Collaborative Playlist Generation

India Heath

N00213565

Supervisor: Sue Reardon

Second Reader: John Montayne

Year 4 2024/25

DL836 BSc (Hons) in Creative Computing

Abstract

Satisfying everyone's preferences when selecting music for a party or road trip can often be frustrating. While existing solutions such as Spotify's Blend and Jam offer a partial fix, they typically exclude non-premium users. This project was developed to bridge the gap by developing a mobile app that allows both free and Premium Spotify users to collaboratively generate a playlist from all their listening histories, with the option for further refinement through real-time track voting. User research identified a strong interest in blended playlist creation and revealed a lack of awareness of existing tools. The application was developed using Flutter for the Android frontend, Express.js for the backend, Firebase for authentication and a NoSQL database and the Spotify Web API for user data and playlist management. Development followed Scrum methodology, with focused sprints addressing tasks like Spotify token management, integrating live voting and user interface design. Automated unit testing using Jest confirmed the backend reliability, while manual integration testing validated core features. Future development may include integrating Spotify's queue for alternative playback and implementing a more sophisticated algorithm for playlist generation.

Acknowledgements

I would like to thank my supervisor, Sue Readon, for the steady guidance and support during this project. Our weekly meetings were invaluable for refining the project's direction and staying focused during challenging phases. I would also like to thank my second reader, John Montayne, for the thoughtful and constructive feedback, and for the encouragement to deepen the complexity of this project.

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by other. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their *own work.*

DECLARATION:

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Student : India Heath

Signed

India Heath

Failure to complete and submit this form may lead to an investigation into your work.

Table of Contents

1 Introduction

Making everyone happy with music choices is a situation most people have been in before. Whether on a long car ride or at a party, someone is responsible for the music and ultimately, not everyone's preferences can be considered.

Not everyone wants to sit down and curate a playlist with their friends; that's very time intensive. Existing options can be limited to paying Spotify users only, excluding many people. There's no simple way of quickly and easily creating a playlist that represents a portion of everyone's tastes that doesn't require all to be Premium Spotify members.

To address this problem, an app was proposed that blends the existing tools into one that's quick to set up and start using. It aims to allow a group of people to blend their listening preferences and be given a playlist in seconds that represents everyone's tastes. It will allow users to further refine the tracks by voting on them in real time and removing tracks that multiple people dislike. Making the process as seamless as possible will be done by using a simple QR code to join the playlist session. The finalised playlist will then be available to save to a Spotify account where the listening can start.

The tech stack includes Flutter for an Android frontend, Express.js for a backend, Firebase Cloud services for authentication and database and Spotify Web API for user history and playlist saving.

This thesis is divided into sections that walk through the research, design, implementation, testing, project management and final reflection of the project. The research section will examine the main themes of the project through a literature review of relevant academic research. Following the research are the requirements for the app's functional and non-functional needs. These requirements are gathered through researching similar apps, conducting a user survey and creating personas. The next section will describe the project design in terms of system architecture, database design, and user interface. The implementation section details the development process of the project using the Scrum methodology and sprints. Each sprint starts with defining the goals and then detailing the progress made. The testing section presents both functional and user testing to confirm the application has met the functional requirements, both technically and in terms of user experience.

The project management section discusses how the project was organised into phases, the tools used and an overall reflection on the project. Finally, the conclusion will summarise each section and detail the steps that could be taken for future development.

2 Research

On-demand streaming services such as Spotify, Apple Music and YouTube Music represent how the majority of people today consume music. The Global Music Report 2025 (IFPI 2025) found that streaming alone was responsible for 69% of global music revenue, a 7.3% increase from the previous year. A key advancement in this area has been the algorithmic generation of playlists and music recommendations through recommender systems. Spotify is most notable for this with their Daily Mixes and Release Radar playlists, which make recommendations tailored to users' listening references.

Using large language models (LLMs) as a tool for developing more advanced and nuanced recommender systems which specialise in lyric analysis is an area of interest for this project. Combining existing sentiment analysis approaches with machine learning opens the possibility for developing recommender systems that can recommend based on lyrical themes. This topic was part of an early research phase, ultimately falling beyond the scope of the project, but should be considered for future development.

2.1 Recommender Systems

Various methods are relied upon by recommender systems (RMs) in order to provide users with personalised recommendations. The two main approaches are collaborative filtering (CF) and content-based filtering (CBF), each with its own pros and cons. Both methods face challenges like the cold start problem and data sparsity, which limits their effectiveness in some scenarios. Hybrid models are a recent advancement, along with deep learning techniques, which have attempted to solve part of these limitations and improve the quality of recommendations.

2.1.1 Types of systems

CBF makes recommendations based on similarities between users and items. This is a very commonly used system, especially in e-commerce, where a user is often shown a list of similar items when viewing an item detail page. This type of system will categorise items into groups and show users other items from that group. For situations like online shopping, this can work well. An issue arises when users are recommended items that are too similar or too low in diversity. Users don’t always want to see movies in just one genre; they enjoy variety. To achieve more nuanced and varied recommendations CF is very useful.

CF uses data from other users to make recommendations. User A and B usually watch the same films, so when user A watches something new it’s likely that user B will also enjoy it. CF is generally

regarded as more accurate but comes with its own downsides, as discussed by Thorat et al. (2015). They also discuss the disadvantages of CF, saying that although the accuracy is high, CF faces the significant challenge of dealing with data sparsity, also known as the cold start problem. The cold start problem describes the situation of a new system having no existing user data and not being able to provide good or accurate recommendations because of that. CF requires a very large initial data set to build its system.

The next challenge faced by RMs is scalability. When platforms reach millions of users and items, it can become extremely difficult to provide fast recommendations. A solution mentioned by Thorat is one currently used by Twitter, where they separate their users into smaller clusters, which are easier to manage and provide recommendations for.

The last issue faced by a lot of RMs, briefly touched on already, is that of diversity. Depending on the system's underlying algorithm, it may not provide a diverse enough range of recommendations. Some systems are particularly bad at recommending new or slightly obscure content, defaulting to what is highly popular. Many platforms have this issue and their own solutions to it. Spotify can feel very repetitive when listening to mixes auto-generated based on a user's main listening genres.

Both systems have advantages, CBF being an appropriate option in a cold start case and CF being well suited when a large user dataset is available.

2.1.2 Advances in Recommender Systems

A solution for the challenges mentioned above has come from recent advances in RM research in the way of integrating CF and CBF into hybrid models. This combines the strengths of both approaches while mitigating the weaknesses and providing enhanced recommendations.

In the 2023 paper, Wang, Y. discusses the use of deep learning in hybrid RMs and its ability to better capture complex patterns and relationships in suer-item data. This enhanced performance improves both the accuracy and relevance of recommendations.

Context-aware RMs are a particularly promising development (Wang, Y. (2023)). Incorporating factors such as location and time will allow the RMs to further refine recommendations to match preferences in more specific contexts. An example of this has already existed in Spotify for some time. Users will be shown playlists that are named after times of day, ‘evening chill time’ for example. These playlists are auto generated based on the users’ listening history. These insights allow users to receive more personalised and effective recommendations.

Wang also addresses the cold start problem and its particular significance to CF. He outlines the advanced techniques being explored as solutions, like active learning, transfer learning and meta-learning. Active learning is something users are already familiar with. It's the direct soliciting of user feedback on specific items, often seen during account setup, to quickly provide personalised recommendations. Users will likely be given a small selection of items, be it movies, songs or products, which they can either rate or mark as of interest. This helps alleviate the initial data sparsity problem RMs must deal with.

2.1.3 Fairness, Bias, and Ethics in Recommender Systems

Both Wang and Thorat acknowledge the importance of fairness, diversity and ethics in RMs. Bias in RM algorithms can and have led to certain user groups being under-represented and users getting stuck in filter bubbles and echo chambers, leading to limited exposure. Adversarial training and fairness-aware matrix factorization are techniques Wang highlights as areas of development meant to address the bias problem while providing accurate and high-quality recommendations.

Explainability in AI is again a viral area of focus, as already looked at through Krugmann & Hartmann in the previous section on large language models. This point is further emphasized by Wang when they talk about the increased user engagement with RMs when they are trusted and understood. A recommendation for incorporating mechanisms that allow users to opt out of one system or another, diverse recommendations or highly personalised, was given as a way to improve user satisfaction and trust.

2.2 Importance of lyrics in music

Understanding the relationship between lyrics and the emotions they evoke can be extremely helpful in learning what listeners resonate with. This understanding can be used to create thematically cohesive playlists by combining multiple users' preferences.

2.2.1 Background on Music and Mood

There are two positions on how music is experienced in people: cognitivist and emotivist. The cognitivist position says that music does not induce or elicit emotions in listeners, but rather, they recognize the emotions cognitively. The emotivist position says that listeners experience the emotions and not just recognize them. The studies looked at in this chapter explore these two positions.

2.2.2 Psychological/Physiological Impacts of Music

A study by Krumhansl, C. L (1997) was conducted in order to find out if music elicits emotional responses in listeners or if it expresses emotions that the listeners recognize in the music. Are the listeners having these emotions themselves, induced by the music, or are they outwardly expressing what the song is presenting? The findings largely supported the emotivist view that music truly elicits emotions from its listeners.

Their findings were that the music chosen to elicit one of the four emotions chosen, Sad, Fear, Happy and Tension, had the expected effect. They found that some emotions correspond to specific physiological reactions. For example, sadness was associated with a change in the cardiac system, whereas happiness showed a change in respiration. This paper proved that music has a physiological effect on people and, furthermore, different effects depending on the mood of the music.

A related study by Gomez and Danuser (2007) explored how 11 structural features of music influenced the physiological and emotional responses of the listeners. Some of the features focused on included temp, mode, harmonic complexity, and rhythmic articulation. They found that musical pieces in a major mode, which featured a simple harmonic structure, were associated with positivity and feelings of happiness. On the other hand, pieces in a minor mode with complex harmonies were associated with more negatively leading feelings and melancholy.

Similar to the findings from Krumhansl's study, Gome and Danuser discovered a link between specific musical features and physiological reactions. Faster temp pieces with strong rhythmic patterns lead to consistently higher heart rates, elevated skin conductance levels, and faster respiration rates.

The study highlights the uses of these findings in practical fields like music therapy and workplace productivity. Having an understanding of the link between specific musical elements and emotional states can be helpful in designing spaces and therapy plans. An example of music therapy could be musical pieces that have a slow tempo and simple harmonies in order to reduce stress and anxiety in a therapy setting.

2.2.3 Lyrics versus Musical Tone

This section will investigate how lyrics can affect a person's perception of a melody and song overall. It should inform whether lyrics are as or more important than the melody when determining the mood/sentiment of a song.

Research conducted by Pond and Leavens (2024) looked at the effects of sad melodies versus sad lyrics. They played the participants three versions of the sad pop-ballad ‘Mad World’, the isolated melody, the isolated lyrics and then the original song. Their findings suggested that the participants were least affected by the melody alone but that the isolated lyrics and the original song were at equal levels of effectiveness in causing a negative impact on the listeners' moods. When asked, the participants mostly attributed the effect on mood to the semantic content of the lyrics, saying that the vocal performance alone and melody had a lesser effect. Some even reported that the isolated melody was relaxing but found the song in its full original state was sadder. Other participants reported the slow tempo of the melody was more gloomy than relaxing.

As already mentioned by Gomez and Danuser (2007), music therapy is a growing field that benefits heavily from such research. Pond and Leavens point to the positive impact music therapy can have on depression patients. They do, however, advise that caution should be taken if patients are to be allowed to choose their own music for the purposes of music therapy. Saying that people are likely to choose familiar music, even if the lyrical sentiment is not positive. They warn that this could lead to the patient experiencing a decreased mood and suggest further study be done in the area of melody and lyrics.

Most of the existing studies done on the relationship between emotions and music have looked at instrumental tracks rather than those with lyrics. Since the majority of people today listen to music with lyrics, the influence of lyrics on emotion shouldn’t be neglected. This study by Mori and Iwanaga (2014) sets out to examine this influence by playing participants happy-sounding foreign songs where they did not understand the lyrics.

The results showed that the happy-sounding music evoked happy emotions, and the translations of the sad lyrics evoked sad emotions. The interesting point came when the lyric translations were combined with happy-sounding music, and they evoked pleasant emotions. This would suggest that the sound of a song has a greater impact than the lyrics alone.

These results present an interesting contrast to the Pond and Leavens study, in which they looked at sad lyrics and sad melodies. These two studies are interesting to compare as they used a mix of happy and sad elements. It seems that people are affected differently by sad vs. happy melodies and lyric sentiment. The lyrics in isolation have the expected effect, but when mixed with happy/sad melodies, the results can vary.

This begs the question, which element of a song has a greater impact and why? Further research into the mix of melody and semantic meaning in modern songs could be incredibly useful in the area of music therapy as well as other fields, such as marketing and creative media.

2.3 Large Language Models

This section will discuss the research surrounding the evolution and current state of LLMs, along with their relevance to lyric analysis. The limitations and potential challenges of using an LLM for sentiment analysis on lyrics will be explored through current research on the topic.

2.3.1 Introduction to LLMs

LLMs have been a big step forward in the field of Natural Language Processing (NLP), making it possible for computers to seemingly understand complex language and produce human-like responses to a wide range of tasks.

2.3.2 Evolution of LLMS

NLP has come a long way in recent years, starting with statistical language models (SLMs) and moving to the now commonplace neural networks that make up the backbone of today's AI landscape. SLMs predict the next word in a sequence based on the input data. They aren't the strongest for NLP due to their limitations in computational efficiency when estimating the huge number of probabilities needed (Naveed, H. et al. (2023)).

The architecture of a neural network in AI was inspired by biological neural networks. This offers many advantages, such as "classification, prediction, filtering, optimization, pattern recognition, and function approximation" (Thakur, A., & Konde, A. (2021)). Using this architecture allows AI models to process information at high speeds in parallel and make sense of what was initially unclear. It also learns to generalize information, which paved the way for further advancements.

A breakthrough for NLP is attributed to the introduction of transformers in a 2017 paper titled 'Attention is all you need' from Vaswani, A. Transformers gave LLMs the ability to perform at a human-like level in many complex language activities. LLMs like GPT3 (Generative pre-trained transformer 3) from OpenAI and BERT (Bidirectional Encoder Representations from Transformers) from Google are a direct result of the power of transformers.

2.3.3 Large Language Models for Sentiment Analysis

An LLM will be used to perform lyric analysis, so it is important to understand how LLMs work and to know their limitations and potential challenges. An important issue with LLMs is that of bias and potential hallucinations which the model may produce. Understanding the AI behind the models will help with identifying their limitations and assessing how reliable their results are.

LLMs have limitations in handling subtleties in language, such as humour and sarcasm, which was found in the 2023 LLM sentiment analysis study by Zhang, W., Deng, Y., Liu, B., Pan, S. J., & Bing, L. This lack of cultural knowledge and context leads to the models having the potential to miss important subtleties in modern texts. This is especially evident when looking at data from social media platforms, which often contain colloquialisms and slag that models aren't guaranteed to have been trained on. The study also found that larger models, like GPT-3.5, are not necessarily better than their smaller counterparts, like Flan-UL2. The reason being that the smaller model in the study had been given instruction-tuning, increasing its precision and enabling it to leverage the dataset more effectively. Comparable performance was found between the models on shorter-form content despite the model size difference. For lyric sentiment analysis, smaller and more specialized models that have been fine-tuned for poetic language might be a good or even superior option to larger, more general-purpose LLMs.

Sentiment analysis is also being explored using LLMs in marketing and business. The ability to analyse and generate appropriate responses for content-heavy texts is especially useful in these fields. Krugmann & Hartmann (2024) focused their research on comparing the performances of GPT-3.5, GPT-4, and Llama 2. The importance of context was a major aspect of their study, noting that models like GPT-4 and Llama 2 performed very well when given sufficient input but were outperformed by traditional methods when it came to simpler and shorter texts. This comes back to LLM's limited ability to deal with linguistic nuances and cultural context.

An important aspect of LLMs and AI models in general is that of transparency and explainability. AI is often perceived as a "black box" into which a user gives input and then receives a response. The inner workings are often a mystery, and in the case of GPT-4 and other models from OpenAI, proprietary information. This leads to concerns around trust and accessibility in AI. Krugmann & Hartmann emphasise the need for transparency for the advancement of AI, pointing out the Llama 2 model from their study. This model is open source, meaning it's

accessible to researchers interested in how it was built or wanting to attempt improvements on it.

The limitations of LLMs in the area of rapidly evolving language are both agreed upon in these papers, and each offers similar suggestions for future research.

3 Requirements

3.1 Introduction

This section investigates the requirements of the application by analysing similar applications and surveying potential users. The application requirements determine what functionalities will be developed and what priority will be placed on them.

Analysing similar applications is helpful in gathering important features and user flows that are common across multiple applications. It also provides a guideline for how users might expect an application to work.

Understanding the requirements from the user's perspective is very important. The developer's decision alone on the requirements could lead to the application going in a direction that is not useful to the end users.

Users were surveyed to learn more about how they interact with the music streaming platform Spotify regarding playlists. All this information was compiled to create personas, and functional and non-functional requirements were outlined based on those personas.

3.2 Requirements Gathering

3.2.1 Similar applications

Two applications were found that have functionalities similar to this project. Both are explored in this section, highlighting relevant features and functionalities.

3.2.1.1 Spotify

Spotify is a popular music streaming platform which offers multiple built-in options for creating playlists. The options being discussed here are collaborative and Blend playlists.

3.2.1.1.1 Collaborative playlists

The collaborative playlist feature allows a user to invite friends to contribute to their playlist. Collaborative users can add, remove and reorder tracks. Each track indicates which user added it by a coloured circle with the user's first initial.

Users can be added to the playlist easily by clicking an icon outlined in yellow in Figure 3-1. This figure also shows the first initial of a user next to the tracks they added, making the collaboration aspect very clear.

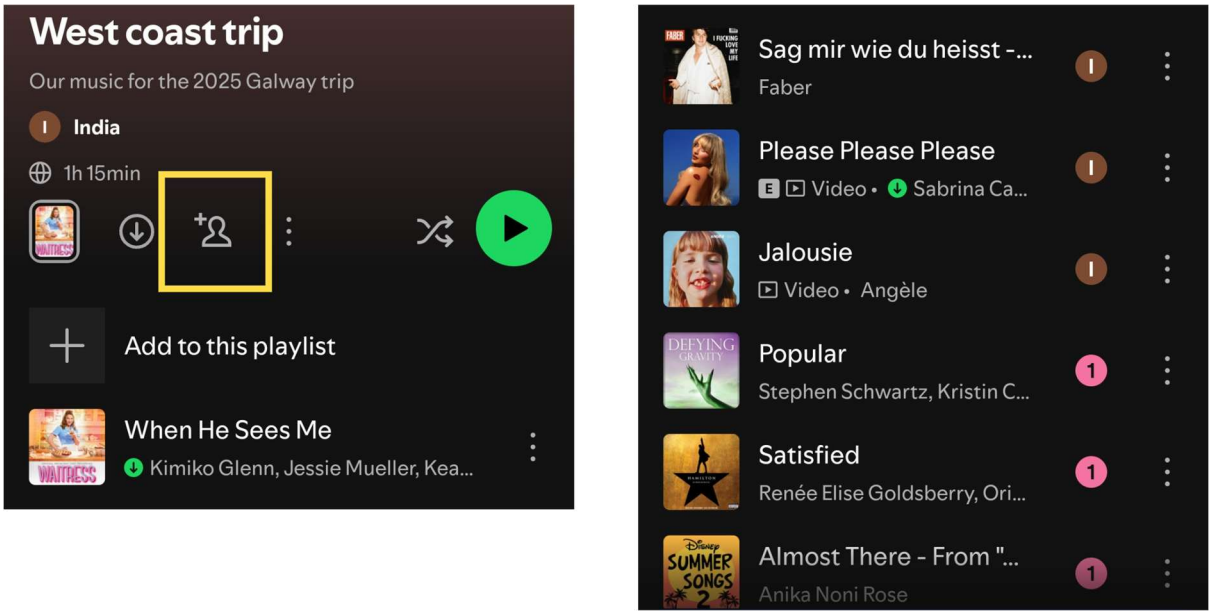

Figure 3-1 - Spotify collaborative playlists.

3.2.1.1.2 Blend playlists

A newer Spotify feature is the Blend playlist (Figure 3-2). This feature allows up to 10 people to join and then generates a playlist that matches all the users' tastes.

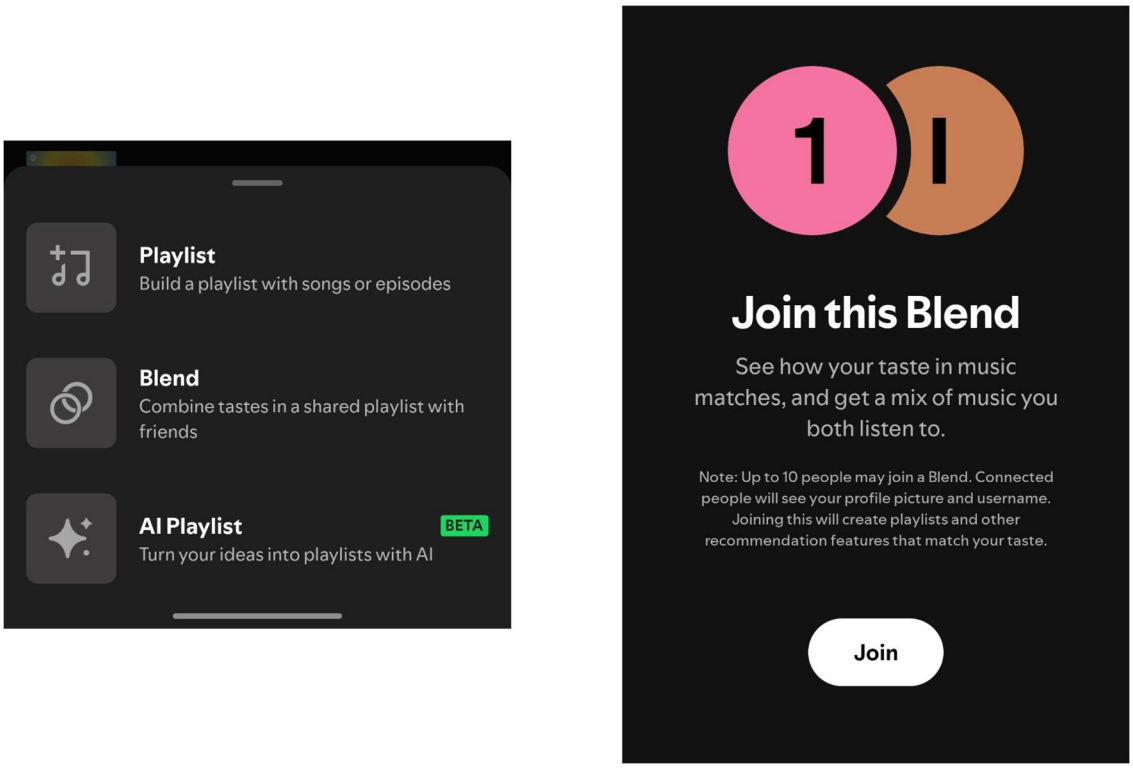

Figure 3-2 - Spotify Blend playlist creation.

It is updated daily, staying current with each user's listening preferences. It has the same user icon indicators for each track (Figure 3-3) as with the collaborative playlists.

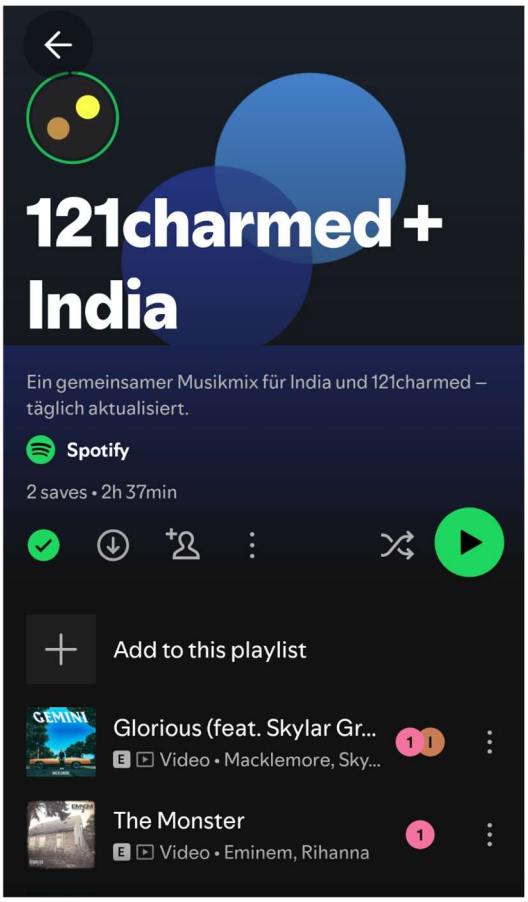

Figure 3-3 - Spotify Blend playlist results.

3.2.1.2 Partyfy.io

Partyfy is a web application that allows multiple users to live vote on a playlist's tracks. The app uses the Spotify Web API for track and artist information and playback. The host creates a playlist and generates the share link/QR code to send to participants, seen in the first screenshot of Figure 3-4.

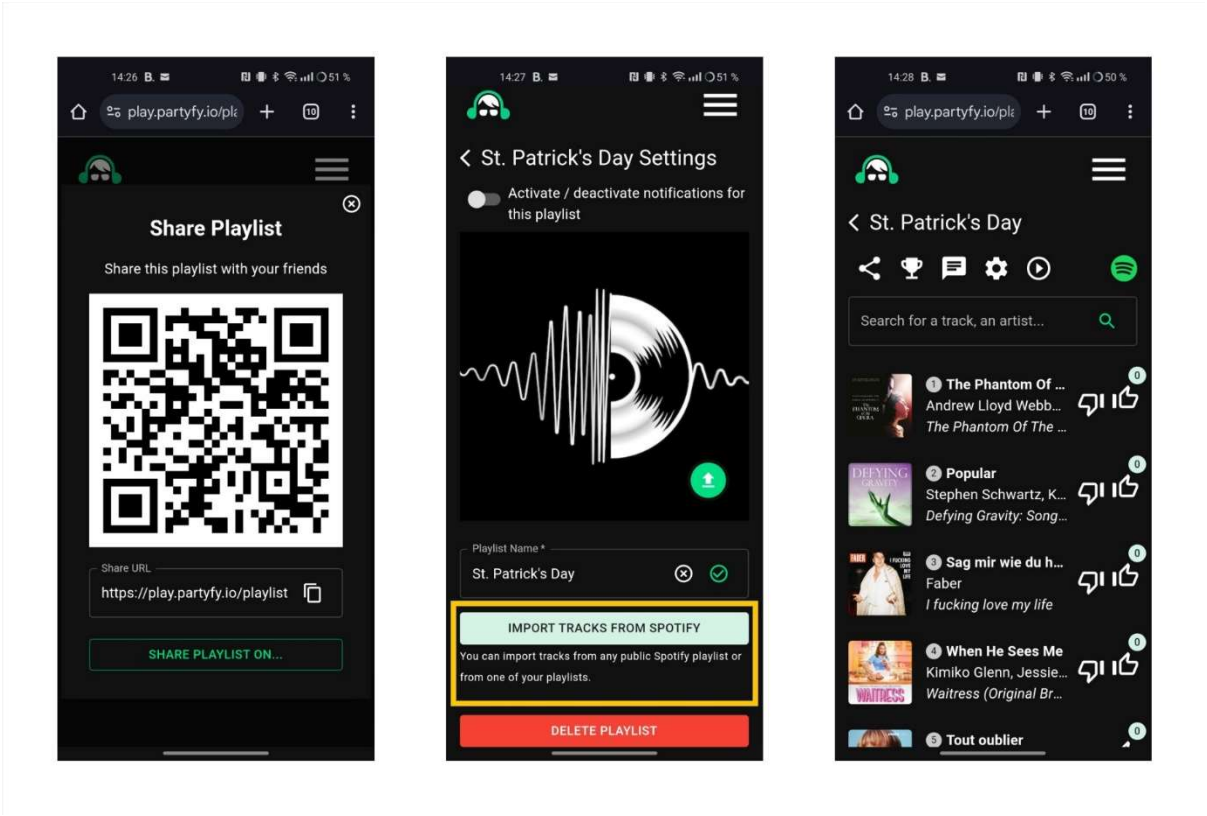

Figure 3-4 – Partyfy QR code, import playlist and track list UI.

The host can import a playlist from Spotify through the settings or just start adding tracks, as can other users, see screenshot two and three in Figure 3-4. Each track can be voted up/down with a green or red UI count indicator (Figure 3-5).

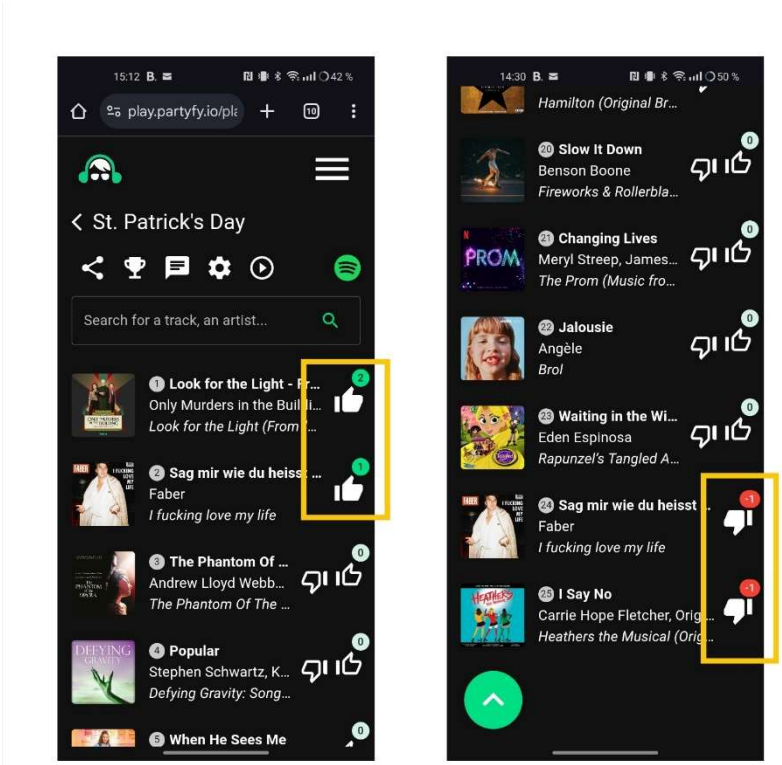

Figure 3-5 - Partyfy track voting UI.

The app has two playback options. It play tracks either by adding them to the hosts Spotify queue, or by acting its own playback device. If using the Spotify queue, all playback controls are handled in the Spotify app (First screenshot in Figure 3-6). Tracks added to the queue are identified by a 'queue' icon, indicated in Figure 3-6 by a yellow arrow. The playback options are not immediately obvious, and the difference are not well explained; this led to confusion.

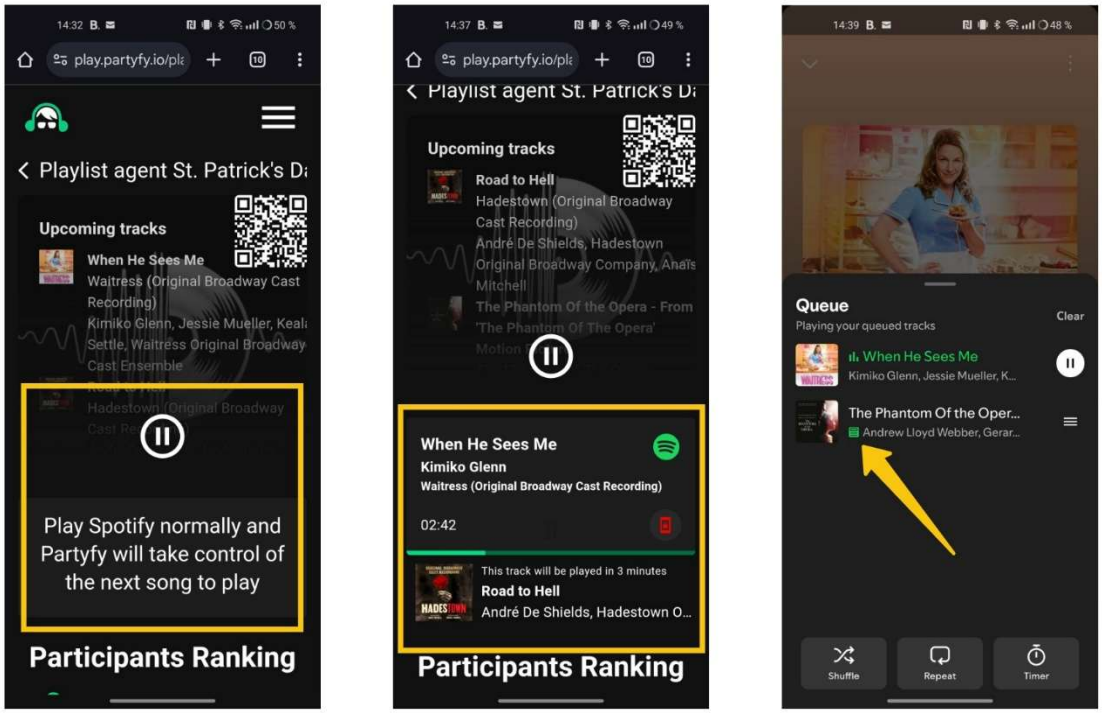

Figure 3-6 - Partyfy playback through Spotify.

The app has a few bugs relating to its playback options. These are a result of Spotify's restrictions for non-premium members who can't add tracks to their queue.

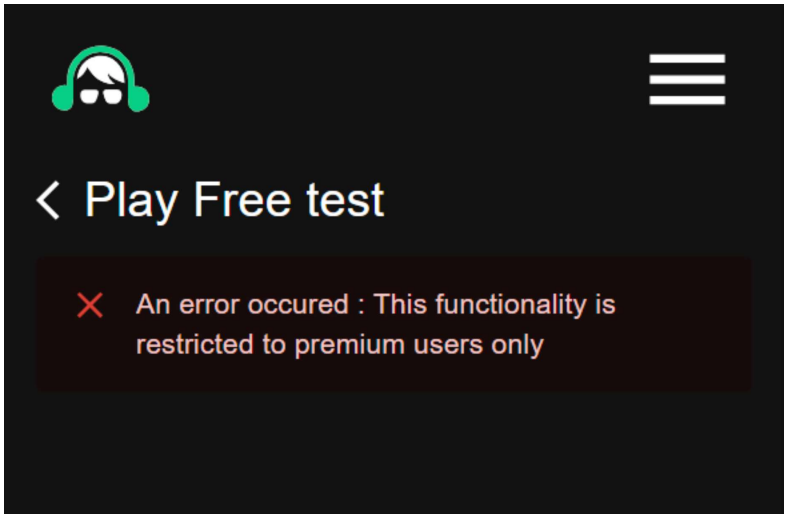

Figure 3-7 - Partyfy premium users only error.

These restrictions aren't properly handled by the app, resulting in some users being shown error messages rather than the app restricting non-premium users accessing premium only features (Figure 3-7).

3.2.2 Survey

A survey was conducted to explore how users interact with Spotify playlists and what features they would like to have in a new app (Appendix A – Spotify Playlist Preferences Survey). A survey was chosen to gather quantitative data from many participants within a reasonable timeframe. It was distributed among 20–30-year-olds, with eighteen responses.

3.2.2.1 Playlist Usage Trends

The most frequently used type of playlists by participants were the Personalised Mixes from Spotify and Listener Playlist (Figure 3-8). Listener Playlist are made by users and not by Spotify. Spotify Mixes are auto generated playlists that use an algorithm to show songs the user is likely to enjoy.

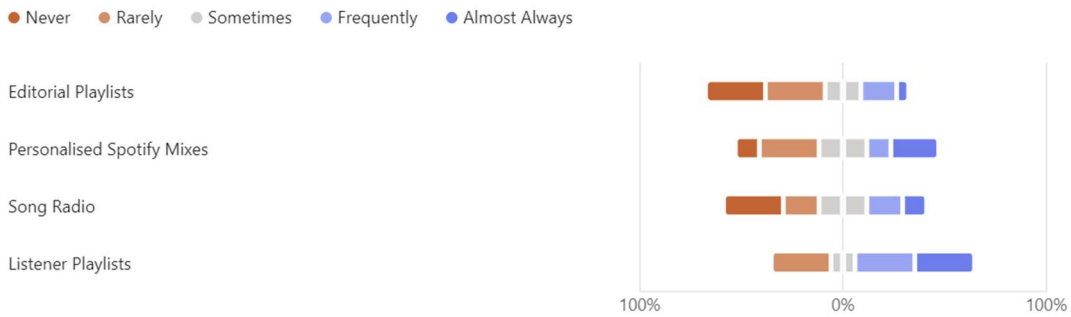

Figure 3-8- Survey question "How frequently do you listen to these types of playlists?"

When asked how often they create playlists, there was a mixed response. Seven said ‘very often’ and six said ‘rarely’, with four users in-between. Over 90% of respondents said they use their mobiles to create playlists.

3.2.2.2 Interest in features

Participants were asked about their interest in certain app features and in what situations they would use them.

Two key features were explained to the participants, and they were then asked to rate how likely they were to use the features, and in what context.

Feature 1: Blend of users’ preferences

On the feature that blends multiple users’ music preferences, over 90% of respondents were somewhat/extremely interested in this feature.

Figure 3-9 shows 60% of respondents said they be very likely, 20% somewhat likely to use if at parties or social gatherings and on road trip or long car rides. Remote listening was the least popular response, with 50% saying they were somewhat/very unlikely to use the feature then.

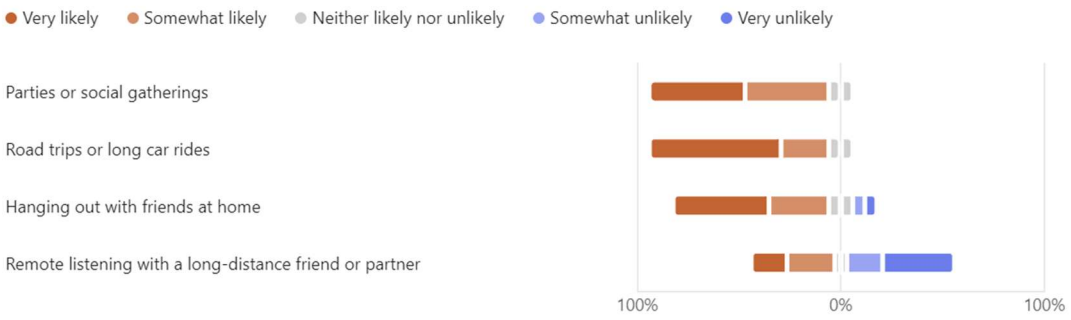

Figure 3-9 - Survey question "How likely would you be to use this feature in the following situations?"

Feature 2: Live voting on tracks in a playlist

The responses for interest on this feature were more varied but still had a positive leaning results (Figure 3-10), although not as strong as with the first feature. The situation question had very similar results as well. Showing that parties were a more likely option, followed by road trips but remote listening was the most unlikely.

Figure 3-10 - Survey question "How interested would you be in an app with the following feature?"

3.2.2.3 Awareness of existing features

The survey found that many users were unaware of existing features in Spotify that accomplish part of the two proposed features, blended user preferences and live voting. When asked about Spotify's Blend playlist feature, 40% of respondents hadn't heard of it before, with only three respondents using it occasionally/frequently.

Almost 30% said they used the feature to find common songs among friends for a trip, party or event. Another 30% saying they used it to see how their music tastes compared to a friends'. Participants could select more than one option as a response to this question so the 30% figures here are likely from the same participants.

When asked about the Spotify Jam feature, 83% were aware of it and 66% had already used it.

3.2.2.4 Survey insights

Key insights:

- Mobile is preferred
- In-person scenario preferred over remote listening
- A focus on use in car rides and social gatherings
- User preference for Listener created playlists over Spotify Mixes
- Shared listening playlists feature is slightly more important than live voting
- Most users aren't aware of the Blend playlists

Over 90% of respondents said they use their mobiles to create playlists. This supports the decision to focus on Android instead of web for this project.

From the results, focusing on in-person app usage is more appropriate for the project. Adding a QR code or simple text code support for joining would be better than sending links. Links would require a separate messaging app that each user would need access to. This mightn't be an issue among friends but at parties with unfamiliar people, this could be an obstacle. The strong preference for using on road trips should be taken into consideration as well, as it outweighs the party situation.

Although many participants use Listener playlists, half of them listen to other people's playlists and not their own. This shows a preference for user curated playlists from other users.

The results show that live voting is less of a priority in general. This feature could be an optional one, being toggled on or off by users. It could also be left out of this project in favour for another, more relevant feature if the scope does not allow for both.

Even though participants said they would like a feature like the Blend playlists, they were unaware it already existed. This suggests it hasn't been marketed well or that it doesn't provide a good enough user experience (UX) to keep users engaged. Creating an app with a very clear purpose and simple user interface could fill this gap in the market. Focusing on the top use cases of car rides and parties could increase awareness of the features.

3.3 Requirements Modelling

3.3.1 Personas

The goal in creating personas is to create a profile or a person who represents a group of similar people who make up a target audience. Personas can be helpful when defining the requirements for a product because they provide a very specific profile rather than a broad target audience. Finding the requirements based on a persona result in more targeted solutions to a problem and can

uncover new areas for consideration. Two personas, Emma (Figure 3-11) and Patrick (Figure 3-12), were created to represent the varied target audience for the application.

Emma:

A 20-year-old college student who already uses Spotify very regularly. She like indie pop. She listens to music with her friends before they go out partying on the weekend, so the music choice is important. She gets frustrated because one person usually ends up dominating the music. They usually choose a popular generated playlist from Spotify that people are ok with.

"We'd like to use Spotify Jams more, but we don't all have premium accounts."

She would like a fairer split between preferences. It would need to be easy because she and her friends aren't going to coordinate an entire playlist together.

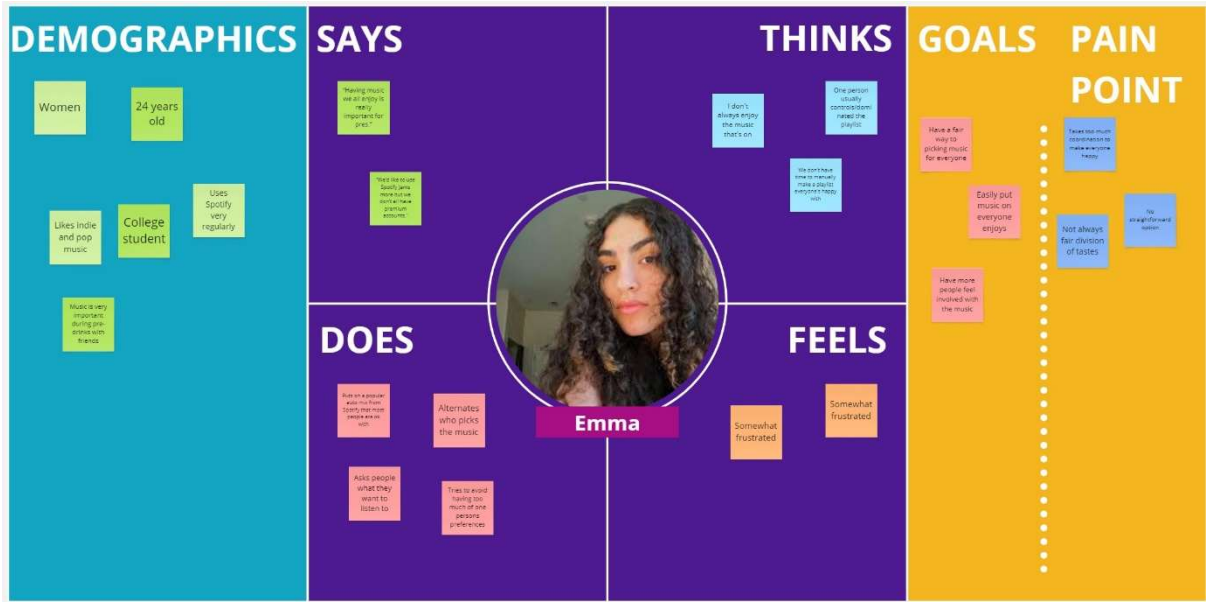

Figure 3-11 - Empathy map for Emma

Patrick:

A 35-year-old retail manager with a partner and child. He listens to car radio on the commute into work and uses Spotify casually. He has a few musician friends with very strong music preferences. When they go on surfing trips, he lets one of those friends decide the music but has gotten tired of his rap music. He skips a lot of tracks of his friend's music but isn't bothered to make his own playlist.

"I'm getting pretty tired of all the back-to-back rap on the drives to Lahinch"

He'd like an option that has something for him and his picky friends but doesn't require manually selecting tracks. He doesn't like fiddling around with playlists so needs a set-it-and-forget-it way to accomplish this.

Figure 3-12 - Empathy map for Patrick

3.3.2 Functional requirements

These are the requirements for the app that are needed for basic functionality, the minimum viable product (MVP). Without these features the app would not work and so are highest priority.

1. Connect to users Spotify and retrieve data
2. Create a playlist containing tracks from each users listening history
3. Save the playlist to Spotify
4. Save user data (access tokens, profile, listening history)
5. Shareable link to sessions

3.3.3 Non-functional requirements

These requirements are not related to core functionality, but they are still relevant. After functional requirements are met, these are the next requirements. Broken into usability, security and performance, they detail features of the app that would enhance it but are not essential for the MVP.

Usability:

- Live voting on tracks in playlist
- Filter track preferences based on artist or genre

- Users can view previous playlist sessions

Security:

- Database security rules, only auth users can edit own data
- Tokens are encrypted in database
- Only invited users can join session

Performance:

- Users listening history is periodically updated (reduce loading times when making playlist)
- Caching frequently used tracks to reduce API calls

3.3.4 Use Case Diagrams

The use case diagram shows the entities involved in the application, like the user and Spotify, and the tasks they need to accomplish as defined in the functional requirements section (Figure 3-13 - Use Case Diagram). For example, the user needs to be able sign-in which requires authentication from Firebase Services.

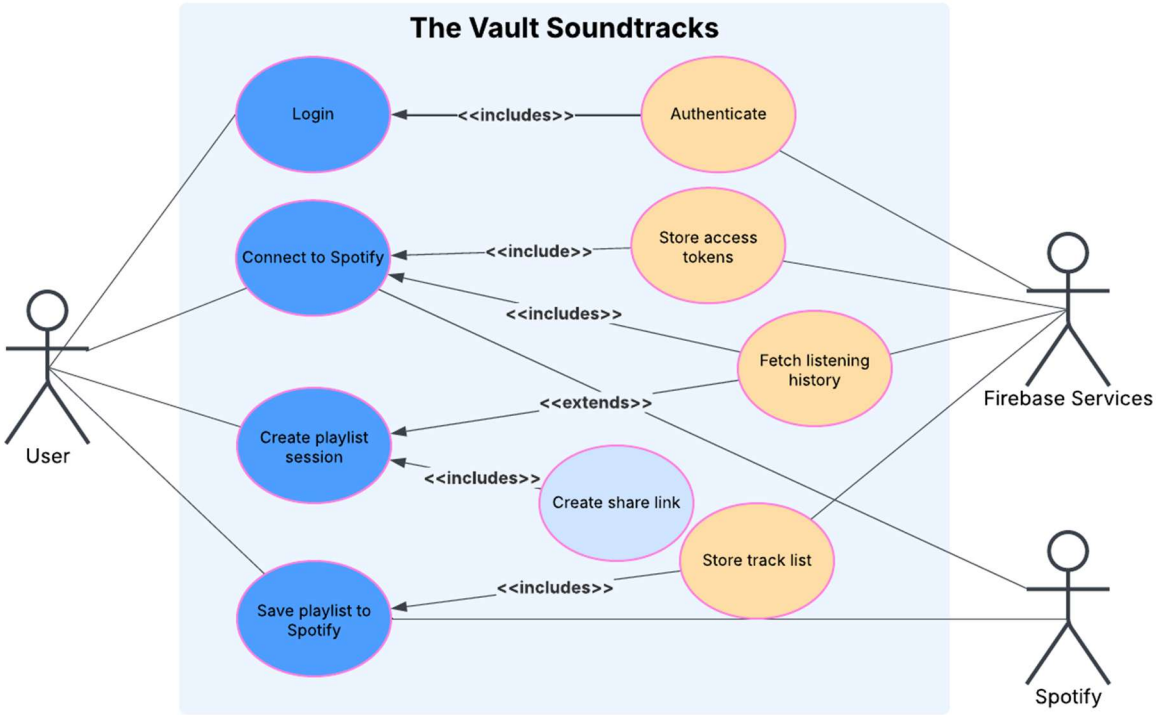

Figure 3-13 - Use Case Diagram

3.4 Feasibility

The app used Flutter in the frontend and Express.js with Node.js for the backend. The backend will handle all API requests to the Spotify Web API. It will also store and retrieve data from a NoSQL database using Firebase Firestore. The Spotify OAuth process and token management will be dealt with in the backend, storing access tokens in the database.

Feasibility Considerations:

- Handling the Spotify authorisation process may be complex as access token storage and refresh functions will be required.
- Real-time collaboration needs to handle multiple users updating data (atomic transactions)
- Spotify user permissions and rate limits

3.5 Conclusion

Through analysing similar apps, it was determined that the features purposed for this project currently aren't available in one existing application. They appear across both apps looked at, confirming they are relevant features. The survey revealed that although many people appeared interested in these features, they were unaware that they already existed, again confirming the relevance of this project.

The functional and non-functional requirements outline the features that users said they would find useful. The app was broken down into use cases to define the specific actions that would be needed. Finally, the feasibility was discussed and the considerations to be taken were outlined. These include how Spotify authentication should be handled and how real-time collaboration will be supported with atomic transactions.

4 Design

4.1 Introduction

This section will focus on the application design in terms of the technologies used and how they are structured, and the user interface (UI). The technologies will be outlined and their relevance and suitability discussed.

The structure and design patterns of the front and backend technologies will be explored. The overall application architecture will outline how the parts of the application interact, including external API's and the database architecture.

The UI design section will look at the initial wireframe designs, user flow diagrams and the style guide followed. The final design will be presented and discussed in terms of support for functional and non-functional requirements.

4.2 Program Design

Each technology will be discussed along with its relevance to the project, noting any potential difficulties that might arise. The application architecture includes the database structure and how different parts of the application interact with each other.

4.2.1 Technologies Overview

Technologies used in this application:

- Flutter
- Node.js using Express framework
- Firebase Firestore (NoSQL) & Authentication
- Spotify Web API
- Jest for JavaScript testing

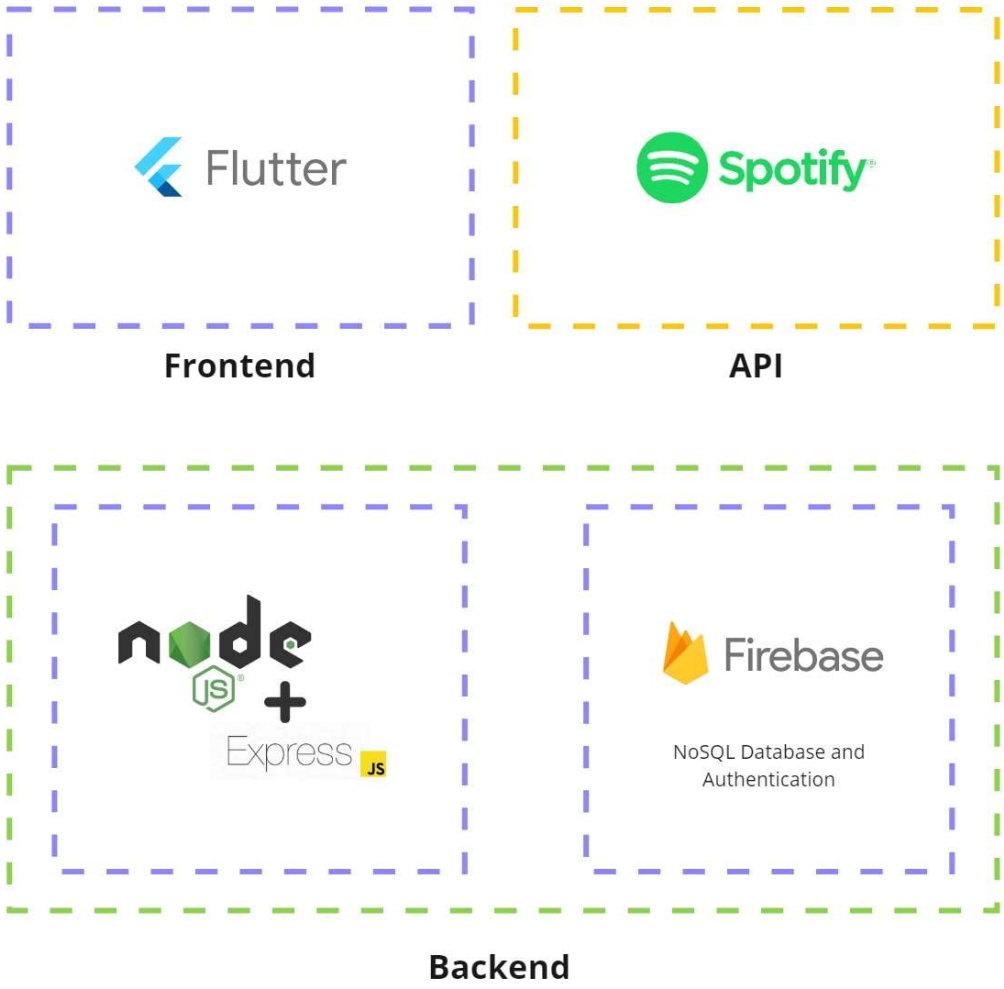

Figure 4-1 - Technology Stack

The technology stack (Figure 4-1) includes Flutter for frontend Android development, Spotify's Web API for song and playlist data and Firebase's User Authentication service. The backend is built using Node.js, with the Express.js framework and uses Firebase Firestore for a NoSQL database. The testing library Jest was used for automated testing of the Node.js backend.

4.2.2 Frontend

4.2.2.1 Flutter

The frontend technology chosen is Flutter, a cross-platform framework that uses the Dart programming language. A cross-platform framework allows developers to build native-like applications for multiple devices like Android, iOS or web, using one codebase. As Flutter was developed by Google, it offers simple integration with Google's Firebase services, which will be discussed in the backend technology section.

The potential difficulties associated with Flutter include its entirely widget driven structure. This differs from React and React Native (RN), which use a HTML like structure that is common for the web. Learning to develop using a widget tree approach required additional research and learning. Understanding how the widget tree builds and rebuilds was essential for developing a smooth UI.

As Flutter is a cross-platform framework, developing for web and Android simultaneously is possible. Android development was focused on because of the results from requirements phase indicating mobile apps were more appropriate. The application may still function on the web, although this platform was not included in the testing phase.

Due to being developed by Google, Flutter enjoys easy integration with Google's other products and services, like Material Design 3. This design system is used in all the default widgets offered in Flutter. This means that many of the needed widgets will require minimal custom styling as they already meet a high standard of UI design. No additional library or styling system needed to be added to the project, reducing dependencies.

RN was considered for the frontend as it is also a cross-platform framework. It would have been technically suitable; however, Flutter was chosen as it was an unfamiliar technology and offered the opportunity to learn a new language and framework. It also offered an opportunity to work within the Google ecosystem of Flutter and Firebase services.

4.2.2.2 Spotify Web API

Spotify offers a free API for developers to interact with the platform within applications. To get access to the API you must create an app in the Developer Dashboard. Access to the 'Client ID' and 'Client Secret' is then give, which is needed when requesting access tokens for the API.

The app defaults to Developer Mode, meaning the number of users and API calls are limited (Apps, Spotify for Developers Documentation). These limits, up to 25 authenticated Spotify users (Quota modes, Spotify for Developers Documentation), are more than sufficient for development and only require that a test user be added to a user whitelist to allow them access.

There is a Node.js based library for the Spotify Web API that allows users to interact with the API more efficiently, providing helper functions. This will ensure a more robust authentication flow and API requests.

4.2.2.3 Firebase Authentication

Firebase is a backend as a service (BaaS) that offers, among other things, an authentication service. This service allows users to register and sign in to an app, with Firebase handling the credential authentication and security aspect. It offers developers the option to authenticate using user email

and password, or the option to use other providers to authenticate. Other providers include Google, Facebook, Twitter and Yahoo. This gives users the option to register and sign in using their credentials from one of these providers, streamlining the authentication process.

4.2.3 Backend

4.2.3.1 Node.js with Express

Express.js is a popular JavaScript framework built on top of Node.js for backend web development. It provides robust features and tools for creating a server, handling routing and more. It was used to build the API to handle all business logic and interactions with the database. Using middleware, endpoints can handle user verification and similar security concerns. Using Node.js means the application has access to the large library of node packages, via the node package manager (NPM).

Firebase Cloud Functions was an alternative to Node.js that was explored. This would have hosted all backend logic in cloud services. This could have been a good solution if token management was the only function needed in the backend, but as requirements grew it became clear a full custom backend would be more suitable. Cloud Functions could still be used for smaller, more isolated logic like updating a user's listening history periodically. This feature was beyond the scope of the project but should be considered for future development.

4.2.4 Front and Backend Structure

4.2.4.1 Flutter

The main folders for development within Flutter are in 'lib', the library. Although Flutter doesn't enforce a strict folder structure, good practices were followed in creating the structure (Figure 4-2).

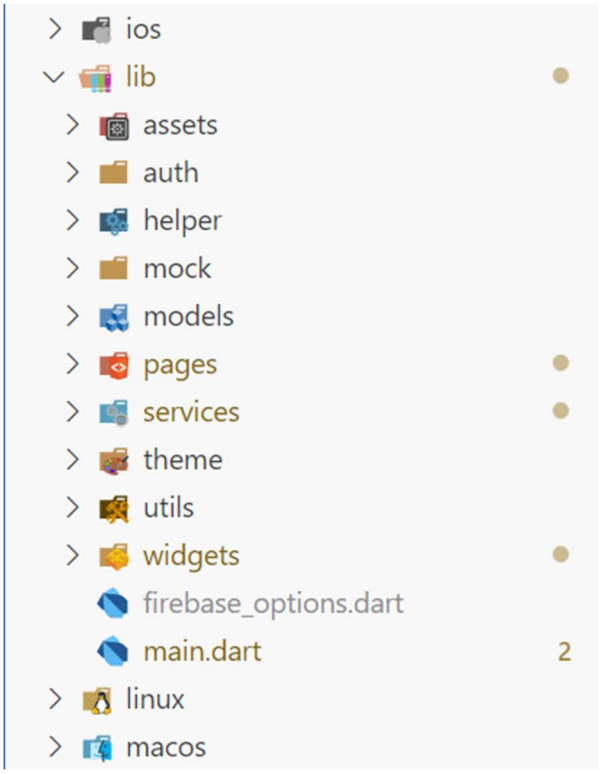

Figure 4-2- Flutter app folder structure

The assets folder contains any static images, such as logos for sign in options. Auth contains pages and logic specifically dealing with checking if users are signed in and displaying the appropriate page.

Mock folder contains static mock data used to test UI rendering before connecting to the API. As Dart is a type-strict language, data returned from the API needs to be correctly typed, so all models are defined in the model's folder. Services are where the main logic for each aspect of the app are kept. For example, operations dealing with Spotify directly, such as creating a playlist, are in the spotify_services.dart file.

Theme contains the configurations for dark and light mode of the applications UI. Utils (utilities) is where any helper functions are kept, like general dialog alerts. The widgets folder contains the custom UI elements that make up each page.

The pages folder contains each individual screen of the UI, such as Login, Homepage and Live Session. All distinct UI screens are stored in 'pages', distinct from 'widgets' which make up the elements of each page.

This structure follows a good Separation of Concern (SoC) practice in keeping each aspect of the application isolated, easily maintained and readable.

4.2.4.2 Express.js

The structure used for the backend (Figure 4-3) is like the Model-View-Controller (MVC) concept used in many frameworks. Although Express doesn't strictly require this structure, the project follows it partially and the SoC principle to ensure good maintainability and scalability.

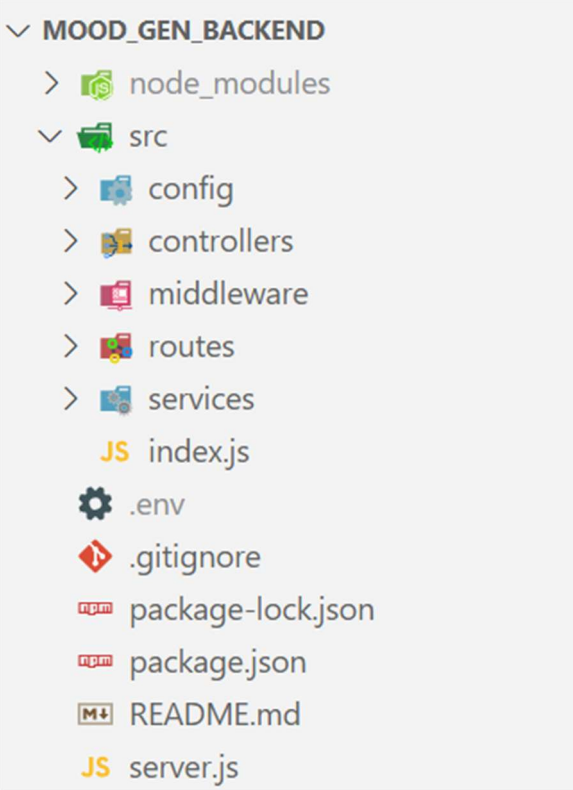

Figure 4-3 - Backend folder structure

As this is an API and not directly displaying visuals to the client, the 'view' part of MVC is not applicable. Routes are where all endpoints are defined. To make the application easily scalable, each aspect of the application has its own folder within routes (Figure 4-4). All Spotify specific endpoints are grouped, as are playlist session endpoints. This keeps each part of the app separate and organised.

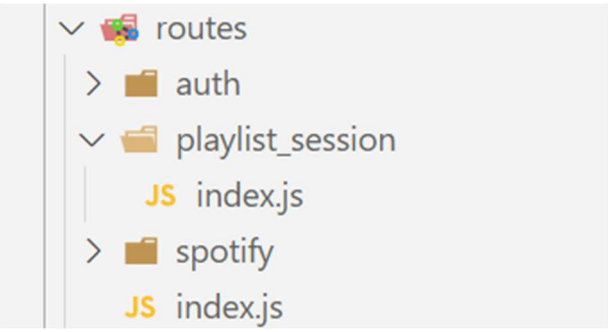

Figure 4-4 - Routes folder structure

The config folder (Figure 4-5) holds the files that handle the initial configuration and setup of the Firebase database and the SpotifyWebApi instance. The keys folder contains all information needed to connect to the apps database, as provided by Firebase. This file should not be committed to git as it contains sensitive keys and would compromise the security of the app if pushed to the public repository.

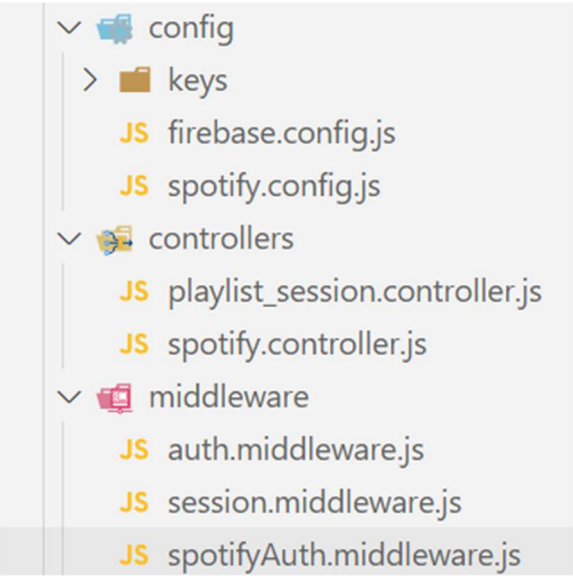

Figure 4-5 - Config, controllers and middleware folder structure

Spotify's config file is used to create an instance of the SpotifyWebApi which is used to make API calls to Spotify. The permission scopes are defined here, as are the clientId, clientSecret and redirect URI from Spotify. This file pulls the variables from the .env file, so it is safe to be committed as it contains no sensitive data.

Middleware is separated by concern, as are the controllers. Each dealing with one area of business logic. The services folder contains methods that deal directly with low level logic, such as updating the database or managing tokens.

4.2.5 Design Patterns

The entire project follows the SoC principle. This principle is fundamental in software engineering and aims to take complex systems and divide them out into smaller, more manageable parts. Each part should focus on just one aspect of the system. Complex processes are then created by combining the small parts together. The now modular system is more easily maintained, scalable and readable for future developers.

An example of SoC in the backend can be seen in the Spotify Controller method 'callback' (Figure 4-6). This method is part of the Spotify authentication flow and returns access token data. The logic to extract the access token from the request is stored in the TokenService.getAccessToken method and not directly in the Spotify controller.

```
22    try {
23      // Get the access token from Spotify
24      const accessTokenData = await TokenService.getAccessToken(req);
25
26      // Save the token to Firestore
27      await UserService.saveSpotifyToken(userId, accessTokenData);
28
```

Figure 4-6 - Code snippet from Spotify controller 'callback' method

The controller is for coordinating multiple services to accomplish the main task. If all the code needed for this 'callback' method was in the controller, it would be difficult to read, understand and maintain. The code would also not be reusable. Separating each step means the blocks of code can be reused and that the controller is easier to read. This is called encapsulation. It is especially helpful during testing because each step is small and only handles one piece of logic.

To save the token to the database, a few different things need to happen across multiple services. Handling this in the dedicated User Services class groups this logic together. Figure 4-7 shows two of the actions needed, both relying on the Firebase Services class.

```
6   class UserService {
7     // High-level business operations
8     static async saveSpotifyToken(userId, accessTokenData) {
9       const userEmail = await FirebaseService.getUserEmail(userId);
10
11      // Use FirebaseService for the actual database operation
12      await FirebaseService.setDocument("users", userId, {
13        spotify: accessTokenData,
14        spotifyConnected: true,
15        email: userEmail,
16      });
17    }
```

Figure 4-7 - Code snippet from User Service saveSpotifyToken method

Figure 4-8 shows the Firebase Services method to retrieve the user email by Id. Encapsulating the logic that deals directly with the database ensures the code can be easily reused and keeps the User Service method clean.

```
135   static async getUserEmail(userId) {
136     try {
137       const user = await admin.auth().getUser(userId);
138       return user.email;
139     } catch (error) {
140       console.error("Error getting user email:", error);
141       return null;
142     }
143   }
```

Figure 4-8 - Code snippet from FirebaseServices getUserEmail method

4.2.6 Application architecture

The system architecture follows a hybrid approach to the Layered Architecture pattern (Figure 4-9). In this design pattern each layer communicates with the above or below layer only. This isolates each part of the application, ensuring security and SoC. It works very well for teams as each team need only be concerned with their layer.

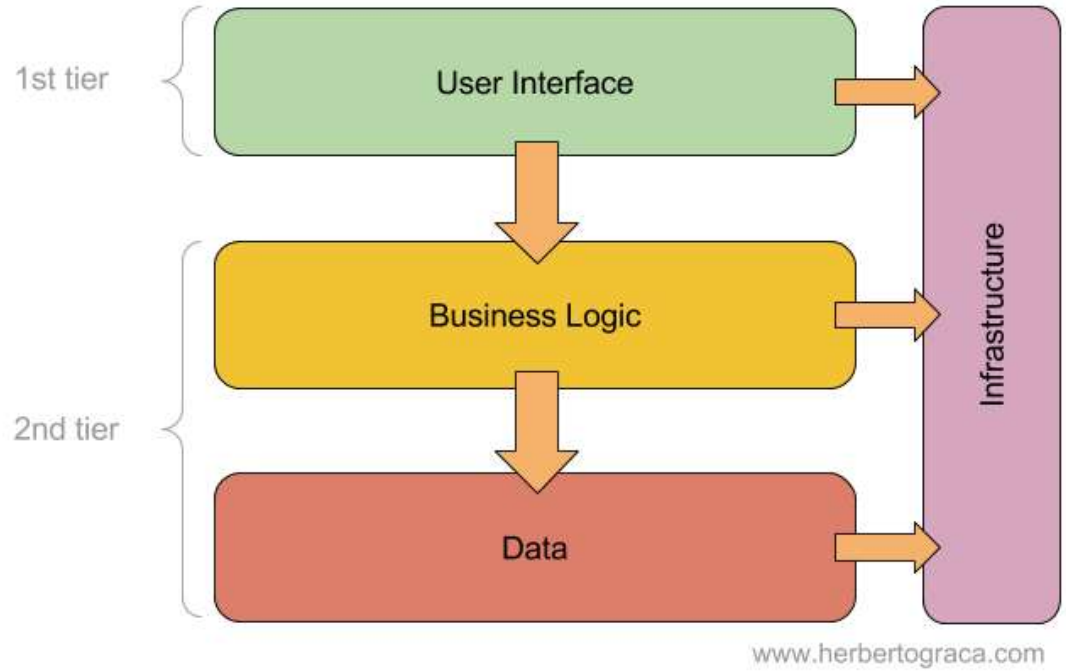

Figure 4-9 - Layered Architecture diagram

This application follows the pattern as its main principle but deviates when dealing with the live voting and authentication features (Figure 4-10). The voting feature requires live updates which

would not be possible if following the strict layer pattern. To accomplish the live updates needed, the user interface layer communicates directly with the data layer. However, it is important to note that no database writes are possible in this scenario, it is read only. This maintains the security and data integrity achieved through the layered architecture pattern.

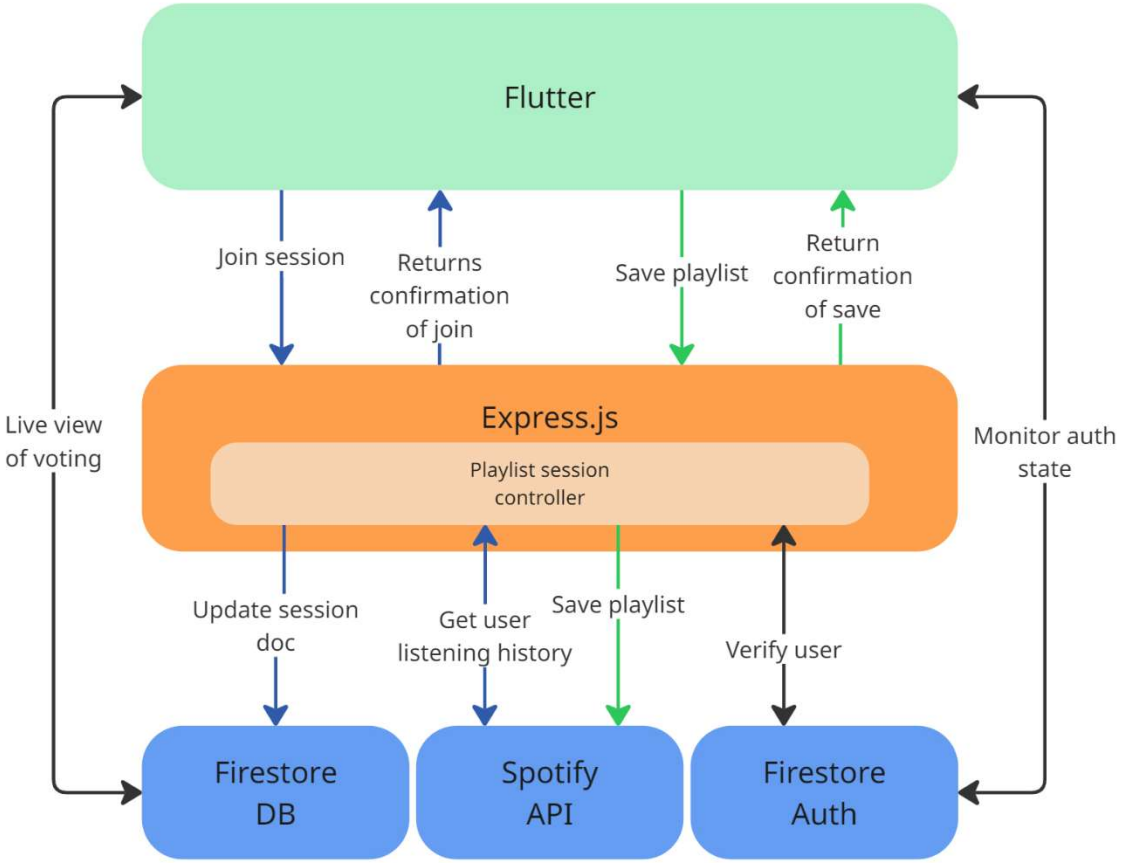

Figure 4-10 - Application architecture diagram

4.2.7 Database design

The database used is Firebase's Firestore NoSQL. This database uses a JSON like structure to store unstructured data in documents. This gives the database lots of flexibility as not all fields are required. Instead of using tables and rows like in a SQL relational database, NoSQL uses collections and documents.

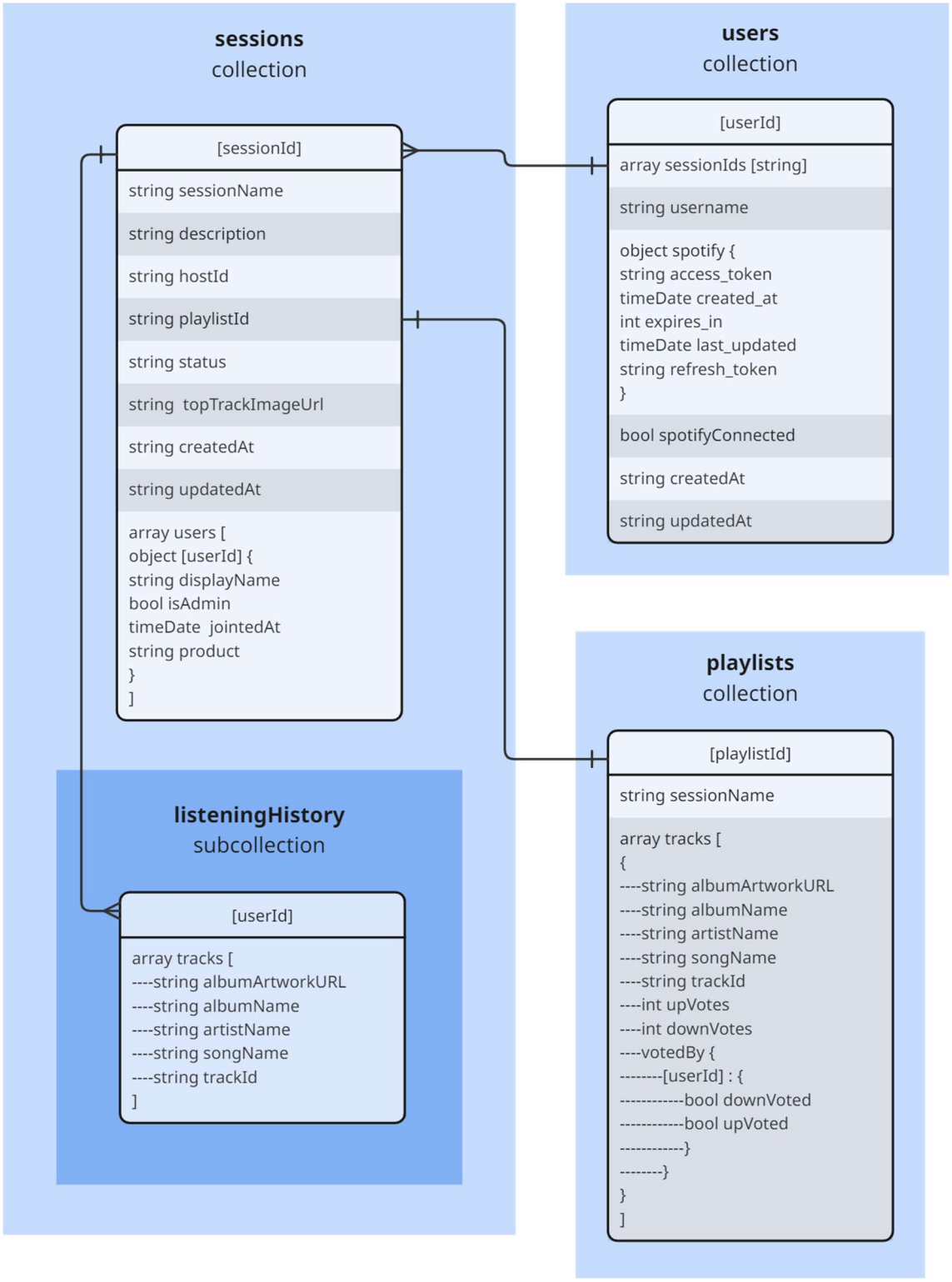

Figure 4-11 – The applications Firestore database structure

The user's collection, for example, contains a document for each user (Figure 4-11). This document stores all the user data in a JSON like format. NoSQL also allows for sub collections. The sessions collection contains a sub collection that stores each user's listening history. Meaning that for each session document, there is a nested collection that holds documents on listening history. Separating

data like this ensures that only data that is displayed together is stored together. This approach reduces the amount of data stored in each document. There is a limit to document size so separating the data can be a good approach to ensuring easy scalability. If all listening history was embedded directly in the session document, it has the potential to grow very large if many users join that session. Keeping it separate allows for easy growth and better querying capabilities.

Referencing other data is not as straightforward as in an SQL database. There are no foreign keys or indexes and join queries do not exist. This application doesn't require many join queries but to achieve the same result, the data is modelled to contain document Ids as references. An example of this is in the user document, where past session Ids are stored.

To enhance read performance on a NoSQL database, it is sometimes more efficient to denormalise data, also called embedding. This means duplicating the data to avoid multiple database queries. This approach was used multiple times across the application. The session document contains embedded data on the host user, namely the hosts Id and display name. Denormalising data is good for read-heavy operations that return large amounts of data at once. The downside is that the data can become stale and would need to be updated in many places if it changes.

The alternative would be to reference the data, also called normalisation. Normalisation keeps data separate and relies on a document reference Id being stored in a separate document. The advantages to normalisation are that data stays consistent and very large dataset are easier to manage this way. Disadvantages include the need for multiple queries and the references need to be updated if a document is deleted.

Data that is often displayed together should be stored together, even if it means duplication at times. Considerations on how the data is read/written need to be made when designing the database to achieve optimal performance in speed and data consistency.

4.3 User interface design

The aim of the design is to resemble Spotify so it's clear to users that Spotify powers the app. Having similar colours and fonts, as well as interactions will give a familiar feel to the users, as they are already used to Spotify. It could give the user more confidence in the app. This is done simply by using the three main Spotify colours, black, white and their signature green.

4.3.1 Wireframe

The first wireframes were done on paper at the beginning of the frontend development, before certain features had been refined. They include a numeric code as a joining option (Figure 4-12) and swipeable track cards for voting (Figure 4-13).

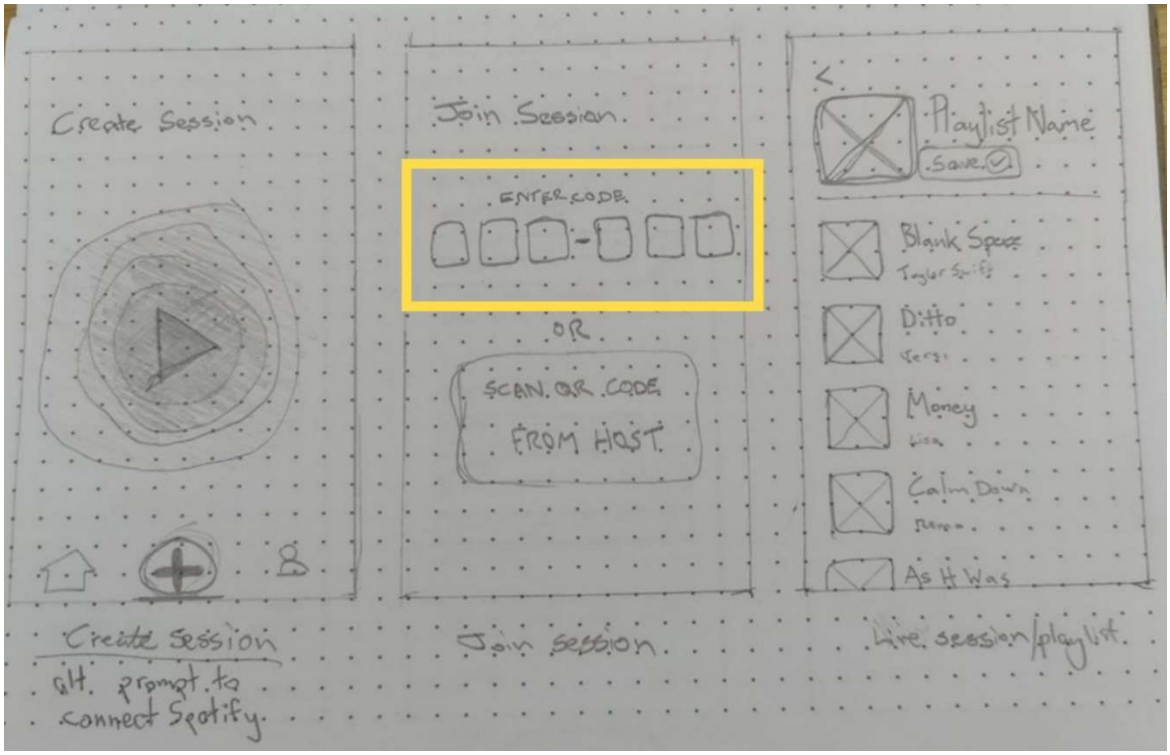

Figure 4-12 - Paper prototype for the create, join and live session pages

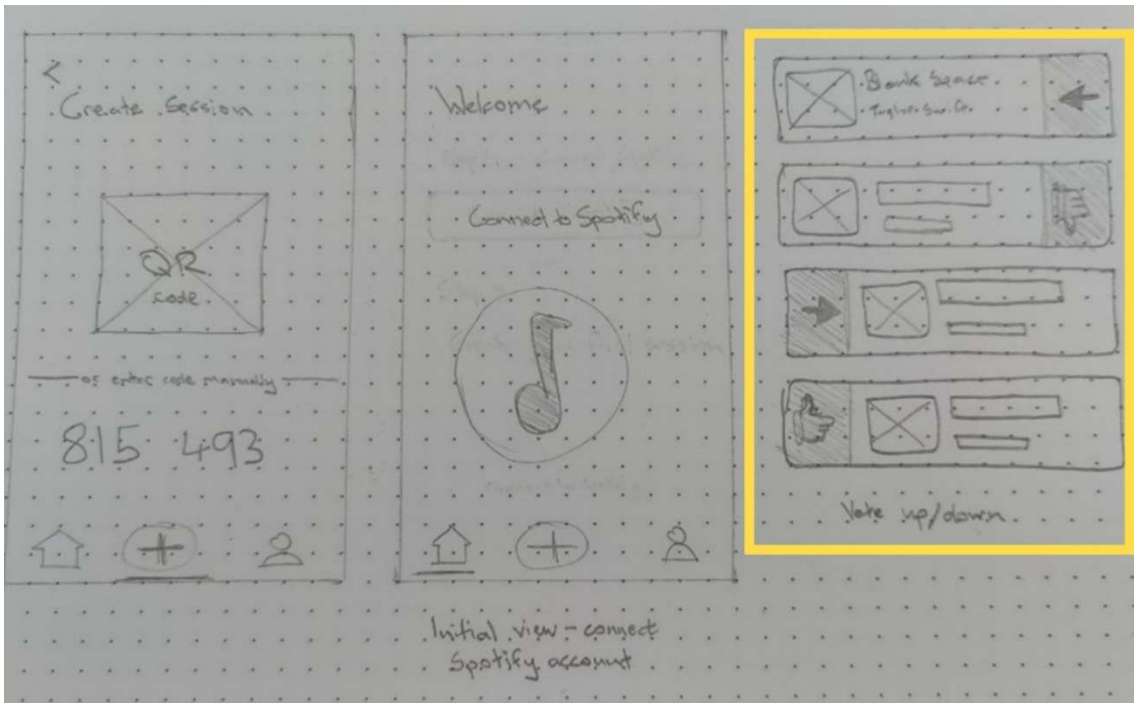

Figure 4-13 - Paper prototype for the share session, connect to Spotify pages and track card voting interactions

Going from the paper prototype, an initial high-fidelity prototype was made using the design software Figma (Figure 4-14). This prototype used the apps colour schema and Material Design 3 (M3) typography presets and card layouts. As M3 is the built-in design system for Flutter, using it during the prototyping phase made sure the UI would look as close to the end result as possible without needing to customise the Flutter interface much.

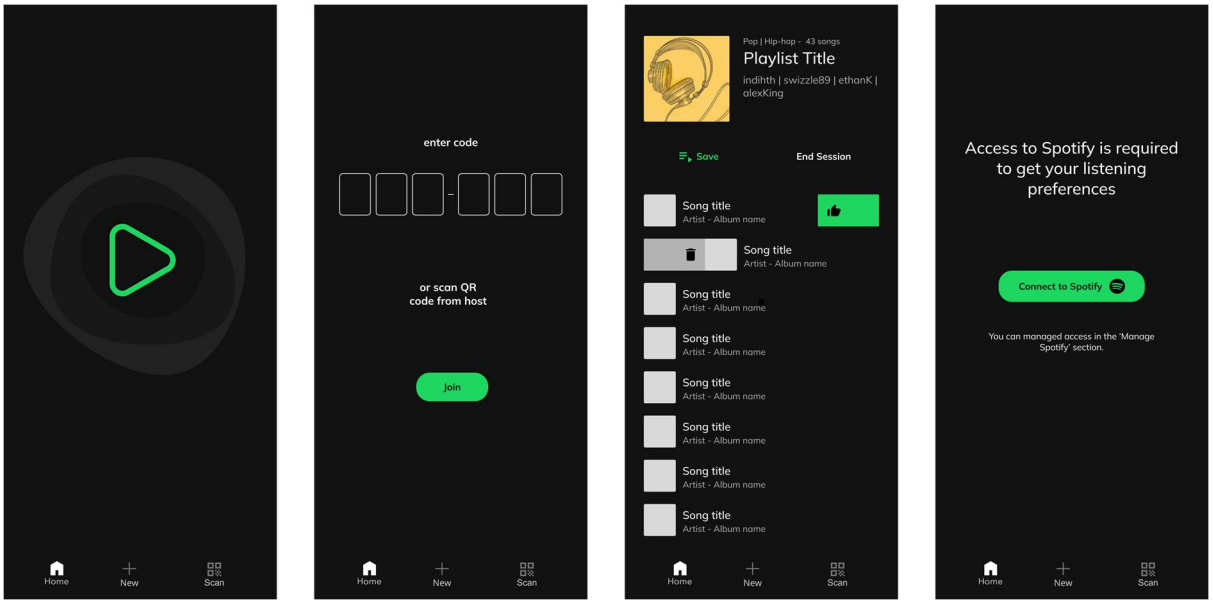

Figure 4-14 - High-fidelity prototype for session start, join, live and connect pages

The live session page was updated slightly after user testing revealed that some instructions would be useful. To help users quickly get an understanding of the voting feature some helper text was added (Figure 4-15).

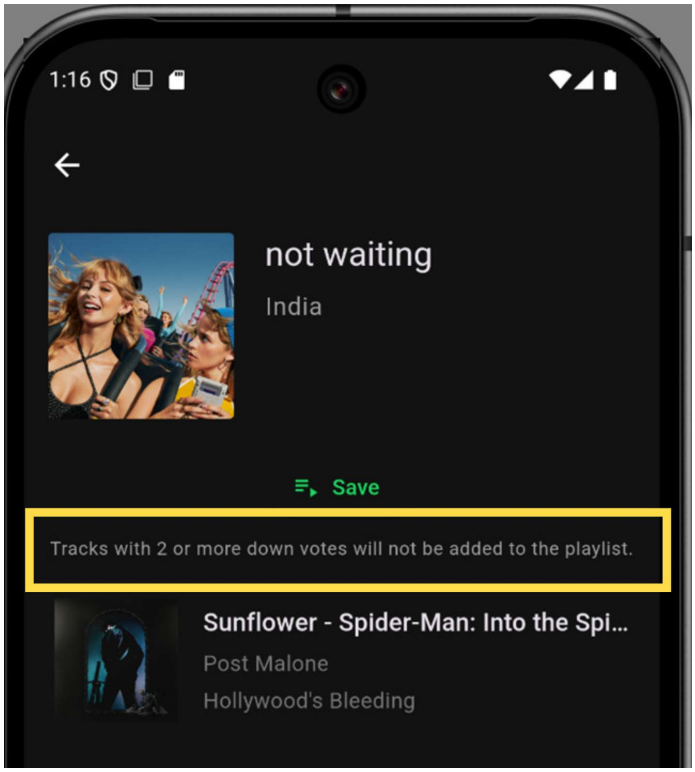

Figure 4-15 - Live session page with helper text

Figma offers options to create components and variants. This is very helpful when designing buttons and cards that have different states or variations (Figure 4-16). This allows components to be easily reused throughout the design, keeping everything consistent.

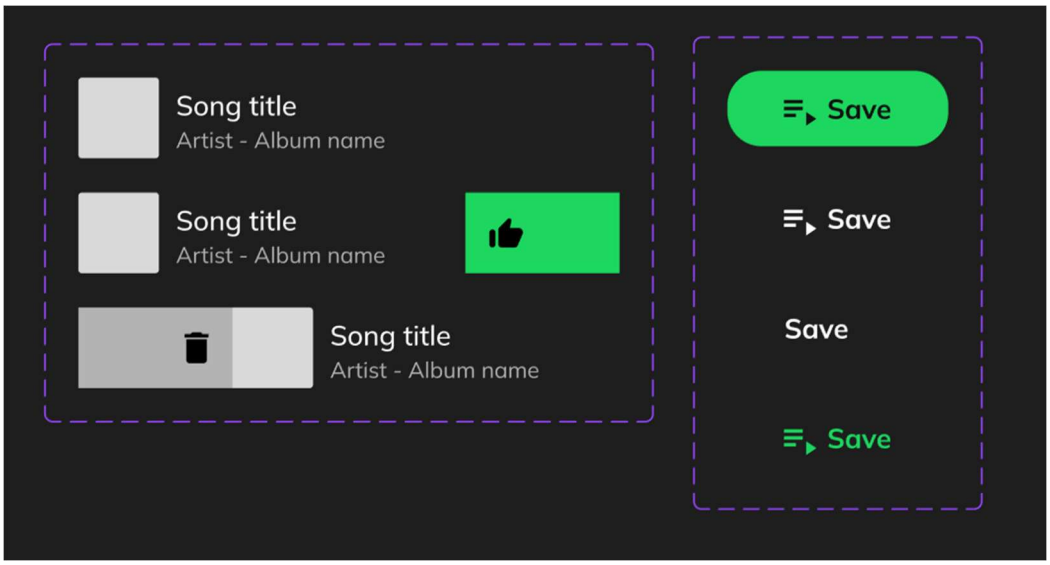

Figure 4-16 - Figma component variants

In later sprints, the homepage was completely redesigned and underwent many iterations (Figure 4-17). The layout of Spotify's homepage was taken into consideration, how different sized elements

are arranged and the ratio of images to text. The first designs were very text heavy, so efforts were made to incorporate images to bring more colour to the design.

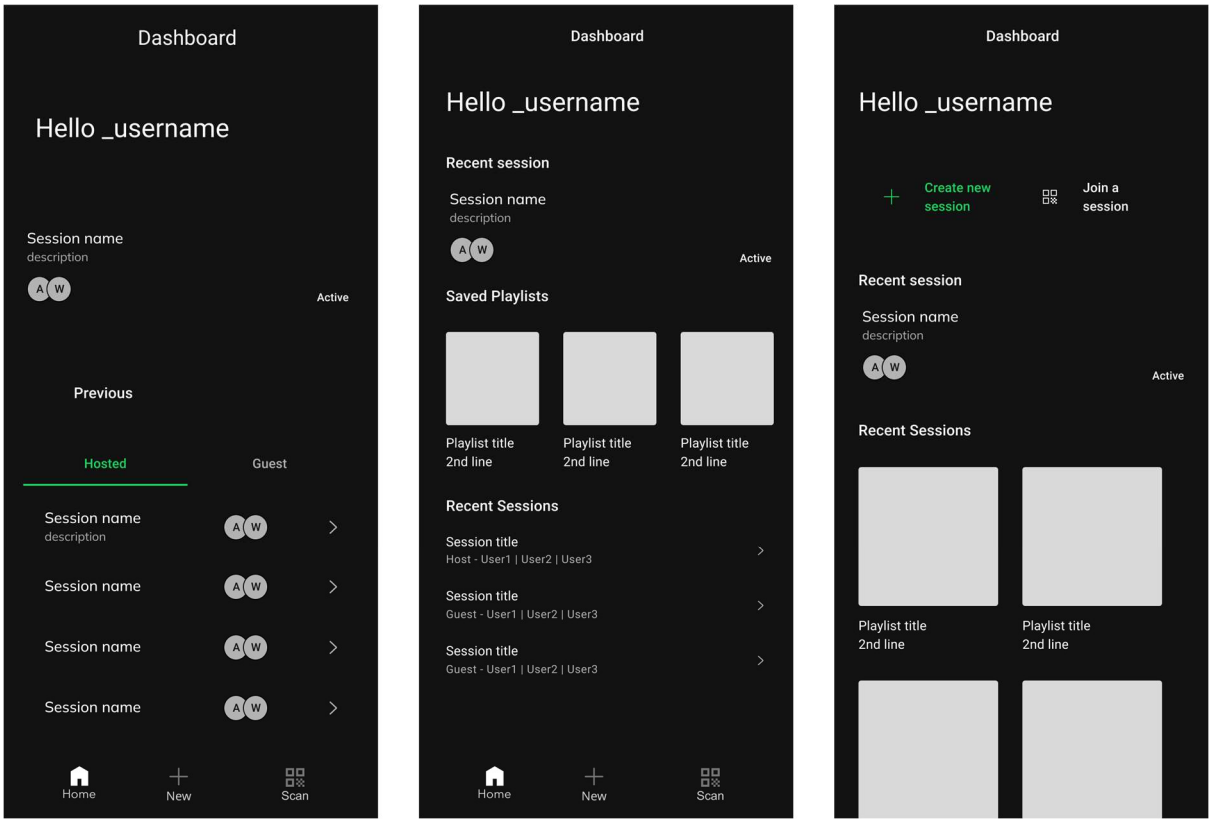

Figure 4-17 – Homepage design iterations

After the homepage, the Spotify connect page and the waiting-room page were redesigned, incorporating more colour through images (Figure 4-18).

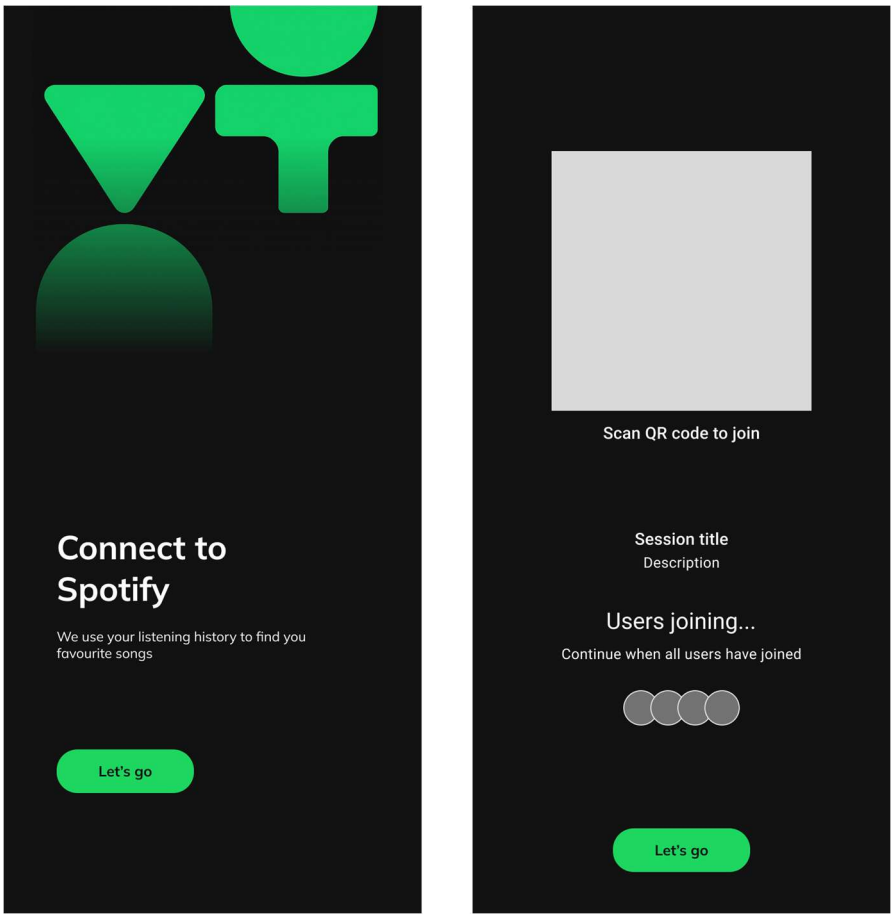

Figure 4-18 - Spotify Connect and Session Waiting pages

4.3.2 User Flow Diagram

The diagram (Figure 4-19) shows the user flow starting with either a login or register. The register flow includes the Spotify authorisation and the user's decision to allow the app access to their Spotify account, without which the app cannot function as designed. The home page has three options, create, join or view a past session. The create and join flows converge at the Live Session page. The host must choose to end the session.

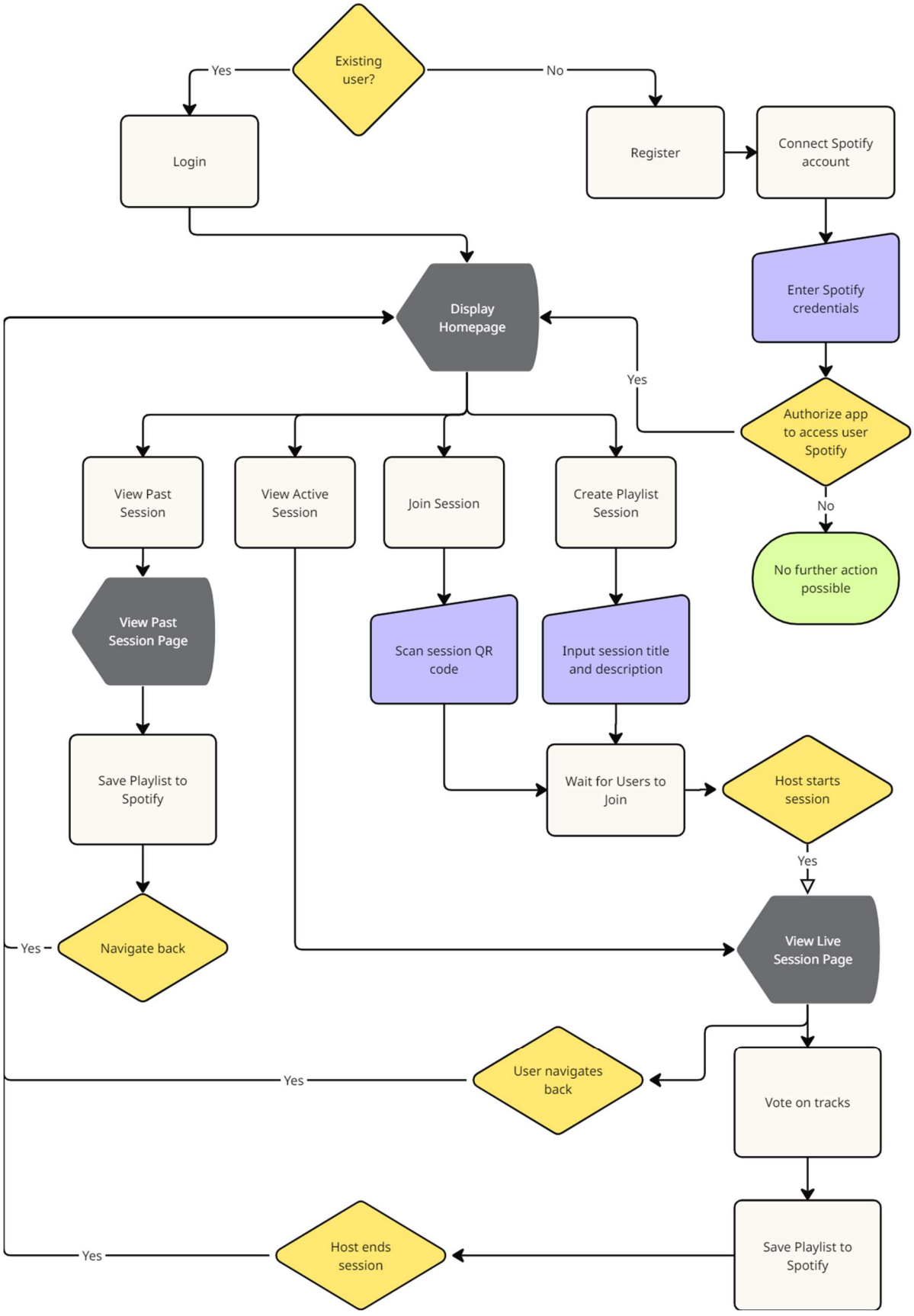

Figure 4-19 - User Flow Diagram for login/register, create, join and view a session

4.3.3 Style guide

The Spotify Design & Branding Guidelines (Design & Branding Guidelines | Spotify for Developers, n.d.) were followed where applicable for the colour scheme and displaying album artwork. Typography followed the M3 style guide as this was easily used in Figma during the design process as well as in Flutter as it is the default style setting.

4.3.3.1 Typography

A selection of the M3 typography was used throughout the Flutter app (Figure 4-20), using the default Android font of Roboto. Keeping to these predefined font sizes and weights-maintained consistency in the design.

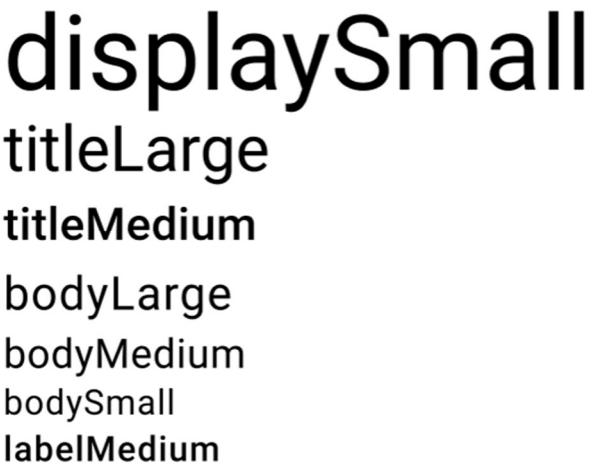

Figure 4-20 - Typography Styles used

4.3.4 Colour Scheme

The colour scheme was inspired by Spotify, consisting of three colours (Figure 4-21). The Spotify developer documentation provides exact values for the colours used, making it simple to define the theme settings in Flutter (Figure 4-22).

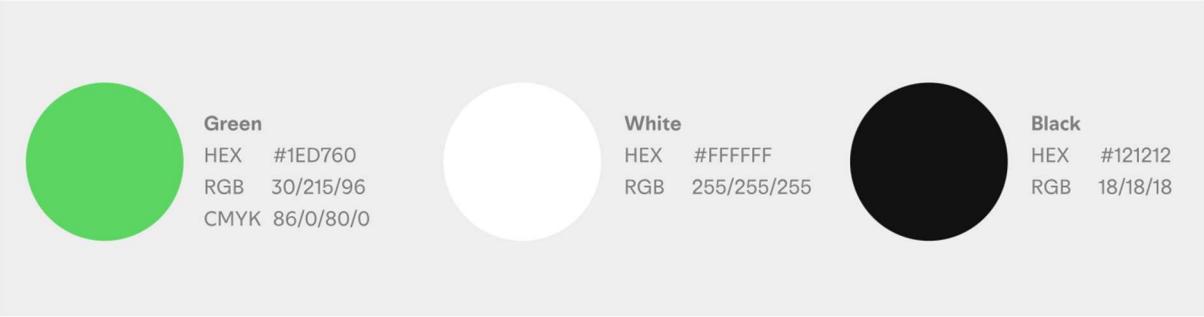

Figure 4-21 - Spotify colours

```
ThemeData darkMode = ThemeData(
    brightness: Brightness.dark,
    colorScheme: ColorScheme.dark(
      surface: Color(0xFF121212),
      primary: Color(0xFFFFFFFF),
      secondary: Color(0xFF777777),
      tertiary: Color(0xFF1ED760),
      inversePrimary: Color(0xFF121212),
    ), // ColorScheme.dark
```

Figure 4-22 - Flutter theme settings

4.4 Conclusion

This section discussed the technologies chosen for the front and backend of the application, explaining their relevance, advantages and disadvantages. The application followed good SoC practices, resulting in readable and reusable code that's easily maintained. The Layered Architecture design pattern provides modularity and security while being flexible where needed. The UI design process discussed the progression from wireframes to final implementations, with an emphasis on a consistent colour scheme and typography based on existing design guidelines from Spotify and Material Design.

5 Implementation

5.1 Introduction

This section will discuss each stage of the project's implementation, and the SCRUM methodology used.

The application has been developed using the following technologies:

- Flutter as the frontend framework
- Express.js as the backend framework
- Spotify Web API as an external service
- Firebase to handle authentication and database
- Jest for automated unit testing

5.2 Scrum Methodology

Scrum (Systematic Customer Resolution Unravelling Meeting) is an Agile development framework. Before Scrum became so prevalent Waterfall was a preferred methodology for many. Waterfall is another framework that is now less popular as it doesn't allow for as much flexibility as Scrum. Both Scrum and Waterfall are instances of Agile development.

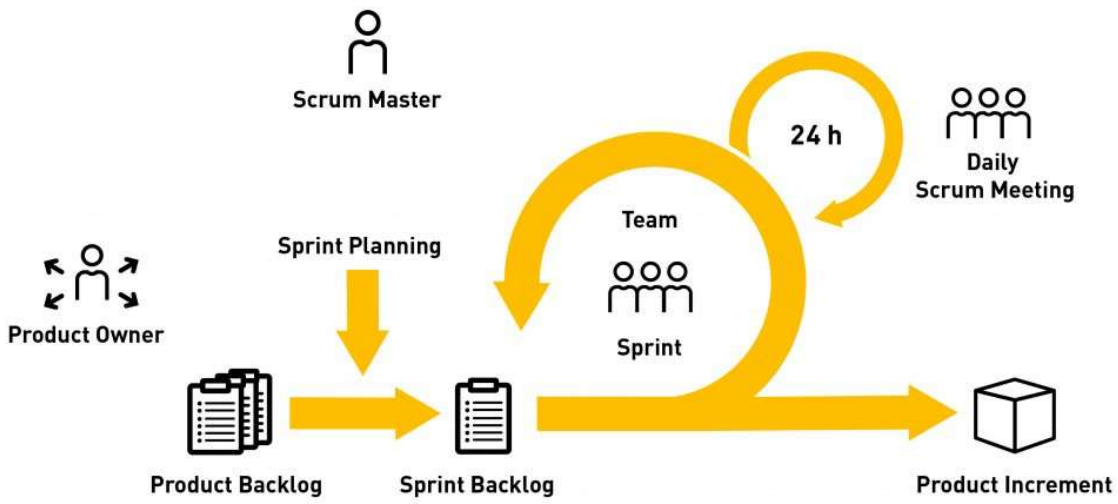

Figure 5-1 - Diagram of the SCRUM process

Scrum (Figure 5-1) works in small, distinct cycles known as sprints. At the very beginning of the project, the overall requirements are defined by the Product Owner. These requirements are then put into the Product Backlog which is used during the Sprint Planning phase. Each item on the

backlog is assessed and given a timeframe. Items are further broken down and added to the Sprint Backlog. This is all managed by the Scrum Master and passed on to the development team.

A daily Scrum meeting is done to update the team on what backlog items are done. At the end of the Sprint, the team and Scrum master will assess what was accomplished and what is still on the Product Backlog. If a feature was completed to a shippable standard, that product will be released. This process is repeated until all Product Backlog items have been implemented and the product is complete. This allows for incremental releases and for the team to react quickly and flexibly when issues arise.

This approach is favoured above the Waterfall method which is more linear. Waterfall (Figure 5-2) follows a stricter project plan that works in larger stages, delivering a product at the very end. It can be difficult to go back to an earlier stage to fix issues discovered later on. For example, a flaw could be found during the testing phase which requires a change in the design. Altering the design could mean that changes also need to be made in the programming phase, creating significantly more work for the team.

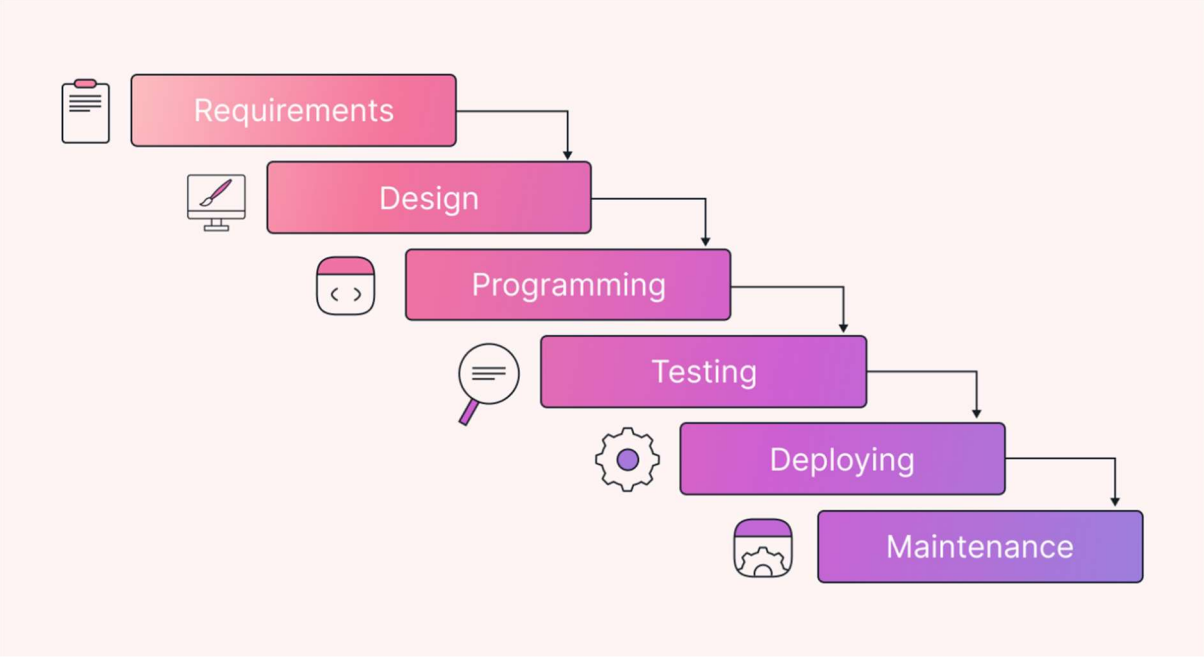

Figure 5-2 - Waterfall methodology

Agile methodologies like Scrum allow teams to work on smaller, more isolated features rather than the entire project at once. In Scrum, if a problem is found during a sprint's testing phase, the team won't need to adjust the entire project timeline to compensate. The issue will be raised during the daily Scrum meeting and if it can't be resolved within the sprint, it gets added to the backlog and

dealt with in a future sprint. The project is developed iteratively, allowing the team to deal with issues without bring the entire project to a halt.

This development methodology was used for the implementation of this project. It was managed using Miro, an online whiteboard platform. The Miro board (Appendix B – Project Miro Board) contained a Kanban (Figure 5-3) for tracking the backlog of implementation steps and boards for detailing each sprint (Figure 5-4). Tags were used to indicate what part of the project the task was related to. This was useful to quickly see if the backlog was, for example, mostly backend tasks. This made planning the focus of future sprints straightforward.

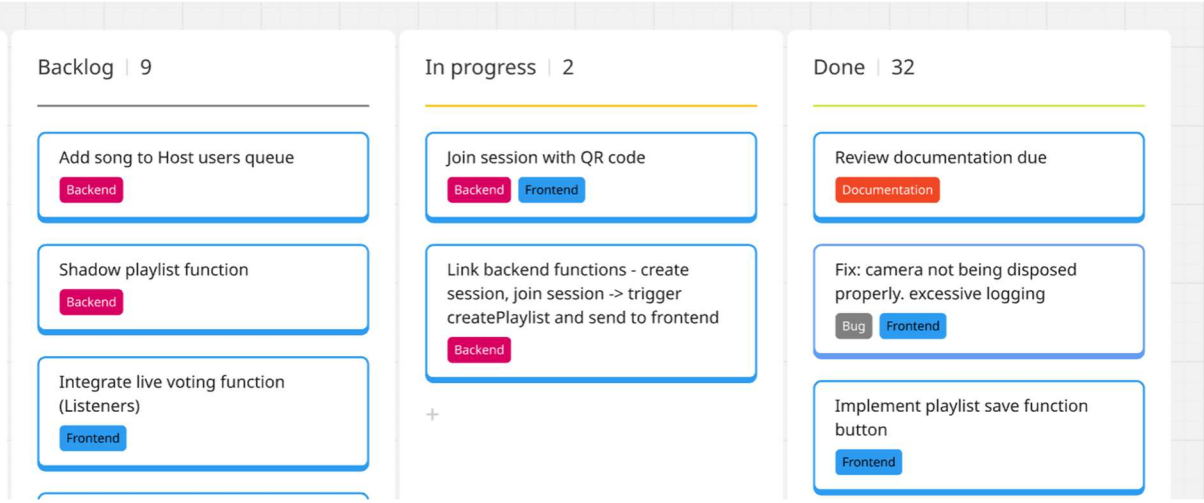

Figure 5-3 - Miro Kanban from Sprint 5

At the beginning of a sprint the goals were defined, and any relevant work was tracked. At the end, the progress was accessed, and it was determined whether the goals were met or not. If not, the issues were noted and added to the backlog to be worked on during a future sprint .

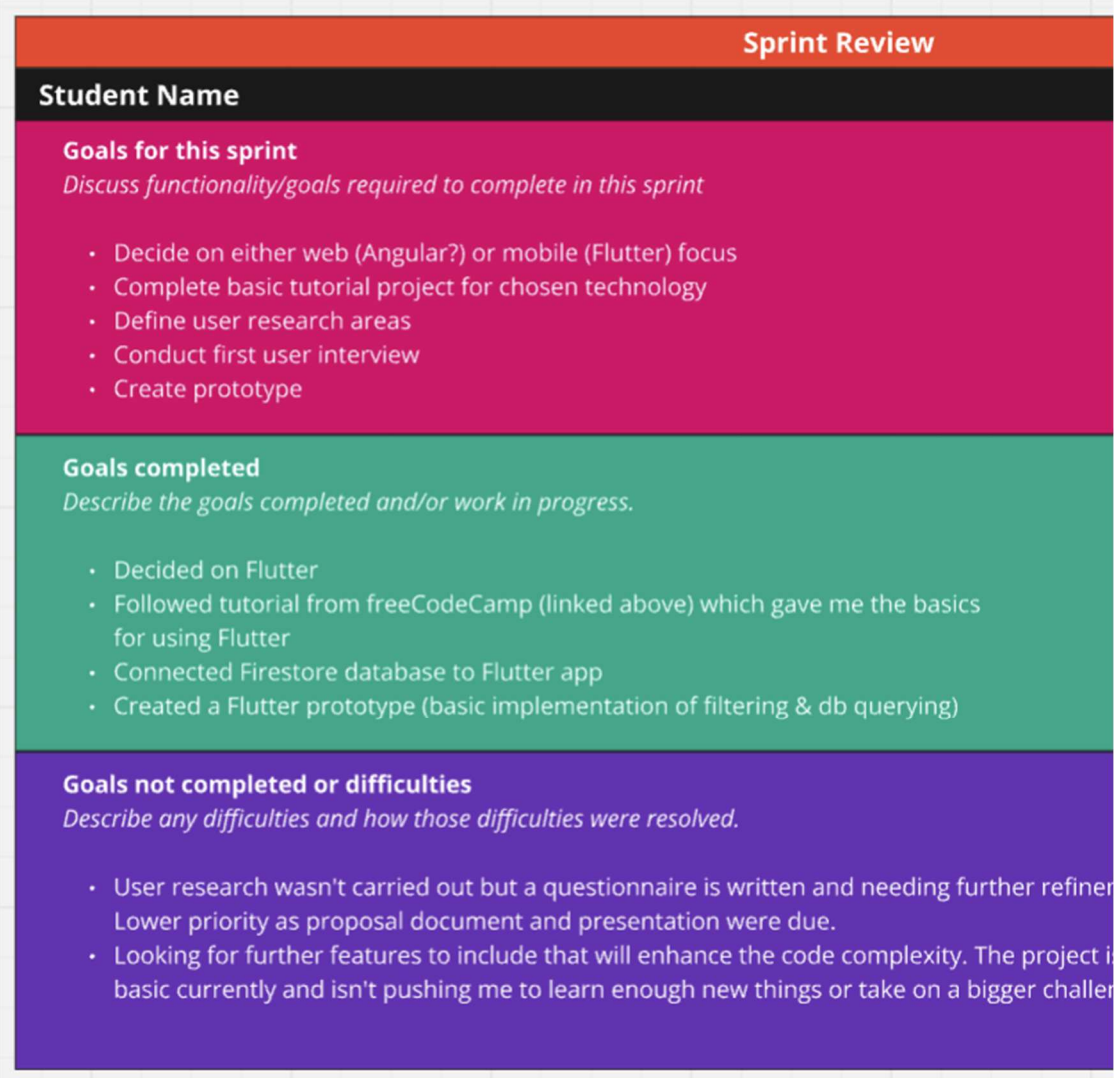

Figure 5-4 - Miro board sprint review format

5.3 Development environment

The integrated development environment (IDE) used was VS Code. It has extremely useful features like IntelliSense that offer smart code suggestions and a large extensions library for adding further language specific tools. It's integrated terminal and debugging tools means there is little need for additional software during development.

Having access to Dart and Flutter specific extensions was a must. The way Flutter uses a structured widget setup would make manually adding widgets quite tedious. For example, adding a Padding widget to a ListView can be done with a few mouse clicks using the Flutter extension.

The other important development software used was Android Studios. This provided easy use of an Android emulator that could be connected to VS Code to run the Flutter app on the development PC. It is also possible to use a physical mobile device in USB-Debug mode with VS Code.

Enabling the developer mode and USB Debugging on a smartphone makes it discoverable in the IDE and easily used as a testing device. This is particularly helpful when testing the camera integration for the app. Some additional setup is needed when running the front and backend locally as the phone doesn't have direct access to the PC's localhost address. Port-forwarding allows any traffic from the app on a physical device to be sent through the PC, resolving these network issues.

VS Code has integrated support for Git control, using GitHub. Once the project was initialised and the base file were generated, a repository was created on GitHub. When working on a specific feature, changes to the related files were all committed to the repository with a descriptive message. Commits should be logically grouped and focus on small, meaningful changes relating to a specific feature, bug fix or refactoring. Commit messages should be clear and descriptive. It is good practice to commit frequently and push at the end of a work session.

Effective use of branches keeps the repository well-structured and functional. Implementing a new feature or changing code for an urgent fix are good reasons to use separate branches. The changes made will not affect the main branch but can be merged later when the feature is complete, or bug is fix. This ensures that the main branch is always stable.

5.4 Sprint 1

5.4.1 Goals

This sprints main goals focused on:

- Investigating using the OpenAI API to generate additional attributes for songs through lyric analysis
- Finding the most suitable type of recommender system

5.4.2 OpenAI API chat completion

Using the OpenAI API, values were generated for predefined attributes for songs by analysing the lyrics. Creating a consistent response structure that could be added to a database was very important.

Track and lyric data were gathered from Kaggle. Kaggle is a platform where users upload datasets for use by others for the purpose of data analysis, machine learning and data visualisation (Appendix C – Kaggle Spotify Dataset Sample).The track and lyric data were merged using Python in a Jupyter Notebook.

The request required the AI model name, a system message to define the AI's job and a user message to define the user input, in this case song data. The API offers the 'response_format' option

(Figure 5-5). This ensures the models response conforms to a predefined structure, ensuring consistent results.

```
// Prepare the chat input
openai.chat.completions
  .create({
    model: process.env.OPENAI_MODEL,
    messages: [
      {
        role: "system",
        content: ` You are a lyrical analysis assistant. analyze the following
      },
      {
        role: "user",
        content: JSON.stringify(song),
      },
    ],
    response_format: {
      type: "json_schema",
      json_schema: {
        name: "song_emotions",
        schema: {
          type: "object",
          properties: {
            fear_anxiety: {
              type: "number",
              description:
                "Quantifies the fear or anxiousness about the future and what
            },
            hopefulness: {···
            },
```

Figure 5-5 - OpenAI API chat completion request structure

The response was then stored in the existing JSON file, adding the newly generated attributes for each song. The OpenAI analysis results are stored under ‘results’ (Figure 5-6).

```
},
"results": {
  "fear_anxiety": 0,
  "hopefulness": 0.8,
  "triumph_overcoming_adversity": 0,
  "heartbreak_level": 0.3,
  "longing_yearning": 0.5,
  "sadness_grief_level": 0.2,
  "romantic_focus": 0.7,
  "love_compassion": 0.6,
  "empowerment_level": 0.8,
  "revenge_anger_level": 0,
  "regret_remorse": 0.2,
  "nostalgia_factor": 0.5,
  "energy_intensity": 0.8,
  "euphoria_joy": 0.7,
  "resignation_acceptance": 0.2,
  "lyric_vs_sound": [
    {
      "result": 1,
      "reasoning": "The upbeat and carefree lyrics
    }
  ]
}
```


Figure 5-6 - OpenAI API analysis results

These lyric analysis requests could be done as batch requests (Figure 5-7). OpenAI offers a 50% discount when doing batch requests as they can take up to 24hrs to complete. A regular request is processed immediately, using the available resources from OpenAI. A batch request will wait to process until the systems resources are less strained, like off-peak discounts for electricity usage. The test batch request completed in under 10 minutes.

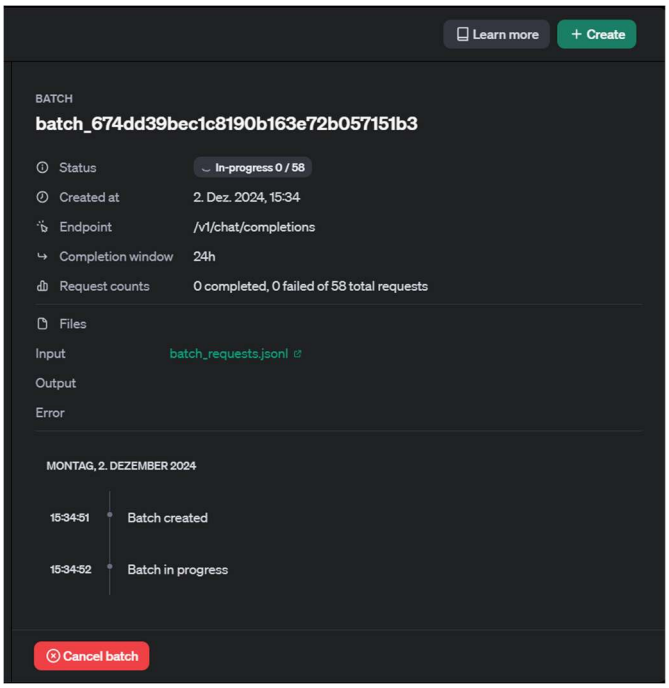

Figure 5-7 - OpenAI Batch Requests

Much trial and error was needed to get back the desired response from the model. Discovering the 'response_format' option greatly improved the responses. A future issue would be the number of tokens used in each request. Words are broken down into tokens. There are restrictions on the number of tokens to be sent in each request, meaning only a handful of track lyrics could be processed at a time. This required batch requests to process a high number of songs.

5.4.3 Machine learning recommender system vs filtering

Using TensorFlow.js for machine learning was briefly investigated as a possible option for the recommender system/filtering. This was ultimately discarded because it fell quite far outside of the project scope and general focus of the application. It would require a large amount of existing playlist data for the model to train on. Collecting this data would have been complicated as a metric for scoring how good/bad a playlist was would be needed. This kind of data isn't readily available and would require a lot of time to generate. It was instead decided to use logic filtering. Content-based verses collaborative filtering methods were explored.

5.5 Sprint 2

5.5.1 Goals

- Design and develop a basic prototype of the application for proposal presentation that has relevant data and connects to the Firestore database.

5.5.2 Flutter prototype

Using Figma, a user interface (UI) was designed for the Filter and Results page of the prototype (Figure 5-8).

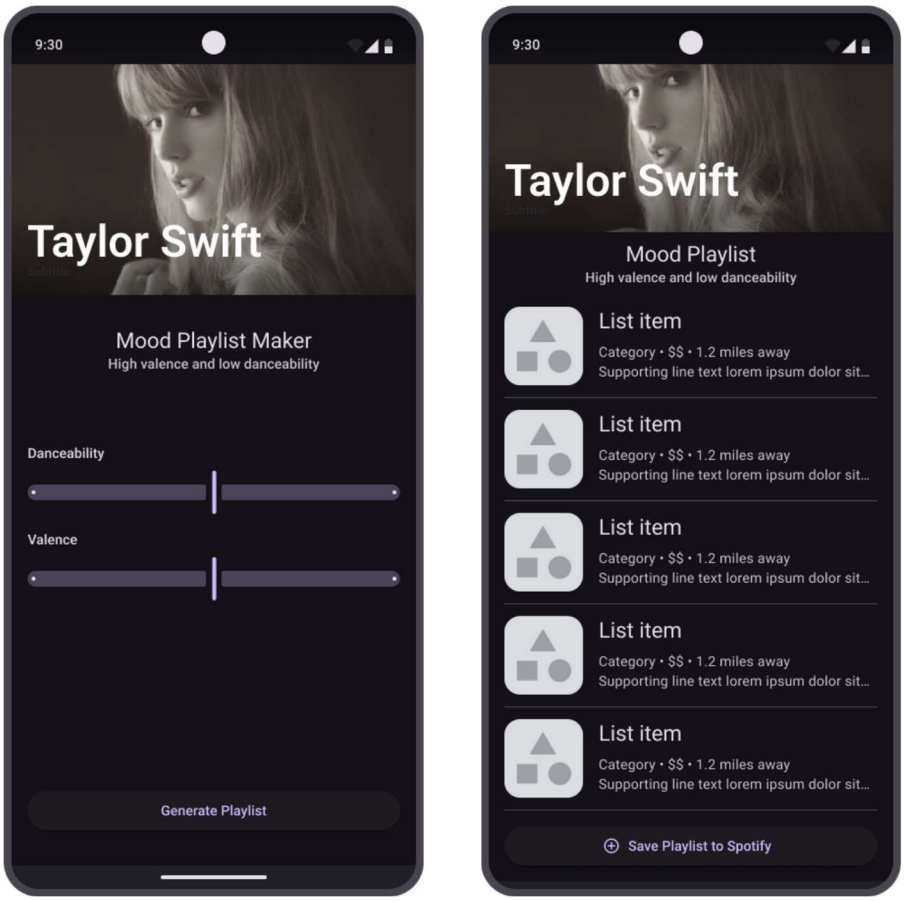

Figure 5-8 - Prototype UI Design

The user must be able to control the filter options using interactive sliders for each attribute. These slider values must be used to query the database and return the resulting tracks in a scrollable list view. Each track must display the track and album names.

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform,
  );
  runApp(const MyApp());
}
```

Figure 5-9 - Initialise Firebase in app

To ensure Firebase services are available in the app it needs to be initialised before the main app runs. This is done why using 'await' with the Firebase.initializeApp() function (Figure 5-9). This function requires the Firebase options, provided as a JSON file in the Firebase web console.

On the filter page, multiple slider widgets are used to control each attribute value (Figure 5-10).

```
// Valence slider
SliderFilter(
    label: 'Valence',
    value: userFilter.valence!,
    onChanged: (value) =>
        _updateFilterValue('valence', value)), // SliderFilter
```

Figure 5-10 - SliderFilter widget

This is a custom widget that uses Flutters Slider widget with the pre-defined values for the divisions and label options. A Filter model was created to define the properties and types of each filter object (Figure 5-11).

```dart
class Filter {
  double? valence;
  double? danceability;
  String? album; // optional filter
  double? popularity; // optional filter
  double? energy;

  Filter({
    this.valence,
    this.danceability,
    this.album,
    this.popularity,
    this.energy,
  });

  // Convert to a Map for Firestore queries
  Map<String, dynamic> toMap() {
    return {
      if (valence != null) 'valence': valence,
      if (danceability != null) 'danceability': danceability,
      if (album != null) 'album': album,
      if (popularity != null) 'popularity': popularity,
      if (energy != null) 'energy': energy,
    };
  }
}
```

Figure 5-11 - Filter model

Each time the filter slider was changed the _updateFilterValue() function was run to set the new state for the attribute.

A FirestoreMethods class was created to handle logic related to the database. The fetchData() function returned a Future (similar to a Promise in JavaScript) of a list of songs. The Song model was also defined, as required by Flutters strict typing. Very limited results were being returned because the combination of filters was removing too many tracks. To combat this, the acceptable range of results for each attribute was widened by giving each attribute filter a range of 0.40 (Figure 5-12). Attribute values range from 0-1, so adding this buffer increased the results for the small test dataset.

```
if (filter.danceability != null) {
  query = query
      .where('danceability',
          isGreaterThanOrEqualTo: filter.danceability! - .20)
      .where('danceability',
          isLessThanOrEqualTo: filter.danceability! + .20);
  print(
      'Added filter: danceability between ${filter.danceability! - .20} and $
      {filter.danceability! + .20}');
}
```

Figure 5-12 - Firestore danceability query

To display the results, a snapshot of the query results needed to be returned and displayed in a ListView widget using the builder method (Figure 5-13). The builder takes each item from a list and renders it once. Understanding what a builder was took some time as it's a new concept.

```
Expanded(
  child: ListView.builder(
    itemCount: results.length,
    itemBuilder: (context, index) {
      final song = results[index];
      return ListTile(
        title: Text(song.name),
        subtitle: Text('test'),
      ); // ListTile
    },
  ), // ListView.builder
), // Expanded
```

Figure 5-13 - Displaying returned tracks from database

The Flutter prototype was developed successfully and demonstrated in the interim presentation for this project (Appendix D – Sprint 2 Flutter Prototype – Video Demo).

5.5.3 Spotify API Audio Features endpoint deprecated

The endpoint that returns Spotify's audio features, as seen in Kaggle dataset in an earlier sprint, was announced as deprecated November 27, 2024 (Introducing Some Changes to Our Web API, 2024) (Figure 5-14). This meant no new developer apps would have access to the endpoint. This was likely done to prevent the data from being used for machine learning. Alternatives include large datasets of Spotify track data on Kaggle. The limitation of this is that there won't be any updates to this dataset, making it static as opposed to the endpoint being dynamic and frequently updated.

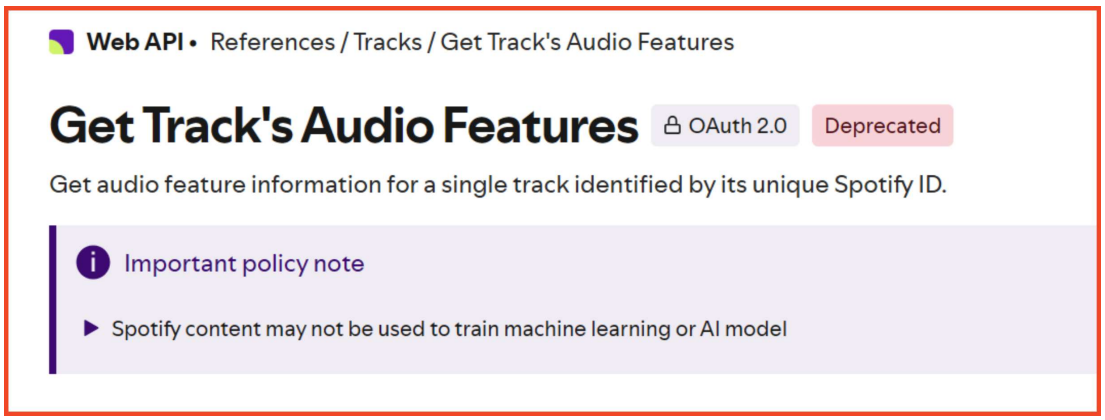

Figure 5-14 - Spotify track audio features endpoint reference

5.6 Sprint 3

5.6.1 Goals

- Node.js backend setup for Spotify token management
- Better manage Firebase and Spotify actions, services files
- Firebase Authentication Flutter integration
- Store user data and Spotify tokens in Firestore

5.6.2 Node.js

The backend was created with Node.js and the Express.js framework and integrated the Spotify auth flow. Using a browser to test, going to the '/login' route will start the Spotify authentication process. After the user has agreed to authorise the app to use their Spotify (through Spotify's own UI and Api) the user is redirected back to the app. This is done by setting the callback URL in the Spotify Developer Console.

Using a package called 'spotify-web-api-node' makes much of the process quite simple. It handles a lot of the code needed to generate the 'state' variable which is a random string used for security against cross-site request forgery. The package provides methods for storing the access and refresh tokens in an instance of the SpotifyWebAPI (Figure 5-15) that can be used across the app to make further API calls easily.

```
spotifyApi
  .authorizationCodeGrant(code)
  .then((data) => {
    const access_token = data.body["access_token"];
    const refresh_token = data.body["refresh_token"];
    const expires_in = data.body["expires_in"];

    // set access tokens to the api object
    spotifyApi.setAccessToken(access_token);
    spotifyApi.setRefreshToken(refresh_token);

    // upload the tokens to Firestore
    const dataUpload = {
      access_token: access_token,
      refresh_token: refresh_token,
      expires_in: expires_in,
    };
```

Figure 5-15 - Spotify Auth Flow code

Configuring the SpotifyWebApi (Figure 5-16) required the clientId, clientSecret and custom redirectUri from the Spotify developer dashboard. The permissions scopes also need to be defined.

```
const SpotifyWebApi = require("spotify-web-api-node");
require("dotenv").config();

const spotifyApi = new SpotifyWebApi({
  clientId: process.env.SPOTIFY_CLIENT_ID,
  clientSecret: process.env.SPOTIFY_CLIENT_SECRET,
  redirectUri: process.env.SPOTIFY_REDIRECT_URI,
});

const scopes = [
  "playlist-modify-public", // Allow creating public playlists
  "playlist-modify-private", // Allow creating private playlists
  "user-read-private", // Read user profile
  "user-read-email", // Read user email
  "user-top-read", // Read user's top tracks and artists
];
```

Figure 5-16 - Spotify Web Api configuration

The '/login' endpoint is where the user will start the authentication process for Spotify. This could be tested using a browser. The '/callback' route was more complicated as it needed to store the returned data. The issue that arose at this stage was that the access token data was being stored in

the SpotifyWebApi instance and was reset every time the server updated. This meant that whenever a code change was saved, the instance lost the access token data. To address this issue, the tokens would need to be stored in a database and retrieved after a server restart.

This was done by creating a SpotifyAuth middleware (Figure 5-17) that runs on every endpoint needing valid Spotify access tokens. It fetches the Spotify token data from the database and sets the spotifyApi instance. Ensuring the data is always correct but resulting in possible unnecessary database reads. This can be improved on in a later sprint. It first checks if the database is initialised and then queries the user collection with the userId, then setting the spotifyApi instance with the results.

```
try {
  const db = getFirestoreDb(); // Ensures Firebase is initialized first

  // if Firebase db isn't initialized yet, initialize it
  if (!db) {···
  }

  const result = await TokenService.getSpotifyTokenData(userId); // get the
  token data from Firestore

  const tokenData = result.data;

  // set token data on each request to ensure consistency
  spotifyApi.setAccessToken(tokenData.access_token);
  spotifyApi.setRefreshToken(tokenData.refresh_token);

  next();
```

Figure 5-17 - SpotifyAuth middleware

5.6.3 Flutter setup

A Flutter package to handle Spotify API calls on the frontend was initially used but the package was causing major errors with running the app. Troubleshooting resulted in discovering the package was not compatible with the version of Flutter and the Android SDK being used. Handling all Spotify API calls in the backend instead will result in more security and control so this doesn't pose a major issue.

Firestore Authentication was implemented with register and login functions, using the Firebase_auth package. Text controllers are used in the form and capture the users' entered values, then passed to the create user function.

An authenticate state was used in the AuthState page to check the authentication status of the user. If not authenticated, show login/register page, otherwise show the homepage. The Firebase_auth package provides methods that indicated the authentication state. Using a StreamBuilder widget (Figure 5-18), the app listens to changes in the FirebaseAuth instance and dynamically displays the appropriate page.

```
body: StreamBuilder(
  stream: FirebaseAuth.instance.authStateChanges(),
  builder: (context, snapshot) {
    // if user is logged in
    if (snapshot.hasData) {
      return HomePage();
    } else {
      return LoginOrRegister();
    }
```

Figure 5-18 - Authentication checking

5.7 Sprint 4

5.7.1 Goals

- Integrate Spotify Auth flow in Flutter
- Make Spotify API calls requiring OAuth 2.0, get playlists, get user information in backend
- Spotify token refresh
- Get and display users recent top tracks from Spotify

5.7.2 Spotify API endpoints

The app successfully received Spotify user data using browser to authenticate. Using the current spotifyApi instance and the getUserPlaylists() method (Figure 5-19) to get the data, sending the access token with the request. The spotifyAuthMiddleware ensures the access token data is valid.

```
// Example route to get user's playlists
router.get("/playlists", spotifyAuthMiddleware, (req, res) => {
  spotifyApi
    .getUserPlaylists()
    .then((data) => {
      res.json(data.body);
    })
    .catch((err) => {
      console.error("Error fetching playlists:", err);
      res.status(500).send(`Error fetching playlists: ${err.message}`);
    });
});
```

Figure 5-19 - Get users Spotify Playlists route

5.7.3 Frontend Spotify Auth Flow

Implementing the authentication flow in Flutter was more complicated because the user needs to be brought to a webpage that displays the Spotify authentication process. To do this, a Flutter package called 'flutter_web_auth_2' was used. This package opens an in-app browser to the defined URL, in this case the backend route that opens Spotify's authentication process. If successful, the defined callback URL is used to redirect the user back to the app (Figure 5-20). Initially, using deep links was explored as a solution but this became unnecessary after discovering the 'flutter_web_auth_2 package'.

```
final result = await FlutterWebAuth2.authenticate(
  url: '${ApiConstants.baseUrl}/spotify/login?token=$encodedUserToken',
  callbackUrlScheme: "spotifyauth",
);
```

Figure 5-20 - Flutter web auth

The 'callbackUrlScheme' defines where the browser should bring the user to on successful authentication. The value "spotifyauth" refers to the Flutter app. Defining the name of the app is done in the AndroidManifest.xml file (Figure 5-21).

```xml
<!-- Add separate CallbackActivity for flutter_web_auth -->
<activity
    android:name="com.linusu.flutter_web_auth_2.CallbackActivity"
    android:exported="true">
    <intent-filter android:label="flutter_web_auth_2">
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="spotifyauth" />
    </intent-filter>
</activity>
```

Figure 5-21 - Android Manifest defining and android scheme

5.7.4 Spotify token refresh

Refreshing the Spotify token requires the 'refresh_token' to be sent, returning a new 'access_token'. The spotifyApi package has a method for this. Firstly, the validity of the token needs to be checked. Tokens are valid for one hour so comparing the time of last update and the value of "expires_in", part of Spotify's token data returned, indicates if a refresh is required (Figure 5-22). This is part of the SpotifyAuth middleware.

```javascript
const isTokenExpired = (tokenData) => {
  if (!tokenData?.last_updated || !tokenData?.expires_in) {
    console.log("Token data missing last_updated or expires_in");
    return true;
  }

  // Convert Firestore timestamp to milliseconds
  const lastUpdated = tokenData.last_updated._seconds * 1000;
  const now = Date.now();
  const expiryTime = lastUpdated + tokenData.expires_in * 1000;

  // If token expires in less than 5 minutes, refresh early to avoid bad requests
  return now >= expiryTime - 300000; // 5 minutes buffer
};
```

Figure 5-22 - Spotify token expired check

5.7.5 User's top tracks

A new backend route, '/top' was created which gets the current user's top tracks with time range, limit and offset parameters being implemented (Figure 5-23). The relevant fields in the response were mapped to a new array and included in the response to the frontend.

```
const data = await spotifyApi.getMyTopTracks({
  time_range: "short_term",
  limit: 20,
  offset: 0,
});

const mappedData = data.body.items.map((track) => {
  return {
    id: track.id,
    artistName: track.artists[0].name,
    songName: track.name,
    albumName: track.album.name,
    albumArtworkUrl: track.album.images[0].url,
  };
});

res.json([...mappedData]);
```

Figure 5-23 - Get users top track endpoint

To receive the data back, it needed to be correctly typed so a model called 'ListeningHistory' was created. It included a factory method that converts the returned JSON to a DART object (Figure 5-24).

```
  factory ListeningHistoryItem.fromJson(Map<String, dynamic> json) {
    return ListeningHistoryItem(
      id: json['id'] ?? '',
      artistName: json['artistName'] ?? '',
      songName: json['songName'] ?? '',
      albumName: json['albumName'] ?? '',
      albumArtworkUrl: json['albumArtworkUrl'] ?? '',
    );
  }
}
```

Figure 5-24 - Listening History factory method

The FutureBuilder widget uses this model and a nested ListView widget (Figure 5-25) to render the data when returned from the server. The ListView renders each item of returned data using the ListeningHistoryCard custom widget.

```
return ListView.builder(
  padding: const EdgeInsets.all(16.0),
  itemCount: snapshot.data!.length,
  itemBuilder: (context, index) {
    final item = snapshot.data![index];
    // Create a card for each history item
    return ListeningHistoryCard(item: item);
  },
);
```

Figure 5-25 - Listening History ListView widget

5.8 Sprint 5

5.8.1 Goals

- Create/join session functionality
- Create basic playlist from session user listening history and save to Spotify
- Build core Flutter UI
- Db redesign - move listening history to user document instead of session document
- Bug: after login, loading indicator hanging,
- Bug: 403 Bad OAuth request from Spotify

5.8.2 Create/join session functionality

Creating a session uses the ‘getRecentHistory’ and ‘getUserProfile’ from the SpotifyServices file (Figure 5-26).

```
const [listeningHistory, userProfile] = await Promise.all([
  SpotifyService.getRecentHistory(),
  SpotifyService.getUserProfile(),
]);

const sessionData = {
  sessionName: "Test Session",
  users: {
    [req.session.uid]: {
      displayName: userProfile.display_name,
      product: userProfile.product,
      // listeningHistory: [...listeningHistory],
      isAdmin: true,
      joinedAt: new Date(),
    },
  },
};
```

Figure 5-26 - Creating session document

The new session is added to the sessions collection with the listening history being added to a sub collection. Adding a new document to the sub collection was straightforward but updating a field in an existing document required some research into Firestore on how to send the data. It needed to be added to the existing field data but initially it was overwriting the entire field. Using a transaction to update this field ensured no conflicts would occur if multiple users tried to update at the same time. This keeps the operation atomic and the database integrity intact. Adding this method to the Firebase Services class makes it reusable across the codebase.

The join functionality follows the same process except it requires the existing 'sessionId' to the new user data can be added.

5.8.3 Create basic playlist

To create a playlist from all users listening history, the listening history sub collection in the session doc is queried and stored in 'listeningHistoryDocs' (Figure 5-27).

```
// Flatten all listening histories into a single array
let allTracks = [];

allTracks = listeningHistoryDocs.map((user) => ({
  id: user.id,
  tracks: Object.values(user.tracks).slice(0, 10), // Get the first 10 tracks, convert object to array
}));

// Combine all tracks from each user into a single array
const justTracks = allTracks.reduce((acc, user) => {
  return acc.concat(user.tracks).splice(0, 20);
}, []);
```

Figure 5-27 - Getting all users listening history

This is then mapped over, taking only 10 tracks from each user. This is then flattened into a single array and stored a session sub collection called 'shadow_playlist'.

To save to Spotify, a new empty playlist needs to be created and then tracks added afterwards. This is easily done using the spotifyApi instance (Figure 5-28). The tracks array and title were hard coded to keep early implementation simple.

```
const playlist = await spotifyApi.createPlaylist("My playlist", {
  description: "My description",
  public: true,
});
const playlistId = playlist.body.id;
console.log("Created playlist with id:", playlist);

// add tracks to playlist, returns snapshot id for playlist
const addedTracks = await spotifyApi.addTracksToPlaylist(
  playlistId,
  tracks
);
```

Figure 5-28 - Create and add tracks to Spotify playlist

5.8.4 Core Flutter UI

Pages for all main actions were created. Create, join, waiting room and view live session pages. Navigation between these pages was implemented through buttons on a homepage.

5.8.5 Db redesign

The users listening history was moved from a sub collection in the session document to a sub collection in the user document. A larger database redesign was planned but after some testing it

was decided that implementing this change was outside of the projects scope and so the listening history change was reverted.

5.8.6 Login bug

This item was not gotten to as it was low priority and so was returned to the backlog.

5.8.7 403 response status

Spotify's documentation says: "Bad OAuth request (wrong consumer key, bad nonce, expired timestamp...). Unfortunately, re-authenticating the user won't help here."

The error occurred after authenticating a new Spotify test account. However, the previous test account worked still. The problem was related to the app being in development mode in Spotify API dashboard. This mode allows a max of 25 users, whose names and email addresses must be whitelisted to be granted access to the app (Figure 5-29). After adding the accounts in the Dashboard, the 403 error was resolved.

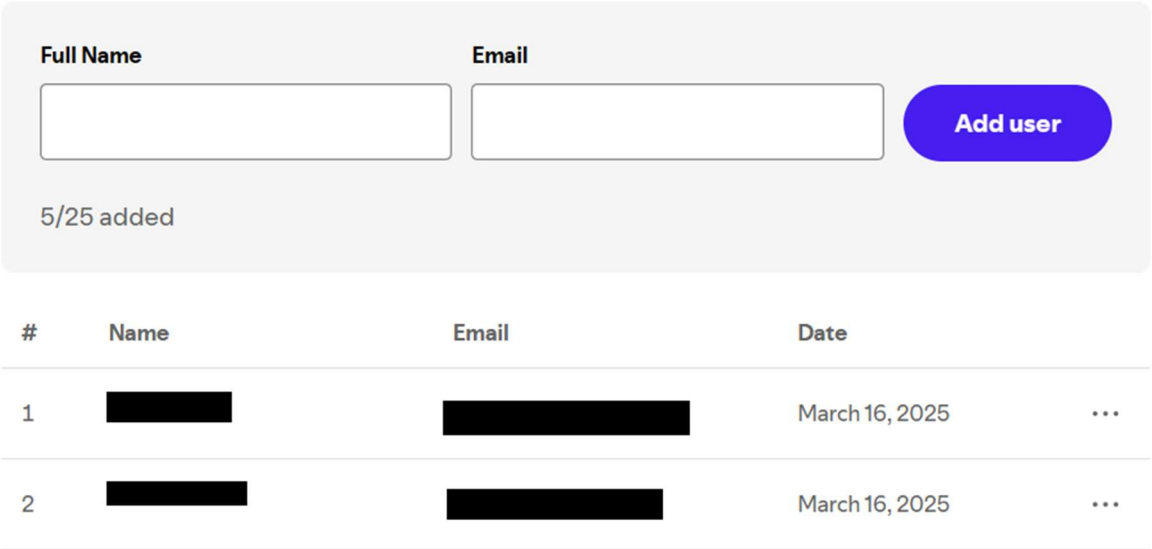

Figure 5-29 - Spotify API user management settings

5.9 Sprint 6

5.9.1 Goals

- QR code to join session
- Improve state management
- Voting functionality
- Bug: Spotify token refresh issue

5.9.2 State management

Hard coded values were being used to join and display sessions. To make the app dynamic, the state management package called Provider was used. Provider stored the session Id, allowing it to be access throughout the app. An instance of 'sessionState' is created in the apps main() method and then accessed elsewhere.

A 'sessionState' class was created to manage all session state related logic. Methods for creating and joining a session first call the main create/join methods from the PlaylistSessionServices class and then set the states with the returned data.

Only the host can start the session, other users need to be brought to the live session page when this happens. This is done by setting up a listener that monitors changes to the session status field in the database (Figure 5-30).

```
// Listen to the session changes in Firestore
_sessionStateSubscription = FirebaseFirestore.instance
    .collection('sessions')
    .doc(sessionId)
    .snapshots()
    .listen((snapshot) {
  if (snapshot.exists) {
    final data = snapshot.data() as Map<String, dynamic>;
    final status =
        data['status'] ?? 'waiting'; // Default to 'waiting' if not set

    if (status == 'active' && !_isActive) {
      // Session is now active
      setIsActive(true);
```

Figure 5-30 - Listener for session status

In the waiting room page, this status state is monitored for changes by using the 'didChangeDependencies()' method, which checks for changes while the page widget is mounted (Figure 5-31). Once the session is changed to 'active', the navigator sends the user to the live session page.

```dart
@override
void didChangeDependencies() {
  super.didChangeDependencies();

  // Check if session is active and needs to redirect
  final sessionState = Provider.of<SessionState>(context);
  sessionState.listenToSessionStatus(sessionState.sessionId);


  if (sessionState.isActive && !sessionState.isHost) {
    WidgetsBinding.instance.addPostFrameCallback((_) {
      Navigator.pushReplacementNamed(context, '/live-session');
    });
  }
}
```

Figure 5-31 - Waiting room didChangeDependencies()

5.9.3 QR code

The 'qr_flutter' package was used to generate a QR code and the 'mobile_scanner' package to scan and read the QR code. To generate the QR code, the QrImageView widget (Figure 5-32) requires the data to be used and then some UI information such as size and colour. It takes an 'errorStateBuilder' that displays on error.

```dart
QrImageView(
  data: sessionState.sessionId,
  // 'sample://open.my.app/#/join-session/${sessionState.sessionId}',
  version: QrVersions.auto,
  size: 200.0,
  backgroundColor: Colors.white,
  padding: EdgeInsets.all(10),
  errorStateBuilder: (cxt, err) {
    return Center(
      child: Text(
        "Uh oh! Something went wrong...",
        textAlign: TextAlign.center,
      ), // Text
    ); // Center
  },
), // QrImageView
```

Figure 5-32 - QR code generator widget

Scanning the QR code was more complicated. The setup was very simple to begin but there were issues with the debug console being spammed with messages when the camera was in use even

after the page had been left. This was caused by the camera controller lifecycle not being properly managed. The package provided good documentation which solved this problem.

When the camera detects a QR code, the camera is stopped and only started again if the user presses 'cancel' on the dialog box. This stops the camera from constantly scanning for a QR code. The camera also needs to be properly disposed of after the user leaves the page. Adding the camera dispose method to the parent widget 'dispose()' method (Figure 5-33) properly solved the excessive console messages and correctly managed the camera function.

```
@override
void dispose() {
  // dispose of the controller when the widget is removed
  cameraController.dispose();
  super.dispose();
}
```

Figure 5-33 - QR scanner dispose method

A physical device was needed to test the camera which introduced an issue with running the server on local host, which the physical device could not access. VSCode includes an in-built feature called Port Forwarding that allows for remote access to applications. Setting this up solved the problem in development but is not a long-term alternative to deploying and hosting the application.

5.9.4 Voting

Voting is handled by sending a request to the backend to be verified and then update the database, optimistically updating the UI and rolling back changes if the server responds with an error. The voting controller uses the 'VotingServices' class to cast an up or down vote, checking the vote type in the request. Up voting, for example, first checks if the user has already voted and what the vote type was, reverting that vote if necessary and/or updating the database with an initial vote (Figure 5-34).

```
const { hasVoted, voteType } = this.checkUserVote(
  playlist,
  trackId,
  userId
);

// Check if user has already voted
if (hasVoted) {
  if (voteType === "up") {
    // If already upvoted, remove downvote
    return await this.removeUpvote(playlistId, trackId, userId);
  }
  // If downvoted, remove downvote first to avoid up and down votes both
  if (voteType === "down") {
    await this.removeDownvote(playlistId, trackId, userId);
  }
}
```

Figure 5-34- If user has voted logic

Handling voting on the frontend was done by creating a 'VotingMixin' that contains the voting functionality. The thumbs up/down icon is optimistically updated while the total vote count text changes to a loading indicator until the server confirms the vote was successfully cast. If the vote fails, the UI reverts to the original vote state. To prevent the user from sending rapid API requests if hitting the vote icons quickly, a 'isVoteInProgress' flag is used (Figure 5-35).

```
if (_isVoteInProgress)
  return; // If a vote is already in progress, do nothing

_isVoteInProgress = true; // Set vote in progress flag at beginning

final originalIsUpVoted = isUpVoted;
final originalIsDownVoted = isDownVoted;
```

Figure 5-35 - Vote in progress code

The 'TrackCard' widget which displays each track in the playlist uses the 'VotingMixin', giving it access to the voting logic while maintaining separation of logic (Figure 5-36).

```
// Voting logic handled in mixin
class _TrackCardState extends State<TrackCard> with VotingMixin {
  final userId = FirebaseAuth.instance.currentUser?.uid;
```

Figure 5-36 - TrackCard using VotingMixin

A custom widget was created to display the vote icon, taking the states to update dynamically (Figure 5-37).

```
VoteIcon(
  isUpVote: true,
  isVoted: isUpVoted,
  isLoading: isUpVoteLoading,
  voteCount: widget.item.upVotes,
  onTap: () => handleVote(context, widget.item, "up"),
), // VoteIcon
```

Figure 5-37 - VoteIcon custom widget

There is a small bug in the vote icon. Sometimes it incorrectly indicates a vote was cast when a vote was removed. This was added to backlog for next sprint.

5.10 Sprint 7

5.10.1 Goals

- Unit and integrated testing
- Playlist track sorting on frontend – most/least votes
- Fix voting icon bug
- UI V2 Design
- Session images
- API deployment

5.10.2 Testing

Testing on the backend was done using the framework Jest. This involved learning how to mock dependencies. Mocking is used to predefine responses from functions and can be very useful when conducting tests that rely on external dependencies, like a database or an API. The responses from the Firestore database were mocked as well as those from Spotify. Unit tests were written for a select key features of the backend, including token refresh and adding a user to a session.

Manual integration testing was conducted on the frontend. Using an Android emulator, several tasks were carried out on the app. The expected and actual results of each step were noted. These tasks include the user creating an account and linking their Spotify, creating a playlist session, voting on playlist tracks and joining an existing session. The tests were all successful. The only note is that there is a small UI bug while logging in where the home screen is rendered briefly before being replaced by the Spotify connect page after a user registers. This bug has been added to the backlog. These tests are to discussed in detail in the Testing section.

5.10.3 Playlist sorting and icon bug

Sorting the tracks was accomplished by creating a function that uses the .sort() method on a list that compares each time and sorts by the criteria given. The total votes were calculated and then used to compare each track. The returned list sorted tracks by descending order, most votes to least votes. The only issue with this logic is where two tracks have the same vote count. There is no rule on which track should appear first. This leads to tracks reordering in an unexpected way. The solution would be to introduce a second sorting criteria. An option would be to add a timestamp field to each track and have the most recently voted track as the top amongst same total vote tracks. This improvement has been added to the backlog.

The voting icon bug has been resolved. The state of the icon was not being properly attached to the item in the List View. This was solved by adding the trackId as key to each track. Flutter was then able to reorder based on the key and not the location within the list, meaning the icons where being properly reordered with each track.

5.10.4 UI V2 Design

The second version of the UI design was created and partially implemented. The Spotify connect screen was completely redesigned, adding a new background image created by an AI image generator using the apps colour scheme and title letters. The dashboard was a new addition, combining what were previously multiple pages (Figure 5-38). The create and join buttons as well as past sessions are all displayed on this dashboard page. It looked quite bland without much colour so it was decided that adding images to playlist session would give it the needed colour.

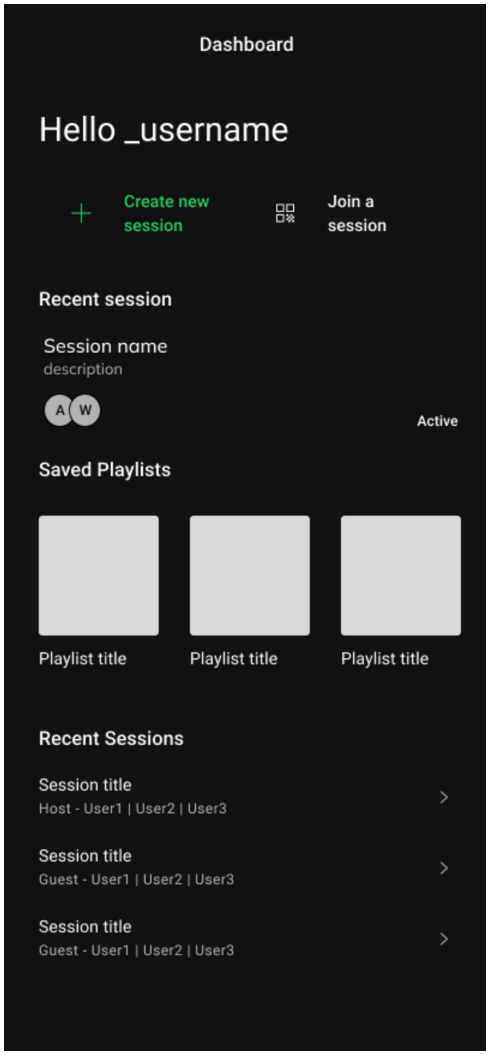

Figure 5-38 - Dashboard UI design

5.10.5 Session images

To support image upload during session creation, the Flutter frontend was first tackled. Adding the 'firebase_storage' package allowed for easy integration and access to Firebase Storage. Another package, 'image_picker', was added to handle the picker UI and access to the device's gallery. To manage this feature a new services class was created that would store information on the image upload process in state (Figure 5-39).

```dart
class SessionImageServices extends ChangeNotifier {
  File? _image;
  String? _downloadUrl;
  bool _isUploading = false;
  String? _error;

  // Getters
  File? get image => _image;
  String? get downloadUrl => _downloadUrl;
  bool get isUploading => _isUploading;
  String? get error => _error;

// define the image picker instance
  final ImagePicker _picker = ImagePicker();

  Future<void> pickAndUploadImage({required String sessionId}) async {
    try {
      // set initial states and notify listeners
      _error = null;
      _isUploading = true;
      notifyListeners();

      // pick image from gallery
      final picked = await _picker.pickImage(source: ImageSource.gallery);
      if (picked == null) {
        _isUploading = false;
        notifyListeners();
        return;
      }

```

Figure 5-39 - Session Image Services

After implementing the needed frontend logic, it was discovered that the options for using Firebase's Storage service had recently changed and now required an upgrade to the Blaze (Pay as You Go) account with mandatory billing details. This was discovered after receiving an error when trying to access Firebase Storage (Figure 5-40). Due to this, the feature was set aside and another option explored.

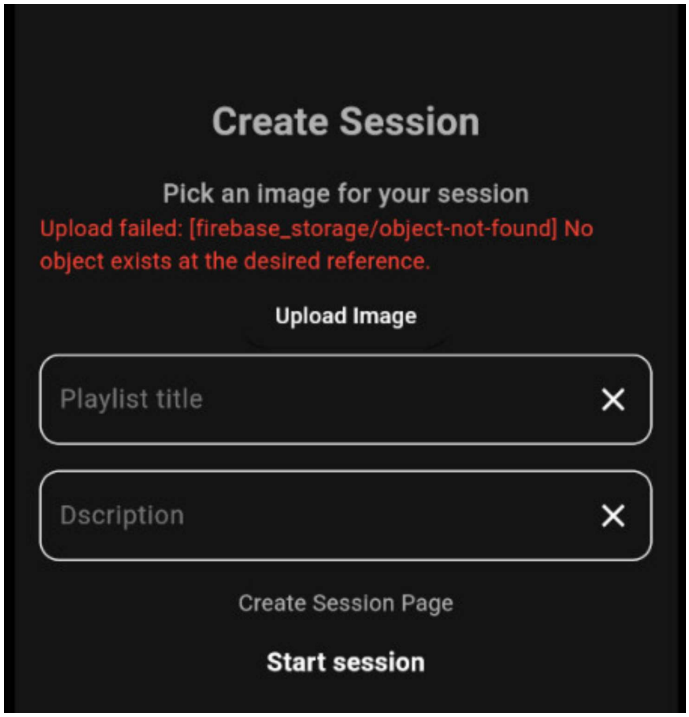

Figure 5-40 - Image upload, Firebase Storage error

The alternative was to add existing album artwork to each session document. To accomplish this the top voted track was selected from the session playlist and its image URL was added to the session document. This process is carried out only when the session changes from 'active' to 'ended', storing the final top voted track image.

5.10.6 User past sessions

Implementing the dashboard design required backend modifications. To get all sessions that a user was a part of a new 'sessions' field that was added to the user document in the database. When a user joins a session, the session Id is added to the 'sessions' array in the user document. This field is then referenced in order to retrieve all of the users' sessions. A new endpoint was created to handle the getting the users past sessions. These sessions are displayed with album artwork on the Dashboard page.

5.10.7 State management

Properly managing state was a significant issue during this sprint. A user state was added in order to track the user display name, is they were a new user and if they had connected Spotify. This would be used to dynamically display the Spotify connect page after a new user registers and again if an existing user somehow bypassed connecting to Spotify. These user states were being set in either registration or login pages and being read in the AuthPage. The AuthPage was checking for isSpotifyConnected in order to display the Connect page. This caused lots of problems because pages

were trying to set the state while the widget was being built, resulting in major errors. One solution was to wrap the 'setState' method in the 'addPostFrameCallback()' method inside the initState function which runs once when the widget is mounted (Figure 5-41 – addPostFrameCallback). This ensures that 'setState' isn't called until the widget is fully built.

```
@override
void initState() {
  super.initState();

  // initialize UserState
  _userState = Provider.of<UserState>(context, listen: false);

  // waits until the widget is fully built before calling _initializeUserState
  WidgetsBinding.instance.addPostFrameCallback((_) {
    _initializeUserState();
  });
}
```

Figure 5-41 – addPostFrameCallback

This solved part of the problem but there were still many issues with how the user state was being set so it was put on the backlog to focus on other tasks.

5.10.8 API Deployment

Deployment using Vercel was explored, and a test deployment was attempted. The logs revealed that there were errors related to the in-memory session storage that the API was using to handle authentication IDs in middleware. The solution for this would be to switch the session storage to something like Redis. This would require refactoring that is out of the scope of the project. Implementing Redis and deploying the API is an option for future development.

5.11 Sprint 8

5.11.1 Goals

- User state issues
- Auth page bug – logout not redirecting to Signin page
- Viewing an ended session
- Redesign remaining pages
- User testing

5.11.2 User State issues

It was decided that many of the User state variables and functions were over complicating the application and resulting in bad state management. This late in development, it was decided that the time required to fix this issue would be better spend on other tasks, so some of the User state functionality was removed.

The display name state was set during registration, taking the username from the registration form. The display name was not set during login however, resulting in an inconsistent UX. The users Spotify display name is used elsewhere in the app but is not used to set the User state displayName state. This results in another inconsistency in the UX. The display name should use either the chosen username from registration or the Spotify display name. This is an issue for future development.

The Spotify connection check function would have been good to integrate fully, with the backend endpoint already existing. This would be a good feature for future development as it's mostly implemented in the frontend already. An alternative approach was taken regarding how the Spotify connect page is shown to users.

5.11.3 Auth page bug – logout and Spotify connect

The Auth page used conditional logic from the User state that was no longer being correctly set, resulting in unexpected behaviour when the user clicked the logout button. This was resolved by simplifying the conditionals that checked if the Firebase auth state had changed. If the auth state indicated a user was logged in, a Consumer widget was used to monitor the UserState which indicated if the user was new or not (Figure 5-42). The 'isNewUser' flag was set to true during registration and ensures that newly registered users will always be shown the Spotify connect page. The flag is set to false on successful authorisation from Spotify.

```
// listens to auth state - if user is logged in or not
body: StreamBuilder(
  stream: FirebaseAuth.instance.authStateChanges(),
  builder: (context, snapshot) {
    print('Auth state changed: hasData=${snapshot.hasData}');
    print('Auth state user: ${FirebaseAuth.instance.currentUser?.uid}');
    // Show login/register if not authenticated
    if (!snapshot.hasData) {
      print('No auth data - should show login page');
      return LoginOrRegister();
    } else {
      // Check if user is new and needs to connect to Spotify

      return Consumer<UserState>(
        builder: (context, userState, _) {
          // If this is a new registration, show the Spotify connect page
          if (userState.isNewUser) {
            return ConnectSpotifyPage();
          }
          // Otherwise show the home page
          return HomePage();
        },
      ); // Consumer
    }
```

Figure 5-42 - Auth page logic

This setup, as well as manually redirecting the user to the '/auth' page after logout, created the expected behaviour around login/logout and the Spotify Connect page. The only remaining issue is that the Home page is briefly shown to a newly registered user before the Spotify Connect page. This as a bug in previous sprints but was not successfully resolved.

5.11.4 Viewing an ended session

A new endpoint was created in the backend that retrieved all of the sessions from the sessionsId array in the user document and returned them. The frontend then filters the sessions and stored those with status 'ended' in a 'pastSessions' array state. Another set of state variables was created to view a past session. Initially, these variables were used with the existing session variables and a 'isViewingMode' flag (Figure 5-43). This setup dynamically set the sessionId and related variables to whatever session had been clicked. The issue with this was that the active session details, viewable on the dashboard, would be switched to the past session that had been clicked on. The 'isViewingMode' flag also wasn't being reset after navigating out of the past session view.

```
// Variables for viewing past sessions - these won't affect the active session
bool _isViewingMode = false;
String _viewingSessionId = '';
String _viewingPlaylistId = '';
String _viewingSessionName = '';
String _viewingSessionDescription = '';
String _viewingHostDisplayName = '';
String _viewingImageUrl = '';
```

Figure 5-43 - isViewingMode session state variables

To resolve this, the viewing mode variables were completely separated from the active session variables. This solved all issues and works as expected.

An error occurred when trying to retrieve past session when the users 'sessionIds' array contained over 30 sessionIds (Figure 5-44). Firebase has a limit of 30 comparisons when using the 'IN' query.. This means that a user can't have more than 30 past sessions. To handle this, sessions should be deleted either after a set period or when the user approaches 30 past sessions. This is an important feature for future development. If not implemented, the user will not be able to see any past sessions when they exceed 30.

```
d\node_modules\@google-cloud\firestore\build\src\telemetry\enabled-trace
-util.js:102:28) {
  code: 3,
  details: "'IN' supports up to 30 comparison values.",
  metadata: Metadata {
    internalRepr: Map(1) { 'x-debug-tracking-id' => [Array] },
    options: {}
  }
}
Error getting user's sessions: Error: Firebase get documents by IN query
 failed: 3 INVALID_ARGUMENT: 'IN' supports up to 30 comparison values.
    at FirebaseService.getDocumentsByInQuery (C:\dev\projects\mood_gen_b
ackend\src\services\firebase.services.js:135:13)
```

Figure 5-44 - Firebase error on user sessionIds array query

5.11.5 Redesign remaining pages

The join and waiting room pages were redesigned to match the V2 UI design from the last sprint. The waiting room page was updated to dynamically display the users Spotify names as they joined (Figure 5-45). The waiting room page was made less cluttered, with clear instructions from the users and host. The users are told the host will start the session.

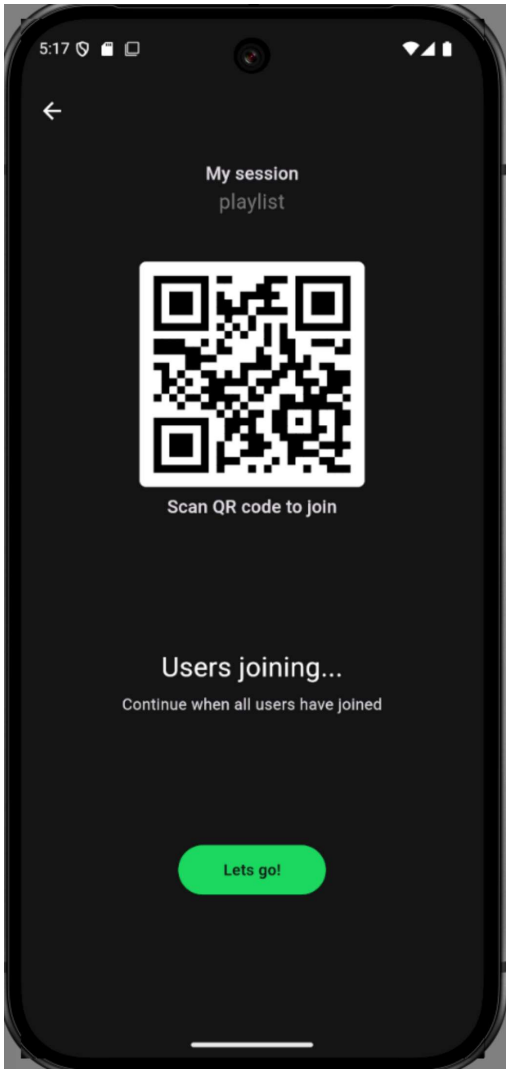

Figure 5-45 - Join page redesign

All pages were checked on a smaller device to be sure there was no issues with pixel overflow. The create and join buttons needed to be made dynamic in size due to this and so were wrapped in a FittenBox widget which allowed them to resize to the given space.

5.11.6 User testing

User testing was conducted with a physical Android phone and an emulator on the development PC. A series of tasks were defined for the user and notes were made during the testing. The user didn't have many issues overall and reported the app was straightforward to use. There was a discussion about the features after the testing where the user suggested improvements and noted features that were not clear. The voting feature was not well explained in the app so it was suggested that adding a short tutorial or tooltips would be good. These suggestions were all noted and will be considered for future development. The user testing is discussed in more detail in the following section.

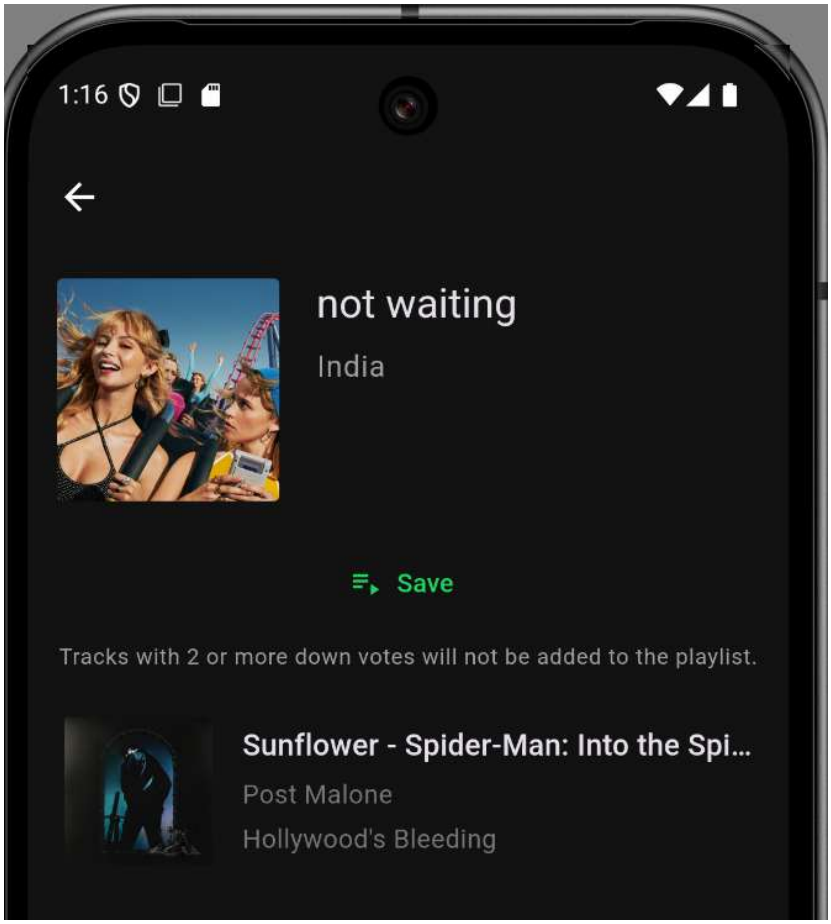

Figure 5-46 - Live session helper text added

A simple UI update was made to include text explaining that a track with two or more down votes would not be added to the playlist (Figure 5-46). This is a simple solution that will inform users but should be more thoroughly implemented using the previous suggestions.

5.12 Conclusion

The implementation phase of the project used Scrum methodology to build the functional requirements incrementally. Each sprint focused on developing one or more features at a time, starting with early lyric analysis using OpenAI and progressing to working with Node.js and Flutter as the project developed. The early sprints were used to explore the direction of the project and investigate its feasibility. During this time the fundamental elements were steadily worked on, namely the Spotify Web API integration. Once the project refocused on collaborative playlists, the pace picked up and each sprint made strong progress towards meeting the requirements.

Challenges like Spotify token management and Flutter state handling were addressed and improved on frequently throughout the sprints, reflecting the iterative and flexible development process.

6 Testing

6.1 Introduction

This section will describe the testing conducted on the application. The testing is split into two categories, functional and user testing.

Functional testing deals with the code and programmatical aspect of the app, rather than the user interaction. There are multiple types of functional testing such as unit and integration testing. The goal of these tests is to see if the application gives the expected output to the input.

User testing focuses on how the end user interacts with the application. This is important because even if functional testing was successful, the user may still experience difficulties completing a task with the application if the UI is not well designed.

6.2 Functional Testing

The functional tests were split into unit testing and integration testing. Unit testing describes testing a small, isolated function. This is the opposite of integration testing which takes a larger section of code that is made up of many functions and tests if the functions work together as expected. This kind of testing is very important as it simulates full sections of the user flow. An example of an integration test is creating a new playlist session. This requires multiple steps and functions that deal with database interactions and internal logic. An example of a unit test would be testing the function that handles track shuffling. Functional testing is concerned only with the output and if it that matches what was expected.

Testing categories:

- Backend - Automated integration testing
- Frontend - Manual integration testing

6.2.1 Backend – Automated testing

The automated tests were written using Jest, a JavaScript testing framework. The application has two major dependencies, the Firestore database and the Spotify Web Api. During testing, these dependencies were mocked in order to keep the tests completely independent and fast. Mocking means that rather than calling the dependencies, the expected results were predefined. This follows good practice because the dependencies themselves are robust and don't need to be directly tested. It can be safely assumed that they will produce reliable results. An example of mocking a dependency would look like defining the information that should be returned from a Firestore

database query, given the test input. This predefined database response is then used in the rest of the test.

Structing a test starts by describing what is being tested and then asserting what the result should be. Within the test there are three steps, arrange, act and assert. Arranging is defining any test or mock data, like mocking a database response or defining a test user Id. Act is calling the function being tested. Finally, assert is describing the expected outcomes. The test passes if all the assertions are correct.

Core backend functionality was tested using integration tests due to most functions using external dependencies, making unit testing less applicable. All automated tests written and conducted are described in the following sections. They test for success as well as for error handling.

6.2.1.1 Token Refresh testing

Test No	Description of test case	Input	Expected Output	Actual Output	Result	Comment
1	Spotify token refresh - successful	userId	An object containing a new access token, expiry time and updated timestamp	An object containing a new access token, expiry time and updated timestamp	Pass	Firebase and Spotify responses were mocked.
2	Spotify token refresh – no refresh token error	userId	Throw error of "No refresh token available"	Throw error of "No refresh token available"	Pass	Mocked. (sans refresh token from db.)
3	Spotify token refresh – no user document found	Invalid userId	Throw error of "Token refresh	Throw error of "Token	Pass	Error message should

			failed - No refresh token available"	refresh failed - No refresh token available"		reflect Firebase doc not found.
4	Spotify token refresh – Spotify Api error	userId	Throw error of "Token refresh failed - Spotify API Error (401): Invalid refresh token"	Throw error of "Token refresh failed - Spotify API Error (401): Invalid refresh token"	Pass	spotifyApi error response mocked using example error from package docs. Firebase mocked.

6.2.1.2 Adding a user to a session as host

Test No	Description of test case	Input	Expected Output	Actual Output	Result	Comment
1	Add a user to a session as host - successful	sessionId, userId and isHost flag	An object with correct user and session data	An object with correct user and session data	Pass	Firebase and a UserService method (Spotify dependency) was mocked.
2	Add a user to a session as host – User Service	sessionId, userId and	Throw error of "Failed to	Throw error of "Failed to	Pass	Mocked.

	method unsuccessful	isHost flag	add user to session: Failed to get user data and history"	add user to session: Failed to get user data and history"		
3	Add a user to a session as host – Firebase addDocument service unsuccessful	sessionId, userId and isHost flag	Throw error of "Failed to add user to session: Firebase add document failed"	Throw error of "Failed to add user to session: Firebase add document failed"	Pass	Mocked.

6.2.2 Frontend

Manual integration testing was performed by interacting directly with the app via an Android emulator and a physical Android device. This method was chosen because it most closely reflects how the user will interact with the app via it's frontend. This tests the integration between the frontend input and the results received from the backend. It will also test the external dependencies (Spotify and Firebase) mentioned earlier, which is not possible with the previous tests alone.

6.2.2.1 Register user and connect Spotify account

Test description:

Register a new user and connect their Spotify account when prompted – success test case.

Pre-requisites:

The user has an existing Spotify account with login credentials. The users Spotify account email has been whitelisted in the Spotify Web API dashboard.

Step	Action	Input	Expected Output	Actual Output	Comment
1	Launch app	Navigate	Login screen	Login screen	
2	Go to the Register page	Click on 'Register Here'	Registration screen loads	Registration screen loads	
3	Register as new user	Valid user details with matching passwords, click 'Register'	Loading indicator and then redirect to Connect Spotify page	Loading indicator, flash of homepage and then redirect to Connect Spotify page	Flash of homepage caused by AuthPage state management
4	Proceed to Spotify authorisation process	Click 'Lets go'	Spotify login screen	Spotify login screen	Login handled by Spotify
5	Login to Spotify account	Spotify login details, click 'Login'	Spotify – Authorise App page	Spotify – Authorise App page	Authorise handled by Spotify
6	Authorise the app	Click 'Authorise'	Loading indicator and then app Homepage. On screen	Loading indicator and then app Homepage. On screen	

			success message	success message	

6.2.2.2 Create, start and end a playlist session

Test description:

A logged in user creates a new session with a name and description. They start the session and then end it after the playlist is displayed.

Pre-requisites:

The user has already connected their Spotify account to the app.

Step	Action	Input	Expected Output	Actual Output	Comment
1	Go to 'Create session' page from Homepage	Click 'Create session'	Create session screen	Create session screen	
2	Create a session	Input a session name and description and click 'Create session'	Few seconds loading, then session waiting room with QR code	Few seconds loading, then session waiting room with QR code	
3	Start the session	Click 'Lets go!'	Live session page with tracks from users recent listening. 'End session'	Live session page with tracks from users recent listening. 'End session'	End session button only available to session creator.

			button available.	button available.	
4	End session	Click 'End session'	Confirmation dialog popup, 'Cancel' and 'End session' options	Confirmation dialog popup, 'Cancel' and 'End session' options	
5	Confirm end session	Click 'End session' in dialog	Loading with 'Ending session' text, redirect to Homepage	Loading with 'Ending session' text, redirect to Homepage	
6	Authorise the app	Click 'Authorise'	Loading indicator and then app Homepage. On screen success message	Loading indicator and then app Homepage. On screen success message	

6.2.2.3 Vote on tracks and save playlist to Spotify

Test description:

A user will vote up/down on multiple tracks in a live playlist session, resulting in the tracks being reordered. User will then save the playlist to their Spotify account. Tracks with 2 or more down votes will not appear in the saved playlist on Spotify.

Pre-requisites:

The user has connected their Spotify account and is already in a live playlist session with another user. The 2nd user has already down voted one track.

Step	Action	Input	Expected Output	Actual Output	Comment
1	Vote up on a track	Click the thumbs up icon next to a track	Icon is solid colour, and the number below is 1	Icon will turn filled and the number below increments	
2	Vote same track down	Click the thumbs down icon on the same track	Down icon is outlined, and number below is 0, thumbs down is solid colour and number below is 1	Down icon is outlined, and number below is 0, thumbs down is solid colour and number below is 1	User cannot maintain both up and down votes on same track, either or.
3	Vote a track down	Click thumbs down icon on track	Track moves to bottom of the list, icon and number update accordingly.	Track moves to bottom of the list, icon and number update accordingly.	Track list is dynamically ordered by vote count
4	Vote tracks down which has existing down vote	Click thumbs down icon on track	Icon remains filled and below number is 2	Icon remains filled and below number is 2	
5	Save playlist to Spotify	Click ‘Save’	Success message in	Success message in	The saved playlist will

			green banner at bottom of screen	green banner at bottom of screen	not contain the track which had 2 down votes.

6.2.2.4 Join an existing session

Test description:

A user will join a session created by another user.

Pre-requisites:

The user has connected their Spotify account and the QR code to join a live playlist session is available.

Step	Action	Input	Expected Output	Actual Output	Comment
1	Open the 'Join session' page	Click 'Join session' button	Join session page with active camera preview	Join session page with active camera preview	
2	Scan QR code	Place the QR code in view of the active camera	Dialog window with session info and 'Join' button	Dialog window with session info and 'Join' button	User cannot maintain both up and down votes on same track, either or.

3	Join session	Click 'Join' button in dialog	Loading and then Waiting room page with hosts username	Loading and then Waiting room page with hosts username	Guest user has no button to start session, host only.
4	Wait until host starts session	(Host actives the session on own device)	Live session view of playlist	Live session view of playlist	Guest has no button to end session, host only.

6.2.3 Discussion of Functional Testing Results

All functional requirements were tested and shown to be working as expected. The core features of the application consisting of creating/joining a session, voting on track and saving the playlist to Spotify, all worked as expected.

The manual integrated testing of the frontend confirmed that the backend was working as expected with the given user input. The external dependencies were also proven to be working as expected. Testing these dependencies was difficult in automated testing so thoroughly testing them manually was necessary as the app heavily relies on them. Automated integration testing provided a strong result for core backend functionality such as token refreshing, without which the functional requirements would all fail. After the functional testing was successfully completed, the application was ready to move to user testing.

6.3 User Testing

User testing was conducted with a physical phone in debug mode running the Flutter app. An Android emulator was also running on the development PC. The user was given three main tasks to complete and was asked to comment on any difficulties or confusions during the process.

6.3.1 Task 1 – Register and connect Spotify account

The user reported the registration process and Spotify connection page were clear and easily understood. While filling in the form, the user scrolled through the page and then clicked outside of the input fields to hide the keyboard and click continue. The Spotify connect flow was completed with no additional comments.

Findings: Seeing the user scroll the page while the keyboard was visible and then tap outside of the input fields confirms that the scrolling functionality is an expected part of the app and including it enhanced the UX. The only comments on this task from the user were that it made sense, was simple and straightforward.

6.3.2 Task 2 – Join a session, vote on a track and save the playlist to Spotify

Task setup: The testers PC was running an Android emulator with a new session created, displaying the QR code to join.

The user started on the homepage and was asked to join a session. They clicked the join button and scanned the QR code off the Android emulator running on the testers PC. When the QR code was captured and the dialog box appeared, they clicked join.

They were then asked to vote on some tracks and to make sure at least one track had two down votes. The tester voted down a track using the emulator. The user voted a few tracks up and then scrolled through the list to find a down voted track.

They were asked to navigate back to the homepage and then re-enter the session. They navigated back successfully but became confused looking for the session. The session was named ‘user testing’ and the user was not sure that meant the session but ultimately clicked it and navigated back to the live session page.

They were then asked to save the playlist to their Spotify. They clicked the save button and saw the green confirmation message that the save was successful. The session was then ended by the host account on the Android emulator and the user was brought back to the home screen.

Findings: The user noted that it was very clear which button was to join and that they would need to scan a QR, due to the QR code icon. The user did not think automatically that the down voted tracks would be at the bottom and so was slowly scrolling through the list looking for one. Navigating away and back to the live session page was confusing because the user wasn't sure what 'user testing' meant on the home screen. This was an oversight by the tester. The session should have been more appropriately named to replicate a real-world scenario.

6.3.3 Task 3 – Create a session, vote on tracks and end the session

The user was asked to create a new session. They clicked on the create button and entered a session title and description into the form and clicked continue. The tester told them to assume another person had joined and the session was ready to start. The user clicked continue and proceeded to vote on tracks again. They were asked to end the session. They clicked 'end' and confirmed in the popup dialog.

Findings: The user did not appear to have any difficulties with any part of this task.

6.3.4 Insights

Overall, the user was able to easily complete the tasks asked and used the app as expected. This user is a regular Spotify user and so is familiar with how that app works.

They had a few remarks during testing, saying registration process was simple and the QR code icon was helpful (Figure 6-1). At the end of testing, they were asked if they had any thoughts on the app or features, they would enjoy being added to the app.

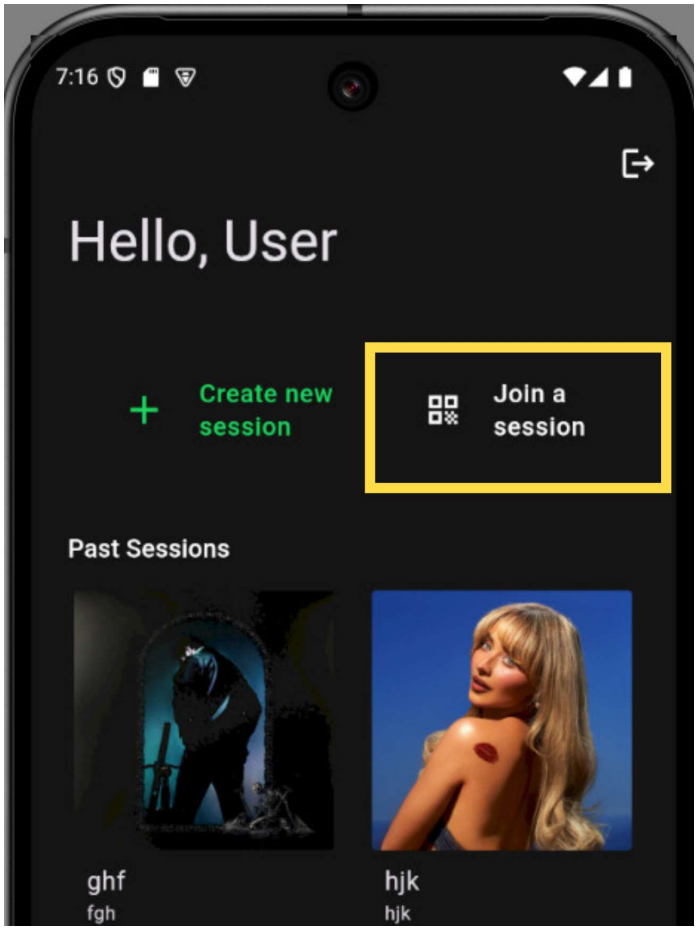

Figure 6-1 - QR code icon on homepage

The main suggestion was to add tooltips that explained what certain buttons did or to have a tutorial for new users. They said they were not sure what the voting feature did and had not noticed that the tracks were being reordered as they voted. The tester explained the voting feature and noted that there was currently no indication for the user that a track with -2 votes would not be included in the final playlist (Figure 6-2).

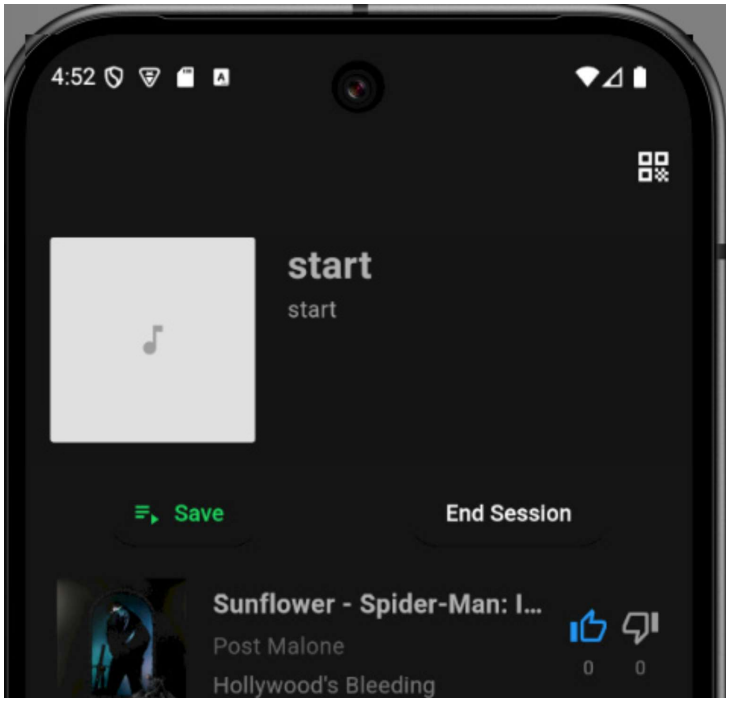

Figure 6-2 - Live session page lacking voting feature explanation

Their other main suggestion was about how users join a session. They said they would enjoy a feature where they could invite their Spotify friends from within the app. A list of people they'd connected with on Spotify and the option to either invite them to join the app, or if already a user, to join the session. They gave the example of how WhatsApp works when a phone contact does not have WhatsApp so a message prompting them to join is sent.

A smaller suggestion was to add a screen after the session ends where some small but funny statistic is given. Their example was the song with the most/least votes, saying "You guys really didn't like this song: song x". They also suggested adding a notification when a song got 2 down votes and would be removed. Saying "someone might want to fight for their song".

6.4 Conclusion

The testing results show that all functional requirements were met, and that the non-functional requirement of live voting was also met. The automated unit testing was successful, as was the manual integrated testing. The functional testing results provide good evidence that the app has met the requirements and is stable, with not major bugs or issues.

Following the functional testing, the user testing results paint a similar picture. The user found the app intuitive to use but would have liked more guidance about what the voting feature did. They used the app as expected and confirmed that implementing the tap to close keyboard feature was a

necessary part of typical UX design. The test user provided valuable feedback on how the app could be improved. This feedback will be used when proposing future development ideas.

7 Project Management

7.1 Introduction

The project was managed using sprints and a kanban board to track tasks. The project was split into distinct phases, initial research, prototype development, external API integration, backend API development and frontend development.

At the beginning of each sprint, the goals were defined and then tasks added to the Kanban board. As each task was completed, it was moved from in-progress to completed. If a task couldn't be completed, it was returned to the backlog. As each sprint progressed, certain tasks were split into smaller tasks.

To manage the uncertainty of the early phase, development focused on setting up the core Spotify Web API OAuth 2.0 flow. This was the backbone of the application.

Early milestones kept development on track, like the proposal presentation. Working towards a prototype to demo at this stage made the sprint planning at this phase clearer. Creating a detailed timeline to achieve a functional prototype helped greatly (Figure 7-1).

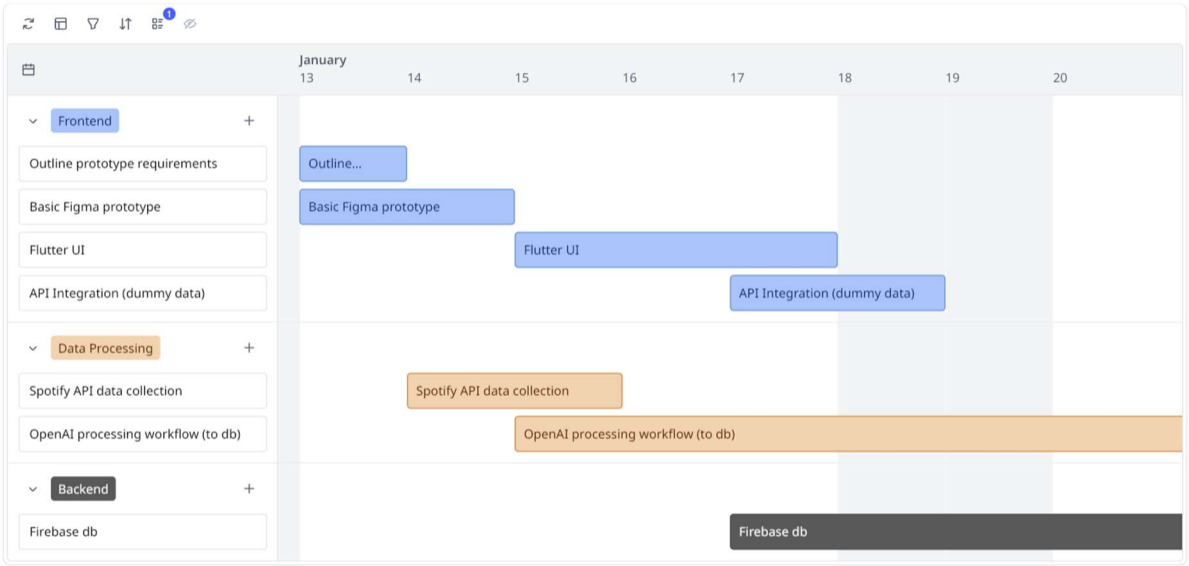

Figure 7-1 - Proposal prototype timeline

7.2 Project Phases

7.2.1 Proposal

The initial proposal focused on functionality that used mood and lyrical themes to filter songs, using the OpenAI API for lyric analysis and attribute creation. It was discovered at this time that the Spotify Web API endpoint that played a central role, had recently been deprecated. This posed a large issue going forward. Similar data could be found in existing datasets but would render the application completely static. This was discussed with the project supervisor and 2nd reader during the proposal phase. Recommendations were made to expand the project, which were considered for the subsequent sprint.

7.2.2 Requirements

Researching similar apps was the first step in gathering requirements. This research informed the survey that was created to gauge users' interactions with existing features and their interest in the purposed features.

The survey results were then used to create personas, which in turn informed the functional requirements and use cases. Each step in this requirements phase informed the next, building upon itself and creating a very clear image of what the app would need to achieve.

Managing each phase separately meant only moving on once enough information was gathered. Waiting for survey results was the only issue during this phase. If the survey didn't get enough responses, it would be more difficult to gauge the results' accuracy. The survey was given some extra time to gather responses.

7.2.3 Design

The program design underwent many iterations as the application requirements were refined. Due to the uncertainty of the project focus at the beginning of the project, the database design went through many changes. During each sprint where new features were added, it would become clear that the database needed to be adjusted to accommodate these changes. For example, when getting the users Spotify listening history it was decided that a subcollection would be the best way to go instead of embedding the data.

During the main development of the frontend, a missing route or feature needed in the backend would be uncovered. Due to the backend used a strong SoC principle, many of the needed functions already existed and merely needed to be put together to achieve the required new route. A good

example of this was the ‘updateSessionStatus’ function in the ‘playlist_session.controller’. All of the service methods to accomplish the task existed in the Firebase Services class, so they were easily assembled.

Similar issues would arise during many sprints, the need for a new or slightly altered function in the backend to support a frontend feature. Managing this was made easy the focused sprints and good practices early on when creating the base service classes for the backend.

The UI design went through two major iterations. The first version laid out the needed features and UI interactions whereas the second was a lot more refined and combined previously separate pages into one. This was only possible because the functionality was complete and only needed to be well organised visually. Designing in iterations allowed for more flexibility. The first design couldn’t account for issues that would arise during development, but it did give a good guideline when developing the initial interface pages.

There are design features, such as swiping left or right on tracks to vote, which were not implemented due to time constraints and unexpected complexity. Not every part of the design was implemented but will be considered during future development.

7.2.4 Implementation

Early in the project, discovering the core API endpoint from Spotify was deprecated introduced the need for a major change in direction. During this phase the project felt stalled as it took some time before there was a concrete goal again. This resulted in some development time being lost, however this time was used for further research and scoping of the project.

As the project grew and changed, so did the requirements for the database. Deciding between embedded data versus separate collections was a frequent issue. Separating the data would provide greater scalability for the future but required significant code refactoring. Determining if these improvements were within the project scope required frequent consideration.

Implementing state management became a large aspect of the frontend. This introduced new bugs and at times caused the application to completely crash. Learning to manage state correctly in Flutter required a lot of research into widget lifecycles.

A major aspect of the project was deciding how and when the frontend should have direct access to the database. The layered system architecture pattern conflicts with the presentation layer and the data layer coming in direct contact. It was decided that the voting features need for live updates

outweigh the benefits of strictly adhering to the architecture pattern. The security was maintained by only allowing the presentation layer to read the database directly and not write to it.

7.2.5 Testing

The backend was tested first as it was the most complete and stable. Having focused on the backend first meant that all major features were working reliably by the time the frontend was being developed. Therefore, testing was made relatively straightforward.

Once the frontend functional requirements were met, it could be tested manually. Although the UI design was not finalised, functional testing was still able to be conducted. The UI design did not change much of the functionality related to the functional requirements.

User testing confirmed that the apps UI was clear, and no major errors occurred. Each task given to the test user was completed without issue. The user gave feedback after testing, noting that an explanation of the voting feature would have made the UX better as it was not immediately clear what the voting did.

7.3 Project Management Tools

7.3.1 Kanban

A Kanban is a digital board with usually three columns, backlog, in-progress and completed. It provides a way of visualising work and allows for tasks can be easily moved between columns.

It helped greatly in keeping track of backlogged tasks and tasks that were at the time low priority but still important to later stages of development. Reviewing all the completed tasks was also motivating. Towards the end of the project a fourth column was added for discarded tasks. Rather than simply deleting tasks that were outside of the project scope, they were moved to the discarded column (Figure 7-2). This helped keep ideas for future iterations of the application documented.

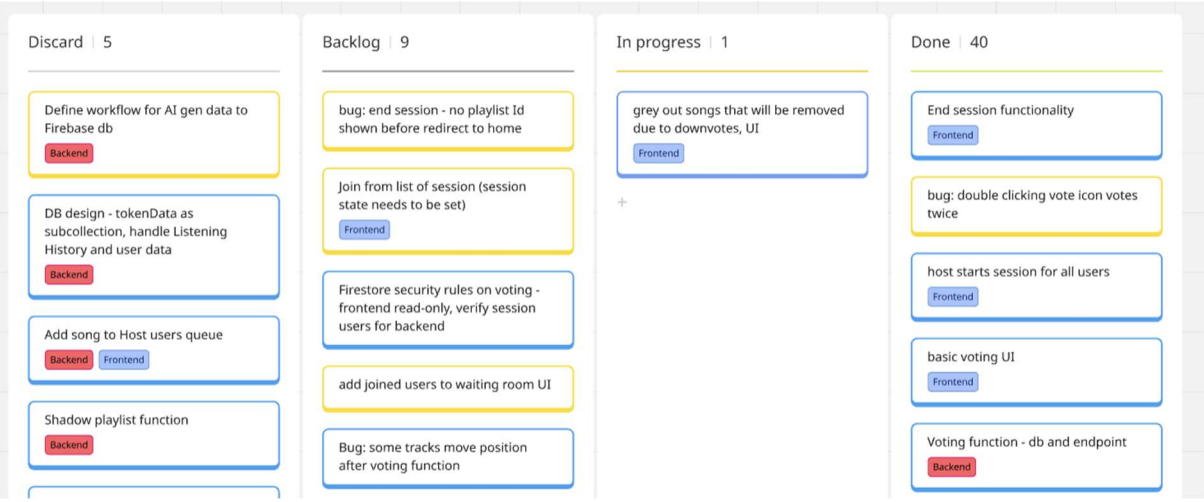

Figure 7-2 - Miro Kanban board

7.3.2 GitHub

GitHub is cloud-based platform for developers to store and manage their code. Using git version control with GitHub makes it very easy and simple to backup and track progress in a project. It also has features for teams working on a project together, managing pull requests and code reviews.

In this project it was used to manage code throughout development. Code changes were committed with comments describing the changes. Branches were used when implementing new features. This meant that the main branch was kept in a stable and functional state, while allowing development on a different feature which might break the code.

GitHub and git in general made it easy to track all my code changes and review previous work. Viewing the state of the entire repository at the time of any commit was very useful while documenting the implementation process. It also gave the option to revert to a previous commit. This was not required during development but had there been a significant issue that was traced back to already committed work, it would be simply to revert the codebase to an earlier commit.

7.3.3 Journal

Keeping notes on the development process created a very clear timeline and record of all research and work. This was especially useful for later documenting the implementation phase.

Each sprint was given its own frame on the project Miro board (Figure 7-3). All journal notes and research was documented here.

Figure 7-3 - Miro sprint 3 board

Including screenshots of code, whether it be successful feature implementation or bugs, helped with the documentation process as well as dealing with backlogged items. Bugs that were in the backlog for multiple sprints had been documented thoroughly and so going back to them was made a lot easier. The Miro board offers a very good visual representation of the journey the project took. Going from the initial focus to doing more research and eventually landing on the final application concept. All research into API's, technologies and UI design is documented.

7.4 Reflection

7.4.1 Your views on the project

The project started off quite differently and with a much smaller goal but ultimately became a very complete and functional application. There were some weeks where not much progress could be made because of the uncertain direction but once the project concept was redefined, development

picked up pace considerably. With a strong direction in mind, the application started to form rapidly, and new aspects of the chosen technologies were explored in more depth.

Working with Flutter required learning a lot about widget lifecycles and good state management. These were both areas of difficulty in the project, with similar bugs being added to the backlog each sprint. Over time, these issues reduced as debugging in Flutter became more intuitive.

The non-functional requirements of the project were not all met, largely due to underestimating the complexity of working with new technologies and external APIs. However, the progress made was not insignificant. The final application accomplished all the functional and some of the non-functional requirements with no major errors.

From a team perspective, the project took time to start but once it had direction there was very steady progress made. There were many discussions about additional features, the feasibility of them and what features should be prioritised. These discussions helped with getting the functional requirements met while still looking for areas for improvement.

7.4.2 Completing a large software development project

The most significant thing learned during the project is the importance of structuring data in a way that aligns with the requirements. An issue that appeared frequently was the database structure needed to be changed to support a feature. It was not clear what data structure would be needed for some of the non-functional requirements, such as voting, and this resulted in adjusting the existing structure. Optimally, the entire applications database would have been modelled to support all requirements in the beginning, but this wasn't realistic due to the timeline. Thoroughly thought-out database design is integral to development; this had to be learned through trial and error.

The second significant point learned was the benefits of following good practice principles and design architecture patterns, particularly SoC and Layered System architecture. Having a good understanding of how and most importantly why to use these principles and patterns has been invaluable and will be of great use in future work. Learning about system architecture has opened up the world of program design patterns and how they influence development. Going forward, more effort will be put into learning how to use these patterns and principles effectively. The benefits of following these patterns and principle in this project, albeit not as strictly as possible, is already very apparent and was rewarding to see pay off as development progressed.

7.4.3 Working with a supervisor

Having weekly supervisor meetings was invaluable as provided another layer of structure and accountability to the project. It offered a space to discuss issues and receive help on prioritising work. The early stages of the project were uncertain but discussing these concerns with the supervisor provided some direction and encouragement.

7.4.4 Technical skills

Through working extensively with Flutter and Dart, writing code in a type-safe language has become significantly more intuitive. Previous experience with using React Native with TypeScript had proved quite difficult and frustrating because the concept of type safety was still new. Coming to understand the data types used in Dart took some time as they are sometimes named differently to JavaScript data types. Working with a type-safe language has also shown the benefits when compared to regular JavaScript. The issue of confusing errors due to incorrect data types was far less of an issue during this project than in previous ones. Developing with Dart will make it far easier to choose type-safe languages like TypeScript in the future.

Developing with the Flutter framework was also a challenge due to its structure being vastly different to React.js but became quite enjoyable to work with. Understanding the widget lifecycle and how that affects state management was a hurdle that came up many times. Once the widget lifecycle became clearer, it was much more straightforward to track down and resolve bugs and errors. Dealing with state management is a universal skill when it comes to web development so all the progress made during this project will be a great asset to future work.

7.5 Conclusion

The project management approach for this application focused heavily on Scrum methodology, Kanban boards and maintaining a progress journal on Miro. This kept the project organised and progress easily trackable. As the projects focus evolved over time, it was clear that the early phases of research and then implementation had paid off as they had laid a solid foundation that could accommodate the shift in focus. With early sprints dedicated to Spotify token management it wasn't too difficult to shift the project direction after a key Spotify Api endpoint was discovered to be deprecated. Through discussions with the project supervisor, the application was redefined and work continued steadily.

Overall, the consistent task tracking and sprint planning provided the needed structure to complete the application while handling scope change and challenges along that way.

8 Conclusion

The aim of the project was to develop an Android application that allowed users to create a collaborative playlist that blended all users current listening preferences, dealing with the common issue song selection in a group.

The application was built using Flutter as the frontend framework to develop the Android application. The backend API was built using Node.js with Express as a framework. These technologies created the basis for the application, but the data needed was from Spotify. The Spotify Web Api was integrated with the Express server, dealing with token management and data processing. Firebase was the last piece of the tech stack and was responsible for both the NoSQL database and the authentication service. Firebase was integrated mainly in the backend, but the Flutter app was given access to read certain parts of the database directly.

The academic research conducted for this project focused heavily on the initial application concept of playlist generation powered by lyric analysis. As the project ultimately took a different direction, this research was less relevant to the final application. It does, however, still have relevance when looking at the future development opportunities for the project. Integrating a more sophisticated algorithm to sort the users' songs could involve using recommender systems.

The system design relied on a Layered Architecture pattern to ensure good security and followed good practices like Soc to keep code easily maintained and readable. With these foundations, the entire application was built modularly, making it simple to add or adjust functions. The UI design focused on emulating the visual look of Spotify while keeping the user flow simple and intuitive.

The implementation phase followed Scrum methodology, working in sprints and using a Kanban board to manage the backlog and to-do tasks.

Automated tests were written for the Express server to test the most important functions, while manual testing was conducted on the frontend. This consisted of going through the user flows and ensuring all external dependencies worked as expected with the application. User testing was conducting by setting a user a set of tasks to complete and analysing where they had difficulties or got lost. The results of the user testing informed recommendations for future improvements to the UI.

Despite the early challenges with deprecated endpoints and project scope, the project achieved all the functional requirements and multiple non-functional ones. The final product provides a solid application that accomplishes the goal of providing a collaborative playlist function, with lots of opportunities for future expansion.

The key skills developed were managing application state, working with Flutters widget structure and lifecycle and following Soc principles. State management and SoC are two skills that will transfer to many other areas and provide great value during development

8.1 Future development

There were some non-functional requirements not met that could still be very valuable for this project. Future development options will be discussed here, pulling from non-functional requirements, user testing results and features from the initial project concept regarding lyric analysis.

8.1.1 Past session management

Due to Firebase having a hard limit of 30 elements for an 'IN' query, a user cannot store more than 30 past sessions. The solution is either to start deleting past sessions after a set period or delete when the sessionsId array approaches 30 Ids. This is quite a limiting solution, but user research would be needed to decide if a user would ever need or want access to over 30 past sessions.

8.1.2 Spotify queue

Future development could include a Spotify queueing option as an alternative to saving the playlist and then playing it. This feature was outside of the scope of the project but was researched in an early sprint. The queue feature would allow users to start playing the music directly from the application. It would also allow them to reorder tracks live, making the app more interactive. This feature exists in the app Partify.io, which was researched as part of the requirement gathering phase.

8.1.3 Recommender systems and algorithms

Improving and advancing the algorithm used to create the playlist from user listening history could greatly enhance the value of the application. Recommendation systems were researched in an earlier phase and found to be very prominent and valuable in the context of music selection. The app currently pulls the top 10-15 tracks from a user's listening history from the past six weeks. The total number of tracks returned from Spotify is hundreds. Using an algorithm to determine the most relevant tracks could improve the results of the final playlist.

8.1.4 UI improvements

UI improvements such as a swipe left/right to vote and visual indications for a track falling below the cutoff threshold would provide a more complete UX. The swipe to vote feature was considered

during the UI design phase but was a non-functional requirement that was outside of the project scope.

8.1.5 Tutorials and tooltips

Pulling from the user testing feedback, a very important feature to add would be tooltips to explain the voting process, among other features. The test user reported being confused about the purpose of this feature and suggested either a tutorial or popup messages to explain it.

8.1.6 Invite Spotify friends

The test user also suggested integrating Spotify friends as a way of inviting people to join. This feature would include a page that displayed the users' friends on Spotify and would indicate if they were already registered with the app. The user could directly invite friends from within the app, sending a push notification to them. This feature would require getting a user's friend list from Spotify and cross-referencing the database to check if that Spotify user is already registered with the app. It would require more Spotify data be saved in the 'user' document. It would also require implementing push notifications on the invite receiver's device.

9 References

Apps | Spotify for Developers. (n.d.). https://developer.spotify.com/documentation/web-api/concepts/apps

Design & Branding Guidelines | Spotify for Developers. (n.d.). https://developer.spotify.com/documentation/design

Gomez, P., & Danuser, B. (2007). Relationships between musical structure and psychophysiological measures of emotion. Emotion, 7(2), 377.

IFPI GLOBAL MUSIC REPORT 2025. (n.d). https://globalmusicreport.ifpi.org/

Introducing some changes to our Web API. (2024, November 27). Spotify for Developers. https://developer.spotify.com/blog/2024-11-27-changes-to-the-web-api

Krugmann, J. O., & Hartmann, J. (2024). Sentiment Analysis in the Age of Generative AI. *Customer Needs and Solutions*, *11*(1), 3.

KRUMHANSL, C. L. (1997). An Exploratory Study of Musical Emotions and Psychophysiology. Canadian Journal of Experimental Psychology, 51, 4-336.

Mori, K., & Iwanaga, M. (2014). Pleasure generated by sadness: Effect of sad lyrics on the emotions induced by happy music. Psychology of Music, 42(5), 643-652.

Pond, N., & Leavens, D. (2024). Comparing effects of sad melody versus sad lyrics on mood. Psychology of Music, 52(2), 217-230.

Quota modes | Spotify for Developers. (n.d.). https://developer.spotify.com/documentation/web-api/concepts/quota-modes

Thakur, A., & Konde, A. (2021). Fundamentals of neural networks. *International Journal for Research in Applied Science and Engineering Technology*, *9*(VIII), 407-426.

Thorat, P. B., Goudar, R. M., & Barve, S. (2015). Survey on collaborative filtering, content-based filtering and hybrid recommendation system. International Journal of Computer Applications, 110(4), 31-36.

Vaswani, A. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*.

Wang, Y. (2023). Unravelling the Future of Recommender Systems Recent Advances and Emerging Possibilities. *J Robot Auto Res*, *4*(2), 387-391.

Zhang, W., Deng, Y., Liu, B., Pan, S. J., & Bing, L. (2023). Sentiment analysis in the era of large language models: A reality check. *arXiv preprint arXiv:2305.15005*.

10 Appendices

Appendix A – Spotify Playlist Preferences Survey

https://forms.office.com/e/396Ak52X8u

Appendix B – Project Miro Board

https://miro.com/app/board/uXjVLI76N7M=/?share_link_id=965555503311

Appendix C – Kaggle Spotify Dataset Sample

A sample of the Spotify data on individual tracks. Attributes include acousticness, danceability, energy, instrumentalness, liveness, loudness, speechiness, temp, valence and popularity.

acousticn	danceabili	energy	instrumentalness	liveness	loudness	speechine	tempo	valence	popularity
0.502	0.504	0.386	1.53E-05	0.0961	-10.976	0.0308	192.004	0.281	82
0.0483	0.604	0.428	0	0.126	-8.441	0.0255	110.259	0.292	79
0.137	0.596	0.563	0	0.302	-7.362	0.0269	97.073	0.481	80

Taylor Swift Spotify Dataset - https://www.kaggle.com/datasets/jarredpriester/taylor-swift-spotify-dataset?select=taylor_swift_spotify.csv

Appendix D – Sprint 2 Flutter Prototype – Video Demo

https://youtu.be/kU8jPYbMbas