AURA - Automated Radio

The AURA Team

Jul 28, 2022

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 Schedule 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.

1.1.1 Single collision, fully overlapping

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

1.1.2 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.

1.1.3 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.

1.2 Dashboard

The dashboard is a simple browser application, allowing hosts to upload and manage their shows.

1.2.1 Login

```
/* tbd */
```

1.2.2 Logout

```
/* tbd */
```

1.2.3 Navigation Bar

```
/* tbd */
```

1.2.4 User Profile

```
/* tbd */
```

1.2.5 User Settings

```
/* tbd */
```

1.2.6 Switch Language

```
/* tbd */
```

1.2.7 Show Management

```
/* tbd */
```

1.2.8 File & Playlist Management

```
/* tbd */
```

1.2.9 Calendar & Schedule Management

```
/* tbd */
```

The calendar is able to detect collisions with existing, conflicting timeslots. To learn more about this behaviour, check out the *Timeslot Collision Detection* page.

1.3 Administration

Programme Coordinators have additional features for radio management at hand.

1.3.1 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.

1.3.2 Categorization options

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

Categories

```
/* tbd */
```

Funding Categories

```
/* tbd */
```

Languages

```
/* tbd */
```

Music Focus

```
/* tbd */
```

Topics

```
/* tbd */
```

Types

```
/* tbd */
```

CHAPTER

TWO

ADMINISTRATION GUIDE

Learn how to make and keep AURA up- and running.

2.1 Overview

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 channels.
- **Tank**: Just spleen without steam. That's where your shows, tunes and recordings are management. 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.
- **Player**: Every car has some fancy *Auto Radio Player*. And every radio station has the need for some modern frontend experience. Player is a library of web components. Making web development a breeze. Just like *playing with blocks*.

2.1.2 Services

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

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

In order to keep administrators life simple, we have defined two component spaces:

- AURA Web
- AURA Playout

These spaces are represented in form of Docker Compose bundles.

AURA Web

Service	Required	Description
steering		Single-source of truth, holding all radio, show, hostand scheduling data
tank		Upload, download and media-management
dashboard		Backend User Interface
dashboard-clock		Studio Clock
player		Library for building frontends based on web components

The *required* services are needed for minimal functionality.

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

The *required* services are needed for minimal functionality.

2.2 Features

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

Find details on additional functionality in the configuration files of individual services.

2.2.1 Steering

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

2.2.2 Dashboard

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

2.2.3 Dashboard Clock

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

2.2.4 Tank

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

2.2.5 Tank Cut & Glue

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

2.2.6 Engine

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

2.2.7 Engine Core

- **Multi-channel input:** Play audio from various sources including queues, streams and audio interface inputs
- **Multi-channel output:** Analog audio or digital stream outputs
- **Icecast connectivity:** Connect to an Icecast Server, provide different encoding formats and bitrates
- **Auto DJ:** Play fallback audio triggered by a silence detector to avoid *Dead Air*. Play randomized music from a folder or M3U playlist.
- **ReplayGain:** Normalization done using passed *ReplayGain* meta data.
- **Metadata handling:** Send information on playlogs via REST calls to external services like Engine API.

2.2.8 Engine API

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

2.2.9 Engine Recorder

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

2.3 Planning the integration into your ecosystem

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 infrastructure integration.

2.3.1 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.

2.3.2 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

2.3.3 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.

2.4 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.

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

The following network diagrams outline ideas on deployment scenarios.

2.4.1 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.

2.4.2 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.

2.4.3 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.

2.5 Docker Compose Installation

2.5.1 Requirements

- Docker Engine 20.10+ (installation documentation)
- Docker Compose 2.2+ (installation documentation)
- Git (installation documentation)

2.5.2 Preparations

Clone the meta repository to the server(s) you want to deploy AURA Web and/or AURA Playout:

```
git clone https://gitlab.servus.at/aura/meta.git
```

and check out the latest release:

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

If you want to deploy an older version, check out the corresponding release tag in the repository.

To see which releases are available you can check in the Gitlab UI Or you can check the tags in the repository you have just cloned:

```
git tag
```

will show you all tags which should correspond to the releases.

Unstable releases

As long as there is no release, just take the latest commit on the `main` branch, which is checked out by default

---

2.5.3 Deploy AURA Web

- All required files can be found under `meta/docker-compose/aura-web`. For all described commands you should be in that directory.
- Copy the `sample.env` to `.env` and set all values as needed
- Initialize the steering-container:

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

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` 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/steering/admin/oidc_provider/`.

- Start the services with docker-compose:

```
docker-compose up -d
```

In case you need/want to install the fixtures for initial programme data you need to *bash* into the already running container via

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

and then install the program fixtures by using the command

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

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.

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`
- **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`.

2.5.4 Deploy AURA Playout

- All required files can be found under `meta/docker-compose/aura-playout`. For all described commands you need to change to this directory.
- Copy the `sample.env` to `.env` and set all values as needed
- In the folder `service-config/engine-core` copy the `sample.engine-core.ini`-file to a new `engine-core.ini` file and set values as needed
- Start the services with docker-compose:

```
docker-compose up -d
```

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 bare-metal installation of Engine Core. We hope to solve the Docker latency issue in one of the coming releases.

2.5.5 Update Containers

For updates of the components containers:

1. Pull the meta 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
```

2.5.6 Update Databases

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 apply the migrations simply run:

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

2.5.7 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.

Show logs

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

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

In AURA Web the services are: steering, dashboard, dashboard-clock and tank (and icecast if you are using it). In AURA Playout the services are: engine, engine-core and engine-api.

(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, were changed in any way docker-compose recreates 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 destroying it with `docker-compose`:

```
docker-compose down
```

If you also wish to delete all data (in 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
```

2.5.8 Deploy other Docker Images

If you prefer some individual deployment scenario, you can also *install single components using Docker only*. These Docker images are hosted on https://hub.docker.com/u/autoradio.

2.6 Docker Installation

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

All docker-containers can be configured with environment variables. Furthermore it's possible to forward ports into the containers and persist data in volumes.

2.6.1 Steering

For the steering container a postgresql-database is required. You either can use a dockerized database or you can host it manually.

For configuring the database you need to set the following environment variables

- `POSTGRES_PASSWORD`: The password for the postgres-database. Should match the password configured in postgres.
- `POSTGRES_USER`: The user to use on the postgres-database.
- `POSTGRES_DB`: The postgres database to use.
- `POSTGRES_HOST`: The host of the postgres-database. Should be reachable from this docker-container.

In general it is also required to provide a secret key:

- `SECRET_KEY`: A secret key for django

You should also set the hostname in the following env-variable:

- `ALLOWED_HOSTS`: A list of allowed hosts. In production this usually should only include the host this container will be reachable from, but it's also a comma-separated list allowing multiple hosts.

If you want to auto-generate the super-user on first start-up, you can set the following environment-variables:

- `DJANGO_SUPERUSER_USERNAME`: The username for the root user
- `DJANGO_SUPERUSER_PASSWORD`: The password for the root user
- `DJANGO_SUPERUSER_EMAIL`: The e-mail for the root user

With all of this configured, you should be able to start the container, which then will configure everything on the initial run.

The steering image exposes two volumes which contain static data for the webserver. To use these, you need to use the following directories as volumes and mount them in the Dashboard container:

- `/steering/steering/static`
- `/steering/steering/site_media`

The image exposes port `8000`, so you need to use that port to connect to steering (either directly or through a reverse proxy).

2.6.2 Tank

For the tank container a PostgreSQL database is required. You either can use a dockerized database or you can host it manually.

To configure the database you can use the following environment variables:

- `TANK_DB_HOST`
- `TANK_DB_NAME`
- `TANK_DB_PASSWORD`
- `TANK_DB_USERNAME`

You also need to configure the OIDC client for tank with the following variables:

- `OIDC_CLIENT_ID`
- `OIDC_CLIENT_SECRET`

To persist Tank's data you need to use the following directory as a volume:

- `/srv/audio`

The image exposes port `8040`, so you need to use that port to connect to tank (either directly or through a reverse proxy).

2.6.3 Dashboard

In production you only need to deploy the Dashboard files on a web-server.

Our Docker image is based on NGINX and contains all files for Dashboard already built. You need to configure some parameters via the environment variables for everything to work.

- `AURA_HOST`: the hostname of aura

Furthermore, you need to configure the OIDC client for Dashboard with the following variable:

- `DASHBOARD_OIDC_CLIENT_ID`

By default the container uses certbot to generate SSL certificates. You should configure the following variables for this:

- `RUN_CERTBOT`
- `AURA_PROTO`
- `CERTBOT_EMAIL`

You can persist the certificates of certbot by mounting the following dir as a volume:

- `/etc/letsencrypt`

You also should mount the static data from steering:

- `/usr/share/nginx/html/steering/static`: for the static-data from the steering container
- `/usr/share/nginx/html/site_media`: for the site-media from the steering container

To make the container accessible from the outside, you should forward ports `80` and `443`.

2.6.4 Dashboard Clock

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

Engine Core

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

Engine API

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

Engine

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

2.7 Setting up the Audio Store

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

Assuming AURA Engine and Tank are hosted on different machines, the `audio_source_folder` must by shared using some network share.

In case you are hosting Engine and Tank on the same machine (e.g. in development), you can skip this documentation. Just think about pointing Tank's audio directory to `engine-core/audio/source` or create a symlink to do so behind the curtains.

By default Engine Core expects audio files shared by Tank in `engine-core/audio/source`.

Now, this folder must be somehow writable by Tank.

2.7.1 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 maintainence 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 no 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 no 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.

2.7.2 Share Type

Then, there's the question how the share is managed. Beside other you have the options to use NFS, SSHFS or even something like Gluster.

For our initial setup we have chosen to use *SSHFS*.

Please share your experience with other share types, and we will include it in further releases of this documentation.

2.7.3 Setting up SSHFS

SSHFS allows you to access the filesystem on a remote computer via SSH. Interaction with files and folders behaves similar to any local data.

This example is setting up the `audio_source_folder` on the Engine instance.

Configuring Engine

First, you'll need to create an user which enables Tank to access the `audio_source_folder` on Engine:

```
adduser tankuser
chown tankuser:engineuser /var/audio/source
```

Ensure that `engineuser` has no permissions to write the directory:

```
chmod u=+rwx,go=+rx-w /var/audio/source
```

Configuring Tank

On the Tank side you need to install `sshfs`:

```
sudo apt-get install sshfs
```

Then create an `audio-store` folder inside the AURA home:

```
:/opt/aura/$ mkdir audio-store
```

Try if you can connect to the engine over SSH using your `tankuser`:

```
ssh tankuser@192.168.0.111 -p22
```

Replace `-p22` with the actual port number your SSH service is running with. For security reasons it's recommended to run SSH not over the default port 22.

Uncomment following setting in `/etc/fuse.conf` to allow the tank-user access the share with write permissions:

```
# Allow non-root users to specify the allow_other or allow_root mount options.
user_allow_other
```

Now create the mount:

```
sudo sshfs -o allow_other -o IdentityFile=~/.ssh/id_rsa tankuser@192.168.0.111:/var/
↪audio/source /opt/aura/audio-store -p22
```

Replace `192.168.0.111` with the actual IP for your Engine and `-p22` with the actual port number your Engine's SSH service is running with.

To make this mount persistent i.e. keep it alive even after a system reboot, you'll need to add a configuration in the `/etc/fstab` file by adding this at the end:

```
# Audio Store @ AURA Engine
sshfs#tankuser@192.168.0.111:/var/audio/source /opt/aura/audio-store fuse auto,port=22,
↪identityfile=~/.ssh/id_rsa,allow_other,_netdev 0 0
```

Again, check for the correct port number in the line above.

To take this into effect you'll need to remount the filesystem with `sudo mount -a` or reboot the machine. When mounting you'll need to authenticate with the `tankuser` password once.

Then review if your Tank's Docker configuration mounts the exact same volume (`/opt/aura/audio-store`). If not edit the tank configuration `/etc/aura/tank.env` and set following property:

```
TANK_STORE_PATH=/opt/aura/audio-store
```

Finally, do some testing if the directory is writable from Tank's system (`touch some-file`) and if the Engine's side can read this file. Then restart your Tank Docker container and you should be good to go.

2.7.4 Read more

- User Guide
- Installation Guide
- Maintenance Guide
- API Specification
- Conflict Resolution

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

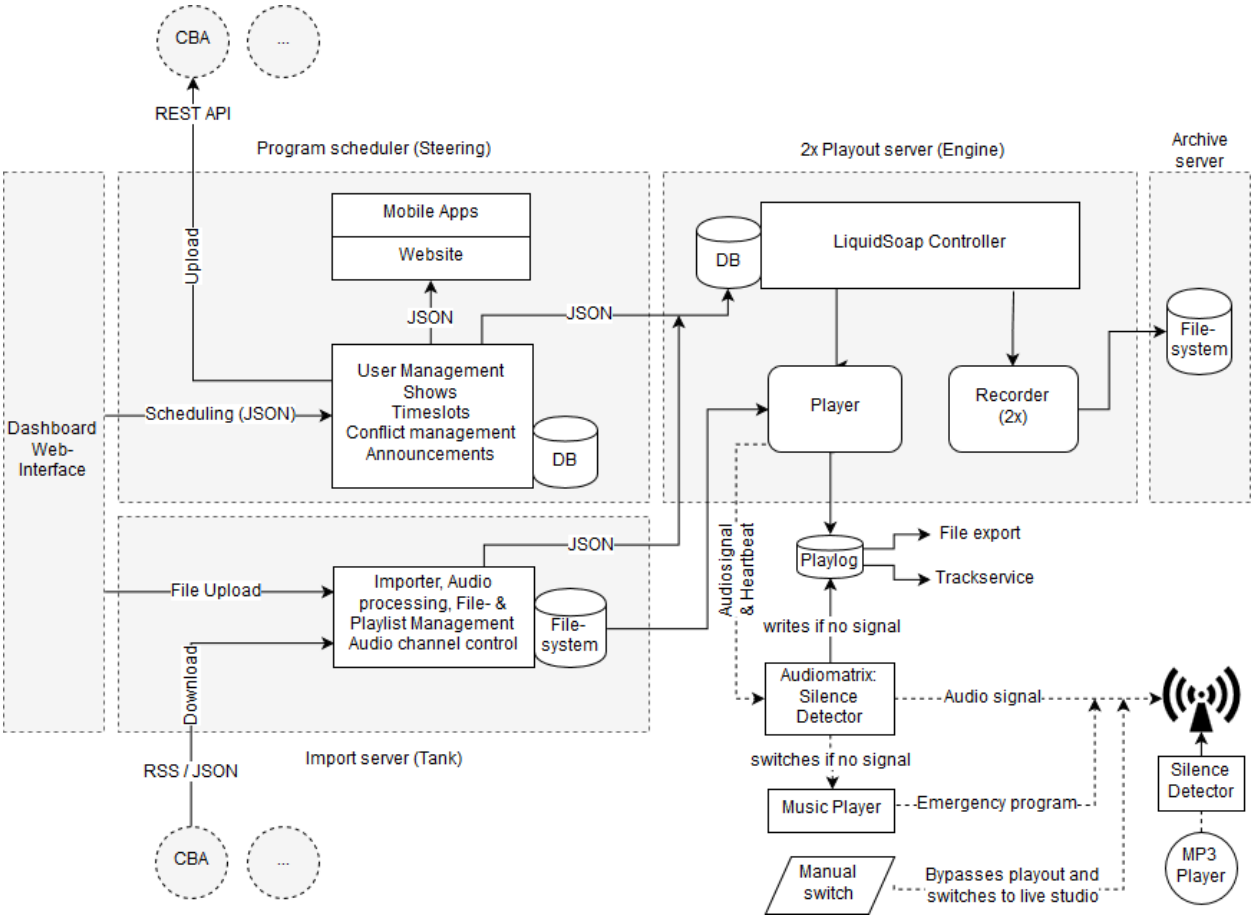
Some of our core organisational and architectural requirements for AURA are:

- **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 Diagramm

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 `player#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 momentarely 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/meta
~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. The gitlab CI/CD pipeline for releases is configured to only accept tags that conform to this versioning scheme.

3.3.2 Current Release

You can check pm.aura.radio to learn about the project status & any available releases in our Wiki.

Find releases of the components (Engine, Tank, Steering, etc) on the respective Gitlab repository page under "*Deployments -> Releases*". For releases of the pre-configured deployments (Docker Compose) you can look at the releases in the meta repository.

3.3.3 General Release Workflow

Releasing a single component

1. Note down relevant changes in the CHANGELOG of the components repository (preferably done continuously during development). Try to follow best practices when compiling the CHANGELOG contents.
2. Decide for a release version (e.g. `1.0.3`). You can check which version is the previous one, by reviewing the repositories "*Deployment -> Releases*" page in the Gitlab UI. The latest release note in the release-notes folder of the repo or the latest git tag represents the most recent version.
3. Update the version in the application specific version file in the repository and commit the change. For example for *Node* apps the app version is located in `package.json`.

All of the following steps should be performed on the latest commit on the `main` branch:

4. Add a general description of the release and a release headline for it to the top of the `CHANGELOG`.
5. Commit the changes to the repository and the release version as a git tag (the existence of the tag on the remote repository triggers the Gitlab CI/CD pipeline).
6. Push the changes and the tag.
7. Wait for the release pipeline to run sucessfully and check the outcome in the "*Deployment -> Releases*" page of the repository.

Releasing a software bundle

The release of a complete software bundle is triggered from within the `meta` repository.

For the meta repository there is the speciality, that you should copy the release notes from the upgraded component releases since the last meta release into the meta release notes.

To document compatible versions of single components, use the `<component>_VERSION` variables found in the `sample.env` settings of the docker-compose setups of aura-playout and aura-web.

3.3.4 When is the CI/CD Release Pipeline triggered and what does it do?

The pipeline has two jobs:

1. It builds and pushes new docker images of the component to the respective AURA Docker Hub Organization.
2. It creates a release in Gitlab using the content of the `CHANGELOG` as release notes.

The first job is triggered when a commit is made to the `main` branch or when a tag is pushed to the Gitlab repository of the component.

When there is a commit on the `main` branch, the built Docker image will be tagged `unstable`. Unstable versions can be used in development/testing environments but should not be used in production environments.

When a tag is pushed (technically on any branch, but it should probably be done only on commits on the `main` branch), the Docker image built will be tagged both `latest` and `<release version>`.

When a tag is pushed, and the build/push job is successful, it will then trigger the release job.

Semantic versioning of releases

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.

3.4 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

```
steering$ python manage.py creatersakey
```

2. Create OIDC client for Dashboard

```
steering$ python 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

```
steering$ python 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_weekday*	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_date*	date	Start date of schedule (e.g. "2017-01-01")
start_time*	time	Start time of schedule (e.g. "16:00:00")
end_time*	time	End time of schedule (e.g. "17:00:00")
last_date**	date	End date of schedule (e.g. "2017-12-31")
is_repetition	boolean	Whether the schedule is a repetition (default: false)
default_playlist_id	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_days_no	int	Add a number of days to the generated dates. This can be useful for repetitions, like "On the following day" (default: null)
add_business_days_only	boolean	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	boolean	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,
                    "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
```

(continues on next page)

(continued from previous page)

```
    },
    /* 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
player	localhost:5000	Svelte Development Server

3.4.4 AURA Command Line Interface (CLI)

Each project as it's own `run.sh` script for developer-centric operations.

Alpha Status

Not yet all projects have CLI support implemented consistently. See meta#57 for the current state. In the future the `run.sh` scripts should be replaced by a proper CLI Framework allowing simple maintenance of the overall project.

Default action

If you start the script with `./run.sh` it performs the default action, which is usually `dev`. Since this is most often used by developers, such default action should speed up start-up time for development.

In case of e.g. `steering` it's starting the Django development server. In case of `engine-core` it's starting the playout server.

Local Commands

Start Development Server

```
./run.sh dev
```

Start Production Server

```
./run.sh prod
```

Run the test-suite

```
./run.sh test
```

Docker Commands

Every command prefxed with `docker:` issues an Docker operation.

Docker Build

Create a docker build.

```
./run.sh docker:build
```

Docker Push

Pushes the build image to https://hub.docker.com/u/autoradio.

```
./run.sh docker:push
```

Docker Run

Development

Start the project within a docker container using any existing, local build.

Also note:

- The container is running in the foreground, allowing you to see the logs (Opposite of *detached mode*, where the Docker parameter `-d` is passed)
- The container is removed when exiting (Docker parameter `--rm`)

```
./run.sh docker:dev
```

Debugging

Some projects like Engine Core provide a `debug` target to debug container development.

In contrast to the development mode, only the log file is `tail`ed, after starting the container.

```
./run.sh docker:debug
```

After the container is up- and running, you can open a shell inside the container.

Taking Engine Core as an example, you do that by executing:

```
docker exec -it aura-engine-core bash
```

Production

To run some project in production, use the instructions for Docker or Docker Compose in the *META CLI for SysOps* above.

Docker Test

Runs the test suite.

```
./run.sh docker:test
```

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 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

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

PARTNERS