Software Design Document

Automatic Audio Labeler for Speaker Verification (A2LSV)

Project Owner

Ali AGDENIZ

Contents

FIGURES CONTENT

1. Introduction

1.1. Purpose

The purpose of this document is to describe the implementation of the Automatic Audio Labeler for Speaker Verification (A2LSV) Software described in the A2LSV Requirements. The A2LSV Software is designed to create speaker verification datasets easily.

1.2. Scope

This document describes the implementation details of the A2LSV Software. The software will consist of a three major parts. First is web application that the labelings are done, the second is the database for storing the labels, speaker encodings, audios, users and the third is kafka system to automate the processes which are downloading and diarizing speaker of audios.

2. Design Overview

2.1. Description of Problem

For creating speaker verification software, we need suitable dataset. For English language, there are suitable datasets but for other languages it is hard to find public available dataset or creating your own dataset. This software will make it easy to create datasets with automatically downloading audios and automatically pre-labeling speakers.

2.2. Technologies Used

A2LSV will be a web application supported with Django, Kafka and MongoDB. The programming language will be Python. The target platform will be Ubuntu, and the development environment is Spyder.

Here is the list of used frameworks, techs, tools and API services:

django, mongodb, pytorch, Apache Kafka, YouTube Data API, youtube-dl, ffmpeg, webrtcvad.

2.3. System Architecture

Figure 1 depicts the high-level system architecture. The system will be constructed from multiple distinct components:

• **Django:** The web framework for serve users a web server to access and use system.

• **MongoDB:** The database for storing the labels, speaker encodings, audios and users.

• **Apache Kafka:** The distributed messaging system to automate the downloading and diarizing speaker of audios.

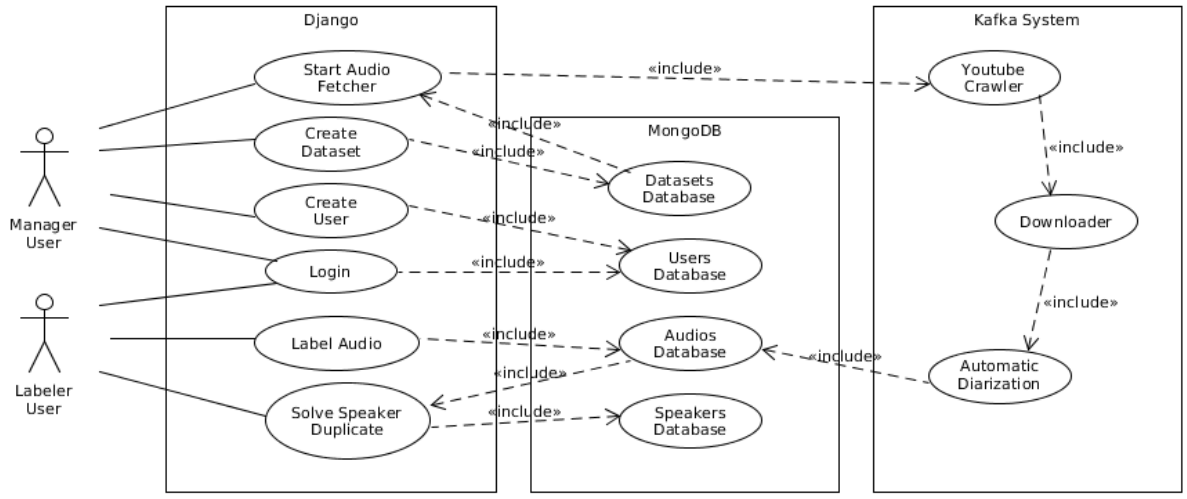

Figure 1 A2LSV Architecture

2.4. System Operation

Figure 2 is the typical sequence of events that occur during A2LSV labeling.

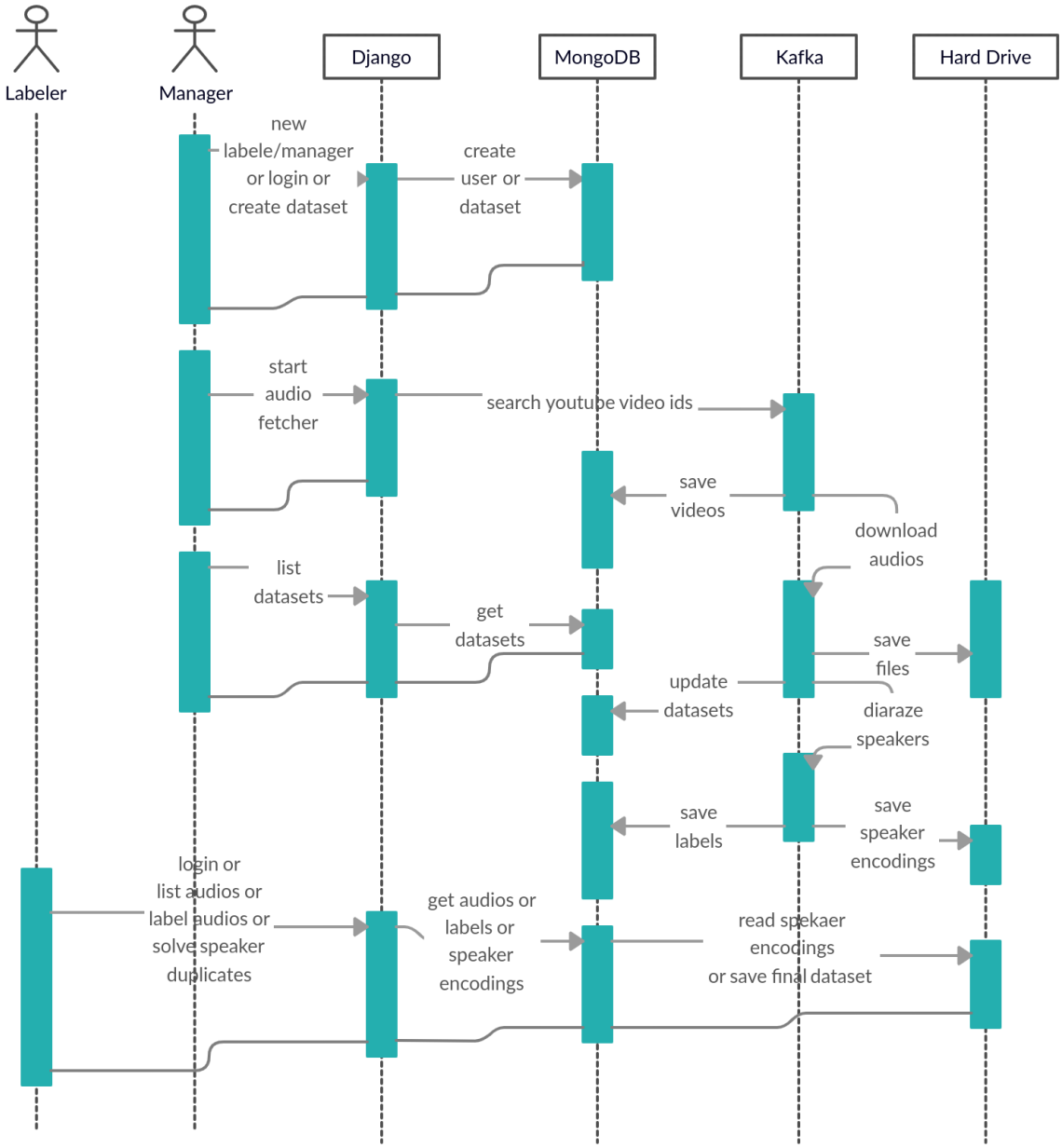

Figure 2 A2LSV Sequence Diagram

3. Django Module and Interface Description

This project will be using django at its core. This allows me to set up the website using a commonly used MVC framework. The MVC framework consists of 3 main components; Model, View, and Controller. The model portion of the framework consists of all of the classes that we will need for the project. The View is basically what will appear on the webpage. The controller is what links the Model and the View together.

In Django the underlying MVC architecture is actually slightly different from the classic MVC approach. Models are still Models in Django, but a View is actually called a Template, and a Controller is actually called a View. This means that a Django Template is actually what you see on the webpage, and a View links the classes in a Model together with a Template.

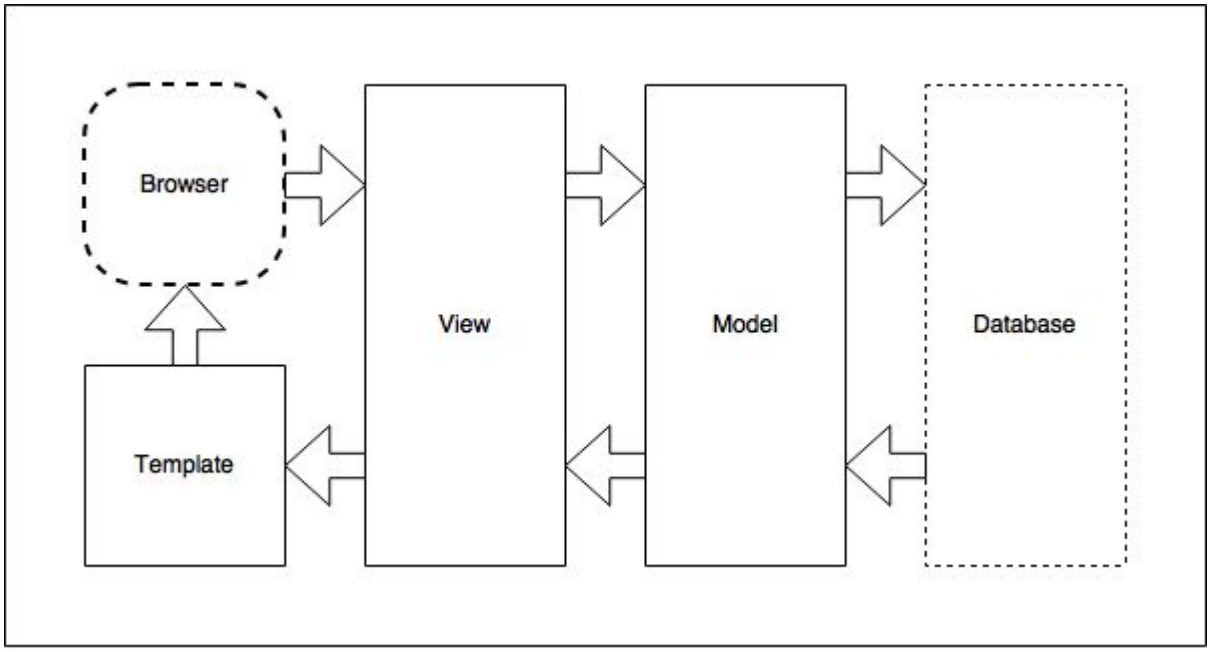

Figure 3 Django MTV Diagram

3.1. Flow of data in a Django MTV framework

Figure 3 shown above showcases a brief overview of the flow of data in a basic Django application, beginning with a request from a browser and resulting in a web page produced back to the browser.

When a request is made to view a webpage provided by a Django application, it is first referenced in a list of url patterns located in a file “url.py”. The url patterns in this file will link directly to the View portion of the MTV framework by accessing a file called “views.py”

The file “view.py” basically holds all of the functionality for the Django application (which explains why we reference these as “controllers”), and uses the classes defined in your Model to manipulate the data before sending the data to a template.

The Model keeps all of the models in a file labelled “models.py”. Once a class is defined in this file, any objects created from each class will automatically be added to the defined database that is

maintained inside the Django app. The requested data from the database will then be returned back to the View, and then returned to template.

Templates are used to dictate how the processed data will look on a webpage after it has been requested. A template consists of all of the basic utilities that can be included in any html document. Each page in a Django project will require its own template.

3.2. Urls.py

The url patterns in the “urls.py” file include “web_interface.urls”, “admin.site.urls”, “django.contrib.auth.urls”. Each url pattern sends a request to the View which calls a function by the same name in the “views.py” file.

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('web_interface.urls')),
    path('accounts/', include('django.contrib.auth.urls')),
    path('accounts/signup/manager/', manager.ManagerSignUpView.as_view(), name='manager_signup'),
    path('accounts/signup/labeler/', manager.LabelerSignUpView.as_view(), name='labeler_signup'),
]
```

Figure 4 main urls

The url patterns in the “web_interface.urls” file include the paths for labeler, manager and home paths with the View to be redirect.

```
urlpatterns = [
    path('', web_interface.home, name='home'),

    path('labeler/', include(([
        path('', labeler.listAudios, name='listAudios'),
        path('labelAudio', labeler.labelAudio, name='labelAudio'),
        path('solveSpeakerDuplicate', labeler.solveSpeakerDuplicate, name='solveSpeakerDuplicate'),
    ], 'web_interface'), namespace='labeler')),

    path('manager/', include(([
        path('', manager.listDatasets, name='listDatasets'),
        path('dataset/add/', manager.DatasetCreateView.as_view(), name='dataset_add'),
        path('keyword/add/', manager.get_keyword, name='keyword_add'),

    ], 'web_interface'), namespace='manager')),
]
```

Figure 5 web_interface urls

3.3. Models.py

In Django, database tables are created via python classes. These classes are individually referenced as a “model” and all together we call our database entries the “Models”. In the A2LSV System we currently have three models which are the Datasets, Languages, and User.

Figure 6 depicts the UML class diagram for the all Django Models.

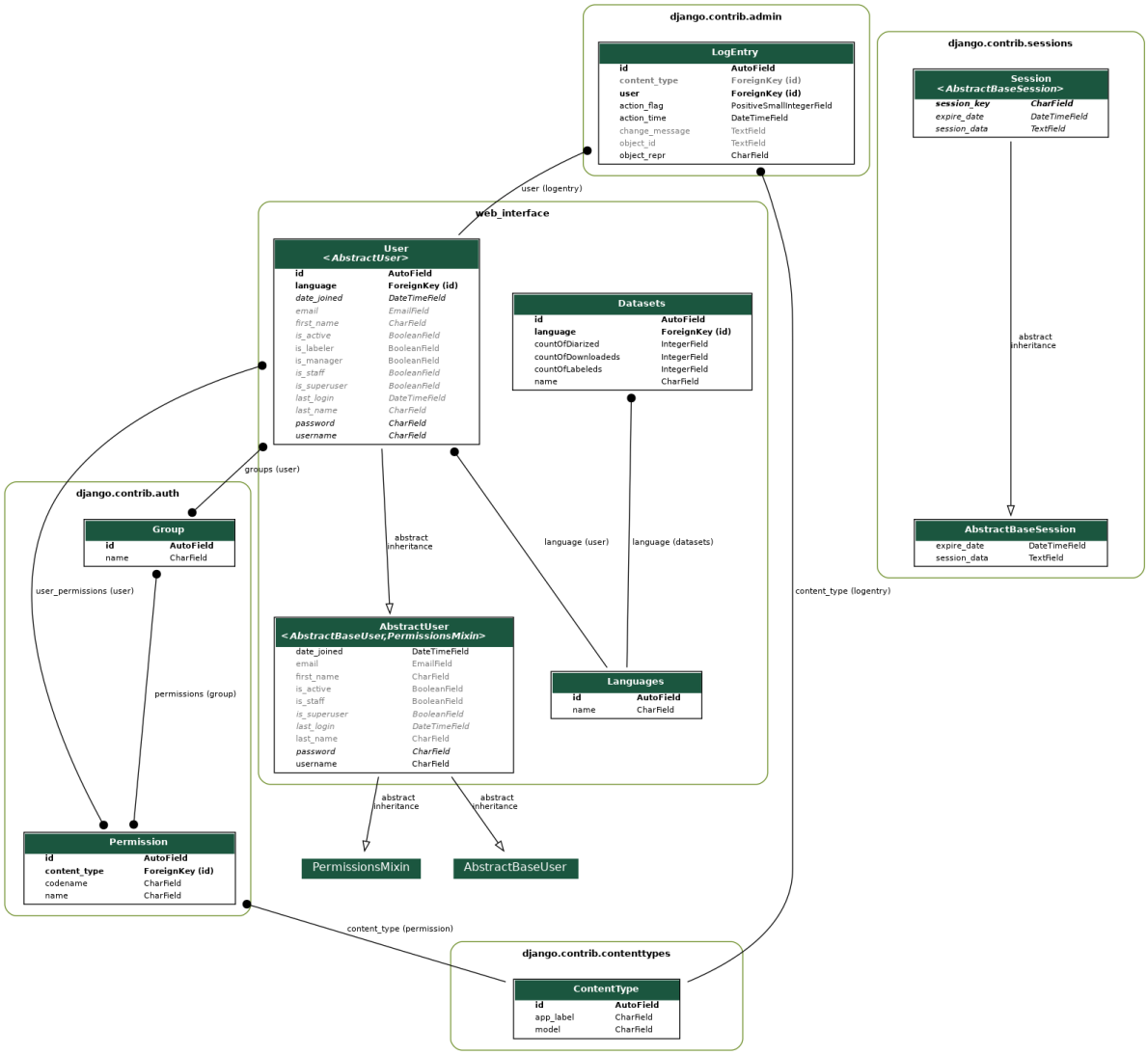

Figure 6 Django Models

Figure 6 depicts the UML class diagram of “Models.py”.

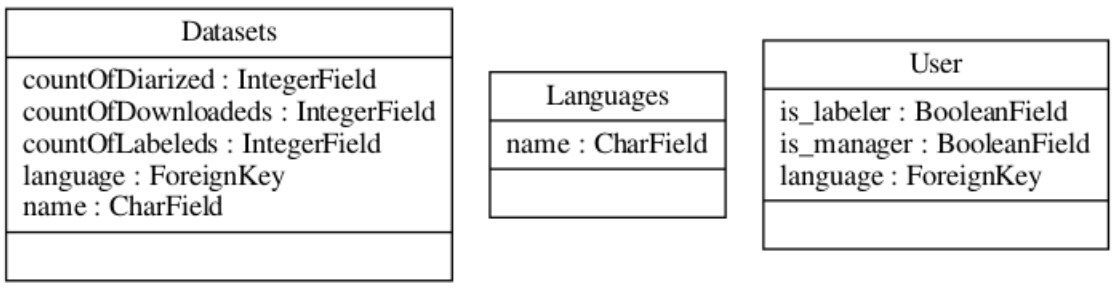

Figure 7 Class Diagram of Models.py

3.3.1.Datasets Model

The fields for the Datasets Data are “name”, “language”, “countOfDownloadeds”, “countOfDiarized”, “countOfLabeleds”. This model is displayed within our template so that managers can view the datasets data at anytime. Kafka producers consumers will update these fields.

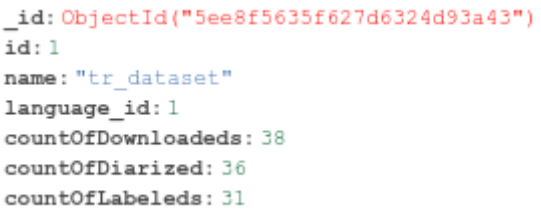

Figure 8 Sample Dataset Data

3.3.2. Languages Model

The fields for the Languages Data is just “name”. This field also exists as a foreign key in Datasets and User models. This field will be used when searching youtube video ids and listing audios for users. This model is displayed within our template so that managers can view the languages data at datasets creation step.

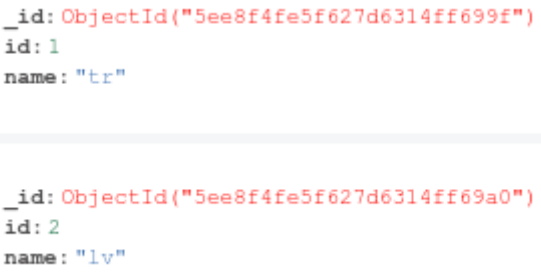

Figure 9 Sample Language Datas

3.3.3. User Model

This model is AbstractUser model from Django. So it inherits some fields from Django. The fields we add for the Datasets Data are “is_labeler”, “is_manager”, “language”. After user logged in, “is_labeler”, “is_manager” will be used to identify the user. This model is displayed within our template so that managers can create new users.

```
_id: ObjectId("5ee8f4fe5f627d6314ff699e")
id: 1
password: "pbkdf2_sha256$100000$T32SloPSm4Aw$l6zHl51eE3V7/IM5LDO7aRtIhLxWDR3baYlt..."
last_login: 2020-06-18T08:58:38.652+00:00
is_superuser: false
username: "manager"
first_name: ""
last_name: ""
email: ""
is_staff: false
is_active: true
date_joined: 2020-06-07T13:15:38.059+00:00
is_labeler: false
is_manager: true
language_id: null
```


Figure 10 Sample User Data

3.4. Views

We have 3 views. These are “web_interface”, “labeler” and “manager”. Functions in these views return data to specific web pages within our web application. Each of the functions in the View are referenced by the “urls.py” file, and are called after being requested by the corresponding URL address in the Django application.

The View is not responsible for how the data is displayed, but rather what data will be displayed. The View is the section in which all of the functionality for the web application will reside. It is responsible for requesting queries from the database, and then manipulating and organizing the data before passing it off to a template for displaying to a browser.

3.4.1. web_interface View

“home” function checks if user is authenticated. If yes, redirects to manager or labeler home page based on the user type.

```
def home(request):
    if request.user.is_authenticated:
        if request.user.is_manager:
            return redirect('manager:listDatasets')
        else:
            return redirect('labeler:listAudios')
    return render(request, 'web_interface/home.html')
```

Figure 11 web_interface View

3.4.2. Manager View

“listDatasets” function check user, fetch datasets data from mongoDB database and list dataset that are available.

“get_keyword” function check user and starts audio fetcher with the keyword input from manager.

“ManagerSignUpView” class check user and create a view for creating new manager.

“LabelerSignUpView” class check user and create a view for creating new labeler.

“DatasetCreateView” class check user and create a view for creating new dataset.

```
@method_decorator([manager_required], name='dispatch')
class ManagerSignUpView(CreateView):

@method_decorator([login_required, manager_required], name='dispatch')
class LabelerSignUpView(CreateView):

def listDatasets(request):

def get_keyword(request):

@method_decorator([login_required, manager_required], name='dispatch')
class DatasetCreateView(CreateView):

@method_decorator([login_required, manager_required], name='dispatch')
class DatasetUpdateView(UpdateView):
```

Figure 12 Manager View

3.4.3. Labeler View

“listAudios” function check user, fetch audios data filtering by user language from mongoDB database and list audios that are available.

“labelAudio” function check user and if request is a GET request redirect to labeling page for selected audio. If request is a POST request, save final labeling to database and redirect to solving speaker duplicate page.

“solveSpeakerDuplicate” function check user and redirect to solving speaker duplicate page for selected audio.

```
def listAudios(request):

def labelAudio(request):

def solveSpeakerDuplicate(request):
```

Figure 13 Labeler View

3.5. Templates

The templates used in the Django application are typical of any webpage in that they can be written in HTML, and use tools to augment the page such as CSS and JavaScript. The fields in the Model classes are called and manipulated by the View which results in a query of instances from the Model classes. These instances can then be utilized in a template by referencing the returned results from the View. This makes the content displayed in the webpage to be dynamic.

In a Django application, the html content you view is not always static, which is great for modern dynamic web applications. Templates can be fed dynamic queries that are requested by the View before the pages are actually displayed to the user. For example, every time a user visits the “list_audios.html” template within our project, Django’s templating system will go and fetch the latest database information to be loaded into our template. This is a very powerful tool, because this allows our index page to always display the latest data from the database, without the need for a POST request or a page refresh.

4. MongoDb

MongoDB is a document database with the scalability and flexibility with the querying and indexing . MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time. The document model maps to the objects in your application code, making data easy to work with. Ad hoc queries, indexing, and real time aggregation provide powerful ways to access and analyze your data. MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use.

4.1. User Collection


```
_id: ObjectId("5ee61c975f627d39e013976a")
id: 1
password: "pbkdf2_sha256$100000$T32SloPSm4Aw$16zH151eE3V7/IM5LDO7aRtIhLxWDR3baY1t..."
last_login: 2020-06-15T07:29:28.710+00:00
is_superuser: false
username: "manager"
first_name: ""
last_name: ""
email: ""
is_staff: false
is_active: true
date_joined: 2020-06-07T13:15:38.059+00:00
is_labeler: false
is_manager: true
language_id: null
```


Figure 14 Sample User Record

is_labeler: The boolean value indicates that user is a labeler.

is_manager: The boolean value indicates that user is a manager.

4.2. Dataset Collection


```
_id: ObjectId("5ee8f56b5f627d64d81fb74f")
videoId: "xliHVgo-Peo"
channelId: "UC6mGWkXiLSB6RSQR2RTANVQ"
titles: "KOMPLE ATLET LİGİ VE SALON OLİMPİK DENEME YARIŞMALARI 05 OCAK 2020 İst..."
channelTitles: "TAF TV"
language: "tr"
downloaded: true
speakersDiarized: true
labelsConfirmed: true
jsonSpeakerSlice: "{"2": [1, 2, 4, 5, 8, 14], "1": [9, 15, 18]}"
speakersAdded: true
labelList: "{"1":"2","2":"2","4":"2","5":"2","8":"2","9":"1","14":"2","15":"1","18..."
```


Figure 15 Sample Audio Record

“videoId”: Video ID of downloaded youtube video

“channelId”: Channel ID of downloaded youtube video

“titles”: Titles of downloaded youtube video

“language”: Language of downloaded youtube video

“downloaded”: The boolean value indicates that downloading is done

“speakerDiarized”: The boolean value indicates that pre-labeling is done

“jsonSpeakerSlice”: The dumped string of labelings json object.

“labelsConfirmed”: The boolean indicates value that user confirmed the labelList.

“speakersAdded”: The boolean indicates value that speaker in this audio is added.

“labelList”: The dumped string of labelings that user confirmed.

4.3. Datasets Collection

Figure 16 Sample Dataset Record

“Name”: Name of dataset

“language_id”: Foreign key of language

“countOfDownloadeds”: Total count of download audios

“countOfDiarized”: Total count of pre-labeled audios

“countOfLabeleds”: Total count of confirmed labelings

4.4. Speakers Collection

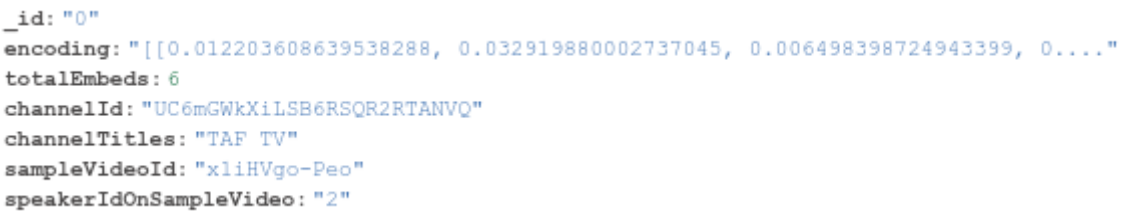

Figure 17 Sample Speaker Record

encoding:	The dumped string of encoded speaker values.
totalEmbeds:	Total count of how many audios used to calculate encoding.
channelId:	The Youtube Channel Id where speaker is added.
channelTitles:	The Youtube Channel Titles where speaker is added.
sampleVideoId:	The sample Video ID where speaker is added.
speakerIdOnSampleVideo:	The labeled speaker ID where speaker is confirmed on the sample video.

4.5. Languages Collection

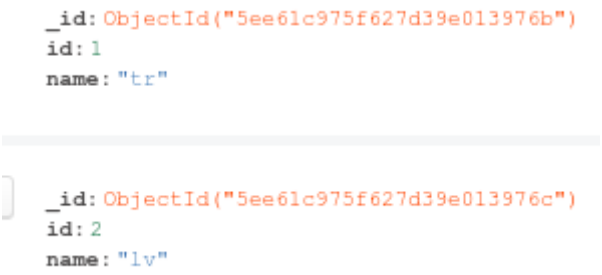

Figure 18 Sample Language Records

“name”: Language code for Youtube Data API Search.

5. Kafka

Apache Kafka is a distributed streaming platform. A streaming platform has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.

Kafka is generally used for two broad classes of applications:

- Building real-time streaming data pipelines that reliably get data between systems or applications
- Building real-time streaming applications that transform or react to the streams of data

Some concepts of Kafka:

- Kafka is run as a cluster on one or more servers that can span multiple datacenters.
- The Kafka cluster stores streams of records in categories called topics.
- Each record consists of a key, a value, and a timestamp.

We will use 2 core API of Kafka:

- The Producer API allows an application to publish a stream of records to one or more Kafka topics.
- The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them.

5.1. Youtube Searcher Comsumer Producer

The consumer part listens for “searchByKeyword” topic. The values passed to this topics are dataset name, keyword and language. We use “Youtube Data API” to search releated videos to only given language. We use “relevanceLanguage” and “regionCode” to filter result according to given language, but “Youtube Data API” does not guarantee that. So we found the exact related videos, those video informations are save to related dataset collection.

```
_id: ObjectId("5ee8f50f5f627d638e856bb0")
videoId: "g9W2w0kdLA0"
channelId: "UCz2NLNVkWnYQyJypCNG455Q"
titles: "KPSS 2020 GK Vatandaşlık Denemesi (Canlı Yayın)"
channelTitles: "Sosyal Hocam"
language: "tr"
downloaded: false
speakersDiarized: false
labelsConfirmed: false
speakersAdded: false
```


Figure 19 Sample Audio Record

The producer part publishs a message with “downloadAudios” topic. The values are dataset name and videoId.

5.2. Youtube Audio Downloader Comsumer Producer

The consumer part listens for “downloadAudios” topic. The values passed to this topics are dataset name and videoId. We use “youtube-dl” to download youtube videos from videoId as audio. Then we use “webrtcvad” pthon library to detect activated audio parts. To process these audios, we need to convert audio format to wav,sample rate 1600. Also we need to merge the channels. We use “ffmpeg” for this process. Then we do the detection part and save the activated audio parts to local hard drive. The location is based on the dataset name and videoId. Then we update ‘downloaded’ record to ‘true’ in the related dataset collection.


```
_id: ObjectId("5ee8f56b5f627d64d81fb74d")
videoId: "d3PDY7qx8-w"
channelId: "UC7gEEcPlzeHul4_g2CVcSEA"
titles: "ÜCRETSİZ TYT DENEMESİ - COĞRAFYA ÇÖZÜMLERİ"
channelTitles: "Muba Yayınları"
language: "tr"
downloaded: true
speakersDiarized: false
labelsConfirmed: false
speakersAdded: false
```


Figure 20 Sample Downloaded Audio Record

The producer part publishs a message with “diarizeAudio” topic. The values are dataset name and videoId.

5.3. Automatic Diarization Consumer

This consumer listens for “diarizeAudio” topic. The values passed to this topics are dataset name and videoId. In this part, we need to diarize speakers automatically and save the pre-labeling values to related dataset collection. Also we need to save the speaker encodings to hard drive. So we don’t need to process those audio part again and again.

For automatic diarization, we used a speaker encoder module of “Real time voice cloning” project. You can access more info from this link: https://github.com/CorentinJ/Real-Time-Voice-Cloning

After calculating the speaker encodings, we used Umap non-linear dimension reduction algorithm to calculate speaker projections. Then we used hierarchical clustering method to automatically label the speakers. Then we saved these pre-labeling values as “jsonSpeakerSlive” to related audio record in related dataset collection. Also updated the “speakersDiarized” record to ‘true’. Now labeler users can see this audios and start to control and fix labels.


```
_id: ObjectId("5ee901ad5f627d7a0a92d985")
videoId: "1KsZauQSX7E"
channelId: "UCYQA9sMy3CGrGU_fYCDZk_Q"
titles: "Bağlama virtüözü Erdal Erzincan&#39;dan korona mesajı!"
channelTitles: "Semra Topcu"
language: "tr"
downloaded: true
speakersDiarized: true
labelsConfirmed: false
speakersAdded: false
jsonSpeakerSlice: "{"2": [6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 18, 20, 21, 22, 23, 25, 26,..."
```


Figure 21 Sample Diarized Audio Record

6. Conclusion

I feel that this project has great potential and very useful for who want to create their speaker verification dataset. This system will enable multiple user them to label audios at the same time in a faster, easy and effective way.