ILLINI HARMONICS

FINAL REPORT

- **Please list out changes in the directions of your project if the final project is different from your original proposal (based on your stage 1 proposal submission).**

1. We changed how the Spotify API was utilized in our application. We had originally intended to use it to fetch user listening history, but had instead used it to automatically populate our database with acquired and generated data.
2. The user interface underwent some changes, where we made more pages in the final product in order to divide data and information better. The original proposal had only a couple different pages, where the original discover page had most of our database information displayed. The new UI divided music, artists, and posts into separate tabs instead.
3. We also changed our direction in regards to the tag system, which was removed altogether as it wasn't the priority of the site.

- **Discuss what you think your application achieved or failed to achieve regarding its usefulness.**

  We managed to successfully create a website that allowed users to sign up, post from their local device, and rate any music of their choice. We had achieved the base social media platform that we initially desired, and managed to integrate enough features that we deemed useful. This included creating new users, posting from those users, making all key data visible, and generally making an easily navigable platform.

  In regards to shortcomings, we believe the only failure we had in usefulness was not hosting our site entirely on the GCP for public access. While we had used the GCP heavily for our database, we could have extended this usage to making the site accessible on multiple devices at once. This would require some changes in our user log-in functionality, but it would certainly be worth it.

- **Discuss if you changed the schema or source of the data for your application**

  We made some minor adjustments to our original schema in regards to most of the tables, we additionally added more tables to our schema - as will be discussed in the ER diagram changes. The changes made to our schema that exclude table and ER changes mainly pertain to our removal of the listening history relation in the final implementation.

Generally speaking, we had removed a lot of extraneous data from our schema that was not practical for our final implementation. This involved tags, artist descriptions, artist age, song genres, and song stream counts. The latter two parts of the schema were removed as they were not directly retrievable from the Spotify API. Instead, we had added genre as an artist attribute, and replaced stream counts with a metric called popularity - which the API did provide.

We did not change the source of our data, as we still utilized Spotify, but we did make the change of auto-generating data from Spotify's API, which wasn't originally planned. This involved generating random users, and making posts with them utilizing music data taken from Spotify.

- **Discuss what you changed to your ER diagram and/or your table implementations. What are some differences between the original design and the final design? Why? What do you think is a more suitable design?**

The ER diagram for our project underwent extensive changes over the stages of the project. Within stage 2, we initially had many attributes in tables such as Songs and Artists that did not end up being feasible. For example, song genre was not retrievable from the Spotify API so we removed it from our table in the stage 3 revisions. Similarly, artist age and descriptions were not directly retrievable without external web scraping, so we also decided that this info was unnecessary. We also removed some attributes from our tables between stages 2 and 3 as they were not needed and easily determinable through SQL keywords. An example of this would be the artist name that we initially had in our "Songs" table. This didn't follow a minimal ER diagram basis, so we had to remove this as it already was in the "Artists" table.

Some other key differences from our tables in the final implementation would be the addition of new tables to our database. We added tables titled "Login" and "ActiveUsers" in order to better keep track of who is logged in. This was added towards the end, as we realized that we needed some sort of system to keep our user tracked so their posts could contain their name and so forth. Thus, when a user logged in, we entered their userid into the tables to do so.

We believe that the new additions and cut-downs from the original implementation make our final design superior. We evidently excluded many extraneous and unneeded data points that were also hard to obtain. Similarly, we added new tables to our design when needed, which made login implementation much easier than using authorization tokens. Conclusively, the final design is more suitable for the final product.

- **Discuss what functionalities you added or removed. Why?**

  In terms of functionality, we added additional search features that we did not initially plan. We added the ability to search through posts by username, song name, and rating through multiple search bars. We figured that this would be more useful for users looking for specifics in posts. Whereas, initially, we had only had a simple single search feature. We also added more tabs to our website than initially planned. Rather than having a singular "Discover" page as we originally intended, we separated our data more clearly through various tabs. This was helpful in dividing our information better into categories such as "Music", "Artists", "Posts", and "Friends".

  In regards to what we removed, we had removed tagging posts from our original ideas. While the data that we auto-generated did include randomly generated tags, we figured that adding the tag functionality was secondary to our primary goals. Thus, we did not have the time remaining to add this feature. We also removed one use of the Spotify API from our project, which was fetching listening history and displaying it. This was intended to be used for rating the songs, but we replaced this with a manual post creating system. This ended up reducing our workload, and allowed a greater range of options in terms of songs that could be posted. Displaying Spotify API listening history proved to be more difficult than planned, as we needed to include Python scripts into our Java-focused React application - so it wasn't prioritized either.

- **Explain how you think your advanced database programs complement your application.**

  Our mySQL database complimented our application through our use of advanced stored procedures and triggers. Our frontend application heavily relied on our advanced procedures in order to fetch data to be displayed. An example of this involves our friend functionality, wherein users could add or remove friends seamlessly. This was complemented by three procedures: "ShowFriends", "AddFriend", and "RemoveFriend".

  The first procedure was utilized to fetch data for the currently logged in user, and display the data on the "Friends" frontend page. The other two procedures were used to directly alter our advanced database and either delete or add to certain user's friend relations. In terms of removing friends, we had to also check that the friend relation was removed from both ends of the users, which was done with a simple double check per call of the procedure.

  Other procedures existed to verify login and signup validity, whilst logging out was also performed through our database procedures. Furthermore, our database

programs assisted in regards to triggers. For instance, when a user created a post, we formed a trigger that would update artist popularity corresponding to what artist's song was being posted about.

If we were to consider Spotify as another advanced database program, then Spotify's API also complemented our application through providing plenty of data that we generated user posts through. Through fetching the top 1,000 songs off of Spotify through its API, we populated our initial database with plenty of songs and artists, alongside any key information they had as attributes (e.g. song length or artist genre). This aided us in making the application feel more complete with an extensive amount of posts to query and search through.

- **Each team member should describe one technical challenge that the team encountered. This should be sufficiently detailed such that another future team could use this as helpful advice if they were to start a similar project or where to maintain your project.**

  Michael - We struggled with connecting our GDB to our front-end initially. While the actual code to set up the connection seemed fine, our data would not display. I eventually figured out that we needed to be on the Illinois wifi network, as apartment wifi did not work for our connection. Using a VPN fixed this issue after a long time of searching for the answer.

  Jacob - Towards the final stages of our project, we faced the final connections of our SQL backend to our frontend. For many cases like signing users up, logging in, or creating new posts, we had to input arguments into our SQL queries and stored procedures directly from our code. These arguments were being inputted from our frontend, so I had to figure out how to pull data from text inputs and place them properly. The solution ended up being using our local server to input arguments through the server-side endpoint and URL.

  Paul - One technical challenge of this project was learning an entirely new programming language for the frontend. I had never done CSS nor TypeScript, so the entire frontend, which Michael and I had split up, definitely had a learning curve. Michael had set up our frontend initially, so I used his code as a template to look and understand what was going on. Otherwise, doing a lot of research on how to implement certain features helped me pick up fast and resolve this technical gap of knowledge.

  Joe - We had problems with sharing our code at first and running it locally. This had to do with some of the packages and plugins we installed on the first device. After some

debugging and reinstalling the tools such as Vite for React, we managed to get it running for everyone. Since I have a MacBook it took some additional steps to set up Vite and React, but it eventually was fixed. Github then easily allowed us to share code and work together.

- **Are there other things that changed comparing the final application with the original proposal?**

  There were a couple minor changes that were not maintained from the original proposal, but apart from what was already discussed, they are very minor. One of these things pertains to data integrity, where we could have utilized a few more triggers upon creating new data. The original proposal outlines that we planned to use a trigger to prevent a single user from creating multiple posts for the same song. This was something we excluded as it wouldn't be apparent unless explicitly tested.

- **Describe future work that you think, other than the interface, that the application can improve on**

  If we were to continue working on this project, we would certainly begin by implementing the features that we omitted through this stage. This would include tagging and using listening history. We could then extend our API usage to also use Apple Music in addition to Spotify for listening history once implemented. On top of this, we could then implement new features such as concurrent logins, alongside hosting our website publicly so multiple people can utilize it at once. Small features that the application could also improve on would be including album covers, which would help diversify our "Posts" page visually and help contextualize some songs people may be unfamiliar with. Likewise, including information such as albums would be a great way to extend our website - we could even make a separate "Albums" page.

- **Describe the final division of labor and how well you managed teamwork.**

  Our final division of labor was much simpler than in our original proposal. Rather than having four roles, we split up front-end and back-end evenly amongst the group. Michael and Paul had handled the front-end development of the site, which involved most TypeScript, JavaScript, and CSS development. This involved creating all of the components for displaying our data. Meanwhile, Jacob and Joe had focused on the back-end, which involved creating the SQL queries we needed for our front-end. This also involved setting up the GCP database, with help from Michael as well. The back-end team then took the front-end team's framework that was created, and inserted all queries that were made in order to display things like user posts, songs, artists, etc.

In terms of managing teamwork, we did well with keeping on track, and finishing our product was not difficult due to the division of labor and constant work we put in. Rather than scheduling times that we wanted things done by, we had all put in heaps of work over the weeks that we had, which prevented cramming or last-second work. We had also done well with communicating, and any changes that were pushed or made were clearly expressed and merged with other people's work.