JANUARY 13, 2023

CORAL

YOUR PERSONAL MUSIC STREAMING SERVICE

JOHN PATRICK GLATTETRE
DE MONTFORT UNIVERSITY
Leicester, United Kingdom

Table of Contents

Acknowledgements

I would like to give a special thank you to my supervisor Dr. David Smallwood for allowing me to work on a project I've always wanted to create but never had the time to devote to. I am grateful to my father for providing incredible technical guidance, insightful conversations about software architecture and overall helping me become a better engineer.

I want to thank my partner, Ana, for her continuous support throughout the development process, whether it be through the surprise snacks and drinks or being the best rubber ducky ever. I would also like extend my gratitude to my friends in the *comfy* Discord server, for inspiring me and constantly giving me something to think about.

Finally, my cat Mochi for being a never-ending source of dopamine.

Literature Review

Introduction

As a DJ and artist, I maintain a significant collection of lossless encoded music on my personal server. I purchase music from independent record labels and receive music from other artists. This means that most of the music I listen to is stored locally on my devices. One potential solution to make this extensive collection accessible across multiple devices is to utilize file syncing software to mirror files across various devices. However, this approach might not be feasible for devices with less storage. On those devices one can employ audio transcoding to a format which occupies less space. Transcoding files can be cumbersome as it necessitates managing two separate collections of music if the original copies are to be retained. Furthermore, if one forgets to convert and sync an album before traveling to an area with limited internet access, it may not be available for playback. Coral aims to solve these challenges by providing a web application to access remote music collections on even the slowest connections using modern audio compression and content delivery techniques.

Comparing self-hosted players with their commercial alternatives

Self-hosting is the practice of managing applications running on infrastructure you control rather than using relying on products created by large companies. An example of this would be listening to music through Coral as opposed to subscribing to Spotify or another streaming service. The popular self-hosted streaming platforms Plex[1] and Jellyfin[2] allow users to stream movies, TV shows and music from their own collection. They provide a more traditional music player experience, where you pick what you want to listen to. Commercial streaming platforms enhance the listening experience by using machine learning models to provide recommendations for what to listen to next and creating dynamic playlists based around a theme (Pastukhov, 2022).

Figure 1 - Spotify recommending playlist additions

[1] https://plex.tv/
[2] https://jellyfin.org/

Spotify provides an API which allows developers to query their library for musical analysis metadata such as key and speed in beats per minute as well as other metrics that describe the musical properties of a song (Spotify, 2023). Their API can be leveraged to create an experience like that of a commercial streaming platform with song recommendations based on what's being played and what the user likes (Spotify, 2023), which can contribute a better user experience than self-hosted music streaming platforms such as Navidrome[3]. However, there are cases where a user owns music that is unavailable on Spotify, which means that those songs would be out of reach for the recommendation system. It is possible to mitigate this problem by utilizing libraries used by the open-source DJ software Mixxx. The software employs the library KeyFinder among other tools to perform a similar musical analysis to that of Spotify to assist DJs in finding harmonically compatible music (Holthuis, 2021).

Picking an audio codec

There are two fundamentally different methods of compressing audio, lossy and lossless. Lossless compressors encode audio in such a way that the original file can be reconstructed fully, whereas lossy encoders rely on psychoacoustics to locate portions of audio that are imperceptible to the human ear, which are then discarded to preserve storage space (Brandenburg, 2000, p. 2). Lossy encoders can therefore produce smaller files and are better suited for streaming over the internet. A set of testing methodologies were made to measure the accuracy of the numerous audio codecs available to consumers in the mid-to-late 1990s by the ITU Radiocommunication Sector (ITU-R), an organization responsible for managing the global use of the radio-frequency spectrum and satellite orbits (ITU, 2022). Recommendation ITU-R BS.1116-1 describes this testing framework. The framework dictates a set of criteria's to be met by the subjects participating in the test and how the listening test, known as the "double-blind triple-stimulus with hidden reference" test, should be performed (ITU-R, 1994-1997, pp. 3-4).

Scientists at the Communications Research Center in Canada used this framework to test the differences between six codec families at different bitrates. In this study, the Advanced Audio Codec (AAC) encoder created by the Fraunhofer Institute for Integrated Circuits performed the best out of all the codec families at a bitrate of 128kbps (Soulondre, et al., 1998). More recently, audio enthusiasts on the internet forum HydrogenAudio have also been using this framework to perform listening tests of modern audio codecs. In a study run by forum user "Kamedo2", 38 participants rated different audio codecs and found that Opus slightly edged out over AAC at a bitrate of 96kbps, ahead of Ogg Vorbis and MP3 (Kamedo2, 2014). For reference, the music streaming service Spotify uses 256kbps AAC for their premium tier subscriptions (Spotify Inc., 2022).

[3] https://navidrome.org/

Based on this information, one could think to use Opus as a primary audio codec for a project like Coral. Opus encoded audio can be packaged into various container formats such as WebM, Ogg and MPEG-4. This can be confusing to work with as browsers may support one container configuration but not another, making Opus an inconvenient choice from a content delivery perspective (CanIUse, 2023). Safari users reported in 2022 that Opus in a WebM container wasn't supported, a combination known to be supported by other browsers (Gullen, 2022). This makes AAC, the second-best performer a better choice as it is natively supported on more platforms.

Picking the right AAC encoder frontend for a given platform

Picking the right AAC encoder frontend for a given platform is a complex task. Not only does the encoder need to run on the platform, but it must also output the AAC stream in the right format (Pantos & May, 2017, p. 8). On Windows, Apple's AAC encoder can be used via *qaac,* a utility made to interface with Apple's CoreAudio library used by iTunes and QuickTime (nu774, 2022). MacOS has a built-in AAC encoder that is exposed via application *afconvert*; however, it cannot write the stream to memory and must commit the file to disk, which makes it impossible to use efficiently in a streaming application. The multi-platform audio and video processing tool *FFmpeg* has among many other things, its own AAC encoder that functions on Windows, MacOS and Linux. However, as previously seen in listening tests and in recommendations from audio enthusiasts on the forum HydrogenAudio, its performance leaves much to be desired and it is preferred to use other encoders where possible (HydrogenAudio, 2017). As FFmpeg supports Apple's encoder on MacOS via their AudioToolbox API, it is possible to use their encoder as opposed to the built-in variant (FFmpeg Developers, 2022). As previously established, Apple's encoder is not easily usable on Linux, therefore the Fraunhofer's encoder is recommended, which is available via the *fdkaac* command-line utility (nu774, 2022) or the *libfdk_aac* library with FFmpeg (FFmpeg Bug Tracker and Wiki, 2022). It is worth mentioning that FFmpeg must be compiled with support for the Fraunhofer AAC encoder by the user themselves as the respective software licenses of FFmpeg and the encoder are incompatible with one another regarding distribution (Free Software Foundation, 2022).

Audio Playback on the Web

HTTP Live Streaming, also known as HLS, was developed by Apple Inc. in 2009. HLS is an efficient and cost-effective protocol to stream audio and video content over the internet. By allowing the receiver to adapt the media's bitrate to the current network conditions, it ensures a seamless playback experience with the best possible quality (Pantos & May, 2017). HLS only supports a few audio codecs for stereo audio, primarily different variations of AAC (Advanced Audio Coding) and FLAC (Free Lossless Audio Codec) (Apple Inc., 2022). Many browsers can playback other types of audio files using the *<audio>* HTML tag, but there are no built-in fallback mechanisms to swap out files should the users' connection speed be too slow (Mozilla Developer Network, 2022). Media Source Extensions (MSE) is a standard written by the World Wide Web Consortium, edited by employees from companies in the content delivery space such as Google and Netflix, to allow external players to use JavaScript to pre-process unsupported media streams for them to be consumable by the browser (W3C, 2022). As HLS can provide the fallback mechanisms needed for a smooth streaming experience on a variable connection, MSE can be used to allow the browser to playback HLS playlists.

Application Architecture Choices

Microsoft recommends that developers architect their web applications using the Clean Architecture pattern in order to achieve dependency inversion (Smith, 2022). This architectural pattern was initially created by Jeffery Palermo in 2008, as “The Onion Architecture”. Palermo places emphasis on separation of concerns throughout the whole system to maintain simple extensibility and testability (Palermo, 2008).

Due to the complexity around figuring out what audio encoders to target on what platforms, an abstraction layer should be made to create a unified API for accessing encoders in a frontend and platform agnostic way.

Design Patterns

Software design patterns are ubiquitous as they are not restricted to certain programming languages and used as starting points which are then expanded upon by the developer (Shvets, 2022). Coral makes extensive use of design patterns, including the Factory, Builder and Façade patterns, to ensure the codebase is highly maintainable. The factory pattern is used where an application needs to interface with systems that fundamentally do the same thing, but their implementations differ.

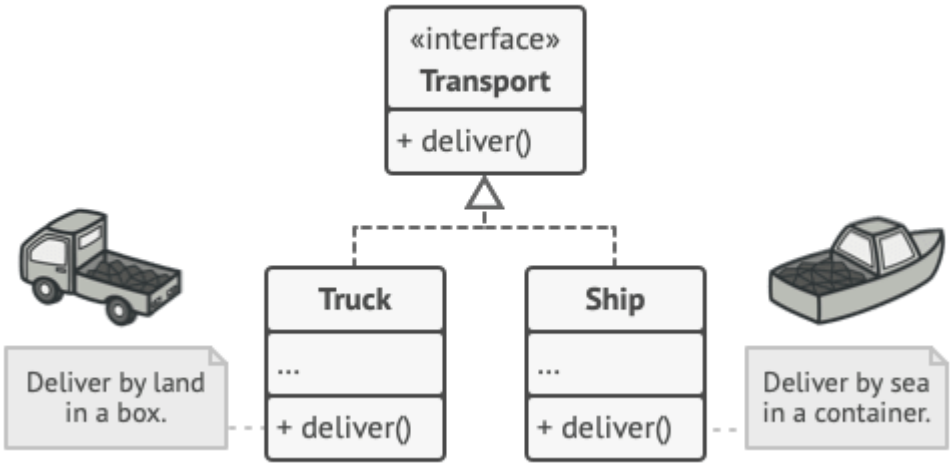

Figure 2 – Common interface example, used in the Factory pattern (Zhart, 2022)

The pattern works by creating a common interface that all the individual components implement such that the underlying implementation of the component is abstracted away from the system. The factory provides a mechanism for classes to request an object matching a criterion, which the factory then creates. In Coral, the factory is also aware of what platform the encoder runs on and what platform it is running on, thereby being able to return the correct encoder implementation for a given platform.

The second pattern, Builder, is used to simplify the configuration of complex objects. The encoder frontends that Coral interfaces with are all command line tools which take in arguments in a specific order. The Builder pattern can be implemented by setting up classes that implement setter functions for each of their properties, which then return themselves such that the setter methods can be chained together. Finally, the builder provides a method that builds the object, in Coral's case, an array with command-line arguments in a specific order.

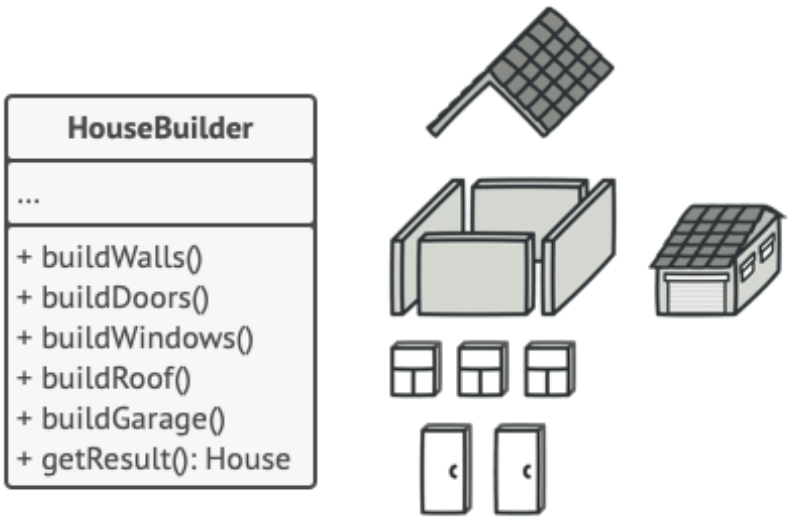

Figure 3 - Example of a class applying the builder pattern (Zhart, 2022)

As design patterns were originally coined in the architecture design world, terms that represent similar things in both worlds commonly use the same name. The façade pattern is one of them. The pattern is used to simplify complex systems into a single usable class. This way the encoder factory and configuration are abstracted away from the main program and the class consuming the façade can call a single function that takes care of the encoding process in a platform agnostic way.

Conclusion

By utilizing musical analysis data within Coral, users can be in control of their music library while also enjoying the experience of a commercial streaming platform. Using the test framework created by ITU, it was established that the lossy stereo codecs Opus and AAC performed better than their competition, achieving source transparency at a low bitrate. However, AAC was chosen as the primary audio codec to be used by Coral for its superior browser and system compatibility. AAC can be delivered via Apple's streaming protocol HLS, which can adapt to slower connections by automatically changing the quality of the content being streamed. Coral uses a set of different AAC encoders, dependent of the platform it is running on and uses the Factory pattern to utilize the appropriate one at runtime.

System Design

Backend Architecture

Coral's overall project structure is architected following the Clean Architecture pattern. Each application component has been split into a C# project, ensuring high testability. This architecture pattern pushes developers to think about how the components should interact with one another to achieve low coupling.

The Model–View–Viewmodel (MVVM) pattern is used to have more control over the data flowing through the application. It is possible to create smaller models that contain just the information that is relevant to the operation being performed called Data Transfer Objects (DTO) or view models. This means that we have control over what data we send to the client by explicitly creating the DTO models as well as the data we serialize back from the user. An example of this is handling user information. Using automatic object mappers such as AutoMapper, it is possible to ensure that outgoing API requests should omit sensitive fields such as the user's password hash. It is also possible to enforce rules on incoming requests from the user. An example of this would be preventing a user from modifying server-managed timestamps.

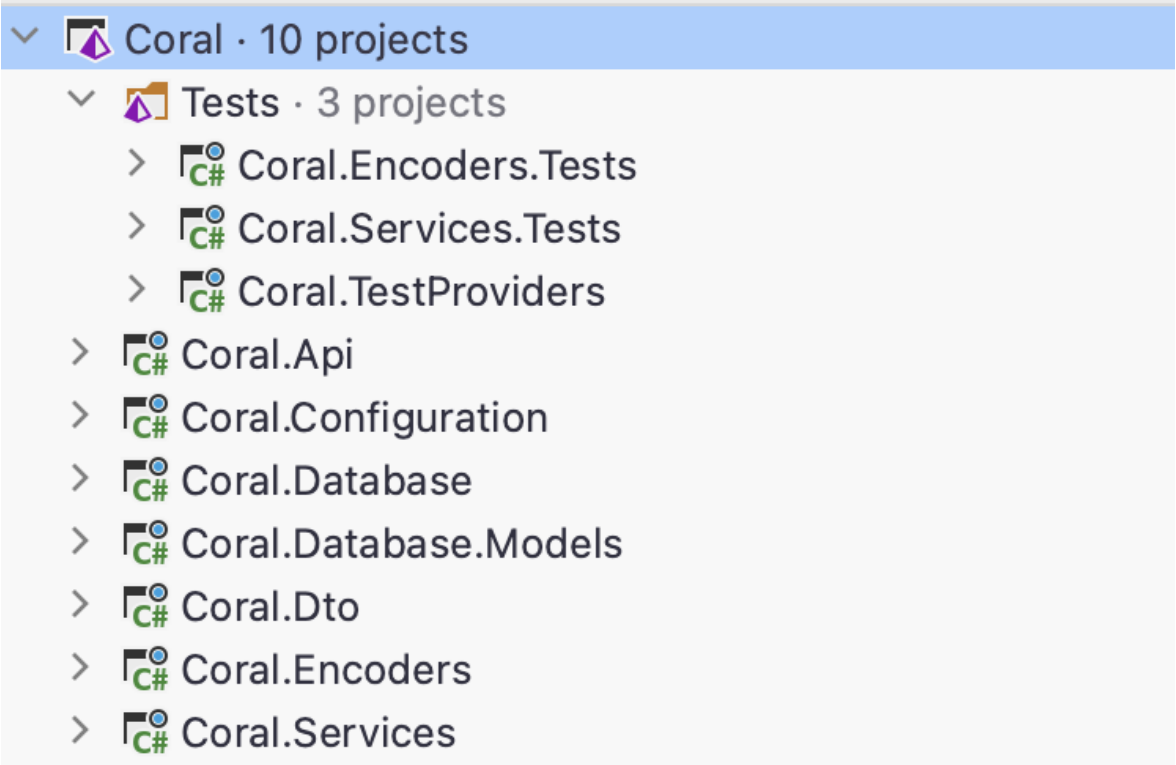

Figure 4 - Backend codebase structure

To utilize a class originating from another C# project, a project reference must be made. This creates dependency chain that is easy to visualize. In this diagram, the purple lines indicate a single project reference between the two nodes. The API project, a REST API powered by ASP.NET Core, serves as the primary point of entry for users. The Services project contains all the business logic used in the application, such as the content indexer, audio transcoder and content library. The audio transcoder relies on the encoder factory which is configured by classes in the Encoders project. The Services project also returns DTOs for every public method used by the API. The DTO project depends on the database models, as it needs to know what data sources it can map data from. Finally, the database is configured via the Configuration project.

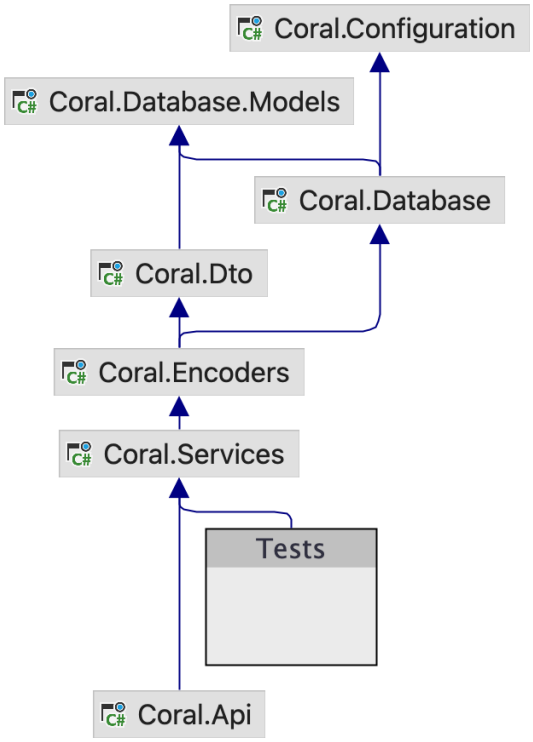

Figure 5 - Backend dependency chain

Coral's main backend components include a content indexer, content packaging system that creates HLS playlists for the browser and an audio transcoder. The content indexer is responsible for storing the metadata of all the music present in a user's library in an SQLite database. This is done to ensure highly performant collection queries and fast application startup times. The database is managed via Microsoft's database ORM, Entity Framework Core. The core component of the content indexer is a third-party library that parses audio files of different formats called "Audio Tools Library for .NET" (Zeugma440, 2022). The content indexer is aware of file system changes and can read both new and existing files as soon as they are changed. The database schema contains the minimum amount of information needed to organize and display a simple music collection, which can be expanded via database migrations in the future.

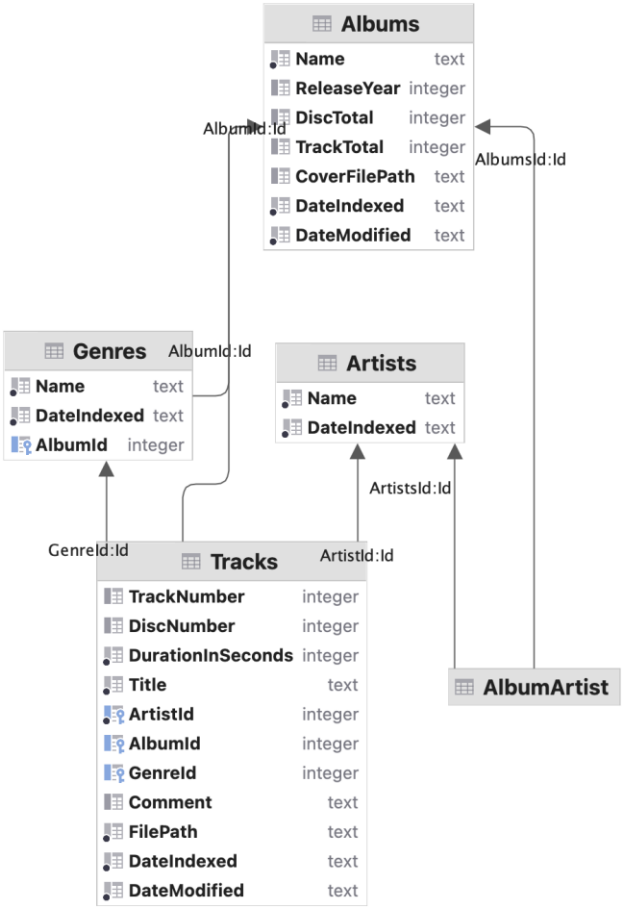

Figure 6 - Database ER Diagram

The content packaging system uses the media processing application FFmpeg to generate HLS playlists while the audio file is being transcoded, to deliver a real-time listening experience to the end user. The stream is now playable within a second instead of waiting for an entire audio file to finish converting - which depending on processing performance of the system Coral is running on, can take some time. This is achieved by running the playlist generator and audio transcoder simultaneously. The playlist generator streams data from the audio transcoder and writes an HLS playlist using the playlist type *EVENT*. The playlist type allows the packaging system to create a playlist that is playable by the browser as soon as the first segment is made. Then, once the processing and transcoding process is complete, the final duration of the transcoded file is announced to the player and the whole file can be played from start to finish.

Frontend Architecture

The frontend is written in TypeScript with Next.js, which is built on React. It uses Zustand for state management and Axios as its HTTP client. In order to save development time, the backend is configured to generate an OpenAPI specification which the frontend can use to generate its API client with. This means that any backend changes are automatically implemented in the frontend, however any abstractions created over the generated API client must updated when existing functionality changes.

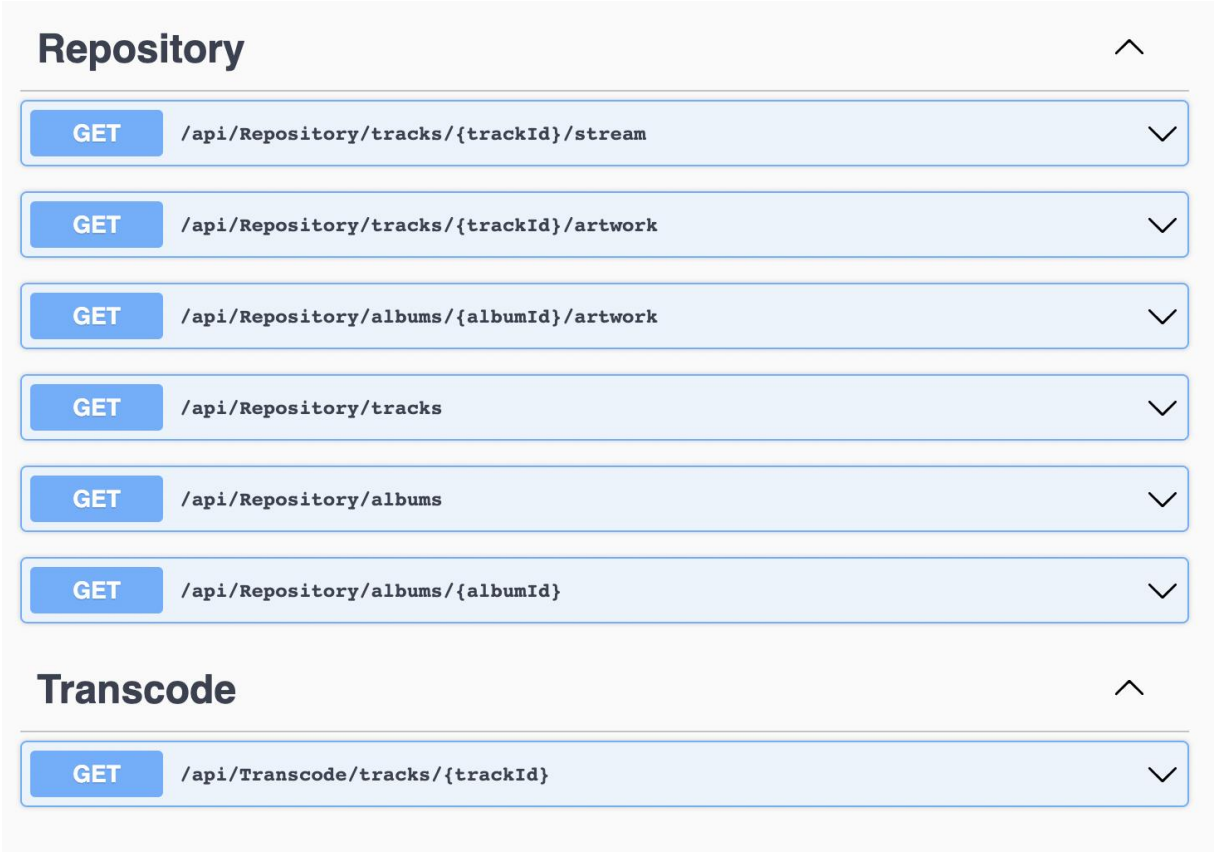

Figure 7 - API routes displayed via Swagger UI

Individual React components are assembled to make up web pages. In this example, the Album page combines an album information, playlist, and media player component to make up the album page, which can be seen in Figure 17.

```
export default function Album() {
  const [album, setAlbum] = React.useState({} as AlbumDto);
  const router = useRouter();
  let { id } = router.query;
  React.useEffect(() => {
    const getAlbum = async () => {
      let targetId = id != null ? +id : 1;
      let targetAlbum = await RepositoryService.getAlbum(targetId);
      if (targetAlbum != null) {
        setAlbum(targetAlbum);
      }
    };
    getAlbum();
  }, [id]);

  const Player = dynamic(() => import("../../components/Player"), {
    ssr: false,
  });

  return (
    <div>
      <AlbumInfo album={album}></AlbumInfo>
      <Playlist tracks={album.tracks}></Playlist>
      <Player tracks={album.tracks}></Player>
    </div>
  );
}
```

Figure 8 - Album page code

Functional requirements

Coral's web interface will be the primary method of user interaction. As such, the user interface must be able to control every aspect of the application, from configuring the indexer and transcoding system to managing a music library. A common feature in many music players and streaming platforms is the ability to browse a library via audio metadata present on tracks in the library.

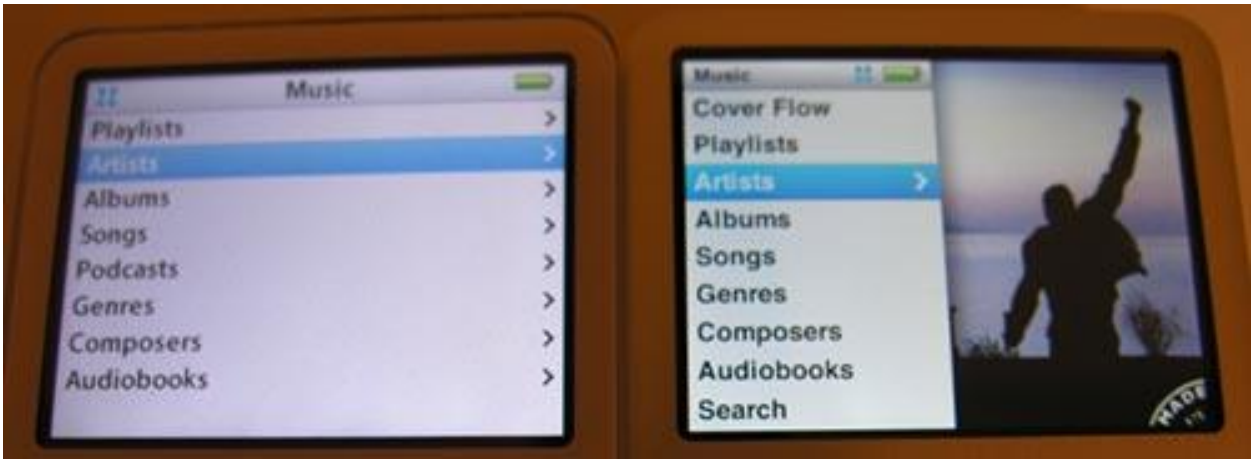

Figure 9 - iPod user interface showing common library items (Strietelmeier, 2007)

Coral will only be used by the user that has configured the system. This is because UK law mandates that digital copies of copyrighted media shall be consumed by none other than its owner (Copyright, Designs and Patents Act 1988 - Part I, c. III, Section 28B). However, even though the system may only have a single user, the system still requires an authentication mechanism as it will be exposed to the internet. With these things in mind, it is possible to sketch out a use case sheet. In order to rank the importance of each requirement, they have been structured following the MosCoW (must have, should have, could have, won't have) methodology.

Must have

- Create an account
- Configure the content indexer, pointing it to their music library on disk
- Browse their library via albums, artists and tracks
- Playback tracks
    - In their original format
    - Transcoded to AAC by Coral
- Create playlists
- Control the player using media keys on their keyboard or system-wide media controller
- Search for their music

Should have

- Update their collection on the fly and be able to play new music in Coral once scanned
- Monitor scanner status and request re-scans on demand
- Playback logging to Last.fm, a service that let users log what they listen to across multiple platforms

- Listen to dynamically created playlists
    - Unheard music
    - Genre mixes
    - Artist mixes

Could have

- Use data from Spotify to supplement the library metadata (picture of artist, biography, album artwork if it doesn't already exist, etc.)
- Use data from Spotify to create recommendations for music indexed by Coral
- A notification system to let the user know of system events (new library additions, recent failures, failed login attempts)
- Quick access to music through a simple menu that can be triggered by a keyboard shortcut (think: Spotlight on MacOS)
- Modify the metadata of music in their collection
- A listening test to determine which AAC bitrate sounds the best to the user
- Support for other lossy codecs

Won't have

- A mobile-compatible version of the website
- Integration with the KeyFinder library used by Mixxx to provide a harmonic recommendation for music unavailable to Spotify

Test strategy

I'm developing Coral's backend following the test-driven development (TDD) method along with the Arrange-Act-Assert test pattern, following Microsoft's recommended testing practices[4]. The unit tests are written before the functionality is implemented, ensuring that you write code that behaves like you would expect rather than a potentially incorrect implementation. Another benefit to TDD is high code coverage due to the nature of writing tests first. High code coverage and using continuous integration services such as GitHub Actions to validate your latest changes can help reduce the chance of regressions before the code is deployed and allows you to deploy changes with confidence. However, the TDD approach starts to fall apart once you are attempting to prototype a system without a clear specification. It is hard to write tests for a system you're not sure how works yet - which Is why I have initially decided not to write automated tests for the frontend as it is changing rapidly. It is not worth investing the additional development time writing a test suite for a system that can change at any minute.

Example of a backend test – content indexer

There are many ways to organize a music collection. Collections can be sorted via albums, artists, genres, release date and so on. Some also choose to dump their music into a single folder and call it a day. This means that Coral will need to accommodate for different types of music collections, both pristinely organized and tagged as well as files without any metadata stored in the same folder as many other unrelated tracks. To accommodate these scenarios in my test data, I have created a test library without any musical content to be used while testing the content indexer.

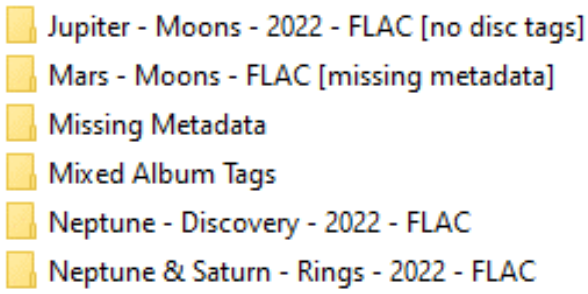

Figure 10 – A sample of the test music library

Coral uses the libraries xUnit for unit-testing along with NSubstitute for mocking services. I have also implemented an in-memory database that exists with the lifetime of the test, in order to validate indexing behavior using EF Core via SQLite. See Figure 11 for a content indexer test.

[4] https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

```
[Fact]
public async Task ReadDirectory_JupiterMoons_CreatesValidMetadata()
{
    // arrange

    // act
    _indexerService.ReadDirectory(TestDataRepository.JupiterMoons);

    // assert
    var jupiterArtist = await _testDatabase.Artists.FirstOrDefaultAsync(a => a.Name == "Jupiter");
    var moonsAlbum = jupiterArtist?.Albums.FirstOrDefault();

    Assert.NotNull(jupiterArtist);
    Assert.NotNull(moonsAlbum);

    Assert.NotNull(moonsAlbum.CoverFilePath);
    Assert.Equal(3, moonsAlbum.Tracks.Count);

    var metisTrack = moonsAlbum.Tracks.Single(t => t.Title == "Metis");
    Assert.Equal(1, metisTrack.TrackNumber);
    Assert.Equal("Noise", metisTrack.Genre?.Name);
    Assert.Equal("this has a comment", metisTrack.Comment);
}
```

Figure 11 - Example of an indexer test

Test case table

ID	Part of System	User Action	Expected Result	Success
001	Album page	Navigates to an album	Sees album name, artist name, number of tracks. Release date and genre are also available if supplied by the user. Album tracks are listed below the metadata header and can be interacted with.	Yes
002	Album page	Clicks on artist name in header	Gets redirected to artist page	No
003	Album playlist	Double clicks on song in album playlist	Coral plays song and announces media playback to the browser	Yes
004	Album playlist	Plays song	Song is highlighted in the playlist, to show that it is being played, artwork is displayed by the player	Yes
005	Album playlist	Song is hovered over with a mouse	Track number is replaced with a play button	Yes
006	Album playlist	Attempts to play an unavailable song	Song is grayed out and upon hover, a popup is shown showing why the song is unplayable	No
007	Player	Clicks next song	Next song in the queue is played	Yes
008	Player	Clicks pause	Pauses song	Yes
009	Player	Clicks previous	Goes to the previous song in the queue	Yes

010	Player	Interacts with player via the MediaSession API (in-browser controls or system-wide media controls)	Commands are sent to main player and actions are executed.	Yes
011	Player	Clicks on a point in the seek bar	Player seeks to that point in the song	Yes
012	Scanner	Adds new music to their collection and reboots Coral	Scanner picks up the change and adds the content to their library	Yes
013	Player	Requests a transcode of a song on MacOS	The song is encoded with FFMpeg using the AudioToolbox-based AAC encoder	Yes
014	Player	Requests a transcode of a song on Windows	The song is encoded by qaac	Yes
015	Player	Requests a transcode of a song on Linux	The song is encoded by fdkaac	No
016	Player	Requests the file with no encoding performed	The song is read from disk and delivered as-is	Yes
017	Indexer	Points the indexer to their music library	The indexer reads the library and writes track metadata to a database	Yes
018	Indexer	Indexes album with a single new artist	A new artist and album are created, and they are referenced to each other	Yes
019	Indexer	Indexes album with no metadata	The indexer uses its parent folder's name as the album title	Yes
020	Indexer	Indexes an album with two artists, one that exists from before and one new	The album is created along with any new artists that exist on it. The existing artist is also found and linked with the same album	Yes
021	Indexer	Indexes an album with the same name as one that already exists	The album is created and has no reference to the existing one.	Yes
022	Indexer	Performs an unexpected action that leads to an error	Indexer broadcasts status and continues scanning other items	No
023	Login page	Submits incorrect credentials	User is not granted access and an error message is shown	No
024	Login page	Submits correct credentials	User is granted access and the home page is shown	No

User Interface

My goal with designing Coral's user interface is that it should feel familiar from the moment you use it. The frontend is built using the React component library Mantine, which provides pre-styled buttons, forms, dropdowns and other components needed to build a web application. However, the site layout will be written without any frameworks in plain CSS. Over the years, a set of patterns have emerged to set the standard for what a music player on the web should look like. In order to understand what I needed to implement design-wise, I used a couple media players and observed their behaviors and stylistic approaches before landing on what I chose for Coral.

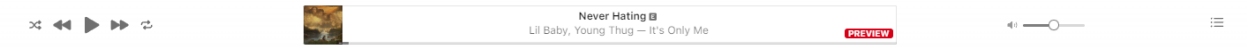

Figure 12 - iTunes media controls

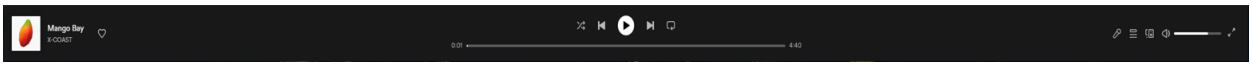

Figure 13 - Spotify media controls

Figure 14 - Amazon Music media controls

There are various approaches one can take to a create a media player with regards to how the controller elements should be placed. Spotify's approach clearly shows the seek bar and the playlist control buttons while also showing the metadata clearly. I preferred their approach over Amazon and Apple's as they didn't show the player's current position, nor made the seeker easily accessible. The player is still in its prototype phase and is lacking a volume slider and playlist management controls.

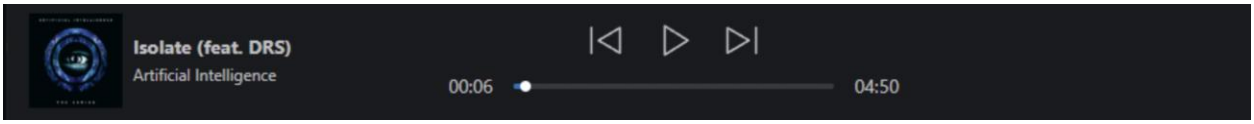

Figure 15 - Coral's media controls as of November 2022

Implementation Report

I have been keeping track of my progress and findings in my Notion notebook and I've used the Issue page in my GitHub repository to monitor what features I would like to have in Coral. I created user stories for each feature in my application and prioritized them by creating monthly milestones. For the entirety of November I focused on implementing the core functionality of the application such as the indexer, transcoder and player for the frontend and completed 76% (13 out of 17) of the tasks I set out to do. However, it is worth mentioning that the tasks remaining were deemed less urgent - and longer, complex tasks were prioritized instead. I also had the flu towards the end of the month, which knocked me out for a week.

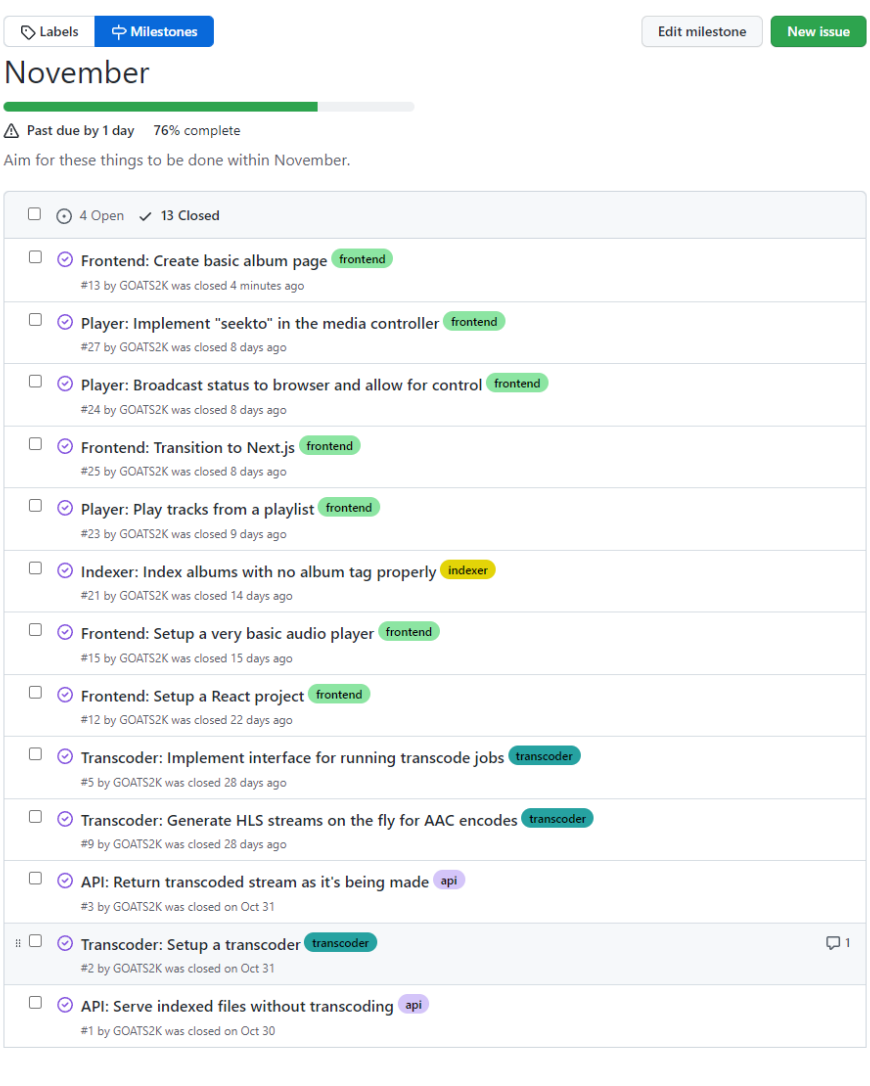

Figure 16 - Completed tasks in November

So far, Coral can index music collections specified by the user via an environment variable, transcode audio to AAC on MacOS and Windows using FFMpeg and qaac, and playback albums in a web browser. There is an album page that can display the current album name and artwork, artist, duration and genre if present, with a playlist and music player. The player can be used to skip or seek through tracks as well as show the currently playing song and its duration and artwork.

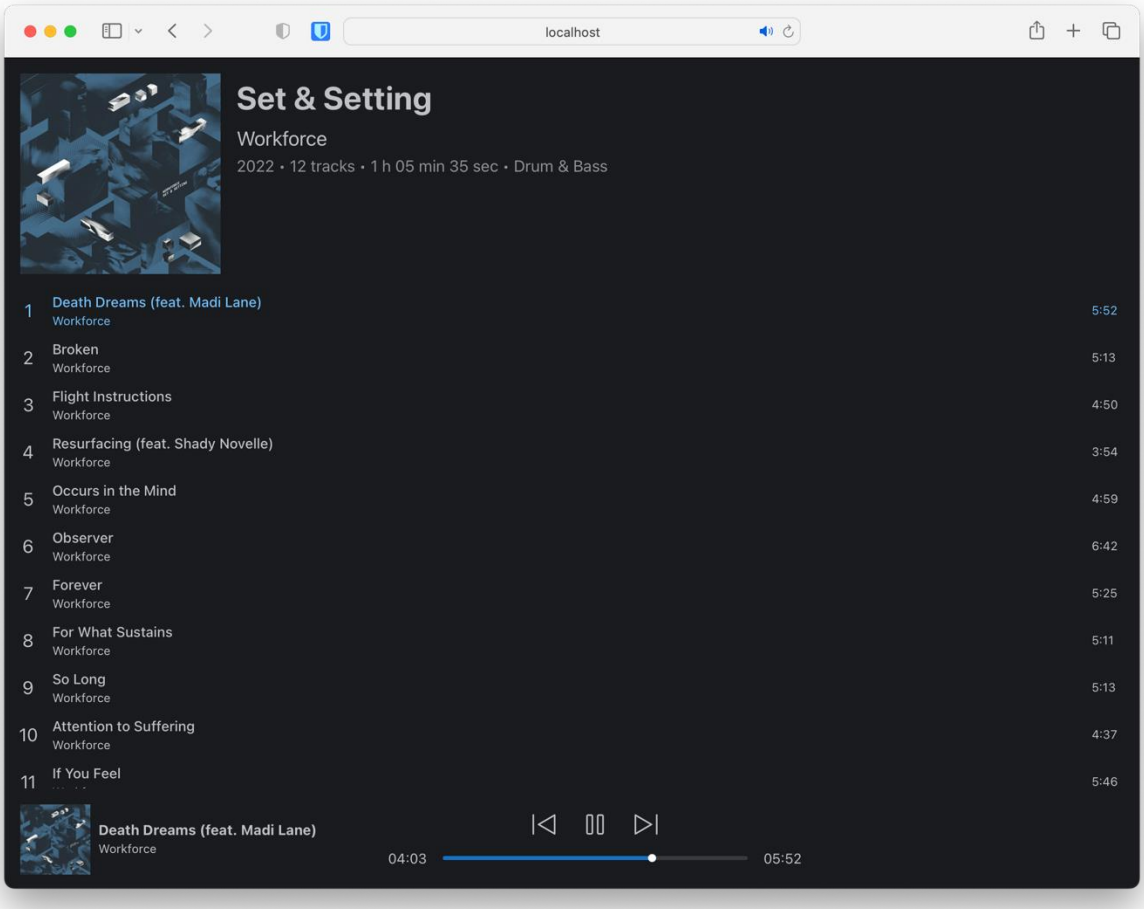

Figure 17 - UI as of December 2022

Bibliography

Apple Inc., 2022. *Apple Developer Documentation - HTTP Live Streaming (HLS) Authoring Specification for Apple Devices.* [Online]
Available at:
https://developer.apple.com/documentation/http_live_streaming/http_live_streaming_hls_authoring_specification_for_apple_devices
[Accessed 08 12 2022].

Brandenburg, K., 2000. *MP3 and AAC Explained.* [Online]
Available at: https://graphics.ethz.ch/teaching/mmcom12/slides/mp3_and_aac_brandenburg.pdf via https://web.archive.org/web/20170213191747/https://graphics.ethz.ch/teaching/mmcom12/slides/mp3_and_aac_brandenburg.pdf
[Accessed 13 December 2022].

CanIUse, 2023. *Opus audio format | Can I use... Support tables for HTML5, CSS3, etc.* [Online]
Available at: https://caniuse.com/opus
[Accessed 10 January 2023].

Copyright, Designs and Patents Act 1988 - Part I, c. III, Section 28B, 2014. *Copyright, Designs and Patents Act 1988.* [Online]
Available at: https://www.legislation.gov.uk/ukpga/1988/48/section/28B

FFmpeg Bug Tracker and Wiki, 2022. *Encode/AAC.* [Online]
Available at: https://trac.ffmpeg.org/wiki/Encode/AAC#fdk_aac
[Accessed 19 December 2022].

FFmpeg Developers, 2022. *GitHub.* [Online]
Available at: https://github.com/FFmpeg/FFmpeg/blob/master/libavcodec/audiotoolboxenc.c#L231
[Accessed 15 December 2022].

Free Software Foundation, 2022. *Various Licenses and Comments about Them - GNU Project - Free Software Foundation.* [Online]
Available at: https://www.gnu.org/licenses/license-list.html#fdk
[Accessed 15 December 2022].

Gullen, A., 2022. *WebKit Bugzilla.* [Online]
[Accessed 10 January 2023].

Holthuis, J., 2021. *Mixxx.* [Online]
Available at: https://mixxx.org/news/2021-04-08-new-in-2-3-keyfinder/
[Accessed 11 January 2023].

HydrogenAudio, 2017. *AAC Encoders.* [Online]
Available at: https://wiki.hydrogenaud.io/index.php?title=AAC_encoders
[Accessed 14 December 2022].

IgorC, 2011. *Results of the public AAC listening test @ 96 kbps (July 2011).* [Online]
Available at: https://listening-tests.hydrogenaud.io/igorc/aac-96-a/results.html
[Accessed 14 December 2022].

ITU, 2022. *Welcome to ITU-R.* [Online]
Available at: https://www.itu.int/en/ITU-R/information/Pages/default.aspx
[Accessed 13 December 2022].

ITU-R, 1994-1997. *Rec. ITU-R BS.1116-1 - METHODS FOR THE SUBJECTIVE ASSESSMENT OF SMALL IMPAIRMENTS.* [Online]
Available at: https://www.itu.int/dms_pubrec/itu-r/rec/bs/R-REC-BS.1116-1-199710-S!!PDF-E.pdf
[Accessed 13 December 2022].

Kamedo2, 2014. *Results of the public multiformat listening test (July 2014).* [Online]
Available at: http://listening-test.coresv.net/results.htm
[Accessed 8 12 2022].

Mozilla Developer Network, 2022. *<audio>: The Embed Audio element.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/audio
[Accessed 8 12 2022].

nu774, 2022. *nu774/fdkaac: command line encoder frontend for libfdk-aac.* [Online]
Available at: https://github.com/nu774/fdkaac
[Accessed 19 December 2022].

nu774, 2022. *nu774/qaac: CLI QuickTime AAC/ALAC encoder.* [Online]
Available at: https://github.com/nu774/qaac
[Accessed 15 December 2022].

Palermo, J., 2008. *Programming with Palermo.* [Online]
Available at: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
[Accessed 16 December 2022].

Pantos, R. & May, W., 2017. *RFC 8216: HTTP Live Streaming.* [Online]
Available at: https://www.rfc-editor.org/rfc/rfc8216

Pastukhov, D., 2022. *Music Tomorrow.* [Online]
Available at: https://www.music-tomorrow.com/blog/how-spotify-recommendation-system-works-a-complete-guide-2022
[Accessed 10 January 2023].

Reese, J., 2022. *Best practices for writing unit tests - .NET | Microsoft Learn.* [Online]
Available at: https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

Shvets, A., 2022. *refactoring.guru.* [Online]
Available at: https://refactoring.guru/design-patterns/what-is-pattern
[Accessed 15 December 2022].

Smith, S., 2022. *GitHub.* [Online]
Available at: https://github.com/dotnet-architecture/eBooks/tree/main/current/architecting-modern-

web-apps-azure
[Accessed 16 December 2022].

Soulondre, G. A., Grusec, T., Lavoie, M. & Thibault, L., 1998. *Subjective Evaluation of State-of-the-Art 2-Channel Audio Codecs.* Amsterdam: Audio Engineering Society.

Spotify Inc., 2022. *Audio Quality.* [Online]
Available at: https://support.spotify.com/us/article/audio-quality/
[Accessed 8 12 2022].

Spotify, 2023. *Spotify for Developers.* [Online]
Available at: https://developer.spotify.com/documentation/web-api/reference/#/operations/get-recommendations
[Accessed 10 January 2023].

Spotify, 2023. *Spotify for Developers.* [Online]
Available at: https://developer.spotify.com/documentation/web-api/reference/#/operations/get-audio-analysis
[Accessed 10 January 2023].

Strietelmeier, J., 2007. *Apple iPod classic.* [Online]
Available at: https://the-gadgeteer.com/assets/apple-ipod-classic-7.jpg

W3C, 2022. *Media Source Extensions.* [Online]
Available at: https://www.w3.org/TR/2022/WD-media-source-2-20220921/
[Accessed 8 December 2022].

Zeugma440, 2022. *'Zeugma440/atldotnet: Fully managed, portable and easy-to-use C# library to read and edit audio data and metadata (tags) from various audio formats, playlists and CUE sheets'.* [Online]
Available at: https://github.com/Zeugma440/atldotnet

Zhart, D., 2022. *Refactoring Guru.* [Online]
Available at: https://refactoring.guru/design-patterns/builder
[Accessed 16 December 2022].