**Web Service and Captive Portal Management System**

**Overview**

The **Web Service and Captive Portal Management System** is designed to provide a customizable, self-hosted web interface via a captive portal that can either use an existing Wi-Fi connection or operate as an isolated access point. The system integrates a FastAPI-based web service with backend logic to handle communication with the Wi-Fi interface, web requests, and timeout management.

The module supports various functions, including starting the web server, controlling access points, managing timeout behavior, and interacting with external services via a message queue. It is primarily intended for environments where users need to manage a network device's web service and Wi-Fi connections seamlessly.

**Key Features**

1. **Web Service Integration**: Utilizes FastAPI with Uvicorn to serve a web interface that can either run on an active Wi-Fi connection or as a captive portal.
2. **Access Point Management**: The system can switch between using an existing Wi-Fi network and setting up its own access point.
3. **Timeout and State Management**: A configurable timeout mechanism ensures that the web service stops automatically after a period of inactivity. The service can also be manually reset or stopped.
4. **Message Queue**: Communicates status updates and control messages via a queue, enabling integration with other system components or external services.
5. **Logging**: Provides detailed logs to track service operations, errors, and user interactions, aiding in debugging and system monitoring.

**Components**

**1. Server Class**

The Server class extends uvicorn.Server to provide non-blocking, threaded execution of the FastAPI application. This class allows the server to run in the background without halting the main process, making it possible to interact with the web service and other processes simultaneously.

* **Key Methods**:
  + install\_signal\_handlers: Disables default signal handling to avoid blocking.
  + run\_in\_thread: Runs the server in a separate thread to keep the main thread free for other tasks.

**2. Web Service Class (web\_service)**

The web\_service class is the core component of the module, managing the web service lifecycle, Wi-Fi connection, and timeout behavior. This class controls the web server, manages the state of the web service, and sends/receives messages via a queue.

**Key Methods:**

* **\_\_init\_\_**:  
  Initializes the web service, sets up the message queue, creates events for resetting timeouts and stopping the service, and initializes the Wi-Fi service.
* **send\_web\_message**:  
  Sends the current state of the web service and any errors via the message queue to notify other components or external services.
* **start**:  
  Starts the web service, or if already running, resets the timeout counter. It can also force the system to use its own access point if specified.
* **stop**:  
  Stops the web service by setting the stop event flag.
* **reset\_timeout**:  
  Resets the timeout counter, keeping the web service active if it's running.
* **get\_state**:  
  Returns the current state of the web service, either active or idle.
* **run**:  
  The main loop that runs the web server and manages timeout logic. It checks periodically for reset events, stop signals, and ensures that the server is still running within the timeout constraints.

**3. Wi-Fi Service (wifi\_service)**

The wifi\_service class manages Wi-Fi operations such as switching between access point mode and connecting to an existing network. It ensures that the system can provide network connectivity either through its own access point or by joining a specified Wi-Fi network.

* **Key Methods**:
  + **access\_point\_start**: Starts the access point mode.
  + **access\_point\_stop**: Stops the access point mode and switches back to the original Wi-Fi connection if possible.
  + **send\_message**: Sends a Wi-Fi status message through the message queue.

**4. Utilities and Logging (oradio\_utils, oradio\_logging)**

* **Logging**: The oradio\_log utility is used to track operations and errors in the system. It ensures that every critical action, such as starting/stopping services or running commands, is logged for debugging and auditing purposes.
* **Shell Commands**: The run\_shell\_script utility is used to execute shell commands, such as configuring network port redirection or modifying system settings.

**5. Message Queue and Interprocess Communication**

The module uses Python's Queue to enable communication between different processes or threads. The message queue allows for:

* Sending and receiving control messages (e.g., service state updates).
* Passing status information between the web service and other system components.

The check\_messages function listens to the message queue, allowing it to handle incoming messages such as error notifications or status updates.

**System Architecture**

1. **Main Process**:  
   The system runs as a standalone script where the main process initializes the web service and listens for user commands. It interacts with the message queue to send and receive messages regarding the state of the web service.
2. **Web Service**:  
   The FastAPI web service is run in a separate thread by the Server class, which enables non-blocking operation. The web service listens for incoming HTTP requests and serves the captive portal or web interface.
3. **Wi-Fi Management**:  
   The wifi\_service class ensures that the system can operate either in access point mode or as a Wi-Fi client. This allows for flexible networking based on the system's needs.
4. **Timeout Management**:  
   A timeout mechanism is in place to stop the web service after a defined period of inactivity (600 seconds by default). The timeout can be reset by events or user actions, and the service can be stopped manually or automatically.
5. **Logging**:  
   Logging is implemented throughout the system to track key actions, errors, and system states. This provides transparency and helps with debugging in case of issues.

**Usage and Command Flow**

**Initialization**

* The system initializes by creating a message queue and starting a message listener process. It also sets up the Wi-Fi service and web server.

**User Interaction**

* Users interact with the system through a text-based menu, where they can:
  + Check the current web service state.
  + Start or stop the web service.
  + Reset the timeout counter to keep the service running.
  + Force the system to switch to access point mode.

**Process Flow:**

1. **Start Web Service**:
   * When the user selects to start the service, the system checks if the service is already running. If not, it starts the FastAPI server and configures network port redirection.
2. **Stop Web Service**:
   * The user can stop the service, which will clean up resources, remove network configurations, and stop the Wi-Fi access point if necessary.
3. **Timeout**:
   * If there is no user activity or interaction for the defined timeout period, the system will automatically stop the web service.

**Conclusion**

The **Web Service and Captive Portal Management System** is a modular, configurable system that provides a web interface and captive portal functionality. It integrates FastAPI with Wi-Fi management and event-driven timeout control, making it a robust solution for managing network devices that require web-based interaction. The system's message queue and logging mechanisms allow for flexible integration and easy monitoring, making it suitable for embedded systems and network management tasks.