D3.2: HRADIO Communication Platform – Third iteration

Editor: Markus Friedrich, Stefan Langer (LMU)

This deliverable includes the software documentation of the HRADIO Communication Platform. This document is the third and final iteration of this documentation.
Basic Information

| Work package | 3  |
| Due date     | 30/09/2019 |
| Submission date | 05/11/2019 |
| Deliverable lead | LMU |
| Version      | 3.0 |
| Authors      | Markus Friedrich, Stefan Langer (LMU) |
| Reviewers    | François Daoust, Iris Jennes |

Document Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description of change</th>
<th>List of contributor(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0.1</td>
<td>07/06/2018</td>
<td>Initial version</td>
<td>Markus Friedrich</td>
</tr>
<tr>
<td>V0.2</td>
<td>15/06/2018</td>
<td>Version for review</td>
<td>Markus Friedrich, André Ebert</td>
</tr>
<tr>
<td>V0.3</td>
<td>26/06/2018</td>
<td>Version for coordinator review</td>
<td>Markus Friedrich</td>
</tr>
<tr>
<td>V1.0</td>
<td>27/06/2018</td>
<td>Final version</td>
<td>Simon Delaere</td>
</tr>
<tr>
<td>V1.1</td>
<td>22/01/2019</td>
<td>Second iteration ready for internal review</td>
<td>Markus Friedrich</td>
</tr>
<tr>
<td>V1.2</td>
<td>30/01/2019</td>
<td>Version for coordinator review (2nd iteration)</td>
<td>Markus Friedrich, François Daoust</td>
</tr>
<tr>
<td>V2.0</td>
<td>31/01/2019</td>
<td>Final version of second iteration after coordinator review</td>
<td>Simon Delaere</td>
</tr>
<tr>
<td>V2.1</td>
<td>21/10/2019</td>
<td>Version for review (3rd iteration)</td>
<td>Markus Friedrich, Stefan Langer</td>
</tr>
<tr>
<td>V2.2</td>
<td>30/10/2019</td>
<td>Version for coordinator review (3rd iteration)</td>
<td>Markus Friedrich, Iris Jennes, François Daoust</td>
</tr>
<tr>
<td>V3.0</td>
<td>05/11/2019</td>
<td>Final version after coordinator review</td>
<td>Simon Delaere</td>
</tr>
</tbody>
</table>
Disclaimer

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 731677. This document reflects only the authors’ views and the Commission is not responsible for any use that may be made of the information it contains.

<table>
<thead>
<tr>
<th>Nature of the deliverable:</th>
<th>R + DEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dissemination Level</td>
<td></td>
</tr>
<tr>
<td>PU</td>
<td>Public, fully open, e.g. web</td>
</tr>
<tr>
<td>CL</td>
<td>Classified, information as referred to in Commission Decision 2001/844/EC</td>
</tr>
<tr>
<td>CO</td>
<td>Confidential to HRadio project and Commission Services</td>
</tr>
</tbody>
</table>
This document describes the architecture and implementation of the third iteration of the HRADIO Communication Platform. Main focus of this iteration is the implementation of the federated search functionality, the integration of the privacy-preserving user data collection module as well as transport layer security mechanisms (TLS) for all publicly available REST services. In addition, the component for radio station recommendations was extended with a flexible architecture allowing for the integration of multiple recommendation systems forming a recommender ensemble.

In the first iteration, the provided architecture was built from the ground up by using industry proven technologies. It allowed complex metadata queries as well as simple integration of external metadata sources. The second available version of the architecture encompassed improvements and extensions regarding the integration of new sources of metadata (service and programme metadata) and made their traceability simpler and more powerful. The internal bus system introduced in the first iteration was also used by a centrally maintained crawler network, which provided a new level of separation and flexibility. The current iteration adds completely new components (federated search, privacy-preserving user data collection, transport layer security) and improves existing ones (radio station recommendation system, deployment).

The developed data model for radio-related metadata which is based on the RadioDNS standard was further extended to fulfil the needs of the project partners within the first and the second pilot phase (e.g. genre definitions and programme event structure). In addition, future changes of the data model are made less complex due to the increased flexibility.

External service interfaces for metadata maintenance and search functionalities were also further improved (e.g. federated search, station recommendations). In addition, a new service interface for user statistics, privacy-preserving user data, cross-device listening and password management was introduced. In order to visualize platform data, a visualization and dashboard component was added.

Finally, the HRADIO Communication Platform deployment mechanism was extended and simplified in order to be able to cope with the integration of the newly developed components, while necessary documentation was updated accordingly.
# TABLE OF CONTENTS

1. **INTRODUCTION** ........................................................................................................... 11

2. **THE HRADIO COMMUNICATION PLATFORM** ......................................................... 13

   2.1. Requirements ............................................................................................................... 13

   2.2. Architecture ................................................................................................................ 17

      2.2.1. Metadata Search & Storage .................................................................................... 19

      2.2.2. Music Metadata Storage ....................................................................................... 23

      2.2.3. Metadata Importer ................................................................................................. 23

      2.2.4. Recommendations ................................................................................................. 24

      2.2.5. Privacy-Preserving User Data Collection ............................................................... 29

      2.2.6. Dashboard ............................................................................................................. 32

      2.2.7. Password Manager ............................................................................................... 32

   2.3. Technologies ................................................................................................................ 33

      2.3.1. RabbitMQ ............................................................................................................... 33

      2.3.2. Elasticsearch ......................................................................................................... 35

      2.3.3. Kibana .................................................................................................................. 37

      2.3.4. PostgreSQL .......................................................................................................... 38

   2.4. Data Model .................................................................................................................. 38

   2.5. Modules ........................................................................................................................ 41

      2.5.1. Metadata Search & Storage .................................................................................... 41

      2.5.2. Music Metadata Storage ....................................................................................... 72

      2.5.3. Metadata Importer ................................................................................................. 73

      2.5.4. Recommendations ................................................................................................. 81

      2.5.5. Privacy Preserving User Data Collection ............................................................... 87

      2.5.6. Dashboard ............................................................................................................. 116

      2.5.7. Password Manager ............................................................................................... 116

   2.6. System Requirements ................................................................................................. 118

   2.7. Deployment .................................................................................................................. 118

3. **HOW TOS** ....................................................................................................................... 120

   3.1. How To: Test the Platform .......................................................................................... 120
LIST OF FIGURES

FIGURE 1: THE ARCHITECTURE OF THE HRADIO COMMUNICATION PLATFORM. ........................................... 19
FIGURE 2: THE BASIC IDEA OF THE FEDERATED SEARCH. .................................................................................. 22
FIGURE 3: THE MAIN DATA FLOW BETWEEN CRAWLER AND IMPORTER MODULES. ..................................... 24
FIGURE 4: THE RECOMMENDER RESULT AS A WEIGHTED COMBINATION OF THE RESULTS OF A SET OF RECOMMENDERS. .................................................................................................................. 26
FIGURE 5: PIPELINE OF THE AUDIO FEATURE EXTRACTION PROCESS. THE RESULTS ARE 11-DIMENSIONAL FINGERPRINTS FOR EACH SERVICE FOR WHICH DISTANCES CAN BE CALCULATED TO CREATE RECOMMENDATIONS. ...................................................................................... 27
FIGURE 6: SILHOUETTE COEFFICIENT APPLIED ON EM CLUSTERING. IT SHOWS A PEAK FOR A NUMBER OF 11 CLUSTERS. ........................................................................................................................................ 28
FIGURE 7: DATA IS DISTORTED LOCALLY (MICRODATA BEFORE, AFTER). AFTERWARDS, THE BIG PICTURE (ALL DATA AFTERWARDS) IS STILL VISIBLE ........................................................................................... 30
FIGURE 8: HIGH-LEVEL OVERVIEW OF THE PROTOTYPICAL PRIVACY-PRESERVING USAGE DATA COLLECTION SYSTEM BASED ON SURVEYS AND CLIENT-SIDE DATA DISTORTION. ................................................. 31
FIGURE 9: THE CLIENT AUTHORIZATION MECHANISM. ..................................................................................... 32
FIGURE 10: COMMUNICATION SETTING WITH A PUBLISHER P, AN EXCHANGE ENTITY X AND TWO CONNECTED QUEUES. SOURCE: ............................................................................................................. 35
FIGURE 11: COMMUNICATION SETTING WITH A PUBLISHER, A CONNECTED QUEUE AND TWO CONSUMERS. SOURCE: ........................................................................................................................................ 35
FIGURE 12: TYPICAL ELASTICSEARCH ARCHITECTURE WITH DATA SOURCES, PROCESSING, STORAGE AND SEARCH. SOURCE: ........................................................................................................... 36
FIGURE 13: SCREENSHOT OF AN EXEMPLARY KIBANA DASHBOARD. ............................................................... 38
FIGURE 14: THE DESIGNED METADATA DATA MODEL. ...................................................................................... 39
FIGURE 15: WELCOME PAGE OF THE SERVER’S UI. ......................................................................................... 87
FIGURE 16: THE UI FOR THE CREATION OF A NEW SURVEY. ........................................................................ 88
FIGURE 17: THE UI FOR CHANGING A SURVEY’S STATE. .................................................................................. 89
FIGURE 18: RESULT VISUALIZATION VIA BAR CHARTS. ................................................................................... 89
FIGURE 19: COMMUNICATION PLATFORM CONTAINERIZATION. PLEASE NOTE THAT THE SURVEY REST SERVICE IS NOT PART OF THE DIAGRAM SINCE IT IS DEPLOYED SEPARATELY. ................................................................................ 118
FIGURE 20: A VIRTUAL MACHINE RUNS A FULL OPERATING SYSTEM AS GUEST ON A HOST OPERATING SYSTEM. DOCKER CONTAINERS RUN DIRECTLY ON THE SO-CALLED DOCKER ENGINE AND SHARE THE HOST OPERATING SYSTEM. SOURCE:..........................119
LIST OF TABLES

TABLE 1: LIST OF DOCUMENT CHANGES. ........................................................................................................ 12
TABLE 2: LIST OF REQUIREMENTS. ............................................................................................................... 13
<table>
<thead>
<tr>
<th>ABBREVIATIONS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>DAB</td>
<td>Digital Audio Broadcasting</td>
</tr>
<tr>
<td>DNS</td>
<td>Dynamic Name Server</td>
</tr>
<tr>
<td>DRM</td>
<td>Digital Radio Mondiale</td>
</tr>
<tr>
<td>EPG</td>
<td>Electronic Programme Guide</td>
</tr>
<tr>
<td>FM</td>
<td>Frequency modulation</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>PI</td>
<td>Programme Information</td>
</tr>
<tr>
<td>SI</td>
<td>Service Information</td>
</tr>
<tr>
<td>SPI</td>
<td>Service and Programme Information</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>STOMP</td>
<td>Simple (or Streaming) Text Oriented Messaging Protocol</td>
</tr>
<tr>
<td>ICY</td>
<td>“I Can Yell” (Metadata tag format for the SHOUTcast Protocol)</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>TVA</td>
<td>TV-Anytime standard</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

The HRADIO project aims to provide sustainable technologies to exploit the potential of hybrid radio services that combine the power of broadcasting radio with modern internet-based services. An important puzzle piece of hybrid radio enabled technology is the storage, maintenance and indexing of radio related metadata. It powers recommender systems, station search engines and other services which focus on the personalization of the radio experience. The HRADIO Communication Platform will help radio stations to manage metadata, to be visible to potential new listeners, to provide meaningful recommendations and to collect user data in a privacy-preserving manner.

This document describes the third iteration of the HRADIO Communication Platform. The focus of this iteration is the new federated search module, the improved station recommender system architecture, the new privacy-preserving user data collection module and the security features for public service interfaces. It now includes all features as described in the project proposal.

The first chapter describes the requirements that should be met by the system. It is followed by an overview of the system's architecture, a description of the metadata model and an in-detail view of the different components/modules, interfaces, and services, which are used and provided by the platform. The second chapter contains a set of How-Tos that help getting started with the metadata platform. The third and last chapter gives an outlook on future developments.

Compared to the second iteration, the following changes have been made:
Table 1: List of document changes.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Architecture</td>
<td>The architecture diagrams and component descriptions were added and/or updated to the latest state of development.</td>
</tr>
<tr>
<td>2 Data Model</td>
<td>The data model was updated to the latest version.</td>
</tr>
<tr>
<td>2 Modules</td>
<td>The module descriptions were changed to match the current development state. New module descriptions were added. System requirements and deployment overview were changed as well.</td>
</tr>
<tr>
<td>3 How TOs</td>
<td>How To guides were updated.</td>
</tr>
<tr>
<td>4 Future Developments</td>
<td>New aspects of future developments were added.</td>
</tr>
</tbody>
</table>
2. THE HRADIO COMMUNICATION PLATFORM

This chapter introduces and describes the main aspects of the HRADIO Communication Platform.

2.1. REQUIREMENTS

Requirements were collected and analysed from the project’s proposal document, from technical project partners, from user scenarios, that have been developed in the second work package and in the pilot 1 planning phase. Requirements that were added or changed in the second and third iterations are marked in the third column.

Table 2: List of requirements.
<table>
<thead>
<tr>
<th>Aspect</th>
<th>Description</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>The system should provide a publicly available interface for radio related metadata search.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The system should be able to process complex queries (range queries, queries needed for text mining, spatial nearest neighbour queries).</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The system’s search engine should scale with the number of search requests.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The system’s search engine should be able to use other search engine instances for searches and build a network of search nodes (federation).</td>
<td>3</td>
</tr>
<tr>
<td>Storage</td>
<td>The system should store metadata persistently in a human readable format.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The system’s storage engine should be able to scale with the amount of stored metadata.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The system’s storage engine should be exchangeable without major refactorings.</td>
<td>1</td>
</tr>
<tr>
<td>Data Model</td>
<td>The system’s data model should cover the use cases defined in the second work package (information about stations, station services, schedules and programmes).</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The system’s data model should have a schema (needed for type-based queries, like date/time range queries).</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The system’s data model should be extensible.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The system’s data model should be based on existing standards (specifically on RadioDNS).</td>
<td>1</td>
</tr>
<tr>
<td>Import</td>
<td>The system should provide interfaces to maintain and retrieve metadata.</td>
<td>1</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>The system should provide interfaces for metadata importer extension modules.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The system should be able to import RadioDNS metadata from a configurable set of sources.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The system should be able to import JSON-formatted Dynamic Label+ data provided by a STOMP server to enrich programme event information.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The system should be able to import ICY Tag-based metadata (as transferred via radio IP streams) to enrich programme event information.</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>The system should be able to import DISCOGS song metadata to enrich programme event information.</td>
<td>2</td>
</tr>
</tbody>
</table>

The system’s data model should be able to express genre information from different data sources (e.g. Discogs, RadioDNS).

In general, the data model must be able to represent the specified input data formats and sources, described within the overall HRADIO system architecture document, D2.3, Derived Technical Requirements (see DAB-DL+, DAB-SPI, DAB-SLS, RadioTAG, and RadioWEB).
Corresponding to the requirements onto the data model, the specified importers must be able to process the specified input data and feed it into the architectures metadata base (see overall HRADIO system architecture document, D2.3, Derived Technical Requirements: DAB-DL+, DAB-SPI, DAB-SLS, RadioTAG, and RadioWEB)

<table>
<thead>
<tr>
<th>Recommendations</th>
<th>The system should be able to offer a centralized instance for importer configuration and monitoring.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The system should provide a recommendation engine that recommends services based on user-selectable services. It should target the recommendation engine requirements, proposed within the overall HRADIO system architecture document, D2.3, Derived Technical Requirements (see item Recommendation Engine)</td>
</tr>
<tr>
<td></td>
<td>The system should provide a modular recommendation engine for service recommendations that can be configured by the user (which recommendation modules to select, which data to use)</td>
</tr>
<tr>
<td>Deployment</td>
<td>The system should provide a standardized way for deployment. Deployment should be as simple as possible.</td>
</tr>
<tr>
<td></td>
<td>The system should be deployable on Ubuntu-based machines</td>
</tr>
<tr>
<td>Security</td>
<td>All public service endpoints should be secured with a combination of basic access authentication and transport layer security (TLS)</td>
</tr>
<tr>
<td>Feature</td>
<td>Requirement</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Privacy-preserving User Data</td>
<td>The system should provide an interactive survey-based component for the collection of user data in a privacy-preserving manner.</td>
</tr>
<tr>
<td>Collection</td>
<td></td>
</tr>
<tr>
<td>Privacy-preserving User Data</td>
<td>The system should provide a component for the automatic collection of user data in a privacy-preserving manner.</td>
</tr>
<tr>
<td>Collection</td>
<td></td>
</tr>
<tr>
<td>Visualization</td>
<td>The system should provide a component for user and metadata visualization (graphs and dashboards).</td>
</tr>
<tr>
<td>Cross-Device Listening</td>
<td>The system should provide a component for storing and exchanging necessary user information for cross-device listening.</td>
</tr>
</tbody>
</table>

### 2.2. ARCHITECTURE

This section provides an overview of the system architecture and the functional components of the Metadata Integration & Dissemination Platform. The architecture is based on the aforementioned requirements.

The Metadata Integration & Dissemination Platform consists of a set of connected nodes that span a network of federated search nodes. Each node consists of metadata specific to a particular set of service providers. The idea is to have an easy-to-deploy platform that is also suitable for small radio stations. In our approach, resource usage is low since every node does not have to store the whole metadata database but only those parts that are relevant for the provided radio services. Searches are distributed across the search federation and might be locally restricted. Thus, resource requirements for smaller stations are kept as low as possible.

Compared to the second iteration of the architecture, the following changes were made:
The containing sub chapters describe the architecture of the following, currently existing platform modules:

- The search & storage module
- The music metadata storage module
- The metadata importer module
- The recommendations module
- The privacy preserving user data collection module
- The dashboard module
- The password manager module

All modules are connected via a message bus which itself uses the RabbitMQ communication middleware (see chapter "Technologies"). See Figure 1 for an overview. A description and evaluation of the architecture (mainly search & storage with the federated search module, music metadata storage, metadata importer and parts of the recommendations module) were published in the Proceedings of the International Conference on Innovations for Community Services (I4CS 2019) [2].

The privacy-preserving user data collection module was added.
The statistics module was added.
The password manager module was added.
The recommendations module was changed and extended.
The search processor component was removed.
The search & storage module was extended.
The data model was extended.
The deployment was updated and simplified.
As depicted in Figure 1, the metadata REST service is accessible from within the company network while the Search REST, the Recommendations REST and the Statistics REST service are publicly available.

### 2.2.1. Metadata Search & Storage

The search and storage module stores radio metadata using two different storage engines (PostgreSQL and Elasticsearch) and provides REST services for metadata maintenance and search. It consists of the following core components...
Search Federation

In this iteration, the search federation was implemented. The search REST is now able to forward search requests to multiple search nodes and to return an aggregated response list. With this architecture extension, we reached one of the central goals of the platform, which is the distributed metadata search and storage (distribution in terms of data storage but also in terms of search query processing). Small radio stations storing only their own (rather small) amount of metadata are connected to a network of metadata search nodes and thus are visible for a larger number of potential listeners. The technical details are as follows:

- **Document Database**: The document database processes search queries and stores the metadata as JSON encoded documents. Currently, the Elasticsearch engine is used (see chapter “Technology”) but the design allows for other document databases as well.

- **Search Indexer**: The search indexer crawls metadata from the PostgreSQL storage, converts it to a specific, search-friendly format and stores it in the document database. This happens at a configurable interval.

- **Relational Database**: Stores metadata in a relational scheme. It is used as the primary source for metadata. The design is based on the Spring Framework and thus is very flexible when it comes to database choice. Currently, PostgreSQL is used (see chapter “Technology”).

- **Search REST Service**: The search REST service can be used for service, service use (station listening trends) and programme search. The search requests are converted to a document database specific query format and sent to the document database. The results are converted to the internal response format and sent back to the search requester. It follows RESTful service principles, uses JSON as transport format and is pageable. The service is publicly available and only HTTPS access is enabled. Some methods are secured using the security module.

- **Metadata REST Service**: The metadata service can be used for metadata maintenance and retrieval. It follows RESTful service principles, uses JSON as transport format and is pageable. The service is available internally.
Each search node has a publicly available description for broad search node filtering (search node description, maintainable via REST endpoints, see Chapter 2.5.1). The federated search module uses a broad query to find other search hosts in its local search node index. Once matching search nodes are found, the search request is forwarded to those hosts. The search results are then collected, merged and sent back to the requester. Request loops are avoided using a request ID. These IDs are kept in a local cache for a configurable time interval. Incoming requests with a cached ID will not be processed. See Figure 2 for a visual explanation of that basic idea. The forwarding process can be recursively continued. The maximum number of hops as well as the maximum number of contacted search nodes per hop can be specified in the search request.

To enter the federation, the search node administrator has to contact a federation member who generates an entry token which is then sent to the requester (see also api/v1/generateToken and api/v1/index REST endpoint in Chapter 2.5.1). Using this token, the administrator can register his own search node in the search node’s (the one that generated the token) search node index.

Changes in the search node description of a host are not propagated. Rather, each search node runs a polling-based updater instance which updates the search node descriptions in its own index in a specified interval.
Figure 2 depicts the basic idea of the Federated Search which is to forward incoming client search requests to other nodes stored in a local table called the “search node index” (1). Each incoming query has two components: The first component (top-level query) specifies a query with which the search node index of the receiving search node is filtered. The second component (detailed query or local query) contains the actual query (2). In the example in (3), the top-level query contains a set of genres which is used to filter the search node index of search node 1 which contains search node descriptions of search nodes 1-3. The third search node is a good match. Thus, the request is forwarded to it.
2.2.2. Music Metadata Storage

The music metadata storage module provides a service to retrieve detailed metadata for specific songs. It consists of the following components:

- **Relational Database**: A database (currently: PostgreSQL) which stores song-related metadata (artist, title, release year, etc.).

- **Discogs Importer**: An importer for Discogs data (https://www.discogs.com/). Discogs is an online music metadata database which provides a rich set of song information. Amongst other endpoints, it provides a downloadable XML file containing a snapshot of the complete database. The importer automatically downloads the file and imports it into the relational database in configurable intervals (e.g. once a month).

The song metadata is used by the ICY Tag crawlers and the programme event importer to enrich incoming programme event data. Programme events are the basis for the service recommender system.

2.2.3. Metadata Importer

The metadata importer module manages the connection between metadata crawlers (for both, service and programme event metadata) and the metadata processing units (service and programme event importer). The whole pipeline ensures the provisioning of high-quality metadata and is designed for high throughput and flexibility. Crawlers for new data sources are simple to develop and integrate since they only have to adhere to a basic messaging protocol.
Figure 3: The main data flow between crawler and importer modules.

Figure 3 shows the main data flow between the crawler, the importer module and the REST service and the underlying generic message protocol. The crawled data is sent either in the JSON ServiceSeed or ProgrammeEventSeed format to the responsible importer module via the corresponding message bus queues. (service_importer_queue or programme_event_importer_queue). The importers store the evaluated and processed data via the Metadata REST service in the internal storage system.

2.2.4. Recommendations

The recommendations component consists of a REST service endpoint with which recommendations can be requested for a specific service hash. The recommendation engine is mostly content-based and offers 6 recommenders which can be combined to deliver service recommendations based on different data sources.

**Trend Recommender:** The trend recommender uses station usage data received from users to distil trends for particular user groups and contexts (age, gender, region, time). Based on those trends, recommendations are
provided for a given user, i.e. “Users that are in the same age group listen to service x.”. The trend recommender is not content-based in that it neither uses programme-accompanying metadata nor metadata extracted directly from a station’s audio stream for recommendations.

- **Expert Recommender:** The expert recommender recommends services based on given expert opinions. An expert opinion consists of a list of services (identified through their hashes) that should be recommended for a particular service. Expert opinions are stored in JSON-format in a folder in the recommender system directory tree named “/expert_opinions”. New expert opinions can be added using an endpoint of the Recommender REST service (see Chapter 2.5.4).

- **Category Recommender:** The basis for category recommendations are the TVA genre tags of services which can be requested with the metadata REST service. The algorithm performs a tree comparison of nested TVA genre categories (TVA genres are hierarchical, e.g. 1.4.1 and thus can be represented as a tree which can be compared to other trees).

- **Histogram Recommender:** The histogram recommender is used to provide service recommendations on the basis of service fingerprints. These fingerprints are the result of a separate processing in which audio features are extracted from the audio streams of the services and post-processed (audio feature extraction). The final result is a vector representation with 11 dimensions for each service stored in the file “/fingerprints/fingerprints.csv” (See Chapter “Fingerprint Extraction” for details). The histogram recommender measures the Euclidean Distance between these representations to generate recommendations (close services are recommended).

- **Location Recommender:** The location recommender recommends the spatially closest services. Therefore, service location data is used. It uses the user’s location as basis.

- **More-Like-This (MLT) Recommender:** This recommender delivers services that are similar based on a set of keywords. Keyword similarity is expressed using the TFIDFS distance metric. The keywords of a service are generated by a speech-to-text module which takes service audio streams as input. The results are vector representations of different dimensionality containing keywords in a specific language e.g. English. They are stored in a .csv file, added to the corresponding service data objects and imported into the data storage engine by the service importer.
Which recommenders to use can be configured and the returned end result is a weighted (weights can also be configured) combination of all recommender results, as can be seen in Figure 4:

![Diagram](image)

Figure 4: The recommender result as a weighted combination of the results of a set of recommenders.

**Fingerprint Extraction**

The audio feature extraction process is designed to create a representative feature vector representation for each service (fingerprints). The pipeline for this process is visualised in the following Figure 5:
Figure 5: Pipeline of the Audio Feature Extraction process. The results are 11-dimensional fingerprints for each service for which distances can be calculated to create recommendations.
The data base for the feature vector calculation are the services HTTP streams. The stream bytes are consumed by the ICY tag crawler and delivered to the audio storage service via the internal message bus. The storage service stores the bytes in the data storage engine and ensures that each audio file covers 24h of the corresponding audio stream. When the length of 24h is reached a new file is started. Next, audio segments are extracted from these files in certain intervals. These segments constitute the data base for feature extraction. This is realized using the Python Toolkit auDeep. First, for each audio segment a Mel spectrogram is created. These Mel spectrograms are then used to train a Recurrent Sequence-to-Sequence Autoencoder. The output of the encoder part is a 1024-dimensional vector representation. These vectors are subsequently clustered into 11 clusters using EM clustering algorithm. The number of clusters was set to 11 because the Silhouette Coefficient showed that this is the optimal cluster number for our setting. The Silhouette Coefficient is a measure for the quality of a clustering. The score lays between 0 and 1 and the higher it is, the stronger is the clustering. Figure 4 shows the Silhouette Coefficient for EM clustering with 9 to 16 clusters.

![Silhouette Coefficient](image)

Figure 6: Silhouette Coefficient applied on EM clustering. It shows a peak for a number of 11 clusters.

Based on the cluster IDs resulting from the EM clustering a service histogram is generated for each service. These histograms are considered as service fingerprints and used by the Histogram Recommender to determine similar services.
For a given service, the recommendation engine returns the n closest services, where closeness is defined by the abovementioned respective distance conception of the applied recommenders. In addition, the user can receive aggregated recommendations by specifying a weighting for the results of multiple recommenders. Per default all requested recommender scores are weighted equally.

The fingerprint extraction process as well as the fingerprint-based recommender system was submitted to the 45th IEEE International Conference on Acoustics, Speech and Signal Processing [4].

2.2.5. Privacy-Preserving User Data Collection

This chapter describes developed technologies for the privacy-preserving collection of user data. All developed modules (surveys and statistics) make use of Randomized Aggregatable Privacy-Preserving Ordinal Response (RAPPOR) which guarantees that the user's privacy is preserved by client-side data-distortion that is conducted before user data is transmitted to the server.

For randomization with RAPPOR, the original response is encoded into an array of 0s and 1s. Then, each individual binary value is inverted based on a certain probability or not. This probability depends on the privacy parameters f, p and q. If this randomization takes place only once, it is calculated with the parameter f and results in a PRR (Permanent Randomized Response). Another possibility of randomization is an additional inversion of the PRR with a probability of p and q. The resulting IRRs (Instantaneous Randomized Responses) are currently not relevant for our use cases but necessary parameters are already included in the data structures in order to be able to extend the system later.

The basic idea behind the proposed system is shown in Figure 7.
Figure 7: Data is distorted locally (microdata before, after). Afterwards, the big picture (all data afterwards) is still visible.

The parameter $f$, also called the “lie-factor” or the “longitudinal privacy guarantee” can be chosen from the range $[0,1]$ and specifies the probability with which a certain bit in the aforementioned bit array is flipped. A bit is set to

- 1 with probability $f/2$
- 0 with probability $f/2$
- the original value with probability $1-f$

$f$ determines how truthfully the input value is output again by the PRR. The higher $f$, the less likely it is that the output value is identical to the input value and the greater the added randomness by randomizing the value. Dependent on the use case, different values for $f$ make sense. We usually choose 0.5 for testing purposes, but Google Chrome’s visited homepage reporting system e.g. uses an 0.75. See more details in [1].

**Interactive Surveys**

In order to gain experience with privacy-preserving, interactive and survey-based user data collection technologies, a client-server system was developed. The server allows for the creation and maintenance of surveys which can be requested by a set of clients. Furthermore, the server collects the filled-in surveys and aggregates a result distribution. Results can be displayed in a small, bar chart-based visualization. The client-server communication is done via REST. Data is exchanged via the JSON format. See a diagram of the architecture in Figure 8.

**Server:** The server is written in Python and can be configured and maintained via a web interface. The server is designed in such a way that all user interactions are possible both via a graphical user interface and REST endpoints. For persistent data storage, a SQLite database is used.
The surveys consist of an unlimited number of questions with freely selectable answer options. Depending on the selected question type (multiple choice (MC) or checkbox (CBX)), only one or more answer options can be specified by the clients at a time. Each created survey can be activated, paused or set to the state “done”. Only if it is active it can be requested and answered by the client. The server can be triggered to actively request all client responses (reports) for all active surveys. The answers are summarized and each question is visualized using a bar chart. Until the associated survey is set to status “done”, the result summaries must always be actively updated.

**Client:** The client is implemented as an Android library. Each client actively sends requests for new active surveys to the server via the REST end point. New surveys are answered by the client (user interaction needed). Depending on the configurable privacy parameter f, the selected answer options of each question are randomized and sent back to the server.

![Diagram of survey process](image)

Figure 8: High-level overview of the prototypical privacy-preserving usage data collection system based on surveys and client-side data distortion.

**Statistics**

The statistics module has two main purposes:

- Privacy-preserving collection of user data. Based on the learnings from the development of the “Interactive Surveys” module, a collection of machine-answered surveys (e.g. Has the user used the “Favourites” feature of the platform-centric app more than once?).

- Privacy-preserving collection of radio service usage which allows for recommendations that are even more individual since they are also based on user data that were accumulated to trends (The trend recommender uses this data).
User data (arbitrary user data or radio service usage) is sent to the platform via a REST service interface. The data is then aggregated and stored in the document database and the relational database. Before sending it to the platform, the client distorts parts of the user data. In general, a user data message consists of two parts: The first part contains the user’s context (location, age group, gender, time zone, …) which is not distorted. The second part consists of a report (e.g., a collection of stations the user listened to in the last couple of hours) which is distorted like in the interactive survey component. There is explicitly no specific id transmitted that could identify a single user. Only the group a user is member of can be identified. This is the privacy trade-off the platform provides.

2.2.6. Dashboard

The dashboard module allows for the easy creation and composition of data visualizations for user-, service- and programme metadata. It is based on Kibana (see Chapter 2.3.3 for details).

2.2.7. Password Manager

The password manager REST service was developed to control user access (authorization) to safety-relevant operations such as manipulating metadata. It is not publicly available and is used to identify and decline unauthorized REST requests in all available REST services (e.g., the metadata REST Service). New users can be generated internally via the password manager REST service. See Figure 9 for details to the authorization mechanism:

![Figure 9: The client authorization mechanism.](image)

The client sends a request to one of the platform’s REST services. The request contains a user name and password combination. To avoid the unencrypted transfer of user name and password, the HTTP-based transport is encrypted using
TLS. Before a REST service handles the request, it checks its authorization information using the Password Manager REST service. All services can be configured to use TLS and authorization. Both mechanisms can be disabled as well (rest.runSecure=true).

2.3. TECHNOLOGIES

This chapter describes key technologies used within the platform.

2.3.1. RabbitMQ

The message bus connects all system modules and allows easy extensibility and scalability. We use RabbitMQ\(^1\) as a message broker which is based on the publish/subscribe communication pattern. The main advantages of RabbitMQ are the following:

\(^1\)https://www.rabbitmq.com/
Industry-proven technology: RabbitMQ has more than 35,000 production deployments and is a well-known option for efficient and secure message-based architectures.

Deployment: RabbitMQ is lightweight and easy to deploy in different environments. We use a pre-defined docker container for deployment. For dealing with high load scenarios, clustering options are available.

Client libraries: RabbitMQ provides good support for a multitude of different operating systems and programming languages (C, C++, Java, Python, JavaScript, C#, ...). Thus, architecture components can be developed in different languages. We used mainly Java but switched to Python for the importer modules.

Communication patterns: RabbitMQ supports most common communication patterns. We use publish/subscribe (RadioDNS and Dynamic Label+ importers) and request/response (Search and Metadata REST service).

Learning curve: RabbitMQ basics are easy to learn. Since we defined our importer interface in terms of RabbitMQ’s implementation of the request/response communication pattern, it is crucial that the technology has a flat learning curve.

Management & monitoring: A built-in tool for infrastructure management and monitoring is part of the RabbitMQ bundle.

RabbitMQ’s basic principles and structures are publishers, queues and consumers: Data from a publisher P (cyan) is sent to an exchange entity X (purple) which transmits incoming messages to a set of connected queues (red):
Figure 10: Communication setting with a publisher P, an exchange entity X and two connected queues. Source: ²

Consumers C_i (blue) listen to new data arriving in a queue and consume it:

Figure 11: Communication setting with a publisher, a connected queue and two consumers. Source: ³

2.3.2. Elasticsearch

Elasticsearch is a document-based search engine based on Apache Lucene. It is implemented in Java and stores documents in JSON format together with an (optional) type mapping for type safe storage and retrieval. Its focus is on scalability through a sophisticated clustering approach and complex queries on textual content. The list of query types includes:

³ https://www.rabbitmq.com/tutorials/tutorial-two-python.html
Thus, Elasticsearch fulfills all our requirements regarding metadata search. For more information on queries, see an explanation of Elastic’s Query DSL online\(^4\).

A typical architecture (which we use as well in a basic configuration) is shown in the following diagram:

![Diagram of Elasticsearch architecture]

**Figure 12: Typical Elasticsearch architecture with data sources, processing, storage and search.** Source:\(^5\)

Data sources (in our case: importers) send their data to the data processing unit (in our case: the metadata indexer). Subsequently, all data is indexed and sent to the Elasticsearch system which distributes it among existing clusters. Search clients then send queries to the clusters and retrieve results. In our case, we implemented an additional, intermediate search processor module, that transfers queries from

---


clients to the Elasticsearch cluster and retrieves results back to the corresponding clients.

Elasticsearch stores documents at a certain location in the file system. We use a pre-configured Docker image that stores data outside of the container on the host’s hard disk.

2.3.3. **Kibana**

Kibana⁶ is a rich, web-based platform for dashboard creation and management (see for more details). It is part of the Elastic Stack and mainly used for the visualization of data stored in Elasticsearch instances. It is seamlessly integrated in the HRADIO communication platform and can be used to visualize all data stored in the deployed Elasticsearch instance (user data, service and programme metadata).

⁶ [https://www.elastic.co/de/products/kibana](https://www.elastic.co/de/products/kibana)
2.3.4. PostgreSQL

We use PostgreSQL\(^7\) as relational database for radio and music metadata storage. PostgreSQL is an open source SQL database system that is very reliable and robust. To map the Java-based data model to the relational scheme, we use the Spring Framework and in particular the Spring Data module\(^8\).

2.4. DATA MODEL

The data model describing radio related metadata is partially a subset of the RadioDNS standard that was extended in several aspects (genre definition, programme events, service fingerprints, service keywords and so on). Compared to

---

\(^7\) [https://www.postgresql.org/](https://www.postgresql.org/)

\(^8\) [https://spring.io/projects/spring-data-jpa](https://spring.io/projects/spring-data-jpa)
In the second iteration, it was further extended to meet the requirements of the service recommender system. The schema for the Elasticsearch search engine was updated accordingly to reflect the accompanying changes made to the data model.

The following diagram depicts the elements of the metadata data model:

![Diagram of the designed metadata data model](image)

Figure 14: The designed metadata data model.

The metadata model consists of the following elements:
Service Provider: A service provider (i.e. radio station) is the superordinate entity of services and contains a name, description and id. Example: Bayerischer Rundfunk (BR).

Service: A service describes a radio service with name and description. In addition, a service has a link to its parent (service provider). Media descriptions, bearers and a locale are as well associated to a service. Furthermore, fingerprints and keywords data fields were added in this iteration to represent services.

Schedule: A schedule defines a set of programmes for a service and a specific time period defined by start and stop time (usually, the time period is exactly one day).

Scope: Defines the scopes of a schedule (identifiers of the services including programmes are part of).

Bearer: Defines bearers for services and programmes. A bearer consists of an address, a bit rate and a certain type (e.g. DAB).

Media Description: Defines the media descriptions for services and programmes. A media description consists e.g. of an image path, a description and a resolution.

Location: Defines the time zone, GeoPoint and country code of a certain service.

GeoPoint: Geo points consist of latitude and longitude coordinates to describe a location.

Programme: Defines programmes with description, start and stop time and genres. Programmes are part of a schedule. A programme contains a link to its schedule (scheduleId) and a link to its service (serviceId). A programme might also be associated with one or multiple web contents.

WebContent: Web contents consist of a link to e.g. a podcast of a certain programme, a description and a mime type.

Programme Event: Programmes consist of programme events. A programme event can be, e.g. a certain song that is played (information might come from the Dynamic Label+ converter). A programme event has a start and stop time and a content and genre field to describe the content of the event.
The genre information is generally stored in a generic JSON formatted string which is depicted in the following listing:

```json
{
    "genres": "[
    {
        "data_source": "tva",
        "genres": [
        {
            "name": "spezielle Musik",
            "description": "urn:tva:metadata:cs:ContentCS:2002:3.6",
            "ratio": 0.5
        },
        {
            "name": "Klassische Musik",
            "description": "urn:tva:metadata:cs:ContentCS:2002:3.6.1",
            "ratio": 0.5
        }
    ]
}
}
```

Genre data can be categorized by its data source, respectively the underlying data schema. Currently, the source types “tva” and “DISCOGS” are available. The Dynamic+ Label data structure also provides a GENRE field, but is currently not in use.

The metadata model can be extended easily. What has to be changed is the Java data model in the eu.hradio.metadata package, its corresponding Elasticsearch mapping definitions (as implemented as well in the eu.hradio.metadata package) and the Python data model.

### 2.5. MODULES

This chapter focuses on the description of the different modules the platform consists of, and on how to use them (configuration, service endpoint descriptions) in practice.

#### 2.5.1. Metadata Search & Storage

The search and storage engine stores metadata and processes search requests. It uses two different storage schemes: One relational database (currently PostgreSQL) as primary storage that stores data using a fine grained, relational data model and a document database optimized for fast text searches (Elasticsearch) that stores
normalized (flat) representations of service and programme data. The initial architecture had only a single database system (Elasticsearch) but it turned out that a relational database is better suited to store the complete data model with its hierarchical structure (service providers – services – schedules – programmes – programme events).

**Relational Database**

The relational database is the main storage for metadata. It contains the complete, hierarchical structure of the data model. We currently use the PostgreSQL database system. For database maintenance, we use the GUI-based tool pgadmin⁹.

**Document Database**

The Search REST service endpoint is able to process complex queries. These queries are executed by the document database which is currently Elasticsearch. The document database stores a specific, flat structure of programme and service information which is optimized for fast queries.

For maintenance tasks, we suggest using external tools that are part of the Elasticsearch ecosystem, like Curator¹⁰. For additional data visualization, Kibana¹¹ is a suitable tool which is also integrated in the platform.

**Search Indexer**

The search indexer reads service and programme metadata from the metadata REST service and stores it in the document storage (Elasticsearch).

The search indexer can be configured by editing the application.properties file located in search_indexer/build/resources/main/:

```
rabbitmq.address=rabbit
```

⁹ [https://www.pgadmin.org/](https://www.pgadmin.org/)

¹⁰ [https://www.elastic.co/guide/en/elasticsearch/client/curator/5.6/about.html](https://www.elastic.co/guide/en/elasticsearch/client/curator/5.6/about.html)

¹¹ [https://www.elastic.co/products/kibana](https://www.elastic.co/products/kibana)
rabbitmq.port=5672
rabbitmq.user=rabbitmq
rabbitmq.password=rabbitmq
rabbitmq.retries=100
rabbitmq.retryInterval=3000

elastic.protocol=http
elastic.address=elasticsearch
elastic.port=9200
elastic.requestTimeout=3000
elastic.retries=100
elastic.retryInterval=3000

# Important: If metadata should be persisted independent of system restarts.
# this should be set to false.
elastic.deleteIndicesIfExists=false

metadata.baseURL=http://localhost:8090/api/v1
metadata.requestTimeout=3000

index.updateInterval=60
index.initialDelay=2
index.timeUnit=MINUTES

logging.file=./log/search_processor.log
logging.level.root=INFO

After a change of this file the search indexer must be restarted. Log files can be found in search_indexer/log.

**Search REST Service**

The search REST service is the interface for service and programme as well as service use searches. The search requests are sent to the search & storage engine where they are processed by the Elasticsearch engine. Additionally, the search REST service is used to configure the behaviour of the federated search processing using a search node index of search node descriptions. This service is publicly available and restricted to HTTPS access. The following endpoints are available:
The REST interface looks like this:

- `/api/v1/services`: Search for services
- `/api/v1/programmes`: Search for programmes
- `/api/v1/index`: Maintenance of the search node index
- `/api/v1/generateToken`: Generation of tokens for index maintenance
- `/api/v1/service_uses`: Search for service uses
- `/api/v1/kvs`: A non-persistent key-value store used for cross-device listening

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>URL Query. Syntax: ^12 Returns a list of ranked results.</td>
<td>q: Query string (example: <a href="http://141.84.213.235:8080/api/v1/services?q=name:B">http://141.84.213.235:8080/api/v1/services?q=name:B</a>*) \n</td>
</tr>
</tbody>
</table>

### POST Complex Query

**Syntax:**
- Returns a list of ranked results.

**Parameters:**
- `size`: Number of items in the result set
- `from`: Start index in the result set.

---

**Example 1:**

```bash
gPost http://141.84.213.235:8080/api/v1/services?size=1
```

**BODY:**

```json
{
    "localQuery": {
        "syntax": "ELASTIC_COMPLEX",
        "query": {
            "nested": {
                "path": "location",
                "score_mode": "avg",
                "query": {

Example 2:

POST http://141.84.213.235:8080/api/v1/programmes

BODY:

```json
{
  "localQuery": {
    "syntax": "ELASTIC_COMPLEX",
    "query": {
      "bool": {
        "must": [
          {
            "range": {
              "startTime": {
                "gte": "Dec 19, 2018 23:00:00"
              }
            }
          },
          {
            "locations": {
              "geo_distance": {
                "distance": "2000km",
                "location": {
                  "lat": 50.8530341070593,
                  "lon": 4.402697564996515
                }
              }
            }
          }
        ]
      }
    }
  }
}
```
Example 3 Federated Search:

GET http://141.84.213.235:8080/api/v1/services?q=name:B*&tlq=keywords:music&maxBreadth=1&maxDepth=1&size=100

Example 4 Federated Search:

GET http://141.84.213.235:8080/api/v1/services

BODY:

{  
  "topLevelQuery": {  
    "syntax": "ELASTIC_COMPLEX",  
    "query": {  
      "bool": {  
        "must": [  
          {  
            "match": {  
              "genres": "classical"  
            }  
          ]  
        ]  
      }  
    }  
  },  
  "localQuery": {  
    "range": {  
      "stopTime": {  
        "lte": "Dec 20, 2020 23:05:00"  
      }  
    }  
  }  
}
"syntax": "ELASTIC_COMPLEX",
"query":{
  "bool":{
    "must":[
      {
        "wildcard":{
          "name":"forest*fm"
        }
      }
    ]
  }
},
"maxBreadth": 2,
"maxDepth": 2,
"timeout": 2000,
"searchFederationMode": "FEDERATED",
"id": 2000
}

Please note that special characters are escaped with "\\".

**Result (Example 1):**

RETURN CODE: 200

BODY:

```json
{
  "content":{
    "source":"http://141.84.213.235:8080/api/v1/",
    "rank":1.0,
    "content":{
      "type":"service",
      "name":"Fix Radio",
      "description":null,
      "providerName":"UNKNOWN",
      "genres":",
      "features":",
      "keywords":"fixradio",
```
"bearers": [ 
    {
        "address":"dab:ce1.c186.cdda.0",
        "bitrate":0,
        "type":"DAB",
        "mimeType":"audio/mpeg"
    }
],
"mediaDescriptions": [ ... ],
"location":{
    "timezone":"Europe/London",
    "geoPoint":{
        "lat":51.522938333333336,
        "lon":-0.18310516666666662
    },
    "countryCode":"GB",
    "distance":0
},
"hash":"59F1E0689C8560D1D9EE66C3EDBBF4B4"
"pageable":{
    "sort":{
        "sorted":false,
        "unsorted":true
    },
    "pageSize":1,
    "pageNumber":0,
    "offset":0,
    "paged":true,
    "unpaged":false
},
"totalPages":1,
"last":true,
"totalElements":1,
"sort":{
    "sorted":false,
    "unsorted":true
}
Result (Example 2):

RETURN CODE: 200

BODY:

{  
  "content": [  
    {  
      "source": "http://141.84.213.235:8080/api/v1/",  
      "rank": 2.0,  
      "content":  
        {  
          "type": "StandaloneProgrammeDTO",  
          "mediaDescriptions": [],  
          "webContents": [],  
          "name": "Ohrenbär - Radi",  
          "longName": "Ohrenbär - Radiogeschichten für kleine Leute",  
          "description": "Das gestohlene Haus

Von Heidi Knetsch und Stefan Richwien

Es liest Jürgen Thormann

Aufnahme des RBB",  
          "hash": "726545181E96436E0F1974DCBFF6083F",  
          "genres": "",  
          "features": "",  
          "serviceHash": "95E450A5CC8BC2347BFAB53787D6C973",  
          "startTime": "Oct 15, 2019 17:50:00",  
          "stopTime": "Oct 15, 2019 18:00:00",  
          "programmeEvents": []  
        }  
    }  
  ]
}
Result (Example 3):

RETURN CODE: 200

BODY:

{  
  "content": {  
      "source": "http://141.84.213.235:8080/api/v1/",  
      "rank": 1.0,  
      "content": {  
          "type": "service",  
          "name": "BAYERN 1 Franken",  
          "description": null,  
          "content": [  
              "source": "http://141.84.213.235:8080/api/v1/"]
      }
  }
}
"providerName":"BR",
"genres":",
"features":",
"keywords": news traffic weather bavaria best region podcast separate broadcast hour",
"bearers":[]
"mediaDescriptions":[]
"location":{
"timezone":"Europe/Berlin",
"geoPoint":{
"lat":49.45313,
"lon":11.07433
},
"countryCode":"de",
"distance":0
},
"hash":"A529607000BE8C00D9359AD4ADE4C801"
}
},
{
"source":"http://141.84.213.231:8080/api/v1/",
"rank":1.0,
"content":{
"type":"service",
"name":"B5 aktuell",
"description":null,
"providerName":"BR",
"genres":",
"features":",
"keywords": live import podcast current world repeat watch background report political",
"bearers":[]
"mediaDescriptions":[]
"location":{
"timezone":"Europe/Berlin",
"geoPoint":{
"lat":48.142689,
"lon":11.553805
},
"countryCode":"de"
"distance": 0

"hash": "12623581A54338B80026419C0EB0CEBD"

"pageable": {
  "sort": {
    "sorted": false,
    "unsorted": true
  },
  "pageSize": 2,
  "pageNumber": 0,
  "offset": 0,
  "paged": true,
  "unpaged": false
},
"last": true,
"totalPages": 1,
"totalElements": 2,
"sort": {
  "sorted": false,
  "unsorted": true
},
"numberOfElements": 2,
"first": true,
"size": 2,
"number": 0

Result (Example 4):

RETURN CODE: 200

BODY:

{
  "content": [
    {
      "source": "http://141.84.213.235:8080/api/v1/
    },
Please note that search request parameters (e.g. maxDepth, maxBreadth, ...) are checked and sanitized before entering the search processing process in order to avoid scenarios where the system is overwhelmed due to parameter values beyond reasonable bounds.

- `/api/v1/index(/self | [search node id])`:

Search node descriptions contain an id and a URL. They are described by genres, keywords and their location. The hash value is unique across systems and is used to identify search node descriptions.

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Retrieval of the whole search index.</td>
<td>With additional <code>/[search node id]</code>, a particular search node description can be selected, with <code>/self</code> the description of the currently deployed node is returned.</td>
</tr>
<tr>
<td>Access</td>
<td>Public, no authorization</td>
<td></td>
</tr>
<tr>
<td>Body</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

GET `http://141.84.213.235:8080/api/v1/index`

**Result:**

RETURN CODE: 200
BODY:
[
  {
    "id":0,
    "genres":[
      {"name":"hip hop","description":"","ratio":"0.4"},
      {"name":"blue","description":"","ratio":"0.5"},
      {"name":"jazz","description":"","ratio":"0.1"}
    ],
    "url":"https://searchrest:8080/api/v1/",
    "keywords":[
      "hip hop",
      "jazz",
      "music"
    ],
    "location":{
      "timezone":"Europe/Amsterdam",
      "geoPoint":{
        "lat":11.59428311,
        "lon":48.1500233
      },
      "countryCode":"de",
      "distance":1
    },
    "hash":"605b0c61f7fbcadd7cb7b9b436b4f9f09bbf7af"
  },
  {
    "id":4,
    "genres":[
      {"name":"classical","description":"","ratio":"1.0"}
    ],
    "url":"http://irt.de:8080/api/v1",
    "keywords":[
      "music",
      "news"
    ],
    "location":{
      "timezone":"Europe/Amsterdam",
      "geoPoint":{
        "lat":1.0,
        "lon":1.0
      }
    }
  }
]
Search node descriptions can also be updated (PUT) and deleted (DELETE). These methods can only be accessed privately and need authorization. The process of adding a new search node description uses the POST operation looks as follows:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Add a search node description to the index</td>
<td>Token: A token (e.g. 86faf13a-cd27-417b-b354-5949a2f2e3ba)</td>
</tr>
</tbody>
</table>

**Access**

Public, token-based authorization

**Body**

The body must contain a valid search node description.

**Example:**

```
POST http://141.84.213.235:8080/api/v1/index?token=86faf13a-cd27-417b-b354-5949a2f2e3ba

BODY:
{
  "genres": [{
    "name": "hip hop",
    "description": "",
    "ratio": 0.4
  }, {
    "name": "blues",
    "description": "",
    "ratio": 0.5
  }, {
    "name": "jazz",
    "description": "",
    "ratio": 0.1
  }],
  "url": "https://searchrest:8080/api/v1/",
  "keywords": [
    "jazz",
    "blues"
  ],
  "location": {
    "countryCode": "de",
    "distance": 1,
    "hash": "dad1073c8472c0946995344f2195667a421e529c"
  }
}
“timezone”:”Europe/Amsterdam”,
“geoPoint”:{
“lat”:13.59428311,
“lon”:48.1500233
},
“countryCode”:”fr”,
“distance”:123
}

RESULT:
RETURN CODE: 200
BODY:
[Same body as the request had]

• /api/v1/generateToken:

Tokens are generated and sent to search nodes operators whose search node should be added to the index. The token can be used by the operator to add itself to the index of the search node that generated the token. For that, the operator can use the POST request on the api/v1/index endpoint with its search node description as body and the token as parameter.

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Generates a token</td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>Private, authorization</td>
<td></td>
</tr>
<tr>
<td>Body</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GET http://141.84.213.235:8080/api/v1/generateToken

RESULT:

RETURN CODE: 200

BODY:
86faf13a-cd27-417b-b354-5949a2f2e3ba

- /api/v1/service_uses:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Generates a token</td>
<td></td>
</tr>
</tbody>
</table>

Access  Public, no authorization

Body

- Example:

GET http://141.84.213.235:8080/api/v1/generateToken

RESULT:

RETURN CODE: 200

BODY:
86faf13a-cd27-417b-b354-5949a2f2e3ba

- /api/v1/kvs/[key]:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Retrieves a value for a certain key. Returns code 200 with value in the body if entry for key exists, code 404 if not.</td>
<td>Key parameter encoded in URL</td>
</tr>
</tbody>
</table>
### Verb | Description | Parameters
--- | --- | ---
POST | Creates a new key value pair. Returns code 201 if it was created, code 409 if an entry with that key already exists. If the key is has not between 3 and 128 characters or the value has more than 1024 characters, code 400 is returned. | **key**: The entry’s key. Must have between 3 and 128 characters (configurable via the service’s application.properties). |

### Access | Public, no authorization

### Body

A string representing the entry’s value. Maximum string size is configurable in the service’s application.properties file (standard: 1024 characters).

### Example:

**POST** [http://141.84.213.235:8080/api/v1/kvs?key=123](http://141.84.213.235:8080/api/v1/kvs?key=123)

**BODY:**

123123
RESULT:
RETURN CODE: 201
BODY:
123123

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>Updates an existing entry’s value. Returns code 200 if it succeeded, code 404 if entry with specific key does not exist and code 400 if the size of the new value exceeds the limit of 1024 characters</td>
<td>Key parameter encoded in URL</td>
</tr>
</tbody>
</table>

Access
Public, no authorization

Body
A string representing the entry’s new value. Maximum string size is configurable in the service’s application.properties file (standard: 1024 characters).

Example:
PUT [http://141.84.213.235:8080/api/v1/kvs/123](http://141.84.213.235:8080/api/v1/kvs/123)
BODY:
new value
RESULT:
RETURN CODE: 200
BODY:
new value
<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>Deletes an entry based on its key. Returns code 200 with value in the body if entry for key exists, code 404 if not.</td>
<td>Key parameter encoded in URL</td>
</tr>
</tbody>
</table>

**Access**

Public, no authorization

**Body**

- 

**Example:**

DELETE http://141.84.213.235:8080/api/v1/kvs/123

**RESULT:**

**RETURN CODE:** 200

**BODY:**

123123

The search REST service can be configured by editing the `application.properties` file located in:

`search_rest/build/resources/main/`

```properties
service.version=1.2.1
server.servlet.contextPath=/api/v1/
server.port=8080
server.ssl.key-store=./security/.keystore
server.ssl.key-store-password=hradio
server.ssl.key-password=hradio

# Authentication
password_manager.baseUrl=http://localhost:8111/api/v1
```
## ElasticSearch
```
elastic.protocol=http
elastic.address=localhost
elastic.port=9200
elastic.requestTimeout=3000
elastic.retries=100
elastic.retryInterval=3000
elastic.deleteIndicesIfExisting=true
```

## REST related config
```
rest.requestTimeout=3000
rest.federatedRequestTimeout=10000
rest.dailyPollingInterval=1
rest.runSecure=true
```

## Request Cache
```
cache.size = 20
```

## Logging configuration
```
logging.file=./log/search_rest.log
logging.file.max-size=10KB
logging.file.max-history=1

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
```

## Search Node Description
```
description.url=https://localhost:8080/api/v1/
description.keywords = music,news
description.genres = hip hop, pop, electronic
description.regions = bavaria
description.location =
{'countryCode':'de','geoPoint':{'lon':1.5942831,'lat':48.1500233},'timezone':'Europe/Amsterdam','distance': 1}
```

## KVS
```
kvs.maxValueSize=1024
kvs.minKeySize=3
kvs.maxKeySize=128
```

After a change of this file, the search REST service must be restarted. Log files can be found in search_rest/log.

**Metadata REST Service**

The metadata REST service provides a maintenance interface for metadata. The following endpoints are available:

- `/api/v1/providers/{providerId}/services/{serviceId}/schedules/{scheduleId}/programmes/{programmeId}/events/{eventId}`

Description of the place holders:
Ids are assigned automatically upon resource creation. All endpoints are private and cannot be accessed from outside the company network. The service has the following interface:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Paged retrieval of service providers, services, schedules, programmes and programme events. If an item id is given, a particular item is returned. If a collection is given (e.g. providers) an item list is returned. Returns 404 if item does not exist, 200 if it does exist.</td>
<td>size: Size of a page page: Page index</td>
</tr>
</tbody>
</table>

Example (1):

GET http://141.84.213.235:8090/api/v1/providers?size=1

RESULT:

RETURN CODE: 200

```
{
    "content": [
        {
            "id": 1,
            "name": "BR",
            "description": null,
            "services": [
```
290,
2,
67,
1765,
1863,
234,
162,
399,
529,
340,
501,
602,
1722,
1789,
126
],
}
"pageable": {
  "sort": {
    "unsorted": true,
    "sorted": false
  },
  "pageSize": 1,
  "pageNumber": 0,
  "offset": 0,
  "unpaged": false,
  "paged": true
},
"last": false,
"totalElements": 7,
"totalPages": 7,
"sort": {
  "unsorted": true,
  "sorted": false
},
"numberOfElements": 1,
"first": true,
"size": 1,
"number": 0
Example (2):

GET http://141.84.213.235:8090/api/v1/providers/1/services/2/schedules/11/programmes?size=1

RESULT:

RETURN CODE: 200

BODY:

{
   "content": [{
      "id": 20,
      "parentId": 11,
      "startTime": "Jan 06, 2019 23:00:00",
      "stopTime": "Jan 06, 2019 23:05:00",
      "programmeEvents": [],
      "mediaDescriptions": [],
      "name": "Nachrichten, Wet",
      "longName": "Nachrichten, Wetter, Verkehr",
      "description": null,
      "genres": ""
   }],
   "pageable": {
      "sort": {
         "unsorted": true,
         "sorted": false
      },
      "pageSize": 1,
      "pageNumber": 0,
      "offset": 0,
      "unpaged": false,
      "paged": true
   },
   "last": false,
   "totalElements": 19,
   "totalPages": 19,
   "sort": {
      "unsorted": true,
**POST**

Creation of service providers, services, schedules, programmes and programme events.

**Body**

The body must contain the data necessary for the particular item.

**Example:**

POST http://141.84.213.235:8090/api/v1/providers

**BODY:**

```
{
  "name": "ServiceProviderTest1",
  "description": "A test service provider"
}
```

**RESULT:**

**RETURN CODE: 201**

---

**PUT**

Updates an existing service provider or service.

**Body**

The body must contain the data necessary for the particular item.
Example:
POST http://141.84.213.235:8090/api/v1/providers/1

BODY:
{}
  "name": "ServiceProviderTest1",
  "description": "A test service provider"
}

RESULT:
RETURN CODE: 201

Example bodies for the creation and update of metadata items:

- **Service providers:**

  ```json
  {
    "name": "BR",
    "description": "A sample service provider"
  }
  ```

- **Services:**

  ```json
  {
    "name": "BAYERN 3",
    "genres": [
      {"data_source": "dns",
       "genres": [
         {"name": "Pop Musik",
          "description": "urn:tva:metadata:cs:ContentCS:2002:3.6.4.1",
          "ratio": 0.5}, ...
       ],
      "bearers": [
        {"address": "http://streams.br.de/bayern3_1.m3u",
         "bitrate": 56,
         "type": "HTTP"}, ...
      ],
      "mediaDescriptions": [
        {"field0": "http://epg4br.irt.de/radioDNS/spi/3.1/logos/br_bayern-
```
Schedules:

```
{
  "startTime": "Jan 08, 2019 23:00:00",
  "stopTime": "Jan 09, 2019 23:00:00"
}
```

Programmes:

```
{
  "startTime": "Jan 08, 2019 23:00:00",
  "stopTime": "Jan 09, 2019 04:00:00",
  "name": "Die Nacht",
  "description": "Mit Lea Geishauser jeweils zur vollen Stunde BAYERN 3-
Nachrichten"
}
```

Programme Events:

```
{
  "startTime": "Jan 08, 2019 23:06:52",
  "stopTime": "Jan 08, 2019 23:10:28",
  "content": {
    "raw_data": {
      "ICY": {
        "programme_event_type": "SONG",
        "data": {
          "artist_names": [
            "lady gaga"
          ]
        }
      }
    }
  }
}
```
"bradley cooper",
"track_title": "shallow",
"start_time": "Jan 08, 2019 23:06:52",
"stop_time": "Jan 08, 2019 23:10:29",
"DISCOGS": {
  "data": {
    "artist_names": [
      "Lady Gaga"
    ],
    "track_title": "Shallow",
    "track_duration": "0:03:36",
    "release_released": 2018,
    "release_genres": [
      "Rock",
      "Non-Music",
      "Pop",
      "Stage & Screen"
    ]
  },
  "start_time": "Jan 08, 2019 23:06:52",
  "stop_time": "Jan 08, 2019 23:16:52",
  "programme_event_type": "SONG"
},
"artists": [
  "lady gaga",
  "bradley cooper"
],
"title": "shallow",
"duration": "0:03:36",
"release_year": 2018
},
"genres": [
  {
    "data_source": "DISCOGS",
    "genres": ["Rock",
  ]
]
"Non-Music",
"Pop",
"Stage & Screen"
]
}
]
}

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>Deletion of a single service provider, service, schedule, programme or programme event. Returns 404 if item does not exist, 200 if it does exist.</td>
<td>-</td>
</tr>
</tbody>
</table>

**Body**

```
```

**Example:**
DELETE http://141.84.213.235:8090/api/v1/providers/1/services/2

**RESULT:**

**RETURN CODE: 200**

The metadata REST service can be configured by editing the application.properties file located in metadata_rest/build/resources/main:

```properties
server.servlet.contextPath=/api/v1/
server.port=8090

rabbitmq.address=rabbit
rabbitmq.port=5672
rabbitmq.user=rabbitmq
rabbitmq.password=rabbitmq
rabbitmq.retries=100
rabbitmq.retryInterval=3000

rest.requestTimeout=3000
```
After a change of this file, the metadata REST service must be restarted. Log files can be found in metadata_rest/log.

### 2.5.2. Music Metadata Storage

The music metadata storage holds song-related metadata in a relational database. An importer imports data from the “Discogs” online service in a configurable time interval. Song metadata is used by the metadata importer component to enrich programme events.

This component is designed as a stand-alone tool which can be used without any other module of the platform. It can be configured by changing the src/settings.py file in the source directory:

```python
DATABASE = 'user=postgres password=postgres host=db_dc port=5431 sslmode=disable'
DBNAME_TMP = 'discogs_tmp'
DBNAME = 'discogs'
DC_FILES = './discogs_files/
DISCOGS_DATA_URL = 'https://data.discogs.com'
INTERVALL = 4 # in hours
TABLES = [
    'artist',
    'label',
    'master',
    'master_artist',
    'release',
    'release_artist',
    'release_company',
    'release_extraartist',
    'release_format',
    'release_identifier',
    'release_label',
    'release_video',
    'track',
]```
Most important is the "INTERVALL" field which sets the interval in which the importer checks for new data versions. See https://gitlab.irt.de/hradio/platforms/CommunicationPlatform/importer_discogs/blob/master/README.md for a detailed manual with installation instructions.

2.5.3. Metadata Importer

The metadata importer system includes the crawling and importing of service and programme event related metadata. Its flexible architecture allows for different data crawlers.

Currently, a service importer exists for the RadioDNS standard. Programme event importers are available for Dynamic Label+ data (coming from a DAB Tuner, sent over a STOMP broker and converted to JSON) and IP stream-based ICY-tags. For internal communication, the message bus is used. The final and processed data is stored in the search and storage module using the metadata REST service.

Central element is the scheduler which coordinates the initialization of crawlers and importer components. It is fully configurable via a JSON file:

```json
{
    "services": {
        "rmq_queue": "service_metadata_queue",
        "crawlers": [
            {
```


"name": "dns",
"python_module_path": "./crawler_dns/main.py",
"config_path": "./crawler_dns/config/dns_config.json",
"required": ["rmq"]
},

"importer": {
  "name": "service",
  "python_module_path": "./service_importer/main.py",
  "required": ["rmq", "rest"]
},

"programme_events": {
  "rmq_queue": "programme_event_metadata_queue",
  "crawlers": [
    {
      "name": "dl",
      "python_module_path": "./crawler_dl/main.py",
      "config_path": "./crawler_dl/config/dl_config.json",
      "required": ["rmq"]
    },
    {
      "name": "icy",
      "python_module_path": "./crawler_audio_icy/main.py",
      "config_path": "./crawler_audio_icy/config/icy_config.json",
      "instances": [
        {
          "service": "BAYERN 3"
        },
        {
          "service": "BAYERN 1 Franken"
        }
      ],
      "required": ["dc", "rmq"]
    }
  ],
  "importer": {
    "name": "programme event",
    "python_module_path": "./programme_event_importer/main.py",
    "required": ["dc", "rmq", "rest"]
Most important configuration elements are:
Services: Describes the service information crawlers (currently RadioDNS only) that should be started (with script and configuration file) and the service importer to use.

Programme Events: Describes the programme event information crawlers (Dynamic Label+, ICY) that should be started (with script and configuration file) and the programme event importer to use.

RMQ: Message bus connection information.

REST: URL of the metadata REST service to use.

DISCOGS: The connection data of the music database is necessary only for the programme event importer. It enhances the song programme events with genre, track duration and release year data stemming from the Discogs music database.

Please note that if the programme event importer receives data for the same programme event stemming from multiple sources (like different crawlers or Discogs) the data is labelled with the corresponding data source and merged. Before a programme event is considered as ready for import, the importer waits 10 minutes because other crawlers might send data for the same event.

In the following chapters, available crawlers are described.

General Message Protocol

The general message protocol used by the metadata importer system is depicted in the following table. The destination and source queues are part of the message header to enable bidirectional module communication.

```
{
  "header": {
    "date": "2019-14-01T00:00:00",
    "destination": <target_message_queue>,
    "source": <source_message_queue>
  },
  "body": {
    one of the following JSON objects:
    - <service_provider>
  }
```
Service Crawlers

1.1.1.1.1 RadioDNS Crawler

The Importer for RadioDNS data is written in Python and can be configured with a JSON-based configuration file as depicted in the following example:

```json
{
  "Crawlers": [
    {
      "Every": 86400,
      "Config": [
        {
          "Url": "http://epg4br.irt.de/radiodns/spi/3.1/SI.xml"
        },
        {
          "Url": "http://epg4rbb.irt.de/radiodns/spi/3.1/SI.xml"
        },
        {
          "Url": "http://epg4mdr.irt.de/radiodns/spi/3.1/SI.xml"
        },
        {
          "Url": "http://epg4ndr.irt.de/radiodns/spi/3.1/SI.xml"
        },
        {
          "Url": "http://epg4wdr.irt.de/radiodns/spi/3.1/SI.xml"
        },
        {
          "Url": "http://epg4swr.irt.de/radiodns/spi/3.1/SI.xml"
        },
        {
          "Url": "http://epg4hr.irt.de/radiodns/spi/3.1/SI.xml"
        }
      ]
    }
  ]
}```
RadioDNS provides SI files which contain service specific data and PI files which store schedule and programme data for a corresponding service. The crawler downloads SI files from a set of URLs and tries to combine them in each case with the corresponding PI file of the current and the next day. It is executed in a certain frequency (Example: every 86400s, i.e. once per day).

The output is a set of service providers and services with schedule information attached to each service. The result is provided in XML (since RadioDNS uses XML for content description) and sent to the service importer which does the conversion to the internal metadata format.

**Programme Event Crawlers**

1.1.1.1.2 *Dynamic Label*+

The importer for Dynamic Label+ data is written in Python and can be configured with a JSON-based configuration file as depicted in the following example:

```json
{
    "Stomp": {
        "Config": "tcp://193.96.226.192:61613",
        "Queue": "/hradio/metadata"
    }
}
```

Its essential parameters are the connection parameters for the STOMP server.

The Dynamic Label+ information is encoded in JSON and follows the structure described in the following example:

```json
{
    "DynamicLabel": {
        "DlsText":"DO YOU REALLY WANT TO HURT ME – CULTURE CLUB",
        "EnsembleEcc":224,
        "EnsembleId":4284,
        "EnsembleLabel":"DR Deutschland",
        "ItemRunning":true,
        "ItemToggle":false,
        "ServiceId":6138,
    }
}
```
The Dynamic Label crawler converts this data structure into the internal programme event JSON format:

```
{
  "service_id":1765,
  "programme_event_type":"SONG",
  "stop_time":"Jan 14, 2019 07:37:45",
  "start_time":"Jan 14, 2019 07:41:52",
  "data":
  {
    "artist_names":
    [
      "Lorde",
    ],
    "track_title":"Green Light (Chromeo Remix)",
    "track_duration":"0:04:07",
    "release_released":2017,
    "release_genres":
    [
      "Pop"
    ]
  }
}
```
The crawler collects data from the STOMP server and sends it via the internal message bus to the programme event importer. The programme event importer converts the incoming Dynamic Label+ data into programme events by evaluating the toggle bit in order to get exact start and stop times.

The system that converts the Dynamic Label+ data to the defined JSON format and sends it via STOMP was developed at IRT. It consists of a PC with multiple USB-Stick DAB tuners connected to it and a free implementation of a STOMP server running on it.

1.1.1.1.3 ICY Tag

The Importer for ICY Tag data is written in Python and can be configured with a JSON-based file configuration as depicted in the following example:

```json
{
   "audio_module_path":<audio_storage_service_module_path>,
   "audio_config_path":<audio_storage_service_config_path>
}
```

For each crawled service, a stand-alone ICY Tag crawler instance is executed. If it is defined in the configuration file, each ICY crawler executes an audio storage service instance to store the audio data of the service audio stream in parallel. The configuration of the audio storage service consists of the storage root path for the stream data. The resulting mp3 files have a maximum length of 24 hours.

The metadata of each ICY Tag stream is structured individually and provides inter alia song information or news. Some examples are shown in the table below.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;StreamTitle='Oasis - Whatever';\x00&quot;</td>
</tr>
<tr>
<td>&quot;StreamTitle='PULS - Dieses neue Radio';\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'</td>
</tr>
<tr>
<td>&quot;StreamTitle='Internet:www.bayern3.de';\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'</td>
</tr>
<tr>
<td>&quot;StreamTitle='+++ Tausende Helfer kämpfen weiter gegen ungeheure Schneemassen';\x00&quot;</td>
</tr>
</tbody>
</table>

The ICY Tag crawler analyses the incoming ICY tags using the Discogs database to test if the tag contains song data. If it does, it creates a song programme event and delivers it via the internal message bus to the programme event importer. In the present state only song events are forwarded by the ICY crawler.
2.5.4. Recommendations

The recommendations module provides a REST service which is publicly available and secured using HTTPS. With it, recommendations for a given service can be retrieved. The recommendation engine offers different recommendation strategies whose similarity scores can be retrieved stand-alone or aggregated using a user-defined weighting. Per default, all recommender scores are weighted equally, but the weights can be defined for each request. The implemented recommenders differ in their data basis and concept of similarity, i.e. their mathematical foundation and are described in Chapter 2.2.4.

The following endpoints are available:

- `/api/v1/recommendations`: Generation of service recommendations.
- `/api/v1/opinions`: Adding expert opinions to the system

The REST interface looks as follows:

- `/api/v1/recommendations`:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Paged retrieval of a service recommendation for an existing service. Returns 200 if service could be found, 404 otherwise.</td>
<td><code>serviceHash</code>: service for which recommendations should be provided. The hash is identical for each service across systems in contrast to the service id.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>size</code> (optional): Size of a page</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>page</code> (optional): Page index</td>
</tr>
</tbody>
</table>

Access: Public, no authorization

Body
**Example:**

GET
http://141.84.213.235:8100/api/v1/recommendations?serviceHash=A529607000BE8C00D9359AD4ADE4C801&size=1

**RESULT:**

**RETURN CODE:** 200

```
{
    "content": [
        {
            "score": 1,
            "serviceHash": "348DFCFAC963F3268D4728CCC9A11B72"
        }
    ],
    "pageable": {
        "sort": {
            "unsorted": true,
            "sorted": false
        },
        "pageSize": 1,
        "pageNumber": 0,
        "offset": 0,
        "unpaged": false,
        "paged": true
    },
    "last": false,
    "totalPages": 3,
    "totalElements": 3,
    "sort": {
        "unsorted": true,
        "sorted": false
    },
    "first": true,
    "numberOfElements": 1,
    "size": 1,
    "number": 0
}```
Recommendations can also be retrieved using POST requests. If the trend recommender is used the request body must contain at least one of the fields age group, gender and location. For the location recommender location information and the desired distance radius must be provided.

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Paged retrieval of a service recommendation for an existing service. Returns 200 if service could be found, 404 otherwise.</td>
<td>size (optional): Size of a page page (optional): Page index</td>
</tr>
</tbody>
</table>

**Access**

Public, no authorization

**Body**

A valid recommender request. The request contains a list of recommenders to use together with a weight per recommender which is used for merging recommender results.

**Example:**

POST http://141.84.213.235:8100/api/v1/recommendations?size=1

BODY:

```json
{
  "serviceHash": "348DFCFAC963F3268D4728CCC9A11B72",
  "context": {
    "demographics": {
      "ageGroup": "SixteenToTwenty",
      "gender": "Diverse",
      "location": {
        "countryCode": "DE",
        "distance": 100,
        "geoPoint": {"lat": 48.142689, "lon": 11.553805},
        "timezone": "Mitteleuropäische Normalzeit"
      }
    }
  }
}
```
RESULT:

RETURN CODE: 200

BODY:

{
    "content": [{
        "score": 0.25,
        "serviceHash": "12623581A54338B80026419C0EB0CEBD"
    }],
    "pageable": {
        "sort": {
            "unsorted": true,
            "sorted": false
        },
        "pageSize": 1,
        "pageNumber": 0,
        "offset": 0,
        "unpaged": false,
        "paged": true
    },
    "last": false,
    "totalPages": 4,
    "totalElements": 4,
    "sort": {
        "unsorted": true,
        "sorted": false
    }
}
• /api/v1/opinions:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Adds a new expert opinion to the system. Returns code 200 if it succeeded, code 400 if not.</td>
<td></td>
</tr>
</tbody>
</table>

**Access**

Public, authorization

**Body**

A valid expert opinion definition.

**Example:**

POST https://141.84.213.235:8100/api/v1/opinions

BODY:

```json
{
    "recommendations": [
        {
            "serviceHash": "A529607000BE8C00D9359AD4ADE4C801"
        },
        {
            "serviceHash": "6527C73BB49F843A1B51F72118163839"
        },
        {
            "serviceHash": "0153368FD4070CC897B4809EC701E4D"
        }
    ]
}
```
RESULT:

RETURN CODE: 200

BODY:

```json
{
  "recommendations": [
    {
      "serviceHash": "A529607000BE8C00D9359AD4ADE4C801"
    },
    {
      "serviceHash": "6527C73BB49F843A1B51F72118163839"
    }
  ],
  "serviceHash": "0153368FD4070CC897B4809EC7015E4D"
}
```

The technical details of the service recommender can be configured in the file:

`recommendations_rest/build/main/resources/application.properties`

```properties
service.version=1.2.1
server.servlet.contextPath=/api/v1/
server.port=8100
server.ssl.key-store =./security/.keystore
server.ssl.key-store-password =hradio
server.ssl.key-password =hradio

## Authentication
password_manager.baseUrl=http://localhost:8111/api/v1
rest.requestTimeout=3000
rest.runSecure=true

## Logging configuration
logging.file=./log/recommendations_rest.log
logging.file.max-history=10
```
metadata.baseURL=http://localhost:8090/api/v1
metadata.requestTimeout=3000

search.baseURL=https://localhost:8080/api/v1
search.requestTimeout=3000

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

recommender.updateInterval=60
recommender.updateDelay=0
recommender.updateTimeUnit=MINUTES
recommender.expertOpinionFolder=./expert_opinions
recommender.fingerprintFile=./fingerprints/fingerprints.csv

2.5.5. Privacy Preserving User Data Collection

Interactive Surveys

The server for the interactive surveys can be operated via a web interface:

![Figure 15: Welcome page of the server’s UI.](image)

Surveys can be created using the button “Create Survey”: 
Surveys consist of an unlimited number of questions with freely selectable answer options. Depending on the selected question type (multiple choice (MC) or checkbox (CBX)), only one or more answer options can be specified by the clients at a time.

Each created survey can be activated, paused or set to state “done” over the menu “Manage Surveys”. Only if it is active it can be actively requested and answered by the clients:
The server can actively request all client responses (reports) for all active surveys via the button “New evaluation”. The results are visualized via bar charts for each question:

![Figure 17: The UI for changing a survey's state.](image1)

![Figure 18: Result visualization via bar charts.](image2)
In the following, the REST endpoints of the server are listed:

- `/listsurveys`

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Returns list of all active surveys in JSON format. Returns 200 if service could be found. Else 404.</td>
<td>None</td>
</tr>
</tbody>
</table>

**Access**

Public, no authorization

**Body**

```
```

**Example:**

GET http://localhost:5000/listsurveys

**RESULT:**

**RETURN CODE: 200**

**BODY:**

```
{
  "surveys": [
    {
      "surveyid": "radio_gaga_2019-09-16_140522",
      "serviceprovider": "radio_gaga",
      "surveyname": "General behavior",
      "status": "active",
      "created on": "16.09.2019 14:05:22",
      "sdescription": "General questions",
      "questions": [
        {
          "qid": 1,
```


```
{
  "name": "frequency",
  "type": "cbx",
  "description": "How often are you listening?",
  "options": [
    "Every day",
    "Once a week",
    "Once a month",
    "Never"
  ]
},
{
  "qid": 2,
  "name": "Location",
  "type": "mc",
  "description": "Where are you listening?",
  "options": [
    "At home",
    "At work",
    "In the car",
    "Other location"
  ]
}
```

- `/surveys/<string:surveyname>`

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Returns Survey with surveyname as name in JSON format and list of received reports.</td>
<td>None</td>
</tr>
</tbody>
</table>
Returns 200 if service could be found. Else 404.

| Access   | Public, no authorization |

Example:

GET `http://localhost:5000/surveys/General behavior`

RESULT:

RETURN CODE: 200

BODY:

```json
{
  "surveyid": "radio_gaga_2019-09-16_140522",
  "serviceprovider": "radio_gaga",
  "surveyname": "General behavior",
  "status": "active",
  "created on": "16.09.2019 14:05:22",
  "sdescription": "General questions about listening to the radio",
  "questions": [
    {
      "qid": 1,
      "name": "frequency",
      "type": "cbx",
      "description": "How often are you listening to the radio?",
      "options": [
        "Every day",
        "Once a week",
        "Once a month",
        "Never"
      ]
    }
  ]
}```
{  
  "qid": 2,  
  "name": "Location",  
  "type": "mc",  
  "description": "Where are you listening to the radio?",  
  "options": [  
    "At home",  
    "At work",  
    "In the car",  
    "Other location"
  ]
}  
],  
"reports": [
{  
  "surveyid": "radio_gaga_2019-09-16_140522",  
  "prr": true,  
  "irr": true,  
  "f": 0.5,  
  "p": 0.75,  
  "q": 0.5,  
  "answers": [
    {  
      "qid": 1,  
      "question": "frequency",  
      "options": [  
        1,  
        1,  
        1,  
        0 
      ],  
      "f": 0.5,  
      "p": 0.75,  
      "q": 0.5
    },  
    {  
      "qid": 2,  
      "question": "Location",  
      "options": [  
        1,  
        1,  
        1,  
        0 
      ],  
      "f": 0.5,  
      "p": 0.75,  
      "q": 0.5
    }
  ]
}
1,
0,
1
],
  "f": 0.5,
  "p": 0.75,
  "q": 0.5
}
]
]
]
]

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Creates new survey. Returns 201 if survey could be created.</td>
<td>None</td>
</tr>
</tbody>
</table>

**Access**

Public, authorization

**Body**

A valid survey definition.

**Example:**

POST http://localhost:5000/surveys/New Survey

BODY:

```
{
  "surveyid": "radio_gaga_2019-09-16_140522",
  "serviceprovider": "radio_gaga",
  "surveyname": "New Survey",
```
RESULT:

RETURN CODE: 201

BODY:

```json
{
  "surveyid": "radio_gaga_2019-09-16_163823",
  "serviceprovider": "radio_gaga",
  "surveyname": "New Survey",
  "status": "active",
  "created on": "16.09.2019 16:38:23",
  "sdescription": "General questions about listening to the radio",
  "questions": [
    {
      "qid": 1,
      "name": "frequency",
      "type": "cbx",
      "description": "How often are you listening to the radio?",
      "options": [
        "Every day",
        "Once a week",
        "Once a month",
        "Never"
      ]
    }
  ]
}
```
“Once a week”,
“Once a month”,
“Never”
]
]
]

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>Changes status of survey</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Returns 200 if status was changed.</td>
<td></td>
</tr>
</tbody>
</table>

**Access**
Public, authorization

**Body**

A valid survey definition.

**Example:**

PUT http://localhost:5000/surveys/New Survey

**BODY:**

```json
{
  "surveyid": "radio_gaga_2019-09-16_140522",
  "serviceprovider": "radio_gaga",
  "surveyname": "New Survey",
  "status": "paused",
  "created on": "16.09.2019 14:05:22",
  "sdescription": "General questions about listening to the radio",
  "questions": [
  ```
"qid":1,
"name":"frequency",
"type":"cbx",
"description":"How often are you listening to the radio?",
"options": [ 
    "Every day",
    "Once a week",
    "Once a month",
    "Never"
]
]
]
]

RESULT:

RETURN CODE: 200

BODY:

{ 
    "message":"Status of survey 'New Survey' changed from 'active' to 'paused' "
}

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>Deletes a survey.</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Returns 200 if survey could</td>
<td></td>
</tr>
<tr>
<td></td>
<td>be deleted</td>
<td></td>
</tr>
</tbody>
</table>

Access: Public, authorization

Body

- 

Example:
DELETE http://localhost:5000/surveys/New Survey

RESULT:

RETURN CODE: 202

BODY:

{  "message": "Survey with name 'Test' deleted." }

• /reports/<string:surveyid>

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Returns all Reports to the survey. Returns 200 if service could be found. Else 404.</td>
<td>None</td>
</tr>
</tbody>
</table>

Access Public, authorization

Body

Example:

GET http://localhost:5000/reports/radio_gaga_2019-09-16_140522

RESULT:

RETURN CODE: 200

BODY:

{  }
"reports": [
{
  "surveyid": "radio_gaga_2019-09-16_140522",
  "prr": true,
  "irr": true,
  "f": 0.5,
  "p": 0.75,
  "q": 0.5,
  "answers": [
    {
      "qid": 1,
      "question": "frequency",
      "options": [
        1,
        1,
        1,
        0
      ],
      "f": 0.5,
      "p": 0.75,
      "q": 0.5
    },
    {
      "qid": 2,
      "question": "Location",
      "options": [
        1,
        1,
        0,
        1
      ],
      "f": 0.5,
      "p": 0.75,
      "q": 0.5
    }
  ]
}]
}
• /smmrs/<string:surveyid>

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Returns the summaries for the survey. Returns 200 if service could be found. Else 404.</td>
<td>None</td>
</tr>
</tbody>
</table>

**Access**

Public, authorization

**Body**

- 

**Example:**

GET http://localhost:5000/smmrs/radio_gaga_2019-09-16_140522

**RESULT:**

RETURN CODE: 200

**BODY:**

```json
{
  "summaries": [
    {
      "surveyid": "radio_gaga_2019-09-16_140522",
      "name": "frequency",
      "type": "cbx",
      "answers": [
        {
          "option": "Every day",
          "frequency": 1
        },
        {
          "option": "Once a week",
          "frequency": 1
        },
        {
          "option": "Once a month",
```
DELETE

Deletes a summary. Returns 202 if summary could be deleted.

None

Access

Public, authorization

Body

Example:

DELETE  http://localhost:5000/smmrs/radio_gaga_2019-09-16_140522

RESULT:

RETURN CODE: 202

BODY:

{  
  "TEST API (no productive use)": "Summaries belonging to survey id 'radio_gaga_2019-09-16_140522' deleted."
}

• /smmrss/create/<string:surveyid>

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Retrieves a summary for the survey.</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Returns 200 if service could be found. Else 404.</td>
<td></td>
</tr>
</tbody>
</table>

Access

Public, authorization

Body
Example:

GET http://localhost:5000/smmrs/create/radio_gaga_2019-09-16_140522

RESULT:

RETURN CODE: 200

BODY:

```
{
  "summaries": [
  {
    "surveyid": "radio_gaga_2019-09-16_140522",
    "name": "frequency",
    "type": "cbx",
    "answers": [
    {
      "option": "Every day",
      "frequency": 1
    },
    {
      "option": "Once a week",
      "frequency": 1
    },
    {
      "option": "Once a month",
      "frequency": 1
    },
    {
      "option": "Never",
      "frequency": 0
    }
  ],
  "created": "16.09.2019 17:04:25",
  "counter": 1
  },
  {
    "surveyid": "radio_gaga_2019-09-16_140522",
    "name": "listening",
    "type": "srl",
    "answers": [
    {
      "option": "Every day",
      "listening": 1
    },
    {
      "option": "Once a week",
      "listening": 1
    },
    {
      "option": "Once a month",
      "listening": 1
    },
    {
      "option": "Never",
      "listening": 0
    }
  ],
  "created": "16.09.2019 17:04:25",
  "counter": 1
```
The client is implemented as an Android library which can be used in smartphone apps (the platform-centric app implements surveys using this library).

The API has the following methods and structures:

```java
class Survey {
    String surveyid; // unique id of every survey
    String serviceprovider; // name of the serviceprovider
    List<Question> questions; // List of Questions
}
```

The client is implemented as an Android library which can be used in smartphone apps (the platform-centric app implements surveys using this library).

The API has the following methods and structures:

```java
class Survey {
    String surveyid; // unique id of every survey
    String serviceprovider; // name of the serviceprovider
    List<Question> questions; // List of Questions
}
```
### class `Question`:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Integer</td>
<td>unique id of question inside one survey</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>name of the question</td>
</tr>
<tr>
<td>type</td>
<td>QuestionType</td>
<td>any of the enum QuestionType containing 'bool', 'mc' and 'cbx'</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>description of the question/the question itself</td>
</tr>
<tr>
<td>options</td>
<td>String[]</td>
<td>answer options to the asked question</td>
</tr>
<tr>
<td>responded</td>
<td>boolean</td>
<td>represents if the question has been answered</td>
</tr>
</tbody>
</table>

### enum `QuestionType`:

<table>
<thead>
<tr>
<th>question type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>The question has only two answer options where only one can be chosen</td>
</tr>
<tr>
<td>mc</td>
<td>only one of many answer options can be chosen</td>
</tr>
<tr>
<td>cbx</td>
<td>any number of answer options can be chosen</td>
</tr>
</tbody>
</table>

### class `Report`:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>surveyid</td>
<td>String</td>
<td>unique id of the survey answered</td>
</tr>
</tbody>
</table>
### Privacy Parameters

<table>
<thead>
<tr>
<th>Role</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>priv</td>
<td>PrivacyParameters</td>
<td>default privacy parameters for all questions</td>
</tr>
<tr>
<td>answers</td>
<td>Answer[]</td>
<td>List of Answers to the Questions</td>
</tr>
</tbody>
</table>

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>buildReport(survey, privacyParameters)</td>
<td>Returns a report object responding to the given survey with fitting survey id, an Answer object for every Question object and the given privacy parameters as default</td>
<td>survey (Survey): survey object privacyParameters (PrivacyParameters): object containing a value for f, p and q for randomization</td>
</tr>
</tbody>
</table>

### Class `Answer`

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qid</td>
<td>Integer</td>
<td>unique id of question inside one survey</td>
</tr>
<tr>
<td>question_name</td>
<td>String</td>
<td>name of the question</td>
</tr>
<tr>
<td>answer</td>
<td>Integer[]</td>
<td>array of 0 and 1 with 1 where the answer options was right</td>
</tr>
<tr>
<td>priv</td>
<td>PrivacyParameters</td>
<td>individual privacy parameters for this question</td>
</tr>
</tbody>
</table>

### Class `PrivacyParameters`

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>double</td>
<td>parameter used for the permanent randomized response PRR</td>
</tr>
<tr>
<td>p</td>
<td>double</td>
<td>parameter used for the IRR</td>
</tr>
</tbody>
</table>
q: double parameter used for the IRR

class API Calls:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getSurveys(url, requestQueue, callback, errorListener)</code></td>
<td>Requests all active surveys from given REST-endpoint; The resulting surveys can be accessed via the given callback and method <code>onTaskComplete()</code></td>
<td><code>url (string): url to REST-endpoint</code>&lt;br&gt;<code>requestQueue(RequestQueue): Volley RequestQueue for API calls</code>&lt;br&gt;<code>callback (RestTaskCallback): callback to wait for asynchronous REST request</code>&lt;br&gt;<code>errorListener (Response.ErrorListener): ErrorListener Object to react to error responses</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sendReport(report, url, requestQueue, callback)</code></td>
<td>Randomizes the answers in the given Report object as PRRs. Then it formats the object into JSON and posts them at the given url REST-endpoint</td>
<td><code>report (Report): Report object that should be send to the REST endpoint</code>&lt;br&gt;<code>url (string): url to REST-endpoint</code>&lt;br&gt;<code>requestQueue(RequestQueue): Volley RequestQueue for API calls</code>&lt;br&gt;<code>callback (RestTaskCallback): callback to wait for asynchronous REST request</code></td>
</tr>
</tbody>
</table>

Statistics

The following REST end points exist:

- `/api/v1/service_use`: Add or request for service use reports (context and a list of services the user has listened to).
- `/api/v1/user_reports`: Add user reports (context and any survey answer).
The REST interface looks like this:

- `/api/v1/service_use`:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
</table>
| GET    | Returns all service uses in a specific time frame | **startTime**: All service uses since start time (format: "MMM dd, yyyy HH:mm:ss").  
**stopTime**: All service uses until stop time (format: "MMM dd, yyyy HH:mm:ss"). |

**Access**: Public, no authorization

**Body**

```
-
```

**Example:**


RESULT:

RETURN CODE: 200

BODY:

```
{
  "content": [
    {
      "id": 59087,
      "context": {
        "demographics": {
          "location": {
            "timezone": "",
            "geoPoint": {
              "lat": 0,
              <!-- Partially cut off -->
```
"lon":1,
"countryCode":"de",
"distance":1,
"ageGroup":"EightToFifteen",
"gender":"Female",
"time":"Jul 24, 2019 00:00:02",
"services":["s1", "s2", "s3"]}
]
}
"pageable":{
"sort":{
"sorted":false,
"unsorted":true
},
"pageSize":20,
"pageNumber":0,
"offset":0,
"unpaged":false,
"paged":true
},
"last":true,
"totalPages":1,
"totalElements":1,
"sort":{
"sorted":false,
"unsorted":true
},
"first":true,
"numberOfElements":1,
"size":20,
"number":0}
<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Sends a new service use report to the server.</td>
<td></td>
</tr>
</tbody>
</table>

**Access**

Public, no authorization

**Body**

A valid service use definition.

**Example:**

POST http://141.84.213.235:8110/api/v1/service_use

**BODY:**

```json
{
   "context": {
      "demographics": {
         "location": {
            "timezone": "",
            "countryCode": "",
            "geoPoint": {"lat":0, "lon":1}
         },
         "ageGroup": "EightToFifteen",
         "gender": "Female"
      },
      "time": "Jul 24, 2019 00:00:02"
   },
   "services": ["s1hash", "s2hash", "s3hash"]
}
```

**RESULT:**

RETURN CODE: 200
BODY:
{
   "id": 59087,
   "context": {
      "demographics": {
         "location": {
            "timezone": "",
            "geoPoint": {
               "lat": 0,
               "lon": 1
            },
            "countryCode": "",
            "distance": null
         },
         "ageGroup": "EightToFifteen",
         "gender": "Female"
      },
      "time": "Jul 24, 2019 00:00:02"
   },
   "services": [
      "s1hash",
      "s2hash",
      "s3hash"
   ]
}

• `/api/v1/user_reports`:

A user report consists of the following fields:

class `Report`:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context</td>
<td>Context</td>
<td>Context information (demographics, ...)</td>
</tr>
<tr>
<td>reportId</td>
<td>String</td>
<td>Id of the report this instance belongs to.</td>
</tr>
</tbody>
</table>
### Description

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>String</td>
<td>Description of the report this instance belongs to.</td>
</tr>
<tr>
<td>values</td>
<td>String Array</td>
<td>Array of report values. Each value is one of the elements of the labels array.</td>
</tr>
<tr>
<td>labels</td>
<td>String Array</td>
<td>Possible labels for the values array.</td>
</tr>
</tbody>
</table>

### Context:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>demographics</td>
<td>Demographics</td>
<td>Demographic information (e.g. age group)</td>
</tr>
<tr>
<td>time</td>
<td>Time</td>
<td>Current time in the format: MMM dd, yyyy HH:mm:ss</td>
</tr>
</tbody>
</table>

#### Class Demographics:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| ageGroup      | Enumeration | Age group enumeration. Possible values:  
- YoungerThanEight  
- EightToFifteen  
- SixteenToTwenty  
- TwentyOneToTwentyNine  
- ThirtyToForty  
- FortyOneToFifty  
- FiftyOneToSixty  
- SixtyOneAndAbove  
- NotSpecified |
| location      | Location | Location information (timezone, ...) |
| gender        | Enumeration | Possible values:  
- Diverse  
- Female  
- Male |
### Class Location:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>countryCode</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>timezone</td>
<td>String</td>
<td>Time zone. E.g.: GMT+00:00</td>
</tr>
<tr>
<td>distance</td>
<td>Integer</td>
<td>Distance (not used in this case)</td>
</tr>
<tr>
<td>geoPoint</td>
<td>GeoPoint</td>
<td>GPS coordinates</td>
</tr>
</tbody>
</table>

### Class GeoPoint:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lat</td>
<td>Double</td>
<td>Latitude</td>
</tr>
<tr>
<td>lon</td>
<td>Double</td>
<td>Longitude</td>
</tr>
</tbody>
</table>

### Verbs:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Sends a user report to the server.</td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>Public, no authorization</td>
<td></td>
</tr>
<tr>
<td>BODY</td>
<td>A valid user report definition.</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

POST http://141.84.213.235:8110/api/v1/user_reports

**BODY:**
{  
"context":{  
  "demographics":{  
    "location":{  
      "timezone":"GMT+00:00",  
      "geoPoint":{  
        "lat":11.59428311,  
        "lon":48.1500233  
      },  
      "countryCode":"de",  
      "distance":0  
    },  
    "ageGroup":"EightToFifteen",  
    "gender":"Female"  
  },  
  "time":"Jul 24, 2019 13:45:14"  
},  
"reportId":"report1",  
"description":"report one",  
"values": ["yes"],  
"labels": ["yes","no","maybe"]  
}

RESULT:

RETURN CODE: 200

BODY:

{
  "reportId":"report1",  
  "id":"261889134",  
  "description":"report one",  
  "context":{  
    "demographics":{  
      "location":{  
        "timezone":"GMT+00:00",  
        "geoPoint":{  
          "lat":11.59428311,  
          "lon":48.1500233  
        },  
        "countryCode":"de",  
        "distance":0  
      },  
      "ageGroup":"EightToFifteen",  
      "gender":"Female"  
    },  
    "time":"Jul 24, 2019 13:45:14"  
}
User reports are stored in the document storage and thus can be visualized using the Dashboard component.

The statistics REST service can be configured by editing the application.properties file located in statistics_rest/build/resources/main/:

```
# service version configuration
service.version=1.2.1

# server configuration
server.servlet.contextPath=/api/v1/
server.port=8110
server.sslactivate=true
server.ssl.key-store=./security/.keystore
server.ssl.key-store-password=hradio
server.ssl.key-password=hradio

# authentication
## Authentication
password_manager.baseUrl=http://passwordmanager:8111/api/v1
rest.runSecure=true
rest.requestTimeout=3000
```
### Logging configuration
logging.file=./log/statistics_rest.log
logging.file.max-history=10

### Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.url=jdbc:postgresql://db:5432/hradio
spring.datasource.username=postgres
spring.datasource.password=postgres

# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect

# Charset of HTTP requests and responses. Added to the “Content-Type” header if not set explicitly.
# spring.http.encoding.charset= UTF-8
# Enable http encoding support.
# spring.http.encoding.enabled=true
# Force the encoding to the configured charset on HTTP requests and responses.
# spring.http.encoding.force=true

# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update

2.5.6. Dashboard

The Dashboard module can be accessed via web browser. It is possible to download visualizations and dashboards for later use. The dashboard module uses Kibana version 6.2 as its UI and dashboard engine. See [https://www.elastic.co/guide/index.html](https://www.elastic.co/guide/index.html) for a detailed documentation and learning source.

2.5.7. Password Manager

The password manager REST service was developed to restrict unauthorized requests to safety-relevant REST endpoints. The following endpoints are available:
Please note that all endpoints of the password manager are only available in the private company network. Via the first endpoint, “/api/v1/generateUserData”, user data, i.e. user name and password combinations, can be generated. Therefore, only the desired username must be given as a parameter. An associated password is automatically generated by the password manager REST service which is then returned in the response if the operation was successful.

The second endpoint, “/api/v1/isValid”, allows GET-requests containing the parameters “username” and “password” and validates the combination. The response always has the status code 200 and the content “OK” or “INVALID”.

With the third endpoint, “/api/v1/userdata”, it is possible to get all user data entries (GET) or to delete a certain user data entry (DELETE, with parameter “username” specifying the name of the user to delete).

The password manager REST service can be configured by editing the application.properties file located in password_manager/build/resources/main:

```java
service.version=1.2.1
server.servlet.contextPath=/api/v1/
server.port=8111

### Logging configuration
logging.file=./log/password_manager.log
logging.file.max-size=10KB
logging.file.max-history=1

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

authentication.file=./authentication/userdata.csv
```
2.6. SYSTEM REQUIREMENTS

The platform was tested on Ubuntu Linux 17.10. But all Unix-based operating systems supported by Docker should work. 4GB of RAM as a minimum and a dual core processor are required (recommended: 8GB, quad core processor) but both, required CPU and RAM, heavily depend on the usage. The same goes for the required disk space.

2.7. DEPLOYMENT

The platform’s architecture foresees simple deployment independent of operating system and hardware. In order to support that, we’ve used Docker to partition the whole platform into isolated containers. Containers are executed, maintained and orchestrated using the tool Docker Compose. The following diagram depicts the currently used container partitioning:

![Diagram of container partitioning]

Figure 19: Communication Platform containerization. Please note that the survey REST service is not part of the diagram since it is deployed separately.

Docker can be described as a light-weight virtualization of resources provided by the operating system. Instead of having a hypervisor that manages multiple guest operating systems on a host operating system, Docker containers are executed in
isolation by a so-called Docker engine, that directly communicates with the host. Thus, not each container needs its own kernel instance. Instead, containers share the host operating system, but still run in isolation. Different containers might share binaries and libraries but don’t have to. See the following figure for explanation:

Figure 20: A virtual machine runs a full operating system as guest on a host operating system. Docker containers run directly on the so-called Docker Engine and share the host operating system. Source:\(^\text{14}\)

\(^{14}\) http://www.falnic.net/news/hpe-docker-pledge-support
3. HOW TOS

This chapter contains a set of How To guides for practical work with the platform.

3.1. HOW TO: TEST THE PLATFORM

If the platform should be tested without any installation or building, it is possible to access a test system at 141.84.213.235. Examples:

- **Metadata REST:** [http://141.84.213.235:8090/api/v1/providers/1/services/2](http://141.84.213.235:8090/api/v1/providers/1/services/2)

- **Search REST:** [http://141.84.213.235:8080/api/v1/services?q=name:B*](http://141.84.213.235:8080/api/v1/services?q=name:B*)
The search REST api can be tested via the search portal which can be accessed here: [http://141.84.213.235/search_portal](http://141.84.213.235/search_portal). In the search field the query data for the service request is entered (e.g. name:B*).

- **Recommendations REST:** [http://141.84.213.235:8100/api/v1/recommendations?serviceHash=A529607000BE8C00D9359AD4ADE4C801&size=100](http://141.84.213.235:8100/api/v1/recommendations?serviceHash=A529607000BE8C00D9359AD4ADE4C801&size=100)
A list of recommendations for the requested service (here “BAYERN 1 Franken”) will be shown. For GET requests the scores of all available recommenders are weighted equally.

Please note that response times (especially for the first request) might be high due to the hardware configuration of the test system.

3.2. HOW TO: DEPLOY THE PLATFORM USING DEPLOYMENT SCRIPT

This guide describes how to deploy the platform using a deployment bash script.

**Preliminaries:**

- Download the deployment bash script deploy_hradio_platform.sh from [https://gitlab.irt.de/hradio/platforms/CommunicationPlatform/deployment_maintenance/tree/master/src](https://gitlab.irt.de/hradio/platforms/CommunicationPlatform/deployment_maintenance/tree/master/src)
- Provide a virtual machine (VM) for the installation of the HRADIO platform. The operating systems **Ubuntu** (preferred) and **Debian** (at least version **stretch**) are supported. The VM might be set up locally via VirtualBox:
  - **Install VirtualBox:**
    - Install VirtualBox on your device according to your operating system. Please follow the instructions on [https://www.virtualbox.org/](https://www.virtualbox.org/).
    - The VM disk image should at least have 100GB.
    - The VM disk image should at least have 4GB RAM.
    - In the **Network Settings** choose **Bridged Adapter**
    - Install VirtualBoxGuest Additions.
  - **Install Ubuntu:**
    - Download Ubuntu Server. You can download it e.g. from [https://wiki.ubuntuusers.de/Startseite/](https://wiki.ubuntuusers.de/Startseite/)
    - Follow a tutorial and install Ubuntu Server as a VM in VirtualBox. [https://www.smarthomebeginner.com/install-ubuntu-server-on-virtualbox/](https://www.smarthomebeginner.com/install-ubuntu-server-on-virtualbox/)
    - Ubuntu installation steps to follow:
      - Language: **English**
      - Country: other -> **Germany**
      - locale: **United States** - en_US.UTF-8
      - Detect keyboard layout: Yes ...
      - Hostname: **hradio**
      - Full name for the new user: **hradio**
      - Username for your account: **hradio**
      - Password: **hradio**
      - Timezone: **Europe/Berlin**
      - Partitioning method: **Guided – use entire disk and set up LVM**
      - Select disk to partition: **sda**
      - Write the changes to disks and configure LVM? Yes
      - Amount … Continue
      - Write the changes to disks? Yes
      - HTTP proxy information (blank for none): **Continue**
      - How do you want to manage upgrades on this system? **Install security updates automatically**
      - Choose software to install: OpenSSH server -> **Continue**
      - Install the GRUB boot loader to the master boot record? Yes
      - Finish: **Continue**

**Please note:** The chosen values for language, country, locale, keyboard, user name, password, etc. are not critical for a
smooth operation of the HRADIO platform and can be adjusted arbitrarily.

- Run downloaded deployment script on VM with the command 
  
  `/deploy_hradio_platform.sh`

  The script will guide you through the deployment of the HRADIO platform. All information asked throughout the deployment process is only used for the installation of the HRADIO platform and will not be used for any other purpose. The values are saved in “$HOME/hradio_env”. The script asks for the following information:
  
  - The systems root password
  - E-mail address for SSH key generation
  - Deployment path for the HRADIO platform (default: $HOME/hradio)
  - Username for IRT Gitlab access
  - System distribution and codename (in case the `lsb_release -a` command is unknown)
  - If TLS access should be activated (Y/N). If so, all public interfaces of the HRADIO platform will only be accessible via HTTPS connection.
  - If the system should be running in security mode (Y/N). If so, all safety-relevant actions require username and password (administered by the PasswordManager REST service).

### 3.3. HOW TO: INSTALL THE PLATFORM USING INSTALLATION SCRIPT

This guide shows how to install the platform using installation script.

**Preliminaries:**

- The previously deployed bash script `$DPL_PATH/install_hradio.sh`

**Steps:**

The installation script will perform the following steps:

- Create a backup of the file `/etc/profile` and `$HOME/.bashrc`
- Install packages (see `$CONFIG_PATH/pkglist_*.txt`)
- Setup Docker environment
  - Install and configure Docker Engine
  - Install and configure Docker Compose
- Setup Java (version 1.8)
3.4. HOW TO: BUILD THE PLATFORM USING BUILD SCRIPT

This guide shows how to build the platform using the build script.

Preliminaries:
- The previously deployed bash script `$DPL_PATH/build_hradio_platform.sh`

Steps:
- The build script for the HRADIO platform builds the following Docker containers:
  - `search_and_storage`
  - `import_scheduler`: Builds one Docker container which hosts all importer and crawler processes.

3.5. HOW TO: RUN THE PLATFORM USING PRE-BUILD ARTEFACTS

This guide shows how to run the platform using pre-built artefacts.

Preliminaries:
- The previously deployed bash scripts
  - `$DPL_PATH/start_hradio_platform.sh`
  - `$DPL_PATH/start_search_and_storage.sh`
  - `$DPL_PATH/start_crawler_and_importer.sh`

Steps:
The main script `start_hradio_platform.sh` starts a TMUX session containing 4 panes:
- `search_and_storage`: This pane holds the search and storage Docker stack.
- `crawler & importer`: This pane holds all crawler and importer processes.
- 2 control panes: These panes can be used by the user for controlling purposes. One of them holds an `htop` instance to give an overview of running processes and system resources.
4. FUTURE DEVELOPMENTS

Candidate topics to improve the HRADIO Communication Platform in the future include:

➡️ **Password Manager:**

Currently, user authorization does not have the notion of different roles within the platform. In future, it might make sense to implement a role system with different access rights. E.g., only a user having the administrator role is allowed to remove other users from the user database.

➡️ **Privacy-Preserving User Data Collection:**

Sending user reports and service uses to the statistics system currently does not include a user authentication or authorization mechanism since user data should be collected without the need of storing user-specific information that could be used to identify single users. Thus, it is currently not possible to assure that user data is valid beyond already available, server-side sanity checks and stems from valid users. For future work, a more sophisticated detection of fake-user data which in the same time prevents single users from being identified would improve its data quality significantly.
Metadata Search & Storage:

A web-based maintenance and administration UI would improve the usability of the platform. It could simplify the maintenance of the search federation (updating the node description and search index, generation of tokens) and increase efficiency when manual metadata changes are necessary.

More sophisticated search index maintenance algorithms could simplify search index maintenance.

Additional crawlers could use the extendable importer platform to add new metadata sources.

Recommendations:

Since the service recommendation engine is highly modular and flexible, it should be easy to integrate non-content-based recommender systems that are based on collaborative filtering. Currently, the only recommender that uses data from users is the trend recommender.

In addition, our trend recommender can be extended with more context information about the user who is requesting recommendations. For example, the user asks for recommendations using a voice interface. From the audio signal of the user’s voice, her level of sleepiness is detected which is one parameter of the user’s context that affects the generation of recommendations. For detecting sleepiness in voice using a specialized deep neural network, we’ve conducted research [3] which could be used as starting point.

Global Logging and Backup Solution:

The distributed and modular architecture of the HRADIO Communication platform allows for the simple integration of new components. However, each component might have its own persistency and logging mechanisms. Having a global umbrella structure for maintaining backups and logging output of all sub systems would simplify platform maintenance.

Search Federation:

Currently, the search index of each search node is maintained manually. In future, a rule-based index maintenance engine could help keeping the index clean. It could, for example, remove search nodes from the index that are not reachable anymore.
5. REFERENCES


