AURA - Automated Radio

The AURA Team

Jun 23, 2023

GUIDES

AURA is a software suite for community radio stations. All code is Open Source and licensed under AGPL 3.0.

Alpha Status

We are currently in alpha status, so do not use AURA in production yet. If you are an early-stage user or developer, we would be happy about your *contributions*.

CHAPTER

ONE

USER GUIDE

Learn about the UI for radio hosts and programme coordinators.

1.1 Dashboard Overview

The dashboard is a simple browser application, allowing:

- *Hosts* to upload and manage their shows.
- *Programme Coordinators* to manage the radio station and its programme.

1.1.1 Login

Click the *Sign In* button to open your personal dashboard.

Use the credentials provided by your radio station.

1.1.2 Logout

To sign out click your profile name in the right top. Then click *Sign Out*.

1.1.3 Navigation Bar

Here you can open tabs to manage:

- Shows
- Media
- Programme Calendar

1.1.4 User Profile

/* tbd */

1.1.5 User Settings

/* tbd */

1.1.6 Switch Language

To change the language click the language picker on the right top. Then choose your preferred language.

1.1.7 Version

When reporting bugs you will be asked about the Dashboard version your are using.

You find the version in the footer of the application.

1.2 Show Planning

To manage your show and its episode navigate to the tabs for *Show* and *Media Management*.

1.2.1 Show Management

/* tbd */

1.2.2 Media Management

/* tbd */

1.3 Programme Planning

The *Programme Coordinators* home is the calendar tab.

1.3.1 Calendar & Schedule Management

Here you can get a quick overview what the current programme looks like. This is also the place to create schedules and timeslots based on recurrence rules.

/* tbd */

Creating schedules and timeslots

To create a new timeslot, click the desired area in the calendar. The following dialog opens.

The calendar is able to detect collisions with existing, conflicting timeslots.

To learn more about this behaviour, check out the chapter below.

Updating schedules and timeslots

Click the timeslot to open its dialog.

Deleting schedules and timeslots

Click the timeslot to open its dialog.

On the bottom you find various options to either

- Delete all timeslots and its schedule
- Delete all timeslots in the future
- Delete the current timeslot only.

1.3.2 Timeslot Collision Detection

When using the Dashboard Calendar to create new schedules or timeslots, there is some logic in place to avoid collisions of conflicting timeslots.

When a collision is detected, the user interface offers you options to solve the conflict.

Single collision, fully overlapping

In this situation an existing and your newly created timeslot are fully overlapping.

Single collision, partly overlapping

Here only some duration of the new timeslot is overlapping an existing one.

Here you have the option to truncate your timeslot or the existing one. Additionally, the solutions offered in the previous scenarios are offered here too.

Single collision, superset and subset

In that scenario, conflicting timeslots are offered to be split. Again, all possible solutions from the previous scenarios are valid too.

1.4 Radio Station Administration

Programme Coordinators have some additional features for radio management at hand.

1.4.1 Manage Users

Create User

To create user accounts log in to the admin interface `/steering/admin/` and click "*Benutzer*". There you need to enter a username and password, then click "*Sichern und weiter Bearbeiten*".

On the following page you can choose to either create one of these account types:

- **Mitarbeiter:** Click the checkbox *Mitarbeiter-Status*
- **Administrator:** Click the checkbox *Administrator-Status*

After that click "*Sichern*".

Granular permissions

There's also a box where you can assign granular permissions to each user account. At the current stage these settings have no effect yet.

```
/* tbd */
```

LDAP Authentication & Authorization

```
/* tbd */
```

1.4.2 Manage Shows

Create Show

In order to create a new show, click *plus* button in the show management tab.

In the dialog enter your show details, select the *show type* and *funding category*. Then click *OK*.

If you are starting from scratch, you might not have any *show types* and *funding categories* defined yet.

Learn how to define categorization options in the *Manage Station Settings* chapter below.

1.4.3 Manage Station Settings

Categorization Options

The following categorization options are defined globally and used to be assigned to individual shows.

Currently they have to be configured in the administration panel located at `/steering/admin/`.

Categories

```
/* tbd */
```

Funding Categories

```
/* tbd */
```

Languages

```
/* tbd */
```

Music Focus

```
/* tbd */
```

Topics

```
/* tbd */
```

Types

```
/* tbd */
```

1.5 Studio Clock

```
/* tbd */
```

CHAPTER

TWO

ADMINISTRATION GUIDE

Learn how to make and keep AURA playful.

2.1 Overview & Features

The following diagram outlines the main elements of AURA.

2.1.1 The naming scheme

Automated Radio, AutoRadio, AuRa or AURA, follows an "*Automobile-naming-scheme*". All the components are named after some crucial parts of a car:

- **Steering**: This is the single source of thruth, holding all the radio data, hence steering some radio's broadcast.
- **Tank**: Just spleen without steam. That's where your shows, tunes and recordings are managed. Fuelling your broadcast with materials.
- **Dashboard**: Observe and control what's happening. That's the backend user interface of your radio with individual views for hosts and programme coordinators.
- **Engine**: The playout server allowing you to broadcast via FM and web audio streams. Includes silence detection and optional recording options.
- **Play**: Every car has some fancy *Auto Radio Play*. And every radio station has the need for some modern frontend experience. *Play* is a library of web components. Making web development a breeze. Just like *playing with blocks of Lego*.

2.1.2 Features

The aforementioned naming scheme is used as a basis for name-spacing individual services.

Here you get an overview of the features of the provided services.

Find details on additional functionality on the *bundle documentation* pages or in the configuration files of individual services.

Steering

```
/* to be defined */
```

Dashboard

```
/* to be defined */
```

Dashboard Clock

Web Application providing a Studio Clock.

```
/* more to be defined */
```

Tank

```
/* to be defined */
```

Tank Cut & Glue

```
/* to be defined */
```

Engine

Scheduling and control for the playout.

- **Scheduler:** Automatically broadcast your radio programme (see AURA Dashboard for a user interface to do scheduling)
- **Autonomous playout:** Schedule information is pulled from Steering into a local cache. This way the playout keeps working, even when the network connectivity might be lost.
- **Versatile Playlists:** Playlists can contain different content types, such as audio files, audio streams and line in channels of you audio interface.
- **Default Playlists:** Define playlists on show and schedule level, in addition to any timeslot specific playlists. This way you always have an playlist assigned, when some host forgets about scheduling a specific programme.
- **Heartbeat Monitoring:** Frequently send pulse to a monitoring server, ensuring your schedule and playout server is up and running.

```
/* more to be defined */
```

Engine Core

The playout server based on Liquidsoap.

- **Multi-channel input:** Play audio from various sources including queues, streams and line-in from the audio interface
- **Multi-channel output:** Output to line out or audio streams
- **Icecast connectivity:** Stream to an Icecast Server, provide different encoding formats and bitrates

- **Auto DJ triggered by Silence Detector:** Play fallback audio triggered by a silence detector to avoid *Dead Air*. Play randomized music from a folder or M3U playlist.
- **Metadata handling:** Send information on playlogs via REST calls to external services like Engine API.
- **ReplayGain:** Normalization done using passed *ReplayGain* meta data.

Engine API

An OpenAPI 3 service to store and retrieve Engine data.

```
/* more to be defined */
```

Engine Recorder

A simple but powerful recorder.

- **Bulk Recorder**: Record blocks of audio for archiving, audit-logging or further processing
- **Sync to archive**: Periodically synchronize recordings to a defined destination

2.2 Migration & Integration Plan

Before getting started you need to carefully plan how you want to fit AURA into your infrastructure.

Check out the provided *Deployment Scenarios* on ideas for integration AURA into your ecosystem.

2.2.1 Individual test & production instances

To keep your radio infrastructure up-to-date, AURA is meant to provide frequent releases.

In order to keep this process free of glitches, it's good to gain experience in maintaining AURA on a test environment.

But also after moving AURA to production it is highly recommended to have a staging- or test-instance, where new releases are deployed before actually updating the production servers.

2.2.2 Migration Plan

Rome wasn't build in a day. And a radio software ecosystem isn't migrated in one day either.

Make a plan with different stages. Decide which parts you want to migrate at a certain stage.

Possible stages are:

1. Migrate and continuously synchronize scheduling and show data
2. Integrate your website and other frontends with AURA
3. Test run the actual play-out
4. Make the switch to the new play-out system

2.2.3 Migration Tools

Several radios are building tools and scripts for migration. Besides offering Best Practices, we can also share some code for a sample migration service, if that's what you need.

Let's talk in Matrix.

2.3 Deployment Scenarios

AURA comes with two Docker Compose bundles:

- **AURA Web**: Combines all essential containers required for the web-facing elements.
- **AURA Playout**: Everything related to scheduling, play-out and (optionally) recording.
- **AURA Record**: A standalone recorder and archive synchronisation.

There are also additional, individual Docker containers available to further extend the provided features.

The following network diagrams outline ideas on deployment scenarios.

2.3.1 Single Instance

This is the most simple scenario for setting things up, as all services reside on the same machine. The downsite is, that any hardware issues or the need for restarting the server has an immediate effect on the full array of services.

2.3.2 Advanced

Here we utilize at least two machines for services: One for all web-facing applications (AURA Web), the other related to scheduling and play-out (AURA Playout). This is the most verstatile approach between simplicity and maintainability, since your play-out will still be functional, while shutting down the web server for updates.

2.3.3 High Availability

That's the most sophisticated approach where your radio should never ever go down. But it's also the most tricky one, which needs careful testing and more valves to maintain.

Future scenario

Please note that this is a future scenario. Not all required service are available yet.

2.4 Deployment Preparations

2.4.1 Requirements

- Docker Engine 23.0.0 or later
- Docker Compose 2.15.1 or later
- Git
- make

2.4.2 Setting up the home and aura user

Clone the `aura` repository to the server(s) you want to deploy *AURA Web* or *AURA Playout* at.

We recommend using `/opt/aura` as your home for the AURA installation.

```
sudo git clone https://gitlab.servus.at/aura/aura.git /opt/aura
```

Next, move to that directory and create the `aura` user. This user is used for running all services.

```
make aura-user.add
```

The directory under `/opt/aura` is now owned by `aura:aura`.

User permissions

All resources will be owned by the user `aura`, hence we recommend performing all subsequent commands as this user or to add the current user to the `aura` group.

2.4.3 Selecting the release

First check for available releases.

```
git tag
```

This command will show you all tags which should correspond to the releases available. Alternatively you can take a look at releases.aura.radio.

Then switch to the release you'd like to use.

```
git checkout tags/<release-version>
```

Latest, unreleased state on the main branch

In case you want deploy the current development state, just take the latest commit on the `main` branch. That's the state which is checked out by default. While we try to keep the `main` branch as stable as possible, we cannot guarantee it being functional at all times. So please use it at your own risk.

2.4.4 Setting up the Audio Store

The *Audio Store* is a folder which is utilized by *Tank* and *Engine* to exchange audio files.

Assuming both, *Engine* and *Tank* are hosted on different machines, audio folders must by shared using some network share.

In case you are hosting *Engine* and *Tank* on the same machine, you can skip this step. Just think about pointing the settings the relevant audio directories, or create a symlink to do so behind the curtains.

By default the audio store is located in `/opt/aura/audio`. There are following subdirectories expected:

- `source`: holding all audio files from the media asset repository. Written by *Tank* and read by *Engine*.
- `fallback`: populate this folder with audio files to be played randomly, in cases where nothing is scheduled.
- `playlist`: used for M3U audio playlists.
- `recordings`: the recorder stores its recorded blocks here.

Share Type

Then, there's the question how the share is managed. Feasible options include:

- NFS
- SSHFS
- Gluster

Please evaluate for yourself what the most failsafe solution is. The next chapter outlines pro and cons of different scenarios.

Share Location

You have following options where your share can be located:

1. **Engine and all other AURA components (Tank, Dashboard, Steering) are running on the same instance.** This is the most simple solution, as Engine and Tank can share the same directory locally. But this scenario requires some more sophisticated tuning of the system resources to avoid e.g. some overload of multiple Uploads in Tank may affect the performance of engine. You can eliminate this risk by setting CPU and memory limits for Steering, Dashboard and Tank using Docker or `systemd-cgroups`. A disadvantage here is the case of maintenance of system reboot. This would mean that all components are offline at once.
2. **Physical directory where the Engine lives, mounted to Tank**. This may cause an issue with the mount, when the network connection to Engine is unavailable or the instance is rebooting.
3. **Physical directory where the Tank lives, mounted to Engine.** This may cause an issue with the mount, when the network connection to Tank is unavailable or the instance is rebooting.
4. **Central Data Store or *Storage Box*** which is mountet to Engine and Tank. In this case a downtime of the store make both, Engine and Tank dysfunctional.
5. **Replicated storage solution using Gluster, both Engine and Tank have their virtual audio directory mounted.** That's the ideal approach, because if any of the instances is down, the other has all the data available.

In any case, you should think about some backup solution involving this directory.

2.4.5 Advanced Configuration

In the following steps, you will be advised to copy the provided `sample.env` file to `.env`. In most common setup scenarios all configuration done in such `.env` file is sufficient.

For some more advanced setups or debug purposes, there are also sample configuration files for each service under `/opt/aura/config/services/sample-config` available. To overwrite any service configuration, simply copy its configuration file to the parent `services` folder.

Only use these overrides if you are an advanced user, or are advised to do so.

2.4.6 Deployment

AURA can be deployed using Docker and Docker Compose, allowing custom-tailored orchestration.

In general we recommend the Docker Compose variant.

2.5 Deployment Bundles

Same as a car has fixed and moving parts, AURA likewise has parts which are fundamental and parts which are optional.

In order to keep administrators life simple, we have defined component spaces reflecting to most common deployments:

- AURA Web: Contains all the web facing services, including digital asset management
- AURA Playout: Contains all things required for scheduling & broadcasting
- AURA Record: Record audio and periodical sync to archive

These spaces are realized as of Docker Compose bundles.

2.5.1 AURA Web

Service	Required	Description
steering		Single-source of truth, holding all radio, show, hostand scheduling data
tank		Upload, download, normalization and media asset management
dashboard		Backend user interface
dashboard-clock		Studio clock
play		A library for building frontends based on web components

The *required* services are needed for minimal functionality.

2.5.2 AURA Playout

Service	Required	Description
engine		Control and scheduling for the play-out
engine-core		Play-out Engine
engine-api		API for playlogs and track service
engine-recorder		Record audio blocks, including archiving

The *required* services are needed for minimal functionality.

2.5.3 AURA Recorder

Service	Required	Description
engine-recorder		Record audio blocks, including archive sync

The *required* services are needed for minimal functionality.

2.6 AURA Web

2.6.1 Deploy AURA Web

All required files can be found under `/opt/aura/config/aura-web`. For the described commands change to that directory.

Copy the `sample.env` to `.env` and set all values as needed.

Update the configuration file

Update at least following environment variables in the `.env` file:

- `AURA_AUDIO_STORE_SOURCE`: The location where Tank is storing audio sources. It defaults to `aura/audio/source`.
- `AURA_TANK_ENGINE_PASSWORD`: The password should match the one you'll be configuring in AURA Playout. Avoid using the default one.
- `AURA_HOST_NAME`: Set the host or domain name where the service are used from. See below why to avoid `localhost`.
- `AURA_HOST_PROTO`: Defaults to `https`. Set to `http` if you are running locally.
- `AURA_HOST_CERTBOT_ENABLE`: Enable in production in order to get TLS certificates automatically.
- `AURA_HOST_CERTBOT_EMAIL`: Set a valid email for certbot.

Defining the host for production setups

You can use a public domain name or any locally defined hostname. For example by configuring `aura.local` in `/etc/hosts`. Keep in mind to use a LAN IP though. Using `localhost` or `127.0.0.1` for the production setup is not supported. Here you'd need to make adjustments on your own or use the dev-setup.

Changing the host

Any update of `AURA_HOST_NAME` in your `.env` file is not reflected in the actual configuration. In such cases you either need to manually update the relevant OIDC client database tables, or you simple create new OIDC client IDs in the configuration. After that you can delete any old OIDC clients via the Steering admin interface `$AURA_HOST_NAME/steering/admin/oidc_provider/`.

In case you have deployed AURA Playout to a different instance, also set the value for `AURA_ENGINE_API_INTERNAL_URL`.

Initialize the Steering container:

```
docker compose run --rm steering ./run.sh init-db
```

Start the services with Docker Compose:

```
docker compose up -d
```

We use named volumes for the data-volumes. That way all data will live in your docker-directory.

This deployment will make *AURA Web* reachable in the following way:

- **Dashboard** is reachable directly under the domain given as `$AURA_HOST_NAME`
- **Steering** is reachable under `/steering`
- **Tank** is reachable under `/tank`
- **Track Service** endpoints of `engine-api` will be reachable under `/trackservice` (if `engine-api` is running)
- **Icecast**: If you enabled the optional Icecast server for the reverse proxy it will be reachable under `/icecast`. If you only enabled it, it will be reachable under port `8000` of the machine running *AURA Web*.
- **Dashboard Clock** will be reachable from the configured network under the configured port. If you enabled it for the reverse-proxy, it will be reachable under `/clock`.

Create sample data using fixtures

Fixtures are required for now

Currently you need to install the fixtures in order to make Aura Web working correctly. We will change this in a future release.

In case you want to setup the fixtures with initial sample data, you need to *bash* into the running container via

```
docker compose exec steering bash
```

and then install the program fixtures by using the command

```
poetry run ./manage.py loaddata fixtures/program/*.json
```

2.6.2 Check the logs

Now everything should be up and running. You can check the logs with

```
docker compose logs -f --tail=100 <service-name>
```

See Maintenance and updates for more Information.

2.6.3 NGINX and Reverse Proxy

In the *AURA Web* setup all services are made reachable behind a reverse Proxy. For this NGINX is used. Here we also offer support for using SSL via Let's Encrypt.

Be aware that the docker-compose by default opens ports `80` and `443`. If you want to run *AURA Web* behind a reverse proxy on a different machine, you can just use port 80 of this machine. At the moment running another reverse proxy on the same machine isn't supported with this setup, since there's no possibility to close ports with an override.

If you wish to not use the NGINX whatsoever, you can override the docker-compose with the following content:

```yaml
services:
  nginx:
    deploy:
      replicas: 0
```

This disables the internal NGINX. Keep in mind the nginx does a lot of heavy lifting to handle all redirections and URL rewrites correctly, which you will then need to handle manually.

If you wish to expand the nginx-config you can put further configs into the `custom`-folder. Keep in mind the configs need to end on `.conf`.

2.6.4 Extending the settings for Steering

If you need to extend the settings for `steering`, you can first make the necessary adjustments to the Docker Compose setup in a `docker-compose.override.yml` file.

```yaml
# docker-compose.override.yaml
version: "3.4"

services:

  steering:
    environment:
      DJANGO_SETTINGS_MODULE: steering.production_settings
    volumes:
      - ./production_settings.py:/steering/steering/production_settings.py
```

This will point Django to the new settings module and mount the local `production_settings.py` file into the `steering` container.

Then, you need to create a new `production_settings.py` file with all that you need.

Refer to `steering/sample_settings.py` in the steering repository on how to load and override the settings, and to the Settings Reference for the available Django settings.

Creating Timeslots

Currently the start time of the timeslots created via the dashboard differs from the start time that is actually stored. Therefore you need to create your timeslots one hour in the past. Yes, we are working on this time travel problem steering#116.

2.7 AURA Playout

2.7.1 Deploy AURA Playout

All required files can be found under `config/aura-playout`. For the described commands change to that directory.

Then copy the `sample.env` to `.env` and set all values as needed.

Update the configuration file

Update at least following environment variables in the `.env` file:

- `AURA_TANK_ENGINE_PASSWORD`: The password should match the one configured in AURA Web. Avoid using the default one.
- `AURA_AUDIO_STORE_SOURCE`: The location where Tank is storing audio sources. It defaults to `audio/source`. It should match the one set in AURA Web.
- `AURA_AUDIO_STORE_PLAYLIST`: The location where M3U playlists are provided. This is optional and defaults to `audio/playlist`.
- `AURA_AUDIO_STORE_FALLBACK`: The location where fallback audio is retrieved from. Such audio is played when nothing is scheduled. It defaults to `audio/fallback`.

Now populate the `AURA_AUDIO_STORE_FALLBACK` location with some audio files.

Configure the audio interface

AURA Playout requires you to have a ALSA compatible audio interface.

Disable PulseAudio server

PulseAudio introduces huge delays when controlling the play-out. In order to experience more exact timings in scheduling ensure PulseAudio is disabled.

Now create the file `config/aura-playout/asound.conf` for the ALSA configuration. The configuration should hold custom PCM device named `pcm.aura_engine`.

There is a sample ALSA configuration `sample.asound.conf` in the same directory. For more information about the ALSA configuration visit the official documentation.

Your ALSA Device

To find the name and even more information about your audio device you can use `aplay -L` and aplay -D <_audio_device> –dump-hw-params`. See our FAQ for more information.

Start the services with Docker Compose:

```
docker compose up -d
```

After successful start-up, you should hear some music playing already.

Latency issues with live analog audio

If you are facing latency issues with Engine Core while playing live from the analog audio source, we suggest you try to deploy Engine Core natively. To do so you'll need to comment out the `COMPOSE_PROFILES` variable in you `.env` file. Then, follow the documentation for a native installation of Engine Core. We hope to solve the Docker latency issue in one of the coming releases.

2.7.2 Playout channel routing

Playout channels are routed this way:

`/* FIXME: render Mermaid diagram */`

```
graph TD
    iq0[Queue A] -->|in_queue_0| mix
    iq1[Queue B] -->|in_queue_1| mix
    is0[Stream A] -->|in_stream_0| mix
    is1[Stream B] -->|in_stream_1| mix
    il0[Line In 1-5] -->|in_line_0..4| mix
    ff[Fallback Folder] -->|fallback_folder| which_fallback
    fpls[Fallback Playlist] -->|fallback_playlist| which_fallback
    mix["  Mixer  "] --> silence_detector
    which_fallback{or} -->| | silence_detector{Silence Detector}
    silence_detector -->| | output[Output]
    output --> |output.alsa| C[fa:fa-play Audio Interface]
    output --> |output.icecast| D[fa:fa-play Icecast]
```

2.7.3 Configuring playout

Configure the audio interface

By default only audio output is enabled using the systems default ALSA device.

Ensure PulseAudio server is disabled

You get the most glitch-free experience when using ALSA devices directly. To avoid any play-out malfunctions ensure that no PulseAudio server is running.

To check if PulseAudio is started, run `pactl list`. If this command returns an error, PulseAudio is deactivated.

If you want to choose a different device, edit the configuration file and set a value for `output_device_0`.

It's also possible to set ALSA hardware device IDs like `hw:0,0`. Check the FAQ below on how to retrieve available audio device IDs.

Recommended audio device configuration

For better I/O performance it's recommended to create an ALSA PCM device named `pcm.aura_engine`. You can use the sample configuration `sample.asoundrc` as a basis for creating such device. Consult the ALSA documentation on details. After creating such device verify its properly assigned in the configuration file as `output_device_0="pcm.aura_engine"`.

2.7.4 Configure the audio source locations

Engine Core is requires different audio sources in order to perform the playout.

Configure the location for fallback music

By default fallback audio is retrieved from the `fallback` folder. A local folder for any emergency playback, also called *Station Fallback*.

```
audio_fallback_folder="audio/fallback/"
```

All audio files inside are played in a randomized order, in situations where nothing is scheduled. The folder is being watched for changes. So you can add/remove audio on the fly.

This fallback feature is enabled by default, but can be turned off in via the configuration.

Instead of the fallback folder you can use a playlist in the `playlist` folder for fallback scenarios. Its default file name is `station-fallback-playlist.m3u` and located in:

```
audio_playlist_folder="audio/playlist"
```

Also this playlist is being watched for changes. You'll need to set the configuration option `fallback_type="playlist"` to enable this instead of the fallback folder.

Configure the audio source folder

This is the location for actually scheduled audio files. They are provided by Tank.

```
audio_source_folder="audio/source"
```

If you are running all AURA services on a single instance you should be fine with just creating a symbolic link to the relevant Tank folder (`ln -s $TANK_STORE_PATH $PLAYOUT_AUDIO_SOURCE`). But in some distributed and redundant production scenario you might think about more advanced options on how to sync your audio files between machines.

2.7.5 Features and how they work

Scheduler

Engine provide a scheduling functionality by polling external API endpoints frequently. Those API endpoints are provided by Steering to retrieve schedule information and Tank to retrieve playlist information. To define your schedule you'll also need AURA Dashboard which is an elegent web user interface to manage your shows, playlists and schedules.

Ideally any audio is scheduled some time before the actual, planned playout to avoid timing issues with buffering and preloading. Nonetheless, playlists can also be scheduled after a given calendar timeslot has started already. In such case the playout starts as soon it is preloaded.

If for some reason the playout is corrupted, stopped or too silent to make any sense, then this triggers a fallback using the silence detector (see chapter below).

> Note: If you delete any existing timeslot in Dashboard/Steering this is only reflected in Engine until the start of the scheduling window. The scheduling window is defined by the start of the timeslot minus a configured offset in seconds (compare your Engine configuration).

Versatile playlists

It is possible to schedules playlists with music or pre-recorded shows stored on the **file system**, via external **streams** or live from an **line input** in the studio. All types of sources can be mixed in a single playlist.

The switching between types of audio source is handled automatically, with configured fadings applied.

> Note: Any live sources or streams not specifying a length property, are automatically expanded to the left duration of the timeslot.

Default playlists

While a timeslot can have a specific playlist assigned, it is also possible to define default playlists for schedules and shows:

- **Default Schedule Playlist**: This playlist is defined on the level of some recurrence rules (*Schedule*). In case the timeslot doesn't have any specific playlist assigned, this playlist is broadcasted.
- **Default Show Playlist**: This playlist can be assigned to some show. If neither the specific timeslot playlist nor the default schedule playlist is specified the *default show playlist* is broadcasted.

If none of these playlists have been specified the *Auto DJ* feature of *Engine Core* takes over (optional).

Heartbeat Monitoring

Instead of checking all status properties, the Heartbeat only validates the vital ones required to run the engine. If all of those are valid, a network socket request is sent to a defined server. This heartbeat is sent continuously based on the configured `heartbeat_frequency`. The service receiving this heartbeat ticks can decide what to do with that information. One scenario could be switching to another Engine instance or any other custom failover scenario. Under `engine/contrib/heartbeat-monitor` you'll find some sample application digesting these heartbeat signals.

2.7.6 FAQ

I am using the `default` audio device. How can I set another default device?

You can check the systems default audio hardware by executing `aplay -L` on the command line.

You can set the default device in `/etc/asound.conf` or `~/asoundrc`.

How can I retrieve available ALSA audio devices

- To see only the physically available sound cards: `cat /proc/asound/cards`
- To see sound cards with all additional devices (e.g. HDMI): `aplay -l`
- To see devices configured by ALSA with additional plugins: `aplay -L`
- The default devices that should be used: `aplay -L | grep default`

I have configured an audio device but still hear no sound (native installation)

To test if you device is able to output audio at all, independently from Engine Core, try executing `speaker-test`. Also checkout out the `-D` argument to test specific devices. If you system doesn't provide `speaker-test` you have to install or use your preferred way of testing also audio.

I have configured an audio device but still hear no sound (Docker installation)

If you are running Engine Core using Docker, run the aforementioned `speaker-test` from within your docker container by perform following:

1. Bash into the container using `docker exec -it aura-engine-core bash`
2. Now run `speaker-test`. It that's working, you now know that your audio device is at least available from within Docker and you'll need to further check your Liquidsoap device configuration.
3. Next you can run `liquidsoap tests/test_alsa_default.liq`. This is a basic script which tries to play the supplied MP3 using the default ALSA device.

I'm getting `clock.wallclock_alsa:2` Error when starting output lineout: Failure("Error while setting open_pcm: No such file or directory")!**

Assure you have set the correct device ID. To do so read the paragraph above. Review the audio interface configuration settings and verify if the default settings `input_device_0` and `output_device_0` are valid device IDs.

In case your are *not* running Engine Core within Docker, also check if your executing user (è.g. `engineuser`) belongs to the group `audio`.

How to solve 'Error when starting output output_lineout_0: Failure("Error while setting open_pcm: Device or resource busy")!'?

You probably have set a wrong or occupied device ID. The device could be reserved by another software using the ALSA sound system. Or you might be accessing a device using ALSA which is already assigned to the Pulse Audio sound system. Here it could help to remove the device from PulseAudio before accessing it.

How to avoid stutter, hangs, artifacts or in general glitchy sound?

This can have various reasons, but first of all it's good to check the `engine-core.log` file. Also check your CPU usage. Lastly review the settings of your audio device.

Incorrect ALSA buffer settings: If the ALSA settings provided by your system are not working cleanly the Engine Core settings provide to option to override parameters such as `alsa_buffer`. The correct settings are individual to the used soundcard but in general this is a tricky topic and deeper ALSA knowledge is very helpful.

These problems occur while having Icecast streaming enabled: Try to reduce the quality of the stream, especially when you are experiencing hangs on the stream. Check your Icecast connection. Is it up and running? Maybe there is some authentication issue or an *Icecast limitation for max clients*.

The hardware is hitting its limits: Also check the relevant logs and the system utilization. Are there other processes using up the machines resources? You might even be hitting the performance limit of your hardware. Maybe using a realtime linux kernel could help too.

2.8 AURA Recorder

2.8.1 Deploy AURA Recorder

All required files can be found under `config/aura-recorder`. For the described commands change to that directory. Make sure to follow the preparation steps to add the user: `aura` first.

Then copy the `sample.env` to `.env` and set all values as needed.

```
su aura -c "cp sample.env .env"
```

Update the configuration file

Update at least following environment variables in the `.env` file:

- `AURA_RECORDER_AUDIO_DEVICE`: The audio device from which the audio will be recorded. It is also possible to set a stream.
- `AURA_RECORDER_AUDIO_STORE_HOST`: The location where the recorder saves all recordings.

Make sure that the user `aura` can write to `AURA_RECORDER_AUDIO_STORE_HOST`.

```
source .env
sudo mkdir -p "${AURA_RECORDER_AUDIO_STORE_HOST}/recordings/block"
sudo chown -R aura:aura "${AURA_RECORDER_AUDIO_STORE_HOST}"
```

Using a ALSA audio device

Currently AURA Recorder only supports ALSA devices or audio streams. When using a audio interface, make sure to provide the sample format in the override config. For future releases we plan to read the sample rate directly from the audio device.

Start the services with Docker Compose:

```
docker compose up -d
```

After successful start-up, you should see the `AURA_RECORDER_AUDIO_STORE_HOST` get populated.

2.9 Maintenance and updates

2.9.1 Update Containers

For updates of the components containers:

1. Pull the `aura` repository:

```
git pull
```

and check out the new release:

```
git checkout tags/<release-version>
```

1. Compare your `.env` file with the (now updated) `sample.env` and change your `.env` file accordingly. Take special note of new versions of the components. For `aura-playout`: If you use the docker container of `engine-core` you also want to compare the `sample.engine-core.ini` with your `engine-core.ini`
2. For the components having new versions: Check the release notes for changes you might need to take into account.
3. Pull the new images from Docker Hub:

```
docker compose pull
```

5. Recreate the containers:

```
docker compose up -d
```

2.9.2 Update the database by applying migrations

Manual steps for updating of the databases should only be necessary for `steering`. The other databases should be updated automatically by their services.

The Django migration scripts for `steering` are created during development and only need to be applied by the service to the database. This has to be done after updating the service.

To display any available migration run:

```
docker compose run --rm steering /opt/poetry/bin/poetry run ./manage.py showmigrations
```

To apply the migrations simply run:

```
docker compose run --rm steering poetry run ./manage.py migrate
```

2.9.3 Upgrade the database version

The postgres-version is saved in the `POSTGRES_VERSION` variable. When upgrading Postgres, it is not sufficient to change this version though. New major versions of Postgres cannot read the databases created by older major versions. The data has to be exported from a running instance of the old version and imported by the new version.

Thankfully, there is a Docker container available to automate this process. You can use the following snippet to upgrade your database in the volume `aura-web_steering_db_data`, keeping a backup of the old version in `aura-web_steering_db_data_old`:

```
# Replace "9.4" and "11" with the versions you are migrating between.
export OLD_POSTGRES=9.4
export NEW_POSTGRES=11
doker-compose stop steering-postgres
docker volume create aura-web_steering_db_data_new
docker run --rm \
  -v aura-web_steering_db_data:/var/lib/postgresql/${OLD_POSTGRES}/data \
  -v aura-web_steering_db_data_new:/var/lib/postgresql/${NEW_POSTGRES}/data \
  tianon/postgres-upgrade:${OLD_POSTGRES}-to-${NEW_POSTGRES}
# Add back the access control rule that doesn't survive the upgrade
docker run --rm -it -v aura-web_steering_db_data_new:/data alpine ash -c "echo 'host all␣
↪all all trust' | tee -a /data/pg_hba.conf
# Swap over to the new database
docker volume create aura-web_steering_db_data_old
docker run --rm -it -v aura-web_steering_db_data:/from -v aura-web_steering_db_data_old:/
↪to alpine ash -c "cd /from ; mv . /to"
docker run --rm -it -v aura-web_steering_db_data_new:/from -v aura-web_steering_db_data:/
↪to alpine ash -c "cd /from ; mv . /to"
docker volume rm aura-web_steering_db_data_new
```

Please double check all values and that your local setup matches this. Of course this needs to be done for all postgres-dbs.

2.9.4 Overriding the docker-compose.yml

If you need to make changes to the `docker-compose.yml` you can create a `docker-compose.override.yml` and make the necessary adjustments in there. That way, your adjustments won't create conflicts.

2.9.5 Useful `docker` and `docker compose` commands

Here you can find the official documentation of docker compose commands.

Generally the commands are to be called from the folder in which the respective `docker-compose.yml` is in.

Below you find some useful examples.

List running services

To get a list of services currently running use:

```
docker ps
```

Show logs

To see the current logs of the containers running with `docker compose`:

```
docker compose logs -f --tail=100 <service-name>
```

(Re)Create and start the containers

```
docker compose up -d
```

Starts all containers of services defined in the `docker-compose.yml` file of the folder the command is called from (and creates them if they didn't exist before). If services, that are already running and were changed in any way, Docker Compose re-creates them.

The parameter "`-d`" means it starts them as daemons in the background.

Stop and remove services

If you wish to delete your deployment, all you need to do is shutting it down:

```
docker compose down
```

If you also wish to delete all data, including the Docker Volumes, you can run the following command:

```
docker compose down -v
```

Pull images from Docker Hub

To pull the newest images from Docker Hub:

```
docker compose pull
```

Delete unused images

Since Docker does not automatically delete old images it can happen, that too much space is used by them on the server. To delete images of containers not currently running, use:

```
docker system prune
```

This will not delete the Docker volumes (where the databases and therefore the persistent data lives).

Log into a container

To *bash* into an already running container execute

```
docker compose exec -it steering bash
```

To log into the database of a PostgreSQL container you can run

```
docker compose exec steering-postgres psql -U steering
```

2.9.6 Deploy other Docker Images

If you prefer some individual deployment scenario, you can also run single Docker Images.

These Docker images are hosted on https://hub.docker.com/u/autoradio.

Work in progress

These images are not yet fully documented. We will update the documentation on Docker Hub as we move along. If you need such image please consult the documentation and *Makefiles* in the relevant repositories.

CHAPTER

THREE

DEVELOPER GUIDE

This guide holds general information for AURA architecture and development.

3.1 Architecture

Find details on the AURA Architecture here.

3.1.1 Architecural principals

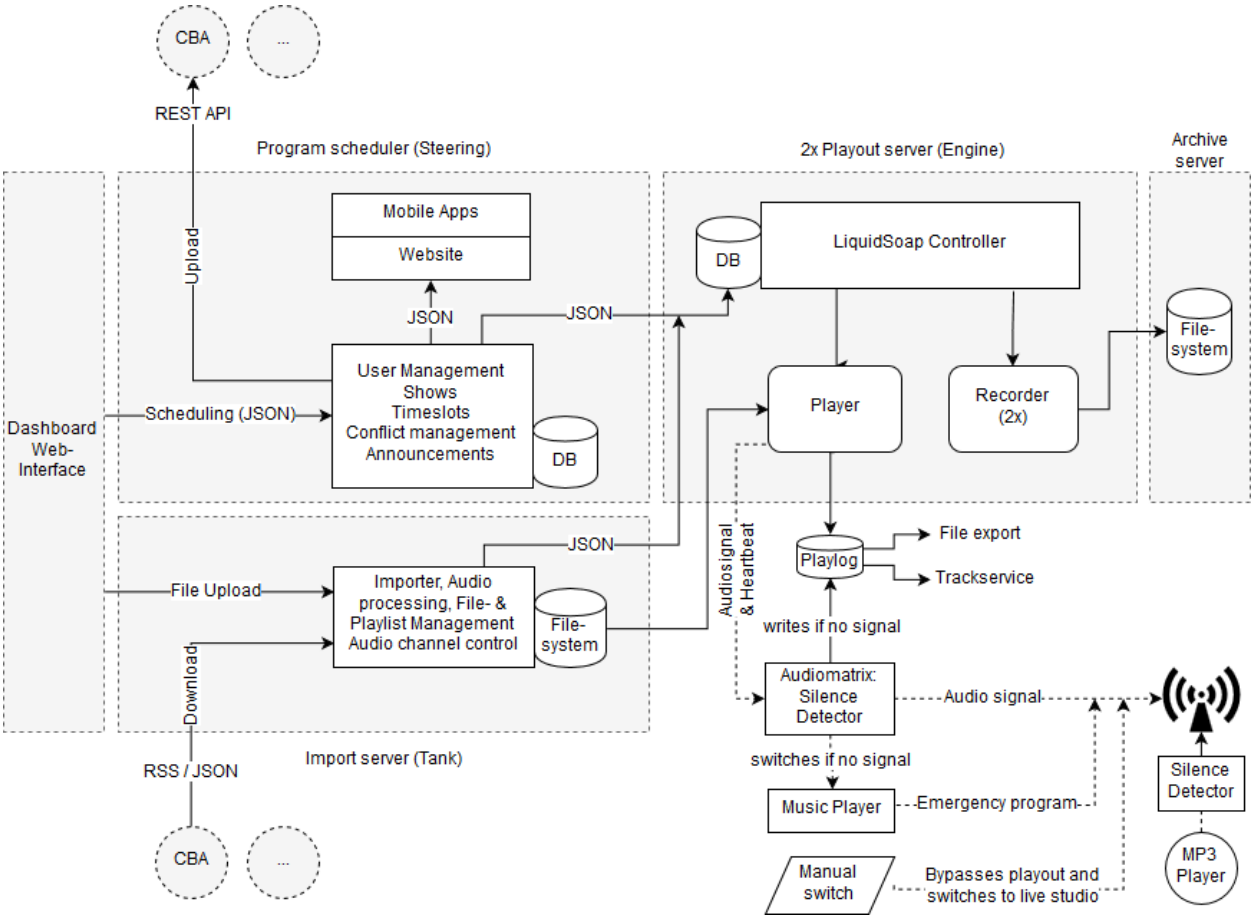
Some of our core organisational and architectural requirements for AURA are:

- **modular architecture**: the whole suite should be made up of modular components which could be exchanged with other custom components
- **transparent API**: every component shall provide a well-documented API through which other components can interact with it, ideally as a REST API.
- **reuse of existing components**: we do not want to reinvent the wheel. Several stations already developed single components as free software and we can adapt and build on those
- **modern frameworks**: we do not code from scratch but use modern application development frameworks which provide maintainability as well as security

---

Outdated diagram

The following component diagram doesn't reflect all the details of the current implementation. It will be updated at some point.



3.1.2 Diagrams

Network Diagram

Check out the provided *Deployment Scenarios* on ideas how the individual projects can be integrated within your infrastructure.

Data Model

Simplified Data Model

This data model is an abstraction of the main entities used by Steering and Tank.

Steering Data Model

This is almost the complete picture of the Steering data model.

3.1.3 Conflict Resolution for the scheduling timetable

Check out the *Conflict Resolution Documentation* page.

3.2 Development

3.2.1 Coding Conventions and Guidelines

Here you find an overview of our conventions on coding and version control.

git

- We use GitHub Flow
- Keep the `main` branch stable, as releases are derived from it
- Avoid crunching commits and rebasing; set `git config pull.rebase false` to use *recursive* as your default merge strategy
- We use *the seven rules of a great Git commit message*, and optionally *conventional commits*
- Mention related ticket IDs where applicable, like `#123` or `play#123` when cross-referencing between repositories
- Use atomic commits

Python

We use Black with default settings, enforced by Flake8.

We use the default settings, except for a maximal line-length of 99 characters. If you are using a Black IDE integration, think about adapting its settings.

For code documentation we use the Flake8 Docstrings extension with Google-style formatting enabled (`docstring-convention=google`).

ECMAScript and TypeScript

We use ESLint as the common Linter. When your code is based on VueJS or Svelte, install the official IDE extensions. These extensions provide additional style checks and auto-formatting options.

API

We utilize an *API-first* approach. APIs are specified using OpenAPI 3. Find the API at api.aura.radio.

All the main aspects are documented within the spec. In some cases you may need some additional documentation in the docs. For example the *API: Schedules and conflict resolution* document can be found in "*Developer Guide -> Misc*".

API-first

At the moment only Engine API is based on API-first. Steering API and Tank API are momentarily generated out of the code base.

Documentation

The general documentation is located in `meta/docs` and hosted at docs.aura.radio. When working on any component, also check if this documentation has to be updated or extended.

3.2.2 Developer Installation

For Development the native installation as outlined in the `README` of individual repositories recommended.

Docker Compose Deployments

For production we highly recommend to run AURA using Docker and Docker Compose as outlined in the *Administration Guide*. But also as a developer you can benefit from the ease of an Docker Compose installation. For example when developing some Engine feature, you may want to run AURA Web with Docker Compose while testing.

Prepare your Development Environment

It is recommended to clone all projects for example in such folder structure:

```
~code/aura/aura
~code/aura/steering
~code/aura/dashboard
~code/aura/engine
~code/aura/engine-api
~code/aura/engine-core
...
```

Order of configuration

After that, you need to configure the projects in following order:

Web Projects

1. Steering - Administration interface for schedules and programme information.
2. Dashboard - Frontend to manage schedules, program info and audio files.
3. Tank - Upload, pre-processing and storage of the audio files.
4. …

Play-out Projects

1. Engine Core - Playout-engine to deliver the actual radio to the audience.
2. Engine API - API Server to provide playlogs and information for the studio clock.
3. Engine - Scheduling and remote control for the playout-engine.
4. …

Configuring the OpenID Clients

Dashboard and Tank authenticate against Steering using OpenID. We use OpenID Connect (OIDC) to implement OpenID.

Check out the OIDC configuration page, on how to get them talk to each other.

3.2.3 Component Documentation

For more detailed documentation read the `README` files in the individual repositories.

You get an overview of all repositories at code.aura.radio.

3.3 Release Management

3.3.1 Semantic Versioning

Release names are defined according to the SemVer 2.0.0 versioning scheme.

Semantic versioning of releases in CI/CD Pipelines

The chosen git tag will be the release name and the version of the Docker Image. Therefore the pipeline script will enforce it to conform to Semantic Versioning 2.0.0.

3.3.2 Keep a Changelog

For changelogs we use the format suggested by Keep a Changelog. Try to follow best practices when writing the changelog. But remember, changelogs are for humans. So do apply best practices when compiling your changelog.

The general AURA changelog is located in `CHANGELOG.md`.

Additionally, all individual service repositories hold their own changelog file. At each release they are bundled in the `aura/changelog` directory.

3.3.3 Current Release

You can find available releases at releases.aura.radio or by executing `git tag`.

Find releases of individual services in the Gitlab repository page under *Deployments* > *Releases*.

3.3.4 Release Workflow

Releasing a service

This release step is **done by the repository maintainers**.

To perform a service release do:

1. Bump the version in the file holding the current version. Commit the version update.
2. Release the version by running `make release`. This command tags and pushes the current branch.

As soon a version tag is pushed, the *CI/CD Pipeline* performs these steps:

- Create a GitLab release with the provided `CHANGELOG.md` as its release notes.
- Pushes the image to Docker Hub.
- Build the *Docker Image* and automatically tags the release with `latest` and `<release version>`.

Releasing the AURA software bundle

The release of a complete software bundle is triggered from within the `aura` repository and **performed by the release manager**.

Before the bundle can be released, the repository maintainers are informed about a code-freeze in a timely manner and asked to releases their individual services (see 'Releasing a service' above).

Preparing the release

1. Create a new branch for the release: `git checkout -b %VERSION%`.
2. Update the versions of referenced services in the *Docker Compose* `config/<docker compose>/sample.env` files and in `.env.version`. Commit the change and publish the branch.
3. Update the configuration files and changelogs by running `make dev.prepare-release`.
4. Bump the version with `poetry version <version>`.
5. Commit and push the branch.

Now inform any testers to checkout and test the given branch.

Performing the release

After the tester(s) have informed the product owner and development team, that the *release criteria* is met, following steps are performed:

1. Update `CHANGELOG.md` with relevant information for the release. Check for relevant information to be merged from the individual changelog files in `aura/changelogs`. The common changelog is also compiled based on entries of the individual service changelogs. These entries are prefixed with the services names.
2. Review and commit the changes.
3. `make dev.release`
4. Checkout `main` and merge everything changed, except the version updates. They should remain as `unstable`. Alternatively cherry-pick the relevant commits.
5. Update `CHANGELOG.md` with a new template section for the next release.
6. Push the changes to `main`.
7. Delete the branch used for release preparation.

3.4 Misc

3.4.1 OpenID Client Configuration

AURA is using OpenID for authentication and authorizing access to restricted API endpoints.

More specifically we are using OpenID Connect (OIDC) for the OpenID handshakes.

Steering is the central OpenID provider. All applications requesting access, need to get an authorization from Steering.

Those applications are called *OIDC clients*.

Required OIDC Clients

In order to properly setup AURA, you'll need to configure OpenID clients for Dashboard and Tank.

The registration and configuration steps below use *the default hosts & ports*.

In case of a *Production Deployment* you'll probably have substitutions like following:

- Steering: *localhost:8080* → *aura-host.org/admin*
- Dashboard: *localhost:8000* → *aura-host.org*
- Tank: *localhost:8040* → *aura-host.org/tank*

Registering clients at Steering

Registering OIDC clients on the command-line

First navigate to your Steering project location.

1. Create an RSA Key

```
$ poetry run ./manage.py creatersakey
```

2. Create OIDC client for Dashboard

```
$ poetry run ./manage.py create_oidc_client dashboard public -r "id_token token" -u␣
↪https://localhost:8080/oidc_callback.html -u https://localhost:8080/oidc_callback_
↪silentRenew.html -p https://localhost:8080/
```

Important: Remember to note the client id and secret for the configuration section below.

1. Create OIDC client for Tank

```
$ poetry run ./manage.py create_oidc_client tank confidential -r "code" -u https://
↪localhost:8040/auth/oidc/callback
```

Important: Remember to note the client id and secret for the configuration section below.

Registering OIDC clients via the admin interface

Follow these three steps to register Dashboard and Tank in the OpenID admin section of Steering.

Create an RSA Key

In the admin interface navigate to *OpenID Connect Provider* and *generate a RSA Key*.

Create OIDC client for Dashboard

Here you'll need to choose following settings:

```
Client Type: Public
Response Type: id_token token (Implicit Flow)
JWT Algorithm: RS256
Require Consent?: No
Reuse Consent?: Yes
```

And enter these redirect URLs:

```
http://localhost:8080/static/oidc_callback.html
http://localhost:8080/static/oidc_callback_silentRenew.html
```

Note, that these URLs have to match exactly the ones you configure in your `.env.development` or `.env.production` files here in the dashboard source. This also means that if you use `localhost` in steering, you must not put `127.0.0.1` or any aquivalent in your dashboard config, but use exactly the same string (and vice versa).

Note the *Client ID* to use in your Dashboard config file.

TODO

Replace image with a current screenshot of Steering

Create OIDC client for Tank

Here you'll need to choose following settings:

```
Client Type: Confidential
Response Type: code (Authorization Code Flow)
JWT Algorithm: RS256
Require Consent?: No
Reuse Consent?: Yes
```

And enter that redirect URL:

```
http://localhost:8040/auth/oidc/callback
```

Note the *Client ID* and secret to use in your Tank config file.

TODO

Replace image with a current screenshot of Steering

Setting the client configuration

When configuring a client, always remind yourself to use the actual hostname. When using the IP address for OIDC redirect URLs you might get unexpected behaviour or being unable to authenticate at all.

Configuring Dashboard

In the Dashboard folder, edit your `.env.production` or `.env.development` respectively, and carefully review if these URLs are matching the ones in the the Steering client settings. These URLs should match your Dashboard host:

```
VUE_APP_API_STEERING_OIDC_REDIRECT_URI = http://localhost:8080/oidc_callback.html
VUE_APP_API_STEERING_OIDC_REDIRECT_URI_SILENT = http://localhost:8080/oidc_callback_
↪silentRenew.html
```

Then set the client id and secret, which you noted from the previous step:

```
VUE_APP_OIDC_CLIENT_ID = %YOUR_ID%
```

Additionally, confirm that your configured Steering URL and port is also matching the instance Steering is running at:

```
VUE_APP_API_STEERING_OIDC_URI = http://localhost:8000/openid
```

Configuring Tank

In the Tank configuration file `tank.yaml` replace `${OIDC_CLIENT_ID}` and `${OIDC_CLIENT_SECRET}` with your client ID and secret, or set the environment variables accordingly.

Also review the given URLS.

```
oidc:
  issuer-url: http://localhost:8000/openid
  client-id: ${OIDC_CLIENT_ID}
  client-secret: ${OIDC_CLIENT_SECRET}
  callback-url: http://localhost:8040/auth/oidc/callback
```

3.4.2 API: Schedules and conflict resolution

This document outlines handling conflict resolution using the API.

Find an overview of the API spec at api.aura.radio. To learn about conflict resolution check out *Timeslot Collision Detection* in the User Guide.

Overview

Creating timeslots is only possible by creating/updating a schedule.

When creating/updating a schedule by giving the schedule data:

1. Projected timeslots are matched against existing timeslots
2. API returns an array of objects containing 2.1. the schedule's data 2.2. projected timeslots (array of objects), including an array of found collisions (objects) and possible solutions (array)
3. To resolve, POST/PUT the `"schedule"` object and the `"solutions"` objects to `/api/v1/shows/{show_pk}/schedules/{schedule_pk}/`

Create/update schedule

- To create: POST `/api/v1/shows/{show_pk}/schedules/`
- To update: PUT `/api/v1/shows/{show_pk}/schedules/{schedule_pk}/`

To start the conflict resolution POST/PUT the schedule object with key `"schedule"` containing:

Variable	Type	Meaning
by_week	int	Weekday number: Mon = 0, Sun = 6
rrule*	int	Recurrence rule: 1 = once, 2 = daily, 3 = business days, 4 = weekly, 5 = biweekly, 6 = every four weeks, 7 = every even calendar week (ISO 8601), 8 = every odd calendar week (ISO 8601), 9 = every 1st week of month, 10 = every 2nd week of month, 11 = every 3rd week of month, 12 = every 4th week of month, 13 = every 5th week of month
first_dat	date	Start date of schedule (e.g. "2017-01-01")
start_tim	time	Start time of schedule (e.g. "16:00:00")
end_tim	time	End time of schedule (e.g. "17:00:00")
last_date	date	End date of schedule (e.g. "2017-12-31")
is_repeti	boole	Whether the schedule is a repetition (default: false)
default_pla	int	A tank ID in case the timeslot's playlist_id is empty: What is aired if a single timeslot has no content? (default: null)
show*	int	Show the schedule belongs to
add_day	int	Add a number of days to the generated dates. This can be useful for repetitions, like "On the following day" (default: null)
add_bus	boole	Whether to add *add_days_no* but skipping the weekends. E.g. if weekday is Friday, the date returned will be the next Monday (default: false)
dryrun	boole	Whether to simulate the database changes. If true, no database changes will occur, instead a summary is returned of what *would* happen if dryrun was false. (default: false)

* = required

** = if `last_date` is not provided, timeslots will be generated until the end of the year if `AUTO_SET_LAST_DATE_TO_END_OF_YEAR` is `True`, otherwise timeslots will be generated until `AUTO_SET_LAST_DATE_TO_DAYS_IN_FUTURE` after the `start_date` **and** `end_date` will remain unset.

Return

After sending the schedule's data, the response will be in the form of:

```
{
    /* Projected timeslots to create. Array may contain multiple objects */
    "projected": [
        {
            "hash": "20180116143000201801161600041",
            "start": "2018-01-16 14:30:00",
            "end": "2018-01-16 16:00:00",
            /* Collisions found to the projected timeslot. Array may contain multiple␣
↪objects */
            "collisions": [
                {
                    "id": 607,
                    "start": "2018-01-16 14:00:00",
                    "end": "2018-01-16 15:00:00",
                    "playlist_id": null,
                    "show": 2,
                    "show_name": "FROzine",
                    "is_repetition": false,
                    "schedule": 42,
```

(continues on next page)

(continued from previous page)

```
                    "memo": "",
                    "note_id": 1 /* A note assigned to the timeslot. May not exist */
                }
            ],
            "error": "An error message if something went wrong. If not existing or empty,
↪ there's no problem",
            /* Possible solutions to solve the conflict */
            "solution_choices": [
                "ours-start",
                "theirs",
                "ours",
                "theirs-start"
            ]
        },
    "solutions": {
        /* Manually chosen solutions by the user (if there's a key it has to have a␣
↪value):
           Key is the hash of the projected timeslot while value must be one of 'solution_
↪choices' */
        "20180116143000201801161600041": ""
    },
    "notes": {
        /* To reassign an existing note to a projected timeslot, give its hash as key␣
↪and the note id as value (may not exist) */
        "20180116143000201801161600041": 1
    },
    "playlists": {
        /* To reassign playlists to a projected timeslot, give its hash as key and the␣
↪playlist id as value (may not exist) */
        "20180116143000201801161600041": 1
    },
    /* The schedule's data is mandatory for each POST/PUT */
    "schedule": {
        "rrule": 4,
        "by_weekday": 1,
        "show": 3,
        "first_date": "2018-01-16",
        "start_time": "14:30:00",
        "end_time": "16:00:00",
        "last_date": "2018-06-28",
        "is_repetition": false,
        "default_playlist_id": null,
        "automation_id": null,
        "dryrun": false
    }
}
```

Solutions

The "solution_choices" array contains possible solutions to the conflict.

To solve conflicts, POST the "schedule" and "solutions" objects to /api/v1/shows/{show_p}/schedules/ or PUT to /api/v1/shows/{show_pk}/schedules/{schedule_pk}/ with "solutions" containing values of solution_choices. Any other value will produce an error.

As long as there's an error, the whole data structure is returned and no database changes will occur. If resolution was successful, database changes take effect and the schedule is returned.

A Schedule is only created/updated if at least one timeslot was created during the resolution process.

Maximum possible output:

```
"solutions": [
    "theirs",
    "ours",
    "theirs-start",
    "ours-start",
    "theirs-end",
    "ours-end",
    "theirs-both",
    "ours-both"
]
```

"theirs" (always possible)

- Discard projected timeslot
- Keep existing timeslot(s)

"ours" (always possible)

- Create projected timeslot
- Delete existing timeslot(s)

"theirs-start"

- Keep existing timeslot
- Create projected timeslot with start time of existing end

"ours-start"

- Create projected timeslot
- Change end of existing timeslot to projected start time

"theirs-end"

- Keep existing timeslot
- Create projected timeslot with end of existing start time

"ours-end"

- Create projected timeslot
- Change start of existing timeslot to projected end time

"theirs-both"

- Keep existing timeslot
- Create two projected timeslots with end of existing start and start of existing end

"ours-both"

- Create projected timeslot
- Split existing into two:
  - Set existing end time to projected start
  - Create another timeslot with start = projected end and end = existing end

Multiple collisions

If there's more than one collision for a projected timeslot, only `"theirs"` and `"ours"` are currently supported as solutions.

Errors

Possible error messages are:

Fatal errors that require the schedule's data to be corrected and the resolution to restart

- `"Until date mustn't be before start"`: Set correct start and until dates.
- `"Start and until dates mustn't be the same"` Set correct start and until dates. (Exception: Single timeslots with recurrence rule 'once' may have the same dates)
- `"Numbers of conflicts and solutions don't match"`: There probably was a change in the schedule by another person in the meantime.
- `"This change on the timeslot is not allowed."` When adding: There was a change in the schedule's data during conflict resolution. When updating: Fields `start`, `end`, `by_weekday` or `rrule` have changed, which is not allowed.

Conflict-related errors which can be resolved for each conflict:

- `"No solution given"`: The solutions value was empty or does not exist. Provide a value of `solution_choices`
- `"Given solution is not accepted for this conflict."`: The solution has a value which is not part of `solution_choices`. Provide a value of `solution_choices` (at least `"ours"` or `"theirs"`)

3.4.3 Default hosts and ports

Here you find the default hosts and ports for all AURA applications.

Development environment

Component	Host:Port	Description
steering	localhost:8080	Django Development Server
dashboard	localhost:8000	VueJS Development Server
dashboard-clock	localhost:5000	Svelte Development Server
tank	localhost:8040	Go Development Server
engine	localhost:1337	Network Socket for Liquidsoap
engine-api	localhost:8008	Werkzeug Development Server
engine-core	localhost:1234	Liquidsoap Telnet Server
play	localhost:5000	Svelte Development Server

CHAPTER

FOUR

BUG REPORTS

Tell us about your problems!

But please use the bug report template below.

4.1 Contact us via Matrix

You can find as on Matrix which is also our primary channel for communication.

4.2 Create a ticket on GitLab

If you don't have an Gitlab account, you can sign up here.

1. Head over to our GitLab instance at code.aura.radio.
2. Search across all projects for existing tickets relevant to your issue.
3. If something is available already, vote for it and place your comment.
4. If nothing is available, first choose the relevant project. If you are unsure about the project, use the `meta` repository. Then file a new ticket.

4.3 Bug Reporting Template

It's helpful if you structure your report similar to that template.

```
# Title

*summary describing your issue*

## Steps to Reproduce

 1. ...
 2. ...
 3. ...

## Expected Result

...
```

(continues on next page)

(continued from previous page)

```
## Actual Result

...

## Logs & configuration

- Contents of your `.env` or other configuration files. Keep in mind to remove any␣
↪sensitive data like password, as the report will be visible publicly.
- Any errors in your browser console. In Firefox you can reach it by pressing `CTRL +␣
↪SHIFT + K`, in Chrome or Chromium via `CTRL + SHIFT + J`.
- In case you are using your own Docker Compose override (`docker-compose.override.yml`),
↪ please share the file contents.
- The output of `docker-compose ps --all`, ensuring all services are started␣
↪successfully.

## Environment

*optional details on your environment*
```

CHAPTER

FIVE

CONTRIBUTING TO AURA

5.1 Code of Conduct

We inherit the *Contributor Covenant*.

5.2 How can I contribute?

You don't need to be a developer to contribute to the project.

- Join the AURA Matrix Space.
- Check out the source code and try AURA for yourself.
- *Create bug reports, feature requests and provide thoughts in GitLab*.
- Become an active developer or maintainer. To do so, check out the *Developer Guide*, especially the *Coding Conventions*.
- Provide sponsorship. We are happy to list you on the front page.

5.3 Contribution Guidelines

to be defined

5.4 Contributors

- Code contributors can be found in the `git` logs.
- Martin Lasinger from Freies Radio Freistadt designed the AURA logos and icons.
- The foundation of Steering is based on Radio Helsinki's PV Module by Ernesto Rico Schmidt.
- The foundation of Engine is based on Comba, by Michael Liebler and Steffen Müller.

5.5 Partners and Sponsorship

Current partners

- Radio Orange 94.0 - Verein Freies Radio Wien
- Radio Helsinki - Verein freies Radio Steiermark
- Radio FRO - Freier Rundfunk Oberösterreich
- Freies Radio Wüste Welle
- Freies Radio Freistadt
- Radio free FM
- FREIRAD Freies Radio Innsbruck

Previous partners and sponsors

- Radiofabrik - Verein Freier Rundfunk Salzburg

5.6 Licensing

By contributing your code you agree to these licenses and confirm that you are the copyright owner of the supplied contribution.

5.7 License

- Logos and trademarks of sponsors and supporters are copyrighted by the respective owners. They should be removed if you fork this repository.
- All source code is licensed under GNU Affero General Public License (AGPL) v3.0.
- All other assets and text are licensed under Creative Commons BY-NC-SA v3.0.

These licenses apply unless stated differently.

CHAPTER

SIX

CONTRIBUTOR COVENANT CODE OF CONDUCT

6.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, caste, color, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

6.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

6.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

6.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

6.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at aura-dev (at) subsquare.at. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

6.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

6.6.1 1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

6.6.2 2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

6.6.3 3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

6.6.4 4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

6.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.1, available at https://www.contributor-covenant.org/version/2/1/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

6.8 License

Contributor Covenant is released under the Creative Commons Attribution 4.0 International Public License.

```
[meta]: Hello, welcome, this is meta!
[you]: Oh, hi meta, what are you meta about?
[meta]: I am the place where all the work behind the 'real' work is happening.
[meta]: I collect stuff about the AuRa project and greet new visitors.
[you]: Ah, kool. Well... I am a new visitor, what can you tell me about this?
[meta]: **loading project overview info**
[meta]: Hello new visitor, here you go:
```

CHAPTER

SEVEN

ABOUT AURA

What is this about? You probably already know. If not, here are some links that explain in more detail what the community radio stations in .AT land and .DE land are which call themselves *Freie Radios* (literally it would be translated to *free radios*, but as the thing with *free* here is the same as with free software in general).

Unfortunately most of those links are in German language as our constituency is located primarily in Austria and German. We will provide a short intro in English language as soon as we find some time.

About *Freie Radios*:

- http://freie-radios.at - Austrian association of community radio stations
- http://freie-radios.de - German association of community radio stations
- https://cba.fro.at - Podcast platform of Austrian community radio stations
- https://freie-radios.net - Audio portal of German community radio stations
- https://amarceurope.eu - European association of community radio stations

And what is this here now?

AuRa is a suite of radio management, programme scheduling and play-out automation software that fits the needs of free radio stations. The initiative took of in Austria, where several stations are still using and depending on *Y.A.R.M.* (which is just yet another radio manager). *Y.A.R.M.* was an awesome project that provided a software which was tailored to how community radio stations create their diverse programmes. Unfortunately it is also a piece of monolithic Java code that has come into the years. Also it never really took of as a free software project and was depending on a single developer. Today nobody really wants to touch its code anymore.

Now we urgently need something new, and all those other solutions out there (FLOSS as well as commercial) do not really fulfill all our requirements. Therefore we decided to pool our resources and develop something new, while reusing a lot of work that has already been done at one or another station.

CHAPTER

EIGHT

MATRIX

You can get in touch and contribute via Matrix and Element. This is our main communication channel.

Join the discussion at our Matrix Space and its main channels:

- AURA Matrix Space
  - AURA-general
  - AURA-dev
  - AURA-support
  - AURA-notifications
  - AURA-offtopic

CHAPTER

NINE

MAILINGLIST

We sporadically share newsletters at the `users (at) aura.radio` Mailinglist.

9.1 Subscribing to the list

In order to subscribe send a mail with following content to `majordomo (at) aura.radio`:

```
subscribe users-aura-radio
```

Then follow the instructions for confirmation in the reply mail. Alternatively you can message the development team directly via `dev (at) aura.radio`, and we will add you to the list.

9.2 Unsubscribing from the list

In order to unsubscribe send this to `majordomo (at) aura.radio`:

```
unsubscribe users-aura-radio
```

CHAPTER

TEN

PARTNERS