Technical Module Description: web\_service.py

Created on: December 23, 2024

Authors: Henk Stevens, Olaf Mastenbroek, Onno Janssen

Organization: Stichting Oradio

License: GNU General Public License (GPL)

Version: 3

Email: oradioinfo@stichtingoradio.nl

Status: Development

Summary: Provides a robust, asynchronous HTTP server for managing Oradio device functions through a browser-based interface.

# 1. Overview and Purpose

The `web\_service.py` module implements a fully asynchronous, non-blocking HTTP server tailored for the Oradio embedded audio system. Its primary purpose is to allow remote configuration and control via a browser-based interface, either over a standard network connection or through a captive portal when operating as a Wi-Fi access point. By decoupling UI interactions from core playback logic, the module enhances usability in headless deployments and provides a flexible configuration layer for playlist management, network settings, and system commands.

Key functional areas include:  
- Dynamic playlist and preset management through RESTful endpoints.  
- Real-time USB scanning and content enumeration.  
- System operations such as timed shutdown and log retrieval.  
- Integrated event reporting via a shared multiprocessing queue.  
- Safe concurrent startup and shutdown via custom server wrappers.

# 2. Class `Server`

The `Server` class is an extension of `uvicorn.Server` designed for seamless integration within a multi-threaded environment. By overriding default signal handlers and providing context manager support, this class ensures that the HTTP server can be started and stopped cleanly without impacting the main Python process or other threads.

Detailed Features:  
- `install\_signal\_handlers`: Disabled to prevent uvicorn from capturing signals meant for the main application.  
- `\_\_enter\_\_` and `\_\_exit\_\_`: Enables `with Server(app, config) as server:` syntax, automatically managing startup and teardown.  
- `run\_in\_thread()`: Launches `self.serve()` in a separate daemon thread, allowing the server to handle requests while the main loop continues to execute other tasks.  
- Error management: Catches and logs any exceptions during `serve()`, preventing server crashes from bringing down the entire Oradio system.

# 3. Class `web\_service`

The `web\_service` class orchestrates the lifecycle of the HTTP server and provides tightly integrated endpoints for user interaction. On initialization, it sets up event flags, integrates network control, and publishes shared resources to the FastAPI application state.

## 3.1 `\_\_init\_\_(self, queue)`

The constructor establishes the internal messaging infrastructure and server control flow:  
- \*\*Message Queue (`msg\_q`)\*\*: A `multiprocessing.Queue` instance used for sending state updates and user commands back to the main Oradio control loop. This decouples UI events from core logic.  
- \*\*Event Flags\*\*:  
 - `event\_reset`: Signals to the running server loop that activity has occurred and should reset the idle timeout.  
 - `event\_stop`: Indicates that the server should shut down gracefully.  
 - `event\_active`: Marks the server as active, useful for status reporting and health checks.  
- \*\*Network Integration\*\*: Instantiates a `wifi\_service` object to manage network mode (infrastructure vs AP) and sends an initial state message to ensure the web interface reflects current connectivity.  
- \*\*FastAPI State Injection\*\*: Attaches `msg\_q` and `event\_reset` to `api\_app.state` so that endpoint handlers have direct access to these resources without circular imports.

## 3.2 `get\_state(self)`

Returns the current operational state of the web service component, enabling the main controller to react appropriately:  
- \*\*`STATE\_WEB\_SERVICE\_IDLE`\*\*: Server not running or has been stopped.  
- \*\*`STATE\_WEB\_SERVICE\_RUNNING`\*\*: Server actively handling client HTTP requests.  
- \*\*`STATE\_WEB\_SERVICE\_ERROR`\*\*: An unrecoverable error has occurred within the server loop.  
This method reads internal event flags and logs all transitions using `oradio\_log` for traceability.

## 3.3 `send\_web\_message(self)`

Packages the current state and any error codes into a standardized message and places it onto `msg\_q`. Message format:  
```json  
{  
 "type": MESSAGE\_WEB\_SERVICE\_TYPE,  
 "state": <state\_constant>,  
 "error": <error\_code or MESSAGE\_NO\_ERROR>  
}  
```

## 3.4 `start(self, force\_ap=False)`

Begins or resets the web server’s operation, handling network mode and idle timeouts:  
1. \*\*DNS and Port Forwarding\*\*: Uses `iptables` to redirect HTTP traffic to the web service port, especially when operating as a captive portal.  
2. \*\*AP Mode Handling\*\*: If `force\_ap` is `True` or no valid network is detected, triggers `wifi\_service.access\_point\_start()` to ensure clients can connect.  
3. \*\*Server Initialization\*\*:  
 - Builds `uvicorn.Config` for `api\_app`, setting host, port, log configuration, and worker settings.  
 - Creates an instance of the custom `Server` class.  
 - Launches `server.run\_in\_thread()` in a dedicated daemon thread.  
4. \*\*Timeout Logic\*\*: If the server is already active, sets `event\_reset` to refresh the idle timer, preventing premature shutdown.

## 3.5 `run(self, server)`

Contains the main server loop, executed in a separate thread:  
- Calls `send\_web\_message()` to announce server start.  
- Enters a `with server:` context to call `server.run\_in\_thread()`, allowing request handling.  
- Monitors `event\_stop` and `event\_reset` flags every second:  
 - If `event\_reset` is set, clears it and resets the idle countdown.  
 - If the countdown reaches zero or `event\_stop` is set, breaks the loop.  
- Upon exit, calls `server.should\_exit = True` to shutdown the HTTP server and sends a final state message.

## 3.6 `stop(self)`

Sets `event\_stop`, signaling the `run` loop to complete and perform a clean shutdown of the HTTP server. Ensures no new requests are accepted and resources are freed correctly.

## 3.7 `reset\_timeout(self)`

Sets `event\_reset`, notifying the `run` loop to reset its idle timeout counter. Used to extend the server’s lifetime when clients interact with the web UI.

# 4. Standalone Test and CLI

When executed as a standalone script, `web\_service.py` provides an interactive CLI for testing server lifecycle and network redirection behaviors. A helper `Process` runs `check\_messages()` to display messages from `msg\_q` in real time. The user can select options to start, stop, reset timeout, force AP mode, or query current status, ensuring full coverage of the web service’s operational scenarios.

# 7. Sequence Diagram

