Audio Ember

ROBERT GONZALEZ AND MICHAEL WOODHAM

12/04/2016

Introduction

Our database systems project is a web app called Audio Ember which fetches Billboard charts then sorts, scores, and graphs the data for the user to interact with. This project was agreed upon because Billboard has a wealth of raw data about popular music, and pop culture over time, but provides it to the end user in a rather limited fashion [1]. This project is just the beginning as only 6 years of selected chart data was sampled. In the future we would like to expand the database to include years 1990 - Present as well as international music charts. The key components of our project are the scoring mechanism which enables us to rank songs and artist over time, as well as the information derived from the Billboard charts. Throughout the project the consistent use of github was immensely helpful and consequently all our code is located there. However as our website is live for the sake of security and potential entrepreneurial power the code is private but can be made public for grading upon request [2][3].

Database Details

We designed the database with the explicit intention of reducing redundancies. As a result, our Database is in BCNF. To accommodate for functional dependencies we did the following. As shown in Figure 1 our model has 4 separate tables. The first table is a song table which stores the SongID as a Primary Key, and has a related Artist and Title to go along with the specific SongID. The second table is a points table which holds a Primary Key of the song's SongID with 2 additional Primary Keys of the Date that the song was on the specific Genre chart, and the specific Genre chart. The two additional attributes are Rank and Points. Rank is the rank that the song holds during the current week on the specific chart. Points are the accumulation of ranks brought about by our formula which determines the total points that the song has generated across all weeks up to the current week. As a result a song which was on the hot-100 on 10-31-2016 will hold less points than that same song when it was on the hot-100 chart a week prior. Therefore, the date determines both

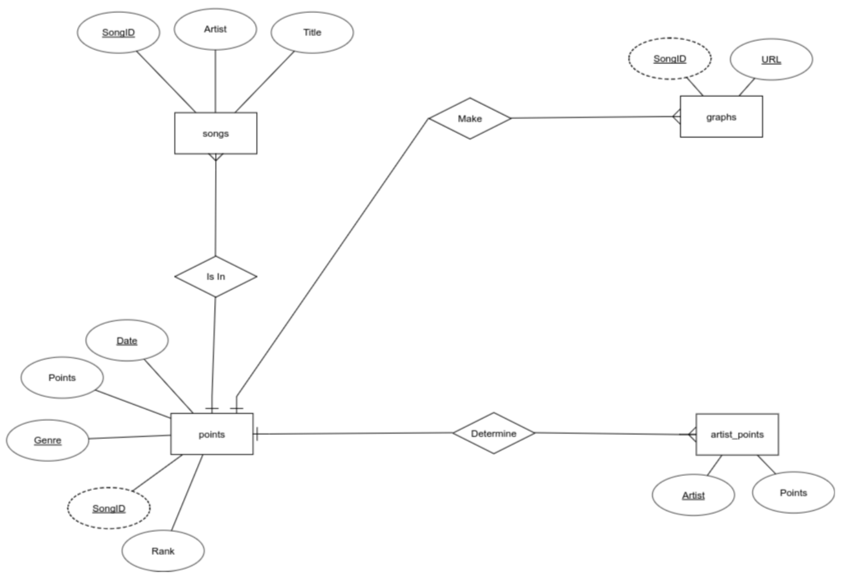

Figure 1: ER diagram of project database system.

the Rank and Points. The third table graphs holds two attributes. The first is a SongID which is a foreign key from the Song Table, and the URL for the graph based on the values accumulated for the song by the points by week in the points table. The URLs are redirected to plotly, which will be explained in our Implementation Details. The fourth table is an artist points table which takes the total points earned in the points table by the top Points an artist has earned in any genre and adding that to the Artist's total points stored in the Points attribute. In this table the Artist determines the Points generated.
Code Implementation:

```
class songs (db.Model):

    SongID = db.Column(db.Integer,primary key=True, autoincrement=False)
    Artist = db.Column(db.String(120), primary key=True)
    Title = db.Column(db.String(120), primary key = True)
    def repr (self):
    return ”%s, %s, %d” %(self.Artist, self.Title, self.SongID)

class points(db.Model):

    SongID = db.Column(db.Integer, db.ForeignKey(”songs.SongID”),
    nullable= False, primary key=True)
    Genre = db.Column(db.String(25), primary key=True)
```

```
        Points = db.Column(db.Integer)
        Date = db.Column(db.Date, primary key=True)
        Rank = db.Column(db.Integer)
        def repr self(self):
        return ”%d, %s, %d” %(self.SongID, self.Genre, self.Points)

class artist points(db.Model):

        Artist = db.Column(db.String(120), primary key=True)
        Points = db.Column(db.Integer)
        def repr self(self):
        return ”%s, %d” %(self.Artist, self.Points)

class graphs(db.Model):

        SongID = db.Column(db.String(120), db.ForeignKey(”songs.SongID”),
        nullable = False, primary key = True)
        URL = db.Column(db.String(120), primary key=True)
        def repr self(self):
```

Functionality Details

Basic Functionality

Add

The ”add” functionality is implemented in 2 ways. The primary way of adding information to the database is through the database initialization script which populates our song and points table from information provided by an unofficial Billboard api by guoguo12 via github [4]. The graph and artist points tables are derived from these tables later. Optionally for the sake of meeting the project requirements the ability to add an artist to the artist points table with a variable number of points has been exposed to the end user.

Delete

The ”delete” functionality is exposed to the end user in the form of delete an artist and all information dependent on that artists existence in the database is removed.

Update

The ”update” functionality exist as a script that updates the database from the most recent entry in the database to the most recent set of billboard charts released.

Song Search

The ”song search” functionality begins by searching for all songs that the end-users search is a sub-string is of, then for each of those songs it searches the points table for the most points earned by that song and on which chart.

Artsit Search

The ”artist search” functionality begins by searching for all songs by artist such that the end-users search is a sub-string, then for each of those songs it searches the points table for the most points earned by that song and on which chart.

Top 50 Artist

The ”Top Artist” functionality queries the artist points table and sorts the results in descending order and returns the top 50 artist in the artist table.

Top 50 Songs

The ”Top 50 Songs” functionality queries the points table and sorts the results in descending order and returns the top 50 songs of the data set.

Top 50 Songs by Genre

The ”Top Songs by Genre” functionality queries the points table and sorts the results in descending order and returns the top 50 songs of the data set in each genre.

Advanced Functionality

Song Graphing

The ”Song Graph” functionality is implemented by querying the points table for all entries where the SongId is equal to the SongID of the song requested. Next it gathers the x and y data for each genre such that the dates that the song was on the chart are the x values and the points earned per week are the y values.

YouTube Embedding

The ”YouTube embedding” functionality uses urllib to query a search string of Title and Artist to youtube’s search URL. It then opens the url with urllib2, and reads in the HTML. Then, the script uses Beautiful Soup to find the first video’s href which youtube stores in class ’yt-uix-title-link’. Finally, it takes that variable, cuts the watch substring off of it, and throws that into an embed url to link it back to the website to allow for video embedding.

Implementation Details

Billboard API

Our project is helped by the information made easily available by the Unofficial Billboard API. It helped us to avoid web scraping Billboard's website with Python's Beautiful Soup library. The API provided us with a relatively easy way to access the charts and songs, which we then added to our database to query later.

Python

All of our scripts were written in Python. Our initialization script for our database back ended off of the API provided to us, then iterated through the values using supporting scripts written by both of us. The scripts were used to calculate the artist points table which our website relies heavily upon, and all of the graphs which is the key element which separates our website from the regular billboard website.

Flask

Flask was the tool used to connect our database to our website . Flask runs off of an init file which links all of our HTML code to our python scripts . This makes our dynamic linking possible even with the absence of PHP. The init file also allows for the possibility of updating our database based on the current date of the week.

SQLAlchemy

Because we opted for a python implementation for our Database, we required a python database structure. Additionally, because we chose for HTML linking with Flask, and Flask's preferred SQL library is SQLALchemy, we chose for that structure. SQLAlchemy allows for SQL queries to be returned in a Python-like structure which make for an easier implementation and integration into our project overall.

Plotly

Plotly was used for the graphing interface of our website. It was used to generate over 18,000 graphs for over 6,000 separate artists in our database. As a result, we over-ran the API limits of this service on three separate occasions while trying to link our graphing table to the appropriate songs.

Beautiful Soup

Beautiful Soup was used for YouTube URL scraping. As the YouTube API proved to be cumbersome to use, we opted for which we neglected in choosing the Billboard API as mentioned above. Beautiful Soup allowed for our songs to be dynamically linked upon song-request from our website. This was not placed into our database due to the volatility of YouTube videos as well as the lack of need, as nothing has to be rendered unlike plotly. Therefore our website speed is not hindered.

Digital Ocean

Digital Ocean is our web hosting platform, first we set up an ubuntu droplet then installed all dependencies and configured it to serve our website. Finally we redirected traffic from our domain to the digital ocean droplet.

Experiences

We went into this project knowing virtually nothing about Python, Flask, SQLAlchemy, API's, Beautiful Soup, and pretty much everything that we used in our project.

Billboard API

The Billboard API was rather frustrating to use at times. Originally we wanted to have a Rap genre within our database, but for some odd reason date incrementation would not work within the Rap genre, so that was scrapped. Dates in general were rather difficult to work with. Because the API takes in dates as strings rather than as DateTime objects there was frequent conversions that had to take place between the API and our database itself. Moreover, the API came packaged with a .next function which was designed to fetch the next week's chart each time it was iterated over. Unfortunately this functionality proved to not be useful as breaking was consistent. We opted for timedelta objects for week incrementation in a substitute. Moreover, if the chart did not exist on the current date, the API would return a null list. However, the API does not fetch a chart.date from a null list, which lead to frequent breaking and patching within our dbinit file.

SQLAlchemy

SQLAlchemy's documentation, while extensive, did not quite translate well to our project. Because of the enviornment that they placed their documentation in, we were only able to gain a basic understanding of how the system worked internally. Often, we found ourself reverting to stack overflow posts whenever we found an issue that we needed to solve. The Flask Mega Tutorial by Miguel Grinberg proved to be vital in getting the setup of our database established, and allowed for us to get the ball rolling [5]. While it was nice dealing with Python based objects, we have had to recompile our database on multiple occasions due to errors within the base models, and SQLAlchemy's refusal to support migration out of the box. That all being said, SQLAlchemy, like all things, works extremely well when you gain a grasp of how to navigate it. It made iterating through lists a breeze, and helped make our jinja2 integration that much easier.

Flask

Flask's documentation was rather similiar to SQLAlchemy's documentation in the aspect of use. But while SQLAlchemy's documentation was rather extensive and difficult to read, Flask's documention was virtually non existant. The Flask Mega Tutorial was absolutely vital towards its use in our project. It provided us with a guideline and basis to initialize our project in the early stages. That being said, the tutorial insists upon the use of WTF Forms for the querying of the database. We did not find this method necessary and opted instead for the use of placing our queries and redirections into our init file. The talk of security with our database was brought up about this route of WTF Forms. After attempting SQL injection through our own means, and through SQLmap, we found our database to be secure without the need for WTF Forms as both of these methods proved to be fruitless. Flask was extremely helpful in naviagting around the usage of PHP, and allowed for our python scripts to run along-side our website. This allowed for data manipulation and the dynamical linking that persists through our entire website to be possible. With the use of flask integeration with our html code we had to learn to use jinja2. Jinja2 itself is not too difficult, but variable manipulation with jinja2 is a hit or miss thing with the subset of python commands included. Back-ending with even greater python script managment was something we had to get used to, and only furthered our knowledge of the language.

References

[1] www.billboard.com

[2] www.audioember.com

[3] www.github.com/labratjazzcat/audioember

[4] www.github.com/guoguo12/billboard-charts

[5] https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world