SDK Documentation Summary and Example

WAC_WS_SDK...rary_Usage
PDF

describe this

This document serves as a technical usage guide for the **WAC WebSocket SDK** (`wac-ws-sdk`), a Python library designed to handle WebSocket communications between a server and WAC devices.

Here is a breakdown of the key information provided in the document:

1. Installation & Setup

- The library is installed via pip using a `.whl` (wheel) file.
- The module can be imported into a Python script using `import wac_ws_sdk`.

2. Public API Components

The SDK exposes several components for developers to interact with the devices:

- **Clients:** Includes `WacClient` for asynchronous operations and `WacClientSync` as a synchronous wrapper.
- **Models:** Defines data structures like `DeviceInfo`, `LogRecord`, `BackupNum`, and `VerifyMode`.
- **Handlers:** Functions like `handle_reg`, `handle_sendlog`, and `handle_senduser` are used to process incoming frames from the device to the server and build standard replies.
- **Errors:** Uses `WacClientError` for transport or usage issues, and `DeviceError` for failures explicitly declared by the device.

3. Command Reference

All communication is done via JSON objects, where requests utilize a `cmd` field and replies echo with a `ret` field alongside a boolean `result`. The document outlines five primary commands:

- **reg (Device → Server):** Used by the device to register itself with the server.
- **sendlog (Device → Server):** Allows the device to push access logs (such as entry/exit event records) to the server.
- **senduser (Device → Server):** Used when the device pushes a user record up to the server.
- **setuserinfo (Server → Device):** Allows the server to store or update user credentials (like passwords/PINs, RFID cards, fingerprint templates, or face photos) directly on the device.
- **deleteuser (Server → Device):** Commands the device to delete specific user credentials or wipe a user's data entirely.

4. Constants & Data Models

The document defines specific integer codes used for credential types (`BackupNum`) and verification modes:

- **Credential Types (`BackupNum`):**
  - `0..9`: Fingerprint slots.

    - `10` : Password / PIN.
    - `11` : Card.
    - `12` : All fingerprints.
    - `13` : All credentials (used to wipe a user completely).
    - `50` : Photo.
- **Verification Modes:**
    - `0` : FP (Fingerprint).
    - `1` : CARD.
    - `2` : PWD (Password).
    - `8` : FACE.

Would you like me to write out a mock Python script demonstrating how you might use these handlers and commands based on the JSON examples in the document?

test_pihr_connection
PY

pihr_client
PY

mock_pihr_server
PY

are these files depict the document

Yes and no. The provided files accurately depict the **communication protocol** (the JSON messages) outlined in the document, but they **do not use the SDK library** the document is actually about.

Here is a breakdown of how your code aligns with and differs from the documentation:

What Matches: The Protocol

- **JSON Payloads**: The files correctly implement the JSON request and reply structures. For instance, requests use the `"cmd"` field, while replies echo the command with `"ret"` and include a `"result"` boolean.
- **Commands Supported**: The custom client and server simulate the exact commands detailed in the PDF, including `reg` , `sendlog` , `senduser` , and `setuserinfo` .
- **Data Fields & Constants**: Specific variables like `enrollid` , `backupnum` (e.g., using `10` for Password/PIN), `sn` , and `record` are implemented exactly as specified in the document's examples.

What Differs: The Implementation

- **Reinventing the Wheel**: The PDF document is an instruction manual for installing and importing a specific, pre-built Python package via `import wac_ws_sdk` . However, `pihr_client.py` does not use this package; it is a completely custom client built from scratch using the raw `websockets` library.
- **Missing SDK Classes**: The document instructs developers to use its built-in API components, such as the `WacClient` class, models like `DeviceInfo` , and specific handlers like `handle_reg` . None of these are imported or used in your files; instead, you have created your own custom `PiHRClient` class and manual event loops to handle the logic.

In short, your code successfully reverse-engineers the raw WebSocket communication that the devices expect, but it bypasses the `wac-ws-sdk` library entirely.

Would you like me to help rewrite `test_pihr_connection.py` to actually import and utilize the `wac_ws_sdk` library as intended by the documentation?

if I want to connect to the app from another pc what do i have to do?

To connect to your server from another PC, you need to change how the server listens for connections and update the client to point to the server's actual network IP address instead of `localhost`.

Here is what you need to do:

1. Update the Server Script

Currently, your `mock_pihr_server.py` is hardcoded to listen only to the local machine. You need to bind it to `0.0.0.0` so it accepts connections from any device on your local network.

Change this line in `mock_pihr_server.py`:

Python

```python
async with websockets.serve(handler, "localhost", 8765): #
```

To this:

Python

```python
async with websockets.serve(handler, "0.0.0.0", 8765):
```

2. Find the Server's Local IP Address

You need to know the IP address of the PC running the server.

- **On Windows:** Open Command Prompt and type `ipconfig`. Look for the "IPv4 Address" (it usually starts with `192.168.x.x` or `10.x.x.x`).
- **On Linux/Mac:** Open the terminal and type `ifconfig` or `ip a`.

3. Update the Client Script on the Second PC

On the other PC, you need to tell the client where to find the server. `localhost` won't work anymore because that tells the second PC to look for a server on *itself*.

Change this line in `test_pihr_connection.py`:

Python

```python
uri = "ws://localhost:8765" #
```

To use the IP address you found in Step 2:

Python

```
uri = "ws://192.168.1.100:8765"  # Replace with your actual server IP
```

4. Check Your Firewall

If the second PC still can't connect, the firewall on the server PC is likely blocking incoming connections to port `8765`.

- **Windows:** You will need to open "Windows Defender Firewall," go to "Advanced Settings," and create an "Inbound Rule" that allows TCP traffic on port `8765`.
- **Linux (UFW):** You can run `sudo ufw allow 8765/tcp`.

Would you like me to write out the specific terminal commands for configuring the firewall rules on your operating system?

does the application connect to supabase

No, the current application does not connect to Supabase.

Based on the code provided, both the client and the mock server only handle the raw WebSocket communication between each other.

Here is what the application is currently doing with the data instead:

- The server receives the JSON payloads (like device registrations and attendance logs) and simply prints them to the console using the `logger`.
- It then sends back hardcoded mock responses without actually saving the data anywhere.
- There are no database drivers, HTTP requests, or Supabase client libraries (like `supabase-py`) imported or utilized in any of the scripts.

If you want to store the device logs or user information permanently, you would need to add database integration to your server script.

Would you like me to show you how to integrate the Python Supabase client into your `mock_pihr_server.py` so it can actually save those incoming device logs to a database?

what operations the user will be able to perform according to the pdf

According to the provided PDF, the SDK library allows you to perform and handle five primary operations between the server and the WAC devices:

- **Register device (`reg`):** Allows a device to register itself with the server.
- **Push logs (`sendlog`):** Enables the device to push access or attendance logs to the server.
- **Device pushes a user (`senduser`):** Allows the device to send a specific user's record and credentials up to the server.
- **Store/update a user on device (`setuserinfo`):** Allows the server to send user information and credentials (such as a password/PIN, card, fingerprint template, or photo) to be stored or updated directly on the device.
- **Delete credentials (`deleteuser`):** Allows the server to command the device to delete a specific credential (like a PIN) for a user, or wipe all of a user's credentials completely.

Would you like me to explain the specific payload structure required to execute any of these commands?

schema
SQL

this is my supabase schema. Read it throughly and find out how can I implement these to maintain my database structure. I am allowed to create new table

Based on the `schema.sql` file you provided, your current Supabase database is heavily optimized for **camera-based AI facial recognition and analytics**. It uses `pgvector` for embedding matching, tracks line-crossing/overcrowding, and uses webhooks/triggers to communicate with an external AI processing server (likely running DeepStream or similar on edge devices).

The WAC WebSocket devices operate differently. They are standalone biometric terminals that send discrete JSON text payloads (logs, pins, fingerprints) rather than RTSP video streams.

To integrate the WAC devices without disrupting your existing AI camera pipeline, you should create a parallel set of tables specifically for the WebSocket hardware. Here is how you can implement this structure:

1. Device Management (For the `reg` command)

Your current `cameras` table is for RTSP streams. You need a table to track the physical WAC biometric machines when they connect and register.

SQL

```sql
CREATE TABLE public.wac_devices (
    sn VARCHAR(255) PRIMARY KEY, -- The WAC serial number (e.g., WAC14089464)
    cpusn VARCHAR(255),
    devinfo JSONB,               -- Store the raw device info JSON
    status VARCHAR(50) DEFAULT 'online',
    last_seen TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

ALTER TABLE public.wac_devices ENABLE ROW LEVEL SECURITY;
```

2. User & Credential Management (For `setuserinfo`, `senduser`, `deleteuser`)

Your current `face` table relies on vector embeddings (`vector(512)`) and UUIDs. The WAC devices rely on an integer `enrollid` and credential types (`backupnum`).

You need a table to store these specific device credentials (PINs, RFID cards, proprietary fingerprint base64 strings) so your server can sync them across multiple devices.

SQL

```sql
CREATE TABLE public.wac_credentials (
    id SERIAL PRIMARY KEY,
    enrollid INTEGER NOT NULL,
    name VARCHAR(255),
    backupnum INTEGER NOT NULL,  -- 0-9 (Finger), 10 (PIN), 11 (Card), 50 (Photo)
    record TEXT,                 -- The actual PIN, Card No, or Base64 template
    admin INTEGER DEFAULT 0,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    UNIQUE(enrollid, backupnum)  -- A user can only have one of each credential type
);

-- Optional: If you want to link WAC users to your existing AI users
ALTER TABLE public.wac_credentials
ADD COLUMN mapped_usr_id VARCHAR(255); -- Could map to face.usr_id
```

3. Attendance Logs (For the `sendlog` command)

Your current `recognized_faces` table has strict foreign key constraints tied to your AI `face` table. To capture the WAC hardware logs cleanly, it is best to create a dedicated log table that matches the JSON structure sent by the device.

SQL

```
CREATE TABLE public.wac_attendance_logs (
    id SERIAL PRIMARY KEY,
    sn VARCHAR(255) REFERENCES public.wac_devices(sn),
    enrollid INTEGER NOT NULL,
    name VARCHAR(255),
    record_time TIMESTAMP WITHOUT TIME ZONE,
    mode INTEGER,
    inout INTEGER,
    event INTEGER,
    verifymode INTEGER, -- 0: FP, 1: CARD, 2: PWD, 8: FACE
    temperature DOUBLE PRECISION,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_wac_logs_enrollid ON public.wac_attendance_logs(enrollid);
CREATE INDEX idx_wac_logs_time ON public.wac_attendance_logs(record_time DESC);
```

How the Python Server Fits In

Once you run these SQL commands in your Supabase SQL Editor, you would update your `mock_pihr_server.py` to use the `supabase-py` library.

When a device sends a `reg` command, your Python script will execute an `upsert` into `wac_devices`. When it sends a `sendlog` command, your script will parse the JSON array and do a `bulk insert` into `wac_attendance_logs`.

Would you like me to write the Python functions using `supabase-py` that handle the database insertions for the `reg` and `sendlog` WebSocket commands?