André Silva
Pedrosa

Desenvolvimento de uma arquitetura escalável para extrair metainformação de bases de dados médicas distribuídas

Development of a scalable architecture to extract metadata from distributed medical databases

DOCUMENTO PROVISÓRIO

Universidade de Aveiro
2021

André Silva Pedrosa

Desenvolvimento de uma arquitetura escalável para extrair metainformação de bases de dados médicas distribuídas

Development of a scalable architecture to extract metadata from distributed medical databases

o júri / the jury

presidente / president   Prof. Doutor -

-

vogais / examiners committee   Prof. Doutor -

-

Prof. Doutor José Luís Guimarães Oliveira

professor catedrático da Universidade de Aveiro

**agradecimentos /
acknowledgements**

Começo por agradecer a todas as pessoas que conheci por meio da minha estadia na residência P.I.L.A. Dos veteranos aos caloiros um grande obrigado por me darem uma razão de continuar a trabalhar e por criarem um ambiente de família.

Aos meu orientador, o professor José Luís Oliveira, e coorientador, João Almeida, por acreditarem no meu trabalho e não desistirem de mim, para além das várias vezes que possa não ter atingido as suas expectativas.

Agradeço aos vários amigos e amigos que fui conhecendo ao longo do meu percurso académico da Universidade de Aveiro. Em especial, obrigado aos fundadores do grupo SASA LELE, Filipe Pires e João Alegria, por suportarem o meu feitio e por estarem ao meu lado durante o nosso curso.

À malta de Leiria que para além de me ter descolado para um local distante, continuarem a receber-me como um amigo chegado. Agradeço especialmente ao Leandro pela paciência e preocupação.

Por último, mas não menos importante, queria agradecer à minha família por estarem lá nos meus altos e baixos e por me darem força para eu seguir em frente.

Palavras Chave

Resumo

Para realizar estudos médicos, tais como o impacto de um fármaco numa determinada população, os investigadores precisam de acesso a dados reais de organizações médicas, tais como hospitais, para que os seus estudos tenham resultados fiáveis. No entanto, para os investigadores, este é um processo difícil e demorado até conseguirem encontrar o conjunto de dados que melhor se adapta ao seu estudo, dado que muitas vezes não têm acesso direto aos dados reais. Para ajudar neste processo, existem plataformas centralizadas que descrevem o conteúdo dos conjuntos de dados através de meta-dados. Atualmente, em muitas destas plataformas, estes meta-dados são atualizados manualmente, um processo lento e tedioso, o que leva a que se tornem muito facilmente desatualizados.

O objetivo deste trabalho é desenvolver um sistema que extrai automaticamente meta-dados diretamente dos dados reais das bases de dados e os envia para várias plataformas que estão a expor meta-dados de modo a que estes se mantenham atualizados. O sistema pretende trabalhar em bases de dados ligadas ao grupo EHDEN (European Health Data and Evidence Network), que pretende criar uma rede federada de bases de dados médicas na Europa, seguindo os mesmos princípios e ferramentas utilizados pela comunidade OHDSI (Observational Health Data Sciences and Informatics). A extração de meta-dados é feita por agentes que são instalados juntamente com o sistema de produção das bases de dados. Os meta-dados são então enviados para um sistema de gestão de meta-dados, construído sobre uma framework de fluxo de dados chamada Kafka. Este sistema de gestão é composto por um conjunto de componentes que foram desenvolvidos de acordo com uma filosofia de microsserviços, tendo sempre em mente a possibilidade de escalar cada um deles.

Abstract

To conduct medical studies, such as the impact of a drug on a certain population, researchers need access to real data from medical organizations such as hospitals for their studies to have reliable results. However, for researchers, this is a difficult and time-consuming process until they can find the dataset that best fits their study, as they often do not have direct access to the actual data. To help in this process, there are centralized platforms that describe the content of datasets through metadata. Currently, in many of these platforms, this metadata is updated manually, a slow and tedious process, which leads to it becoming outdated very easily.

The goal of this work is to develop a system that automatically extracts metadata directly from databases' actual data and sends it to various platforms that are exposing metadata so that they are kept up to date. The system intends to work on databases linked to the EHDEN (European Health Data and Evidence Network) group, which intends to create a federated network of medical databases in Europe, following the same principles and tools used by the OHDSI (Observational Health Data Sciences and Informatics) community. Metadata extraction is done by agents that are installed along with the production system of the databases. The metadata is then sent to a metadata management system, built on top of a data streaming framework called Kafka. This management system is composed of a set of components that were developed according to a philosophy of microservices, always keeping in mind the possibility of scaling each one.

Contents

Glossary

ACHILLES Automated Characterization of Health Information at Large-scale Longitudinal Evidence Systems.

AMQP Advanced Message Queuing Protocol.

API Application Programmable Interface.

BBMRI-ERIC Biobanking and BioMolecular Resources Research Infrastructure-European Research Infrastructure Consortium.

CDM Common Data Model.

CDN Content Delivery Network.

CDW Clinical Data Warehouse.

CRUD Create, Read, Update and Delete.

CSS Cascading Style Sheets.

CSV Comma-separated values.

DATS DatA Tag Suite.

DPC Data Partner Clients.

EHDEN European Health Data and Evidence Network.

EHR Electronic Health Record.

EMIF European Medical Information Framework.

EU European Union.

FAIR Findability, Accessibility, Interoperability, and Reusability.

FTP File Transfer Protocol.

GAAIN Global Alzheimer's Association Interactive Network.

HDFS Hadoop Distributed File System.

HTML HyperText Markup Language.

HTTP HyperText Transfer Protocol.

IDE Integrated Development Environment.

JDBC Java Database Connectivity.

JMS Java Message Service.

JSON JavaScript Object Notation.

NADA National Data Archive.

NPM Node Package Manager.

OHDSI Observational Health Data Sciences and Informatics.

OMOP Observational Medical Outcomes Partnership.

ORM Object–Relational Mapping.

PDS Project Data Sphere.

RD Research and Development.

RDBMS Relational Database Management System.

REDCap Research Electronic Data Capture.

REST Representational State Transfer.

SQL Structured Query Language.

XSS Cross-Site Scripting.

Introduction

1.1 OBJECTIVES

- have a platform capable of holding and displaying metadta in a intuitive and user-friendly way
- find a tool that extracts metadta from several databases
- design a system capable of sending dta to aplication to keep their data up to date
- make use of new and growing popularity tecnologies

1.2 OUTLINE

Background

You can't have happiness without pain, you need a little bit of rain to have a little bit of rainbow
- Felix Lengyel

Oftentimes medical researchers do studies associated with diseases, such as determining the impact of a certain drug or find variables that are characteristic of certain diseases. To perform such studies and have reliable results, a great amount of data is required. To obtain that data, these researchers have to contact medical data owners to have access to relevant data that can help improve their analysis and/or findings.

With this procedure emerges several problems for the researcher such as he has to find institutions willing to share data and the process of contacting the data providers can be cumbersome. To aid in this whole process, several data hubs have been developed with the purpose of making the process of data discovery easier. One important aspect of such data hubs is that they present to the researcher meta data, which is aggregations or summaries of the original data. Metadata has the advantage that one doesn't have to deal with the anonymization process of medical records, since only summaries of the initial data are retrieved [1], [2]. With this dependency on the original data, emerges an important problem of data hubs which is, metadata can easily be outdated after a small time window. This could not raise a big problem, if the records were updated regularly, however this rarely happens, mainly because either the update process is difficult or because metadata has to be manually extracted and uploaded to the data hub. A problem that still might arise from such platforms, is those different datasets very often have different representation for the same concept or the data is organized in a different layout. The research is then hampered since either different approaches have to be taken to analyze each dataset.

The European Health Data and Evidence Network (EHDEN) [3] project has affiliations with several institutions, data sources and data custodians across the European Union (EU), which the main goal is to, within a federated network [4], harmonize their data to the Observational Medical Outcomes Partnership (OMOP) Common Data Model (CDM), which

was developed by Observational Health Data Sciences and Informatics (OHDSI), a multi-stakeholder, interdisciplinary and collaborative organization that brings out the value of health data through large-scale analytics [5]. With a CDM, the problem of having different representations for the same data across distinct data sources is solved. Researchers can now develop a single analysis method and then apply it to all gathered datasets and these methods can be optimized for this specific data model, which allows large-scale analytics. Furthermore, also improves collaborative research [5]. Still, within the scope of the EHDEN project, the project has a database catalog [6], built with the MONTRA framework [2], where data owners fill metadata about their data source manually, which brings the outdated problem already mentioned before. Additionally, whenever new metadata fields are introduced, the data owners of all data sources have to go manually update their metadata form.

To build a valuable data hub is then important to take into account how to:

- extract metadata from a data source
- upload and update the metadata on the data hub
- automatize the two processes mentioned before
- receive and display the metadata on the data hub in a way that facilitates readability.

2.1 Metadata Visualization Tools

It will then be explored existing visualization platforms that enhance data discovery by presenting summaries or metadata of records (data sources, datasets).

In some cases data can't be publicly available because it contains sensitive data or simply the data owner might not want to share some portions of the data, for that the tools analyzed should have privacy protection mechanisms, allowing to customize the access and manipulation of data stored.

Furthermore, considering we want to improve and assist data discovery and reuse it is important to have good data management to simplify such processes. However, humans fail to achieve the necessary processing levels with present-day scientific data. It is then important that data is provided in such a way that machines can fetch, understand, analyze and act on data. For that the Findability, Accessibility, Interoperability, and Reusability (FAIR) Guiding Principles were established which contain a series of considerations for data publishing to supports both human and machine operations such as deposition, exploration, sharing and reuse [7].

Finally, it is preferential for such a tool to be open source since the available solution might need some changes to solve our specific problem, and also it makes it possible to receive contributions from the community.

Search Method

Regarding this subject, there was already done a systematic review of several tools that fit within the current search pool. Its objective was to "identify projects and software solutions that promote patient electronic health data discovery, as enablers for data reuse and advancement of biomedical and translational research" [8]. From the final 20 systems,

they captured their interoperability, what type of data they were providing and their after effect related to scientific results and improvements to better healthcare. To perform their search they only used PubMed [1] considering it indexes a substantially amount of health-care related work and provides a public Application Programmable Interface (API) which allows automation of the retrieval process. The programmatic retrieval was done using the Biopython framework[2] to query PubMed and consisted of eight steps. First, a set of publications was retrieved using the search query ("data" AND "discovery") OR "discovery platform" applied to the publications' titles. From the results set a filtering process was done based on their title, excluding irrelevant publications from the following steps. From each of the remaining publications, the 20 most similar publications were retrieved. The same title filtering process mentioned before was applied again to the new result set. The 5 most similar publications of each remaining publication were fetched. The title filtering process was executed, then an abstract filtering process and finally a full-text assessment. On all searches done the time window was set from January 2014 to September 2018. Moreover, publications associated with molecular biology were excluded.

Initially, the same approach was taken, now considering a time window starting in November 2018. However, after the second step, no new platforms were found. Because of that, a variation of the search method mentioned before was undertaken. The first step was skipped and some of the final tools presented on the systematic review were considered as a base set for the next steps. From the 20 presented, only 8 were used on this initial set, since some tools weren't active or didn't have any feature mentioned in the previous section (FAIR and data protection). Nonetheless, besides being discarded in terms of their metadata visualization features, 1 of these tools was analyzed for its metadata extraction features and 3 of them were analyzed for their metadata sharing network architecture, which will be detailed in the next sections. Coming back to the search method, the 20 similar publications for each of the 8 papers considered were fetched, and the title filtering process was done, followed by an abstract filtering process and finally a full-text evaluation. With this search method, 2 additional tools were considered, making a total of 10 tools to analyze on the section of metadata visualization tools.

eGenVar

Good data management and an organized data sharing can improve work effectiveness, and increase data analysis. One method to accomplish this is by having the data at a central repository and then provide clear and strong interfaces to interact with the data. However, because of legal and privacy rules, not all data can be stored on a public central repository. The software suite called the eGenVar [1] data-management system, allows users to report, track, and share metadata on content, origin and history of files, without compromising privacy or security. The tool can be seen as a metadata portfolio since it could be used to search data while the original files remain in a protected location. Users need to have an account

---

[1]https://pubmed.ncbi.nlm.nih.gov/
[2]https://biopython.org/

to access the system and once created can immediately start using the system for search operations, however, addition, deletion and update operations over content require a personal profile. It is designed to connect current Laboratory Information Management Systems and workflow processing systems and to keep the source of data that is being processed through distinct systems at different locations. Central to the system is a tagging process that allows users to tag data with new or pre-existing information, such as ontology terms or controlled vocabularies, at their convenience. The system includes a server, a command-line client, other clients that can be developed in several programming languages and a web portal interface.

MONTRA

Data catalogs are a good way to gather and present information of different areas, however, having to build a different web-based application for each distinct situation is not feasible. To aid this creation, MONTRA [2] was proposed as a flexible base architecture for composing data integration platforms, mainly associated with the biomedical field, allowing to centralize and share data originating from several and heterogeneous sources. MONTRA can achieve this last point, by requiring the definition of a metadata skeleton within a community of data sources, which describes the original data, so different data sources following the same skeleton template will have a common representation, leading to a homogeneous representation of their metadata. Yet, it is important to salient that the system only contains a skeleton of the underlying data sources, the original data is still stored in the cataloged sources' system. The skeleton definition is an easy process any data custodian can do by saving it as a spreadsheet file and then submitting it through the application's web interface. Then, the framework allows users to view, search, modify and delete information through simple forms, where access is controlled via a Role-Based Access Control system to ensure that proper access restrictions are imposed. Also, a RESTful API is available which provides a set of programmatic endpoints which allows the creation of third-party applications on top of the framework.

This framework is in used to deploy the European Medical Information Framework (EMIF) Catalogue [9], a platform with the goal to be a marketplace where data owners can publish and share information about their clinical databases, allowing biomedical researchers to search for databases that meet their research needs. Currently, the EMIF Catalogue holds many distinct projects, combining, for example, data available in pan-European Electronic Health Records and Alzheimer cohorts. Also, as mentioned at the beginning of this chapter, this framework was as well used to develop the EHDEN portal [6], which beyond allowing the search of databases over the EHNDEN network, makes available all tools and services built under the EHDEN project.

REDCap

Realizing the need for researchers to be able to secure and easily collect and share data, a team at Vanderbilt University developed Research Electronic Data Capture (REDCap) [10], which allows data collecting and metadata gathering. REDCap's data capture tools can either be structured to work as a sequence of forms that investigators fill out as they advance through their projects or as a survey meant to be filled by research subjects. REDCap

allows visualizing collected data, providing views with basic statistical measures and chart visualizations, enabling the export of the data to several common formats. Several features to help assure data quality are available, where Data Quality reports identify missing or incorrect values and outliers, validation errors and also allows the creation of custom rules to evaluate data correctness. Collected Data collected be imported using the Data Import tool, furthermore, the system offers an API to support remote insertion and fetch of data. There are also many features that enable support for various types of clinical and basic science research. REDCap also has a collaboration functionality, enabling investigators, after adding team members to a project, to assign permissions to each based on their roles and data needs.

The tool is used by the Vanderbilt research data warehouse framework [11], which consists of repositories with identified and de-identified clinical data and uses tools top of its data layer to help researchers across the enterprise, REDCap being one of them. Finally, the Ontario Brain Institute's "Brain-CODE" [12] is a platform designed to promote the collection, storage, sharing and analysis of data over several brain diseases, with the intention to understand common underlying causes of each specific dysfunction and find new ways to develop a treatment. REDCap is one of the clinical data management systems used to collect demographic and clinical data.

Data Sphere

In the late phases of a clinical trial, scientists could retrieve a great amount of usable data about the effectiveness of certain therapeutic approaches for oncologic diseases. With the decrease of cancer deaths over the years, another research paradigm is needed to find new or improve these therapeutic approaches. This lead to promote data-sharing attempts to make clinical trial data accessible to the scientific research community. The Project Data Sphere (PDS) [13] provides a platform that meets these data-sharing needs, giving the possibility to share raw data from late-phase oncology clinical trials. To share their data, data owners have to sign a data-sharing agreement that contains some extra data about the data they want to upload. If this data application gets accepts, the responsibility of patient privacy is on the data providers. Authorized users are then given the possibility to access and download all datasets made available on the platform. To become an authorized user, the platform requires that users send an application with their background and an agreement to the terms of use. Having these processes of sharing and getting data, prevents researchers from making different applications for each data set and allows having a more diverse data pool which can improve results from their analysis.

As of October 2021, the PDS website had available cancer trial data from over 200 trials including over 240,000 subjects [14].

Molgenis

For biologists to efficiently capture, exchange and exploit big amounts of molecular data, user-friendly and scalable software infrastructures are needed. For that MOLGENIS [15] was developed as a generic, model-driven toolkit to speed the development of custom big-data biosoftware applications. Biological details of each biological system can be modeled using a

domain-specific language, developed using XML, not requiring extensive, technical or repetitive details on how each feature should be executed, but enabling to compactly specify what kind of experiment database is desired. MOLGENIS can also be used to create web applications to be used by biologists, tailored to their experiments, using reusable components. The creators of MOLGENIS saw an improvement of up to 30 times in terms of efficiency when comparing to hand-writing software, besides providing several features hard to achieve by hand that were not made available by similar projects.

With the goal of developing new biomarkers and drugs, the Biobanking and BioMolecular Resources Research Infrastructure-European Research Infrastructure Consortium (BBMRI-ERIC) project [16] enables the research of basic mechanisms underlying diseases, by providing fair access to human biological samples and their associated biomedical and biomolecular data. Here MOLGENIS is used to develop their Directory 1.0 which presents an overview of the BBMRI-ERIC ecosystem with its distributed structure and helps users find biobanks of their interest.

To create a centralized research resource for Research and Development (RD), the RD-Connect [17] project associates genomic data to subject registries, biobanks, and bioinformatics software. To help RD researchers to search for RD biobanks and registries and also inform availability and accessibility on each database's content, the RD-Connect project has the D-Connect Registry & Biobank Finder tool, which is also a portal to other of their tools. One of them is the RD-Connect Sample Catalogue, which was developed using MOLGENIS, having an inventory of RD biological samples available in associated biobanks.

Cafe Variome

Data discovery applications connect data owners with data seekers and therefore promote data sharing, however, this last process can bring some complications. Cafe Variome [18] is a general-purpose data discovery platform, with the goal not to be a place to store, curate or integrate information but to provide a platform to browse through the existing data. It was developed following design principles that take into account important and emerging standards, easy to use by system administrators since is composed of a single simple software package, and also by data seekers, with flexible options allowing to customize each installation to the area of use, with a special interest on the "genotype-to-phenotype" application. To the administrator, it is given the ability to determine which data fields can be used for discovery and/or be displayed as results, and also which records are allowed to be used for discovery.

For a data seeker, the number of records that match the search term(s) are split by data source and also split into three categories:

- open access: the user is allowed to see the data present on the system.
- linked access: the user can't see directly the data stored in the system. An external data source data link or data source contact information is provided.
- restricted access: the user has to make a data request to access the data.

Mica & Opal

Enhancing the distribution of data of epidemiological studies and making research databases interoperable are some important factors to maximize the reuse of resources and, as a consequence, promoting health developments. Two open-source software tools, Opal and Mica [19], address this by providing off-the-shelf solutions for epidemiological data compatibility, management and distribution, which were proposed by Maelstrom Research.

A centralized web-based data management system is implemented with Opal, where researchers can securely import/export a wide range of data types and formats using a simple interface, which is then converted to a common model. Nevertheless, the tool also gives the possibility to read data directly from a data source. Privacy is ensured by storing participants' identifiers on a separate database, offering tools to manage these to administrators. With data already imported, Opal web tools aids in data curations and quality control processes, enabling also for descriptive statistics computations to be automatized, presenting such on graphical charts. Taking into account all these features, using Opal over multiples studies turns out to be a strong tool to standardize their epidemiological data, which then facilitates data discoverability and metadata browsing using the other tool proposed, the Mica web portal.

Mica is used to create metadata portals for single or consortia of multi epidemiological studies, with a particular enface on supporting observational cohort studies. It is composed of several modules that allow data owners, study or network supervisors to customize detailed information about their epidemiological research datasets, studies and networks facilitating the process to organize and distribute data. After Mica is populated with study metadata, researchers can use the built-in search engine to rapidly encounter the information that they lack for their research projects. Furthermore, on multi-study instances, studies with a certain profile, that gather data of the desired health outcomes, risk factors and/or confounding factors are easily identified and gathered by users. Combining Mica with one or more Opal database(s) enables users to safely query actual study data, present on remote servers, applying searches exceeding the metadata.

Besides developing these two tools, Maelstrom Research and its partners adopted them to produce the Maelstrom Research Catalogue, a flexible collection of epidemiological studies which offer an intuitive web solution for data discovery.

BioSharing

Another interesting tool to promote data dissemination and discoverability is BioSharing [20], recently known as FAIRSharing [21], a portal of connected, informative and discoverable information about standards, databases, and journal and funder data policies in the life sciences. It acts as a "shop window" for the three types of data mentioned, detailing relations between them, presenting metrics, as standards are developed and achieved in the databases, allowing to create a historical view, showing when certain standards are created or deprecated and when updates to a database or policy happen, giving the opportunity to data seekers to check the progression of each resource. Currently, all records are manually curated, however

many of them have been added and updated by community users, instead of FAIRSharing curators themselves. Users are capable of claiming records for resources they manage, allowing not only to make sure that the data on their resource is valid and updated but also to gain credit for their work. This Community curation feature, together with the coupling and enclosure of each record into standards and databases, helps FAIRSharing become a correct and complete representation of metadata standards, databases and policies in the life sciences.

Dataverse

One of the tools that weren't present on the previous systematic review, and were found on the new search process was created by the Dataverse Project [22], an open-source web application to store, share, explore, analyze and cite research data.

Before the Dataverse Project, researchers had to choose between receiving credit for their data, by handling distribution themselves, however, it is difficult to have long-term preservation guarantees. The latter could be solved by sending the data to a third-party archive system but without receiving much credit. The Dataverse Project solves these problems: facilitates the processes of sharing data with others, also allowing to replicate others' work easily, giving the deserved credit and web visibility to all entities involved with the data being shared. It does this by presenting a Dataverse collection (a virtual archive that contains datasets, and each dataset contains detailed metadata and data files) on the data authors' website with its original look, feel and branding along with a citation for the data. Yet, that page is served by a Dataverse repository (an installation of Dataverse, which hosts multiple Dataverse collections), with institutional support, and long-term preservation guarantees.

NADA

Another tool found on the search process performed was National Data Archive (NADA) [23], a web-based cataloging application that can be used as a base structure to create portals that allow users to browse, search, apply for access, compare and download census or survey data. It was originally developed to assist the establishment of national survey data archives and is currently in use by several regional, national and international organizations.

When a NADA installation is deployed, the default catalog created is the Central Data Catalog, where all studies uploaded to NADA are visible, searchable and accessible through it. In most cases, this is enough to have the data stored and organized, however it is possible to split the contents of this central catalog into more specific collections. This brings advantages for both the users and the administrators since the former can more easily filter and search collections of surveys that are related, the latter can better distribute management responsibilities to specific administrators for their specific study area.

The NADA uses the Data Documentation Initiative (DDI) standard to represent the metadata for each study, where such documents are built outside the NADA application and then imported. Users can take advantage of such metadata by performing searches about surveys in the catalog down to the variable level.

The tool also allows to include citations at the study level, pointing to works that used the data of a certain study. These resources are convenient for researchers to see how the data have been used previously and also to show survey funders the study impact and that the data is being used for research purposes.

The level of access to the studies datasets can be controlled at the study level, allowing to have different access restrictions for distinct studies. The available access types are:

- Data not available
- Direct Access Data Files: no login required
- Public Use Data Files: login required
- Licensed Data Files: application required
- Data available in an Enclave: application required and the data is stored on the data owners site
- Data available at an external repository: data is stored on the data owners site

2.2 Metadata Extraction Tools

Associated with some projects mentioned in the previous section, several tools and processes are relevant for the task of extracting/profiling datasets. Next are presented such tools:

Search Method

TODO describe this

search terms: metadata (extration or creation),

"metadata" (extraction or creation) from database,

extraction from database,

metadata fingerprinting a database,

metadata updating,

ACHILLES

The OHDSI project, already mentioned at the beginning of this chapter, offers a variety of open-source tools [24] that can be used on several data-analytics use cases on observational patient-level data. Such tools can communicate with several databases that have adhered to the OMOP CDM, enabling them to homogenize the analytics for different use cases. With this, instead of having to build an analysis from scratch, a standard template can be used, turning the process of building analyses easier, and also enhancing transparency and reproducibility.

One of such tools, Automated Characterization of Health Information at Large-scale Longitudinal Evidence Systems (ACHILLES) [25], can produce summaries and metadata of CDM databases. These allow for characterization and quality assessment of observational health databases. It is implemented as a package written in R [3], which executes a series of SQL queries over the original CDM database to calculate all the specific summaries, which in the context of ACHILLES, are called analyses. The result of these queries is then placed on a target database schema chosen by the user. The resulting summaries are not study-specific

---

[3]https://www.r-project.org/

but can be used by researchers to explore and evaluate if the contents of databases can be used on studies that they intend to perform. However, certain summaries might reveal some information of the original data that the data owner is not willing to share or because is sensitive patient information. With this, variations of these tools are created where only certain summaries are extracted from the data source, removing the process of having to filter the output of the ACHILLES. Such a tool was developed associated with the EHDEN project [26].

DataMed

Facilitating the process to find appropriate datasets is a key aspect to improve data reuse in the biomedical domain, however, it is hard to achieve due to the biomedical data complexity and volume. DataMed's [27] mission is to build a data discovery index enabling users to efficiently search and access existing datasets that are spread across several repositories.

It developed a metadata ingestion pipeline that extracts, maps, and indexes by following the DatA Tag Suite (DATS) based on community input and an analysis of existing metadata from popular repositories. The DATS model mentioned is used to describe the metadata elements and the structure for datasets [28]. DataMed's pipeline was built as a horizontally scalable, message-oriented, extract-transform-load, loosely coupled distributed system, composed of a message dispatcher and one or more data processing segments. The dispatcher functions as an orchestrator by managing the data ingestion and processing pipeline through message queues. Each processing component performs operations on the data such as transformation, cleanup, and/or enrichment, saves the result to a document database and then notifies the dispatcher through the message queues. Next, a pipeline specification is used by the dispatcher to decide the next step, injecting a message on the message queue of the target consumer.

Since different data sources have distinct representations, the pipeline has to abstract retrieval modes and data formats. This is achieved by implementing different ingestors for the possible retrieval modes and data format combinations. Each particular ingestor transforms raw data into the DATS, and they make use of data iterator(s), which allows retrieving data only when necessary, in other words, data streaming, allowing to deal with the situations where the datasets are larger than the available system memory.

An interesting and important note is that there was already some work done on mapping datasets represented in the OMOP CDM to DATS [29].

Xtract

...

MetaExtractor

...

Skluma

...

Considering that we have an agent that extracts or gathers metadata from a data source, it is also important to think on how they will transfer or communicate data with the interface that clinical researchers use. Next it is presented some projects that design some sort of network architecture to retrieve data from the data owners site:

CafeVariome

Coming back again to Cafe Variome [18], it was designed to be used safely, with sensitive datasets. With this, it is often important to limit which persons can access the system and perform their data discovery browsing/queries. This leads to laboratories connecting in closed or semiclosed discovery networks with limited access. Cafe Variome offers the following network architectures:

1. hub and spokes arrangement: it has a single discovery portal where searches can be performed across all partner in the network
2. all-to-all networks: each laboratory has an individual interface over their data, that can be used by other network members
3. combination of the previous architectures
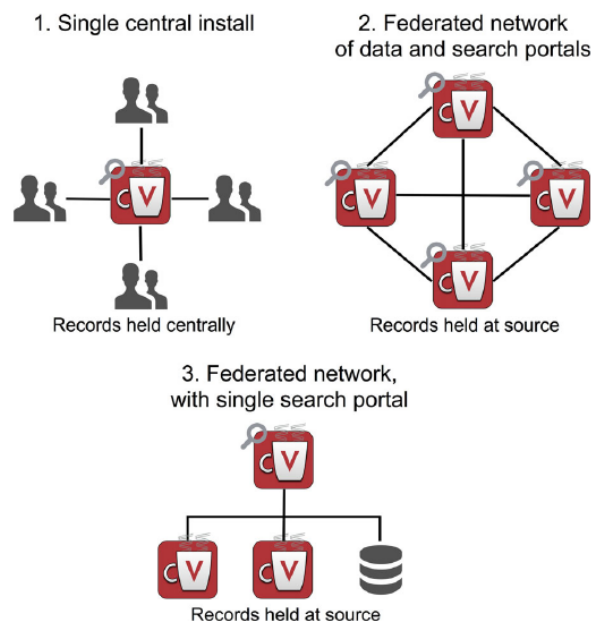
Figure 2.1: The available network architectures in Cafe Variome. Image retrieved from [18].

Despite the chosen arrangement, each node can choose to customize the access policies for specific records or fields, making them available for discovery to certain groups in the network.

GAAIN

One more tool analyzed because of its metadata sharing network its Global Alzheimer's Association Interactive Network (GAAIN) [30], in which its main objective is to organize a community for sharing Alzheimer's-related from diverse repositories around the world, where

data from different medical fields are combined together, taking into account existing policies of these repositories and preserving the ownership of the data being shared.

The architecture of GAAIN contains a central server that interacts with multiple client nodes, which are installed in the data partners site (Data Partner Clients (DPC)). These GAAIN DPCs don't gather data directly from their data partner production environment, instead, data is locally exported and transformed into a CSV file and only then loaded into the DPC. With this process a DPC will have a low footprint, not interfering with the data parters system, and also is given the freedom to the data owners to update their data when they find it convenient. To have this DPC running on the data partner's site, the only requirement imposed is to adapt current firewall configurations to allow incoming HTTPS traffic from the central server into their network. Data owners still have full control over their data, and to do so every GAAIN DPC has "on/off switch" which gives the possibility to immediately detach the data from the network.
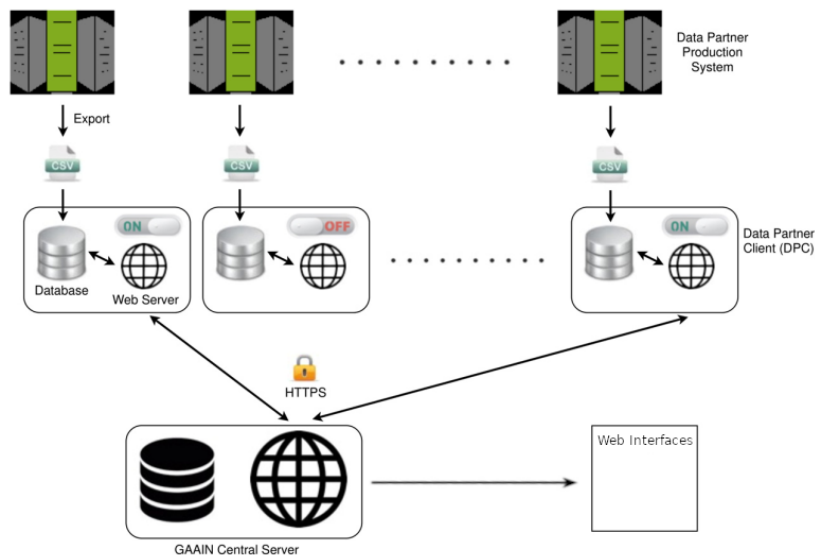
Figure 2.2: Network architecture of the GAAIN. Image created based on an original retrieved from [30].

Then, with this network architecture, GAAIN researchers can query the network through its web interfaces, which forwarded the search requests to the central server component which then queries every individual DPC. The results are sent back to the central server that aggregates them into a response to then be displayed on the web interface.

PopMedNet

With the increase of availability of electronic health information, the engagement to distributed health data research networks has risen. Popmednet [31] appears to facilitate the creation and management of distributed health networks, taking into account the demands of different data owners and researchers. The platform's adaptable architecture allows network designs that satisfy important requirements of data owners of a network, including data privacy and security requirements and network monitoring functionalities. Researchers have

available a set of features that aid in the processes of finding possible data owner collaborators, finding previously carried out research and learn more about the data in the network, thus the system is going beyond of just issuing queries.

The platform is made of two components: a web-based portal for issuing research requests and managing the network, and the clients' DataMart. They communicate with each other following a publish-subscriber philosophy, which removes the need for data owners to have open ports on their systems, solving an important security concern of existing direct external access to local data. With this approach, the DataMart Client is present at the data owner's local machine, behind their firewall, and all queries from the network portal are pulled from it, instead of them being pushed through an open port. Data owners can then use the DataMart Client to review requests fetched from the network portal by analyzing its metadata such as information of the requester and the description and purpose of the request. Then it can opt to execute the query and send the results back to the network portal, keep it for additional review or refuse it. With this asynchronous approach to querying, data owners have full control of their data and all its uses, however, instead of doing these review steps manually, it is also given the possibility to automate them.

EHR4CR

Yet another project with an interesting network design to share Electronic Health Record (EHR) is the EHR4CR [32] project. It aims to create a robust and scalable platform to share data from distributed Clinical Data Warehouse (CDW), taking into account requirements and policies, such as data protection, allowing to identify patient cohorts (a group of patients that satisfy specific criteria) and to extract patient-centric data.

The access to the clinical data present at each CDW endpoint is done through three logical layers:

- Legacy system layer: specific to the CDW
- Legacy Interface layer: deals with the complexity of translating queries against the EHR4RC data model to the terminology used by the specific CDW or EHR system and also convert the results of such queries to the common EHR4RC data model.
- EHR4CR Data source Endpoint layer: exposes a standard EHR4CR endpoint interface so other EHR4CR services can easily access the CDW or EHR system's data.

Once these layers are implemented and properly working, information about the service provider is added to the central registry of the platform, allowing users to discover and browse through them.

The platform's architecture was built around the Protocol Feasibility Scenario (PFS), which retrieved aggregated results from each data source about quantities of patients that fulfill a certain Eligibility Criteria (EC), and the Patient identification and Recruitment Scenario, that has the goal of retrieving patient information with consent to furder be used on research. The most important scenario to analyze is the PFS one, where makes use of three main components:

- Workbench: acts as an interface where an end-user can issue EC queries to execute on the network and see the aggregated results.

- Orchestrator: distributes the queries received from the workbench across the data endpoints of the network, joining the returned results and sending them back to the workbench.
- Endpoint: services with CDWs where the queries will execute
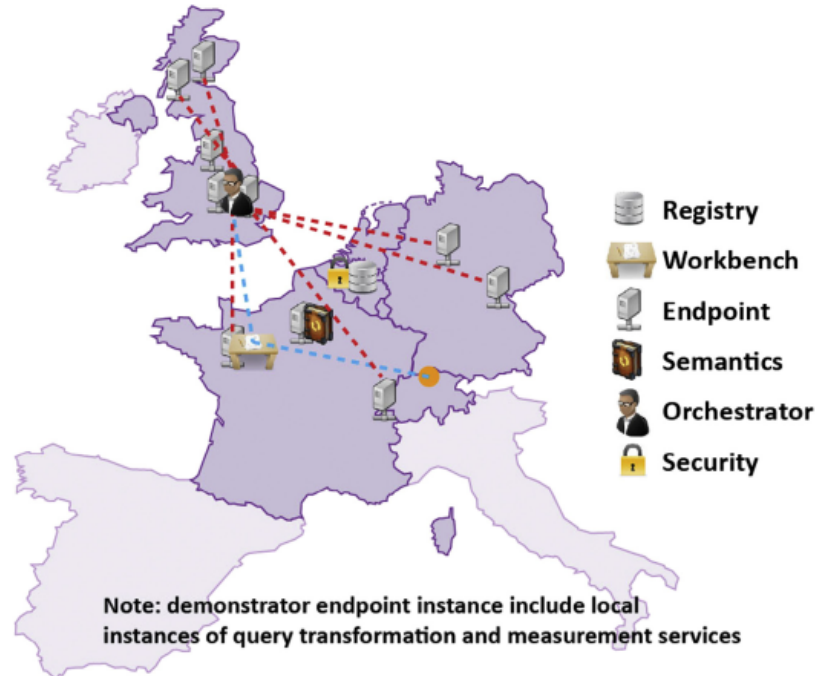
Figure 2.3: Interation between the main services used on the Protocol Feasibility Scenario. Image retrieved from [32].

While building the platform, data owners' common security policies and firewall restrictions were taken into account by supporting the invocation of web services using dynamically configurable transport bindings. Examples of such are publish-subscribe methods that were mentioned in the previous tool, where an endpoint retrieves (pull) incoming queries instead of receiving them directly (push). This guarantees agreement to local data owners' security policies, without jeopardizing existing authentication and authorization mechanisms on end-user and web service clients.

NextGen Connect

Data owners face challenges when trying to share vital patient data quickly since institutions are pressured to keep interoperability a secure cost-effective functionally. To do so, IT administrators need solutions that allow customization and are scalable.

In a continuously changing healthcare economy, NextGen Healthcare Solutions assist in supersede these challenges by: increasing the capabilities of sharing data; efficiently combine and transfer data over multiple systems; diminish costs and satisfy urging needs with Fast Healthcare Interoperability Resources (FHIR, a standard that defines data formats and API to exchange EHR) based APIs. One of such solutions is NextGen Connect, where NextGen Healthcare's goal is to offer the healthcare society a secure and practical way to share health data.

NextGen Connect contains an Integration Engine that, like a language interpreter who translates from one language into another, translates a message represented on a given format into another that the destination system understands. Whenever NextGen Connect receives a message from an external system, the Integration Engine perform:

- Filtering: analyses the message parameters and transfers or discards it from going to the transformation phase.
- Transformation: converts the message from its original format to the desired standard format (e.g., HL7 to XML) (Popular Medical standards supported, ex: DICOM, HL7). The tool also allows more customizable transformations by executing custom Javascript or Java code.
- Extraction: retrieve data from and send to a database.
- Routing: ensure that all messages are received by their destinations.

The components that are configured and execute the jobs mentioned above are called channels. It consists of multiple connectors which are in charge of loading the data into NextGen Connect (source connector) or loading the data to an outside system (destination connector). Each channel has exactly one source connector and one or more destination connectors. A possible configuration is to receive data over HTTP, then send the result of the configured transformations to a file and/or insert it into a configured database.

2.4 FINDINGS

Tool Name	Open Source	Visualization/Interaction		Extraction	Network
		Data protection	FAIR		
eGenVar [1]	✓ [4]	✓ (Users + Permissions)	✓	✗	✗
MONTRA [2]	✓ [5]	✓ (Role based)	✓	✗	✗
REDCap [10]	✗	✓ (Role based)	✓	✗	✗
Data Sphere [13]	✗	✓ (Authorized Users Only)	✗	✗	✗
MOLGENIS [15]	✓ [6]	✓ (Role based)	✗	✗	✗
Cafe Variome [18]	✗	✓ (Role based)	✓	✗	✓
Mica & Opal [19]	✓ [7]	✗	✓	✗	✗
BioSharing [20]	✓ [8]	✗	✓	✗	✗
Dataverse [22]	✓ [9]	✓ (Role Based)	✓	✗	✗
NADA [23]	✓ [10]	✓ (Access Request)	✗	✗	✗
ACHILLES [25]	✓ [11]	✗		✓	✗
DataMed [27]	✓ [12]	✗		✓	✗
Xtract [33]	✗	✗		✓	✓
MetaExtractor [34]	✗	✗		✓	✗
Skluma [35]	✓	✗		✓	✗
GAAIN [30]	✗	✗		✗	✓
PopMedNet [31]	✗	✗		✗	✓
EHR4CR [32]	✗	✗		✗	✓
NextGen Connect	✓ [13]	✗		✗	✓

-Similarity between extraction tools -> all converge different data model to a comon one -As we can see there is no tool that all the cheks ... -como vimos algumas aplicações podem dar insgits importantes que devemo spensar quando formos devenvolver as cenas, como por examplo, aquele paper que disse que segui uma filosofia de push em vez de pull para não pedir portas abertas aos colaboradores do projeto

2.5 SUMMARY

---

[4] https://github.com/Sabryr/EGDMS
[5] https://github.com/bioinformatics-ua/montra
[6] https://github.com/molgenis/molgenis
[7] https://github.com/obiba/mica2
[8] https://github.com/FAIRsharing/fairsharing.github.io/
[9] https://github.com/IQSS/dataverse
[10] https://github.com/ihsn/nada
[11] https://github.com/OHDSI/Achilles/
[12] https://github.com/biocaddie
[13] https://github.com/nextgenhealthcare/connect

Metadata Visualization

Among the presented tools in chapter 2, MONTRA was our go-to mainly because of its data source-centric approach. Also, no real data is shown here, only metadata is handled within the platform, however, it still allows to impose access permissions at several levels.

This chapter will detail the key concepts of the MONTRA framework and its internal data model. After this, there will be presented a refactoring process that was done to the platform, with its improvements and flaws fixed.

3.1 MONTRA

Originally, the MONTRA framework was developed by the Bioinformatics team of the Institute of Electronics and Informatics Engineering of Aveiro associated with the EMIF project with the goal to develop a common patient health information framework with emphasis on the research topics of Obesity and its metabolic complications and Markers for the development of Alzheimer's disease and other dementias. The code is publicly available on github[1], whereby Django 1.4, a high-level Python Web framework[36], was used as a framework to develop the entire system. This framework allows to develop complete web applications, in a faster and easier way. It contains a model layer that allows specifying database tables through python classes called models, following a Object–Relational Mapping (ORM) approach, allowing to perform database queries through python code instead of Structured Query Language (SQL) queries. Django can then check if there are new models or if existing ones have changed. Such changes are then expressed through database migration files which will apply them to the database tables. Next, the developer can set custom URL patterns so specific requests are handled by a specific function of the view layer, where the business logic will be implemented. Finally, Django contains a template layer that allows building dynamic HyperText Markup Language (HTML) pages without requiring to have a separate javascript framework to do such. The developer builds the main static structure of the page and then uses a special syntax that describes how Django should display the data received from the view layer. Django also

---

[1]https://github.com/bioinformatics-ua/montra

has an important form feature that allows to easily create a set of pages that allow performing the usual Create, Read, Update and Delete (CRUD) operation over the database model.

MONTRA's development started at the beginning of 2013 and ended in the middle of 2018. After this date, at the end of 2018, the framework started being used by the EHDEN project, to develop a portal to allow discovery and analysis of health data of a federated network of data sources standardized to the OMOP common data model in Europe. Currently, the project is being developed in a private repository but has intentions to make it public after the code base is more robust and well documented.

3.1.1 Communities

In the first versions of the framework, MONTRA allowed only one level of organization related to data sources, in which they could only be separated according to their skeleton that describe their original data, which will be more detailed in the next subsection. Newer versions created the concept of a Community. This allows having multiple networks of data sources on the same portal, where originally the only option was to have different installations of the framework.

These communities can be created in several ways:

1. the admin can create it through Django's admin console
2. a user can request a community to be created through a form and then the admin will receive an admin with such request
3. the MONTRA installation can be deployed in a single community mode where only one community exists, which is created on the first setup, giving the idea that there is no community concept on the platform

They also can have different access levels:

- Open: Does not require membership. Any user can access the data sources of this community
- Public: Does not require a membership however, the user needs to accept a set of terms and conditions before being able to access the data sources
- Moderated: A user has to request the community managers for approval
- Invitation: Users can only access and see the community by invite and subsequent approval

Plugins

The concept of Community also allows customization at that level, affecting only sections within that given community. One example of such customization is plugins, a way to extend the functionalities of the framework without having to deal with the base code. These plugins can either be full web applications, with different functionalities, that are linked to MONTRA through the community navigation menu or extensions that provide extra data services such as a dashboard about the data of a data source.
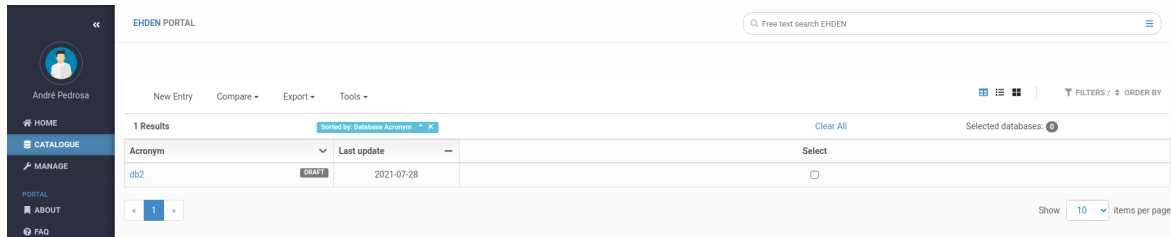
3.1.2 Questionnaires

As the data from different data sources is highly heterogeneous, MONTRA ensures that the data inserted within a given community follow a common structure. This structure is called a skeleton, which is represented in a form of a questionnaire with a set of questions, which can then be grouped in sections called Question Sets. It represents a set of metadata that better describes the original data of data sources that need to remain private. The skeleton schema can easily be defined through a spreadsheet, which will be more detailed on subsection 3.1.4.

Next is presented the available question types which can be used to build a questionnaire for a community

Type	Description
choice	Single choice (radio box)
choice-freeform	Single choice with an open text fields
choice-multiple	Multiple choice (checkbox)
choice-multiple-freeform	Multiple choice with and open text fields
choice-multiple-freeform-options	Multiple choice with and open text fields
choice-tabular	Creates a table with single, multiple choices or text by row
choice-yesno	Single choice with yea and no choices
choice-yesnodontknow	Single choice with yes, no and don't know choices
comment	Used to separate groups of questions
custom	Mirrors another question by its type
datepicker	Field with date picker widget
email	Text input with email validation
numeric	Numeric input
open	Text field with no validation
open-button	Text input with backend validation
open-location	Text input with autocomplete sugestions
open-multiple	Allows to record an history of a value overtime
open-multiple-composition	Allows to record an history of several values overtime
open-textfield	Same as open but a textarea html tag is used
open-upload-image	Image upload
open-validated	Text input with a regex validation
publication	A custom widget that allows to attach a set of publications
sameas	Mirrors another question by its number
timeperiod	Numeric input + select to choose numeric unit
url	Text input with url validation

Table 3.1: All available question types that can be used to build a questionnaire

Figure 3.1: The user interface displayed after selecting a community, which shows the list of the fingerprints of the chosen community
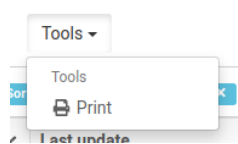
3.1.3 Fingerprints

A fingerprint is a name given to the set of answers to the questions of a questionnaire. In other words, it is the metadata that betters describes the original data of the associated data source. Data owners can start to answer the questions to build the profile of their data sources once there are communities on the platform and, these have at least a questionnaire associated.

On the list presented in figure 3.1, we can notice that the only fingerprint present is marked as draft. This is a state of the fingerprint which prevents from non-ready or non-approved fingerprints won't show to the regular community users. With this, for such users the list showed above would be empty. Fingerprints can be published depending on the chosen settings for the community. After the data source owners request to publish a fingerprint the framework allows to either automatically accept or require a community manger to accept it.

Views

In figure 3.3 it is presented the user interface where a data owner can answer the questions of a questionnaire. Each question of the questionnaire is placed under a container that can be collapsed, as is shown for question *Instituition name*. However, if multiple questions are grouped, the collapsable container will affect all questions of the group. Besides the input widget where the data owner can insert its answers, the user also has some additional control buttons:

1. Allows to Hide or Show Questions that have been answered or that are empty. Besides being an interesting feature is important to note that on the last version it does not work, as clicking on the presented options will result in no visual effect
2. Allows to collapse or expand all questions or question groups containers
3. Allows the data owners to set permissions at the question set level
   - Visibility: Let plugins have access to answers data
   - Allow printing: On the fingerprints list page, showed in figure 3.1, there is a dropdown with tools, being the only one the "Print" tool (3.2). However, this feature is not correctly implemented since it calls the browser's built-int printing function on the fingerprint list page, so no actual fingerprint data will be printed. This permission ends up being useless. Additionally, if a user calls the browser's print function (e.g. hitting Ctrl+P) when viewing the data of a fingerprint, the platform will not block the action.

**Figure 3.2:** Available tools on the fingerprint selection

Figure 3.3: User interface to create a fingerprint

- Allow indexing: If the data owner allows indexing of the answers, which will allow for other users to find fingerprints based on the answers to a question of this specific question set.

- Allow exporting: If the answers to this question set can be included on the export file of a fingerprint.

4. Enable navigation along the question sets of the current questionnaire

5. Permits to save or cancel all the changes made to the current question set

Once the fingerprint is filled and published, a regular user can consult the metadata, which will be displayed by default in detailed view. Similar to the create view, it is presented with some control buttons:

1. Database level plugins associated with this community

2. Statistics of this fingerprint

   - Progress bar + Filled: How many questions of the questionnaire were answered

   - Hits: Number of times this fingerprint showed up on search queries

   - Unique Views: Number of users that visited this fingerprint

3. Question set controls

   - Summary: Allows to switch to the summary view (Figure 3.5)

   - Collapse & Show: The same role as mentioned for the create view

4. Fingerprint control buttons
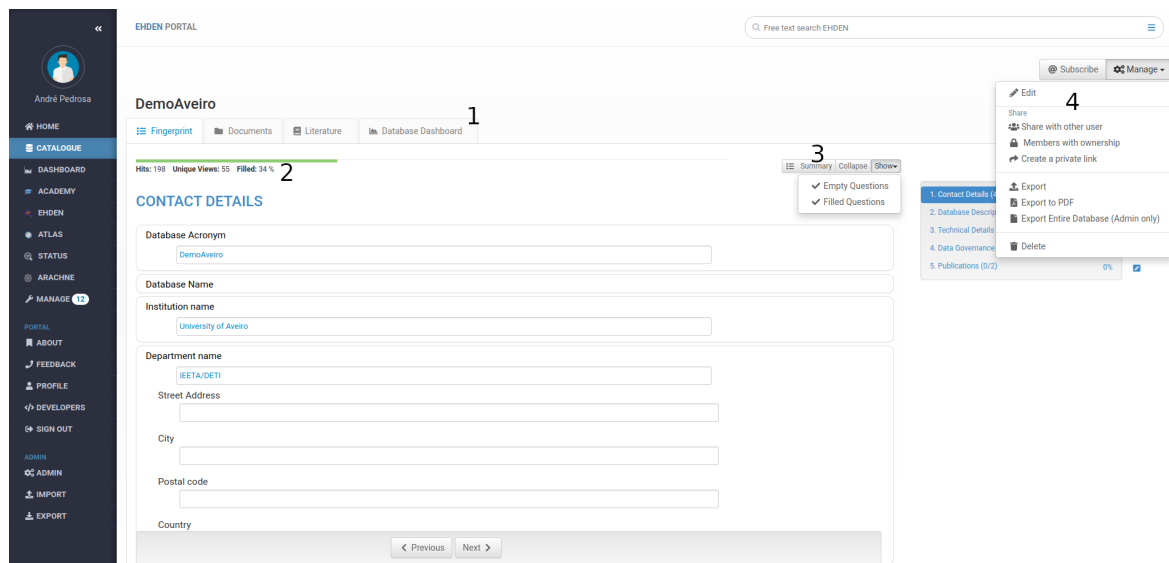
Figure 3.4: A detailed version user interface to view and analyze fingerprints

- Subscribe: Receive notifications whenever changes are made to the fingerprint answers
- Manage: Several fingerprint operations
  - Edit: Enter the edit mode
  - Share: Allows to add other users as owners of the fingerprint and also to create links that enable anonymous users to consult the fingerprint
  - Export: Different forms of export. CSV, PDF, and MONTRA format to import on other installations of the MONTRA framework
  - Delete: Remove the fingerprint from the community

On the show view, if the user enters the summary view (Figure 3.5), the user is presented with a table with three columns where each row contains the question number, name and the answer given. On this view, by hovering over an empty answer container the user can send a request to the database owner to answer the specific question.

The MONTRA framework also offers an "Advanced Search" feature, allowing to perform searches for fingerprints. The overall interface used is identical to the ones presented above, where the user answers the questions of the specific questionnaire of the community. The main difference to the other versions of the fingerprint view is that the only control buttons available are just the navigation ones, and all the questions don't have any validation. Additionally, at the bottom of the page, it is provided to the user a way to customize the search query, allowing to create complex search criteria through a drag and drop interface, as is presented in figure 3.6.

Despite all these variations of the same view having a similar look, as some components appear on several variants, all views have a separate Django template, so there is some duplicated code across the different views. If some changes are made to a shared component, such as the side question set menu bar, those changes have to be applied to all different

24
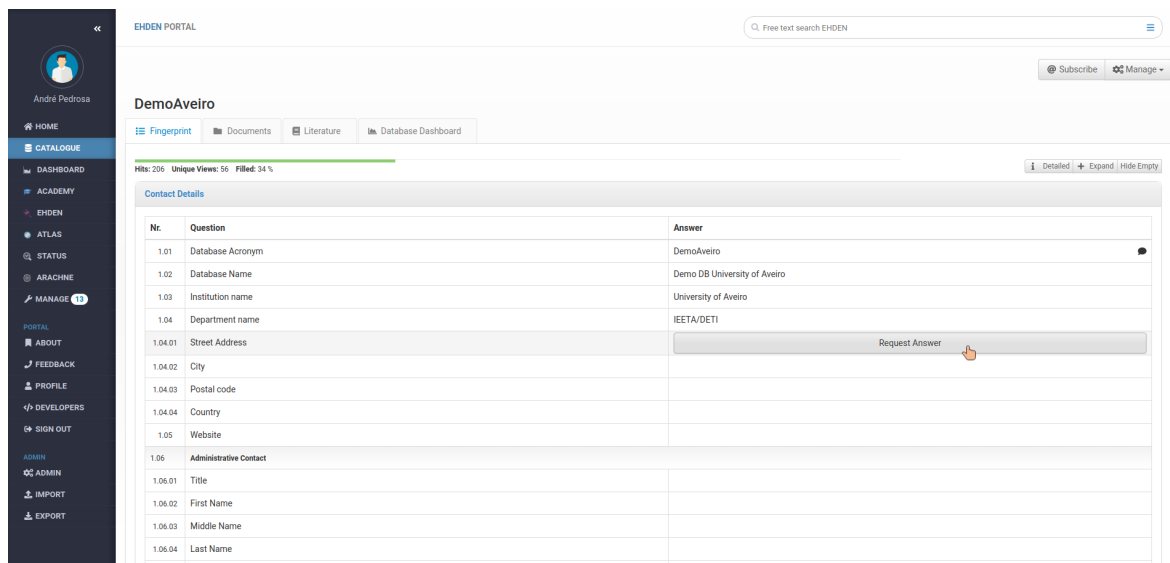
Figure 3.5: A summary version user interface to view and analyze fingerprints
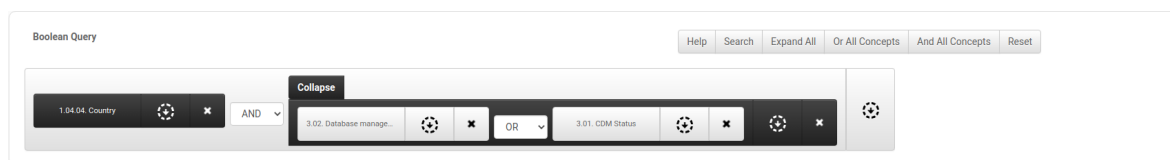
Figure 3.6: Interface to customize the fingerprint search query

templates. This can be avoided since the Django template system allows to both include a shared component into other templates and also supports conditional rendering, for example, the permissions dropdown for a question set of a fingerprint should only be rendered when the user is editing or creating a fingerprint.

These different view versions provide a valid workflow to perform CRUD operations over the metadata related to a data source, however, views that are used to insert or edit metadata of a fingerprint present several flaws related to how the inputs are validated and how they are presented to the user.

First, the validation of the user input is done on the client-side through javascript code that runs after a user submits a question set form. Subsequently, the data is sent to the backend through API calls. However, if one would use the API directly to update metadata of the fingerprint, the validation could be skipped and invalid data could be stored on the database. In some cases there might exist some validation code on the server-side, however, this brings the necessity to maintain two separate code files.

Second, there is no escaping procedure done to the user's input when it is fetched from the database which allows that Cross-Site Scripting (XSS) attacks can be easily performed and put users that consult a compromised fingerprint at risk. As an example, on figure 3.7 on the Database Name field, I added a *script* tag with code that shows a popup and also a valid database name after. Once a user opens the fingerprint to check the data, the code will execute but the dummy name will render on the Database Name field. This can be

Figure 3.7: Example of how an XSS attack could be done on MONTRA

Figure 3.8: Victim of a simple XSS attack

further exploited, where the user visiting the fingerprint will not notice that malicious code was executed.

All these problems exist because such views were developed from scratch without using the provided features that Django has out of the box for form validation and security. For client-side validation, Django takes advantage of HTML5 form validation features [37]. This allows imposing restrictions and validations on the user's input without writing any additional javascript code.

To show these features I wrote a simple HTML file with a simple form that expects a number under 100 and an email.

```html
<html>
  <body>
    <form>
      <label for"num">Number:</label>
      <input id="num" type="number" max="100">

      <label for"mail">Email:</label>
      <input id="mail" type="email">

      <button type="submit">Submit</button>
    </form>
```

Figure 3.9: Error messages that appear after submitting the form on a chromium based browser

```
    </body>
</html>
```

If I try to submit the form with invalid values, error messages are presented as shown in figure 3.9.

Additionally, if the data is sent directly through an API call to the backend, Django forms framework ensures invalid data is rejected. With this, when building a form in Django the developer only needs to specify what fields are present and their type and restrictions and Django will validate all this before data is stored on the database.

Finally, XSS attacks are prevented because Django escapes the values that were previously provided by users when filling the input tags, e.g. the character "<" is transformed in "<", avoiding the browser to interpret user's input as HTML code.

Another problem associated with the views of the MONTRA framework is how the management of javascript dependencies is done. There is no consistency on how such dependencies are imported. Some are imported directly on the template through *script* tags making use of a Content Delivery Network (CDN), as others their entire source were added to the repository and are then referenced based on their location. The first approach has the advantage that an upgrade is simply changing the URL attribute of the *script* tag and also reduces bandwidth from the Django server handling the user requests. The second approach increases the overall size of the repository, and the upgrade process will lead to more changes on the repository. These processes, however, have the problem that easily unnecessary dependencies will still be present on the Django templates after being used on the application. Currently, there are tools such as Node Package Manager (NPM)[2] that help to manage these dependencies by defining, on a single file, the dependency and its version. This software will then download the dependency and subsequent dependencies.

API

It was mentioned several times that some checks are not enforced through the API. It will be detailed now what calls are performed by MONTRA's fingerprint views.

Note that there are two different APIs available here. Fingerprint views use one specific set of API requests that return answers in HTML format, ready to present to the user, and others to send the user's data. For a regular user to use, there are other set o API endpoints

---

[2]https://www.npmjs.com/

27

Figure 3.10: Each question set has its container. Here the current container is presented within the red rectangle. The other existing question sets are represented through the blue lines, which currently are hidden.

that are more human friendly, which is documented and has a How to Use page, however, an advanced user can see what requests are made on the fingerprint views through browser's console and perform the requests themselves.

First, we will go over the API used by the fingerprint views. But before showing the available endpoints, let's go over how the fingerprint views are organized and how they show only the questions of a certain question set at a time.

Let use figure 3.10 as example. According to the right side menu, there are six question sets and currently the question set "Contact Details" is being presented. In the middle of the page, we then see the content of the question set: questions, title, control buttons, and permissions. Note that there is a scroll bar so the question set contains more questions. Also the previous, next, cancel and save buttons are not tied to the question set. The other question sets are also present on the page, however, are hidden. Every question set has its container, so whenever the user clicks on the previous or next buttons or selects another question set on the question set side menu bar, the current question set container is hidden and the new is shown. For questionnaires with a high number of question sets, rendering all available question sets could not be necessary. For that, a question set is loaded only when accessed the first time and following accesses will not require a load.

The fingerprint view can be used for four different use cases:

- Create a new fingerprint: The questions are presented with clear and editable inputs
- Edit an existing fingerprint: All questions are editable but the previously answered questions are filled
- View the answers of a fingerprint: A read-only version of the fingerprint's answers
- Search for fingerprints: Same as creating a new fingerprint, however, no validations are performed on the user's input

For these use cases, the following endpoints are used:

- Create

  ```
  GET [base url]/c/[community slug]/addqs/[fingerprint hash]/[questionnaire id]/[question set id]/
  POST [base url]/c/[community slug]/addPost/[questionnaire id]/[question set id]/[save id]
  ```

- Edit

  ```
  GET [base url]/editqs/[fingerprint hash]/[questionnaire id]/[question set]/
  POST [base url]/c/[community slug]/addPost/[questionnaire id]/[question set id]/[save id]
  ```

- View

  ```
  GET [base url]/detailedqs/[fingerprint hash]/[questionnaire id]/[question set id]/
  ```

- Search

  ```
  GET [base url]/c/[community slug]/searchqs/[questionnaire id]/[question set id]/
  ```

For both cases where new data is stored, besides the usual GET that is used to load the question set, there is also a POST request that saves the progress done to a specific question set. The data for these requests are gathered natively by javascript once each question set container contains a *form* tag englobing all the questions inputs. This way there is no need to iterate over the questions of a question set and append each response to the request's data. The *save id* field of the endpoints tells to which question set the data is related to, however, there is already a *question set* field which always has the value of 1, so one of these fields could be removed from the endpoint.

The GET request is then used to load the HTML to present the questions of a question set. Note that the returned data is just the HTML data to be placed on the respective question set container and not the whole fingerprint page.

Moving now to the set of API endpoints available to the users to both read and update answer data the following are available:

```
GET /api/fingerprints/[fingerprint hash]/answers/
GET /api/fingerprints/[fingerprint hash]/answers/[question slug]
PUT /api/fingerprints/[fingerprint hash]/answers/[question slug]
```

The first two GETs are used to retrieve answers data, which is returned as the JavaScript Object Notation (JSON) object presented next, the only difference being the first one returns an array of the mentioned object instead of just one.

```
{
  "question":"patients_count",
  "data":""
}
```

With the PUT request, a fingerprint owner can update data of specific answers where a JSON object should be sent in the body of the request with the field "data".

```
{
  "data": 10000
}
```

Existing two different types of endpoints to retrieve answers data makes sense since the returned format is returned in a way that is easier to handle data by the target entity that will consume that endpoint. If the fingerprint pages receive the data in HTML format they can just put the HTML in the determined container instead of having to build the entire container and insert the data on each input. Accordingly, if data is returned in a JSON format it can easily access the data and perform some data processing avoiding going transverse the HTML and retrieve the data from each input. However, having two different endpoints to update data doesn't make that much sense since current HyperText Transfer Protocol (HTTP) requests libraries offer an easy way to build a request in the required format as the one's browsers automatically build whenever a form is submitted.

Draft

We will also go over the API endpoints that the front end code uses to request to change the fingerprint state from draft to published. Associated with this feature, there are two endpoints available:

```
POST [base url]/api/pending/[fingerprin hash]
POST [base url]/api/draft/[fingerprin hash]
```

The existence of two endpoints for the same purpose is because Communities have a setting that allows to auto-accept requests to publish a fingerprint. For that, whenever the auto-accept setting is off, the first endpoint must be used, which will send a request to the community owners to publish the given fingerprint. The second must be used otherwise.

The problem with this approach is that the code that decides what endpoint to use is on the client-side and there is no server-side check if the correct endpoint is being used. If the auto-accept setting is off and the second endpoint is used, a user can publish his fingerprint without requiring the approval of the community owners.

3.1.4 Import Questionnaires - Excel

As mentioned before, to define a skeleton of the metadata that describes a data source for a given community, a spreadsheet file has to be submitted. There is already a template with some instructions and columns where a community manager only has to fill the necessary rows to then get the wanted result.

The column defined in the template are the following:

- Type: the type of the specific row. Here are allows 3 different values:
  - QuestionSet: allows dividing the questionnaire into several sections
  - Question: a question specification
  - Category: allows to create a group of questions inside a question set
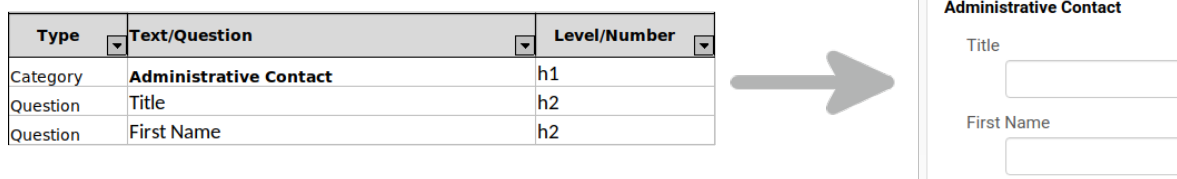
Figure 3.11: Create a group of questions with a title
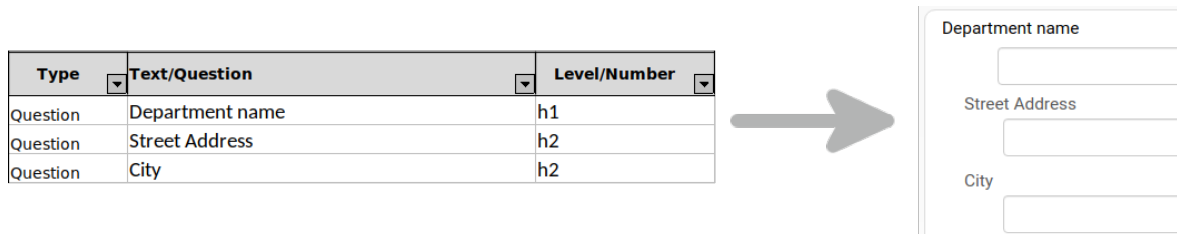
Figure 3.12: Create a group of questions using a question's text as the title

- Text/Question: label/name given to the item being defined
- Level/Number: use as a level for questions and categories and number otherwise. As level allows to create groups of questions inside a question set. As number defines the number, and subsequently the order, of the question sets.
- Data type: used only for rows of type Question, specifying the question type
- Value list: used to indicate extra information to build the question
- Help text/Description: a small text that will be displayed along with the item being defined
- Tooltip: Yes if the Help text/Description should be displayed as a tooltip or No otherwise
- Slug: internal identifier
- Dependencies: used to tell that a question or group of questions can only be answered if a specific choice of a choice-based question was selected
- Stats, Comments Stats and Disposition: Columns that were used for old features but are still present on the spreadsheet
- Include in Advanced Search: if the answers of the question can be used to search fingerprints

Question Groups

From the previous list, question groups were mentioned both when the type column has the Category value and on the Level part of the Level/Number column. The former is used to add a title with no question associated, resulting in what is presented in figure 3.11.

On the latter, the text of the question in the most upper level is used as the title of the questions group, resulting in an output similar to the one in figure 3.12.

It's important to highlight that the category way to create question groups is not independent of levels. If both a category row and a question are on the most upper level, MONTRA will render two separate collapsable containers, where the first one will be empty, as is shown in figure 3.13.
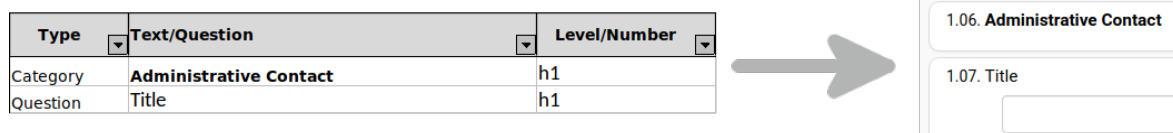
Figure 3.13: Example to show that the category way to create question groups also depends on the values that are set on the level column
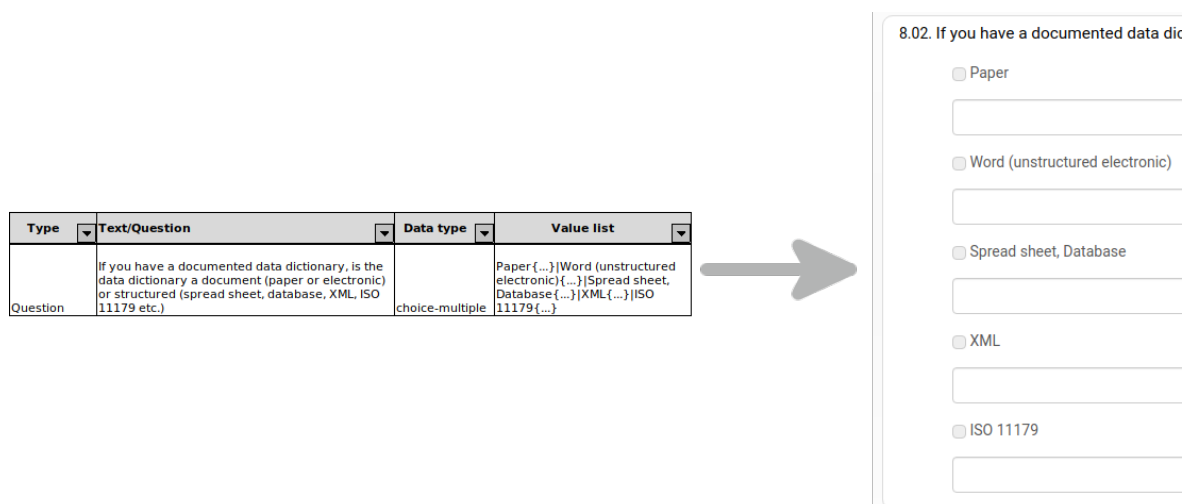
Figure 3.14: Multiple choice question with some extra text fields associated with the several choices

Value list

To avoid having a spreadsheet with several columns that will not be used for all types of rows, the value list column expects some extra information required for some types of questions.

Choices

For choice-based questions, the value list column is used to define the possible choices and to add an extra text field associated with a given choice. Choices are separated by a "|" character and the extra text field can be set by appending "{...}" after the target choice's text.

Although the framework allows the extensibility to have an additional text field, these don't support other types of input and also have no validation. Also, if the number of choices is high and the text of them is long, there starts to exist some clutter on the spreadsheet. With this, the person creating the spreadsheet will have problems perform edits and check if there is something wrong or missing.

To facilitate the job of who is filling the spreadsheet, some question types are shortcuts. For example, the choice-yesnodontknow is a question of type choice with three possible options: Yes, No, Don't Know. Choice variants with freeform on the name, besides the usual choices, always have an additional text field, associated with the question instead of a choice.

Type	Text/Question	Data type	Value list
Question	Clinical test parameters results.	open-multiple-composition	{title:"Weight X", field:"msurex", editor:"input"}, {title:"Height", field:"msurey", editor:"input"},{title:"Msure Z", field:"msurez", editor:"input"}

Figure 3.15: How an open-multiple-composition question type is specified on the spreadsheet and how it is rendered

Open Multiple Composition

Open multiple questions allow showing a history of a given value. For that the simple version of the question is represented in a table of two columns: Date and the value, so no input is expected on the value list column. The composition version of the question type allows having several values, instead of just one. In a way, the simpler version can be used as a shortcut question type, since it can be reproduced with the composition variant.

To render this question type, a third-party widget called Tabulator[3] is being used. The configuration of such widget, expects an array of JSON objects, each specifying some configuration of each column. To configure the open multiple composition question type, the value list column expects these JSON objects, where the date column is implicit. The MONTRA framework will then put the provided objects on the configuration array that the Tabulator widgets expects, adding the configuration for the date column.

Choice Tabular

This type of question allows reusing the same choices across several answering items. There are three variations of this questions types, where the difference between them is what and how is the information connected between answering items and choices. There are two versions where the user can select one (single choice) or more (multiple-choice) choices for each answering item. The other version allows the user to write text for each choice within each

---

[3]http://tabulator.info/

Figure 3.16: How a tabular-choice question type is specified on the spreadsheet and how it is rendered

Figure 3.17: An example of both a question and a category that depend on the selected value for another question.

answering item. The question type is rendered as a table where in the columns are displayed the several choices and on the rows the different answering items are presented. Additionally, there can be a "More" choice column, where the user can insert any text information for a specific answering item since is displayed with a textarea HTML element.

The value list column of a choice tabular question type expects a three-component value. Each component is separated by the characters "\\". Within each component, items are separated by the "|" character. The components are the following:

- choices (columns)
- answering items (rows)
- type of the information: available values are choice, multiple-choice and text

Once again, we encounter the clutter problem, as shown in the spreadsheet section of figure 3.16, which hampers readability and update of the value list field.

Dependencies

In several form applications, e.g. Google Forms, it is common to find situations where if a certain choice is selected, then other questions will show up. E.g. If a user answers "Yes" to the Question "Have you visited Aveiro?", then other questions such as "Would you recommend it to a friend?" would appear.

MONTRA supports this kind of dependencies, which should be defined on the Dependencies column of the questionnaire spreadsheet.

As shown in figure 3.17, first you should indicate the slug of the question on which the specific question depends. Then, separated by a "|" character, it must be specified which choice has to be selected for the dependency to be fulfilled, using its index, starting at 1. On

Figure 3.18: The questions with dependencies are not rendered if the specific choice is not selected

Figure 3.19: Once the dependency of a specific question is meet it will be rendered

figure 3.17, since a choice-yesnodontknow question type has three implicit choices, by having a dependency with the value "Any_blood_collected|1", both the *Blood collection* category and the *Plasma* question will only be displayed to the user once the choice *Yes* of the *Any Blood Collection?* question is selected.

Besides such restrictions are imposed on the user when he visits the web page, if the API was used directly, such dependencies would not be checked, which would lead to unnecessary data be stored on the database since it won't be displayed to the user.
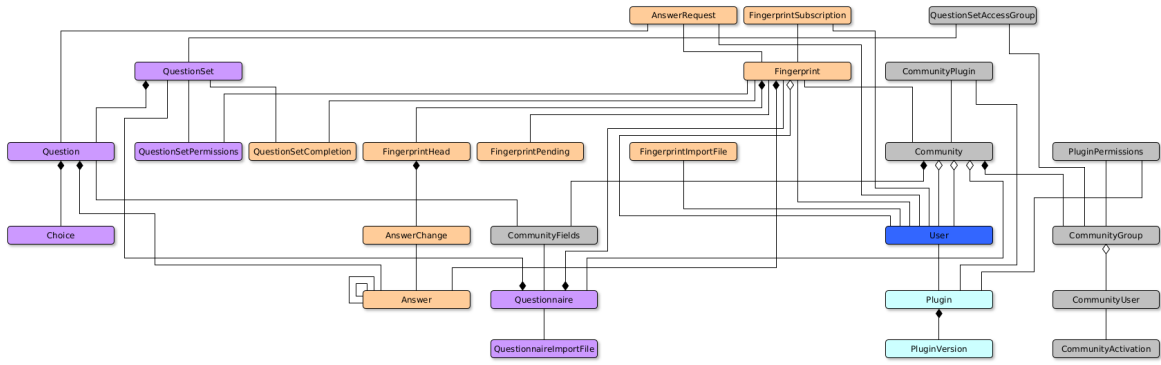
3.1.5 Data Models

This section will be presented how the previous concepts are mapped to database models. Taking advantage of Django's ORM feature, MONTRA's data models are defined as python classes that extend Django's base Model class. These Model classes belong to different applications, which are associated with distinct aspects of the platform.

In figure 3.20 is presented the class diagram of the classes that are related to features and/or concepts that were mentioned in previous sections. Classes with the same color belong to the same Django application, dividing them into specific purposes.

- Blue - Django's built-in authentication system[4].
- Orange - Fingerprint application: Answers to questionnaires questions and other models associated with features that were mentioned on the fingerprint views section, such as AnswerRequest and FingerprintSubscription. MONTRA also keeps a record of all the states of a given Fingerprint. For that, it uses the FingerprintHead model, which maps to a set of AnswerChanges.
- Purple - Questionnaire application: Questionnaires structure information such as question sets, questions, and choices, and import spreadsheets logs. Additionally, associated with each fingerprint, contains a model with the allowed permissions on each question set.

---

[4]https://docs.djangoproject.com/en/1.11/ref/contrib/auth/

Figure 3.20: Class diagram of MONTRA's Model classes. Each color has a Django application associated. Gray: Community; Purple: Questionnaire; Orange: Fingerprint; Light Blue: Plugin; Blue: Django Auth. MONTRA's data model is much more complex, however, it is just presented the ones that impact the features and/or concepts mentioned previously.
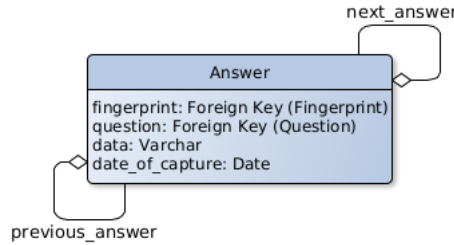
Figure 3.21: Detailed information of the Answer model of the Fingerprint application

- Light Blue - Developer application: Allows adding customizations to different MON-TRA's installations through plugins.
- Gray - Community application: Community's groups, users and access permissions, fields to be presented on the fingerprint list page for each questionnaire and plugins.

Going into more detail on some models we can see some poor design decisions.

Regarding fingerprint answers to a questionnaire, all the data is stored in the Answer model on the data field, which is a variable-length string field type. Although this approach is much simpler in terms of data models, it leads to two problems:

1. this is not the most optimized way to store all the data types. Some question types expect numeric values and other date values, which could use built-in field types of a relational BDMS.

2. for complex question types which the answer contains several fields, before and after storing such data in the database, some processing has to be made to convert the data to the necessary format. One example of this is multiple choice questions, where the value of the several selected choice is joined in a single string to then be stored on the database. Every time the answer needs to be displayed to a user, it is necessary to split that string by its separator. For this situation instead of storing the choice's value, the models of the questionnaire application could be used as a foreign key.

Related to MONTRA itself, when a user provides no data to a specific question, an empty string will still be sent for that question on the submission of the answers to a question set,
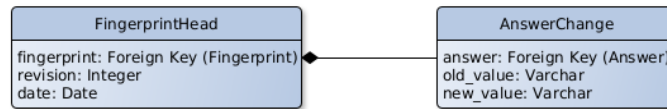
Figure 3.22: Models that store the changes to answers of fingerprint
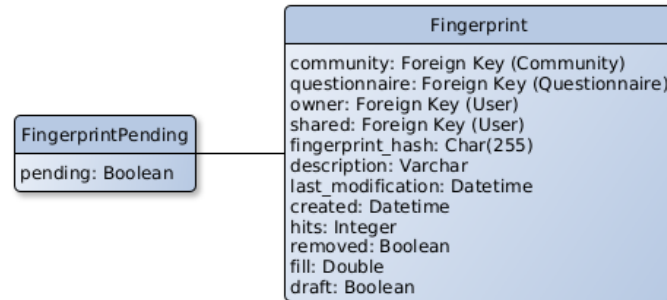
Figure 3.23: The Model that stores the information telling if a fingerprint is waiting to be approved to be published

which will create unnecessary records on the database.

As mentioned previously, MONTRA records the history of submissions for a specific fingerprint. Each submission has an associated FingerprintHead record, which its name might have been inspired by the HEAD concept of Git[5], a version control system. For every FingerprintHead there is a set of answers that suffer changes which are then recorded on the AnswerChange model. In this model, we can get the answers data duplicated three times since it is already stored on the Answer model and can also be stored again as text in both *old_value* and *new_value*.

As mentioned previously, a fingerprint is not immediately available to all regular users, since it first enters on a draft state. To transit to the published state, a request needs to be made to the community owner. Such requests are stored on the FingerprintPending table, where the pending value will be *true*. Once a request is rejected or accepted, the value of the pending value is changed to *false*.

On the Fingerprint model, there is also a *draft* value that indicates if the fingerprint is published or not. Once a fingerprint is published, the value of the *draft* will be true, and on the FingerprintPending table, the associated record will have the pending value at *false*. However, this value is not required to be stored on the database since after a fingerprint is published, the fingerprint will not be pending therefore, the associated FingerprintPending record could be deleted.

Whenever a questionnaire is imported, a QuestionnaireImportFile record is created containing the information of the uploaded file and the user that uploaded it. Also contains a status field to give some feedback to the user. Associated with every import will also be created a Questionnaire record. It contains an in_preview variable that is set to *true* whenever a questionnaire is in the import process. If it fails to import, only the QuestionnaireImportFile

---

[5]https://git-scm.com/
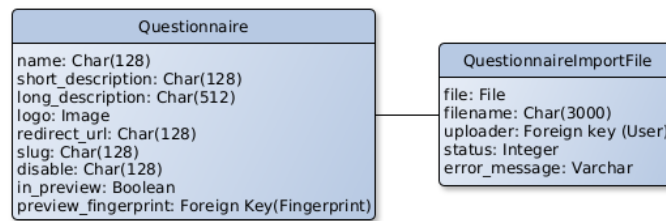
Figure 3.24: Questionnaire model and the model where questionnaire imports information is stored
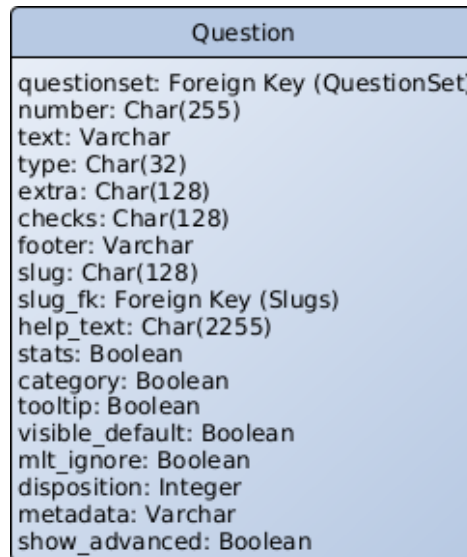
Figure 3.25: Question model and its extensive number of fields

record will be kept, which the status will change to *Failed* and an error message will also be attached to the import record.

If the questionnaire is valid, the user will be redirected to a page where he can preview the result and can accept or reject it. It is important to point out that for the preview process a new Fingerprint object is created so API calls can be performed.

Strangely on the Questionnaire model, the disable column uses a character type column instead of a boolean one since it expects only the value of "False" and "True". If there are some user input errors, it could lead to unexpected behavior or internal server error.

Associated with how the structure of a questionnaire is stored, there are only four models to do so: Questionnaire, QuestionSet, Question, and Choice. All are associated with a specific concept of the questionnaire, although other concepts are not represented. First, there's the category, used to create question groups within a question set. It is represented as with a question record, where the "category" field of the question models has the value "true". For more complex question types that require extra configuration such as choice tabular, which requires information about the name of the rows and column, and open multiple, which expects the configuration of the columns, such data is stored in a metadata field of the associated question model. Question types that do not make use of these fields, will have an empty value, leading to some database space being wasted.

38

When the questionnaire's spreadsheet was explained, some fields were related to some old deprecated features. The question model contains the same fields that map to the questionnaire's spreadsheet deprecated fields: stats, visible_default, and disposition. There is also a slug_fk field that points to a Slugs model that replicates the question's slug and text, which is was used to yet another deprecated feature.

Finally, a question supports for the user to define additional checks which are defined on the "checks" field however, this feature is not exposed through the questionnaire's spreadsheet.

3.2 REFACTORING

Until now it was shown how the MONTRA framework was designed around databases, their metadata, and how that metadata can be shared among the platform users. For regular users the framework provides a good user experience, however, for power users, that make use of APIs, might make some errors which the framework will not prevent, leading to inaccurate data to be stored and presented. Also for developers, such flaws and bad design choices make the maintainability process of MONTRA instance a demanding and tedious process.

With this, there is a clear opportunity to perform a refactoring process over the framework to correct such problems and the issues that have been emphasized in this chapter. The goal of this refactoring process is not to change the framework in a way to transform existing use cases, but to improve its internal structure to make the framework more stable, easier to maintain, and easier for an external tool to communicate with it, mainly to publish database data.

Next, we will propose several changes to be applied to the framework, beginning from the data models, regarding how fingerprint answers data is stored, how questionnaires structure is stored and some other fixes for some defects explained previously. Such refactoring will imply changes on other components, one of them being the rendering of questionnaires and how input validation is done. Finally, several clutter problems were mentioned when describing the spreadsheet to define a questionnaire structure, so the spreadsheet will also undergo a refactor.

3.2.1 Data Models

Since the MONTRA framework uses some outdated software and there are active installations, the refactoring process can not be simply designing new models and views, implementing them, and replacing old ones is not a viable option. The approach taken was to first decide what components would go through a refactoring process and within each, until what level we would apply it.

Taking into account figure 3.20, since both the Community and Developer (Plugins) Django applications target functionalities not so fingerprint-centric and are more related to features of the framework as a whole, we decided not to perform any changes on the models of these applications. This leaves us with both the Fingerprint and Questionnaire applications. Regarding the Fingerprint application, the fingerprint model is one of the main models of the framework, affecting several features of the framework, so we intend to perform minimal
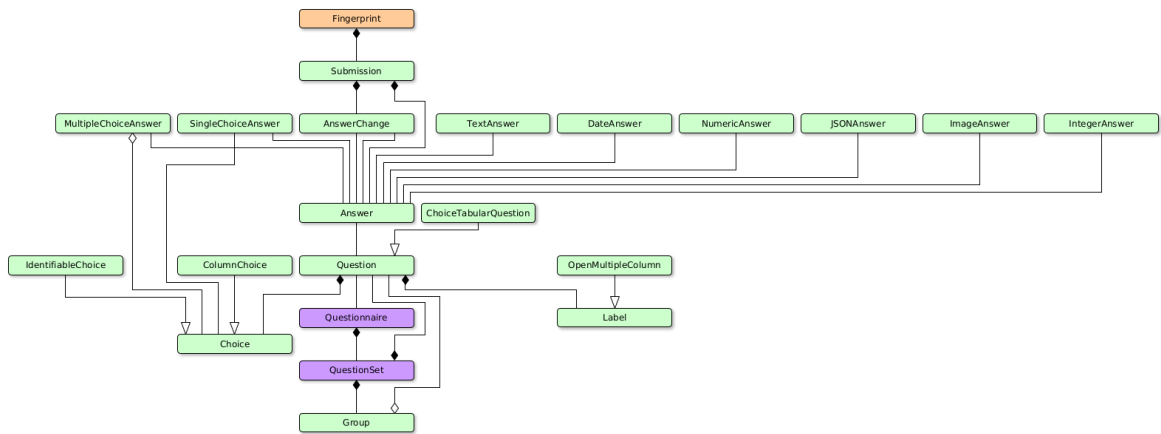
Figure 3.26: Model Diagram of the new models

changes on it. However, the remaining models associated with the answers of a fingerprint and submissions will have a new models design. Respecting the Questionnaire application both the Questionnaire and QuestionSet models are represent well-defined concepts and don't require any changes, yet, the remaining models, mainly the models storing the structure of the questions, their dependencies, choices, etc. will also have a new design.

In Figure 3.26 it is presented the new models in light green. The decided approach to implement these models was to create a new Django application and create all these new models on that application. This way, the migration process is an iterative process where the framework is always operational since the previous models still exist. Then with the help of an Integrated Development Environment (IDE), we can search for usages of the old models and migrate each feature at the time.

Let's start with the fingerprint-related changes. Now there is a new Answer model that doesn't hold directly the content of answers, instead the content is stored on data-specific models such as IntegerAnswer and DateAnswer. For choice-based questions, instead of storing the value of the selected choice(s), a single or multiple choice answer contains a foreign key(s) for the selected choices. To establish the connection between the main Answer model and the data-specific model, the main model contains a *type* field that indicates the data type of the answer. Then the primary keys of the data-specific answer model are the same as the associated record of the Answer model, which allows fetching the data-specific answers of an Answer. Is important to note that the number of different answer types is not the same as the number of question types. Answers of different questions types can be stored in the same answer type models, for example, open text and email questions both can be stored as text in the database. Previously fingerprint submissions were being represented with the model FingerprintHead, now we created a Submission model which contains the same relationships as before, a set of answers, a set of answer changes, all this associated with a fingerprint. The AnswerChange model now does not contain the data of the previous and current answers, instead, it contains the foreign keys for the answer model. In cases where the change was from or to an empty answer, either the previous answer or the next answer field are filled

40

with the NULL value.

Moving now to the models related to the Questionnaire application, there is a new Question model mainly because the previous one had too many fields. An example of these extra fields is the *metadata*, which is used to store the information of the column of the open-multiple question type and the choices and answering items of a choice tabular question type. This field was replaced by the addition of new models (Label) which will be explained in more detail in the next Excel subsection. Another example was the category field, which indicated if a specific question record described a category in the questionnaire which was being used to create subgroups of questions. With the new models, a new Group model was created which has a set of questions associated. Previously this relationship did not exist, so MONTRA would create a group based only on the question's order and level. Additionally, with the addition of this Group model, there is no need to have a level field on the question type, since nested levels are represented through Groups, which can have a parent group. A group with no parent is represented in the root level of the questionnaire. The remaining models (Label, Choice, and their descendants) will be explained in more depth in the next Excel subsection since they were created mainly because of how questions are represented in the new spreadsheet format.

Meanwhile, there are previously existing models that have undergone changes to fix some design flaws mentioned before.

- QuestionnaireImportFile: To provide feedback to the user this model only contained an *error_message* field. However several times there are some errors with the questionnaire, but the framework can continue. In these situations, a warning could be sent to the user just so he is aware that a part of the spreadsheet has some errors and the result could not be his real end goal. Then the new model contains two new fields, *errors*, replacing the old *error_message*, and *warnings* fields, which are stored in JSON, where the keys are the lines where the errors or warnings were found and the values are an array of messages associated with that line.
- Questionnaire: associated with the questionnaire model we mentioned that it had a specific fingerprint attached that was used to show a preview of the result of the questionnaire. This field is not required since the preview mode of a questionnaire is viewed on read-only, so no answers are associated with these preview fingerprints.
- QuestionSetPermissions: From this model, we removed the useless permission of "Allow Printing", considering it is impossible to restrict users from printing the page since a screen capture software can be used as a replacement for the browser's print function. Also, there was a record associated with each question set of every fingerprint, even if the values of each permission were the default ones. The new approach taken is to only create a record if any permission value is different from the default one. Otherwise, an object with the default values is returned.
- Fingerprint: It has a new "submission_token" used for the new API endpoints explained next.

3.2.2 Views

When the MONTRA framework was presented it was highlighted that the different variations of the fingerprint view (create, show, edit, search, preview) were all using a different Django template. Once again, this brings the problem that one change will entail that the same change has to be applied on the remaining fingerprint view variations. On the refactoring process, these views were adapted so all the different pages use a shared fingerprint template which then renders the necessary components according to the page where are being rendered.

The first step of this process is to put different components into separate templates, which will then allow the creation of different arrangements of such components according to the fingerprint view variation being displayed. Note that even within these new separate components, some parts of them might be rendered differently according to the fingerprint view variation where they are being inserted into. In figures 3.3 and 3.4 we can see that some components are present on both views. The question set title, the questions, the question set menu, the fill progress bar, and the navigation buttons. However, we can also see some components in figure 3.4 which are not present on the on figure 3.3 such as the fingerprint statistics (hits, unique views and fill percentage), there are no fingerprint-related buttons, which are now questionnaire-wide buttons. Note that the name of the fingerprint (DemoAveiro), the several tabs (Fingerprint, Documents, ...), and the subscribe and Manage buttons are specific to the show Page. The new fingerprint template will only contain components to display within the Fingerprint tab.
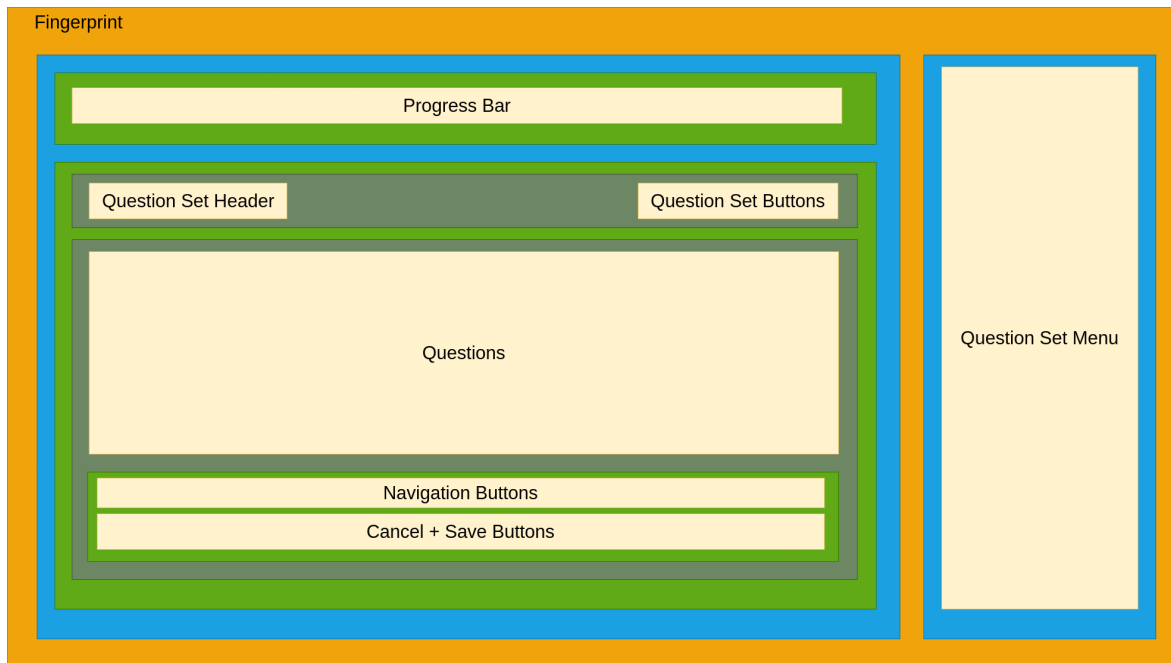
Figure 3.27: New fingerprint template for the create, edit, search and preview fingerprint views variations.

In Figure 3.27 is presented a two-column layout for the create, edit, search and preview variations of the fingerprint view. Although, some components are not displayed in certain

variations, such as the Cancel and Save buttons. The Cancel button is not displayed on the preview variation, and both Cancel and Save buttons are replaced by a Search button on the search button. Additionally, different buttons appear on the question set buttons component according to the fingerprint view variation.
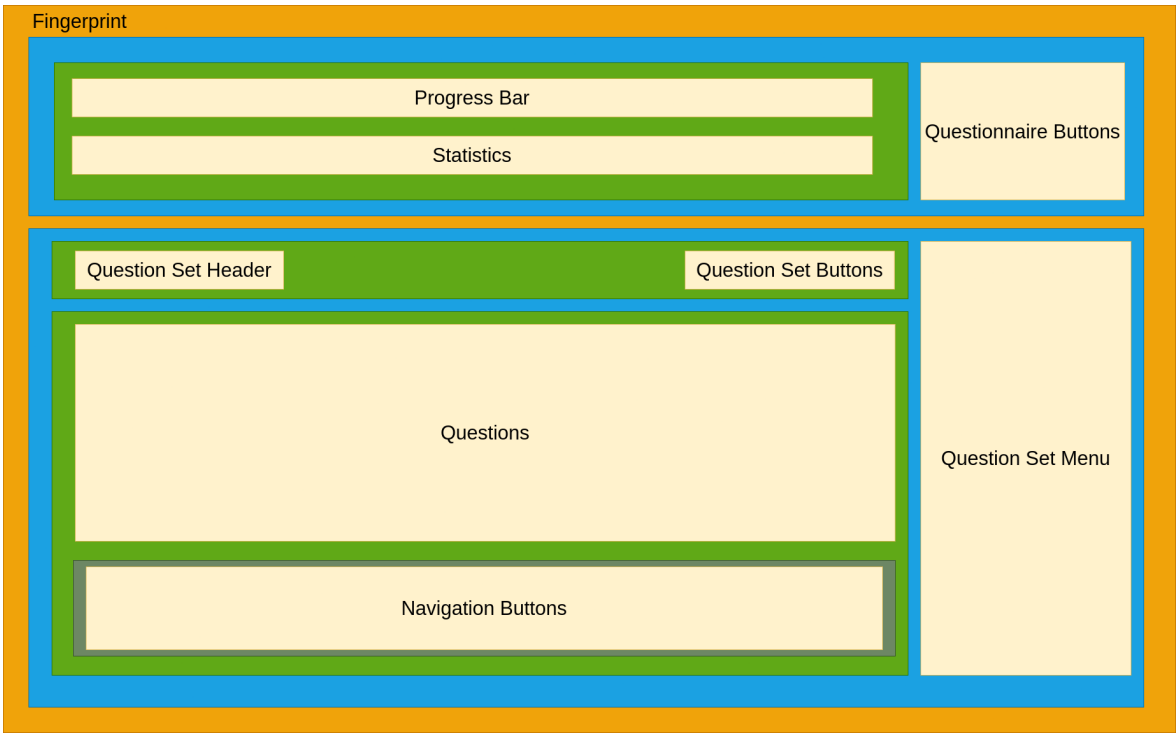
Figure 3.28: New fingerprint template layout for the detailed show fingerprint view variation

The show variation, as shown in figure 3.28, uses a single column with two rows layout. Previously the show variant had no Question set buttons, however, to keep consistency with the other variants, the Collapse and Show buttons were moved to the Question Set Buttons where they were being displayed on the other variants. By default, the only button visible on the Questionnaire buttons is the Summary, which will hide the current detailed view and show an alternative summary layout, where answers are displayed in several tables, one for each question set.
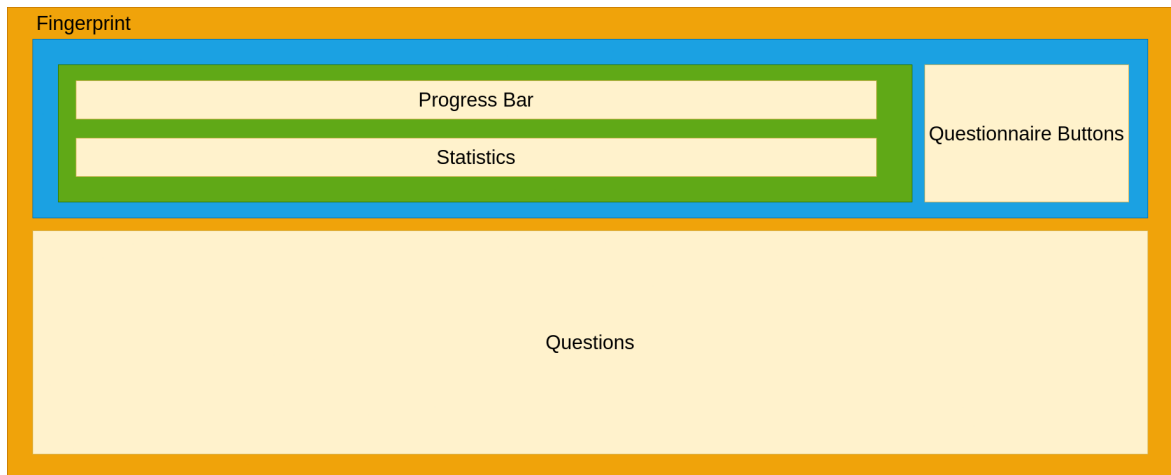
Figure 3.29: New fingerprint template layout for the summary show fingerprint view variation

As can be observed in figure 3.29, when the user switches to the summary view, by clicking on the Summary Questionnaire button, the detailed view container is hidden, and the summary view container is displayed.

It's important to note again that the overall aesthetics and workflow of the page were not changed. The only major change was to divide the fingerprint view into several, reusable components.

When we went over how question sets were rendered, we saw that each question set had its container and they are only rendered when is accessed for the first time. When it needs to be rendered, API endpoints are used to fetch the HTML of a specific question set. On this HTML returned, each question had attached their client-side validation code. The goal is to remove all existing code of client-side validation, making use of only simple built-in HTML validations on the client-side and use Django's forms framework to have all remaining validations on the server-side.

Before going into more details let's go over Django's built-in form system and some of its components. A form in HTML is represented through the form tag and fields are represented through input tags. Each input tag has a type attribute that defines how the data will be inserted by the users, for example, to insert a date the user will be presented with a calendar where it can click on the desired date. These different ways to insert data are called widgets. To work with forms, Django only makes use of two HTTP request methods: GET and POST. GET is used by browsers to fetch the page/HTML of an URL and it can also be used to fetch information. POST on the other side is used to send information to a server. On the backend side, Django has three main classes that handle the data around forms. The main one is the *Form* class that defines how it works and how it will be presented to the user. Then there is the *Field* class that describes a field of a form, which is in charge of performing field-specific validations. Django already has some implementations of this class for the most common field types such as number, text, choice, . . . Finally, the *Widget* class is in charge of processing and transforming the raw data received and also preparing and restructure data to be presented to the user, and again Django already has some implementations of this class. Each *Field*

44

class has a widget class associated. The common development flow is to create a child class of Django's *Form* class and then define its fields with *Field* classes.

However, in the case of MONTRA, the number and type of the fields are dynamic, since both can be customizable by a community manager when he is building a questionnaire. A Submission form was created, child of Django's *Form* class, where its constructor was overridden so the fields of the requested Question Set of a Questionnaire are defined in the form class. Additionally, several question types lead to the need of having to create and associate new *Field* and *Widget* child classes since they were too complex to be represented or validated with the ones that Django already has implemented. To build the widgets of such question types, existing implementations were ported to an associated Django template to then be used in its new widget class.

By removing all validations from the front end, client-side code is now only used to make API calls to the backend, to validate the data inserted by the user, and to add functionality to the page itself, such as hide and/or show elements on the page after a button is clicked. All client-side code related to the fingerprint was either migrated or newly implemented using, as much as possible, plain javascript, avoiding any external library. It avoids adding yet another dependency to the project, and if in the future, upgrades are done to libraries such as JQuery[6], the code implemented here is not affected and does not require any refactoring.

3.2.3 API

As all the validations were moved to the backend, it was necessary to create several API endpoints that perform them. Such validations take advantage of Django's form system, so in the implementation of each endpoint, it is only necessary to build a SubmissionForm then add the data provided by the user and then call its validation method, which will validate all the fields that were defined in the form.

Besides the user input validation, there need to be new endpoints that render the HTML of a question set of a questionnaire using the new questions models and Django's forms system. As different fingerprint view variations might present the questions and answers differently, there are different endpoints for each fingerprint variant:

```
GET  [base url]/questions/[questionnaire id]/[section index]/preview/

GET  [base url]/questions/[community slug]/[questionnaire id]/[section index]/search/

GET  [base url]/questions/[community slug]/[questionnaire id]/[section index]/create/
GET  [base url]/questions/[questionnaire id]/[section index]/create/

GET  [base url]/questions/[fingerprint hash]/[section index]edit/

GET  [base url]/questions/[fingerprint hash]/[section index]/show/
```

There are two different endpoints for the create variant, because certain installations are a single community, for that, the community is implicit, but for multi-community installations, the community slug is required, since the same questionnaire can be used on different communities.

---

[6]https://jquery.com/

45

For the edit and create variants, there need to be additional endpoints that are in charge of performing all the validations associated with each field and saving the progress when the user changes from one question set to another if no error is found. Bellow are such:

```
Submissions Management:
POST [base url]/api/submission/save/[community slug]/[questionnaire id]/[section index]/
POST [base url]/api/submission/save/[questionnaire id]/[section index]/
PUT [base url]/api/submission/save/[fingerprint hash]/[section index]/


Questionnaire Information:
GET  [base url]/api/questionnaire/info/[community slug]/[questionnaire id]/[section index]/
GET  [base url]/api/questionnaire/info/[questionnaire id]/[section index]/
```

Related to the Submissions Management endpoints, which are used to save answers data, the first two endpoints follow the same idea as the ones presented before, which on single community installations there is no need to provide the community slug. However, there is a third alternative. The first two should be used when there is not yet a fingerprint created, and which will return both the fingerprint hash of the created fingerprint and the current submission token. The fingerprint hash is the identifier of a fingerprint, the submission token is used to identify if different calls to the save endpoints are related to the same changes. With that, several changes can be grouped in the same Submission, instead of the previous implementation that would create a FingerprintHead record for each save of a question set.

This also enforces that if the user performs several changes to the same question set with the same submission token, only the last changes will be kept, replacing (figure 3.30), or even deleting (figure 3.31), old answer values sent in the same submission.
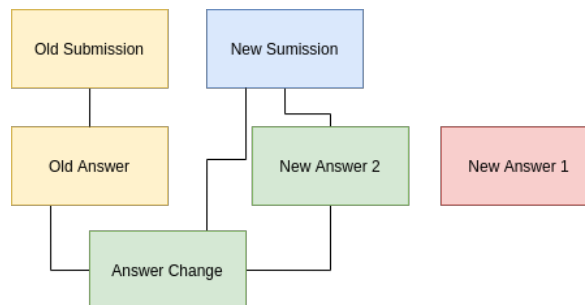
Figure 3.30: When an answer change is submitted and there were already changes made to that question, then the previous one is deleted (New Answer 1) and a new one (New Answer 2) is associated with both the current Submission and the related Answer Change.

Figure 3.31: If the user either provides an empty value or no value to an answer that previously had a value on the current submission, associated Answer and Answer Change records are deleted.

If no submission token or one that does not match the current one is provided, then it is assumed that the changes submitted are related to a new submission, thus the most recent submission is closed and the new provided answers are added to a new submission.
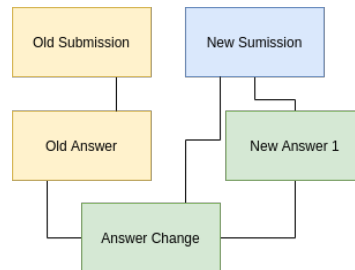
Figure 3.32: Whenever new changes to the answer of a question in a specific submission are submitted, a new Answer model is created, and also an Answer Change record is created connecting with the previous answer. In the case the provided submission token does not match the current one stored in the fingerprint model, then a Submission record will also be created, and changes will be associated with this new Submission.

Previously it was possible to update an answer of a single question through endpoint */api/fingerprints/[fingerprint hash]/answers/[question slug]/*, mentioned in the API portion of section 3.1.3, which was a more user-oriented alternative to the ones that were used on the old fingerprints views. On the new endpoints to validate and save the answers of a question set, if a single answer is sent, Django's form system will consider the rest of the answers to the other questions of the question set as empty. With this, the user is forced to always send the previous answers to all other questions plus the updated answer of the specific question. To avoid this, the */api/fingerprints/[fingerprint hash]/answers/[question slug]/* endpoints were kept, refactoring them to use the new models and Django's form system, but in these, the form used to perform the validation only contains one question instead of all questions of a question set.

With all this changes to the endpoints, choice-based answers are stored as a foreign key to the choice(s) selected. This implies that the answer's data sent on the Submissions Management endpoints for such question types must be these foreign keys, avoiding misspellings and non-existing choices. However, one does not know the keys for each choice and also the available choice if the web view is not used. For that, the Questionnaire Information endpoints are used to get information of choice-based questions, so that the correct data is sent along with the Submissions Management endpoints.

Draft

There were also some modifications to the endpoints to change the publish status of a fingerprint. Instead of having two different endpoints for different community settings (to auto-accept or not fingerprint publish requests), but for the same purpose, now only one exists.

```
POST [base url]/api/draft/[fingerprint hash]
```

This endpoint expects the same data in the same format, though takes different actions according to the community of the fingerprint at issue. Also, on the previous versions of publishing endpoints, there was a possibility to publish fingerprints that weren't complete, not having answers to required answers. Now, a validation is implemented when the user wants to publish its fingerprint, not completing the change on the publish status if the fingerprint is not complete. This avoids notifications being sent to the community manager telling that a user wants to publish a fingerprint, although such fingerprints might be incomplete.

3.2.4 Excel

The previously presented version of the spreadsheet used to define a questionnaire, had some problems in terms of clutter due to the overuse of the "Value list" column to define the extra configuration of specific questions types. The column was used to define all choices and their extra information of choice-based questions, to define columns and their settings of open multiple question and columns, rows, and type of choice tabular questions.

To fix those clutter problems, improvements were done to the spreadsheet by deleting some useless columns that were related to deprecated features and adding new types of rows that define the extra configurations for those questions types that require.

Starting by the columns of the actual spreadsheet now are the following:

- Type
- Text/Question
- Data Type
- Help Text/Description
- Slug
- Dependencies
- Constraints
- Tooltip
- Include in Advanced Search

Most of the columns are already known from the previous version, "Constraints" being the only one that was added to this new version. The columns that were removed are Stats, Comments Stats, Disposition (all associated with deprecated features), Level/Number (the number is now automatically retrieved according to the order of how the rows are defined), and Value List (brought the clutter problem to the table).

Type

On this column, the only possible values available were QuestionSet, Question, and Category. The new version has now eleven possible types of rows for the spreadsheet:

Figure 3.33: Differences between how a choice tabular question was versus how is now defined on the questionnaire spreadsheet.

Type	Description
IntroSection	These represent the same concept as QuestionSet.
Section	In the previous version, it was possible to add hidden question sets to both the beginning of the end of the questionnaire. This was achieved by setting the Level/Number column to either 0 or 99 respectively. Since the Number portion of that column was removed, such Question Sets must be
CloseSection	represented with the IntroSection and CloseSection rows.
Group	This is a replacement to the category row type plus the comment question type.
Question	The old and unchanged Question row type
Choice	This row indicates a choice of the previously defined Question
ChoiceInfo	Allows adding an extra question that can be answered if a specific choice is selected. Such a question will be rendered right below the choice. Note that this is different from the dependencies column.
TabularChoice	Row types to define the configuration of choice tabular question types
TabularRow	
Column	Row types to define the configuration of the open multiple question types
ColumnChoice	

Table 3.2: Available values to use on the "Type" Column of the new version of the spreadsheet to define a questionnaire.

Using the new row types described in table 3.2, the spreadsheet itself will be longer, however, it will be easier to read. In figures 3.33, 3.34 and 3.35, it's presented three before and after of how questions that require extra configuration were and are now defined with the new spreadsheet version.

It was considered to also remove the Level part of the Level/Number column, adding an EndGroup row type. Then whenever the user wanted to create a subgroup of questions after a question would create a Group with empty text and the framework would move the question

Type	Text/Question	Data type	Value list
Question	Clinical test parameters results.	open-multiple-composition	{title:"Weight X", field:"msurex", editor:"input"}, {title:"Height", field:"msurey", editor:"input"}, {title:"Msure Z", field:"msurez", editor:"input"}

Type	Text/Question	Data type
Question	Clinical test parameters results.	open multiple
Column	Weight X	short text
Column	Height	short text
Column	Msure Z	short text

Figure 3.34: Differences between how a open multiple question was versus how is now defined on the questionnaire spreadsheet.

Type	Text/Question	Data type	Value list
Question	Does this take place in participants' homes or at a central location?	choice-multiple-freeform	Home{...}|Central{...}

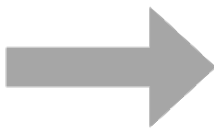| Type	Text/Question	Data type
Question	Does this take place in participants' homes or at a central location?	multiple choice
Choice	Home	
ChoiceInfo		short text
Choice	Central	
ChoiceInfo		short text

Figure 3.35: Differences between how a open multiple question was versus how is now defined on the questionnaire spreadsheet. Note that now it is possible to specify the data type of the extra information that is attached to the extra information of a choice.

of that group to a deeper level with no group label. This however would make the spreadsheet even longer for certain questionnaires that make use of categories to group questions that have some dependency on another question. Figure 3.36 shows how the change would affect the spreadsheet.

Type	Text/Question	Level/Number
Category	**Treatments**	h1
Question	Relapse Therapy	h2
Question	What is the requirement of this data?	h3
Question	Number of registrations	h3
Category	**Disease Modifying Treatments (DMTs)**	h2
Question	Past disease modifying therapies	h3
Question	What is the requirement of this data?	h4
Question	Number of registrations	h4
Question	Start and end dates of past treatments	h4
Question	What is the requirement of this data?	h4
Question	Number of registrations	h4
Question	Current disease modifying therapies	h3
Question	What is the requirement of this data?	h4
Question	Number of registrations	h4
Question	Start date of current treatment	h4
Question	What is the requirement of this data?	h4
Question	Number of registrations	h4
Question	Reasons for discontinuation of DMTs	h3

Type	Text/Question
Group	**Treatments**
Question	Relapse Therapy
Group	
Question	What is the requirement of this data?
Question	Number of registrations
EndGroup	
Group	**Disease Modifying Treatments (DMTs)**
Question	Past disease modifying therapies
Group	
Question	What is the requirement of this data?
Question	Number of registrations
Question	Start and end dates of past treatments
Question	What is the requirement of this data?
Question	Number of registrations
EndGroup	
Question	Current disease modifying therapies
Group	
Question	What is the requirement of this data?
Question	Number of registrations
Question	Start date of current treatment
Question	What is the requirement of this data?

Figure 3.36: A possible solution to avoid relying on the Level column to create subgroups of questions. On the new solution proposed it's much difficult to know at which level the specific question is located, without looking at the rows above.

Even with colors, the user editing has to look several rows above to know at each level it

is so he can match the "EndGroud" rows with the respective "Group" rows. This is the same problem as matching if open and closing brackets on C-like programming languages. Such a problem is alleviated since indentation is allowed, which can't be achieved on a spreadsheet. For those reasons, the Level column was kept, however, when a questionnaire is imported, whenever there are nested levels the framework will create groups with empty text.

Data Type

As it was possible to see in figures 3.33, 3.34 and 3.35, the Data Type column is used to detail the type of data that will be stored and how it will be requested to the user in terms of HTML widgets.

Table 3.1 contains all the previously allowed question types, a total of twenty-five different types. On the new version, this list was reduced, where some questions can now be achieved using simple question types in conjunction with the new row types that extend their base functionality.

Next is the list of question types available after the refactoring:

- short text: Old open. It replaces the open-validated field by making use of the new Constraints column;
- long text: Old open-textfield
- single choice: Used to achieved any old single-choice question type. The extra information input can now be achieved with the ChoiceInfo row type;
- multiple choice: Used to achieved any old multiple-choice question type. The extra information input can now be achieved with the ChoiceInfo row type;
- integer: Old integer;
- date: Old datepicker;
- email: Old email;
- url: Old url;
- numeric: Old numeric;
- publication: Old publication;
- image: Old open-upload-image;
- choice tabular: Old choice-tabular;
- open multiple: Can achieve both the old open-multiple and open-multiple composition

Open-location and timeperiod were removed since they weren't being used on any of the active installations of the MONTRA framework. The same happened with sameas and custom as additionally, they were shortcut type questions.

Slug and Dependencies

Previously the dependencies between questions were detailed by providing the id of the target question and the order of the choice that was needed to be selected, but since now the choices are described by row, the user can also detail an id for them, which such id can then be used on the Dependencies column of the questions that depend on such choice.

Constraints

The old Question model allowed to add custom checks, however, this feature was not possible to make use of through the spreadsheet, it was only possible to define them after the questionnaire was imported. This "Constraints" column intends to enable configuring such additional checks. Checks were previously stored in a custom format, where each check was separated by a space, defined as $name = "value"$. To avoid having to parse a string every time we want to load the constraints, they are now stored in a JSON field. Regarding the spreadsheet, the format chosen was a modified version of the original checks, supporting several types instead of interpreting every value as a string, allowing flexibility on the whitespace between each and within a check. Next is presented some examples:

```
name: "value"  -> string
name: true     -> boolean
name: 1        -> integer
name: 1.5      -> numeric
name: value2   -> also a string
```

JSON was also considered, however it was too much verbose to put on a spreadsheet cell, emerging the clutter problem.

Currently, the supported constraints are the following:

Question Type(s)	Constraint
any	required
short text long text	min_length
	max_length
	regex
integer numeric	min_value
	max_value
date	min_date
	max_date
	format

Table 3.3: Newly available constraints to apply to a question

3.3 SUMMARY

In this chapter, a metadata storing and visualization tool, the MONTRA framework, was detailed and analyzed, highlighting some of its poor design choices, which impact its usability and maintainability. A refactoring process was performed to fix such problems, preventing and giving feedback on user error when submitting data and improving the overall maintainability of components related to the metadata displaying.

The next chapter will go over the remaining part of the process regarding metadata. Extracting it from a database and then send it to update tools that store and display it, such as an installation of the MONTRA framework.

Automatic Metadata Extraction and Update

As we have an application ready to receive and display metadata, it is now crucial to turn the attention to extracting metadata from CDM databases, which contain the actual data, allowing, then, to update the data stored on applications such as the MONTRA framework.

Figure 4.1 presents a high-level architecture of the desired system that extracts data from databases and sends it to the applications that need their local data updated. Agents are software that runs on the data owner's deployment, in charge of extracting metadata from the database and sending this metadata to the applications.
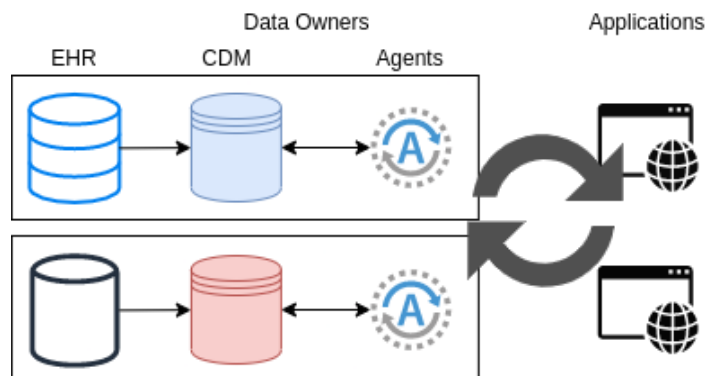
Figure 4.1: High-level architecture of the extraction and update process.

4.1 REQUIREMENT ANALYSIS

When we enter the realm of accessing an institution's sensitive data, some problems start to arise. Not all data owners are willing to, or legally can't, provide direct access to their data. When building this part of the system, we have to consider this and have several options available that use different levels of access and according to the demands of each specific data owner, use the most appropriate one. In one end, a single solution could have full access to

the data, extracting the metadata directly from the original data. At the other end of the scale, the burden of extracting data is removed from us and the system is dependent on the data owner providing the metadata and only then the system processes the metadata. Having several options for different access levels is more flexible, however, it requires maintaining several tools which are not reliable. A more appropriate alternative is to have a single option, that is designed considering the most strict access to the data, however, we still need to provide a tool to extract the metadata. The only difference is that we are not the ones executing it.

As the extracting tool has direct access to the data, using an existing and widely known tool is a must, so data owners are inclined to use it. Additionally, making fewer to no changes is preferable to ensure that data owners don't discard the tool because they don't trust the changes.
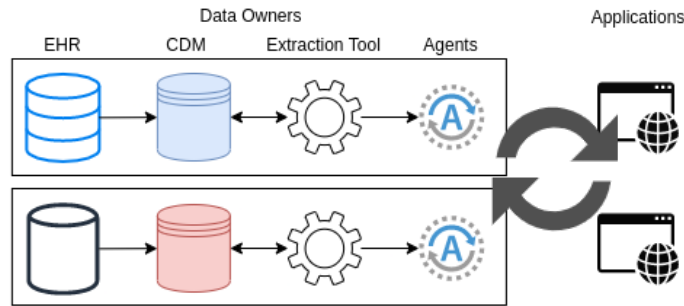
Figure 4.2: High-level architecture of the extraction and update process

In figure 4.2 is presented the overall architecture of the system, where the agent is the tool in charge of parsing the metadata provided by the data owner, which is then sent to the applications. Regarding this last point, there needs to be a way for the data to go from the agents to the applications. One ambitious solution is to create a decentralized peer-to-peer network of agents, avoiding developing a central component. In such solution, the agents are in charge of managing and sending the data to the applications. However, data owners might not want to spend their hardware resources to maintain a network. Since the agents are executed on the data owner's system, they should do only the required and minimal functionality, spending the few resources as possible.

A more executable solution is to have an intermediate component that receives the metadata from the agents and then distributes that data across the applications, following a centralized network architecture as it is presented in figure 4.2.
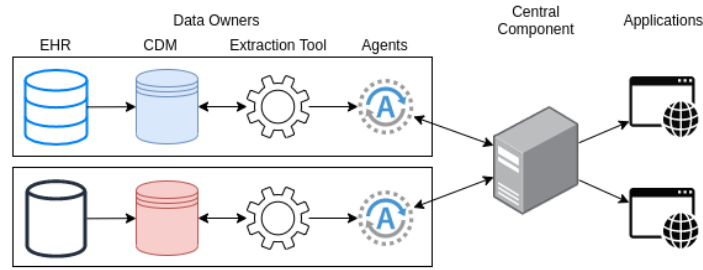
Figure 4.3: High-level architecture of the extraction and update process, with a centralized network approach.

This central component will be used to manage the data and the applications to where such has to be sent. However, it is also crucial that feedback is provided on the system's state, such as how the data is flowing, which applications received which data, and which databases sent data into the system.

On the applications side, there are other important factors to consider before going forward on implementing the intermediary component mentioned before. First, then we talk of application we are considering only web application that expect to receive metadata through a Representational State Transfer (REST) API. Second, the APIs of different applications expect the metadata in distinct arrangements. A way out of this would be to specify a set of endpoints that each application had to implement, yet that would lead the applications to have two sets of endpoints that do the same thing. On the other side, making a flexible option, where the requests where the data is sent are customizable, would be more appealing for the target applications, which would not have to make any changes. Third, not all applications require all metadata data that is extracted from a database. Some use all the data, others only need a value of such data.

4.1.1   Functional Requirements

1. Databases must be registered on the central component
   By having a record of the databases that are attached to the network, the system can avoid accepting data from databases that were not registered in the system.
2. It must be possible to check what agents are active at a certain point
   This allows giving feedback on the state of the network to the admin.
3. The system should allow managing the destination applications
   The applications to where the data of the databases are sent, should not be hard-coded, for that, the network admin should be able to perform CRUD operations on such. The term application should be interpreted as web applications that have REST API endpoints available to receive data.
4. The system should provide statistics on the data that has circulated
   This is yet another feature to give feedback to the network admin about, now allowing to both check the amount of data that each database sent, and the amount of data that each application received.
5. The format of the request sent with the metadata to each application must be customizable

This is important since different applications expect to receive metadata in distinct formats since each application has a different API.

6. The system should allow to group Databases

In a scenario where both all the destination applications and all databases belong to the same project, the metadata of the databases makes sense in all applications.

Now consider a scenario where half of the databases are associated with one project and the rest with another. Each project has a destination application that receives the data. Considering that there is no concept of project or groups, both applications will receive data of both projects. To solve this, it required to have two installations of this system, one for each project. By grouping databases, data of a database can be guided to applications that are also attached to the same group.

4.1.2 Non-Functional Requirements

1. The data owner can stop the agent at any time

This gives yet another layer of control to the data owners over their data. If at any time, they don't want to share their data, they can stop the agent. This also poses a constraint in the system that it should not be designed assuming that the agents, once deployed, will always stay active.

2. The extraction tool should be based as much as possible on an already existing solution

Using a working solution avoids having to build a new tool from scratch. But apart from that, using a known solution as a base helps to gain the confidence of the data owners so that they use the resulting solution in their databases.

3. The agents must not have direct access to the data

This option assumes the least amount of access to the user's data so data owners are receptive to allow agents to be run on their local deployments. This will require that an agent has access to at least a shared storage where agents check for new data and data owners post their extracted metadata.

4. The agent should easily deployable

As the installation of the agent will be done by a person not familiar with the project, this process should be as smooth as possible and well documented.

5. Agents should spend few resources

Considering agents run on data owners' deployment environment, it should have a low resource footprint so it does not impose any disruption. For that, they must be designed to perform only the required task.

6. The system should not require any open ports on the data owners' deployment environment

As there might be communication between the central component and the agents, requesting an open port on the data owners' deployment environment might not be possible since it could require them to change firewall settings as was mentioned in [31].

7. Scalable

As more databases are connected to the network, the system should allow increasing resources so that it handles processes faster.

8. Microservice architecture

The system should be composed of simple and replaceable components that deal with well-defined tasks.

4.1.3 Use Cases

From the previous requirements defined, there can be outlined two actors that will interact with the system:

- Data owners, which maintain the agent on their local deployment;
- System admin, that interacts with the central component to manage the whole system.

In figure 4.4 is presented the use-case diagram, which shows the actions that each actor performs on the system. Such actions were divided into three groups: Agents, Metadata Extraction, actions on the central component related to receiving data from the agents, and Metadata Update, actions on the central component related to sending data to the applications.
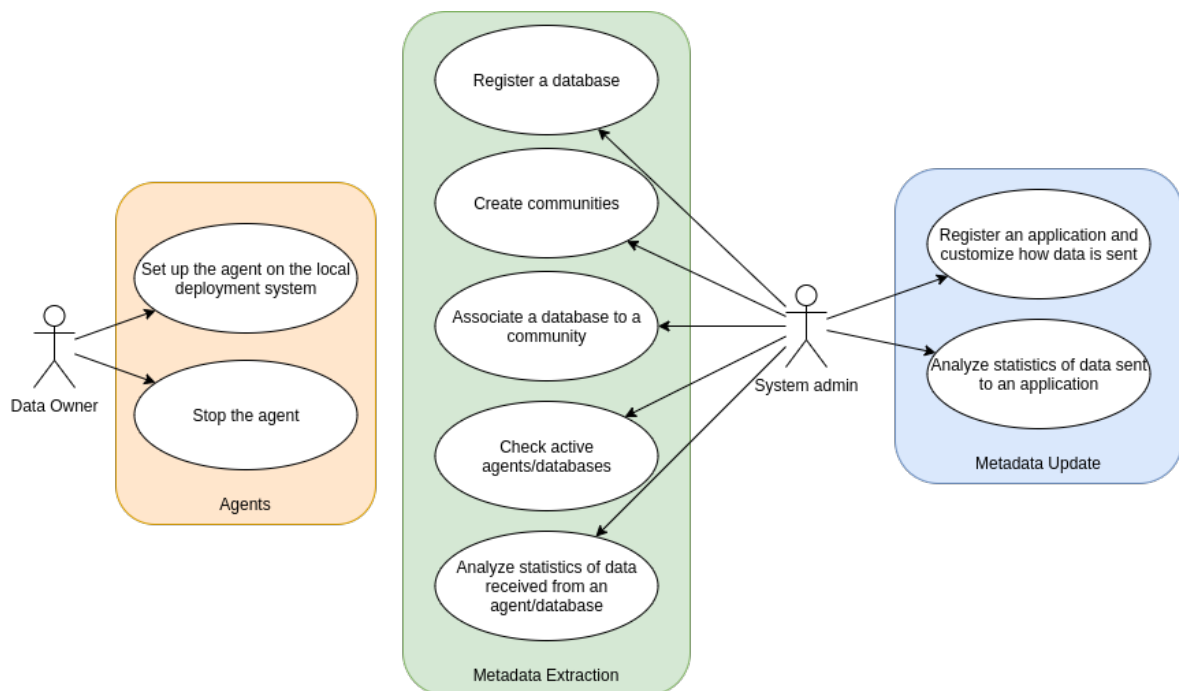
Figure 4.4: Use case diagram of the metadata extraction and update system

1. Set up the agent on the local deployment system

Since it's not expected that data owners provide access to their deployment environment, the responsibility to set up the agent is imposed on them.

2. Stop the agent

As Data owners control their data and the access to it at all times, them stopping the agent should be an action expected by the system.

57

3. Register a database

    As new databases enter the network and agents are deployed to gather their metadata, the system admin needs to register new databases on the central unit, so the data received is accepted into the system.

4. Create communities

    In a scenario where there are several databases from distinct projects, there is the need to group databases to treat their data separately.

5. Associate a database to a community

    Once there are databases and communities registered in the system, the system admin can do many-to-many associations between them.

6. Check active agents/databases

    At any time, the system admin can check the state of the network and verify which databases have an agent active.

7. Analyze statistics of data received from an agent/database

    As agents send data to the central component of the network, statistics can allow the system admin to see what databases contribute more data to the network and its frequency.

8. Register an application and customize how data is sent

    The system admin will also define the destination applications for the data received from the agents, setting also the format of the request that sends the data to such applications.

9. Analyze statistics of data sent to an application

    As metadata enters the system, having statistics of data sent to the application helps check if the system is working properly and if the applications are receiving the expected data.

4.2 Metadata Extraction

With the requirements analysis done, let's first go over the portion of the system which executes in data owners' space.

4.2.1 ACHILLES

Regarding the extracting tool, from the ones presented in chapter 2, the most indicated for our use case is ACHILLES, developed by OHDSI, or a variation of such. This choice was mainly because of its database, instead of the dataset, focused design, and because it was built to extract metadata from databases that follow the OMOP CDM.

The most adequate option is Catalogue Export [26] which was built using ACHILLES as the base, the main difference being that it only exports the required data to the EHDEN project.

These tools are distributed as an R[1] package, which uses a set of third-party packages that allow the extraction of metadata from different Relational Database Management System

---

[1]https://www.r-project.org/

58

(RDBMS)s. Metadata is separated into analyses, where each one captures a specific metadata metric of the database. Each analysis is captured through a different SQL query, allowing to extract all analyses through a multithreading approach. To make use of the package, the user must provide the information to connect to the database, such as user and password, an output directory to export the results, and the name of three schemas, which are, in most RDBMSs, a way to group tables within a database within the RDBMS. The schemas to provide are:

- CDM Data: where the original CDM data is stored;
- Results: that contains the tables that store the results obtained from the execution of analyses queries;
- Vocabularies: as each clinical database certainly will contain their distinct internal representation of clinical concepts, such as a database might encode sex with just a character (f, m, o), others might encode with the whole word (female, male, other), the OMOP CDM makes use of standardized vocabularies that allow creating a homogeneous representation for different databases. Such vocabularies can be acquired on the OHDSI's ATHENA website[2].

As this R package should only perform reads on the original data, the package should use a custom user that only has read permission on the CDM data schema. This can also be applied to the vocabularies schemas, as the package will use it only to perform joins with the CDM data. Regarding the results schema, read and write permissions are required to insert the results data of the analyses queries.

The package also allows exporting the results data to a Comma-separated values (CSV) file, which, on the EHDEN project, is then uploaded into a plugin of EHDEN Portal [6], which was built using the MONTRA framework, the main topic of the previous chapter, called Network Dashboards [38]. This plugin makes use of the metadata from the uploaded file to build several dashboards (figure 4.5), which allow researchers to better analyze and compare the data of clinical databases.

4.2.2 Publish-subscribe Systems

During the requirement analysis, one of the non-functional requirements was that "The system should not require any open ports on the data owners' deployment environment". This implies that the request of communication must be from the agents to the central component, as the contrary can only be achieved if the agent is listening on an open port. However, data still has to flow from the central component to the agents, such as if the central component wants to know if an agent is active. One solution would be to have a persistent connection between both ends, however, this could overload the central component if a high number of databases enter the network and it would be continuously taking resources from the data owner's system.

An alternative is to follow a pull philosophy. The central component can put requests on a message queue, the agent checks periodically this queue and if it finds any request, responds

---

[2]`https://athena.ohdsi.org/`
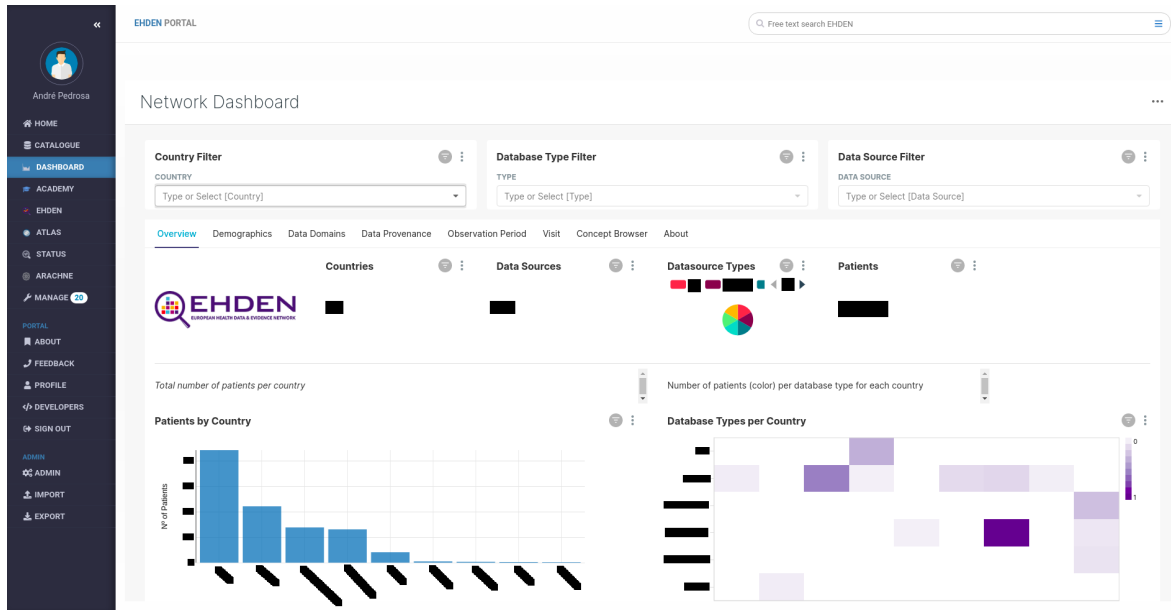
Figure 4.5: Network Dashboards tool used as a plugin on the EHDEN Portal

to it. This can be interpreted as following the publish-subscriber pattern. This solution also aids achieve the "Microservice architecture" non-function requirement as it helps decouple services since they do not require to be aware of each other. Additionally, the publisher service will not necessarily block waiting for the subscriber service to receive the message, also known as an asynchronous message system.

There are two well know and widely used implementations of such systems: RabbitMQ and Kafka[39].

RabbitMQ

It is recognized implementation of the Advanced Message Queuing Protocol (AMQP) protocol, which this last appears appeared from the need to have interoperability between distinct asynchronous messaging systems. Before the specification of the AMQP protocol, there were already several standards for synchronous messaging, however, the asynchronous world did not have such standards. There was the open Java Message Service (JMS), however, it was limited to java and was simply an interface standard, not specifying a standard protocol. AMQP defines a binary protocol implementation, ensuring interoperability between tools implementing the protocol individually.

igure 4.6 presents the architecture of RabbitMQ/AMQP. The message brokering responsibility is divided into exchanges and message queues:

- Exchange: Accepts messages from the publishers and, based on a set of rules routes, messages to the indicated queues;
- Message Queue: Holds messages and sends them to the consumers

60
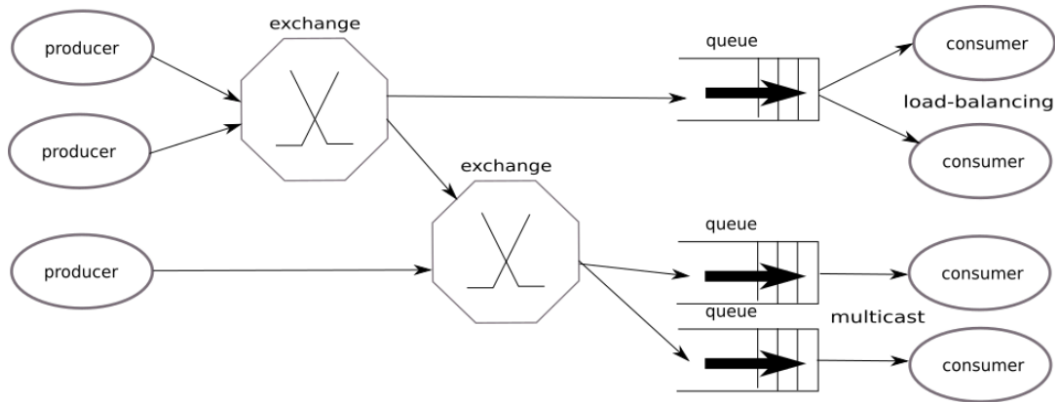
Figure 4.6: RabbitMQ(AMQP) Architecture. Retrieved from [39]
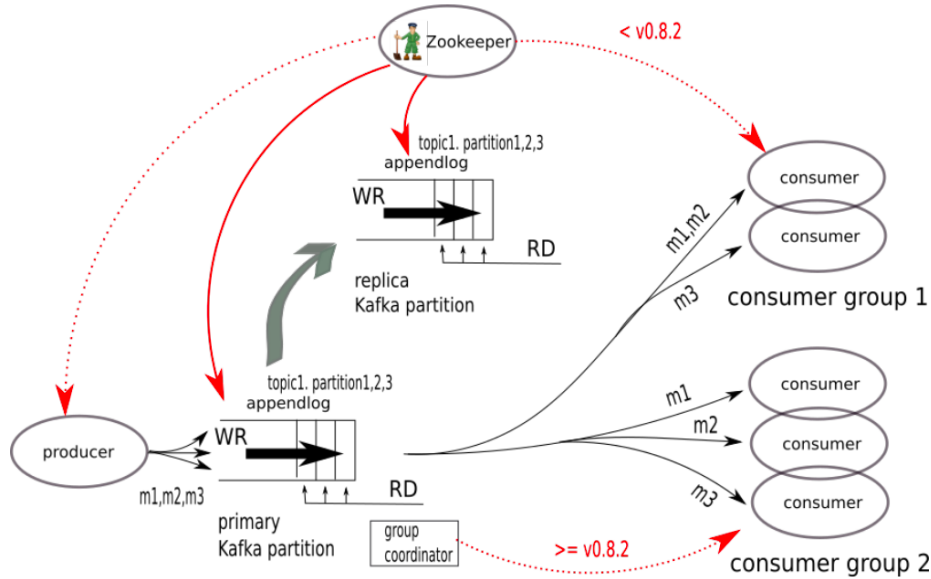
Kafka

Kafka was built at LinkedIn, as its central event pipelining platform. Before developing it, the team tested alternatives, such as ActiveMQ, a popular messaging system based on JMS. However, two problems arose during production tests:

1. If the queue started filling up to the point of not being able to store all the messages in memory, performance would sharply drop due to I/O;

2. To be able to send the same data to a different consumer implies duplicating that data in another queue.

They concluded that the explored messaging systems were designed to target low-latency use cases instead of high-volume scale-out deployment that was needed at LinkedIn. The team decided to build their custom infrastructure that produces efficient persistence, supports several consumers with a low-performance penalty, and mainly supports distributed consumers while maintaining a real-time messaging abstraction usually obtained from messaging systems.

As a result, the produced system is a scalable publish-subscribe messaging system designed around a distributed commit log. With this, high throughput is achieved due to data being written to log files with no immediate flush to disk, which allows to build the system with efficient I/O patterns.

The high-level architecture of Kafka is presented in figure 4.7. Producers send messages to a Kafka topic. Each topic is distributed over a cluster of Kafka brokers, with each broker holding zero or more partitions of a topic. A partition is an ordered append-only log of messages that are persisted in disk. All topics are accessible for reading by any number of consumers, and new consumers have no performance penalty. Due to its simple storage layout, every time a producer publishes a message to a topic, the broker simply appends the message to the file corresponding to the associated partition.

Figure 4.7: Kafka Architecture. Retrieved from [39]

In contrary to the regular publish-subscriber system, the concept of a consumer in Kafka is generalized to be a group of co-operating processes running as a cluster. Because of that, a message in a partition of a topic is delivered to only one consumer of a group. Then, partitions allow to control the level of parallelism of consumers, as different partitions of a topic are assigned to different consumers of a consumer group.

In the end, to achieve the high-throughput requirements, Kafka went on a different road compared to the classic principles of messaging systems in several ways.

- It partitions up data so that the production, brokering, and consumption of such can be handled by a cluster of entities that can be scaled as load increases;
- Messages are not removed from the log, instead, they can be replayed by consumers;
- Reader progress is maintained only by the consumers, so message deletion can only be managed based on a retention policy setting, based either on message count or age;

Originally, the intent was to use a publish-subscriber system just for the communication between the agents and the central component. However, Kafka has some functionalities that appeal to use also to manage metadata, working as a central component. Having long-term message storage allows to easily build a fault-tolerant system since consumers can easily replay the message received on a topic. Furthermore, due to the popularity of Kafka, several libraries and frameworks were developed to extend their usage. One of them is Kafka Connect, a framework that allows to reliably and easily stream data between Kafka and external systems. There is also a Kafka Streams[3] library to perform processing on streams of data on top of Kafka. As stated before, a different application might not be expecting the entire data from the databases, for that, stream processing features could be useful to filter the data before

---

[3]https://kafka.apache.org/documentation/streams/

62

reaching the target applications. Finally, with its easily scalable modular architecture, Kafka is well suited to our use case.

4.2.3   Kafka Source Connectors

At this point, we need to find a way to move the data extracted from the database into Kafka. As mentioned previously, the Kafka Connect framework allows to easily import and export data between Kafka and third-party systems. Such framework provides plugins called connectors that can be of two types: source, which brings data into Kafka, and sink, which transports data from Kafka into another system. At this stage the source connectors are more important, however, the sink connectors will still come into use further in the system.

Since the extraction of metadata through the Catalogue Export R package is a bulk process, in other words, a set of metadata metrics are extracted from the database at once, the source connector should handle the data in the same matter. As there is no way to know, on the central component side, when all the data is in Kafka, the connector should deal with metadata as a bounded source of data, providing some kind of feedback that all data was uploaded to Kafka.

Confluent[4], a streaming platform based on Kafka that aims to aid in optimizing and managing Kafka clusters, has available a search engine, Confluent Hub[5], that allows searching for Kafka connectors and more. The different connectors available range from known database systems (MongoDB, InfluxDB, ...) to internet protocols (HTTP, File Transfer Protocol (FTP), ...) In our case, considering that no other application will be running along with the agents, the only storage options available are either a single table on the data owner's relational database or a directory on the disk where metadata is placed in files.

After researching on the Confluent Hub, three connectors were found to suit the possible storage options:

- Java Database Connectivity (JDBC) Source connector[6]

  It has four different modes of capturing data from tables. Three of them provide incremental information, sending a message to Kafka after a row of a table or a custom query is created or updated. The difference between these three modes is from what columns the connector tries to infer that a row has changed. These are not suitable for our use case, since it treats a table as an unbounded source of data.

  The other mode available is bulk, which sends all the rows of a table in one go. However, this process isn't performed based on updates on data, instead, it is executed periodically on a configured interval. Furthermore, no feedback is provided after all the data is present in Kafka.

- FileSytem Source connector[7]

---

[4]`https://www.confluent.io/`
[5]`https://www.confluent.io/hub/`
[6]`https://www.confluent.io/hub/confluentinc/kafka-connect-jdbc`
[7]`https://github.com/mmolimar/kafka-connect-fs`

On the contrary to the first connector, this one uses file systems as the source to insert data into Kafka. Not only supports fetching files from a local file system, but also widely used cloud storage solutions such as Hadoop Distributed File System (HDFS), Google Cloud Storage, and others. In our case, we would be using the local file system approach, whereby the data owner places the output files of the Catalogue Export R package on a specified directory. As both the extracting tool supports extracting the gathered metadata into a CSV file and the connector has a specialized reader to parse files in that format, that would be the agreed format to use. The connector also supports either moving or deleting files once they are entirely uploaded into Kafka.

As a downside, this connector does not provide any feedback once it finishes uploading the file.

- FilePulse Source connector[8]

  This connector, is similar to the previous one, as it allows to load data from files into Kafka, having the possibility to also load from other file system solutions. It supports reading CSV files, as it allows having a cleanup policy, which can move or delete files after they are uploaded.

  However, contrary to the previous connector, this one sends messages regarding the progress of the upload of the file to a separate topic. Not only sends when it ends but also sends the number of rows/messages that were sent into Kafka, which is ideal for our use case, as we can know when the whole file is processed if the rows go through a processing stream on Kafka.

4.2.4 Agent Final Architecture

At this point, we have all the building blocks to set up an agent and make it ready to deploy on the data owners' local system. The agent will have two internal components:

- One will be the FilePulse Source Kafka connector, which will parse the files provided by the data owners and insert their data into Kafka. It is important to note that both the metadata and the upload notifications use different topics for each database. This was decided because, otherwise, some processing would be required on gathering data of a specific upload from the main topic, that would contain messages from different uploads of different databases. The available CSV reader of this connector allows performing transformations on the data before sending it to Kafka. Such transformations include ignoring empty rows, converting values on a row, among others. One convenient transformation available transforms the string data of a row into a structure, so data sent in Kafka's messages values is a JSON value instead of a string one. This avoids further on the system having to parse strings received from this connector to get a specific field from the data.

---

[8]`https://github.com/streamthoughts/kafka-connect-file-pulse`

- The other component is the Health Check Handler. It is in charge of sending messages periodically, telling that the agent is active, so when the admin checks the status of the agents it gets a more updated Last Time Active metric. As this periodic send of "I'm active" messages will not be that regular, one to two times a month, the admin can request an agent to answer if it is active, and it will be this component in charge of sending a response. Regarding this last component, as it needs to be developed from scratch, it is a great opportunity to test new technologies that are not so commonly taught on a regular Informatics Engineering course. By technologies, we intend to refer to the programming languages used. The most popular and used ones are Python and Java, however, other programming languages appeared and are starting to gain terrain on the wap app development. Examples of such programming languages are Rust[9] and Go[10]. According to the 2020 Stack Overflow Developer Servey [40], Rust is the most loved language among developers, and Go, in terms of most popular languages, just stands behind equivalent languages such as Python, Java and C Sharp. Besides behind the most loved language, Rust's ownership model, which increases memory safety with compile-time checks, is something that is not present in other languages, and newcomers will have a not-so-appealing learning curve. For that, the decided language to use was Go, which will also be considered for other components of the system that eventually require to be built from scratch. The popularity of Go mainly comes from its simplicity, as it has a C-like syntax, and for having concurrency mechanisms builtin in the language, not requiring an external library to implement such concepts.

Now the only missing part of the agent is how to deploy it. As said in the requirement analysis, the deployment of the agent should be an easy process. At its core, the FilePulse connector is a Java application. With that, the agent requires that the data owner has both Go and Java runtime dependencies available so that the agent could be deployed in their local system. This could cause problems for the data owners since he could already have these dependencies, but the wrong versions, and installing a new version could cause problems. Also, different data owners could use a different operating system, which could change the way dependencies have to be installed, so it's hard to create install instruction that covers all the possible deployment environments. With the emerge of cloud computing, virtualization solutions started to be used, allowing to use a single physical machine for several purposes, taking more advantage of the available hardware. To achieve this, a software called Hypervisor sits on top of the host machine, which divides the hardware resources so that they can be used by separate virtual machines [41]. However, such virtualization technology emulates an entire operating system, on every virtual machine, which is a heavy option to deploy a simple agent. A more convenient virtualization technology is containerization. Such technology can be seen as a very lightweight virtual machine since a container is a package of software code with just the application to run and the necessary dependencies. Containers are also more efficient in taking advantage of the hardware available, as portions of the hardware are not

---

[9]https://www.rust-lang.org/
[10]https://golang.org/

statically allocated to each container [42]. Containers run in an operating system as they were yet another process, but for the code running in a container, the appearance is that they are running in an isolated machine.
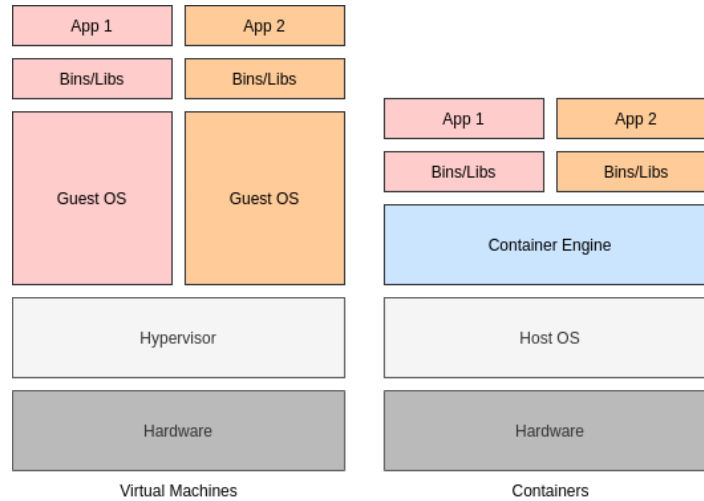
Figure 4.8: Comparison between virtual machines and containers

Figure 4.8 presents a visual comparison between virtual machines and containers. As can be seen, the virtual machine technology has to simulate two operating systems, in contrast to the container solution where there is only one operating system and the container engine.

With a deployment strategy in mind, figure 4.9 shows the final architecture around the agents and pieces that connect to the data owner's system.
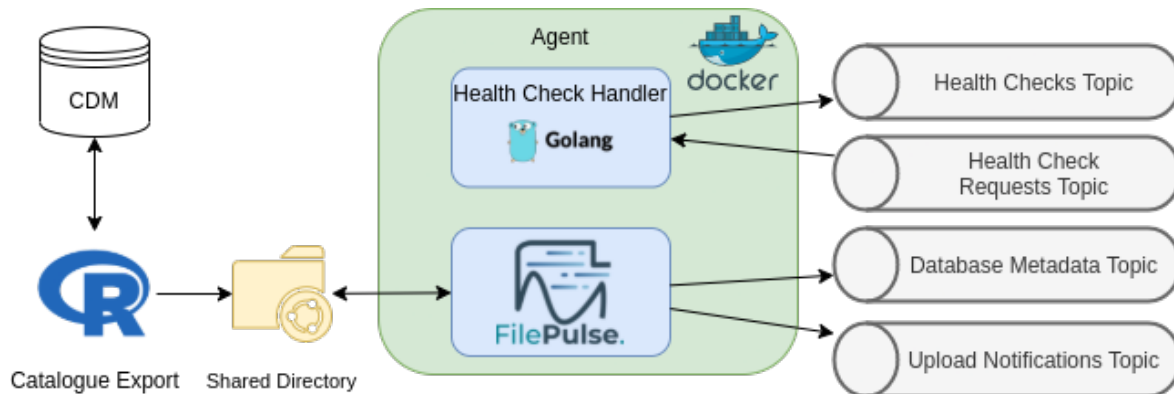
Figure 4.9: Agent + Catalogue Extractor architecture

Docker[11] is an industry standard that facilitates the process of creating, managing and sharing containers. Developers can share their built containers, called Docker Images, in Docker Hub[12], which then can be used to build more specialized containers. For example, there are official Docker images available with Python pre-installed. These images can be used by developers that want to deploy their Python apps with Docker, not having to bother installing a specific Python version.

---

[11]https://www.docker.com/
[12]https://hub.docker.com/

With that, the agent will be distributed to the data owners as a Docker image. However, data owners have to perform two tasks before starting the agent. One is to choose which directory will be shared with the agent's container. This can be achieved with the Docker's volume system, where a directory on the host operating system will be bound to another within the container. The second step is to define a set of environment variables that will be unique to this agent:

- AGENT_DELETE_CLEANUP_POLICY: defines whether or not files must be deleted after their data is uploaded into Kafka. By default, the file is moved into a *succed* directory, so this variable is optional.
- AGENT_DATABASE_IDENTIFIER: an identifier provided by the admin once the database is registered in the system.

After getting the Docker image of the agent, the data owners only have to execute a simple run command where defines the previous decisions:

```
docker run
  -v ./catalogue_export_files:/app/catalogue_export_files
  -e AGENT_DATABASE_IDENTIFIER=unique_db_identifier
  -d
  AGENT_IMAGE
```

Next is the description of each option:
- -v: binds a host directory to the one inside the container where FilePulse will read files from;
- -e: sets the AGENT_DATABASE_IDENTIFIER environment variable
- -d: runs the container in daemon mode

Regarding the extraction tool, we will also distribute a Docker image with the necessary dependencies preinstalled, where data owners will have to provide the necessary information for this tool to access the CDM database and define a volume so files containing the metadata are accessible to the agent. Concerning the automation of the extraction process, the image will contain Linux's cron command-line utility, which allows to schedule tasks. The data owners will only have to provide the periodicity of the extraction process, using Cron's time and date syntax to define a cronjob.

```
.---------- minute
| .-------- hour
| | .------ day of month
| | | .---- month
| | | | .-- day of week
| | | | |
* * * * *
```

If the extraction process were to run two times a month at three in the morning, the following configuration is used:

```
0 1 1,15 * *
```

The task will execute on days 1 and 15 of every month at three in the morning. However, if it is preferable for the data owner, he can set up his automation, or no automation, of the extraction process.

4.3 PUBLISHING

At the moment, we can deploy several agents on the data owners' local systems and have a set of Kafka topics to where data is uploaded when CSV metadata files are generated. We now need to send this data to the web application in form of HTTP requests. Ideally, no other component is required, and Kafka is enough to act as a central component of our metadata publishing system. With that, a search must be performed to find Kafka sink connectors, which should do the inverse of what the FilePulse connector does on the agent side, which is getting data from Kafka into an external system, in this case in form of HTTP requests.

Resorting to Confluent Hub again to search for sink connectors, only one allowed sending Kafka data in HTTP requests: HTTP Sink Connector[13]. However, it does not meet some of the requirements. Since different applications expect the data in distinct formats, requests need to be customizable, however, the connector does not allow to customize the request. The data sent, will be the same as the one present on the value of the Kafka message and the connector only allows to send requests with the HTTP's POST or PUT methods.

Furthermore, by default, this HTTP connector will send a request for each Kafka message. The connector being used on the agent sends a message for each row of the catalogue export CSV file, which would mean that this HTTP connector would send several HTTP to an application related to the same upload. The HTTP connector supports batching messages together, on the same request, however, messages are stored in memory until the batch size is not reached, which could cause memory problems if the upload contains a high number of metadata metrics and also if several databases upload at the same time. Additionally, the number of messages sent by a database is dynamic, which the HTTP connector does not support, since it gathers a finite number of messages. This problem will always exist due to the nature of Kafka, where topics can be treated as streams of data, which do not have determined start and end, and there is always data flowing.
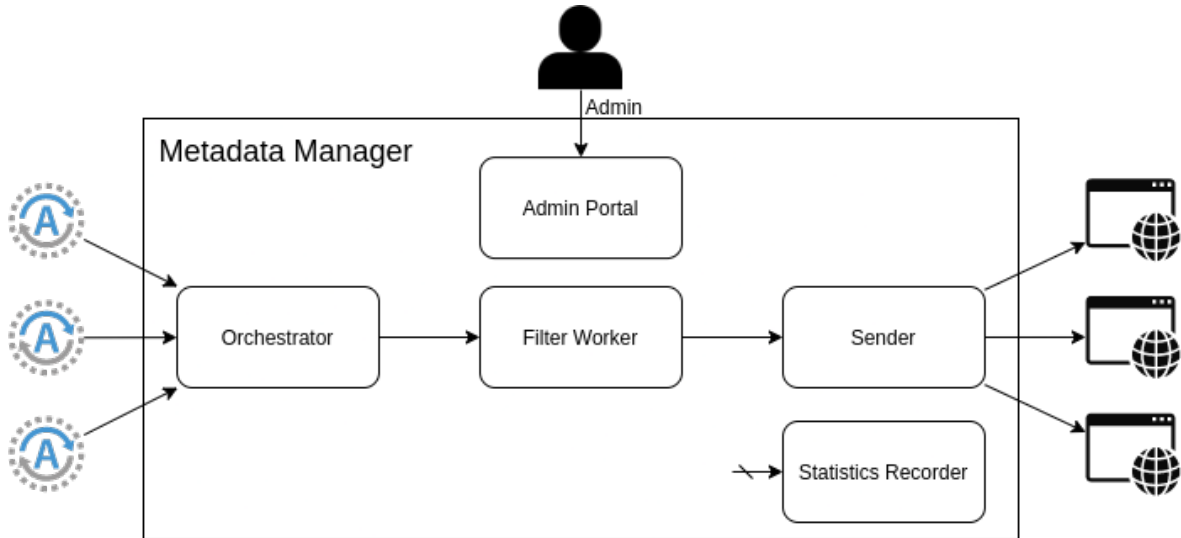
4.4 METADATA MANAGER

To takle the problems mentined previously, new components need to be developed. Such group of components will be called Metadata Manager, which will perform operations on top of Kakfa. The main issues that will be addressed are allowing to customize the requests that are sent to the applications and dealing with the unbounded factor of Kafka topics/streams.

---

[13]https://www.confluent.io/hub/confluentinc/kafka-connect-http

Figure 4.10 presents an overall architecture of how internal components of this Metadata Manager will be organized. For the communication between components, Kafka topics are used, however their organization will be detailed when going over the several components next.

**Figure 4.10:** High-level architecture of the Metadata Manager components

As these components will be implemented from scratch, the programming language that will be used is Go, if applicable, for the same reason explained in the Health Check Handler component of the agent, try new emerging technologies beyond the most popular ones.

4.4.1 Filter Worker

Data comes from the agents in several Kafka messages, which need to be gathered together to then be used to build the custom request for each application. As gathering all the uploaded data in memory is not a scalable option, and implementing a system with memory usage management is a complex task, the system must first filter out unnecessary data and only allow the Admin to build and send the request to the applications. A Filter operation will remove entire rows, however, only a subset of columns might be required, for that, the system should also allow choosing which columns of the metadata are required.

That's what the Pipeline Worker component is supposed to do. Extract only the essential data from all the data uploaded by an agent and then send it to the Sender component, who will be in charge of building and sending the request to the applications.

Since data is already in Kafka, we can take advantage of its stream processing features to filter data received from the agents. But, once again, streams imply an unbounded size of data, so additional processing has to be built on top of such stream processing. If a stream is created to get the data that satisfies a given condition, we need to make sure that all the uploaded data went through that given stream. This can be achieved by creating two streams with opposite filtering conditions (figure 4.12).
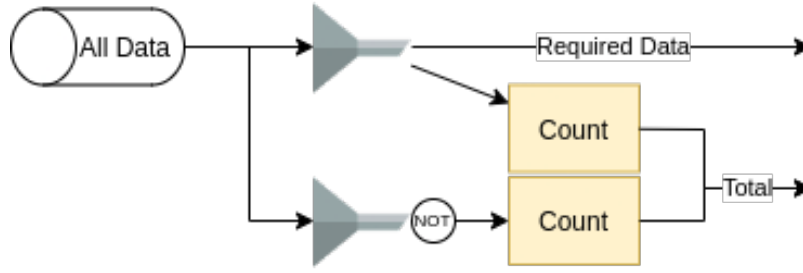
Figure 4.11: How Filter Workers process unbounded data received from agents

One will be used to get the actual data, the other, with the filtering condition negated, will be used just as an auxiliary. At the end of each stream, a component will count the number of records that went through. When both sum up to the number of messages sent by the agent, we can know for sure that all data was filtered and the Sender component can use the result data. As the stream with the negated filtering condition will be used to just get a counter, there is no need to send the actual data on that stream, for that a single byte is sent to avoid wasting memory resources.

To develop these filtering streams, there are two options:

- Kafka Streams: A Java library to process and analyze data that is already present on Kafka. Using it would imply implementing this Filter Worker component in Java, however, from the explanation of how a filter will be implemented, there will be several threads involved, requiring communication between them. Developing such an environment in Go is much easier as concurrency mechanisms are built in the language itself, and also if the number of filters gets high, threads might start to slow and use a lot of memory, as each has its own address space. Go, on the other hand, has goroutines, lightweight threads managed by the Go runtime, which run in the same address space. Using Kafka Streams would also imply that the conditions defined by the Admin, would require to be written in Java code.

- KsqlDB [43]: is a database that allows assembling stream processing applications on top of Kafka. It combines the power of stream processing with the known feel of a relational database through a SQL-like syntax, so stream applications can be set up with just SQL statements. As the data sent by the agents are messages extracted from a CSV file, the data can also be interpreted as a table, so the SQL language feats well in this case. It provides a REST API interface, so using KsqlDB does not bound the implementation to a specific language.

We opted to go with KsqlDB as it allows us to build the application in Go, taking advantage of its lightweight goroutines, allowing the system to scale better and the system to receive data from more databases.

Until now, it was assumed that each application has its filtering condition over the data. However, two distinct applications might benefit from the same filtering condition, for that, the relationship between filters on data for an application is a one-to-many relationship, so the data resulting from one filter might be used to build and send data to several applications.

In terms of internal organization, displayed in figure 4.12, the main goroutine is in charge of managing the several active Filters, stopping existing ones and starting new ones, according to the orders received from the Admin Portal component. Each filter runs in a separate goroutine, which by itself launches other two goroutines. One is in charge of counting the data that comply with the filtering condition and the other is counting the number of messages that do not fit the filter condition. The count of messages filtered is monitored by the main goroutine of each filter. When it reaches the number of messages that the agent sent, it tells the other child goroutines that they can stop reading from the streams, and sends a notification to the Sender component telling that data is ready to be sent.
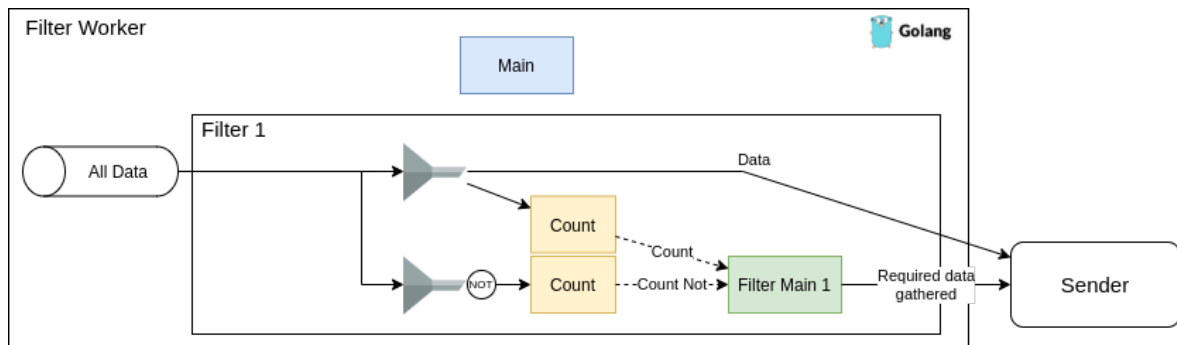
**Figure 4.12:** The internal organization of a Filter Worker

Note that the "All Data" topic is unique to the Filter Worker application, this means that on a Filter Worker application there can be several Filters running at the same time. Also, this means that a Filter Worker application will obtain the data to all the filter conditions that were defined, at the same time, for one single database. The system then allows scaling, by running several Filter Worker applications at the same time, with that, several Filters can be run for different databases at the same time.

4.4.2 Orchestrator

As multiple Filter Worker components might be running at the same time, there needs to be a way to redistribute the load among the running instances. Since agents publish a topic when the upload process is done, a goroutine on each Filter Worker could dynamically create the required processing streams on the data topics of the databases that performed the upload. However, as all data on the data topic of a database might not be related to the same upload, due to the retention policy chosen, we require to start the transmission of messages at a given offset of the data topic. Such a thing is now achievable with KsqlDB since it only allows to read either from the latest or earliest record. As an alternative, Kafka Streams allows achieving this. This will require that this redistribution of work has to be written in Java, but as an advantage removes, this responsibility of reading data from databases' topics from the Filter Workers components, allowing for more concrete and small components.

Previously, it was mentioned that the concept of a consumer in Kafka is a group of co-operating processes running as a cluster, so messages of a topic are distributed in a balancing manner, sending a message to only one process of a given group. For that, there is no need to

implement a balancing algorithm to balance the data uploaded from the agents, as Kafka can achieve that already.
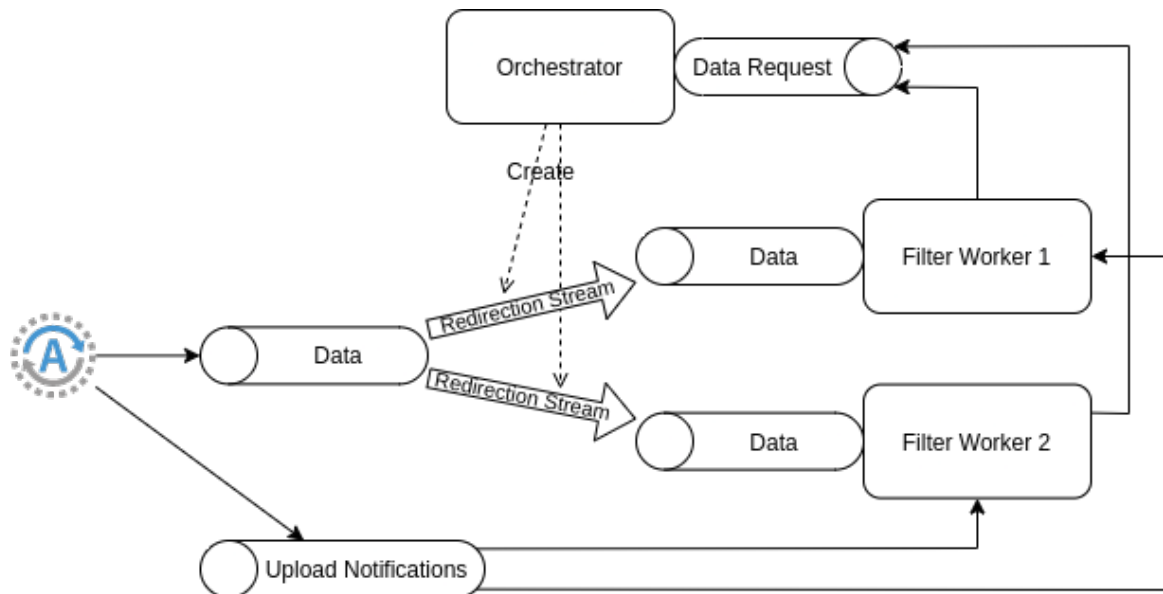
Figure 4.13: The workflow of how data uploaded from the agents is balanced across Filter Workers instances

As shown in figure 4.13, every Filter Worker will have a consumer for the Upload Notifications topic, which all will belong to the same group. This ensures that only one Filter Worker instance will receive a given message, thus implementing a balancing policy with Kafka. On the Filter Worker component, another goroutine will have to be created, which will consume from the Upload Notifications topic and then broadcast this notification to the active Filters (Figure 4.14).
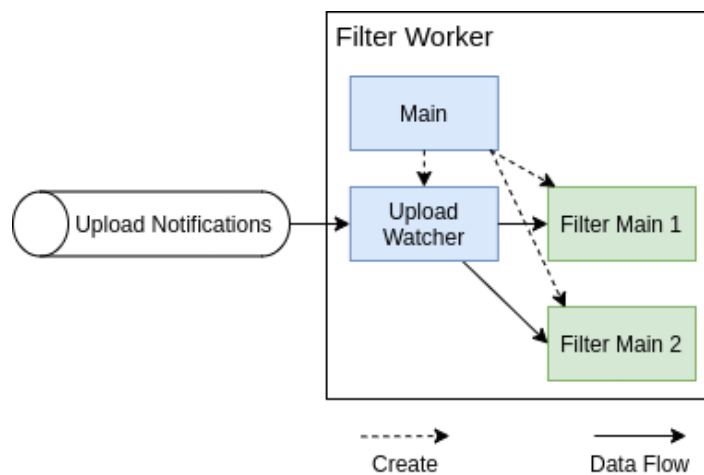
Figure 4.14: New ententies achitecture on a Filter Worker component

This new entity, Upload Watcher, will broadcast the notification using Go's channels, a built-in language feature that allows having pipes of communication between goroutines.

Additionally, this entity will also send a message to the Data Request topic, telling the Orchestrator that his Filter Worker instance is in charge of parsing the data associated with a specific upload of an agent. The Orchestrator will then create the necessary redirection stream between the database's data topic and the specific Filter Worker instance's data topic.

Following the same balancing concept used for the Filter Workers, the same can be applied to the case of scaling the number of Orchestrators. Different Orchestrator instances will consume from the Data Request topic belonging to the same group, balancing the work of redirecting upload data over several Orchestrator instances.

Internally, the Orchestrator is composed of two entities:

- one that consumes from the Data Request topic and creates the redirection streams from databases' data topics into the Filter Workers' data topics;
- for each message received by the previous entity, another entity checks when all data reached the data topic of the target Filter Worker instance and closes the stream once that happens.

4.4.3 Sender

At this point, the system contains the data required to build the requests and then send them to the applications. It is now required to get, from the Admin, the configuration of the HTTP request to send the data. An interface to build HTTP requests would require several components to allow customizations of the several items of an HTTP request: authorization, headers, method, . . . To allow flexibility on the configuration of the HTTP request, such an interface would either be too complex or some features would not be allowed to simplify the implementation of such interface.

Many programming languages have libraries that allow performing HTTP requests and customization is provided by passing different parameters to functions of such libraries. Such parameters will then go over a validation process, avoiding the use of invalid values, such as sending a request with REPLACE as the HTTP method for example. With that, it would be easier to make use of such libraries, and then Admins would provide the arguments for a function that performs HTTP requests. The system would provide the data that resulted from a filter to the Admin, and then he would build a structure where they specify the required and additional optional parameters of an HTTP requests library's function.

On top of that, the request for each database within the same application might be different, one might use the URL "http://app.com/data/db1/" and another use "http://app.com/data/db2/", for that the system must provide additional data besides the data filtered, such as the database information.

Basically what the Admin needs to provide the system is a template defining the parameters to a function of an HTTP request library function. On this template, the Admin can use placeholders to define where the system should insert certain data. Django contains a template system that allows building dynamic HTML pages. The developer can write a normal HTML page and then, using a special syntax, can tell where Django should insert data to fill the page.

```
<html>
  <head>
    <title>{{ app_name }}</title>
  </head>
  <body>
    <h1></h1>
  </body>
</html>
```

In the example above, {{ *app_name* }} will be replaced by the value of the variable app_name.

However, the entire Django framework is not required to build the Sender component, as the component does not require to either render pages or manage database models. Only the templating system is required. Fortunately, there's a Python package called Jinja[14], which was built based on Django's template system. It has a similar syntax to the Django Template System providing some additional features and being more pythonic.

For now, let us assume that the template will be an argument defined in each line, it would look something like this:

```
method: POST
url: http://www.app.com/data/{{ database_identifier }}/
data: {"patients_number": {{ filtered_data.get("patients_number") }}}
```

On Jinja, placeholders are defined within {{...}} tags, so it would look for the database_identifier variable and execute the get method of the filtered_data variable.

While building its template, the Admin must have access to the filtered data, which, on the example above, was exposed through a filtered_data variable. As data is in a Kafka topic, the system must provide an abstraction to easily access such data. Once again, since data is extracted into a CSV, it can be interpreted as a tabular type data. Two programming languages suited to read and manipulate data in this format are Python and R, which are widely used on data analysis applications. R is a more specialized language than Python since is suited for statistical analysis [44], however, Python is more popular, as there are many data analysis and machine learning packages written in it. As the Admin will mainly just want to access the data, the chosen approach was to expose the data using the Pandas Python package[15]. It contains a DataFrame data structure, which is widely used in data manipulation and analysis applications written in Python.

With Jinja and Pandas, the Admin can customize the request that sends the data to the applications, however, there is a use case that there is no possibility to achieve with this implementation. It was mentioned previously that the EHDEN project uses a separate tool as a plugin of the EHDEN Portal to build visualizations allowing for better comparison between databases, which is named Network Dashboards. Data is inserted into the tool by uploading

---

[14]https://palletsprojects.com/p/jinja/

[15]https://pandas.pydata.org/

a file into a form, which is not possible to reproduce with the current implementation. In Python, the most popular library to perform HTTP requests is called requests[16]. It contains several specific functions that allow performing any type of request. Files can be sent with this library by passing a file pointer as one of the arguments. In Python, a file pointer can be acquired by opening a file on disk. With this, the system has to provide a way to allow the Admin to insert this file pointer in its template of the HTTP request.

To avoid having to create a parser to read a custom structure of key-value pairs where Admins define the parameters of the request, it would be ideal that the template could be generated directly into a Python data structure. This can be achieved by running code dynamically. When the template provided is rendered, replacing the placeholders by their values, a string value is generated. We can then treat that string as Python code and execute it. The system will require that this code returns Python's key-value data structure called dictionary, which can then be used to pass arguments to the requests function that performs the HTTP request since Python allows to provide arguments to a function with a dictionary.

Let's sum up the process of building and sending the requests to the applications (Figure 4.15):

1. The Admin defines a template with placeholders;
2. Once data is received by the Sender component, the template is rendered and a string will be generated;
3. The generated string will be executed as a Python code returning a Python's dictionary;
4. The dictionary is used to provide the parameters to the request function to perform the HTTP request.
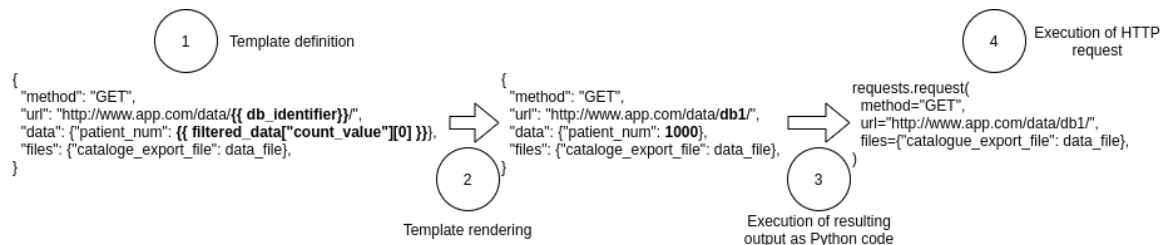
Figure 4.15: Process of generating the parameters of the HTTP request to send to an application

Note that, in figure 4.15, the value of the files dictionary is a variable name, which will be a local Python variable containing the file pointer to a temporary file containing the records received from the Filter Worker component.

---

[16]https://docs.python-requests.org/en/latest/
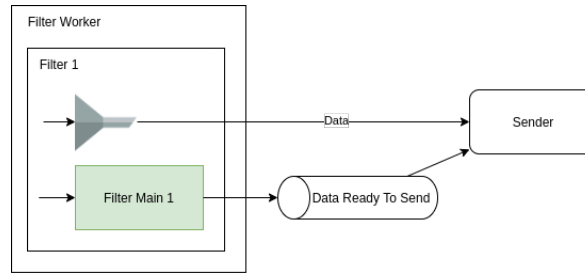
Figure 4.16: Interaction between the Filter Worker and Sender components

Regarding how data is transferred from the Filter Worker component to the Sender, figure 4.16 shows a representation of the process. Once all data was filtered, the main goroutine of each Filter will send a message to the Data Ready to Send topic with the following metadata:

- Filter Worker application id: since several could be running at the same time;
- Filter id: through what filter did the data go;
- Kafka topic offset: wherein Filter's data topic does the data start.
- Records count: number of records that meet the filtering criteria;
- Database identifier: to what database does the data belong;

The internal structure of the Sender component is similar to the Orchestrator components, having the main thread consuming from the Data Ready to Send topic, and then other threads will deal with the work of building and sending the request to the applications.

Scalability is also allowed on this component as multiple Sender components can be running at the same time. For that, consumers used by their main thread all must belong to the same group to ensure that a message is sent to only one Sender application.

4.4.4 Admin Portal

Moving to the component where the Admin will use to manage the whole metadata manager. It will allow performing the several use cases defined on the requirement analysis stage: managing communities, databases, filters and target applications, and checking statistics of the system.

As this component is mainly just a web interface, a search was performed for Go packages that allowed to create some kind of admin dashboard. Some solutions were found such as Qor Admin[17] and GoAdminGroup/go-admin[18]. The first solution had good documentation and features. It creates management pages for each data model defined, allowing a user to perform the regular CRUD operations on such data models. However, there was small flexibility on what to display on a display page of a data model. For example, if a model had an id and a name, only those fields would appear on the display page of such a model. The second solution had better flexibility compared to the first one, however, page customization was done through a series of function calls which lead to big chunks of code to define a page, consequently, contributing to the maintainability of this component to be hard.

With this, we decided to separate this component into two parts, a web interface app and a backend app exposing an API for the web interface app to use. This allows using technologies

---

[17]https://github.com/qor/admin
[18]https://www.go-admin.com/

that are better suited for such types of applications. Regarding the web interface, there are several popular web frameworks/libraries such as React[19], Angular[20] and Vue[21] that intend to facilitate the process of creating web applications. However, we didn't want to build something from scratch, for that, we performed a new search for framework/libraries that used those web technologies, but were specialized to build an admin interface. The solution found was React-Admin[22], a frontend framework to build data-driven applications. React [45] is a library that allows dividing a page into components that can be reused across the web application. Each component them as an internal state, and React will automatically update and render the associated components as the data changes. With React-admin, customizations are easily implemented by just creating new components and then adding them to their appropriate place. Also, such new components are implemented in a combination of HTML, Cascading Style Sheets (CSS) and Javascript, languages that are mainly used to build web interfaces.

Regarding the backend, the React-admin framework already takes care of the communication between the frontend app and the backend app, however, only a set of technologies are supported to use in the backend. One of the available technologies to use is Django, which must be used alongside the framework called Django REST framework[23], which allows exposing Django's models through an interface. Django was chosen over other technologies such as Springboot[24] or GraphQL[25] due to it managing the models by itself, creating migration files every time data models definition are changed on the Django app, allowing to keep both the business logic and the stored data models in a relational database in sync automatically. The database used here in conjuntion with Django will be PostgreSQL[26] since it is one of the most popular open-source and free RDBMS.

With this integration between the frontend app and the backend app set up by React-admin, when defining a page on the front end app, it is only necessary to define to which model is referred to, and then enumerate the wanted fields that must be displayed on the page. There is no need to implement calls to the API exposed by the backend app, as the React-admin framework will take of this automatically. Additionally, this framework supports relationships, which allows displaying extra data associated with a model, for example, if a database belongs to a community we can also display the Community information on the database page.
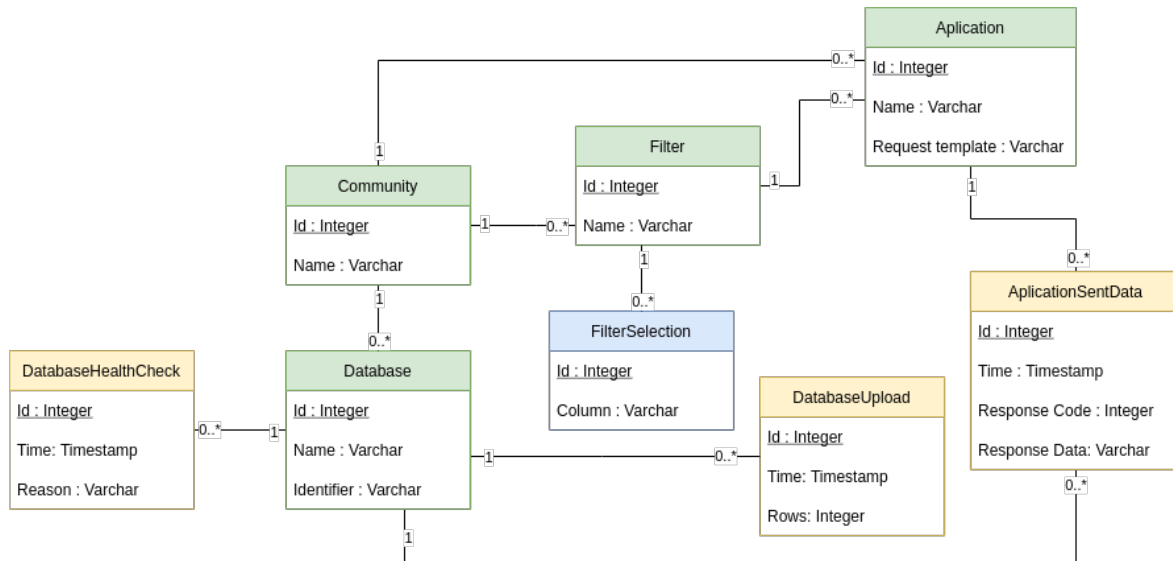
---

[19]https://reactjs.org/
[20]https://angular.io/
[21]https://vuejs.org/
[22]https://marmelab.com/react-admin/
[23]https://www.django-rest-framework.org/
[24]https://spring.io/projects/spring-boot
[25]https://graphql.org/
[26]https://www.postgresql.org/

Figure 4.17: Data models managed on the Admin Portal component

Concerning the models' organization, figure 4.17 shows the relationship between models of the Metadata Manager system. Models marks with green the admin can perform all CRUD operations, models in yellow are only for reading as they are created by the Statistics Recorder component, and models in blue are dependent models, which are defined when the main model is created, in this case, FilterSelection entries are created when the Admin is creating a Filter.

4.4.5 Statistics Recorder

Finally, this last component is in charge of reading messages that are sent between the components described before and persisting some of those messages as statistics of the system, so the admin can consult them on the Admin Portal. This could also be done on the components that send those messages, however, this way, such components are simpler and have well-defined objectives.

To avoid interfering with the data flow of other components, this Statistics Recorder component should consume messages from the necessary topics using a different consumer group, this way Kafka will send a message of a topic to both this component and the other components.

However, there is no need to develop a component from scratch to perform this. As we intend to move messages from Kafka into a relational database, we can make use of a Kafka Sink connector. A connector that allows achieving this is the JDBC Sink connector[27]. We are only required to set up Kafka Connect and launch a task using the JDBC Sink connector for each topic where the messages have to be persisted on the database.

4.5 SUMMARY

This chapter it was explained the entire development process around a system capable of automating the procedure of updating metadata stored on web applications about clinical

---

[27]https://www.confluent.io/hub/confluentinc/kafka-connect-jdbc/

databases. A solution was provided to the data owners to automatize the process of extracting metadata from their databases and then the system will make sure that that data will reach the desired web applications. A complete diagram of the system is present in Appendix A.

The next chapter will present the system developed in this chapter in action, using an installation of the Montra framework improved in chapter 3 as a target web application of the metadata.

Evaluation

- target applications, montra + dashboards
- step by step
- idialmente screenshots

5.1 Deploymente

não há necessidade em falar no kafka regestry. no diagrama de deployment, se o fizermos, simplesmente representamos o kafka como um unico container. eventualmente podemos dividir em kafka broker + kafka zookeper

5.2 Network Data Flow

- step by step

CHAPTER 6

Conclusion

After a thorough study on technologies to extract, mediate and visualize metadata, this work proposed a set of tools that cover and perform all these three actions.

It started by improving a fully-fledged tool for metadata visualization, making it more efficient and robust. Improvements range from making use of the correct data types to store data, enhancing the organization of data models and making use of the underlying framework used to build the system, instead of relying on custom code. As this platform has several active installations, performing such improvements was a challenge, since special attention was required along the development, so such installations could be easily migrated to the new proposed version with no data being lost or features being broken. At the time of writing, a pull request already exists on the repository of the Montra platform and is waiting for review, so all the implemented improvements can be integrated into production installations.

Next, an extraction process of metadata from databases conforming to the OMOP CDM was proposed, by making use of mainly existing tools. This part of the work was developed always following the mentality that the owner of a database has total control over its data, allowing him to control the regularity and automation of the extraction process. The main challenge encountered, was to develop a solution in a way that the owners of the databases could trust them in installing them on their production systems. Few custom components were created here, making use of already built tools with popularity, so the database owner could trust the proposed solution. The result is was a simple solution, that is still capable of achieving the required objective of extracting metadata from a database.

Finally, a system to gather the extracted metadata and send it to platforms exposing it was developed. The system was designed with a microservice approach, being composed of simple components, with well-defined objectives. Such components make sure that: only the required data is sent to the application, allowing to save computational resources; distribute the load of handling the received data from the several databases; allow customization of how data is sent to the applications; provide an intuitive and easy to use interface to manage the system; record statistics of data flowing the system to give better feedback to the person

managing the system. Designing this system with an easily scalable capability was a challenge, requiring analyzing what was the better data flow along the components, that could allow such scalable architecture. Hopefully, this system is used in future production systems or it contributes to the development of future systems with similar goals.

Along the work, new technologies were explored, such as the Go programming language. The overall experience was good, mainly due to its simplicity and also for feating so well on our use cases. Its building concurrency mechanisms are intuitive and powerful. Overall the language proves their growth in popularity.

As future work, as the only validation on data received from the agents it is to check if the database identifier provided is correct, Kafka authentication mechanisms could be used, removing this need to validate the received data. Additionally, when the user defines how data received from the databases should be filtered and how data should be sent to the applications receiving the metadata, a testing mechanism could be provided with some dummy data just for the admin to validate that the data is flowing in the desired format and no errors occur. Also, for now, the communication between the admin portal and the agents running the database owners system is only used to check if the agents are active, however, since the system was designed in a way to allow data to flow from the portal to the agents, it could be used for other extra uses. Finally, although the several components of the update system allow configuring the format of the data that they will manipulate, deployment of a system will only support a given format of metadata. One possible solution would be to define the format of the metadata when at the community level, on the Admin Portal component of the update system.

References

[1] S. Razick, R. Močnik, L. F. Thomas, E. Ryeng, F. Drabløs, and P. Sætrom, "The eGenVar data management system—cataloguing and sharing sensitive data and metadata for the life sciences," *Database*, vol. 2014, Jan. 2014. DOI: 10.1093/database/bau027.

[2] L. B. Silva, A. Trifan, and J. L. Oliveira, "MONTRA: An agile architecture for data publishing and discovery," *Computer Methods and Programs in Biomedicine*, vol. 160, pp. 33–42, Jul. 2018. DOI: 10.1016/j.cmpb.2018.03.024.

[3] *European health data evidence network*, https://www.ehden.eu/, Last Accessed on Oct. 2021.

[4] *Ehden - data partners*, https://www.ehden.eu/datapartners/, Last Accessed on Dec. 2020.

[5] *Ohdsi - data standardization*, https://www.ohdsi.org/data-standardization/, Last Accessed on Dec. 2020.

[6] J. ALMEIDA, A. TRIFAN, N. HUGHES, P. RIJNBEEK, and J. L. OLIVEIRA, "The european health data & evidence network portal,"

[7] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, *et al.*, "The FAIR guiding principles for scientific data management and stewardship," *Scientific Data*, vol. 3, no. 1, Mar. 2016. DOI: 10.1038/sdata.2016.18.

[8] A. Trifan and J. L. Oliveira, "Patient data discovery platforms as enablers of biomedical and translational research: A systematic review," *Journal of Biomedical Informatics*, vol. 93, p. 103 154, May 2019. DOI: 10.1016/j.jbi.2019.103154.

[9] J. L. Oliveira, A. Trifan, and L. A. B. Silva, "EMIF catalogue: A collaborative platform for sharing and reusing biomedical data," *International Journal of Medical Informatics*, vol. 126, pp. 35–45, Jun. 2019. DOI: 10.1016/j.ijmedinf.2019.02.006.

[10] A. Wright, "REDCap: A tool for the electronic capture of research data," *Journal of Electronic Resources in Medical Libraries*, vol. 13, no. 4, pp. 197–201, Oct. 2016. DOI: 10.1080/15424065.2016.1259026.

[11] I. Danciu, J. D. Cowan, M. Basford, *et al.*, "Secondary use of clinical data: The vanderbilt approach," *Journal of Biomedical Informatics*, vol. 52, pp. 28–35, Dec. 2014. DOI: 10.1016/j.jbi.2014.02.003.

[12] A. L. Vaccarino, M. Dharsee, S. Strother, *et al.*, "Brain-CODE: A secure neuroinformatics platform for management, federation, sharing and analysis of multi-dimensional neuroscience data," *Frontiers in Neuroinformatics*, vol. 12, May 2018. DOI: 10.3389/fninf.2018.00028.

[13] A. K. Green, K. E. Reeder-Hayes, R. W. Corty, *et al.*, "The project data sphere initiative: Accelerating cancer research by sharing data," *The Oncologist*, vol. 20, no. 5, p. 464, Apr. 2015. DOI: 10.1634/theoncologist.2014-0431.

[14] *Project data sphere*, https://data.projectdatasphere.org/, Last Accessed on Oct. 2021.

[15] M. A. Swertz, M. Dijkstra, T. Adamusiak, *et al.*, "The MOLGENIS toolkit: Rapid prototyping of biosoftware at the push of a button," *BMC Bioinformatics*, vol. 11, no. S12, Dec. 2010. DOI: 10.1186/1471-2105-11-s12-s12.

[16] M. T. Mayrhofer, P. Holub, A. Wutte, and J.-E. Litton, "BBMRI-ERIC: The novel gateway to biobanks," *Bundesgesundheitsblatt - Gesundheitsforschung - Gesundheitsschutz*, vol. 59, no. 3, pp. 379–384, Feb. 2016. DOI: 10.1007/s00103-015-2301-8.

[17] S. Gainotti, P. Torreri, C. M. Wang, *et al.*, "The RD-connect registry & biobank finder: A tool for sharing aggregated data and metadata among rare disease researchers," *European Journal of Human Genetics*, vol. 26, no. 5, pp. 631–643, Feb. 2018. DOI: `10.1038/s41431-017-0085-z`.

[18] O. Lancaster, T. Beck, D. Atlan, *et al.*, "Cafe variome: General-purpose software for making genotype-phenotype data discoverable in restricted or open access contexts," *Human Mutation*, vol. 36, no. 10, pp. 957–964, Aug. 2015. DOI: `10.1002/humu.22841`.

[19] D. Doiron, Y. Marcon, I. Fortier, P. Burton, and V. Ferretti, "Software application profile: Opal and mica: Open-source software solutions for epidemiological data management, harmonization and dissemination," *International Journal of Epidemiology*, vol. 46, no. 5, pp. 1372–1378, Sep. 2017. DOI: `10.1093/ije/dyx180`.

[20] P. McQuilton, A. Gonzalez-Beltran, P. Rocca-Serra, *et al.*, "BioSharing: Curated and crowd-sourced metadata standards, databases and data policies in the life sciences," *Database*, vol. 2016, baw075, 2016. DOI: `10.1093/database/baw075`.

[21] *Fairsharing*, `https://fairsharing.org/`, Last Accessed on Jan. 2021.

[22] *The dataverse project*, `https://dataverse.org/`, Last Accessed on Jan. 2021.

[23] *Nada | microdata cataloging tool*, `http://nada.ihsn.org/`, Last Accessed on Jan. 2021.

[24] *Software tools - ohdsi*, `https://www.ohdsi.org/software-tools/`, Last Accessed on Jan. 2021.

[25] *Github - ohdsi - achilles*, `https://github.com/OHDSI/Achilles`, Last Accessed on Jan. 2021.

[26] *Github - ehden - catalogueexport*, `https://github.com/EHDEN/CatalogueExport`, Last Accessed on Jan. 2021.

[27] X. Chen, A. E. Gururaj, B. Ozyurt, *et al.*, "DataMed – an open source discovery index for finding biomedical datasets," *Journal of the American Medical Informatics Association*, vol. 25, no. 3, pp. 300–308, Jan. 2018. DOI: `10.1093/jamia/ocx121`.

[28] S.-A. Sansone, A. Gonzalez-Beltran, P. Rocca-Serra, *et al.*, "DATS, the data tag suite to enable discoverability of datasets," *Scientific Data*, vol. 4, no. 1, Jun. 2017. DOI: `10.1038/sdata.2017.59`.

[29] A. N. Gonzalez-Beltran, J. Campbell, P. Dunn, *et al.*, "Data discovery with DATS: Exemplar adoptions and lessons learned," *Journal of the American Medical Informatics Association*, vol. 25, no. 1, pp. 13–16, Dec. 2017. DOI: `10.1093/jamia/ocx119`.

[30] S. C. Neu, K. L. Crawford, and A. W. Toga, "Sharing data in the global alzheimer's association interactive network," *NeuroImage*, vol. 124, pp. 1168–1174, Jan. 2016. DOI: `10.1016/j.neuroimage.2015.05.082`.

[31] M. Davies, K. Erickson, Z. Wyner, J. M. Malenfant, R. Rosen, and J. Brown, "Software-enabled distributed network governance: The PopMedNet experience," *eGEMs (Generating Evidence & Methods to improve patient outcomes)*, vol. 4, no. 2, p. 5, Mar. 2016. DOI: `10.13063/2327-9214.1213`.

[32] G. D. Moor, M. Sundgren, D. Kalra, *et al.*, "Using electronic health records for clinical research: The case of the EHR4cr project," *Journal of Biomedical Informatics*, vol. 53, pp. 162–173, Feb. 2015. DOI: `10.1016/j.jbi.2014.10.006`.

[33] T. J. Skluzacek, "Dredging a data lake," in *Proceedings of the 20th International Middleware Conference Doctoral Symposium*, ACM, Dec. 2019. DOI: `10.1145/3366624.3368170`. [Online]. Available: `https://doi.org/10.1145/3366624.3368170`.

[34] A. Pomares-Quimbaya, M. E. Torres-Moreno, and F. Roldán, "MetaExtractor: A system for metadata extraction from structured data sources," in *Availability, Reliability, and Security in Information Systems and HCI*, Springer Berlin Heidelberg, 2013, pp. 84–99. DOI: `10.1007/978-3-642-40511-2_7`. [Online]. Available: `https://doi.org/10.1007/978-3-642-40511-2_7`.

[35] T. J. Skluzacek, R. Kumar, R. Chard, *et al.*, "Skluma: An extensible metadata extraction pipeline for disorganized data," in *2018 IEEE 14th International Conference on e-Science (e-Science)*, IEEE, Oct. 2018. DOI: `10.1109/escience.2018.00040`. [Online]. Available: `https://doi.org/10.1109/escience.2018.00040`.

[36] *Django project*, `https://www.djangoproject.com/`, Accessed on Jul. 2021.

[37] *Client-side form validation*, `https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation`, Accessed on Aug. 2021.

[38] *Ehden - networkdashboads*, `https://github.com/EHDEN/NetworkDashboards`, Last Accessed on Oct. 2021.

[39] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ," ACM, Jun. 2017. DOI: `10.1145/3093742.3093908`. [Online]. Available: `https://doi.org/10.1145/3093742.3093908`.

[40] *Stack overflow developer survey 2020*, `https://insights.stackoverflow.com/survey/2020`, Last Accessed on Oct. 2021.

[41] *Redhat - what is virtualization*, `https://www.redhat.com/en/topics/virtualization/what-is-virtualization`, Last Accessed on Oct. 2021.

[42] *Containerization explained | ibm*, `https://www.ibm.com/cloud/learn/containerization`, Last Accessed on Oct. 2021.

[43] *Github - confluentinc/ksql*, `https://github.com/confluentinc/ksql`, Last Accessed on Oct. 2021.

[44] *R: The r project for statistical computing*, `https://www.r-project.org/`, Last Accessed on Oct. 2021.

[45] *React – a javascript library for building user interfaces*, `https://reactjs.org/`, Last Accessed on Oct. 2021.

Agent and Metadata Manager Data flow
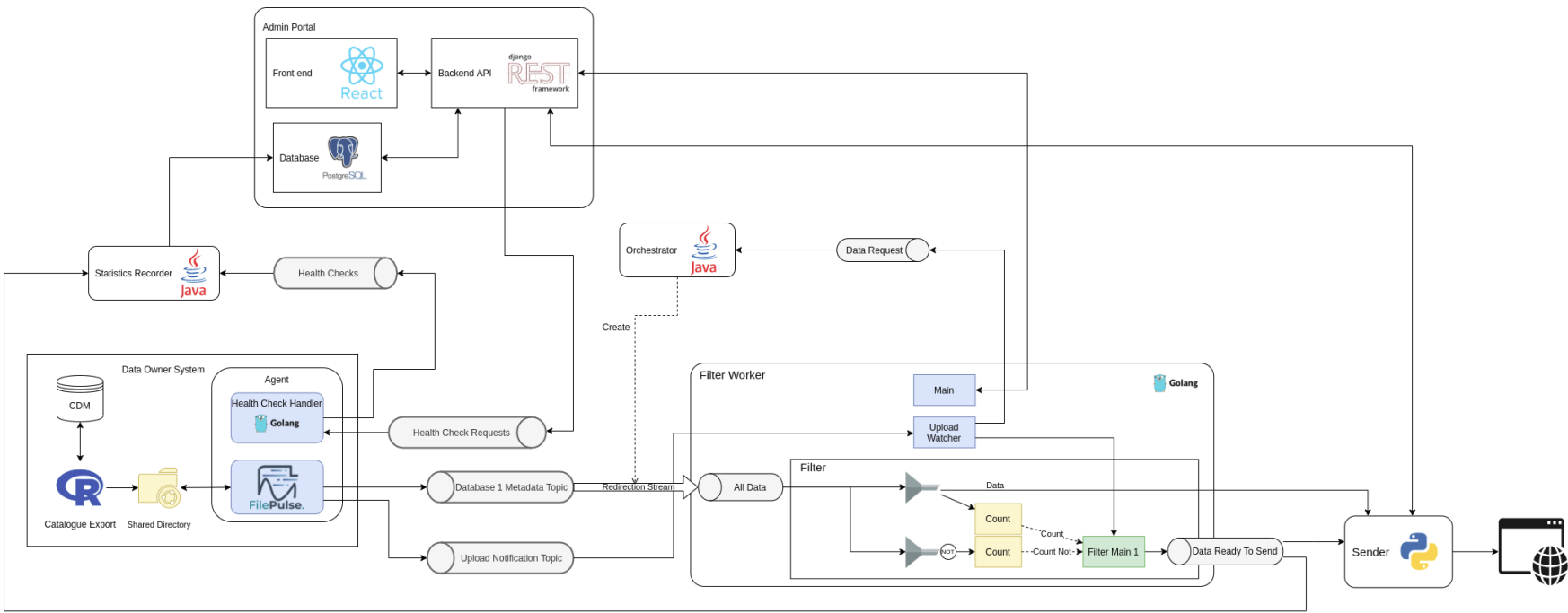
Figure A.1: The architecture of the entire Metadata manager and its interaction with the agent