Tyler Seppala

**Final Project Report**

For this project, I implemented AudioStation: a web application for online collaboration on music projects between users. The idea to build a collaborative online platform for artists has been brewing in my head for quite a while, but the requirements for this project finally got me to take action on those ideas.

The biggest problem that I constantly face as a musician is the lack of connection to other musicians that have the availability to jam, the space to jam, or the equipment to jam. This is really unfortunate, because jamming and making new music is my favorite thing to do, and it's infinitely better with other people. So I started to try to think of ways of connecting musicians, without them having to be in the same space. One idea that I have, that’s been developing for much longer than AudioStation, is a web application on which audio files can be directly manipulated and edited in the browser. This is something that I will definitely be working on in the future, but is a bit complex and large for the scope of this class. Knowing I didn’t have the time to get something like that into a presentable condition before the deadline, I came up with the idea to strip away all of the editing functionality, and just deal with users and file-sharing.

This convenient, and rather simple solution to the problem is as follows: users can upload a single audio file to act as a demo for the project (the full compilation of all the separate audio files that have been contributed so far). Along with that, they also will upload a zip file containing all the separate files of different instruments and sounds (guitar parts, drums, vocals, etc.). Once this project is posted, it will appear in the feed that shows up on other users’ homepages, where they can scroll through, listen to demos, and decide if there is any project they want to work on. A user can download the associated zip file of a project by clicking on it, which navigates to the detail-view page, and from there they can click the big, blue button that says “Download Files”. This references a query to the database that joins the “posts” table with the “post\_files” table, and downloads the files associated with that post.

Once the files have been downloaded, the user has complete freedom as to how to work with them. They can use any DAW or audio editing software of their choosing, add as much as they want, and then export the demo and zip up all the associated files once again for re-uploading. They simply return to the detail-view of the post they are “collabing” with, and click the button that reads “Upload Collab”. This will give them a form basically identical to a “New Post” form, but with some of the fields auto-filled with data from the original post. This is a really cool feature, because once the collab is uploaded, all the previous collaborating artists are automatically tagged. The post itself is also tagged in the “View Collabs…” dropdown on the original post. This way, users don’t have to worry about trying to go through and find all the collaborators, as a lot of people would probably be left out in this case.

I based this approach off of a couple of existing applications, primarily Soundcloud and certain aspects of TikTok. Soundcloud is basically a social media platform for music, which is the layout of my program. And TikTok has a feature called “react”, in which a user can create a video that adds to another user’s video, and upon posting it automatically tags said user. I found a web application called “Kompoz”, which has roughly the same functionality as AudioShare, but in my opinion it has a very messy UI, and doesn’t solve the problem in a clean way. I think that my UI is much cleaner, more user-friendly, and intuitive to navigate through.

For the implementation of this solution, I decided to use the Django framework. I feel that this was a bit of an ambitious project, and Django really made it doable. The one problem with it though, is that I had never used it before, and therefore didn’t know anything about the built-in functionality. To really get an understanding of it, I followed a YouTube series that details how to build a very simple blog application using Django, through which I started to become comfortable with the layout of the framework, and also started to learn how to work with HTML (which I also hadn’t really done before). At a certain point in the tutorial, I found that what I had so far would actually be a great starting point for what I had in mind for AudioStation. I decided to branch away from there, and started implementing my own functionality. I referenced the series in my README, so if any of my code resembles something you find online (I’m pretty sure he uploaded his project on GitHub), that would be why. However, I assure you that I added my own unique functionality to virtually every square inch of the project.

As I am so new to web development, something that really helped me a lot was Bootstrap. This package includes a lot of built-in styling classes, which let me make the UI look very clean without having to do much CSS. Really the only CSS I needed was so small and simple that I could write it inline with the HTML I was applying it to. Probably the smallest feature that took the most amount of time was my use of Jquery. When looking for a way to implement aggregation, I decided that I would make a like button. I ended up using Ajax API control to do this, which was much harder than I expected. However, I did get it working.

This brings me to the project’s requirements. Upon opening the homepage of the app, you are immediately greeted with a list-view displaying records from the “post” table. This display, right off the bat, is performing a join across three tables, returning data from the ‘post’ table, the ‘post\_files’ table and the ‘user’ table. To display posts as they are in this view, the author username is needed from the user table, the post name and description is needed from the post table, and the demo file is needed from the post\_files table. A query of this same format is used a couple of other times in the application, when displaying posts by a single user, and when displaying search results.

Speaking of search results, this is where data is queried with various parameters. Users can search for songs by title, artist username, genre, or keywords in the description, all within the same search bar at the top of the homepage. I created indexes on all of those items except for the description, as it can be significantly longer than the other items, and shouldn’t be as relevant in a search. A join must be made between the user table and the post table to carry out this search, as the artist’s username is found in the user table while all the other information is found in the post table. All of these related tables’ referential integrity is enforced by primary key constraints, through which they are connected.

For the creation of a record, a user can either click “New Post”, or “Upload Collab” within another post. Here, the user is presented with a form that allows them to update both the post table and the post\_files table. I made this entire creation process an atomic transaction, which, amazingly enough, is done simply by placing the decorator “@transaction.atomic” above the given class within views.py. I also did this for “register()” and “profile()” in users\views.py. This makes it so that if any error is met while editing either of the two tables, the entire process roles back automatically. If everything is successful, it commits automatically. Once posted, the user that uploaded a post (and only that user), has the ability to delete the record from the database via a button at the top of the detail-view.

The most painful part of this project, likely because it is what I was struggling with the day before it was due, was the aggregation. I implemented a like button on each post, which in and of itself was a struggle. This like button had to utilize an API through Ajax so that, first, each user could only like a post once, and second, so that the page didn’t need to bounce off a different url and reload every time a user likes a post. When I finally got this working, it displayed the total like count, and changed between saying “Like” and “Unlike” depending on whether or not the user had already liked it. Now what I wanted to do was to use aggregation to sum up the total likes of any given user, and print it at the top of their page. The problem is, many-to-many relationships, such as what I used for likes, create their own table. The post\_likes table has columns for the user id that liked it, the like id, and the post id it is on, but not the user that made that post. Just use a subquery right? Well that’s what I did, but I didn’t realize how difficult it would be to figure out how to use subqueries in Django. Turns out that the “subquery()” function only works in very specific scenarios, and that all I had to do was use “\_\_in=” to signify the start of the subquery I needed. I ended up joining the user table to the post table to find posts from the current user, which was the subquery, and then joining the post table from that subquery to the likes table to find likes that matched. I realized that I had just made a query with a subquery that joined three tables, and to top it all off I added “.count()” to sum the results and check off the aggregation requirement as well.

The result is the total amount of likes received by a user, printed at the top of their page.

To update records, I included a profile link at the top right corner of the navigation bar which allows the user to update their profile information: email, username, password, etc. This function features a “@login\_required” decorator, which keeps non-logged-in users from accessing it. And for the final requirement, I made it so that only users with an admin profile, created from the console, can export csv reports from the user and post tables. This feature appears in the sidebar whenever an admin user is logged in. The other buttons in the sidebar aren’t currently functional, but in the future, I plan to implement a trending posts page, trending user page, and trending genre page that will be accessible to all users through that bar.

I believe that I have really made something valuable through this project, and have plans to deploy the full application on a web server some time in the near future. There still is some functionality that needs to be added, and certain things that need to be polished up before it is ready for real-world use, but I think it is really close as it is. I actually have my own music course video series that I’ve put together over the last few months, which takes someone from never having played music to being able to jam, make up their own songs and record. My plan is to put Audio Station membership in a package with that music course, and use it as extra incentive for people to purchase my program. This will bring challenges of its own, but the type of people I am marketing to are the perfect audience for this sort of thing, and I think it would bring a lot of value into their lives.

Hopefully, once this is all put together and on the market, I’ll start to form a large user base, and get people making some really interesting music. All the meanwhile, I’ll be working on my own music and my next project.