PodcastScrobbler

The development of a mobile application for tracking songs in a podcast

By Johnny Morrison-Howe

Project Unit: PJE30

Supervisor: Dr Dalin Zhou

May 2023

Acknowledgements

I would like to thank my supervisor, Dalin Zhou, for his help and support throughout this project. I'd also like to thank my friends, namely Amber Turner-Brightman for their guidance in writing the report, and Emily Hudson for her invaluable advice and camaraderie in software development.

Abstract

When listening to music podcasts, it is not possible to upload metadata for tracks within to music tracking and discovery services like Last.fm - rather only metadata for the podcast itself. In this project I designed, implemented, and tested an Android application to perform this task.

I used Kotlin to build an audio playing service, with Jetpack Compose on top of that for the User Interface, where I also adhered to Google Material Design guidelines. The application has capabilities for obtaining, parsing, and storing podcasts, as well as playing episodes and showing the current track to the user via a persistent notification. I found that the application parses and manages all podcast feeds tested, with tracklists retrieved from descriptions for 5 out of the 6 podcasts tested.

It was planned to include Last.fm integration so that track metadata could be uploaded to the website, however this was not completed in time. Future work would involve adding this functionality and other features like favouriting and sorting of podcasts to bring it in line with other standard podcast applications.

Table of Contents

Acknowledgements 3
Abstract 4
Table of Contents 5
Table of Figures 8
Legal, Ethical & Social Issues 8
Introduction 10
Literature Review 11
  Popularity of Music Tracking 11
  Music Podcasts 11
  Podcast players 12
    Spotify 12
    Google Podcasts 12
    Apple Podcasts 12
  Identifying a gap 13
  Solutions that get close to bridging the gap 13
  Conclusion 14
Requirements 15
  Project brief 15
  Requirements table 15
User Interface Design 17
  Browsing all podcasts (main screen) 17
  Player 17
  Adding a podcast 18
  Browsing episodes 18
  Episode Details & Tracklist 19
User Flow 19
  Adding a podcast 20
  Playing an episode 20
Components 21
  Overview 21
  Activity 21
    PodcastScrobbler Composable 21
  Service 22
  Data Classes 22

Podcast 22
Episode 22
Track 22
Utilities 23
Podcast Manager 23
RSS To Class 23
Description to Tracks 23
Component Diagram 24
Implementation 25
Tooling 25
Kotlin 25
Android Studio 25
Target Device 25
Repository 25
Data Classes 25
Parsing an RSS Feed 26
Issues with XML 26
Iteration station 26
Parsing the description 26
Regular expressions 27
Persistent storage 28
Persistence is key 29
MainActivity and PodcastScrobbler 29
Remember 30
Scaffold 30
Navigation 30
Browsing podcasts 31
Adding a podcast 31
Browsing Episodes 31
Episode Details 32
Playing Media 34
Media3 34
Creating a service 34
Serialization struggles 34
Posting a notification and starting playback 34
Keeping track of the current track 35

- Playback Controls ... 37
  - Binding the service ... 37
  - Controls Composable ... 37
- Utilising Material Design ... 38
  - Colour Schemes ... 38
  - Dynamic Colour ... 38
- Implementation conclusion ... 38
  - Development experience ... 38
  - Effort Distribution ... 39

Testing ... 40
- RSS Feed Parsing ... 40
- Podcast storage ... 41
- Parsing descriptions ... 41
- Playing audio ... 42
- Uploading metadata ... 43
- Against requirements ... 44

Project Management ... 46
- Methodology ... 46
- Risks ... 47
- Evaluating Project Management ... 47

Conclusion ... 48
- Summary ... 48
- Future Work ... 48
  - Last.fm integration ... 48
  - Parsing tracklists with times ... 48
  - Enhanced playback controls ... 49
  - Lower priority requirements ... 49

References ... 50

Table of Figures

Figure 1 11
Figure 2 12
Figure 3 12
Figure 4 12
Figure 5 13
Figure 6 17
Figure 7 18
Figure 8 18
Figure 9 19
Figure 10 20
Figure 11 20
Figure 12 21
Figure 13 22
Figure 14 24
Figure 15 26
Figure 16 26
Figure 17 27
Figure 18 28
Figure 19 29
Figure 21 30
Figure 20 30
Figure 22 31
Figure 23 32
Figure 24 32
Figure 25 33
Figure 26 35
Figure 27 36
Figure 28 37
Figure 29 37
Figure 30 37
Figure 31 38
Figure 32 40
Figure 33 42
Figure 34 42
Figure 35 42
Figure 36 46
Figure 37 49

Legal, Ethical & Social Issues

Although I don't believe there are any professional or social issues, I will need to take security into account. Recording song metadata using Last.fm's API requires an account, therefore I will need to store the username and a private key for accessing the API. I plan only to store this information on the device however, so I would not need to deal with any data protection laws. Any end user other than myself would have a separate

agreement with Last.fm for their own account on creation and when logging in, however I do not plan to conduct user research for this project.

Introduction

For as long as I can remember, one of my favourite genres of music has been progressive house and dance. Tracks in this genre are all a similar length, pacing and speed, therefore they go great mixed together. One of the most accessible forms of this short of going to a nightclub is the podcast, typically hosted by an artist or label promoting tracks; this of course means there's a tracklist in case you like one, but how do you know which one you're listening to right now? Counting up the list, copying the track name and artist, and pasting it into a streaming service to see if it's the right one often takes a few tries, but what if it were as easy as glancing at the playback notification?

Extrapolating on this idea, what if you could see a history of tracks you've listened to this way, alongside recommendations for other related tracks, shared with your friends?

Although the latter proposition is taken care of using music tracking and discovery services like Last.fm, there is currently no solution to the first problem, one that I've had ever since I started listening to progressive house sets in the form of podcasts. To solve both issues, I would need to create an application that:

- Manages podcasts like any other podcast application, i.e. processing a podcast feed and allowing the user to browse their podcasts
- Plays podcasts, continuing to play in the background after the application is closed
- Interprets a podcast episode's description and shows the user the currently playing track
- Uploads metadata for the currently playing track to Last.fm

Implementing a system that performs these tasks would make it far more convenient for people in my situation to track what they're listening to, and could also both help bring those who listen to progressive house into the world of podcasts, and those who listen to podcasts into the world of automatic music tracking and discovery. The solution would both play podcasts and "scrobble" them(explained in the next section), so I have named the project *PodcastScrobbler*.

Literature Review

Popularity of music tracking

An application like *Podcast Scrobbler* may be useful, as there is clear demand shown for tracking habits in other areas. Spotify's Wrapped is often the first such application that comes to one's mind – when released initially, *"the internet descended into a frenzy of users screenshotting and sharing their streaming statistics all over social media"* (Adenuga, 2022, p. 3). There is an older, format-agnostic service however, which uses a technique called Scrobbling. Coined by Audioscrobbler (BBC News, 2003), Scrobbling is the act of automatically tracking music played on a system and storing it (Antonelli, 2023).

Last.fm is a service created almost alongside Audioscrobbler, for the purpose of using this data to make recommendations of new music to users and make the data sharable in the form of profiles. **Error! Reference source not found.** shows a screenshot of Last.fm, displaying my three most recently played tracks. Although less so now, it proved to be a popular service, having in 2006 won Best Community Music Site at the BT Digital Music Awards (TechRadar, 2006). Last.fm has received a rejuvenation recently, with a Netherlands-based developer building a Discord bot for Last.fm with over 400,000 registered users (Krastrenakes, 2022). It is worth noting that alternatives to Last.fm exist which use the same scrobbling software, like Libre.fm.

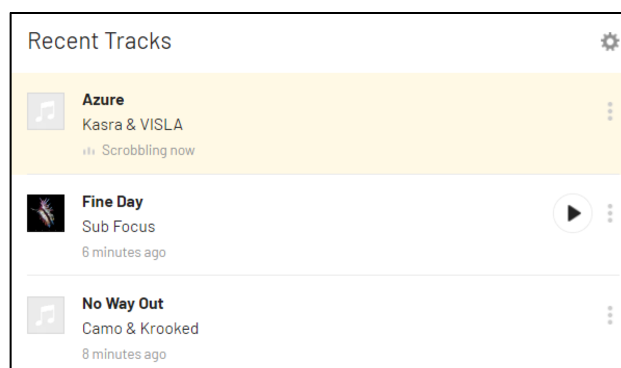

Figure 1

Music podcasts

A podcast is an episodic series of audio files, spanning topics from news to fictional drama to music. The format as it is today was standardised in 2000, by attaching audio files to RSS feeds (Louis, 2000). The term *podcast* was defined in 2004 by Ben Hammersley, relating to the iPod device they were associated with (Hammersley, 2004).

Podcasting has been increasingly popular since its inception; Whereas in 2005 the term *podcast* returned "over 61 million hits" on Google (Berry, 2006, p. 144), the term now returns over 5 billion. There are over 2 million podcasts on Apple Podcasts as of 2023 (Lewis, 2023).

A particular genre of podcast is the music podcast, involving music being played, occasionally with commentary, like a radio show. Tracks are often mixed by the host in the form of a DJ Set – for example, the *Mind Over Matter* Podcast by *Embliss* (Brandwijk, 2022).

Podcast players

The most common applications for listening to podcasts in the US are Spotify, Apply Podcasts and Google Podcasts (Götting, 2022). All have the ability to subscribe to a podcast feed, browse podcasts and episodes, display information like date released and a description, and play episodes.

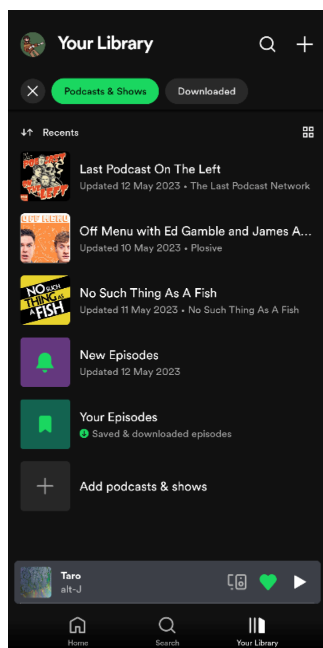

Figure 2

Spotify

Spotify's user interface for the library is uncluttered and gets straight to the point[1], with a list of podcasts(*Figure 2*), and a list of episodes for each item on the list. Menu entries display saved as well as new episodes, and there is a persistent bar at the bottom of the screen for playback controls.

Google Podcasts

Google Podcasts takes a similar approach, however the initial Library screen contains a list of menus – Subscriptions, Queue, Downloads and History, rather than showing these within the list of podcasts. The subscriptions screen (*Figure 3*) uses a grid rather than a list, increasing the prominence of cover art and displaying more entries per screen, at the cost of information like the latest episode date or the publisher. It also has a bottom bar displaying the current podcast.

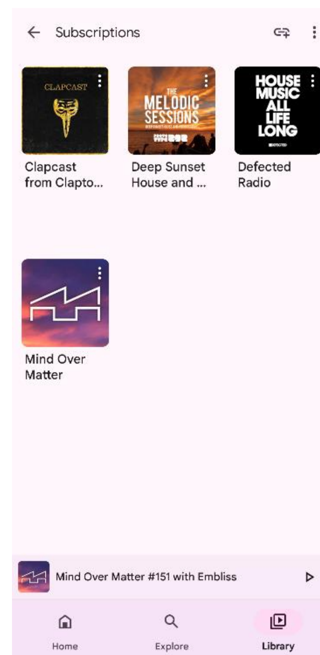

Figure 3

Being an official offering from the creator of Android, it complies with its design framework, Material Design. My current wallpaper has an overall pink tone, and as a result Google Podcasts adopts a pink colour scheme.

Figure 4

Apple Podcasts

Apple Podcasts displays both a list of different menus as well as a grid of the most recently updated episodes in the same screen(*Figure 4*). The cover art is larger than that of Google Podcasts and shows the type of podcast underneath the title. Like the two above applications, a persistent bottom bar is shown for playback.

[1] The home page is a more contested topic (Madaan, 2023), (Perry, 2023), but this does not cater to podcasts alone and so is less in the scope of this project.

Identifying a gap

Music podcasts can be scrobbled, however only by title of the podcast – not the tracks within. For users who repeat-listen to podcasts for the tracks within, listening data is effectively lost or mis-represented. However, tracks are often listed in the description i.e. in the case of *Mind Over Matter*.

Solutions that get close to bridging the gap

On many music services, it is possible to play a 'radio' of sorts, where tracks are played algorithmically based on a track, artist, or genre (Kostek, 2018). It is possible to scrobble these using standard scrobbling applications, and the individual tracks' metadata can be gathered and either shared or used for recommendations. As tracks are only scrobbled when played at least halfway (Last.fm, n.d.), played tracks in a 'radio' are only scrobbled if the user likes them enough to keep listening rather than skip them.

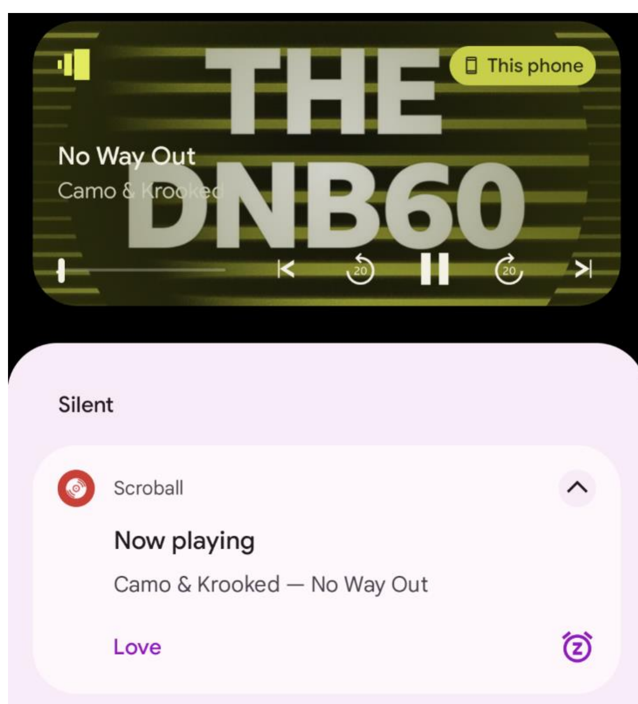

Figure 5

In 2018, the BBC announced that their BBC iPlayer Radio service would be discontinued in favour of BBC Sounds (Taylor-Watt, 2018). When radio programmes with music are played, the media API with the respective platform is provided with the current track, as opposed to the name of the radio show. This effectively provides a scrobbling service reading the media API with the current track.

Figure 5 shows this in action – BBC Sounds is posting a media notification to Android, and Scroball (a separate scrobbling application (Peter Josling, 2020)), is reading the Title and Artist of the current track. Although this is a solution to the problem, it only works with radio shows from the BBC, rather than all podcasts.

When researching user interfaces for podcast applications, I noticed that all music podcasts on Spotify I searched for were presented as artists and albums(from Spotify as a music service), rather than podcasts and episodes. Proton Radio and other labels have introduced an alternative way of publishing DJ sets through music streaming services (Wohlstadter, 2023). A DJ set is sliced into individual tracks which are played with gapless playback. As the tracks are labelled individually, they appear as the standalone track to a scrobbling application. Although the method works, only DJ sets by artists significant enough to be signed to a label can be published this way. This has the advantage for the DJ and artists whose tracks feature in a podcast, as they earn revenue as if a track had been played whereas they would not have done through a standard podcast.

Conclusion

After reading related articles, it is clear there is a gap in the field for a service that can scrobble tracks from podcasts, especially those that haven't been presented in a format that makes this already possible (i.e. extensive back catalogues). I have yet to find literature showing demand for this exact application (minus a lone Reddit comment (c210344, 2016)), however building such an application may induce demand for it as the currently less-related groups of those either listening to podcasts or using music tracking and discovery services could certainly be brought together by the application I propose.

Requirements

Project brief

To fill the gap in products, an application is needed which can obtain, manage, and play podcasts. It must then be able to parse a tracklist within a description, read or estimate when each track is played, and display the current track to the user. It must also be able to allow a user to sign into Last.fm or a similar service, and upload track metadata for the current track when it is played. These core features are marked as high priority and are grouped by their major IDs.

Other requirements would bring it more in line with features other podcast(or similar, i.e. BBC Sounds) applications offer, like the ability to save a podcast for offline play, organise podcasts in some way, or a particular episode. These are less important to the functionality of the app, but as they would improve the user experience, I have still included them albeit with a lower priority.

In the requirements table below, major requirements 1, 2 and 4 are implemented already in standard podcast applications. Requirement 5 already exists in audio players that implement Last.fm scrobbling. This application aims to bridge the gap between requirement 5 and requirement 1, 2 and 4 through the implementation of requirement 3, which is entirely novel.

Requirements table

ID	Requirement Description	Criteria	Priority
1.0	The application must be able to parse an RSS feed	A URL to an RSS feed is provided and serialized to some kind of Kotlin object	High
1.1	The application must be able to store podcasts	Information contained in an RSS feed can be stored and retrieved	High
1.2	The application must be able to update a podcast with new episodes and notify the user	When a new episode is published, a notification is served within 1 hour and the stored podcast is updated	Medium
2.0	The application must be able to present stored podcasts and episodes to the user	Stored podcasts and episodes are viewable in the User Interface	High
2.1	Episode descriptions and details must be visible	Episode descriptions are viewable in the UI	High
2.2	All other details of a podcast must be visible	All other details of a podcast are viewable in the UI	Medium
2.3	Podcasts can be sorted, i.e. by title, most recent episode date, length	Podcasts can be sorted ascending or descending by parameters in the User Interface	Low
2.4	Podcasts can be favourited	Podcast objects have an additional “favourited” field, with a separate view in the UI	Low
3.0	Episode descriptions can be parsed, with Track objects created and viewable	Episode descriptions are parsed either when asked to by the User	High

		or when played. Tracks are viewable within Episode Details UI	
3.1	Track timestamps are interpolated from any tracklists detected	Podcast track timestamps are interpolated and viewable in the UI	High
4.0	Audio for an episode must be able to be played	Episodes can be played in the episode screen, and keep playing in the background outside of it	High
4.1	Tracks in a podcast must be displayed when they are played	When playing a podcast, if its description was successfully parsed, track metadata must be shown to the user either when it is played or predicted to be played if interpolated	High
4.2	Tracks must have controls to adjust playback	There are controls shown when playing an episode, for example: play/pause, skip forward/back	High
5.0	A user must be able to log in to Last.fm	A log in page is shown to the user, and they can log in. Their account details are shown in the UI.	High
5.1	When played, a track's metadata must be uploaded to Last.fm or similar and associated with the user	When a track is played at least halfway (Last.fm, 2023), details must be uploaded.	High

User Interface Design

Material Design 3 is the current recommended design language for Android development (Android, n.d.). As such, although I used the design-language-agnostic Balsamiq Wireframes (Balsamiq, 2023), I adhered to the use of Material Design components throughout my User Interface mock-ups. I designed four main screens:

Browsing all podcasts (main screen)

Figure 6

When viewing podcasts, I created a simple list which will hold a vertical layout of Cards[2]. The Top App Bar[3] contains the name of the application, along with Trailing Icons for secondary functions like removing and sorting podcasts.

A Floating Action Button[4] provides access to the main operation of this screen (other than browsing podcasts) – adding a new podcast from an RSS feed.

Player

Bottom Sheets[5] are typically used for data and controls that must persist throughout the app – all three podcast applications I reviewed contained bottom bars for playback. I have persisted a bottom sheet everywhere in application when an episode is playing, containing the album art, episode title, the currently playing (or estimated) track metadata, along with controls like play/pause and a progress bar. Although not included in the mock-ups, it would also be useful to display an icon representing the scrobble status of a track (i.e. if it has been played for long enough to have the metadata uploaded to Last.fm).

[2] Cards: https://m3.material.io/components/cards/overview
[3] Small Top App Bar: https://m3.material.io/components/top-app-bar/specs
[4] Floating Action Button: https://m3.material.io/components/floating-action-button/overview
[5] Bottom Sheet: https://m3.material.io/components/bottom-sheets/overview

Adding a podcast

Figure 7

I have used the same Small Top App Bar, with a standard text box and buttons beneath. When an RSS feed is inputted, the screen shows a preview of the episodes that are in the podcast, and the Add button is enabled.

Browsing episodes

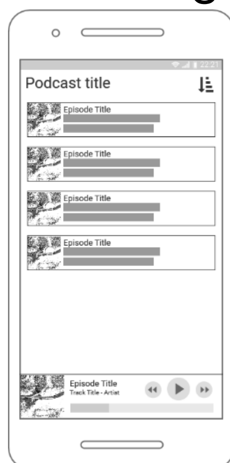

Figure 8

The mock-up for browsing episodes is very similar to the main screen, as it is simply navigating a lower level of the tree. I have omitted the FAB as it would create the wrong associations (it is not possible to add just one episode).

Episode details & tracklist

Figure 9

This page swaps out the Top App Bar for a custom component containing main metadata about the episode, like the episode title and date published. Underneath I used Tabs for the layout – one with the podcast's description, and another with the tracks parsed for the podcast.

User Flow

I created two User Flow diagrams using draw.io to illustrate adding a podcast and playing an episode.

Adding a podcast

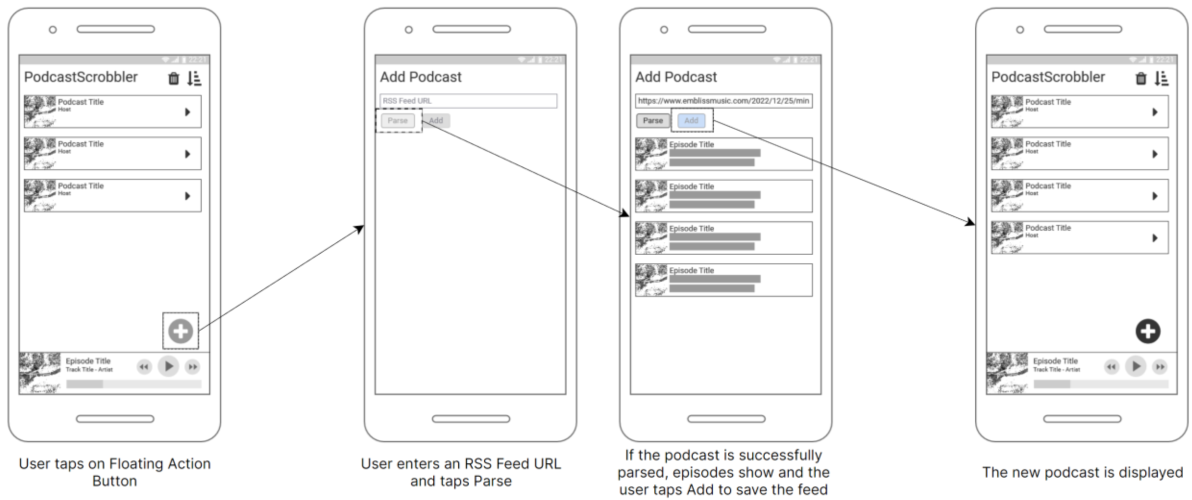

Figure 10

Playing an episode

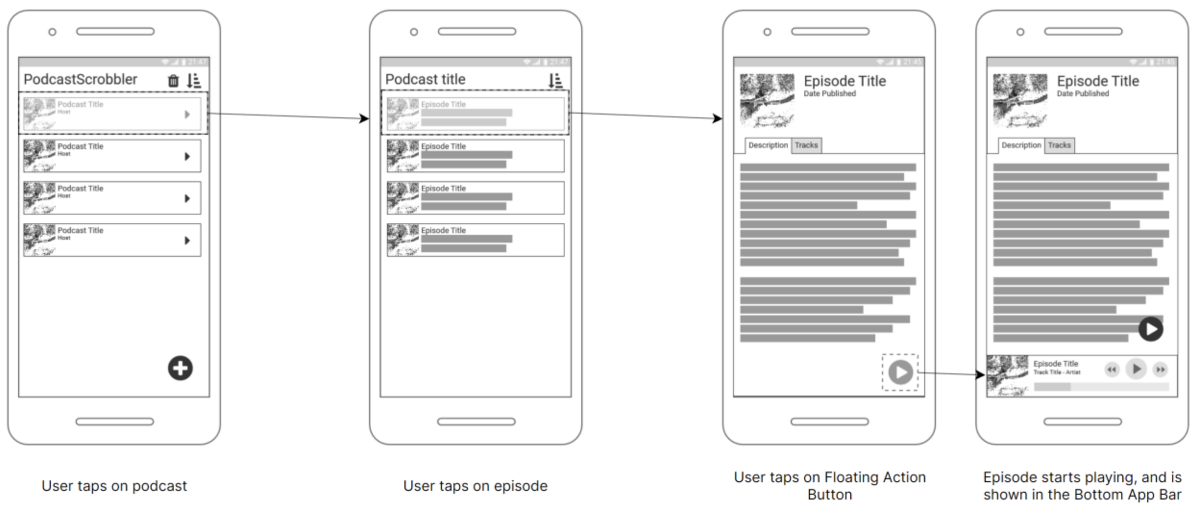

Figure 11

Components

Overview

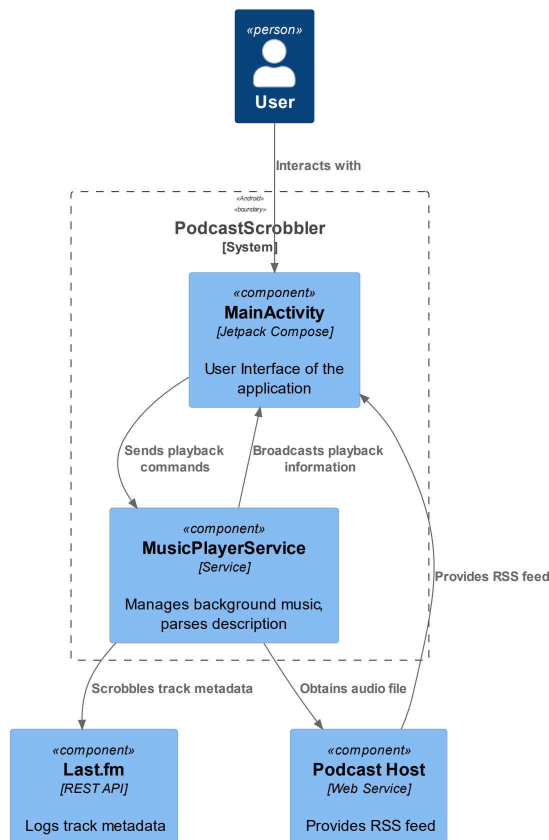

Figure 12

I used PlantUML (PlantUML, 2023) to draw two diagrams using the C4 Model (Brown, 2011). *Figure 12*, left, is a Container Diagram showing a high-level view of the application and how it interacts with outside services. *Figure 14,* at the end of this chapter, is a Component Diagram showing more in depth how each part of the application interacts with each other.

The *MainActivity* is what the user first sees on startup and holds the User Interface. It deals with parsing RSS feeds, storing the data and then fetching it again.

The Android Runtime (ART) garbage collector often prevents foreground activities running in the background, so the actual media player is run in a service instead. A service is created by the system when the application is launched, and then started by the activity through an intent, i.e., when an episode is staged to be played. The activity can then communicate with the service via Android.

The podcast host is used in two cases – first to obtain the RSS feed, which contains the metadata and episode structure, and secondly when an episode is streamed, where it hosts the audio files.

Activity

Jetpack Compose is a declarative UI framework (Android Developers, 2023) – that is, each component contains both its layout and logic. This contrasts with Android's Views, where layout is defined separately to logic, in XML resources. The higher-level name for an object shown to the user is a *component*, whereas the name for the actual code is called a *Composable*. Compose is started in MainActivity's OnStart() method, where it is given a Material parent component to define styles. All components are what's known as a *Composable Function* – annotated with @Composable and calling other Composable Functions within.

PodcastScrobbler Composable

The PodcastScrobbler Composable acts as the root of the tree (other than Material) and contains state that is universal across the app, i.e., a manager for storage, the current navigation position, and a reference to the service for player access. The most important component is the NavController (Android Developers, 2022), which stores a route as a string and shows the corresponding pane. This allows for routes to be defined similarly to a web address, including with parameters. One of four components can be

shown at one time (Podcasts List, Episodes List, Episode Details, and Add Podcast), however when navigating down the tree components above it are kept in a Stack structure. When the user navigates back, the highest component on the stack (the current screen) is popped, revealing the one underneath(the last screen shown).

The other visible UI element is the BottomAppBar which is a parameter to the PodcastScrobbler component rather than the body. This is a similar component to the one found in Google Maps, modified to not extend or contract. It holds one component, Controls.

Service

As described in the overview, the application needs a service as a "sidecar" to perform tasks like playing audio when the activity is not visible to the user. The service in this case extends the MediaSessionService class, part of the Jetpack Media3 API (Android Developers, 2023). It contains functions for staging and playing audio, interacting an instance of Player and Session. *Figure 13* (Android Developers, 2022), below, illustrates the relationship between a MediaSessionService and the Activity that communicates with it.

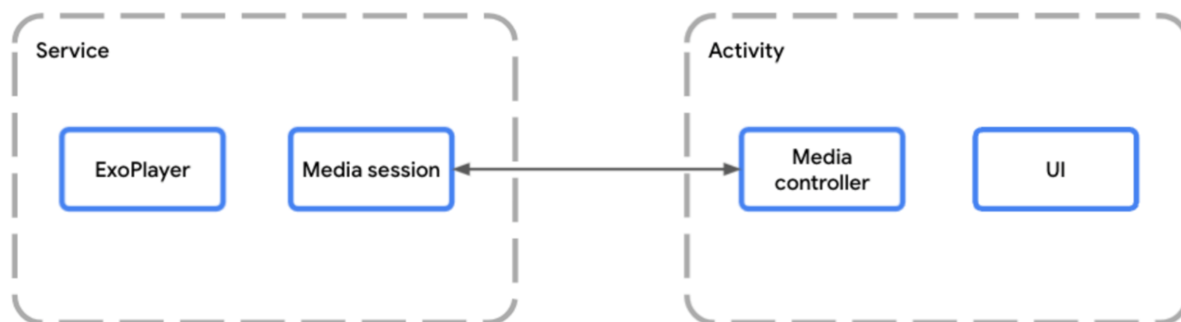

Figure 13

The service also contains a reference to the podcast episode. This allows the service to parse the description and add tracks once playback has commenced. As the service is always running, it sends metadata to Last.fm as opposed to the activity, which could be inactive when playback has reached a point that triggers a Scrobble.

A binder is used to expose the service's public functions. The activity requests a reference to the service via Android System, and then instantiates its own binder which represents the service and can be interacted with.

Data classes

Podcast

Holds metadata for the podcast, as well as an array of Episodes.

Episode

Holds metadata for the episode, as well as audio file links, a function that invokes the tracklist parser on its own description, and an array of Tracks.

Track

Holds the title, artist, record label and the time it starts in an episode.

Utilities

Although unrelated in function, the last three classes are packaged together as they are not dependent on the lifecycle of either the service or the activity but must be accessible by both.

Podcast Manager

A reference to this class is held in the root composable, PodcastScrobbler. It is responsible for serializing and deserializing a list of Podcast objects using JSON. JSON allows for storage in the same structure as the RSS feed, however unlike XML, can be done automatically using Kotlin's Serialization library (Kotlin, 2023). I chose JSON from the available options as it was the only one not marked Experimental. Although binary formats like Protocol Buffers are faster than JSON (Audie Sumaray, 2012), the difference is smaller with text and JSON is human readable, making it more useful when debugging.

RSS To Class

This class contains logic for parsing an RSS feed from XML to a Podcast object populated with Episode objects.

Description to Tracks

The most important class in the project, containing functions which parse a description and returns a list of Tracks if successful.

Component Diagram

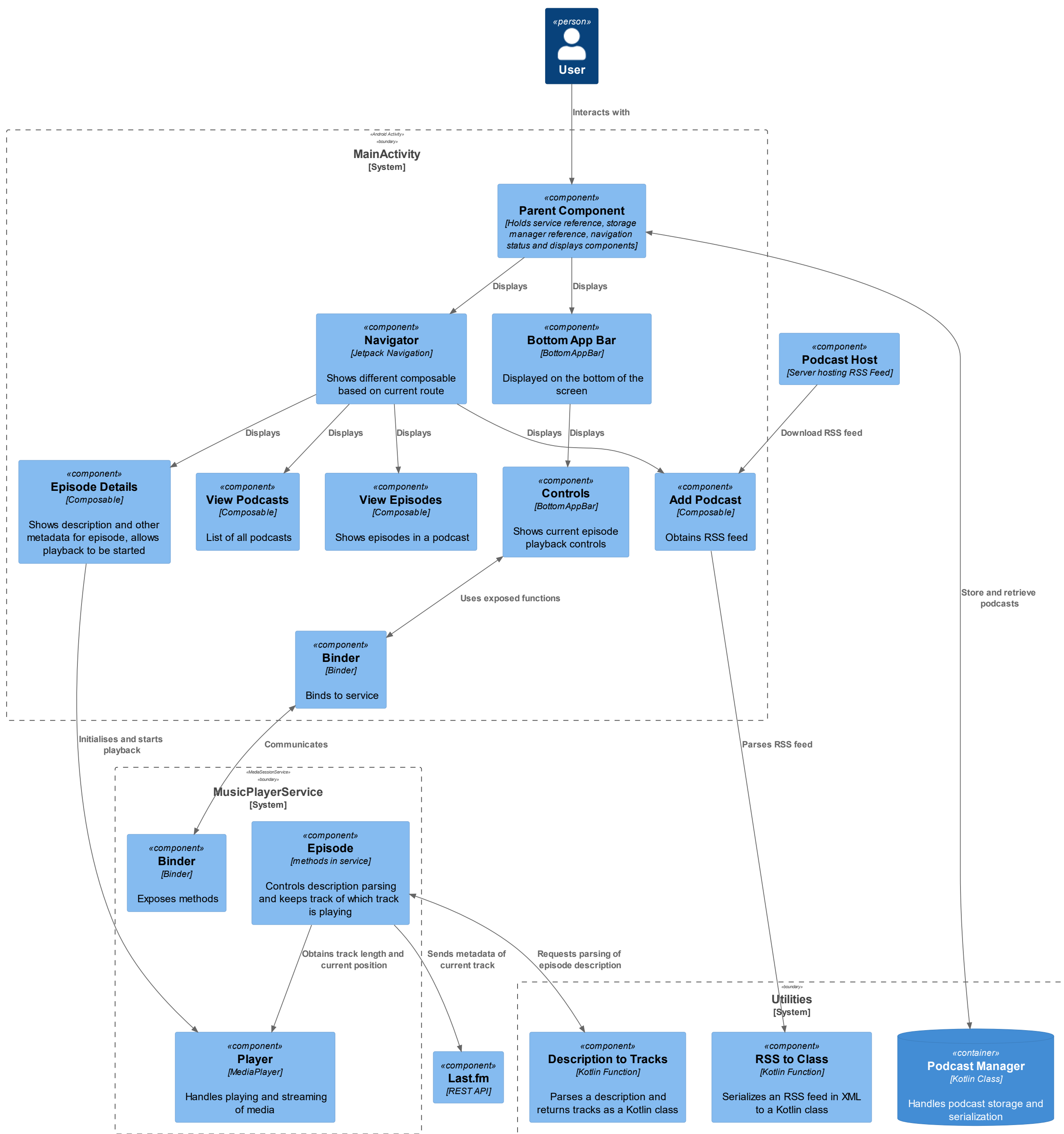

Figure 14

Implementation

Tooling

Kotlin

I decided to use Kotlin, because as of Google I/O 2019 it was announced that Android would be "increasingly Kotlin-First" (Android Developers, 2023). Upon starting this project I had not used Kotlin; I was already familiar with Java, however had become accustomed to conveniences from Dart and JavaScript (in React), like declarative UI, state management and concurrency(i.e. `async await` and Futures).

Kotlin provides these features, through Jetpack Compose and coroutines (Kotlin, 2023). It also has full interoperability with Java, being ran on the JVM. This makes it very suitable for Android development as I can interact with all Android libraries without a bridge like Dart requires (Flutter, 2023).

Android Studio

I used Android Studio as it is the recommended IDE for Android. Being built by Kotlin creator JetBrains, it also includes first-class support for the language.

Target device

Rather than use an emulated Android device, I installed the drivers and connected my own device (a Google Pixel 6) as it can run the application faster and allows for better testing of touch-based inputs.

Repository

I used Git to create a repository, with changes pushed to the remote (in GitHub). The repository is available at https://github.com/Dexyboiiii/Podcast-Scrobbler .

Data classes

I created the 3 data classes listed above (Podcast, Episode and Track). For easier debugging they have custom toString() methods(*Figure 15*) and are annotated with @Serializable for native Kotlin serialization.

```
override fun toString(): String {
    var strToReturn = """

        Title:       $title

        Description: $podcastDesc

        Link:        $link

        Author:      $author
        """.trimIndent()
    var episodesToString = ""
    if (episodes.isNotEmpty()) {
        for (i in episodes.indices) {
            episodesToString += episodes[i].toString()
        }
        strToReturn += "\n\nEpisodes:    $episodesToString"
    }
    return strToReturn
}
```

Figure 15

Parsing an RSS feed

RSS feeds are XML documents with channel elements(metadata) and several items, denoted with the <item> tag (RSS Advisory Board, 2009). For a podcast, the Media RSS format is used, which also includes an <enclosure> tag within each item. XML is well-supported in Java, and because Kotlin has full Java interop I was able to import the org.w3c.dom module and interact with it as if I were writing in Java. I wrote a standalone Kotlin function (outside of an Android project) to test the functionality.

Issues with XML

The first issue I came across was ampersands (&) not being parsed correctly, as some versions of XML, notably HTML up to version 4.0, use ampersands as the escape character (World Wide Web Consortium, 2004). Unescaped ampersands would cause the parser to throw SAXExceptions, so I used a Regular Expression (RegEx) to replace these with & .

```
fileContent = fileContent.replace("&(?!(\\w|#)+;)".toRegex(), "&amp;")
```

Figure 16

Iteration station

I instantiated a skeleton Podcast object and obtained a list of every element in the document. Iterating through the list, I put the elements' text in the corresponding field, creating a new Episode object each time the loop encountered an `<item>` tag. The completed Podcast object is returned.

Parsing the description

Descriptions for podcasts are very unstructured, however a tracklist is typically the most structured part of it. Most are after a few lines of description, so I need to first take the tracklist out of the podcast, and then analyse that line by line. The two podcasts

I first used were *Mind Over Matter* by Embliss, and *The Melodic Sessions* by Prototype202.

Regular expressions

The attribute in common for tracklists with these podcasts is that it is a title, a hyphen, an artist, and sometimes a label surrounded with square brackets. The first task is to take the linebreak tag in XML, (
, <br/> and <br />) and convert it to a newline in Kotlin, \n. Any other HTML tags for emphasis or lists are also removed.

```
// Removes p tags and turns XML line breaks into Kotlin line breaks
description = description.replace("<br />|<br>|<br/>".toRegex(), "\n")
description = description.replace("<.*?>".toRegex(), "")
```

Figure 17

The string is then split by its newlines into an array of strings. The array is iterated and each string tested for whether it counts as a valid track in the list using the criteria above. If the line passes (there's text either side of a hyphen), then the current line is logged as the start of the tracklist and a counter of tracklist length is incremented by 1. Since it's possible to have a non-tracklist line with a hyphen in it, there needs to be at least 3 lines in a row (i.e. the counter is at least 3) with a hyphen to be accepted as a tracklist. If numbering on the tracklist is present, then it's removed. The tracklist is now stored in a separate string array.

To get the actual metadata out of the lines, I wrote RegEx to split each line up into groups, instantiate a Track object with the data and add it to the Track array. The resulting log is printed to the console, and then a Triple containing the tracks array, the parse state (denoting whether it was successful or not) and the log is returned. *Figure 18* shows an extract of this code.

```
val artistTrackSplitterPattern = Pattern.compile("(.+)( – | - )(.+)")
var artistTrackSplitterMatcher: Matcher
val labelScraperPatternPattern = Pattern.compile("(.+)( – | - )(.+)(\\[)
(.+)(])")
var labelScraperPatternMatcher: Matcher
for ((index, unsplitTrack) in rawTracklist.withIndex()) {
    artistTrackSplitterMatcher =
artistTrackSplitterPattern.matcher(unsplitTrack)
    labelScraperPatternMatcher =
labelScraperPatternPattern.matcher(unsplitTrack)
    // If there is a record label present...
    if (labelScraperPatternMatcher.matches()) {
        val trackObj = Track(
            labelScraperPatternMatcher.group(1),
            labelScraperPatternMatcher.group(3),
            labelScraperPatternMatcher.group(5),
            -1
        )
        tracks.add(trackObj)
        // If there isn't a record label present...
    } else if (artistTrackSplitterMatcher.matches()) {
        val trackObj =
            Track(artistTrackSplitterMatcher.group(1),
artistTrackSplitterMatcher.group(3))
        tracks.add(trackObj)
    } else {
        getTracksErrorLog += """

                Could not parse: $unsplitTrack
                """.trimIndent()
    }
}
println(getTracksErrorLog)
```

Figure 18

Persistent storage

To manage podcasts, I created a static utility class to serialize an array of Podcast objects into JSON and save the resulting string to storage. It also deserializes the string in storage when the current list is called and can also perform additions and deletions to the current Podcast array, which are then saved to storage. *Figure 19* shows the

process for retrieving podcasts from storage. Both the add and remove functions save the list of podcasts to storage.

```
fun retrievePodcasts(): SnapshotStateList<Podcast> {
    println("Retrieving podcasts! ${this.context.toString()}")
    val fileDir = context.filesDir
    if (File("$fileDir/podcasts.json").isFile) {
        try {
            return Json.decodeFromString<List<Podcast>>(
                string = File("$fileDir/podcasts.json").readText(
                    Charsets.UTF_8
                )
            )
                .toMutableStateList()
        } catch (e: Exception) {
            println(e.message)
        }
    }

    try {
        File("$fileDir/podcasts.json").createNewFile()
    } catch (e: Exception) {
        println(e.message)
    }
    return mutableStateListOf<Podcast>()
}
```

Figure 19

Persistence is key

One issue I had when rendering lists of Episodes was the names being cut off. I was using the episode title as the key, which seemed logical as no podcast would have more than one episode with the same name - they're always numbered, i.e. Mind Over Matter #151. However, the Navigator, explained later, uses the hash sign (#) as a reserved character in routes, meaning either the route would not resolve or the number would be cut off in the title, making the names often not unique and the list of composables no longer consistent. I solved this by creating a primary key field for the Episode class and generating a key during parsing within RssToClass.

MainActivity and PodcastScrobbler

In Android Studio created a project containing an Android View with Compose. This creates a root composable which is called by MainActivity – I named it PodcastScrobbler, as the name of the application. With Compose you are supposed to move state higher up the component tree so that it can be viewed by all composables that need it. This cuts down on components with their own state, which can lead to more difficult refactoring and tracing of state. As such, the PodcastScrobbler component contains:

- A reference to the Context (an object provided by Android that contains details of the current session)
- An instance of PodcastsManager, which deals with storing and retrieving podcasts

- The state of the BottomSheetScaffold
- The state of the NavController
- A reference to MusicPlayerService
- Attributes from MusicPlayerService that must update state when changed i.e. play state and metadata

Remember

Although calculations are typically ran every time a composable is redrawn, the remember API (Android Developers, 2023) allows the result of a calculation to be stored on the first draw and retrieved on every subsequent draw. When a MutableState object is stored, the component is redrawn every time the object is mutated. *Figure 21* shows how the remember API is used to store variables in PodcastScrobbler.

```
val navController = rememberNavController()
val bottomSheetScaffoldState =
      rememberBottomSheetScaffoldState(SheetState(skipPartiallyExpanded
      = true, skipHiddenState = true, initialValue =
      SheetValue.Expanded));
val context = LocalContext.current
var pm = remember { PodcastsManager(context) }
var podcastsSaved = remember { pm.savedPodcasts }
```

Figure 21

Scaffold

The scaffold is the first component to display something on the UI. It holds a top bar (in this case, a SmallTopAppBar), a floating action button and a bottom sheet. *Figure 20* shows how the scaffold lays out Composable parameters.

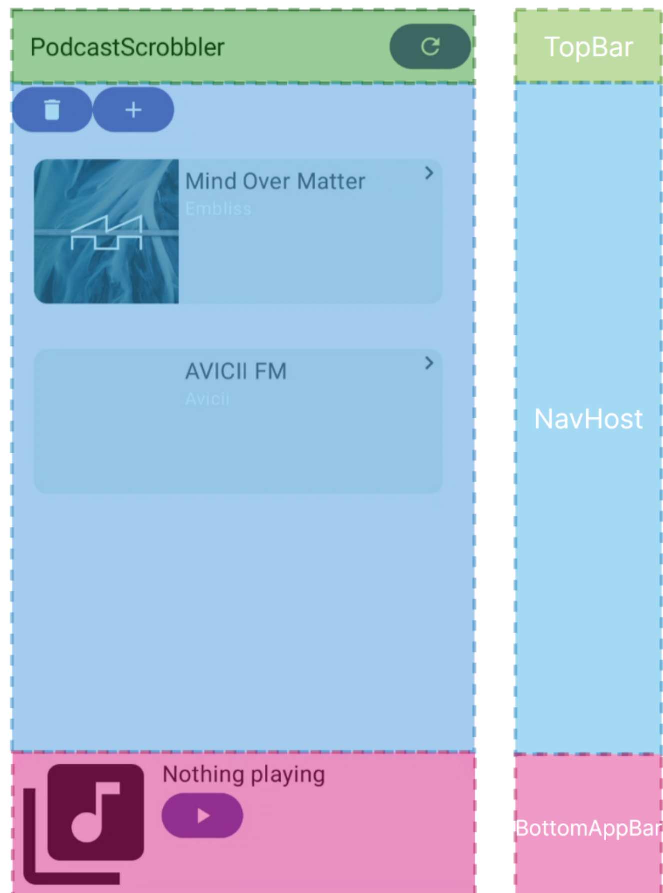

Figure 20

Navigation

The NavController above keeps track of the current route – this is a string with the name of the screen to be displayed, and parameters after it, similar to a HTTP URL. The NavHost component reads the route on each redraw and displays the corresponding composable to the route. Parameters in the route are also passed to the composable. *Figure 22* shows the process for the EpisodeDetails composable – note the {podcastTitle} and {episodeTitle} parameters passed through the route.

```
composable(route =
"${Screen.EPISODE.name}/{podcastTitle}/{episodeTitle}", arguments =
listOf(
    navArgument("podcastTitle") { type = NavType.StringType },
    navArgument("episodeTitle") { type = NavType.IntType }
)) {backStackEntry ->
    EpisodeDetails(service,
backStackEntry.arguments?.getString("podcastTitle"),
backStackEntry.arguments?.getInt("episodeTitle"), podcastsSaved,
navController)
}
```

Figure 22

Browsing podcasts

The default screen is the PodcastsList composable, which requests a list of Podcast objects from the PodcastsManager. It has two buttons, Add and Remove. The first opens the screen below, and the second toggles the action performed by tapping on a podcast Card between opening the episodes list and invoking PodcastsManager to remove it from the list and update the persistent storage with the new list.

Adding a podcast

The AddPodcast composable takes two parameters, a reference to the PodcastsManager and the NavController. After taking a URL to an RSS feed, it starts a suspended (async) function, which in turn uses Volley (Google, 2022) to create a HTTP request. The body of the response is passed to the rssToClass utility, described in *Parsing an RSS feed*. Once these both resolve, a flag is set that triggers recomposition of the screen and renders a set of Cards representing each Episode. The Add button is also enabled, which if tapped saves the new Podcast object to persistent storage via PodcastsManager.

Browsing episodes

When routed to EpisodesList, the NavHost interpolates the podcast title given to it in the route and passes it as a parameter to the EpisodesList composable.

The EpisodesList composable was implemented in the same way as the PodcastsList composable, with a scrollable column of Cards. Due to the size difference (the list of podcasts rarely exceeded 5, whereas one podcast can easily have 150+ episodes), this caused performance issues – for Mind Over Matter the framerate hovered around 30, but the display on the device is 90Hz. The solution to this was to create a LazyColumn composable, which takes an array of objects and instructions to create items in the list for each object. For each scroll position, only the items on screen and slightly off it are drawn, and when scrolled they're drawn and discarded on demand. This resulted in the performance hitting 90 frames per second consistently and dropped loading time for the composable significantly. *Figure 23* and *Figure 24* show the difference between the Column in PodcastsList and the LazyColumn in EpisodesList.

```
Column(Modifier.verticalScroll(rememberScrollState(), enabled = true)) {
    for (podcast in podcasts) {
        PodcastCard(podcast, remove, pm, navController)
    }
}
```

Figure 23

```
LazyColumn(Modifier.verticalScroll(rememberScrollState(), enabled =
true).height(600.dp)) {
    items(podcast.episodes) {episode ->
        EpisodeCard(podcast, episode, navController)
    }
}
```

Figure 24

Episode details

The EpisodeDetails composable takes parameters in the same way as EpisodesList, with another layer. It contains a custom composable, EpisodeDetailsHeader, which displays the album art, metadata, and buttons to parse the tracklist and start playback. Below it is a TabRow that allows for selection of the main component – either the episode description or a list of parsed Tracks. The episode description is displayed as

HTML in a traditional Android View to allow for formatting to be retained if present. *Figure 25* shows EpisodeDetails in action displaying the description and tracks tab.

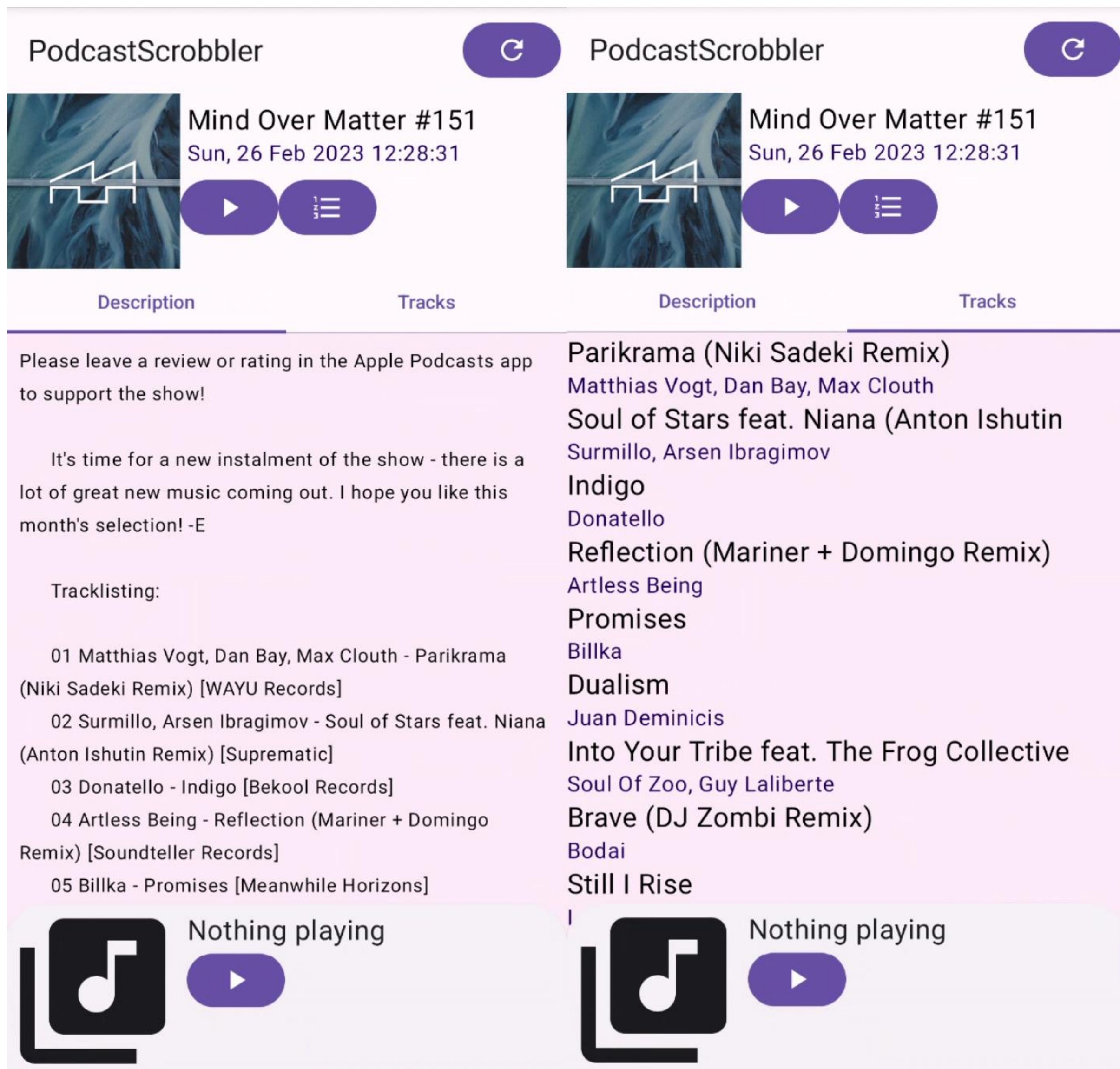

Figure 25

Playing media

Media3

Android has had numerous methods of playing media – MediaPlayer was there first but is now not recommended for new development, MediaSession is more modern, and the third-party ExoPlayer is more feature-rich than Android's own APIs (ExoPlayer, 2023). In 2021 as part of the Android Jetpack initiative, another API was introduced – Media3 (Android Developers, 2021). It promised to unify ExoPlayer and MediaSession, with a new MediaController that shared the same interface (Player) as ExoPlayer. This resulted in less confusion as connectors would no longer be required to bridge a MediaController and MediaSession.

Creating a service

To perform tasks (like playing audio) when the activity isn't in the foreground, or keep it persistent across different app screens, a service must be used. An instance of the service is created on app launch, and the service can be started by issuing an Intent to the system within the activity.

I extended the MediaSessionService class so that I could interact with the player. The overridden onStartCommand function takes the content of the Intent, an Episode object, prepares the player, and starts playback using the audio URL provided.

In MediaSessionService, the onStartCommand is overridden. Messages usually processed by the system are still processed, and messages sent by my activity (either start or pause) are handled by the rest of the function.

Serialization struggles

Intents only allow for data that implements the Serializable class in Java, whereas my classes are annotated with @Serializable from Kotlin. This is one of the (admittedly few) issues with Kotlin-Java interop, as it would be redundant to use both serialization methods.

My solution for this is to serialize the class into JSON in the activity and then deserialize it back into an instance of Episode in the service, as is done with the PodcastsManager for storage.

Posting a notification and starting playback

For a service to run in the foreground (i.e. not get killed by the system), it needs to post an active notification. I create a notification with the title of the podcast and use it to start a foreground activity. *Figure 26* shows this process.

As the instance of Player is already instantiated, I assign it a MediaItem with the URL of the audio in the Episode and set playWhenReady to true. Once the Player buffers enough audio it starts playing immediately.

```
// Creating a notification channel if one doesn't already exist
val notificationChannel = NotificationChannel(
    "podcast_playback",
    "Podcast Playback",
    NotificationManager.IMPORTANCE_LOW
)
notificationManager.createNotificationChannel(notificationChannel)

// Building a notification with the notificationChannel id
val notificationBuilder = NotificationCompat.Builder(this,
notificationChannel.id)
    .setSmallIcon(androidx.media3.session.R.drawable.media_session_servi
ce_notification_ic_music_note)
    .setContentTitle(episode?.title)
val notification = notificationBuilder.build()

// Instructing Android to start a foreground service with the
notification
startForeground(1, notification)
```

Figure 26

Keeping track of the current track

For a tracklist where the time each track starts is unknown, the start times are interpolated, i.e. for a podcast of length 3500 seconds, each track is assumed to be 350 seconds long (for progressive house this is usually the case, as there are few to no interruptions by the DJ). Because the episode's length is not stored in the RSS feed, it can only be obtained on first playback.

I assign an event listener to the player, and when playback starts the service attempts to parse the tracklist if it has not been done already, and then assigns each track the time it should start.

If this is successful, a loop is started using coroutines so that it does not block the main thread. The current track's playback time(stored in a map) is incremented every time the loop runs, and once the track has been sufficiently played (halfway) it is marked to be scrobbled. To keep the notification up to date, its title is updated each time the loop runs with the title and artist of the track. *Figure 27* shows the loop, and *Figure 28* shows the output in Logcat when the player moves from a track that has been sufficiently played to the next track. *Figure 29* shows the notification that is shown while an episode is playing, containing the current track.

I was not able to integrate with Last.fm in the time allotted, however a request would be sent with track metadata at this point once the track had been flagged as sufficiently played.

This method has the advantage of tracking playback time per track as opposed to counting the entire episode's playtime, however it still doesn't address that if the user were to rewind to a previously played track, it would not be flagged again as the loop and corresponding map of tracks are only instantiated when playback is started and not when it is resumed.

```
fun startTrackingLoop() {
    // Start the following loop as a coroutine, so it does not block playback
    CoroutineScope(Main).launch {
        while (true) {
            delay(1000)
            if (player.isPlaying) {
                val ct = currentTrack
                println("${ct?.title} - ${ct?.artist}")
                if (ct != null) {
                    // Get the amount of time the current track has played for,
incremented by 1000ms
                    // If an entry does not yet exist, create one with a value
of 0ms
                    var trackTime =
                        tracksThisSession.computeIfAbsent(ct) { 0 } + 1000
                    // Update the value in the map
                    tracksThisSession.put(ct, trackTime)
                    // Get the amount of time the track should take to play
                    val trackLength: Long? =
                        (episode?.tracks?.get(episode?.tracks!!.indexOf(ct) +
1)?.timestamp
                                )?.minus(currentTrack!!.timestamp)
                    println("${trackTime}/${trackLength}")
                    // If more than 50% played, update the user.
                    if (trackTime > trackLength!! / 2) {
                        println("Track has been sufficiently played!")
                    }
                    // Update the notification with the current track
                    postNotification(ct)
                }
            }
        }
    }
}

fun postNotification(track: Track) {
    // Get notification manager
    val notificationManager = getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager

    // Build notification for the playback notification channel with the current
track and podcast title
    val notificationBuilder = NotificationCompat.Builder(this,
"podcast_playback")
        .setSmallIcon(androidx.media3.session.R.drawable.media_session_service_n
otification_ic_music_note)
        .setContentTitle(episode!!.title)
        .setContentText("${track.title} - ${track.artist}")

    // Post the notification. As the id is the same as the initial notification,
it is updated.
    notificationManager.notify(1, notificationBuilder.build())
}
```

Figure 27

```
com.morrisonhowe.podcastscrobbler    I  Parikrama (Niki Sadeki Remix)  - Matthias Vogt, Dan Bay, Max Clouth
com.morrisonhowe.podcastscrobbler    I  353000/356999
com.morrisonhowe.podcastscrobbler    I  Track has been sufficiently played!
com.morrisonhowe.podcastscrobbler    I  Parikrama (Niki Sadeki Remix)  - Matthias Vogt, Dan Bay, Max Clouth
com.morrisonhowe.podcastscrobbler    I  354000/356999
com.morrisonhowe.podcastscrobbler    I  Track has been sufficiently played!
com.morrisonhowe.podcastscrobbler    D  bufferpool2 0xb4000073728cb648 : 4(32768 size) total buffers - 1(8192 size) used
                                        (fetch/transfer)
com.morrisonhowe.podcastscrobbler    I  Soul of Stars feat. Niana (Anton Ishutin Remix)  - Surmillo, Arsen Ibragimov
com.morrisonhowe.podcastscrobbler    I  1000/356999
pid-2483                             E  Timeout while waiting for metadata to sync for com.morrisonhowe.podcastscrobbler
com.morrisonhowe.podcastscrobbler    I  Soul of Stars feat. Niana (Anton Ishutin Remix)  - Surmillo, Arsen Ibragimov
com.morrisonhowe.podcastscrobbler    I  2000/356999
com.morrisonhowe.podcastscrobbler    I  Soul of Stars feat. Niana (Anton Ishutin Remix)  - Surmillo, Arsen Ibragimov
com.morrisonhowe.podcastscrobbler    I  3000/356999
```

Figure 28

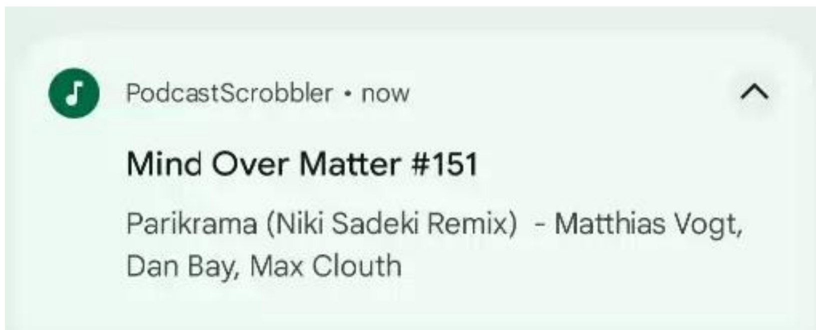

Figure 29

Playback Controls

Binding the service

In the MainActivity the service is bound to the activity, making data transferrable between the two without needing to use intents. The PodcastScrobbler composable contains listeners that update variables within MutableState objects. These variables (i.e. playback state, episode metadata) must be stored in MutableState objects so that the UI updates with said state – it will not update based on the service's own variables.

Controls Composable

The Controls composable keeps references to these values, and so recomposition occurs when they're updated. It has one button for playing and pausing, which updates both the player and its own state. *Figure 30* shows the Controls composable while a podcast is playing.

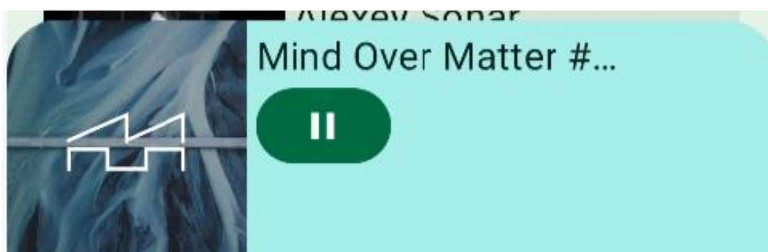

Figure 30

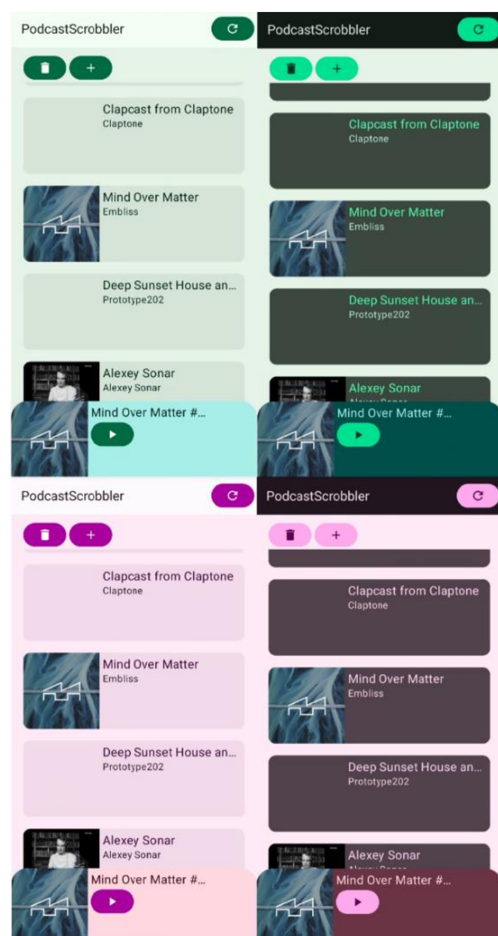

Figure 31

Utilising Material Design

Colour schemes

Material recommends (Google, 2021) the use of the Material Theme Builder to generate colour themes, and then calling their values by their semantic name – i.e. Primary for background and OnPrimary for text.

Dynamic Colour

In 2021 Google unveiled Material You (Google, 2021), with the headline feature being Dynamic Colour. This uses the current wallpaper set by Android to generate a colour palette, which apps on the device can use for theming. The theme I defined checks for Dynamic colour compatibility, and if it exists applies that instead of the colour theme I created. As I used semantic naming rather than hardcoding the colours, the UI is still clear and readable no matter what colours are in the theme. *Figure 31* shows the main screen in both light and dark mode when the system wallpaper is set to green or pink wallpapers.

Implementation conclusion

Due to time constraints, I stopped development after implementing material design. Overall I had created an application that met most of the requirements given and would consider it mostly a success as it has all the functionality of fetching, storing, retrieving, browsing, playing and of course showing the current track. However I was not able to implement Last.fm integration, nor parsing track times from a description (times are estimated and interpolated along the track length instead).

Development experience

I found writing with Kotlin to be incredibly intuitive – Kotlin lived up to the other languages I'd used with more modern features[6], however had unparalleled interoperability with Java. Both when interacting with Media3's Player and Android's intent system, the experience was so frictionless I first assumed they'd both been implemented in Kotlin somehow, yet they both turned out to be implemented with Java.

I also found Android Studio to be very easy to use – I had used Visual Studio Code for most of my projects(and Eclipse when absolutely required) up until now, which while not advertised as an IDE (Microsoft, 2023), had many of its functions. The experience with Android Studio was more cohesive, with many more code suggestions than VS Code. It did however run slower, and I found Git integration to be much more clunky as it separates commits and history into two completely different panes.

---

[6] Async programming, null safety, hot reload, declarative UI, to name a few.

I have mixed opinions of Jetpack Compose, however. The general concepts were easy to grasp, especially coming from React and Flutter, however documentation was not nearly as fleshed out and some use cases had not been covered. Playing media was one of these – there are no pages in Compose's documentation for this use case, which is important as all documentation for playing media relates to Android's old Views architecture. I found it difficult to make components react to state in the Player, as there is no way to call for recomposition like there is in React. Media3 also doesn't contain any pre-built components for Compose like it does for Views (Android Developers, 2021), which is confusing as both Compose and Media3 are initiatives under the Jetpack name.

In Compose's favour, the UI certainly would've taken me longer using the Views architecture given my experience with declarative UI (I have built websites in HTML with JavaScript, which is more like Views, however these have only had very basic interactivity). I came across fewer errors than I did in general than Flutter or React, but this may be down to me having previous experience with declarative UI. It also enabled me to learn Kotlin, language new to me that I'm glad to have tried.

If I were to do the implementation again, I would see how easy it is to build with Flutter or React – at the start of the implementation I criticised them for not running natively on Android and so potentially having a more convoluted method of creating services and calling native Android methods, but I would be interested to actually see for myself whether they'd be more complex.

Effort distribution

As I was new to native Android development, Kotlin and Jetpack Compose, I did not know how long I would be spending on each aspect of development. I had expected to spend about half my time writing both parsers and the podcast storage logic, however working on UI and media playback ended up taking up most of my time instead. Perhaps this was due to me needing to learn a new UI framework, or perhaps this was the UI framework itself. Certainly, if I were to do this again, I'd plan for the UI to take a lot longer.

Testing

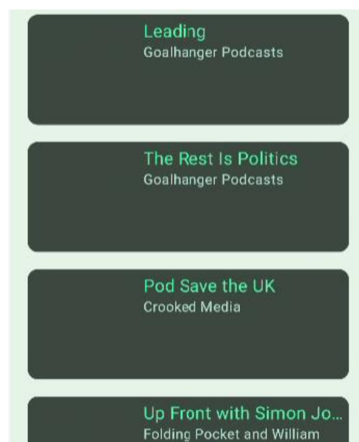

Figure 32

RSS feed parsing

In my development experience, PodcastScrobbler was able to parse the RSS feed of every podcast I gave it. For evaluation, I tried to parse the RSS feeds of the top 5 most popular podcasts in the UK (Chartable, 2023). None of these are DJ sets, however the parser should still work.

The chart below (and *Figure 32*) show the results of this test – all podcasts' RSS feeds were parsed successfully.

One thing to note is that cover images did not appear – the Podcasts List uses the image of the most recent episode, however the podcasts I tested here did not have per-episode images, rather one image for the whole podcast.

Rank	**Title**	**Publisher**	**RSS Parsed**	Reference
1	*Leading*	Goalhanger Podcasts	Yes	(Goalhanger Podcasts, 2023)
2	*The Rest Is Politics*	Goalhanger Podcasts	Yes	(Goalhanger Podcasts, 2023)
3	*Pod Save the UK*	Crooked Media	Yes	(Crooked Media, 2023)
4	*Up Front with Simon Jordan*	Folding Pocket and William Hill	Yes	(William Hill, 2023)
5	*The Diary Of A CEO with Steven Bartlett*	Steven Bartlett	Yes	(Bartlett, 2023)

Podcast storage

For every podcast tested, it persisted between sessions and device restarts. Deletion persists also, as well as tracklists parsed from descriptions in the Episode Details screen. Although descriptions are parsed when playback is started, the times do not persist as the service does not have a reference to the PodcastsManager.

Although metadata like publisher and dates for each episode are viewable, I did not implement podcast-wide images – only per-episode. This means that some podcasts that do not define an image for each episode show with no image at all. I was also not able to implement favouriting or sorting.

Parsing descriptions

I added six podcasts that had description structures suitable for parsing. They are not chosen with a methodology i.e. 5 most popular, as they are a niche in podcasting; In the Apple Podcasts GB Music charts, only 1 of the 20 most popular is a DJ set with a tracklist, *Defected Radio* by Defected (Chartable, 2023).

For each podcast, I attempted to parse the descriptions of the most recent 10 episodes.

Title	**Publisher**	**Descriptions Parsed / 10**	Reference
Mind Over Matter	Embliss	10 correct	(Brandwijk, 2022)
The Melodic Sessions	Prototype202	10 correct	(Prototype202, 2023)
Alexey Sonar	Alexey Sonar	10 correct	(Sonar, 2023)
Defected Radio	Defected	9 slightly incorrect, 1 had no tracklist in the description	(Defected, 2023)
AVICII FM	Avicii	1 correct, 9 incomplete	(Bergling, Avicii FM, 2018)
Clapcast from Claptone	Claptone	10 correct	(Claptone, 2023)

In most podcasts tested, *PodcastScrobbler* accurately obtained track metadata, however two notable exceptions were *Defected Radio* and *Avicii FM*.

The former had track start times in the description, which I did not implement. This resulted in tracks parsed with the start time included, as shown in *Figure 33*. No other podcasts I assessed the parser on included times, and as it was not one I originally wrote the parser against, no reference for a podcast with times.

Podcast from Defected Records

Deep Dish - Stay Gold [Deconstruction] 00:00
Joeski - Groove Me [Tango Recordings] 05:10
DJ Spen - Craze At Midnight (Spen's Live CDJ1000 Mix) [Code Red Recordings] 09:02
Butch & Nic Fancuilli – I Want You [Defected] 13:58
Sandy Rivera feat. Haze - Changes [Defected] 21:38
A Studio feat. Polina – SOS (Skylark Remix – Nic Fancuilli Edit) [Defected] 26:14
Cloud 9 - Do You Want Me Baby [Unkwn Records] 30:36
Terrence Parker - Your Love [Intagible Records and Soundworks] 34:13
Mood II Swing - Move Me (Alternate Version) [Sex Mania] 38:38
Alexander East - Jest For Me [Afterhours] 41:58

Stay Gold [Deconstruction] 00:00
Deep Dish
Groove Me [Tango Recordings] 05:10
Joeski
Craze At Midnight (Spen's Live CDJ1000
DJ Spen
I Want You [Defected] 13:58
Butch & Nic Fancuilli
Changes [Defected] 21:38
Sandy Rivera feat. Haze
Nic Fancuilli Edit) [Defected] 26:14
A Studio feat. Polina – SOS (Skylark Remix
Do You Want Me Baby [Unkwn Records]
Cloud 9
Your Love [Intagible Records and
Terrence Parker

Figure 33

The latter, *Avicii FM*, had a very peculiar issue – the person who wrote the tracklist used a different type of dash between tracks in the tracklist, rendering parsing impossible (only if more than 3 tracks in a row shared the same track-artist delimiter would a tracklist appear, and even then, *only* those 3 would show). Although it may be very difficult to discern in the images provided, *Figure 34* shows a Hyphen character between track and artist, whereas *Figure 35* shows an En Dash character. These are both from the same episode's description.

```
02. Sultan + Shepard - Head Over Heels
```

Figure 34

```
04. Firebeatz X Peppermint feat. Aidan O-Brien – Everything
```

Figure 35

Playing audio

I was able to implement playing episodes, along with background playback, as shown in *Figure 29*. Descriptions are automatically parsed if not done so already, and the estimated track is displayed in a notification to the user.

There are also basic(play/pause) playback controls, available across the app in the bottom bar – shown in *Figure 30*. New episodes can be played, however only when the current episode is paused (I was not able to determine the cause of this).

Uploading metadata

I was not able to implement Last.fm integration in the time that I had – as such, although the application is still named *PodcastScrobbler*, it is currently unable to do the last part(although it is as close as it can get). How this would be implemented is described in the Future Work section.

Against requirements

ID	Requirement Description	Criteria	Priority	Implemented?	Comments
1.0	The application must be able to parse an RSS feed	A URL to an RSS feed is provided and serialized to some kind of Kotlin object	High	Yes	
1.1	The application must be able to store podcasts	Information contained in an RSS feed can be stored and retrieved	High	Yes	
1.2	The application must be able to update a podcast with new episodes and notify the user	When a new episode is published, a notification is served within 1 hour and the stored podcast is updated	Medium	No	Although not possible with this iteration, refresh functionality(from storage) is built in and adding fetching of podcasts would be possible. Automatic fetching of new episodes would require integration with Android's Job Scheduler.
2.0	The application must be able to present stored podcasts and episodes to the user	Stored podcasts and episodes are viewable in the User Interface	High	Yes	
2.1	Episode descriptions and details must be visible	Episode descriptions are viewable in the UI	High	Yes	
2.2	All other details of a podcast must be visible	All other details of a podcast are viewable in the UI	Medium	Somewhat	Podcast metadata and descriptions are visible, however depending on the podcast, cover images are not viewable as they're only stored against episodes.
2.3	Podcasts can be sorted, i.e. by title, most recent episode date, length	Podcasts can be sorted ascending or descending by parameters in the User Interface	Low	No	
2.4	Podcasts can be favourited	Podcast objects have an additional "favourited" field, with a separate view in the UI	Low	No	
3.0	Episode descriptions can be parsed, with Track objects created and viewable	Episode descriptions are parsed either when asked to by the User or when played. Tracks are viewable within Episode Details UI	High	Yes	Parsing tracklists with times is possible and works, however times appear within the track title.

3.1	Track timestamps are interpolated from any tracklists detected	Podcast track timestamps are interpolated and viewable in the UI	High	Yes	
4.0	Audio for an episode must be able to be played	Episodes can be played in the episode screen, and keep playing in the background outside of it	High	Yes	
4.1	Tracks in a podcast must be displayed when they are played	When playing a podcast, if its description was successfully parsed, track metadata must be shown to the user either when it is played or predicted to be played if interpolated	High	Yes	
4.2	Tracks must have controls to adjust playback	There are controls shown when playing an episode, for example: play/pause, skip forward/back	High	Somewhat	Play/Pause is functional, however there are no playback progress bar controls.
5.0	A user must be able to log in to Last.fm	A log in page is shown to the user, and they can log in. Their account details are shown in the UI.	High	No	
5.1	When played, a track's metadata must be uploaded to Last.fm or similar and associated with the user	When a track is played at least halfway (Last.fm, 2023), details must be uploaded.	High	No	

I was able to complete most requirements with a priority of High, apart from Last.fm integration. I'm glad to be able to say that the core idea of taking a podcast, being able to play it, and having the current track displayed does work well. Most other features like favouriting and ordering podcasts, while nice to have, are not essential to the functionality of the app.

Although not listed in the requirements, I implemented Material Design components every step of the way, as well as dynamic colour and light/dark modes. As a result, PodcastScrobbler looks at home on Android 13, where it's still a rarity for third-party

applications to use these features due to the higher target API required to implement them (Toombs, 2022).

Project Management

Methodology

The requirements for my project were very unlikely to change in the timeframe, and there's a clear scope, so I opted to use a Waterfall methodology. Given I am not working part of a team or in a company structure, less emphasis can be given on communication, a strong tenet of methodologies like Agile (Laoyan, 2022). Waterfall is one of the most straightforward methods, with planning done in advance before development starts. It generally contains the following steps (Leeron Hoory, 2022):

1. Requirements gathering
2. Design
3. Implementation
4. Verification
5. Maintenance

As project maintenance is not part of the scope of my requirements, I implemented only the first four steps. I also added extra steps for tasks like writing the report. I created a Gantt chart to show expected completion dates of required tasks.

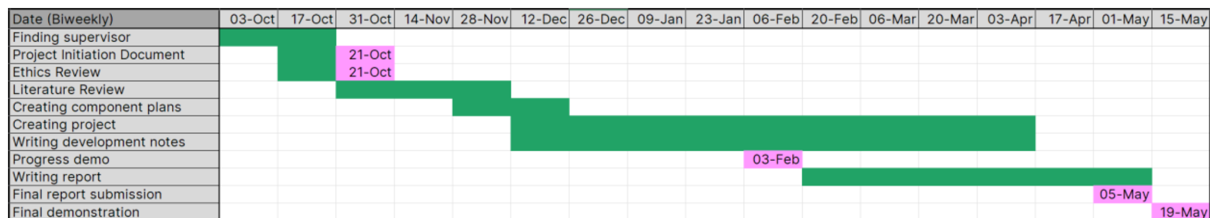

Figure 36

This project requires the use of UI mockups, as well as architecture planning, to decide how both the logic and layout interact with eachother. It is likely to require many components, so I aimed to have plans completed by the time I properly started development. I intended to start development just before planning is complete, so if it's apparent early on that I misjudged something that affects the project plan, I could incorporate it.

As the progress demo was due early February, I planned to be well-into development by that point. As recommended by my supervisor, I took development notes throughout the project ready for the report.

Although Waterfall would typically indicate that I complete my project before I start the report properly, I aimed to start it while I was still implementing the project so that I could document earlier stages like UI planning and the high-level component layout.

Risks

As this project did not involve operating with any sensitive data or dangerous equipment in this project, the risk assessment only contains items risks that would cause project setbacks.

Description	Impact	Mitigation/Avoidance
Last.fm decides to close or restrict its API	High	Although I wouldn't have much influence to avoid this issue, I would be able to compile and display the list of tracks on-device to take Last.fm out of the equation. This would defeat much of the point of the application, however it would still allow me a chance to build a Material UI 3 podcast playing app.
Parsing a podcast's description doesn't prove possible with enough podcasts (it's too variable)	Medium	Although it may be possible to implement looser pattern matching, a completely alternate solution would be using an API like ShazamKit to recognize the song and get track data for every song.
My laptop breaks	High	I plan to store all source code on an SCM like GitHub, therefore I should be able to still develop my application using the University machines (even if less convenient).

Evaluating project management

Overall I was able to manage my project relatively effectively – most stages were completed in their respective timeframes, and each stage was started in the correct order.

Writing my literature review took longer than expected which had the knock-on effect of starting development of the application about a month and a half later than planned (development started in February). This in turn caused the project to be in an almost-complete state a week later, in mid-April. The report was also started about a month later than planned, around the start of April. By this point I did have the added advantage of having most of the deliverables complete, making myself more informed as I came to write the report.

A potential flaw of Waterfall was that my being late in one aspect affected the project as a whole. However, I feel it was necessary to have most of the planning complete by the time I started with implementation.

I wrote a good amount of notes during development documenting classes I'd used and issues I had – these helped me quickly draft my report structure and recall aspects of the development experience quicker than I would've been able to from memory and Git commits alone.

Conclusion

Summary

In this project, I researched different ways tracks from a mix (in the form of a podcast) could be parsed and tracked using Last.fm. Seeing a lack of available tools for this, I built an Android application using Kotlin and Jetpack Compose to bridge the gap.

In testing, the application managed to correctly parse tracklists the majority of the time; Adding track time functionality and consistent tracklist formatting (in the case of Avicii FM) would bring it up to a 100% success rate. RSS Feed parsing, however, did have a 100% success rate, correctly parsing every podcast I tested it with.

Although I almost managed to close the gap from podcast description to Last.fm page, I was not able to implement integration with Last.fm in time, due to the user interface taking longer than expected. If I were to do this again, I would have allocated more time to that task.

With future work on completing the basic project idea and implementing the features that would bring it more in line with other podcast players, I think the USP of this application would make it shine to the potential userbase who both track their music and listen to podcasts. Alternatively, this project could assist those with pre-existing applications implement the tracklist parsing features.

On a personal note, since being introduced to Last.fm about 4 years ago I've wanted to solve the problem of tracking music I discover through podcasts, however I was not close to the level of experience needed to build a project this complex. To finally have the opportunity to (almost) build it has been a great experience for me, and I look forward to actually making it complete in the future.

Future work

I did not have enough time to complete all of the requirements – if I were to continue work on this project, I implement the following:

Last.fm integration

The most obvious first action is to complete the core functionality of the application by integrating with Last.fm (and similar open-source services like Libre.fm).

It would not be too time-consuming to just make an HTTP POST request using the Last.fm API with track metadata. Last.fm uses accounts however, and obtaining a user key to send with the request would likely require a rework of the application for storing said key(it would be ill-advised to just store it as JSON in plain-text) and two new screens: one to host an in-app web browser for obtaining the key via signing into Last.fm, and another to change accounts or revoke access.

Parsing tracklists with times

Although the ratio of tracklists with times to ones without is low, I managed to find at least one(which was tested on earlier). Adding the ability to parse these times would require a minor rework of the parser; it currently only has two exit states (failed and parsed without times) as only the service can currently add times to tracks. However, it

would result in far more accurate track start times than is currently possible, if podcast creators put the extra work in of providing these times.

Slightly relatedly, it may also be useful to allow a user to edit track start times and metadata manually. If a user plays a particular podcast (without times in the description) a lot, they may find it useful to add their own times, or adjust the metadata in case the parser was incorrect.

Enhanced playback controls

In its current state, the application only shows the current episode title and track in a notification, to allow for the service to run in the foreground. It has a greatly reduced feature set relative to a notification that implements a Media NotificationStyle – an example from Pocket Casts is shown in *Figure 37*. Note the album art, progress bar and play/pause, which could be implemented with no modifications to the Player.

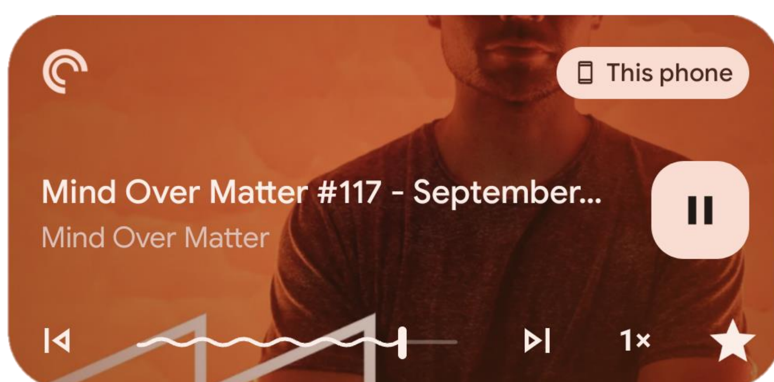

Figure 37

Lower priority requirements

Features like favourites and ordering podcasts are present in almost all standard podcast clients. Pocket Casts (Pocket Casts, 2023), rated Best Podcast App by Android Police (Hagop Kavafian, 2023), is praised for organising episodes by season and having controls for playback speed and a sleep timer.

Features like these would bring *PodcastScrobbler* in line with other popular podcast apps. Most would not be difficult to implement, i.e. favourites would only require a new field and a method of changing said field in the Episode Details screen, and Kotlin has built-in functionality for sorting items in an array based on their attributes which would make ordering by i.e. most recent episode, or titles alphabetically, possible.

Google Cast controls would also be a very useful addition as it would make podcasts playable on Google Cast enabled speakers and displays. This would require a lot more work however – Google Cast works by instructing smart speakers and displays to navigate to a webpage for playback (Google, 2022), and so this would necessitate the creation of an entirely new service on a different platform to facilitate the functionality[7].

[7] There is an alternate mode which streams audio from the device, but the audio must be on the device first, not streamed, therefore this would not be applicable to all scenarios.

References

Adenuga, A. (2022). Spotify 'Unwrapped': An Exploration of Data-Based. *iSChannel*, 3-11.

Android. (n.d.). *Design for Android*. Retrieved from Android Developers: https://developer.android.com/design

Android Developers. (2021, October 27). *Introducing Jetpack Media3*. Retrieved from Android Developers Blog: https://android-developers.googleblog.com/2021/10/jetpack-media3.html

Android Developers. (2022, November 16). *Navigation*. Retrieved from Android Developers: https://developer.android.com/guide/navigation

Android Developers. (2022, October 28). *Play media in the background*. Retrieved from Android Developers: https://developer.android.com/guide/topics/media/media3/getting-started/playing-in-background#using-mediasessionservice

Android Developers. (2023, March 1). *Android's Kotlin-first approach*. Retrieved from Android Developers: https://developer.android.com/kotlin/first

Android Developers. (2023, April 5). *androidx.compose.runtime*. Retrieved from Android Developers Reference: https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary#remember(kotlin.Function0)

Android Developers. (2023, March 1). *Jetpack Compose Overview*. Retrieved from Android Developers: https://developer.android.com/jetpack/compose/why-adopt#less-code

Android Developers. (2023, March 23). *Media3 is ready to play!* Retrieved from Android Developers Blog: https://android-developers.googleblog.com/2023/03/media3-is-ready-to-play.html

Antonelli, W. (2023, September 23). *Last.fm tracks all your music stats by 'scrobbling' them. Here's what that means and how it works.* Retrieved from Business Insider: https://www.businessinsider.com/guides/tech/what-is-last-fm-scrobbling?r=US&IR=T

Audie Sumaray, S. K. (2012). A comparison of data serialization formats for optimal efficiency on a mobile platform. *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication* (pp. 1-6). New York, NY, USA: Association for Computing Machinery.

Balsamiq. (2023, March 8). *Balsamiq Wireframes*. Retrieved from Balsamiq Wireframes: https://balsamiq.com/wireframes/

Bartlett, S. (2023, May 05). *The Diary Of A CEO with Steven Bartlett*. Retrieved from Anchor: https://anchor.fm/s/81678e4c/podcast/rss

BBC News. (2003, March 27). *Website offers new view of music*. Retrieved from BBC News: http://news.bbc.co.uk/1/hi/technology/2888431.stm

Bergling, T. (2018, March 25). *Avicii FM*. Retrieved from Podtree: https://officialaviciipodcast.podtree.com/feed/podcast/

Bergling, T. (2018). Avicii FM #008.

Berry, R. (2006). Will the iPod Kill the Radio Star? *Convergence*, 143-162.

Brandwijk, T. (2022). Mind Over Matter Podcast #150. [RSS Feed]. Retrieved from https://www.emblissmusic.com/2022/12/25/mind-over-matter-podcast-150-year-mix-2022/

Brown, S. (2011). *The C4 model for visualising software architecture*. Retrieved from The C4 Model: https://c4model.com/

c210344. (2016, May 18). *Looking for a podcast app that scrobbles*. Retrieved January 29, 2023, from Reddit: https://www.reddit.com/r/lastfm/comments/4jhl45/comment/d3a2lxr/?utm_source=share&utm_medium=web2x&context=3

Chartable. (2023, May 5). *Podcast Charts - Apple Podcasts - Great Britain - All Podcasts*. Retrieved from Chartable: https://chartable.com/charts/itunes/gb-all-podcasts-podcasts

Chartable. (2023, May 6). *Podcast Charts - Apple Podcasts - Great Britain - Music*. Retrieved from Chartable: https://chartable.com/charts/itunes/gb-music-podcasts

Claptone. (2023, May 1). *Clapcast by Claptone*. Retrieved from This Is Distorted: https://portal-api.thisisdistorted.com/xml/clapcast

Crooked Media. (2023, May 4). *Pod Save the UK*. Retrieved from Simplecast: https://feeds.simplecast.com/snMMEVFU

Defected. (2023, May 5). *Defected Radio*. Retrieved from This Is Distorted: https://portal-api.thisisdistorted.com/xml/defected-in-the-house

ExoPlayer. (2023, April 19). *ExoPlayer*. Retrieved from ExoPlayer: https://exoplayer.dev/

Flutter. (2023, April 26). *Writing custom platform-specific code*. Retrieved from Flutter Docs: https://docs.flutter.dev/platform-integration/platform-channels

Goalhanger Podcasts. (2023, May 1). *Leading*. Retrieved from Acast: https://feeds.acast.com/public/shows/leading

Goalhanger Podcasts. (2023, May 5). *The Rest Is Politics*. Retrieved from Acast: https://feeds.acast.com/public/shows/620cc95e2641e200137b94d8

Google. (2021, May 18). *Cards*. Retrieved from Material 3: https://m3.material.io/components/cards/overview

Google. (2021, May 18). *Floating Action Button*. Retrieved from Material Design 3: https://m3.material.io/components/floating-action-button/overview

Google. (2021, October 27). *Introducing Material Theme Builder*. Retrieved from Material Design Blog: https://material.io/blog/material-theme-builder

Google. (2021, May 18). *Unveiling Material You*. Retrieved from Material Design Blog: https://material.io/blog/announcing-material-you

Google. (2022, August 17). *Google Cast Overview*. Retrieved from Google Cast Docs: https://developers.google.com/cast/docs/overview

Google. (2022, February 18). *Volley overview*. Retrieved from Volley Documentation: https://google.github.io/volley/

Götting, M. C. (2022, February 8). *Most commonly used apps for listening to podcasts among podcast listeners in the United States in 2019 and 2020*. Retrieved from Statista: https://www.statista.com/statistics/943537/podcast-listening-apps-us/#statisticContainer

Hagop Kavafian, J. B. (2023, February 15). *8 best podcast apps on Android in 2023*. Retrieved from Android Police: https://www.androidpolice.com/best-podcast-apps

Hammersley, B. (2004, February 12). *Audible revolution*. Retrieved from The Guardian: https://www.theguardian.com/media/2004/feb/12/broadcasting.digitalmedia

J W Rainsbury, S. M. (2006). Podcasts: an educational revolution in the making? *Journal of the Royal Society of Medicine, 99*, 481-482.

Kostek, B. (2018). Listening to Live Music: Life Beyond Music Recommendation Systems. *Joint Conference - Acoustics*, 1-5.

Kotlin. (2023, April 28). *Coroutines*. Retrieved from Kotlin Docs: https://kotlinlang.org/docs/coroutines-overview.html

Kotlin. (2023, April 28). *Serialization*. Retrieved from Kotlin Docs: https://kotlinlang.org/docs/serialization.html

Krastrenakes, J. (2022, November 22). *Last.fm turns 20 and now has a following on Discord*. Retrieved from The Verge: https://www.theverge.com/2022/11/22/23473358/lastfm-discord-bot-neil-young-spotify

Laoyan, S. (2022, October 15). *What is Agile methodology? (A beginner's guide)*. Retrieved from asana: https://asana.com/resources/agile-methodology

Last.fm. (2023). *When is a Scrobble a Scrobble?* Retrieved from Last.fm: https://www.last.fm/api/scrobbling#when-is-a-scrobble-a-scrobble

Last.fm. (n.d.). *Scrobbling 2.0 Documentation*. Retrieved 01 29, 2023, from Last.fm: https://www.last.fm/api/scrobbling#when-is-a-scrobble-a-scrobble

Leeron Hoory, C. B. (2022, March 25). *What Is Waterfall Methodology? Here's How It Can Help Your Project Management Strategy*. Retrieved from Forbes Advisor: https://www.forbes.com/advisor/business/what-is-waterfall-methodology/

Lewis, D. (2023, January 29). *Apple Podcasts Statistics*. Retrieved from Podcast Industry Insights: https://podcastindustryinsights.com/apple-podcasts-statistics/

Louis, T. (2000, October 13). *Discussion of XML news / announcement / syndication / resource discovery formats*. Retrieved from Yahoo! Groups: https://web.archive.org/web/20131031070818/http://groups.yahoo.com/neo/groups/syndication/conversations/topics/698

Madaan, R. (2023, April 18). *New Spotify Home page UI on Android & iOS faces backlash from users*. Retrieved from PiunikaWeb: https://piunikaweb.com/2023/04/18/new-spotify-home-page-ui-on-android-and-ios-faces-backlash/

Microsoft. (2023, May 3). *Visual Studio Code Frequently Asked Questions*. Retrieved from Visual Studio Code: https://code.visualstudio.com/docs/supporting/FAQ

Perry, A. (2023, March 16). *Spotify's big update isn't just annoying, it misses the point*. Retrieved from Mashable: https://mashable.com/article/spotify-tiktok-fyp-missing-the-point

Peter Josling, C. L. (2020, August 25). *Scroball*. Retrieved from GitHub: https://github.com/peterjosling/scroball

PlantUML. (2023, April 18). *PlantUML*. Retrieved from PlantUML: https://plantuml.com/

Pocket Casts. (2023). *Pocket Casts*. Retrieved from Pocket Casts: https://pocketcasts.com/

Prototype202. (2023, April 19). *The Melodic Sessions*. Retrieved from Prototype202: https://www.prototype202.com/rss/prototype202.xml

React. (2023). *Managing State*. Retrieved from React: https://react.dev/learn/managing-state

RSS Advisory Board. (2009, March 30). *RSS 2.0 Specification*. Retrieved from RSS Advisory Board: https://www.rssboard.org/rss-specification

Sonar, A. (2023, May 6). *Alexey Sonar*. Retrieved from PromoDJ: https://promodj.com/alexeysonar/rss.xml

Taylor-Watt, D. (2018, June 26). *Introducting the first version of BBC Sounds*. Retrieved from BBC: https://www.bbc.co.uk/blogs/aboutthebbc/entries/bde59828-90ea-46ac-be5b-6926a07d93fb

TechRadar. (2006, October 3). *Digital Music Award winners announced*. Retrieved from TechRadar: https://web.archive.org/web/20120229171651/http://www.techradar.com/news/audio/portable-audio/digital-music-award-winners-announced-159274

Toombs, C. (2022, February 12). *Android 13 DP1 opens up dynamic icon theming to third-party apps*. Retrieved from Android Police: https://www.androidpolice.com/android-13-dp1-gives-your-home-screen-icons-a-touch-of-material-you-theming/

William Hill. (2023, May 4). *Up Front with Simon Jordan*. Retrieved from Transistor: https://feeds.transistor.fm/up-front

Wohlstadter, J. (2023, January). *Introducing DJ Mixes on Spotify & Apple Music, powered by Proton*. Retrieved from Proton Radio: https://intercom.help/proton-radio/en/articles/2360688-introducing-dj-mixes-on-spotify-apple-music-powered-by-proton

World Wide Web Consortium. (2004). *HTML Compatibility Guidelines*. Retrieved April 30, 2023, from XHTML: https://www.w3.org/TR/xhtml1/guidelines.html