D3.2: HRADIO Communication Platform

Editor: Markus Friedrich, André Ebert (LMU)

This deliverable includes the software documentation of the first iteration of the HRADIO Communication Platform. In addition, it contains a series of step-by-step guides for the reader to use the software demonstrator which is also part of this deliverable.
Basic Information

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Work package</td>
<td>3</td>
</tr>
<tr>
<td>Due date</td>
<td>31/06/2018</td>
</tr>
<tr>
<td>Submission date</td>
<td>28/06/2018</td>
</tr>
<tr>
<td>Deliverable lead</td>
<td>LMU</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Authors</td>
<td>Markus Friedrich, André Ebert (LMU)</td>
</tr>
<tr>
<td>Reviewers</td>
<td>François Daoust (W3C), Leo Andrews (Radio Player), Simon Delaere (imec)</td>
</tr>
</tbody>
</table>

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>05/06/2018</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>
</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>Project co-funded by the European Commission in the H2020 Programme</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nature of the deliverable:</strong></td>
</tr>
<tr>
<td><strong>Dissemination Level</strong></td>
</tr>
<tr>
<td><strong>PU</strong></td>
</tr>
<tr>
<td><strong>CL</strong></td>
</tr>
<tr>
<td><strong>CO</strong></td>
</tr>
</tbody>
</table>
EXECUTIVE SUMMARY

This document describes the architecture and implementation of the first iteration of the HRADIO Communication Platform. Main focus of this iteration is the integration and dissemination of radio related metadata. Thereby, it forms the foundation for later developments within the area of distributed metadata search and storage, service and programme recommendations as well as privacy preserving user data collection.

The provided architecture was built from the ground up by using industry proven technologies and allows complex metadata queries as well as simple integration of external metadata sources. An internal bus system provides module separation, easy extensibility and scalability.

The developed data model for radio related metadata is based on the RadioDNS standard and covers the use cases defined by the HRADIO consortium during user scenario selection (WP2). If use cases change over time, it is possible to extend or change the data model without data loss due to the flexible nature of the underlying storage engine.

External service interfaces for metadata maintenance and search functionalities are introduced and explained by example. Service interfaces were designed based on the REST paradigm with simplicity and efficiency in mind.

The HRADIO Communication Platform uses Docker container technology for module separation and easy deployment. An extensive guide was conceived that explains the necessary deployment steps in detail.

This deliverable also contains a demonstrator of the developed platform. The last part of this document contains how to guides that help the reader to explore and test it.
# TABLE OF CONTENTS

1. INTRODUCTION .................................................................................................................. 9
2. THE HRADIO COMMUNICATION PLATFORM ................................................................. 10
   2.1. Requirements .................................................................................................................. 10
   2.2. Architecture .................................................................................................................. 12
   2.3. Data Model .................................................................................................................... 15
   2.4. Components .................................................................................................................. 18
       2.4.1. Message Bus ........................................................................................................... 18
       2.4.2. Search & Storage Engine ....................................................................................... 19
       2.4.3. Search Processor .................................................................................................... 20
       2.4.4. Metadata Processor ............................................................................................... 22
       2.4.5. Search REST Service ............................................................................................ 23
       2.4.6. Metadata REST Service ........................................................................................ 25
       2.4.7. Metadata Importer ................................................................................................. 34
   2.5. System Requirements .................................................................................................... 38
   2.6. Deployment ................................................................................................................... 38
3. HOW TOS ............................................................................................................................... 40
   3.1. How To: Test the Platform ............................................................................................ 40
   3.2. How To: Run the Platform using pre-built Artefacts ....................................................... 40
   3.3. How To: Check out Sources, build and run the Platform ............................................... 41
   3.4. How To: Implement your own Importer ......................................................................... 41
4. CONCLUSION AND OUTLOOK ........................................................................................... 44
LIST OF FIGURES

FIGURE 1: THE ARCHITECTURE OF THE HRADIO COMMUNICATION PLATFORM (BLUE) WITH ALL COMPONENTS (WHITE). .......................................................... 13
FIGURE 2: THE FIRST ITERATION OF THE DESIGNED METADATA DATA MODEL .................. 16
FIGURE 3: COMMUNICATION SETTING WITH A PUBLISHER P, AN EXCHANGE ENTITY X AND TWO CONNECTED QUEUES. ........................................................................ 19
FIGURE 4: COMMUNICATION SETTING WITH A PUBLISHER, A CONNECTED QUEUE AND TWO CONSUMERS. ........................................................................ 19
FIGURE 5: TYPICAL ELASTICSEARCH ARCHITECTURE WITH DATA SOURCES, PROCESSING, STORAGE AND SEARCH. ........................................................................ 20
FIGURE 6: REQUEST/RESPONSE PATTERN AS IMPLEMENTED IN RABBITMQ. SOURCE: [4]...37
FIGURE 7: COMMUNICATION PLATFORM CONTAINERIZATION. ........................................... 38
FIGURE 8: A VIRTUAL MACHINE RUNS A FULL OPERATING SYSTEM AS QUEST ON A HOST OPERATING SYSTEM. ........................................................................ 39
FIGURE 9: THE FEDERATED SEARCH MODULE. .................................................................... 44
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE 1</td>
<td>PLATFORM REQUIREMENTS.</td>
<td>10</td>
</tr>
<tr>
<td>TABLE 2</td>
<td>DESCRIPTION OF THE RESTFUL INTERFACE OF THE SEARCH SERVICE.</td>
<td>23</td>
</tr>
<tr>
<td>TABLE 3</td>
<td>DESCRIPTION OF THE RESTFUL INTERFACE OF THE METADATA SERVICE.</td>
<td>26</td>
</tr>
<tr>
<td>TABLE 4</td>
<td>EXAMPLE BODIES FOR ALL METADATA TYPES.</td>
<td>32</td>
</tr>
<tr>
<td>TABLE 5</td>
<td>IMPORTER INTERFACE MESSAGES.</td>
<td>37</td>
</tr>
</tbody>
</table>
## ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</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>STOMP</td>
<td>Simple (or Streaming) Text Oriented Messaging Protocol</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 with 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 first iteration of the HRADIO Communication Platform. The focus of this iteration is the integration and dissemination of radio related metadata. It forms the foundation for later developments in the area of distributed metadata search and storage, service and programme recommendations, as well as privacy preserving user data collection.

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 data model and an in detail view of the different components, 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. This is especially helpful since this deliverable contains a demonstrator of the developed platform that can be tested and evaluated by the reader.

The third and last chapter gives an outlook on future developments.

---

1. This is the reason why the platform is also referred to as the Metadata Integration & Dissemination Platform throughout this document.
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 meetings with project partners (October 11th 2017 at IRT, January 17th /18th 2018 at Konsole Labs, April 10th 2018 at IRT) and from the HRADIO system architecture document (D3.1).

Table 1: Platform requirements.

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>The system should provide a publicly available interface for radio related metadata search.</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>
</tr>
<tr>
<td></td>
<td>The system must be able to process search requests targeting the recommendation engine requirements, proposed within the overall HRADIO system architecture document (D3.1), D2.3, Derived Technical Requirements (see Recommendation Engine)</td>
</tr>
<tr>
<td>Storage</td>
<td>The system should store metadata persistently in a human readable format.</td>
</tr>
<tr>
<td></td>
<td>The system’s storage engine should be able to scale with the amount of stored metadata.</td>
</tr>
</tbody>
</table>
### Data Model

- The system’s storage engine should be exchangeable without major refactoring.
- The system’s data model should cover the use cases defined in the second work package (information about stations, station services, schedules and programmes).
- The system’s data model should have a schema (needed for type-based queries, like date/time range queries).
- The system’s data model should be extensible.
- The system’s data model should be based on existing standards (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 (D3.1), D2.3, Derived Technical Requirements (see DAB-DL+, DAB-SPI, DAB-SLS, RadioTAG, and RadioWEB).

### Import

- The system should provide interfaces to maintain and retrieve metadata.
- The system should provide interfaces for metadata importer extension modules.
- The system should be able to import RadioDNS metadata from a configurable set of sources.
- The system should be able to import JSON-formatted Dynamic Label+ data provided by a STOMP server.
- 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 (D3.1), D2.3, Derived Technical Requirements: DAB-DL+, DAB-SPI, DAB-SLS, RadioTAG, and RadioWEB).
Deployment

The system should provide a standardized way for deployment. Deployment should be as simple as possible.

## 2.2. ARCHITECTURE

This section will provide an overview of the system architecture and the necessary 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 these 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.

This document describes the architecture of a single search and storage node since this is the main requirement for the first iteration of deliverable D3.2. The following diagram shows the system with all its components:
Figure 1: The architecture of the HRADIO Communication Platform (blue) with all components (white).
The metadata REST service is accessible from within the company network while the search REST service is publicly available.

The system consists of the following core components:

- **Message Bus**: The message bus connects all system components and allows for easy extensibility and scalability. Within the current state of implementation, RabbitMQ is used as a message broker, which is based on the publish/subscribe communication pattern. RabbitMQ is discussed in detail in Chapter 2.4.1.

- **Search Processor**: The search processor processes search requests coming from the search REST service, converts them to a search & storage engine specific format and sends them to the search & storage engine. The results are converted to the internal response format and sent back to the search requester (e.g. the search REST service).

- **Search & Storage Engine**: The search & storage engine processes search queries and stores the metadata as JSON documents. We currently use Elasticsearch for this purpose. The design allows for other search & storage engines as well.

- **Metadata Processor**: The metadata processor processes metadata maintenance requests from the metadata REST service (or from importers), converts them to a search & storage engine specific format and sends them to the search & storage engine.

- **Search REST Service**: The search service can be used for service, programme and schedule search. It follows RESTful service principles and uses JSON as transport format. The service is available publicly.

- **Metadata REST Service**: The metadata service can be used for metadata maintenance and retrieval. It follows RESTful service principles and uses JSON as transport format. The service is available internally.

- **Metadata Importer**: The flexible architecture allows for different data integration components. Currently, an importer for RadioDNS data and for Dynamic Label+ data (coming from a DAB Tuner, sent over a STOMP broker and converted to JSON) are designed and implemented. Both use the message bus for data transport to the metadata processor.
2.3. DATA MODEL

The data model describing radio related metadata is a subset of the RadioDNS standard that meets the requirements defined by the chosen user scenarios. It was designed for this first iteration but can easily be extended in future iterations. An implementation in Java exists (eu.hradio.metadata package). For the used JSON document storage & search engine (Elasticsearch), a schema (type mappings to JSON properties) for all metadata elements was implemented as well.

The following diagram depicts the elements of the metadata data model:
Figure 2: The first iteration of the designed metadata data model.

The parts in grey are not yet implemented in the current iteration of the metadata platform.

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: Bayerische 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). Genres and bearers are as well associated to a service.

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

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

Genre: Defines genres for services and programmes. A genre has a name stored as string.

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

Programme: Defines programmes with description, start and end time. Programmes are part of a schedule. A programme contains a link to its schedule (scheduleId) and a link to its service (serviceId).

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 end time and multiple content fields that describe the content of the event (type and value).

Content: Programme part content. Consists of a certain type (type) and the content itself (value).

Programme Group: Programmes and programme parts can be grouped together. Programme groups have a certain type (series, show, program concept, magazine, topic, program compilation, other collection, other choice, see RadioDNS standard). Groups can contain child groups. This element is not implemented in the current iteration of the platform since none of the use cases require it.

Service Group: Services can be grouped together. Groups can contain child groups. This element is not implemented in the current iteration of the platform since none of the use cases require it.
The metadata model can be easily extended. What has to be changed is the Java data model in the eu.hradio.metadata package and its corresponding Elasticsearch mapping definitions (as implemented as well in the eu.hradio.metadata package).

2.4. COMPONENTS

In the following, the system’s components are described in detail.

2.4.1. Message Bus

The message bus connects all system components and allows easy extensibility and scalability. We use RabbitMQ as a message broker which is based on the publish/subscribe communication pattern \[8\]. The main advantages of RabbitMQ are the following:

- **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):

![Diagram](image)

**Figure 3:** Communication setting with a publisher P, an exchange entity X and two connected queues.

Source: [1]

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

![Diagram](image)

**Figure 4:** Communication setting with a publisher, a connected queue and two consumers.

Source: [2]

### 2.4.2. Search & Storage Engine

The search and storage engine stores metadata and allows queries onto it. We use the Elasticsearch Engine for that purpose. 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:

- Full text queries (match a particular phrase, match a prefix, match multiple phrases, ...).
- Term level queries (range queries, regular expression based queries, wildcard queries, queries based on fuzzy logic, ...).
More complex queries based on document similarity measures.

Thus, Elasticsearch fulfills all our requirements regarding metadata search and storage. For more information on queries, see an explanation of Elastic’s Query DSL here: 46[7].

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

![Typical Elasticsearch architecture](image)

**Figure 5**: Typical Elasticsearch architecture with data sources, processing, storage and search.

Source: [3]

Data sources (in our case: importers) send their data to the data processing unit (in our case: the metadata processor). 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.

### 2.4.3. Search Processor

The search processor processes search requests coming from the search REST service (or other search clients) and sends them to the search & storage engine. Besides using the search REST service, one could also write queries against the search processor via the internal message bus. The interface looks like this:

**Search request:**
If the Elasticsearch engine should be replaced with another system, one has to implement the `eu.hradio.metadata.storage.IStorageClient` interface and extend the `eu.hradio.metadata.search.SearchEngine` enumeration.

The search processor can be configured by editing the `application.properties` file located in `search_processor/build/resources/main/`:

```properties
rabbitmq.address=rabbit
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
```
logging.file=./log/search_processor.log
logging.level.root=INFO

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

### 2.4.4. Metadata Processor

The metadata processor is able to process all metadata maintenance requests from the metadata REST service (or other clients, like importers) and to send them to the search & storage engine. The interface for metadata integration is described in Subsection "General Importer Interface" of Section 2.4.7.

The metadata processor can be configured by editing the application.properties file located in metadata_processor/build/resources/main/:

rabbitmq.address=rabbit
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.deleteIndicesIfExisting=false

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

After a change of this file, the metadata processor must be restarted. Log files can be found in metadata_processor/log.
2.4.5. Search REST Service

The search REST service is the interface for service, programme and schedule search. This service is publicly accessible. The following endpoints are available:

- /api/v1/services: Search for services
- /api/v1/programmes: Search for programmes
- /api/v1/schedules: Search for schedules

The REST interface looks like this:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns a list of ranked results.</td>
<td>Size: Number of items in the result set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>From: Start index in the result set</td>
</tr>
<tr>
<td></td>
<td><strong>Body</strong></td>
<td></td>
</tr>
<tr>
<td>POST</td>
<td>Complex Query. Syntax: [7]</td>
<td>Size: Number of items in the result set</td>
</tr>
<tr>
<td></td>
<td>Returns a list of ranked results.</td>
<td>From: Start index in the result set</td>
</tr>
</tbody>
</table>

Table 2: Description of the RESTful interface of the search service.
The search REST service can be configured by editing the application.properties file located in search_rest/build/resources/main/:

```properties
server.servlet.contextPath=/api/v1/
server.port=8080
rabbitmq.address=rabbit
rabbitmq.port=5672
rabbitmq.user=rabbitmq
```
rabbitmq.password=rabbitmq
rabbitmq.retries=100
rabbitmq.retryInterval=3000

rest.requestTimeout=3000

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

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

2.4.6. Metadata REST Service

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

- `/api/v1/providers/{provider id}/services/{service id}/schedules/{schedule id}/programmes/{programme id}`

Description of the place holders:

- **Provider id**: Arbitrary string (URL compliant) representing a unique id.
- **Service id**: Arbitrary string (URL compliant) representing a unique id.
- **Schedule id**: Arbitrary string (URL compliant) representing a unique id.
- **Programme id**: Arbitrary string (URL compliant) representing a unique id.

Ids can be chosen freely but it is recommended to use 64 bit numbers. It is up to the user to guarantee unique ids (uniqueness of all ids of a specific metadata type, e.g. service type). If a new metadata item has the same id as an existing one, the existing item is overwritten. If metadata is imported from RadioDNS, ids are generated automatically based on specific fields (e.g. service id is 64bit hash of the service name string).

The service has the following interface:
Table 3: Description of the RESTful interface of the metadata service.

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Retrieval of service providers, services, schedules and programmes. 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>-</td>
</tr>
</tbody>
</table>

**Body**

-
Example (1)

GET localhost:8090/api/v1/providers

RESULT:

RETURN CODE: 200

BODY:

```json
[
  {
    "name": "BR",
    "description": "Bayerischer Rundfunk",
    "services": [],
    "id": "br"
  },
  {
    "name": "MDR",
    "description": "Mitteldeutscher Rundfunk",
    "services": [],
    "id": "mdr"
  },
  {
    "name": "SWR",
    "description": "Südwestrundfunk",
    "services": [],
    "id": "swr"
  },
  {
    "name": "NDR",
    "description": "Norddeutscher Rundfunk",
    "services": [],
    "id": "ndr"
  },
  {
    "name": "WDR",
    "description": "Westdeutscher Rundfunk",
    "services": [],
    "id": "wdr"
  }
]
```
| "name": "RBB",
"description": "Rundfunk Berlin Brandenburg",
"services": [],
"id": "rbb"
}
GET localhost:8090/api/v1/providers/br/services/bayern2nord

RESULT:

RETURN CODE: 200

BODY:

```
{
  "name": "Bayern 2 Nord",
  "id": "br_bayern2nord",
  "bearers": [
    {
      "address": "dab:de0.1131.df12.0",
      "type": "DAB"
    },
    {
      "address": "http://streams.br.de/bayern2nord_1.m3u",
      "type": "HTTP"
    },
    {
      "address": "http://streams.br.de/bayern2nord_2.m3u",
      "type": "HTTP"
    },
    {
      "address": "dab:de0.1134.df12.0.0",
      "type": "DAB"
    },
    {
      "address": "dab:de0.1147.df12.0.0",
      "type": "DAB"
    },
    {
      "address": "dab:de0.1162.df12.0.0",
      "type": "DAB"
    }
  ],
  "genres": [
  
```
"name": "Kultur"
},
{
"name": "spezielle Musik"
},
{
"name": "Information"
}
],
"schedules": [
{
"startTime": "Jun 11, 2018 12:00:00 AM",
"stopTime": "Jun 12, 2018 12:00:00 AM",
"parentId": "br_bayern2nord",
"id": "br_bayern2nord_11062018",
"programmes": [],
"scopes": [
{
"name": "dab:de0.1131.df12.0"
}
{
"name": "http://br-mp3-bayern2nord-s.akacast.akamaistream.net/7/835/256282/v1/gnl.akacast.akamaistream.net/br_mp3_bayern2nord_s"
}
{
"name": "http://br-mp3-bayern2nord-m.akacast.akamaistream.net/7/841/256282/v1/gnl.akacast.akamaistream.net/br_mp3_bayern2nord_m"
}
{
"name": "dab:de0.1162.df12.0"
}
]
},
"parentId": "br"}
### PUT

**Body**

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

**Example**

```json
PUT localhost:8090/api/v1/providers

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

RESULT:
RETURN CODE: 201
```

**Creation of service providers, services, schedules and programmes. If item exists already, it will be overwritten.**

### DELETE

**Body**

DELETE localhost:8090/api/v1/providers/br

**Example**

```
RESULT:
RETURN CODE: 200
```

**Deletion of a single service provider, service, schedule or programme. If item exists already, it will be overwritten. Returns 404 if item does not exist, 200 if it does exist.**

Example bodies for the creation of metadata items:
Table 4: Example bodies for all metadata types

<table>
<thead>
<tr>
<th>Metadata Type</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Provider</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;id&quot;: &quot;0&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;name&quot;: &quot;ServiceProviderTest1&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;description&quot;: &quot;A test service provider&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Service</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;id&quot;: &quot;0&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;name&quot;: &quot;ServiceTest1&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;description&quot;: &quot;A test service&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
The metadata REST service can be configured by editing the application.properties file located in metadata_rest/build/resources/main/:
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
logging.file=./log/metadata_rest.log
logging.level.root=INFO

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

2.4.7. Metadata Importer

The flexible architecture allows for different data integration modules. Currently, an importer for RadioDNS data and for Dynamic Label+ data (coming from a DAB Tuner, sent over a STOMP broker and converted to JSON) exist. Both use the message bus for data transport to the metadata processor.

Dynamic Label+ Importer

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

```json
{
    "RabbitMQ": {
        "Address": "rabbit",
        "Port": 5672,
        "Queue": "dl_metadata",
        "Username": "rabbitmq",
        "Password": "...",
        "Retries": 50,
        "RetryInterval": 2000
    },
    "Stomp": {
        "Config": "tcp://193.96.226.192:61613",
        "Queue": "/hradio/metadata"
    }
}
```
Its essential parameters are the connection parameters for the internal message bus (RabbitMQ) and those for the STOMP server. The Dynamic Label+ information is encoded in JSON and follows the structure described by 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,
        "ServiceLabel": "Absolut relax",
        "TimeStamp": 1520434249,
        "Tags": [
            {
                "ContentType": 1,
                "ContentTypeName": "ITEM_TITLE",
                "TagText": "DO YOU REALLY WANT TO HURT ME"
            },
            {
                "ContentType": 4,
                "ContentTypeName": "ITEM_ARTIST",
                "TagText": "CULTURE CLUB"
            }
        ]
    }
}
```

As shown within the example, the JSON structure which describes the Dynamic Label+ data implements a straightforward mapping.

The importer has two parts, the collection part which is implemented as a Python script and a conversion part which is part of the metadata processor module, written in Java. The first part collects data from the STOMP server and sends it via the internal message bus to the metadata processor. The metadata processor converts the incoming Dynamic Label+ data into Programme Event items by
evaluating the toggle bit in order to get exact start and end times. The result is stored in the metadata storage module.

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.

**RadioDNS Importer**

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

```json
{
"RabbitMQ": {
"Address": "rabbit",
"Port": 5672,
"Username": "rabbitmq",
"Password": "rabbitmq",
"Retries": 50,
"RetryInterval": 2000
},
"Crawlers": {
{
"Every": 3,
"Queue": "metadata",
"Urls": [
"http://epg4br.irt.de/radiodns/spi/3.1/SI.xml",
"http://epg4rbb.irt.de/radiodns/spi/3.1/SI.xml"
]
}
}
```

Besides the parameters for the connection to the internal message bus (RabbitMQ) there is a section to configure a set of crawlers. A crawler is executed in a certain frequency (Example: every 3s), downloads SI files from a set of URLs and tries to download PI files corresponding to the services defined in the downloaded SI files.

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 metadata processor module which does the conversion to the internal metadata format.
General Importer interface

Together with the metadata data model, the internal bus system and the exact, language agnostic interface definition for metadata retrieval and manipulation, it is straightforward to implement a custom importer for metadata.

For metadata manipulation and retrieval, the following message formats are defined:

<table>
<thead>
<tr>
<th>Message</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get/create/delete</td>
<td>{</td>
</tr>
<tr>
<td>metadata request</td>
<td>&quot;content&quot;: &quot;[content]&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;itemId&quot;: &quot;[id]&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;parentId&quot;: &quot;[parentId]&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;requestType&quot;: &quot;GET</td>
</tr>
<tr>
<td></td>
<td>&quot;metadataType&quot;: &quot;SERVICE_PROVIDER</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Get/create/delete</td>
<td>{</td>
</tr>
<tr>
<td>metadata response</td>
<td>&quot;content&quot;: &quot;[content]&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;type&quot;: &quot;OK</td>
</tr>
</tbody>
</table>

The request has to be sent to the internal bus system to the queue “metadata_request”. It uses RabbitMQ’s request/response communication pattern:

![Request/response pattern as implemented in RabbitMQ. Source: [4]](image)

Potential importers have to implement this communication pattern (see https://www.rabbitmq.com/tutorials/tutorial-six-python.html for a tutorial in a large set of programming languages). An implementation for Java can be found in the
eu.hradio.metadata package (classes: MetadataRequester, MetadataRequest, MetadataResponse).

2.5. SYSTEM REQUIREMENTS

The platform was tested on Ubuntu Linux 17.10. But all 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.

2.6. 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:

![Communication Platform containerization](image)

Figure 7: Communication Platform containerization.

The dotted lines indicate platform modules that are planned but not yet available in the release.

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:

![Diagram showing Containers vs Virtual Machines]

Figure 8: 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: [5]
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/br/services/bayern2nord
- **Search REST:** http://141.84.213.235:8080/api/v1/services?q=name:B*

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: RUN THE PLATFORM USING PRE-BUILT ARTEFACTS

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

**Preliminaries:**

- Install Docker Community Edition (For Ubuntu, see: https://docs.docker.com/install/linux/docker-ce/ubuntu/#install-docker-ce)
- Install Docker Compose (For Ubuntu, see https://docs.docker.com/compose/install/#install-compose)

**Steps:**

- Download build artefacts from https://drive.google.com/drive/folders/1Oh8EOoXLC-I6Wfh7gPKIYWBXsyNAOJXB?usp=sharing (take latest version)
- Extract archive
- Run start script: ./start.sh
3.3. HOW TO: CHECK OUT SOURCES, BUILD AND RUN THE PLATFORM

This guide shows how to build the platform from source.

Preliminaries:

- Install JDK and Gradle (For Ubuntu, see: https://www.vultr.com/docs/how-to-install-gradle-on-ubuntu-16-10)
- Install Docker Community Edition (For Ubuntu, see: https://docs.docker.com/install/linux/docker-ce/ubuntu/#install-docker-ce-1)
- Install Docker Compose (For Ubuntu, see https://docs.docker.com/compose/install/#install-compose)

Steps:

- Clone git repository at https://gitlab.lrz.de/ru28muy/hradio.git
- Goto repository root folder and run build script: python ./build.py
- A new folder “build” was created.
- Goto new folder “build” and run the platform: cd build ./start.sh
- Stop platform execution with Ctrl+C.

Alternative: Run the do_all.sh script in the root folder. It will build and run the platform with one command.

3.4. HOW TO: IMPLEMENT YOUR OWN IMPORTER

This guide shows how to implement your own importer in Java using classes from the eu.hradio.commons and eu.hradio.metadata package. It is possible to implement an importer in a lot of different programming languages. Chapter “General Importer Interface” describes the necessary interfaces.

Preliminaries:
Install JDK and Gradle (For Ubuntu, see: https://www.vultr.com/docs/how-to-install-gradle-on-ubuntu-16-10)

Steps:

- Get metadata and commons library sources from git repo: https://gitlab.lrz.de/ru28muy/hradio.git

- Build metadata and commons library:
  - Goto folder commons
  - Run gradle build
  - Goto folder metadata
  - Run gradle build
  - => Folders [commons|metadata]/build/libs contain the jars now.

- Create Java project e.g. with Spring Boot (https://spring.io/guides/gs/spring-boot/)

- Add commons and metadata dependencies to the build.gradle file of your project:

```java
//...
repositories {
  //...
  flatDir {
    dirs './commons/build/libs' //Choose path
    dirs './metadata/build/libs' //Choose path
  }
}
//...
dependencies {
  //...
  compile name:'metadata-0.1.0'
  compile name: 'commons-0.1.0'
  //...
}
```
Implement importer by using the class eu.hradio.metadata.search.MetadataRequester to send metadata change requests to the system. See the metadata_rest project as an example.

If the importer should be part of the platform deployment, the following steps are necessary:

- Add dockerfile to your project folder:

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
COPY [FULL PATH OF YOUR IMPORTER JAR FILE].jar app.jar
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app.jar", "--spring.config.location=file:./config/"
```

- Change build script build.py in folder [platform root]. See “metadata_rest” entries in the build script as an example.

- Change docker compose file in folder [platform root]/docker. See “metadata_rest” entry as an example.
4. CONCLUSION AND OUTLOOK

The previous chapters described requirements, data models and components of the first iteration of the HRADIO Communication Platform. It currently consists of services to search and maintain metadata and an interface to import metadata. The importer interface is currently implemented by two importers: One for RadioDNS data and one for Dynamic Label + data. In addition, the How to chapter should help potential users to get started with the platform.

As mentioned in the platform description, the next iterations will contain a module to forward search requests to multiple search nodes (search federation). With this architecture extension, we’ll be able to reach 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 basic idea behind the system works as sketched in the following figure (see figure description as an explanation).

This module uses a broad query to find other search hosts in a globally shared routing table (Routing Candidate List). Once matching search hosts are found, the search request is forwarded to those hosts. The search results are then collected, merged and sent back to the requester.

We are currently in the design phase of this extension. However, the actual system in its currently-available state was designed with these changes in mind.
The second goal is the design and implementation of a radio service and programme recommendation system. Based on the metadata (see e.g. the components in dotted lines in Figure 6) provided by the developed platform, a recommendation system that meets the requirements, derived from the user scenarios, will be designed, implemented and evaluated via user studies.

The third goal is the design and implementation of a privacy-aware user statistics collection module which can then be exploited for a further enhancement of the quality of the recommender system (besides other use cases). We are currently investigating technologies and tools in the realm of privacy-preserving data mining (e.g. differential privacy algorithms).
REFERENCES


