# Sentitech System Architecture System

## Sequence Diagrams

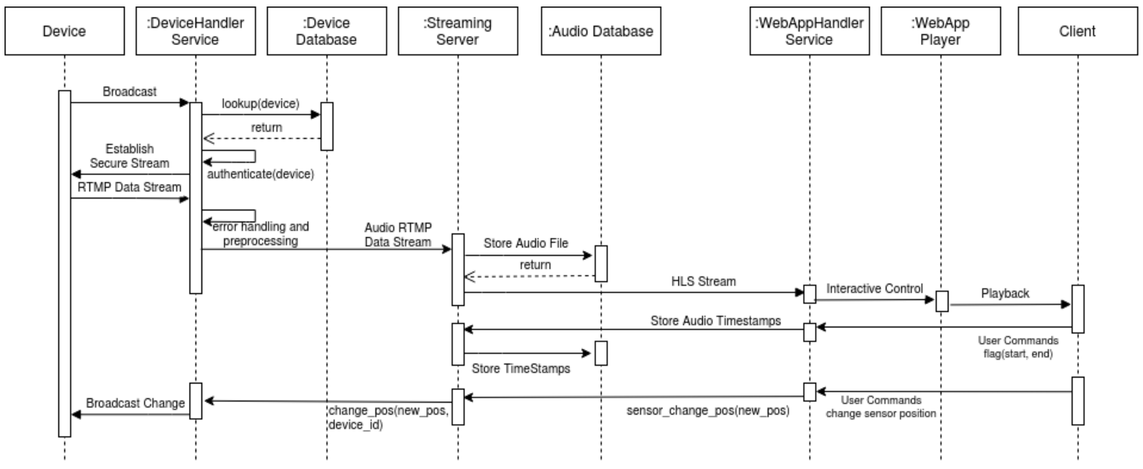

Figure 1: Sequence diagram to show steps taken to stream live audio from device to client

We will start by defining a sequence diagram to show the steps required to stream audio from a device to the client. Here we define 3 microservices:

- Device Handler Microservice. This service accepts, authenticates and preprocesses data streams from devices. For devices to do so, they will interact with a RESTful API, this microservice will also be responsible for tracking device metadata and allow commands to be sent to them. It is the only service that will query the Device Database, which keeps track of device metadata and used to authenticate incoming connections.
- Streaming Server (Stream Handler) Microservice. This service is responsible for receiving audio streams through RTMP or RTSP and storing incoming data streams from device handler (through a RESTful API). It will also be responsible for distributing these audio files to clients. To do so effectively, the audio must be made available in various formats and qualities in order to service different clients with differing devices and bandwidth capacities. also be responsible for its distribution. This Microservice is the only one which can modify and update the audio database.
- Web App Handler Microservice. This service is responsible for communicating with the client front end web and mobile applications. It will allow clients to interact with the stream, such as flag audio for storage as "historical sounds" and change the sensor position. It will also be responsible for authenticating the client and preventing unauthorised access to sensitive patient data. This Microservice is the only one to interact with client database and patient permissions database. The client database is used to authenticate the client, and the patient permissions database is used to state what resources a client has access to

A microservice architecture is beneficial as each service is loosely coupled to one another, allowing them to be implemented and tested independently of each other. By ensuring only one microservice is allowed to interact with any one database allows greater data integrity, we can ensure any changes made to the database are safe and will not be subject to interference from other operations on that db.

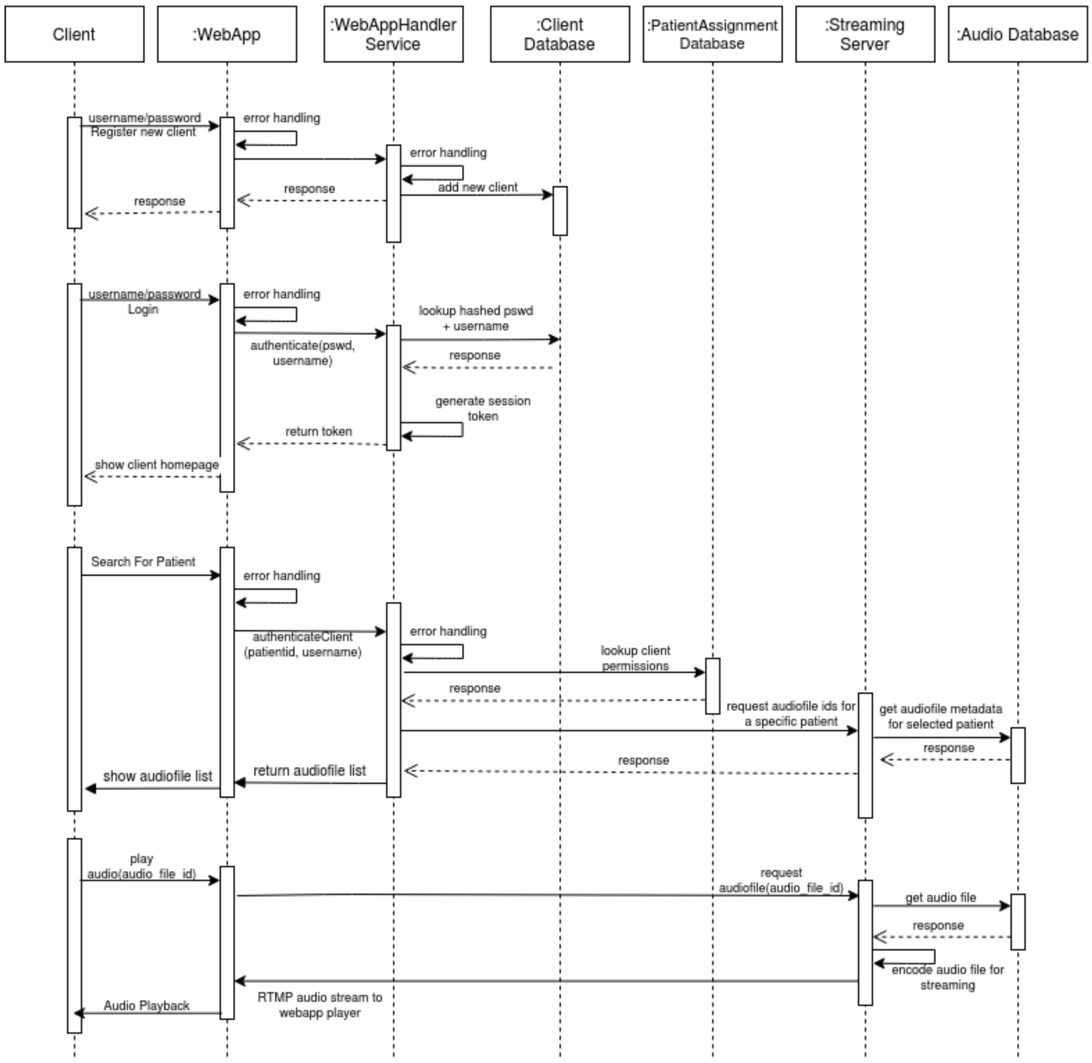
Client
:WebApp
:WebAppHandler Service
:Client Database
:PatientAssignment Database
:Streaming Server
:Audio Database
username/password Register new client
error handling
error handling
response
add new client
response
username/password Login
error handling
lookup hashed pswd + username
authenticate(pswd, username)
response
generate session token
return token
show client homepage
Search For Patient
error handling
authenticateClient (patientid, username)
error handling
lookup client permissions
response
request audiofile ids for a specific patient
get audiofile metadata for selected patient
response
response
return audiofile list
show audiofile list
play audio(audio_file_id)
request audiofile(audio_file_id)
get audio file
response
encode audio file for streaming
RTMP audio stream to webapp player
Audio Playback

Figure 2: Sequence diagram to show steps taken for various client interactions (Registration, Login, Streaming Saved Audio clips)

## Class Diagram

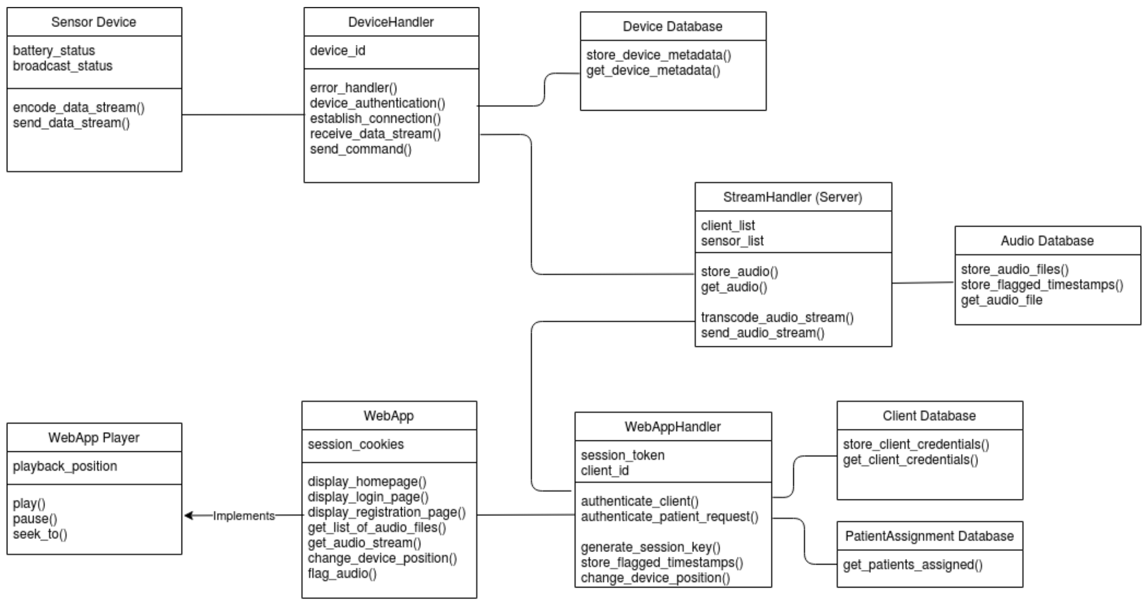

Figure 3: Class diagram to show how each module interacts with another (Registration, Login, Streaming Saved Audio clips)

## Robustness

To maximise uptime, we can use backup servers and databases so that if one fails, other backups can help service incoming requests. We will need to create an automatic load balancer to evenly distribute incoming requests.

We can a CDN (content delivery network) to store data near users, allowing them quick access to audio files without needing to query the host server. We can use meta-data to analyse when requests are likely to surge, and spin up more servers to handle this. For instance, at night-time to handle the influx of device streams.

## Security

We must Authenticate clients (clinicians) on the webapp. To do this we must allow a connection to the authentication microservice using HTTPS. We must also use the hash and salt algorithm before comparing it with the stored value in the database. We can then use session-based or token-based authentication techniques (using OAuth or JWT) to establish 2 way secure connection for the client. We must also Authenticate patient devices in a similar way before allowing them to transmit, this can be done in a similar way as clients.

We must also ensure that clinicians only get access to client information they have permission to access, this can be achieved allowing the user a token which only allows them to access limited patient data. To ensure cloud security, we must encrypt any files stored that is sensitive such as: sensitive audio files, metadata, and client credentials.

We can also use Firewalls to prevent unauthorised access to our server through unused ports. We can utilise anti-DDos software such as fail2ban and intrusion prevention systems (IPS) on our network to prevent data breaches.

## Scalability

Scalability can be done either vertically or horizontally. Vertical scaling is a process by which we allocate more hardware resources for our servers so that they are better able to handle increased demand. This has diminishing returns over-time, where allocating more resources will be become increasingly more costly. Horizontal scaling is an alternative where we utilise distributed computing to coordinate more physical processing units so that they work together to service clients. Horizontal Scalability is also more affordable than vertical scaling because we can utilise cloud computing technologies to handle our computational and storage needs.

By treating our application as a collection of microservices, we can address scalability challenges for Sentitech. Microservices allows for a decentralised architecture, whenever certain services are in demand, we can individually scale that particular microservice to service demand, thereby saving costs.

## Reliability

By utilising microservices, we can individually test, build and update these smaller components rather than requiring the whole solution to be re-tested and built, thereby saving time during updates. This will also make the code-base smaller, hence updates are less likely to have bugs than compared with monolithic architectures.

We can utilise continous integration (CI) technologies and automated testing tools to improve the speed at which updates are rolled out, minimising downtime of services.