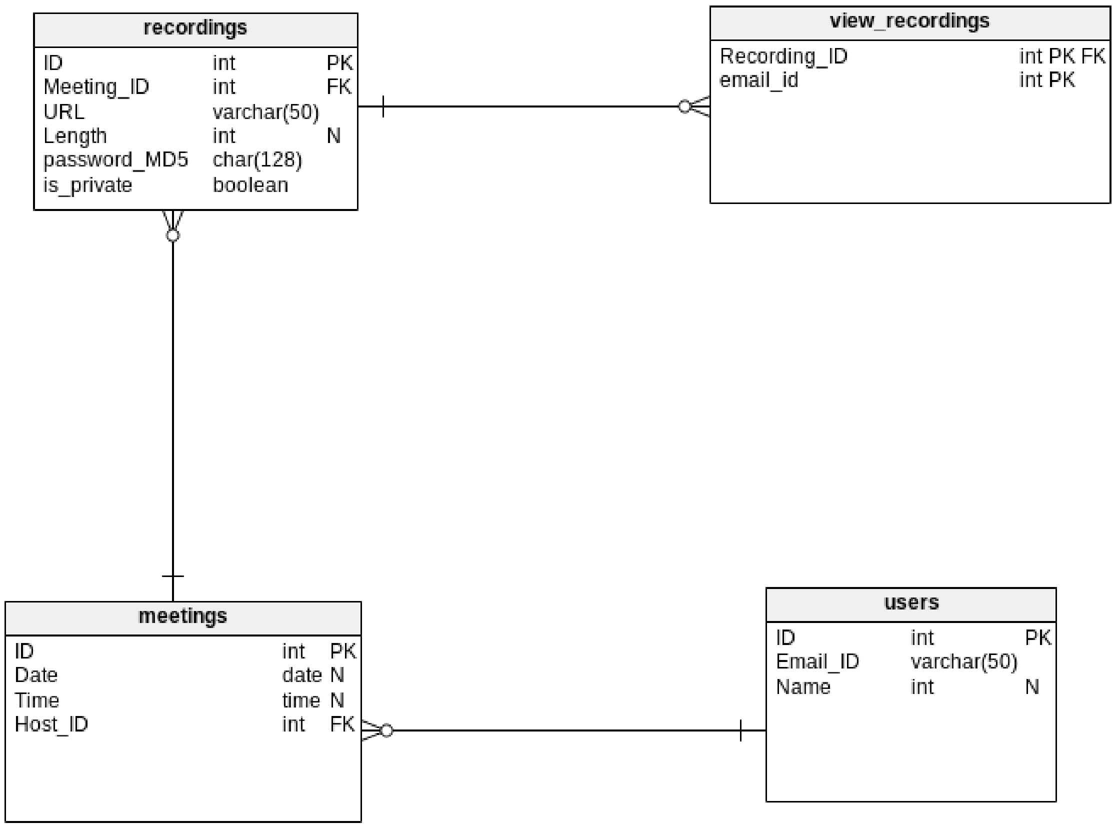
Meetings Backend REST Service

1. Introduction:

REST architechure style and APIs are ubiquitous in recent scalable, distributed systems.
It has various advantages over other api styles in that it is light weight, very less dependency between services etc.
In our meetings backend service, using Python and Flask, we have developed a service that provides few useful APIS to meeting backend service to manage recordings that were created during a meeting. In the next few sections we will discuss the data model and various APIS developed as part of this design.
The design of these APIs was done in such a way that they provide various useful functionalities as well as robustness to the services consuming these apis. Also, certain apis have been provided for bookkeeping, internal usecase purposes. One such example is the **GET api** of ViewRecording to see the contents of the ViewRecording table.
With this design document, I hope to provide clarity on the data models and various APIs that has been exposed.

2. Data Model:

For the purpose of this design, I have created four tables namely **users, recordings, meetings and view_recordings.** Sqlite is used as the backed db and sqlalchemy is used to manipulate these tables from within our python files. Below you can find the diagram of the tables and their relationships.



Users table:

This table deals with registered users or hosts who have access to create meetings. It includes details about a particular user in a row. ID is generated dynamically and email_id is required for any user. That is email ID cannot be null.

meetings table:

This table has various columns. Most important of which is Host_ID which is the foreign key from the table Users. This table records details about various meetings created by a host.

Recordings table:

Recording table has meeting_id as foreign key. The idea here is that in a single meeting, host can create multiple recordings. Each recording has various information associated with it. URL is a non null entity that points to the location of the recording media in S3. By default each recording is private. That is it is accessible only by the host who created the meeting.

View_recordings table:

This table keeps track of recording ids which is a foreign key, and a list of email ids of viewers who has access to that particular recording.

3. API Specification:

/User

Method:

POST

Request Body :

```
{
    "Name" : "String",
    "email_id": "String"
}
```

Description:

This api is used to create Users. email_id cannot be null in the request Body. When a service hits this api with a proper request body, an entry is created in users table for future reference. We can treat this as registering a user to the system.

/User

Method:

DELETE

Request Body:

```
{
    "id": number,
    "email_id" : "String"
}
```

Description:

This API deletes User from the database and also deletes all the meetings and recordings that this host is associated with. This is because, without host the foreign key referencing this will be null in those tables and lose their meaning. That is why these are deleted on cascade.

/User

Method:

GET

Description:

This API is for internal purpose only. We can use this API to query all the users in the Users table.

/Meeting

Method:

POST

Request Body:

```
{
    “host_id”: Number
    “meeting_date” : “datetime”
}
```

Description:

This API is used to create a meeting entry in the table meetings. For this api, request body must contain host id, which is id field of users table. This can be obtained by the GET method of /User. Meeting_date is optional (for simplicity purpose)

/Meeting

Method:

DELETE

Request Body:

```
{
    “id” : “String”
}
```

Description:

The API is used to delete a meeting given you know the meeting id. Just in the case of /User, you can get the meeting id using GET method on /Meeting.

/Recording

Method:

POST

Request Body:

```
{
    “meeting_id” : Number,
    “url” : “Valid URL”,
    “md5_hash”: “String”,
    “length”: Number,
    “private”: Boolean
}
```

Description:

This is a major API. This api can be used to create a recording by sending meeting_id associated with the recording and the url of the recording media. Other parameters are optional. Private by default is true. That means each recording is private by default.

/GetRecordingsFromMeeting

Method:

POST

Request Body:

```
{
    “meeting_id”: Number
}
```

Description:

This api returns all the recordings belonging to a certain meeting id. This is useful in getting a recording and then utilizing the recording_id to delete, share, or check access to the recording.

/ShareRecording

Method:

POST

Request Body:

```
{
    “recording_id”: Number,
    “email_id” : “String”
}
```

Description:

This API is used to share a recording with viewers identified by their email. The API does not allow sharing if the recording identified by recording_id is **private.** Services calling this api must make sure to authenticate the user before he can call this api.

/CanAccessRecording

Method:

POST

Request Body:

```
{
    “recording_id”: Number,
    “email_id”: “String”
}
```

Description:

Client application can use this API to check if a particular viewer identified by an email address has access to a particular recording. This queries view_recording table in the backend to get that information.

/Recording

Method:

DELETE

Request Body:

```
{
    “recording_id”: Number
}
```

Description:

This api is used to delete a recording from the table. All the viewers associated with this recording will also be deleted.

/Recording and /Meeting also has **GET** method defined for internal uses of the services consuming these apis. However care should be taken by those services to not expose the data outside.

All of the apis above return the **response** in json format.

Some of the **status codes** used in the api:

200 – Success ok

201 – Success created

400 – Failure Bad Request

422 – Failure Unprocecessable Entity

4. Assumptions:

In such a design, there is bound to be assumptions made due to various factors like time constraint, requirement specifications constraint, domain constraint etc. Below are some of the assumptions I have made while designing the system.

1. Tables have been defined very minimally assuming that the requirement is to keep the design simple. As we start extending it, these tables can be defined properly.
2. The main responsibility of the system is to provide APIs so that another service can consume it(namely meeting service). The design does not involve any business logic, or authentication mechanisms.
3. We have assumed that when we want to **share a recording,** the main idea is to add that email address with whom we are sharing to a table so that he can access the recording. The api responsible for this does not handle the logic of presenting the recording to the user.
4. Service consuming these apis have access to ids corresponding to an user, recording or meeting. We have exposed GET methods for the service to use.
5. Only **host** can create, share a recording to a viewer or delete the recording.

5. Instructions and test cases:

Instructions on how to set up the service locally and also how to run the tests will be provided in a README. All the major apis we have discussed above has corresponding test cases. I have made use of tavern framework to write and run these test cases. I also used POSTMAN to test my implementation.

There are five test files which contain 18 test cases and each test case has 1 or more stages within the test case. For example, negative test cases for an API are combined into 1 test case. The five files are namely:

test_0_createapis.tavern.yaml

test_1_getrecordingfrommeeting.tavern.yaml

test_2_sharerecording.tavern.yaml

test_3_checkrecordingaccess.tavern.yaml

test_deleteapis.tavern.yaml

The first test file deals with creating various resources including Recording, Meeting, User. Next three test files deal with testing share recording, check user access, get recording apis. The last file deletes and also tests negative scenarios in the case of delete of all the previously created test cases.

6. Example Scenario:

Let us now look at a scenario where these apis can be used. Let us assume we have meetings service which will be consuming these apis.
Meetings service allows a user to register and using our **/User POST** API, his information will be saved. Once the user is registered, He has the ability to create a meeting as host using **/Meeting POST** request. While a meeting is going on, host can create recording using **/Recording POST** which host can set as private or public depending on his choice. If it is private only the host can access the recording. Otherwise, host can share the recording with any number of viewers identified by their email address using **/ShareRecording**. Additionally the service can check if a particular user has access to the recording using **/CanAccessRecording**. Once host decides that host no longer needs the recording, host can ask the service to delete the recording using **/Recording DELETE.**

Service can also consume **GET** api to query various tables. **/GetRecordingsFromMeetings** be used to identify recording associated with a meeting.

This is a typical flow of usage of apis from the service consuming them.