buffer_size - LWS_PRE && conn->wsi) {
02208         memcpy(conn->send_buffer + LWS_PRE, message, len);
02209         int written = lws_write(conn->wsi, (unsigned char *)(conn->send_buffer + LWS_PRE), len,
      LWS_WRITE_TEXT);
02210
02211         if (written < 0) {
02212             result = OBSWS_ERROR_SEND_FAILED;
02213         } else {
02214             pthread_mutex_lock(&conn->stats_mutex);
02215             conn->stats.messages_sent++;
02216             conn->stats.bytes_sent += len;
02217             pthread_mutex_unlock(&conn->stats_mutex);
02218         }
02219     } else {
02220         result = OBSWS_ERROR_SEND_FAILED;
02221     }
02222     pthread_mutex_unlock(&conn->send_mutex);
02223
02224     free(message);
02225
02226     if (result != OBSWS_OK) {
02227         remove_pending_request(conn, req);
02228         return result;
02229     }
02230
```

```
02231     /* Wait for response */
02232     if (timeout_ms == 0) {
02233         timeout_ms = conn->config.recv_timeout_ms;
02234     }
02235 
02236     struct timespec ts;
02237     clock_gettime(CLOCK_REALTIME, &ts);
02238     ts.tv_sec += timeout_ms / 1000;
02239     ts.tv_nsec += (timeout_ms % 1000) * 1000000;
02240     if (ts.tv_nsec >= 1000000000) {
02241         ts.tv_sec++;
02242         ts.tv_nsec -= 1000000000;
02243     }
02244 
02245     pthread_mutex_lock(&req->mutex);
02246     while (!req->completed) {
02247         int wait_result = pthread_cond_timedwait(&req->cond, &req->mutex, &ts);
02248         if (wait_result == ETIMEDOUT) {
02249             pthread_mutex_unlock(&req->mutex);
02250             remove_pending_request(conn, req);
02251             return OBSWS_ERROR_TIMEOUT;
02252         }
02253     }
02254 
02255     *response = req->response;
02256     req->response = NULL; /* Transfer ownership */
02257     pthread_mutex_unlock(&req->mutex);
02258 
02259     remove_pending_request(conn, req);
02260 
02261     return OBSWS_OK;
02262 }
02263 
02318 obsws_error_t obsws_set_current_scene(obsws_connection_t *conn, const char *scene_name,
      obsws_response_t **response) {
02319     if (!conn || !scene_name) {
02320         return OBSWS_ERROR_INVALID_PARAM;
02321     }
02322 
02323     /* Check cache to avoid redundant switches */
02324     pthread_mutex_lock(&conn->scene_mutex);
02325     bool already_current = (conn->current_scene && strcmp(conn->current_scene, scene_name) == 0);
02326     pthread_mutex_unlock(&conn->scene_mutex);
02327 
02328     if (already_current) {
02329         obsws_log(conn, OBSWS_LOG_DEBUG, "Already on scene:  %s", scene_name);
02330         if (response) {
02331             *response = calloc(1, sizeof(obsws_response_t));
02332             (*response)->success = true;
02333         }
02334         return OBSWS_OK;
02335     }
02336 
02337     cJSON *request_data = cJSON_CreateObject();
02338     cJSON_AddStringToObject(request_data, "sceneName", scene_name);
02339 
02340     char *data_str = cJSON_PrintUnformatted(request_data);
02341     cJSON_Delete(request_data);
02342 
02343     obsws_response_t *resp = NULL;
02344     obsws_error_t result = obsws_send_request(conn, "SetCurrentProgramScene", data_str, &resp, 0);
02345     free(data_str);
02346 
02347     if (result == OBSWS_OK && resp && resp->success) {
02348         pthread_mutex_lock(&conn->scene_mutex);
02349         free(conn->current_scene);
02350         conn->current_scene = strdup(scene_name);
02351         pthread_mutex_unlock(&conn->scene_mutex);
02352 
02353         obsws_log(conn, OBSWS_LOG_INFO, "Switched to scene:  %s", scene_name);
02354     }
02355 
02356     if (response) {
02357         *response = resp;
02358     } else if (resp) {
02359         obsws_response_free(resp);
02360     }
02361 
02362     return result;
02363 }
02364 
02405 obsws_error_t obsws_get_current_scene(obsws_connection_t *conn, char *scene_name, size_t buffer_size)
    {
02406     if (!conn || !scene_name || buffer_size == 0) {
02407         return OBSWS_ERROR_INVALID_PARAM;
02408     }
02409 
```

```
02410     obsws_response_t *response = NULL;
02411     obsws_error_t result = obsws_send_request(conn, "GetCurrentProgramScene", NULL, &response, 0);
02412
02413     if (result == OBSWS_OK && response && response->success && response->response_data) {
02414         cJSON *data = cJSON_Parse(response->response_data);
02415         if (data) {
02416             cJSON *name = cJSON_GetObjectItem(data, "currentProgramSceneName");
02417             if (name && name->valuestring) {
02418                 strncpy(scene_name, name->valuestring, buffer_size - 1);
02419                 scene_name[buffer_size - 1] = '\0';
02420
02421                 /* Update cache */
02422                 pthread_mutex_lock(&conn->scene_mutex);
02423                 free(conn->current_scene);
02424                 conn->current_scene = strdup(name->valuestring);
02425                 pthread_mutex_unlock(&conn->scene_mutex);
02426             }
02427             cJSON_Delete(data);
02428         }
02429     }
02430
02431     if (response) {
02432         obsws_response_free(response);
02433     }
02434
02435     return result;
02436 }
02437
02469 obsws_error_t obsws_start_recording(obsws_connection_t *conn, obsws_response_t **response) {
02470     return obsws_send_request(conn, "StartRecord", NULL, response, 0);
02471 }
02472
02498 obsws_error_t obsws_stop_recording(obsws_connection_t *conn, obsws_response_t **response) {
02499     return obsws_send_request(conn, "StopRecord", NULL, response, 0);
02500 }
02501
02522 obsws_error_t obsws_start_streaming(obsws_connection_t *conn, obsws_response_t **response) {
02523     return obsws_send_request(conn, "StartStream", NULL, response, 0);
02524 }
02525
02544 obsws_error_t obsws_stop_streaming(obsws_connection_t *conn, obsws_response_t **response) {
02545     return obsws_send_request(conn, "StopStream", NULL, response, 0);
02546 }
02547
02578 void obsws_response_free(obsws_response_t *response) {
02579     if (!response) return;
02580
02581     free(response->error_message);
02582     free(response->response_data);
02583     free(response);
02584 }
02585
02612 const char* obsws_error_string(obsws_error_t error) {
02613     switch (error) {
02614         case OBSWS_OK: return "Success";
02615         case OBSWS_ERROR_INVALID_PARAM: return "Invalid parameter";
02616         case OBSWS_ERROR_CONNECTION_FAILED: return "Connection failed";
02617         case OBSWS_ERROR_AUTH_FAILED: return "Authentication failed";
02618         case OBSWS_ERROR_TIMEOUT: return "Timeout";
02619         case OBSWS_ERROR_SEND_FAILED: return "Send failed";
02620         case OBSWS_ERROR_RECV_FAILED: return "Receive failed";
02621         case OBSWS_ERROR_PARSE_FAILED: return "Parse failed";
02622         case OBSWS_ERROR_NOT_CONNECTED: return "Not connected";
02623         case OBSWS_ERROR_ALREADY_CONNECTED: return "Already connected";
02624         case OBSWS_ERROR_OUT_OF_MEMORY: return "Out of memory";
02625         case OBSWS_ERROR_SSL_FAILED: return "SSL failed";
02626         default:  return "Unknown error";
02627     }
02628 }
02629
02665 const char* obsws_state_string(obsws_state_t state) {
02666     switch (state) {
02667         case OBSWS_STATE_DISCONNECTED: return "Disconnected";
02668         case OBSWS_STATE_CONNECTING: return "Connecting";
02669         case OBSWS_STATE_AUTHENTICATING: return "Authenticating";
02670         case OBSWS_STATE_CONNECTED: return "Connected";
02671         case OBSWS_STATE_ERROR: return "Error";
02672         default:  return "Unknown";
02673     }
02674 }
02675
02730 int obsws_process_events(obsws_connection_t *conn, uint32_t timeout_ms) {
02731     if (!conn) return OBSWS_ERROR_INVALID_PARAM;
02732
02733     /* Events are processed in the background thread */
02734     /* This function is provided for API compatibility */
02735     if (timeout_ms > 0) {
```

```
02736         struct timespec ts;
02737         ts.tv_sec = timeout_ms / 1000;
02738         ts.tv_nsec = (timeout_ms % 1000) * 1000000;
02739         nanosleep(&ts, NULL);
02740     }
02741
02742     return 0;
02743 }
02744
02756 int obsws_ping(obsws_connection_t *conn, uint32_t timeout_ms) {
02757     if (!conn || !obsws_is_connected(conn)) {
02758         return OBSWS_ERROR_NOT_CONNECTED;
02759     }
02760
02761     obsws_response_t *response = NULL;
02762     struct timeval start, end;
02763     gettimeofday(&start, NULL);
02764
02765     obsws_error_t err = obsws_send_request(conn, "Ping", NULL, &response, timeout_ms);
02766
02767     gettimeofday(&end, NULL);
02768
02769     int latency_ms = (int)((end.tv_sec - start.tv_sec) * 1000 +
02770                            (end.tv_usec - start.tv_usec) / 1000);
02771
02772     if (response) {
02773         obsws_response_free(response);
02774     }
02775
02776     if (err == OBSWS_OK) {
02777         return latency_ms;
02778     }
02779     return (int)err;
02780 }
02781
02794 obsws_error_t obsws_get_recording_status(obsws_connection_t *conn, bool *is_recording,
      obsws_response_t **response) {
02795     if (!conn || !is_recording) {
02796         return OBSWS_ERROR_INVALID_PARAM;
02797     }
02798
02799     *is_recording = false;
02800
02801     obsws_response_t *resp = NULL;
02802     obsws_error_t result = obsws_send_request(conn, "GetRecordStatus", NULL, &resp, 0);
02803
02804     if (result == OBSWS_OK && resp && resp->success && resp->response_data) {
02805         cJSON *data = cJSON_Parse(resp->response_data);
02806         if (data) {
02807             cJSON *recording = cJSON_GetObjectItem(data, "outputActive");
02808             if (recording && recording->type == cJSON_True) {
02809                 *is_recording = true;
02810             }
02811             cJSON_Delete(data);
02812         }
02813     }
02814
02815     if (response) {
02816         *response = resp;
02817     } else if (resp) {
02818         obsws_response_free(resp);
02819     }
02820
02821     return result;
02822 }
02823
02836 obsws_error_t obsws_get_streaming_status(obsws_connection_t *conn, bool *is_streaming,
      obsws_response_t **response) {
02837     if (!conn || !is_streaming) {
02838         return OBSWS_ERROR_INVALID_PARAM;
02839     }
02840
02841     *is_streaming = false;
02842
02843     obsws_response_t *resp = NULL;
02844     obsws_error_t result = obsws_send_request(conn, "GetStreamStatus", NULL, &resp, 0);
02845
02846     if (result == OBSWS_OK && resp && resp->success && resp->response_data) {
02847         cJSON *data = cJSON_Parse(resp->response_data);
02848         if (data) {
02849             cJSON *streaming = cJSON_GetObjectItem(data, "outputActive");
02850             if (streaming && streaming->type == cJSON_True) {
02851                 *is_streaming = true;
02852             }
02853             cJSON_Delete(data);
02854         }
02855     }
```

```
02856
02857     if (response) {
02858         *response = resp;
02859     } else if (resp) {
02860         obsws_response_free(resp);
02861     }
02862
02863     return result;
02864 }
02865
02879 obsws_error_t obsws_get_scene_list(obsws_connection_t *conn, char ***scenes, size_t *count) {
02880     if (!conn || !scenes || !count) {
02881         return OBSWS_ERROR_INVALID_PARAM;
02882     }
02883
02884     *scenes = NULL;
02885     *count = 0;
02886
02887     obsws_response_t *response = NULL;
02888     obsws_error_t result = obsws_send_request(conn, "GetSceneList", NULL, &response, 0);
02889
02890     if (result == OBSWS_OK && response && response->success && response->response_data) {
02891         cJSON *data = cJSON_Parse(response->response_data);
02892         if (data) {
02893             cJSON *scenes_array = cJSON_GetObjectItem(data, "scenes");
02894             if (scenes_array && scenes_array->type == cJSON_Array) {
02895                 int num_scenes = cJSON_GetArraySize(scenes_array);
02896                 if (num_scenes > 0) {
02897                     *scenes = calloc(num_scenes, sizeof(char *));
02898                     if (*scenes) {
02899                         int index = 0;
02900                         cJSON *scene_item = NULL;
02901                         cJSON_ArrayForEach(scene_item, scenes_array) {
02902                             cJSON *scene_name = cJSON_GetObjectItem(scene_item, "sceneName");
02903                             if (scene_name && scene_name->valuestring && index < num_scenes) {
02904                                 (*scenes)[index] = strdup(scene_name->valuestring);
02905                                 index++;
02906                             }
02907                         }
02908                         *count = index;
02909                     }
02910                 }
02911             }
02912             cJSON_Delete(data);
02913         }
02914     }
02915
02916     if (response) {
02917         obsws_response_free(response);
02918     }
02919
02920     return result;
02921 }
02922
02931 void obsws_free_scene_list(char **scenes, size_t count) {
02932     if (!scenes) return;
02933
02934     for (size_t i = 0; i < count; i++) {
02935         free(scenes[i]);
02936     }
02937     free(scenes);
02938 }
02939
02954 obsws_error_t obsws_set_source_visibility(obsws_connection_t *conn, const char *scene_name,
02955                                           const char *source_name, bool visible, obsws_response_t
      **response) {
02956     if (!conn || !scene_name || !source_name) {
02957         return OBSWS_ERROR_INVALID_PARAM;
02958     }
02959
02960     /* First get the source item ID for this source in this scene */
02961     cJSON *req_data = cJSON_CreateObject();
02962     cJSON_AddStringToObject(req_data, "sceneName", scene_name);
02963     char *data_str = cJSON_PrintUnformatted(req_data);
02964     cJSON_Delete(req_data);
02965
02966     obsws_response_t *get_items_resp = NULL;
02967     obsws_error_t result = obsws_send_request(conn, "GetSceneItemList", data_str, &get_items_resp, 0);
02968     free(data_str);
02969
02970     int source_id = -1;
02971     if (result == OBSWS_OK && get_items_resp && get_items_resp->success &&
      get_items_resp->response_data) {
02972         cJSON *data = cJSON_Parse(get_items_resp->response_data);
02973         if (data) {
02974             cJSON *items = cJSON_GetObjectItem(data, "sceneItems");
02975             if (items && items->type == cJSON_Array) {
```

```
02976                 cJSON *item = NULL;
02977                 cJSON_ArrayForEach(item, items) {
02978                     cJSON *name = cJSON_GetObjectItem(item, "sourceName");
02979                     cJSON *id = cJSON_GetObjectItem(item, "sceneItemId");
02980                     if (name && name->valuestring && strcmp(name->valuestring, source_name) == 0 &&
02981                         id && id->type == cJSON_Number) {
02982                         source_id = id->valueint;
02983                         break;
02984                     }
02985                 }
02986             }
02987             cJSON_Delete(data);
02988         }
02989     }
02990
02991     if (get_items_resp) {
02992         obsws_response_free(get_items_resp);
02993     }
02994
02995     if (source_id < 0) {
02996         return OBSWS_ERROR_INVALID_PARAM;
02997     }
02998
02999     /* Now set the visibility */
03000     req_data = cJSON_CreateObject();
03001     cJSON_AddStringToObject(req_data, "sceneName", scene_name);
03002     cJSON_AddNumberToObject(req_data, "sceneItemId", source_id);
03003     cJSON_AddBoolToObject(req_data, "sceneItemEnabled", visible);
03004
03005     data_str = cJSON_PrintUnformatted(req_data);
03006     cJSON_Delete(req_data);
03007
03008     obsws_response_t *resp = NULL;
03009     result = obsws_send_request(conn, "SetSceneItemEnabled", data_str, &resp, 0);
03010     free(data_str);
03011
03012     if (response) {
03013         *response = resp;
03014     } else if (resp) {
03015         obsws_response_free(resp);
03016     }
03017
03018     return result;
03019 }
03020
03035 obsws_error_t obsws_set_source_filter_enabled(obsws_connection_t *conn, const char *source_name,
03036                                               const char *filter_name, bool enabled, obsws_response_t
      **response) {
03037     if (!conn || !source_name || !filter_name) {
03038         return OBSWS_ERROR_INVALID_PARAM;
03039     }
03040
03041     cJSON *req_data = cJSON_CreateObject();
03042     cJSON_AddStringToObject(req_data, "sourceName", source_name);
03043     cJSON_AddStringToObject(req_data, "filterName", filter_name);
03044     cJSON_AddBoolToObject(req_data, "filterEnabled", enabled);
03045
03046     char *data_str = cJSON_PrintUnformatted(req_data);
03047     cJSON_Delete(req_data);
03048
03049     obsws_response_t *resp = NULL;
03050     obsws_error_t result = obsws_send_request(conn, "SetSourceFilterEnabled", data_str, &resp, 0);
03051     free(data_str);
03052
03053     if (response) {
03054         *response = resp;
03055     } else if (resp) {
03056         obsws_response_free(resp);
03057     }
03058
03059     return result;
03060 }
```

## 4.2 libwsv5.h

```
00001 /*
00002 * libwsv5.h - OBS WebSocket v5 Protocol Library
00003 *
00004 * A robust C library for managing OBS connections via the WebSocket v5 protocol.
00005 * Supports multiple concurrent connections, automatic reconnection, and thread-safe
00006 * operations for streaming, recording, scene control, and live production setups.
00007 *
00008 * Author:  Aidan A. Bradley
```

```
00009 * Maintainer:  Aidan A. Bradley
00010 * License:  MIT
00011 *
00012 * Key Features:
00013 * - Connection management with automatic reconnection
00014 * - OBS WebSocket v5 authentication
00015 * - Scene and source control
00016 * - Recording and streaming control
00017 * - Connection health monitoring
00018 * - Thread-safe multi-connection support
00019 * - Comprehensive error handling
00020 */
00021
00022 #ifndef LIBWSV5_LIBRARY_H
00023 #define LIBWSV5_LIBRARY_H
00024
00025 #ifndef _POSIX_C_SOURCE
00026 #define _POSIX_C_SOURCE 200809L
00027 #endif
00028
00029 #include <stddef.h>
00030 #include <stdbool.h>
00031 #include <stdint.h>
00032 #include <time.h>
00033
00034 #ifdef __cplusplus
00035 extern "C" {
00036 #endif
00037
00056 typedef enum {
00057     OBSWS_OK = 0,
00058
00059     /* Parameter validation errors (application layer) - not recoverable by retrying */
00060     OBSWS_ERROR_INVALID_PARAM = -1,
00061
00062     /* Network-level errors (can be recovered with reconnection) */
00063     OBSWS_ERROR_CONNECTION_FAILED = -2,
00064     OBSWS_ERROR_SEND_FAILED = -5,
00065     OBSWS_ERROR_RECV_FAILED = -6,
00066     OBSWS_ERROR_SSL_FAILED = -11,
00067
00068     /* Authentication errors (recoverable only by fixing the password) */
00069     OBSWS_ERROR_AUTH_FAILED = -3,
00070
00071     /* Protocol/messaging errors (typically indicate bad request data or OBS issues) */
00072     OBSWS_ERROR_PARSE_FAILED = -7,
00073     OBSWS_ERROR_NOT_CONNECTED = -8,
00074     OBSWS_ERROR_ALREADY_CONNECTED = -9,
00075
00076     /* Timeout errors (recoverable by retrying with patience) */
00077     OBSWS_ERROR_TIMEOUT = -4,
00078
00079     /* System resource errors (usually indicates system-wide issues) */
00080     OBSWS_ERROR_OUT_OF_MEMORY = -10,
00081
00082     /* Catch-all for things we didn't expect */
00083     OBSWS_ERROR_UNKNOWN = -99
00084 } obsws_error_t;
00085
00103 typedef enum {
00104     OBSWS_STATE_DISCONNECTED = 0,       /* Not connected to OBS, no operations possible */
00105     OBSWS_STATE_CONNECTING = 1,         /* WebSocket handshake in progress, wait for AUTHENTICATING */
00106     OBSWS_STATE_AUTHENTICATING = 2,     /* Connected but still doing auth, wait for CONNECTED */
00107     OBSWS_STATE_CONNECTED = 3,          /* Ready - authentication complete, send commands now */
00108     OBSWS_STATE_ERROR = 4               /* Unrecoverable error occurred, reconnection might help */
00109 } obsws_state_t;
00110
00121 typedef enum {
00122     OBSWS_LOG_NONE = 0,         /* Silence the library completely */
00123     OBSWS_LOG_ERROR = 1,        /* Only errors - something went wrong */
00124     OBSWS_LOG_WARNING = 2,      /* Errors + warnings - potential issues but still working */
00125     OBSWS_LOG_INFO = 3,         /* Normal operation info, good for seeing what's happening */
00126     OBSWS_LOG_DEBUG = 4         /* Very verbose, includes internal decisions and state changes */
00127 } obsws_log_level_t;
00128
00139 typedef enum {
00140     OBSWS_DEBUG_NONE = 0,       /* No debug output - production mode */
00141     OBSWS_DEBUG_LOW = 1,        /* Connection events, auth success/failure, major state changes */
00142     OBSWS_DEBUG_MEDIUM = 2,     /* Low + WebSocket opcodes, event type names, request IDs */
00143     OBSWS_DEBUG_HIGH = 3        /* Medium + full message contents - can include passwords!  */
00144 } obsws_debug_level_t;
00145
00146 /* Forward declaration of connection handle - opaque structure for connection management */
00147 typedef struct obsws_connection obsws_connection_t;
00148
00166 typedef void (*obsws_log_callback_t)(obsws_log_level_t level, const char *message, void *user_data);
00167
```

```
00195 typedef void (*obsws_event_callback_t)(obsws_connection_t *conn, const char *event_type, const char
      *event_data, void *user_data);
00196
00228 typedef void (*obsws_state_callback_t)(obsws_connection_t *conn, obsws_state_t old_state,
      obsws_state_t new_state, void *user_data);
00229
00242 typedef struct {
00243     /* === Connection Parameters === */
00244     const char *host;                    /* IP or hostname of OBS (e.g., "192.168.1.100" or
      "obs.example.com") */
00245     uint16_t port;                       /* OBS WebSocket server port, usually 4455, sometimes 4454
      for WSS */
00246     const char *password;                /* OBS WebSocket password from settings.  Set to NULL for no
      auth.  */
00247     bool use_ssl;                        /* Use WSS (WebSocket Secure) instead of WS. Requires OBS
      configured for SSL. */
00248
00249     /* === Timeout Settings (all in milliseconds) ===
00250 Timeouts are important to prevent hanging.  Too short and you get false failures.
00251 Too long and your app freezes.  Adjust based on network quality.  */
00252     uint32_t connect_timeout_ms;         /* How long to wait for initial TCP connection (default:
      5000) */
00253     uint32_t recv_timeout_ms;            /* How long to wait for data from OBS (default:  5000) */
00254     uint32_t send_timeout_ms;            /* How long to wait to send data to OBS (default:  5000) */
00255
00256     /* === Keep-Alive / Health Monitoring ===
00257 The library sends ping messages periodically to detect dead connections.
00258 If OBS stops responding to pings, the library will try to reconnect.  */
00259     uint32_t ping_interval_ms;           /* Send ping this often (default:  10000, 0 to disable pings)
      */
00260     uint32_t ping_timeout_ms;            /* Wait this long for pong response (default:  5000) */
00261
00262     /* === Automatic Reconnection ===
00263 If the connection dies, should we try to reconnect?  Very useful for production
00264 because networks hiccup, OBS crashes, etc.  The library uses exponential backoff
00265 to avoid hammering the server - delays double each attempt up to the max.  */
00266     bool auto_reconnect;                 /* Enable automatic reconnection (default:  true) */
00267     uint32_t reconnect_delay_ms;         /* Wait this long before first reconnect (default:  1000) */
00268     uint32_t max_reconnect_delay_ms;     /* Don't wait longer than this between attempts (default:
      30000) */
00269     uint32_t max_reconnect_attempts;     /* Give up after this many attempts (0 = retry forever) */
00270
00271     /* === Callbacks ===
00272 These optional callbacks let you be notified of important events.
00273 You can leave any of them NULL if you don't care about that event type.  */
00274     obsws_log_callback_t log_callback;   /* Called when library logs something */
00275     obsws_event_callback_t event_callback;  /* Called when OBS sends an event */
00276     obsws_state_callback_t state_callback;  /* Called when connection state changes */
00277     void *user_data;                     /* Passed to all callbacks - use for context (like "this"
      pointer) */
00278 } obsws_config_t;
00279
00288 typedef struct {
00289     uint64_t messages_sent;              /* Total WebSocket messages sent to OBS (includes ping/pong)
      */
00290     uint64_t messages_received;          /* Total WebSocket messages received from OBS (includes
      events) */
00291     uint64_t bytes_sent;                 /* Total bytes transmitted, useful for bandwidth monitoring
      */
00292     uint64_t bytes_received;             /* Total bytes received */
00293     uint64_t reconnect_count;            /* How many times auto-reconnect kicked in (0 if never
      disconnected) */
00294     uint64_t error_count;                /* Total errors encountered (some might be retried
      successfully) */
00295     uint64_t last_ping_ms;               /* Round-trip time of last ping - network latency indicator
      */
00296     time_t connected_since;              /* Unix timestamp of when this connection was established */
00297 } obsws_stats_t;
00298
00314 typedef struct {
00315     bool success;                        /* true if OBS said the operation worked */
00316     int status_code;                     /* OBS status code:  100-199 = success, 600+ = error */
00317     char *error_message;                 /* If success is false, this has the reason (e.g., "Scene
      does not exist") */
00318     char *response_data;                 /* Raw JSON response from OBS - parse yourself with cJSON */
00319 } obsws_response_t;
00320
00321 /* ============================================================================
00322 * Library Initialization and Cleanup
00323 * ============================================================================ */
00324
00346 obsws_error_t obsws_init(void);
00347
00361 void obsws_cleanup(void);
00362
00372 const char* obsws_version(void);
00373
```

```
00388 void obsws_set_log_level(obsws_log_level_t level);
00389
00405 void obsws_set_debug_level(obsws_debug_level_t level);
00406
00415 obsws_debug_level_t obsws_get_debug_level(void);
00416
00417 /* ============================================================================
00418 * Advanced Logging System - File Logging, Timestamps, Colors, Rotation
00419 * ============================================================================ */
00420
00445 obsws_error_t obsws_enable_log_file(const char *directory);
00446
00457 obsws_error_t obsws_disable_log_file(void);
00458
00473 obsws_error_t obsws_set_log_rotation_hour(int hour);
00474
00492 obsws_error_t obsws_set_log_rotation_size(size_t max_size_bytes);
00493
00512 obsws_error_t obsws_set_log_colors(int mode);
00513
00527 obsws_error_t obsws_set_log_timestamps(bool enabled);
00528
00540 const char* obsws_get_log_file_directory(void);
00541
00542 /* ============================================================================
00543 * Connection Management
00544 * ============================================================================ */
00545
00563 void obsws_config_init(obsws_config_t *config);
00564
00607 obsws_connection_t* obsws_connect(const obsws_config_t *config);
00608
00628 void obsws_disconnect(obsws_connection_t *conn);
00629
00646 bool obsws_is_connected(const obsws_connection_t *conn);
00647
00658 obsws_state_t obsws_get_state(const obsws_connection_t *conn);
00659
00677 obsws_error_t obsws_get_stats(const obsws_connection_t *conn, obsws_stats_t *stats);
00678
00695 obsws_error_t obsws_reconnect(obsws_connection_t *conn);
00696
00719 int obsws_ping(obsws_connection_t *conn, uint32_t timeout_ms);
00720
00721 /* ============================================================================
00722 * Scene Management
00723 * ============================================================================ */
00724
00765 obsws_error_t obsws_set_current_scene(obsws_connection_t *conn, const char *scene_name,
      obsws_response_t **response);
00766
00796 obsws_error_t obsws_get_current_scene(obsws_connection_t *conn, char *scene_name, size_t buffer_size);
00797
00836 obsws_error_t obsws_get_scene_list(obsws_connection_t *conn, char ***scenes, size_t *count);
00837
00863 obsws_error_t obsws_set_scene_collection(obsws_connection_t *conn, const char *collection_name,
      obsws_response_t **response);
00864
00865 /* ============================================================================
00866 * Recording and Streaming Control
00867 * ============================================================================ */
00868
00894 obsws_error_t obsws_start_recording(obsws_connection_t *conn, obsws_response_t **response);
00895
00916 obsws_error_t obsws_stop_recording(obsws_connection_t *conn, obsws_response_t **response);
00917
00946 obsws_error_t obsws_start_streaming(obsws_connection_t *conn, obsws_response_t **response);
00947
00964 obsws_error_t obsws_stop_streaming(obsws_connection_t *conn, obsws_response_t **response);
00965
00992 obsws_error_t obsws_get_streaming_status(obsws_connection_t *conn, bool *is_streaming,
      obsws_response_t **response);
00993
01009 obsws_error_t obsws_get_recording_status(obsws_connection_t *conn, bool *is_recording,
      obsws_response_t **response);
01010
01011 /* ============================================================================
01012 * Source Control
01013 * ============================================================================ */
01014
01047 obsws_error_t obsws_set_source_visibility(obsws_connection_t *conn, const char *scene_name,
01048                                           const char *source_name, bool visible, obsws_response_t
      **response);
01049
01078 obsws_error_t obsws_set_source_filter_enabled(obsws_connection_t *conn, const char *source_name,
01079                                               const char *filter_name, bool enabled, obsws_response_t
      **response);
```

```
01080
01081 /* ============================================================================
01082 * Custom Requests
01083 * ============================================================================ */
01084
01142 obsws_error_t obsws_send_request(obsws_connection_t *conn, const char *request_type,
01143                                   const char *request_data, obsws_response_t **response, uint32_t
      timeout_ms);
01144
01145 /* ============================================================================
01146 * Event Handling
01147 * ============================================================================ */
01148
01175 int obsws_process_events(obsws_connection_t *conn, uint32_t timeout_ms);
01176
01177 /* ============================================================================
01178 * Utility Functions
01179 * ============================================================================ */
01180
01198 void obsws_response_free(obsws_response_t *response);
01199
01220 const char* obsws_error_string(obsws_error_t error);
01221
01238 const char* obsws_state_string(obsws_state_t state);
01239
01269 void obsws_free_scene_list(char **scenes, size_t count);
01270
01271 #ifdef __cplusplus
01272 }
01273 #endif
01274
01275 #endif // LIBWSV5_LIBRARY_H
```

# Chapter 5

# Example Documentation

## 5.1 Parsing

Event callback function type - called when OBS sends an event.

Event callback function type - called when OBS sends an event.OBS sends events to tell you about things happening - scene changed, recording started, input muted, etc. Events come as JSON in event_data. You have to parse it yourself using something like cJSON. We don't parse it for you because different applications care about different events, so we save CPU by leaving parsing to you.

**Parameters**

| | |
|---|---|
| *conn* | The connection this event came from (useful if you have multiple OBS instances) |
| *event_type* | String name of the event like "SceneChanged", "RecordingStateChanged" |
| *event_data* | JSON string with event details - parse this yourself with cJSON or json-c |
| *user_data* | Pointer you provided in the config |

**Note**

The event_data buffer is temporary and freed after this callback returns. If you need to keep the data, copy the string or parse it immediately.

This callback is called from an internal thread, so synchronize access to shared data structures.

Don't block or do expensive work in this callback - events could pile up.

an event might look like: cJSON ∗event_obj = cJSON_Parse(event_data); if (event_obj && cJSON_HasObject↩ Item(event_obj, "eventData")) { cJSON ∗data = cJSON_GetObjectItem(event_obj, "eventData"); // Now examine event_obj to see what changed } if (event_obj) cJSON_Delete(event_obj);

## 5.2 Usage

Create a new OBS WebSocket connection and start connecting.

Create a new OBS WebSocket connection and start connecting.This function creates a connection object and begins the connection process in the background. The connection goes through states: DISCONNECTED ->

CONNECTING -> AUTHENTICATING -> CONNECTED. You'll be notified of state changes via the state callback (if you provided one in config).

This is non-blocking - it returns immediately. The actual connection happens in a background thread. Check obsws_get_state() to see when it reaches CONNECTED.

Design note: We do connection in the background because it can take a while (DNS lookups, TCP handshake, authentication). Blocking would freeze your app.

**Parameters**

| | |
|---|---|
| *config* | Connection configuration (required, cannot be NULL) |

**Returns**

Connection handle on success (never NULL if called with valid config), NULL on failure (usually out of memory)

**Note**

The config structure is copied internally, so you can free or reuse it after calling obsws_connect().

The returned handle is opaque - use it with other obsws_∗ functions.

Connection attempt happens asynchronously - wait for state callback or call obsws_get_state() to check when it's ready.

If auto_reconnect is enabled in config, the library will automatically try to reconnect if the connection drops.

: obsws_config_t config; obsws_config_init(&config); config.host = "192.168.1.100"; config.port = 4455; config.↩ password = "mypassword"; config.state_callback = my_state_callback; config.user_data = my_app_context;

obsws_connection_t ∗conn = obsws_connect(&config); if (!conn) { fprintf(stderr, "Failed to create connection\n"); return 1; } // Wait for state callback to indicate CONNECTED...

## 5.3 Switching

Switch to a specific scene in OBS.

Switch to a specific scene in OBS.This tells OBS to make a different scene active. When you switch scenes, all sources in the new scene become visible (if their visibility is on), and sources from the old scene disappear. This is the main way to change what's being streamed or recorded.

Switching scenes is fast but not instant - OBS processes the request and sends a SceneChanged event when complete. If you have many sources with animations, the transition might take a second or so.

**Parameters**

| | |
|---|---|
| *conn* | Connection handle (must be in CONNECTED state) |
| *scene_name* | Name of the scene to activate (must exist in OBS, case-sensitive) |
| *response* | Optional pointer to receive response. If you pass a pointer to response pointer, you get back a response object that you must free with obsws_response_free(). Pass NULL if you don't care about the response. |

**Returns**

OBSWS_OK if the request was sent successfully (doesn't mean OBS processed it yet)

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if scene_name is NULL

**Note**

The scene must exist in OBS. If you try to switch to a non-existent scene, OBS will return an error in the response.

This is async - the function returns before OBS actually switches scenes. Watch for a SceneChanged event to know when it completed.

If you only care about success/failure, you can pass NULL for response.

The response contains OBS status codes - 100-199 is success, 600+ is error.

to a scene: obsws_response_t ∗response = NULL; obsws_error_t err = obsws_set_current_scene(conn, "Gaming Scene", &response); if (err == OBSWS_OK && response) { if (response->success) { printf("Scene switched successfully\n"); } else { printf("OBS error: %s\n", response->error_message); } obsws_response_free(response); }

## 5.4 Getting

Get the name of the currently active scene.

Get the name of the currently active scene.Asks OBS which scene is currently shown. The answer comes back as a string that you provide a buffer for. If the buffer is too small, the function fails with OBSWS_ERROR_RECV_FAILED (or similar) because the response doesn't fit.

This is useful to check what scene is active before switching, or to sync your UI state with OBS (in case OBS was controlled by something else).

**Parameters**

| | |
|---|---|
| *conn* | Connection handle (must be in CONNECTED state) |
| *scene_name* | Output buffer where the scene name will be written |
| *buffer_size* | Size of the buffer (how many bytes can fit) |

**Returns**

OBSWS_OK on success

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if scene_name is NULL

**Note**

The buffer should be large enough for typical scene names. OBS doesn't limit scene name length, but practically they're usually under 256 characters. 256 or 512 bytes is usually safe.

The string written to scene_name is null-terminated.

This is synchronous - it waits for response before returning (blocking).

current scene: char current_scene[256]; if (obsws_get_current_scene(conn, current_scene, sizeof(current_scene)) == OBSWS_OK) { printf("Current scene: %s\n", current_scene); }

## 5.5 Starting

Start recording to disk.

Start recording to disk.Tells OBS to begin recording the current scene and audio to the configured output file (usually somewhere in your Videos folder). This is separate from streaming - you can record without streaming, stream without recording, or do both.

The actual file path is configured in OBS settings - this library doesn't control where the file goes or what format it uses. Those are OBS preferences.

**Parameters**

| | |
|---|---|
| *conn* | Connection handle (must be in CONNECTED state) |
| *response* | Optional pointer to receive response |

**Returns**

OBSWS_OK if request sent successfully (doesn't mean recording started yet)

OBSWS_ERROR_NOT_CONNECTED if not connected

**Note**

This is async - function returns immediately, recording starts in background.

You'll get a RecordingStateChanged event when recording actually starts.

If recording is already running, OBS ignores this request (no error).

The recorded file format depends on OBS configuration (usually MP4 or MKV).

to record: if (obsws_start_recording(conn, NULL) == OBSWS_OK) { printf("Recording start request sent\n"); }

## 5.6 Checking

Get whether OBS is currently streaming.

Get whether OBS is currently streaming.Returns true if a stream is active (connected to the streaming server), false otherwise. The response parameter (if provided) has more detailed info like bandwidth, frames rendered, etc.

This is useful to check state after startup - maybe OBS was already streaming before your app connected, and you want to know about it.

**Parameters**

| | |
|---|---|
| *conn* | Connection handle (must be in CONNECTED state) |
| *is_streaming* | Output parameter - true if streaming, false if not |
| *response* | Optional pointer to receive full response with stats |

**Returns**

OBSWS_OK on success

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if is_streaming is NULL

**Note**

is_streaming must be non-NULL, but response can be NULL.

Fast operation - just queries current state.

if streaming: bool streaming = false; if (obsws_get_streaming_status(conn, &streaming, NULL) == OBSWS_OK) { printf("Streaming: %s\n", streaming ? "yes" : "no"); }

## 5.7 Hiding

Set whether a source is visible in a scene.

Set whether a source is visible in a scene.Sources are the building blocks of scenes - they can be cameras, images, text, browser windows, etc. Each source appears in one or more scenes, and you can control whether it's shown or hidden. When you hide a source, it doesn't render on the stream/recording until you show it again.

This is useful for:

- Show/hide a watermark or banner
- Toggle overlays on and off
- Control which camera appears (if you have multiple cameras as sources)

Note: A source can exist in multiple scenes. Changing visibility in one scene doesn't affect it in other scenes.

**Parameters**

| | |
|---|---|
| *conn* | Connection handle (must be in CONNECTED state) |
| *scene_name* | Name of the scene containing the source (case-sensitive) |
| *source_name* | Name of the source to hide/show (case-sensitive) |
| *visible* | true to show the source, false to hide it |
| *response* | Optional pointer to receive response |

**Returns**

OBSWS_OK if request sent successfully

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if scene_name or source_name is NULL

**Note**

If the scene or source doesn't exist, OBS returns an error in response.

Changes are instant - the source appears/disappears on stream immediately.

You'll get a SourceVisibilityChanged event when this completes.

a watermark source: obsws_set_source_visibility(conn, "Main Scene", "Watermark", false, NULL);

## 5.8 Disabling

Enable or disable a filter on a source.

Enable or disable a filter on a source.Filters are effects applied to sources - color correction, blur, noise suppression, etc. Each filter can be enabled or disabled. Disabling a filter removes its effect without deleting it, so you can toggle it back on later.

Use this to:

- Dynamically control effects (blur camera when not looking at it)
- Toggle noise suppression on/off for a microphone source
- Enable/disable color correction for different lighting conditions

**Parameters**

| | |
|---|---|
| *conn* | Connection handle (must be in CONNECTED state) |
| *source_name* | Name of the source containing the filter (case-sensitive) |
| *filter_name* | Name of the filter to enable/disable (case-sensitive) |
| *enabled* | true to enable the filter, false to disable it |
| *response* | Optional pointer to receive response |

**Returns**

OBSWS_OK if request sent successfully

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if source_name or filter_name is NULL

**Note**

If the source or filter doesn't exist, OBS returns an error.

Changes are instant on stream/recording.

You'll get a SourceFilterEnabledStateChanged event when complete.

noise suppression: obsws_set_source_filter_enabled(conn, "Mic", "Noise Suppression", false, NULL);

## 5.9 Setting

a source name (with parameters): obsws_response_t *response = NULL; const char *params = "{\"sourceName\": \"OldName\", \"newName\": \"NewName\"}"; obsws_send_request(conn, "SetSourceName", params, &response, 5000); if (response) { printf("Success: %d\n", response->success); obsws_response_free(response); }

a source name (with parameters): obsws_response_t *response = NULL; const char *params = "{\"sourceName\": \"OldName\", \"newName\": \"NewName\"}"; obsws_send_request(conn, "SetSourceName", params, &response, 5000); if (response) { printf("Success: %d\n", response->success); obsws_response_free(response); }

## 5.10 Logging

Get a human-readable string for an error code.

Get a human-readable string for an error code.Converts error codes like OBSWS_ERROR_AUTH_FAILED into strings like "Authentication failed". Useful for logging and error messages. The returned strings are static (not allocated) so don't free them.

**Parameters**

| | |
|---|---|
| *error* | Error code to convert |

**Returns**

Pointer to error string (e.g., "Invalid parameter", "Connection failed") Never returns NULL - unknown error codes get "Unknown error"

**Note**

The returned string is static and valid for the lifetime of the program.

Don't free the returned pointer.

an error: obsws_error_t err = obsws_set_current_scene(conn, "Scene", NULL); if (err != OBSWS_OK) { fprintf(stderr, "Error: %s\n", obsws_error_string(err)); }

## 5.11 Displaying

Get a human-readable string for a connection state.

Get a human-readable string for a connection state.Converts connection state enums like OBSWS_STATE_↩ CONNECTED into strings like "Connected". Useful for debug output and status displays.

**Parameters**

| | |
|---|---|
| *state* | Connection state to convert |

**Returns**

Pointer to state string (e.g., "Disconnected", "Connecting", "Connected") Never returns NULL - unknown states get "Unknown"

**Note**

The returned string is static, don't free it.

connection status: obsws_state_t state = obsws_get_state(conn); printf("Connection state: %s\n", obsws_state_↩ string(state));

## 5.12 Using

Free a scene list array returned by obsws_get_scene_list().

Free a scene list array returned by obsws_get_scene_list().When you call obsws_get_scene_list(), it allocates memory for the scene names and the array itself. You must free all of this using this function when done. Don't try to free the individual strings manually or use plain free() - that won't work correctly.

Why have a special function for this instead of just free()? Because the memory layout needs special handling - there's an array of pointers, each pointing to separately allocated strings.

**Parameters**

| | |
|---|---|
| *scenes* | Array of scene name strings (from obsws_get_scene_list) |
| *count* | Number of scenes in the array |

**Note**

Safe to call with NULL scenes pointer (does nothing).

After calling this, the scenes pointer is invalid - don't use it anymore.

The count parameter must match what obsws_get_scene_list() returned.

and freeing scene list: char ∗∗scenes = NULL; size_t count = 0; if (obsws_get_scene_list(conn, &scenes, &count) == OBSWS_OK) { for (size_t i = 0; i < count; i++) { printf("Scene: %s\n", scenes[i]); } obsws_free_scene_list(scenes, count); // Must do this! }

# Index