Developer Documentation

Open Audio Search

Open Audio Search is a search engine for audio. It uses automated speech recognition to turn spoken word into text, which is then indexed in a full-text search engine. The engine can subscribe to RSS feeds to ingest content from different sources. Through a web user interface, users can search in the generated transcripts of radio shows and podcasts and play them right from the time mark of a search result.

About

Open Audio Search is an open source project by arso collective and cba - cultural broadcasting archive.

Funding

The development of Open Audio Search is supported by:

Prototype Fund

netidee PROJEKTE

Links

GitHub Repository

https://github.com/openaudiosearch/openaudiosearch

Project Website

https://openaudiosearch.org

Demo site

https://demo.openaudiosearch.org

Content

Architecture

The following paragraphs will outline the technical architecture of Open Audio Search (*OAS*). This is directed towards developers and system administrators. We intend to expand and improve this document over time. If something in here is unclear to you or you miss explanations, please feel invited to open an issue.

Core (or backend)

This is a server daemon written in Rust. It provides a REST-style HTTP API and talks to our main data services: CouchDB and Elasticsearch or OpenSearch.

The core compiles to a static binary that includes various sub commands, the most important being the run command which runs all parts of the core concurrently. The other commands currently mostly serve debug and administration purposes.

The core oftenly uses the changes endpoint in CouchDB. This endpoint returns a live-streaming list of all changes to the CouchDB. Internally, CouchDB maintains a log of all changes made to the database, and each revision is assigned a sequence string. Various services in OAS make use of this feature to visit all changes made to the database.

The core uses the asynchronous Tokio runtime to run various tasks in parallel. Currently, this includes:

- A **HTTP server** to provide a REST-style API that allows to GET, POST, PUT and PATCH the various data records in OAS (feeds, posts, medias, transcripts, ...). It also manages authentication for the routes. It can serve the web frontend either by statically including the HTML, JavaScript and other assets directly in the binary, or by proxying to another HTTP server (useful for development). The HTTP server uses Rocket, an async HTTP framework for Rust.
- An **indexer** service that listens on the CouchDB changes stream and indexes all posts, medias and transcripts into an Elasticsearch index. For the index, our data model is partially flattened to make querying more straightforward.
- The **RSS importer** also listens on the changes stream for *Feed* records and then periodically fetches these RSS feeds and saves new items into *Post* and *Media*

records. It also sets a flag on the *Media* records depending on the settings that are part of the *Feed* record whether a transcribe job is wanted or not.
- A **job queue** also listens on the changes stream and may, depending on a *TaskState* flag, create jobs for the worker. The job services currently uses the Celery job queue with a Redis backend.

The core still is rough at several edges. While it works, the internal APIs will still change quite significantly towards better abstractions that makes these data pipelines more flexible and reliable. We need better error handling in cases of failures and better observability. There is *a lot* of room for optimizations. For example, at this point each service consumes a separate changes stream, and there is no internal caching of data records. This also means that any performance issues that might be visible at the moment will have a clear path to being solved.

Worker

The worker is written in Python. It currently uses the Celery job queue to retrieve jobs that are created in the core. It performs the jobs and then posts back its results to the core over the HTTP API exposed by the core. Usually, it will send a set of JSON patches to update one or more records in the database with its results.

Currently, the two main tasks are:

- transcribe: This task takes an audio file, downloads and converts it into a WAV file and then uses the Vosk toolkit to create a text transcription of the audio file. Vosk is based on Kaldi ASR, an open-source speech-to-text engine. To create these transcripts, a model for the language of the audio is needed. At the moment, the only model that is automatically used in OAS is the German language model from the Vosk model repository. We will soon provide more models, and will then also need to implement a mechanism to first detect the spoken language to then use the correct model.
- nlp: This task takes the transcript, description and other metadata of a post as input, and then performs various NLP (natural language processing) steps on this text. Most importantly, it tries to extract keywords through an NER (named entity recognition) pipeline. Currently, we are using the SpaCy toolkit for this task.

We plan to add further processing tasks, e.g. to detect the language of speech, restore punctuation in the transcript, chunk the transcript into fitting snippets for subtitles.

Frontend

The frontend is a single-page web application written with React. It uses the Chakra UI toolkit for various components and UI elements. The frontend talks to the core through its HTTP API. It is mostly public-facing with a dynamic search page that allows filtering and faceting the search results. We currently use ReactiveSearch components for the search page. It also features a login form for administrators, which unlocks administrative sections. Currently, this only includes a page to manage RSS feeds and some debug sections. We will add more administrative features in the future.

Packaging

OAS includes Dockerfiles for the core and the worker to easily package and run OAS as Linux containers. It also includes docker-compose files to easily start and run OAS together with all required services: CouchDB, Elasticsearch and Redis.

The docker images can be built from source with the provided Dockerfiles. We also push nightly images to Dockerhub, which allows to run OAS without building from source.

Development setup

Follow the instructions for the development setup in the README.

Tips and tricks

Development mode

The server can be reloaded automatically when application code changes. You can enable it by setting the `oas_dev` env config, or starting the server with `OAS_DEV=1` server.py.

Frontend development

Requirements

You need Node.js and npm or yarn. yarn is recommended because it's much faster.

On Debian based systems use the following to install both Node.js and yarn:

```
curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee
/etc/apt/sources.list.d/yarn.list
sudo apt update
sudo apt install yarn nodejs
```

Development

For development `webpack-dev-server` is included. In this folder, run yarn to install all dependencies and then yarn start to start the live-reloading development server. Then open the UI at `http://localhost:4000`. In development mode, the UI expects a running oas_worker server at `http://localhost:8080`.

Deployment

Make sure to run yarn build in this directory after pulling in changes. The oas_worker server serves the UI at /ui from the dist/ folder in this directory.

Inspect the Redis databaes

`redis-commander` is a useful tool to inspect the Redis database.

```
# install redis-commander
yarn global add redis-commander
# or: npm install -g redis-commander

# start redis-commander
redis-commander
```

Now, open your browser at http://localhost:8081/

ASR Evaluation

Start worker:

```
cd oas_worker
python worker.py
```

Run transcription using ASR engine in another Terminal:

```
cd oas_worker
# download models if needed
python task-run.py download_models
# transcribe a single file
python task-run.py asr --engine ENGINE [--language LANGUAGE] --file_path
FILE_PATH [--help]
# (e.g).
python task-run.py asr --engine vosk --file_path ../examples/frn-leipzig.wav
```

NLP Evaluation

Generate Devset

Generate and serve Devset RSS feed on localhost port 6650:

```
cd oas_worker/
poetry run python devset/generate_devset.py
sh devset/serve_nlp_devset.sh
```

Import RSS feed:

In UI, login first. Then head over to Importer-Tab. There fill in the URL `http://127.0.0.1:6650/rss.xml` into the Add new feed-Section and make sure to check the Transcribe items-Button.

Access NLP Results

In Search-UI, click on a post you want to inspect. From its URL, copy the Post-ID and paste it as argument to the examples nlp.py script:

```
cd oas_worker/examples
poetry run python nlp.py <OAS-POST-ID>
```

Notes

This section contains various notes, tips and tricks and other discoveries that are somehow related to Open Audio Search development.

Notes on Elastic

We want to encode the ASR metadata for each word (start, end, conf) into the elastic index with the `delimited_payload` filter.

Create an index with a delimited payload filter:

```
{
  "mappings": {
    "properties": {
      "text": {
        "type": "text",
        "term_vector": "with_positions_payloads",
        "analyzer": "whitespace_plus_delimited"
      }
    }
  },
  "settings": {
    "analysis": {
      "analyzer": {
        "whitespace_plus_delimited": {
          "tokenizer": "whitespace",
          "filter": [ "plus_delimited" ]
        }
      },
      "filter": {
        "plus_delimited": {
          "type": "delimited_payload",
          "delimiter": "|",
          "encoding": "float"
        }
      }
    }
  }
}
```

top level "transcript" field,

token|mediaNum,start,end,conf

ffmpeg

some useful ffmpeg commands

Cut mp3 to first 30 seconds:

```
ffmpeg -t 30 -i inputfile.mp3 -acodec copy outputfile.mp3
```