TROY UNIVERSITY

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF MATHEMATICS AND INFORMATICS



FINAL GROUP PROJECT REPORT

Project Topic: Develop a Mobile Application Using Dart: Musik – Music Player

Supervisor: **Assoc. Prof. Dr. Nguyen Dinh Han**

Course: **CS-3370 – Nature of Program Languages**

Group: **[GROUP]**

Students:

1. [STUDENT_1] [ID_1]
2. [STUDENT_2] [ID_2]
3. [STUDENT_3] [ID_3]
4. [STUDENT_4] [ID_4]
5. [STUDENT_5] [ID_5]

HANOI, June 2026

Troy University Resources

Overview

The project aims to demonstrate how Dart can be used to build a realistic mobile application that combines user interface design, media access, audio playback, state management, and asynchronous event handling. Rather than presenting Musik only as a product demonstration, this report analyzes the application as a case study of programming language use in a real software development context.

Musik is designed to provide a simple and responsive music listening experience on mobile devices. The application focuses on common music-player functions such as accessing audio files or media metadata, displaying song information, controlling playback, and updating the interface according to the current playback state. These requirements make a music player a suitable project for studying Dart, because the application naturally involves object-oriented modeling, null-safe data handling, asynchronous programming, reactive streams, and modular software organization.

Dart was selected as the main programming language because its features fit the technical requirements of the project. The language supports strong typing with type inference, sound null safety, generic collections, Future-based asynchronous programming, and Stream-based reactive data flow. These features are especially useful in a music player, where media information may be incomplete, playback events occur continuously, and the user interface must respond quickly to state changes. Flutter complements Dart by providing a declarative UI framework, allowing the interface to be rebuilt consistently whenever the application state changes.

Through this project, Dart language features studied in the course are applied to a real software system, especially in the areas of object-oriented decomposition, asynchronous execution, reactive programming, type safety, and maintainable application structure.

Summary of Contributions

[STUDENT_1] – [ID_1] (Rating: 10/10)

[STUDENT_1] served as the team leader and was responsible for the overall coordination, technical integration, and final quality control of the Musik application.

- Participating in almost all stages of the project, including initial planning, environment setup, playback implementation, UI coordination, playlist features, search support, testing, documentation, and final presentation preparation.
- As the main coordinator, [STUDENT_1] monitored weekly progress, reviewed sprint outcomes, helped rebalance tasks when needed, and ensured that separate modules could be integrated into a consistent application flow.
- His responsibilities included supporting the implementation of the playback flow, seek bar and timing display, shuffle and repeat states, playlist actions, local storage integration, search behavior, YouTube-related flow, and background playback behavior. He also contributed to regression testing, bug fixing, screenshot collection, and preparation of report evidence.
- His main deliverables include the integrated Musik build, feature verification checklist, testing evidence, application screenshots, and coordination notes for the final report and presentation.

[STUDENT_2] – [ID_2] (Rating: 8.5/10)

[STUDENT_2] was responsible for the data-oriented and user-library features of the Musik application.

- His main work focused on building the music catalog foundation, designing the data model, managing playlists, handling local storage, listening history, liked songs, genre/language filtering, and unified history behavior.
- These tasks formed the basis for how Musik stores, organizes, and retrieves user-related music information during normal use.
- [STUDENT_2] contributed to defining and maintaining important models such as songs, artists, playlists, history items, and local cache mappings.
- He also worked on playlist functions, including creating, renaming, deleting playlists, adding songs to playlists, and removing songs from playlists. In addition, he helped persist user data through local storage so that user-related information could remain available across sessions.

[STUDENT_3] – [ID_3] (Rating: 8.5/10)

[STUDENT_3] was responsible for project setup, requirement clarification, architecture planning, and several core implementation tasks in the Musik application.

- At the beginning of the project, he helped clarify the project scope, define the MVP checklist, configure the Flutter project, and establish the initial repository structure and coding conventions.
- He also contributed to important architectural decisions, especially how the application should separate navigation, player logic, data access, and UI modules.
- During the implementation stage, [STUDENT_3] worked on the navigation framework, local music scanning foundation, core playback engine, seek bar logic, playback modes, background playback, and YouTube playback flow.

Troy University Resources

- He also supported integration testing, bug fixing, and final build consolidation to make sure the main features worked together smoothly. In addition, [STUDENT_3] contributed to the final written report and project presentation.

[STUDENT_4] – [ID_4] (Rating: 8.5/10)

[STUDENT_4] was responsible for supporting the technical infrastructure, search-related features, API integration, and optimization work of the Musik application.

- He contributed to environment setup, architecture planning, model design, player engine support, local storage integration, and internal demos.
- His work helped ensure that Musik could handle both local music data and external content sources in a more flexible and reliable way. [STUDENT_4]'s main technical responsibilities included implementing search by song name and artist, improving catalog loading behavior, and supporting thumbnail/cache optimization.
- He also worked on integrating YouTube Data API v3, parsing API responses, handling empty or failed request states, and connecting YouTube-related data into the listening flow.
- In addition, [STUDENT_4] supported regression testing, bug fixing, performance improvement, final build consolidation, and final testing.
- His main deliverables include the search module, API response-handling logic, optimization work, YouTube integration support, and testing evidence for mixed local and online content flows.

[STUDENT_5] – [ID_5] (Rating: 8.5/10)

[STUDENT_5] was responsible for user interface implementation and visual interaction flow within the Musik application.

- His work focused on transforming the planned screen structure into usable Flutter interfaces.
- The main screens he worked on included the Home screen, Search screen, Now Playing screen, Playlist-related screens, song list display, artist information display, and mini-player layout.
- He also contributed to UI communication, layout planning, routing, and polishing the final user experience.
- [STUDENT_5] supported the development of the bottom navigation, application theme, catalog UI, seek bar presentation, playlist interface, search UI, autocomplete behavior, album views, and genre views.
- He also made visual refinements across the Home, Search, and Player screens to improve consistency and usability.
- In the final stages, he helped collect screenshots, prepare documentation materials, run functional tests, and rehearse the group presentation.
- His main deliverables include the main UI screens, navigation-facing interface components, player and playlist interaction screens, screenshots for the final report, and presentation support materials.

Contents

Overview **2**

Summary of Contributions **3**

[STUDENT_1] – [ID_1] (Rating: 10/10) . . . 3

[STUDENT_2] – [ID_2] (Rating: 8.5/10) . . . 3

[STUDENT_3] – [ID_3] (Rating: 8.5/10) . . . 3

[STUDENT_4] – [ID_4] (Rating: 8.5/10) . . . 4

[STUDENT_5] – [ID_5] (Rating: 8.5/10) . . . 4

Introduction **7**

1. Project Requirements . . . 10
2. The Concerning Problem in Technical Terms . . . 15
3. Programming Language Selection Criteria . . . 17

Methodology **20**

1. Architecture Options Considered . . . 21
2. Chosen Component Model . . . 23
3. Implementation of Selected Packages in the Software . . . 25
4. Repository-Driven Code Structure . . . 29
5. Process Flow of a Typical Playback Action . . . 32
6. Use of Selected Programming Language Features in Implementation . . . 33
7. Build, Run, and Test Method . . . 37

Results and Discussion **39**

1. Obtained Functional Results . . . 39
2. Analysis of Language Use and Software Design Ability . . . 41
3. Design Pattern Discussion . . . 42
4. Testing Results and Quality Discussion . . . 43
5. Final Technical Judgment . . . 44

References **45**

Appendices **47**

Appendix A. Project Work Plan and Contribution Tracking . . . 47

Appendix B. Repository Structure and Dependency Evidence . . . 49

Appendix C. Testing and Manual Validation Evidence . . . 50

Troy University Resources

List of Tables

1 Functional requirements and quality requirements of Musik 10
2 Use Case Summary 12
3 Dart and Flutter Suitability Assessment for Musik 19
4 Comparison of Architecture Options for the Musik Application 21
5 Packages selected for Musik 25
6 Repository Structure and Implementation Evidence for Musik 29
7 Functional results of Musik 39
8 Quality-assurance strategy for Musik 44

Introduction

The project presented in this report is Musik, a cross-platform mobile music player application developed using Dart and Flutter. The main purpose of the project is to demonstrate how Dart can be applied to the development of a realistic mobile application, rather than only to present a visual product demo. Through the implementation of Musik, the project examines how programming language concepts such as object-oriented design, asynchronous programming, reactive data flow, type safety, and modular software organization can be used in a practical software system.

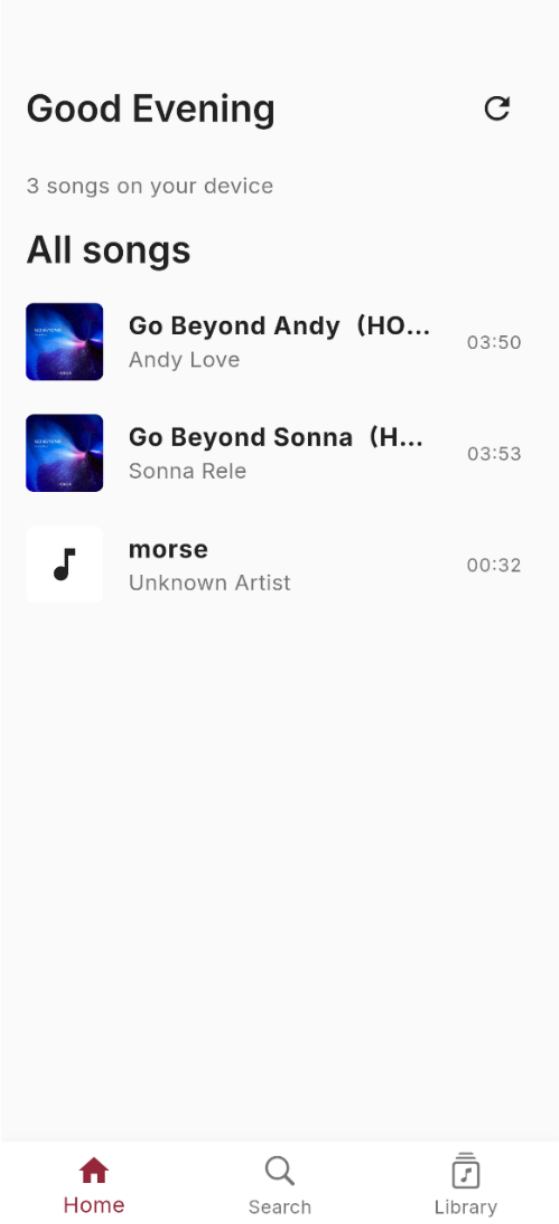
A music player is an appropriate case study for this course because it combines several technical concerns that are simple from the user's perspective but complex from the implementation perspective. The application needs to manage user interface rendering, local media access, song metadata, playback controls, playback progress, queue behavior, permissions, and application lifecycle events. These requirements create a suitable context for analyzing how Dart and Flutter support the development of maintainable and responsive mobile software.

The main problem addressed by Musik is how to build a responsive, maintainable, and extensible music player from a single codebase while handling media access and playback state in a clean and reliable way. Functionally, the application is expected to discover or consume audio content, display song information, support basic playback operations, and present playback progress and queue state to the user. Non-functionally, the application should remain readable, robust against null-related and asynchronous errors, easy to test, and adaptable for future features. Dart and Flutter are well suited to these goals because Flutter provides a declarative user interface model, while Dart provides strong typing, generic collections, asynchronous programming support, Stream-based data flow, and sound null safety. [6]–[10]

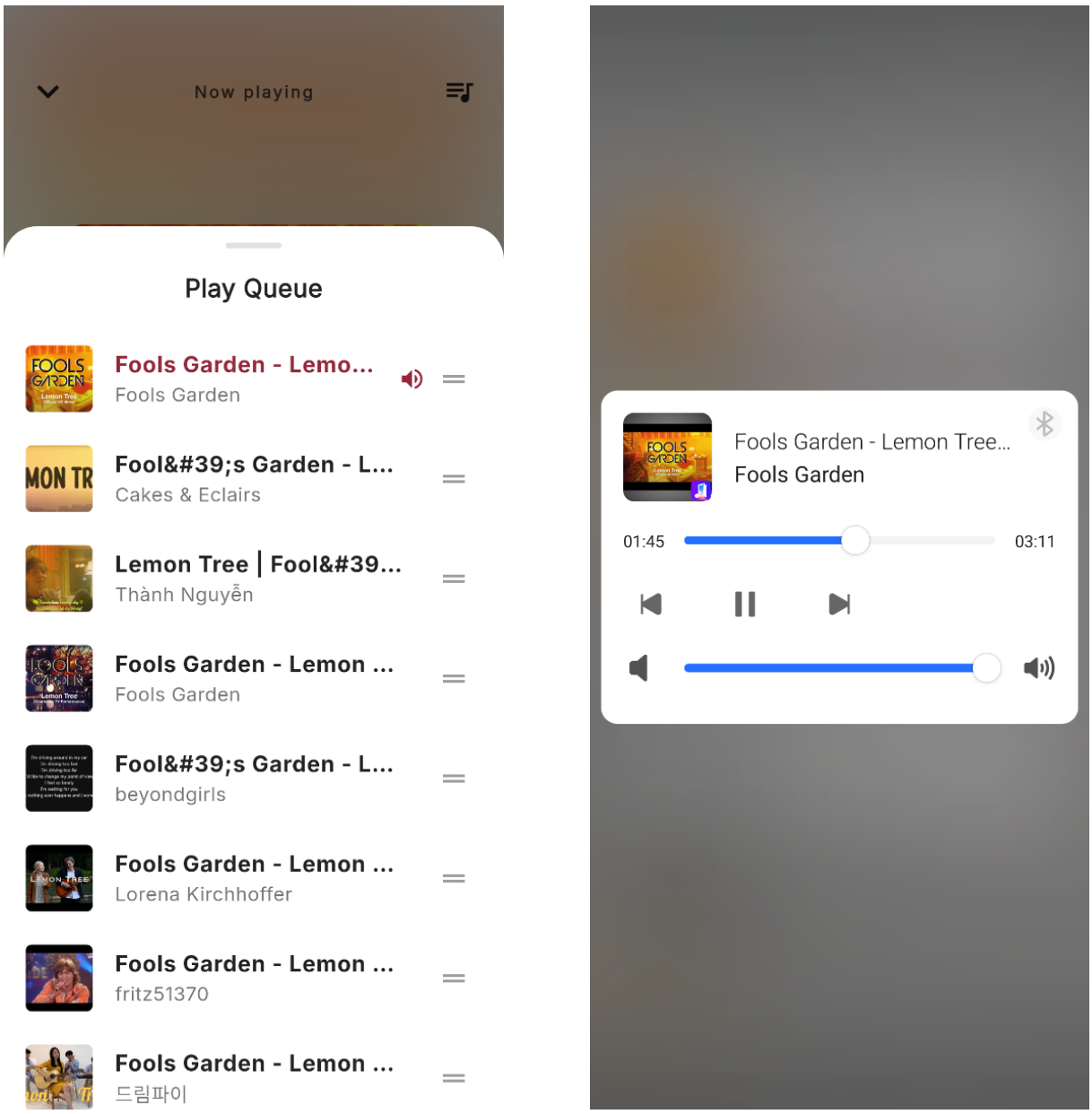
From the perspective of programming language use, Musik demonstrates several programming styles within one application context. Object-oriented programming is used to organize domain entities, service classes, and application components. Imperative control is used where sequential operations are required, such as initializing services, requesting permissions, or executing playback commands. Event-driven and reactive programming appear in the handling of playback state, progress updates, and user interface rebuilding. Functional-style operations can also be used for transforming song lists, filtering search results, and passing callbacks between widgets. These characteristics make Musik a suitable project for analyzing Dart as a modern multi-paradigm programming language.

In terms of product scope, Musik is designed as a Flutter-based mobile music player that can organize music information and provide playback through a reactive user interface. The application may include features such as browsing songs, viewing metadata, controlling playback, managing playlists or queues, and responding to changes in playback state. Supporting packages such as on_audio_query, just_audio, audio_service, provider, and go_router provide the technical foundation for media querying, audio playback, background control, state management, and navigation. [16]–[21] The project connects the theoretical topics of programming language design and implementation with a practical mobile application built using Dart.

Troy University Resources


Good Evening
3 songs on your device
All songs
Go Beyond Andy (HO...
Andy Love
03:50
Go Beyond Sonna (H...
Sonna Rele
03:53
morse
Unknown Artist
00:32
Home
Search
Library


Now playing
FOOLS GARDEN
Lemon Tree
(Official HD Video)
len - Lemon Tree (Official
Fools Garden
00:02
03:11



This figure presents representative interface screens of Musik and gives an overview of the main user flow in the application. The first screen shows the Home or Library view, where the application scans the device and displays the available songs with basic metadata such as title, artist, artwork, and duration. This screen represents the starting point of the listening experience, allowing users to browse local audio content and select a track for playback.

The second screen shows the Now Playing interface after a song has been selected. It includes the current artwork, song title, artist name, playback progress, duration, play/pause control, previous and next buttons, and additional playback options such as shuffle and repeat. This screen demonstrates how Musik presents real-time playback state through a clean and responsive mobile interface.

The third screen illustrates the Play Queue panel, where the current track and upcoming songs are displayed in a structured list. This view helps users understand and manage the playback order instead of interacting with songs one by one. It also shows how the application connects the selected media item with a larger playback flow, which is important for a complete music-player experience.

Troy University Resources

The fourth screen shows the system-level media control interface, where the currently playing song can be controlled outside the main application screen. This demonstrates that the playback experience is not limited to the foreground UI. Users can still view the current track, pause or resume playback, move between songs, and adjust progress or volume through the device media controls.

These screenshots show the core product scope of Musik: browsing songs, starting playback, viewing the current media item, managing the play queue, and interacting with playback controls beyond the main application interface. They provide a visual introduction to the application before the report moves into the technical analysis of Dart, Flutter, state management, asynchronous events, and audio-service integration.

1. Project Requirements

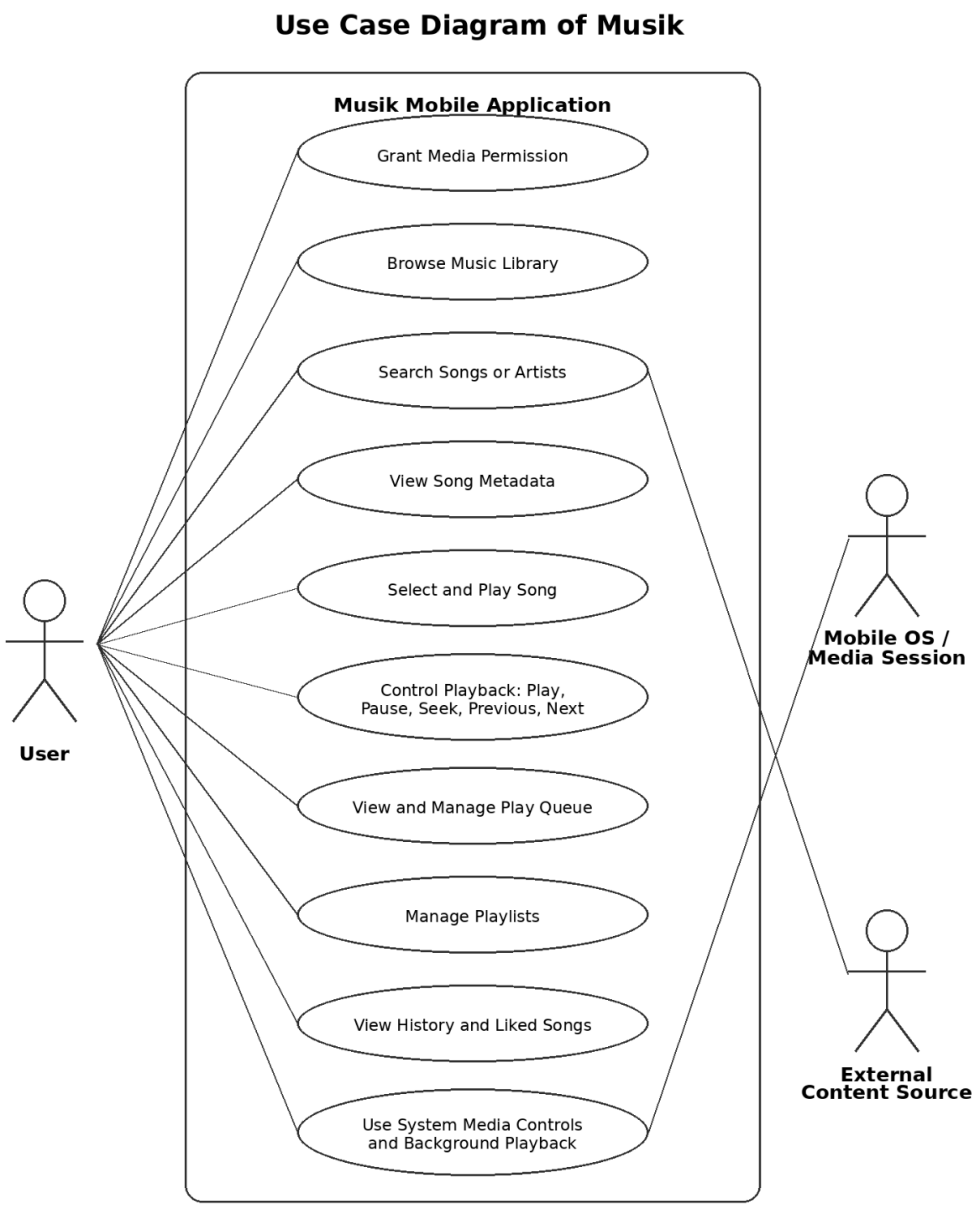
Because Musik is a mobile music player, the core requirements focus on media discovery, metadata display, playback control, queue or playlist management, permission handling, and responsive state updates in the user interface.

Table 1. Functional requirements and quality requirements of Musik

Requirement type	Requirement	Validation
Functional	Query songs from local or available media sources	This is the basic function of a music player, allowing the application to access and display available audio content. [21]
Functional	Display song metadata such as title, artist, album, duration, and artwork	Metadata is necessary for users to recognize songs and browse the music library effectively. [21]
Functional	Support play, pause, stop, seek, next, previous, and queue behaviors	These controls form the standard interaction model of a music player. [16]–[18]
Functional	Represent playback progress and playback state reactively in the UI	The interface should update according to the current song, position, duration, buffering state, and playing status. [17], [18]
Functional	Handle playlist or queue management	Queue and playlist functions allow users to organize listening order instead of playing only one song at a time. [16], [18], [21]
Functional	Manage permissions for media access	Mobile platforms require proper permission handling before the application can access local media files. [21]
Quality	Separate UI logic from data and playback logic	Clear separation of responsibility improves maintainability and reduces coupling between screens and technical services. [4], [5], [22], [23]
Quality	Use strong typing and null safety	Type safety and null safety help reduce runtime errors, especially when media metadata may be missing or incomplete. [7], [9]
Quality	Use testable architectural components	Separating components makes it easier to test playback logic, data handling, and user interface behavior independently. [13], [14]

Requirement type	Requirement	Validation
Quality	Support reasonable cross-platform reuse	Flutter and Dart allow the application to reuse a large part of the codebase across mobile platforms. [4], [6], [15]

The requirements in the table show that Musik is not only expected to play audio files, but also to manage the surrounding interaction flow of a real music player. The functional requirements define the user-facing behavior of the application, including browsing songs, viewing metadata, controlling playback, managing the queue, and handling media permissions. The quality requirements define how these features should be implemented so that the code remains readable, reliable, and maintainable. These requirements provide the foundation for the design and implementation decisions discussed in the following sections.



The diagram summarizes the main user-facing use cases of Musik and the external actors that support system media controls and optional online content access.

Troy University Resources

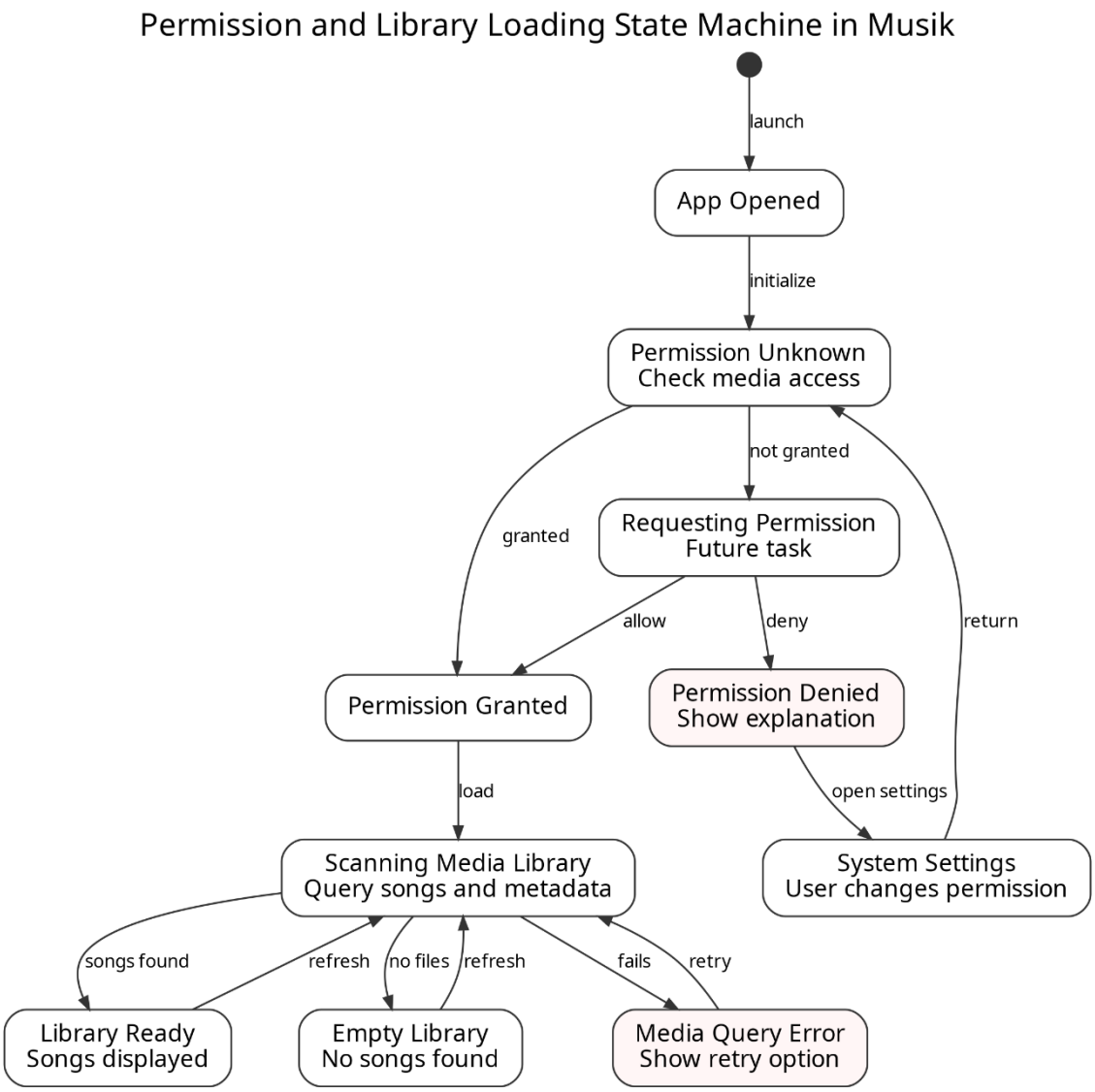
This figure presents the main use cases of Musik from the perspective of external actors. The primary actor is the user, who interacts with the application to grant media permission, browse the music library, search for songs or artists, view song metadata, select and play a song, control playback, manage the play queue, manage playlists, view history and liked songs, and use system media controls. These interactions summarize the product scope of Musik as a mobile music-player application.

The diagram also shows two supporting external actors. The Mobile OS / Media Session actor represents platform-level behavior such as system media controls and background playback. This is important because a realistic music player should not be limited to the foreground application screen. The External Content Source actor represents optional online content access, such as searching or loading external music-related information when that feature is included in the application scope.

This use case diagram complements the requirement table because it turns the functional requirements into visible actor-system interactions. Media permission, library browsing, metadata display, playback control, queue management, playlist management, and system-level playback control can all be traced from this diagram to the later architecture, package, and testing discussions in the report.

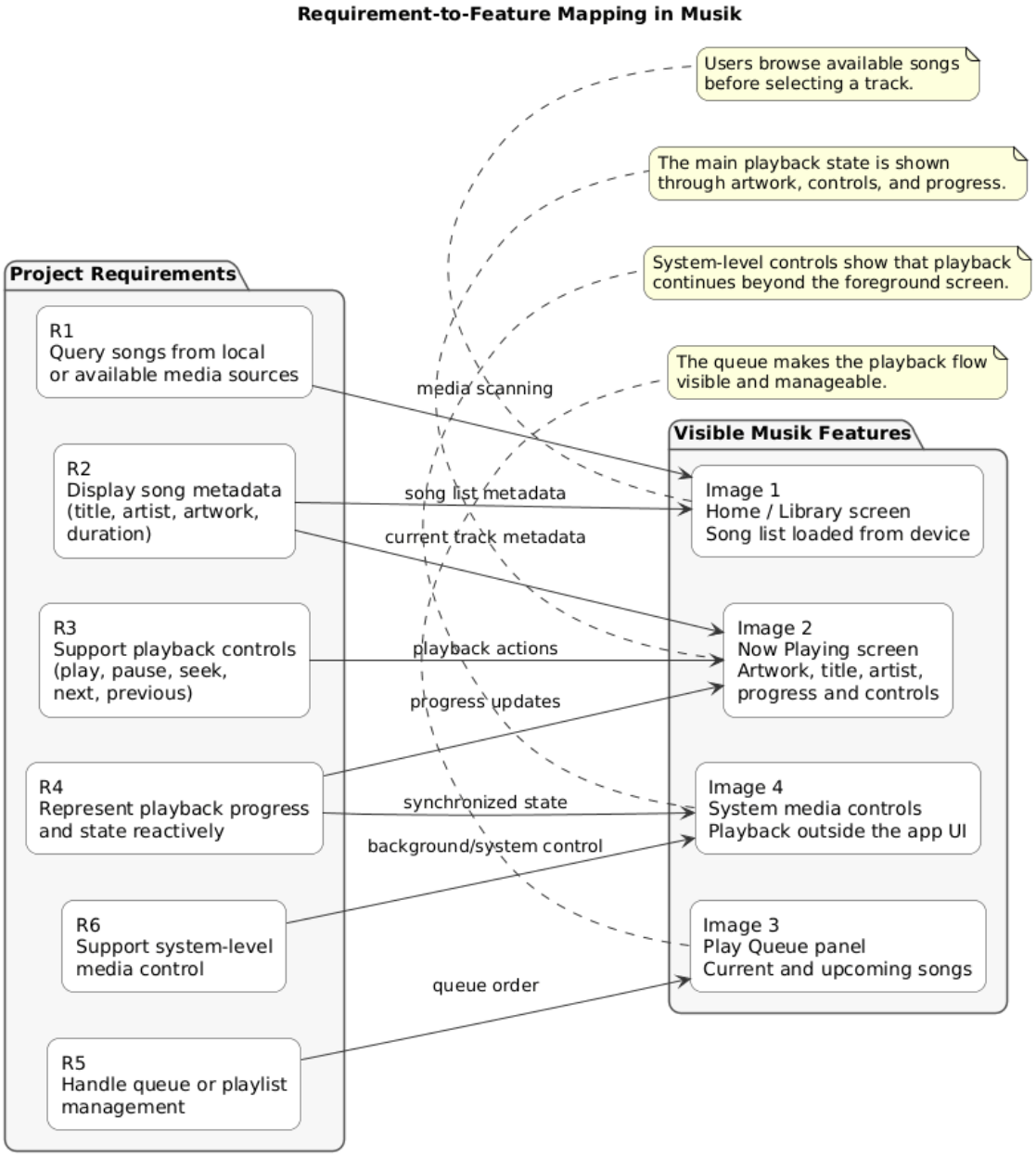
Table 2. Use Case Summary

Actor	Use case	Purpose
User	Grant Media Permission	Allows the application to access local audio files and related metadata.
User	Browse Music Library	Displays available songs and gives the user an entry point into the listening flow.
User	Search Songs or Artists	Lets the user find music by song title, artist name, or related metadata.
User	View Song Metadata	Shows information such as title, artist, artwork, album, and duration.
User	Select and Play Song	Starts playback from a selected song item.
User	Control Playback	Supports play, pause, seek, previous, and next actions.
User	View and Manage Play Queue	Shows the current and upcoming songs in the playback order.
User	Manage Playlists	Supports organizing songs into user-defined collections.
User	View History and Liked Songs	Provides access to previously played or favorited songs.
Mobile OS / Media Session	Use System Media Controls and Background Playback	Keeps playback controllable through platform media controls outside the main app screen.
External Content Source	Search or Load Online Content	Supports optional online content lookup or external music-related data access.



This diagram explains how Musik reaches the music library screen. The application checks media permission, requests access if needed, and then scans the device for songs and metadata after permission is granted. The scan can result in a ready library, an empty library, or a media-query error. This state machine is useful because local media access is not only a UI feature; it also depends on permission state, asynchronous media querying, and error handling. [8], [21]

Troy University Resources



The diagram above explains how the main requirements of Musik are translated into visible features in the application. On the left side, the diagram lists the key project requirements, including media querying, metadata display, playback control, reactive playback progress, queue management, and system-level media control. On the right side, these requirements are connected to four representative application screens: the Home/Library screen, the Now Playing screen, the Play Queue panel, and the system media control interface. This structure helps show that each requirement is not only described theoretically in Table C, but also appears as an observable part of the implemented application.

Requirement R1 focuses on querying songs from local or available media sources. This requirement is mapped to Image 1, the Home/Library screen, because this screen is responsible for showing the songs that the application has loaded from the device or available music source. From the user's perspective, this is the entry point of the music player: users open the application, view the available tracks, and select a song to start listening. From the implementation perspective, this requirement is related to media scanning, permission handling, data loading, and presenting the result in a structured song list.

Requirement R2 focuses on displaying song metadata, such as title, artist, artwork, and duration. This requirement is mapped to both Image 1 and Image 2 because metadata appears in more than one part of the application. In the Home/Library screen, metadata helps users identify songs before playing them. In the Now Playing screen, metadata gives detailed information about the currently selected track. This shows that metadata is not a minor visual detail, but a core part of the user experience and data model of the application.

Requirement R3 focuses on playback controls, including play, pause, seek, next, and previous actions. This requirement is mapped to Image 2, the Now Playing screen, because this is where the user directly controls audio playback. The buttons and seek bar shown on this screen represent the main interaction between the user and the playback engine. These controls demonstrate that Musik is not only able to display songs, but also provides the essential operations expected from a functional music player.

Requirement R4 focuses on representing playback progress and playback state reactively. This requirement is mapped to both Image 2 and Image 4. In Image 2, the playback progress bar, current time, total duration, and play/pause state must update according to the current audio state. In Image 4, the same playback state is reflected in the system-level media control interface. This mapping is important because it shows that the application state is not isolated inside one screen. Instead, playback information must remain synchronized between the app interface and the operating system's media controls.

Requirement R5 focuses on queue or playlist management. This requirement is mapped to Image 3, the Play Queue panel. The queue screen shows the current track and the upcoming songs, allowing users to understand the order of playback. This feature makes the application more complete than a simple single-track player, because users can view and manage the listening flow across multiple songs. It also demonstrates the need for internal state management, since the application must keep track of the current item, next items, and changes in playback order.

Requirement R6 focuses on system-level media control. This requirement is mapped to Image 4, where the current song can be controlled outside the main application interface. This feature shows that Musik supports a more realistic mobile music experience, because users often control playback from the notification panel or system media interface instead of keeping the app open at all times. The connection between R6 and Image 4 also highlights the role of background playback and media-session integration in the project.

So this mapping makes the project easier to evaluate because it connects each major requirement to a concrete screen or interaction in Musik. The Home/Library screen verifies media discovery and metadata display, the Now Playing screen verifies playback control and reactive state updates, the Play Queue panel verifies queue management, and the system media control interface verifies background or system-level playback behavior. Therefore, the diagram acts as a bridge between the requirement table and the later technical analysis of Dart, Flutter, asynchronous programming, state management, and audio playback integration.

2. The Concerning Problem in Technical Terms

The concerning problem for Musik is not simply the ability to play audio files. From an implementation perspective, a mobile music player must coordinate several interacting state domains at the same time. The first domain is media-discovery state, which includes the available songs, artists, albums, playlists, artwork, and other metadata obtained from the device or a supported media source. The second domain is playback state, which includes the currently selected track, playback duration, current position, buffering status, playing or paused state, completion state, and queue position. The third domain is interface state,

Troy University Resources

which includes the active screen or tab, selected item, visible controls, navigation flow, and widget-level interactions.

These state domains need to be separated carefully because they have different lifetimes and different scopes. For example, the currently pressed button or selected tab may only belong to a single screen, while the current song, playback position, and queue should remain consistent across multiple screens and even outside the foreground application. Flutter distinguishes between ephemeral widget state and broader application state, and recommends deciding where each piece of data belongs based on how widely it is used, how long it must persist, and how many components depend on it. [10]

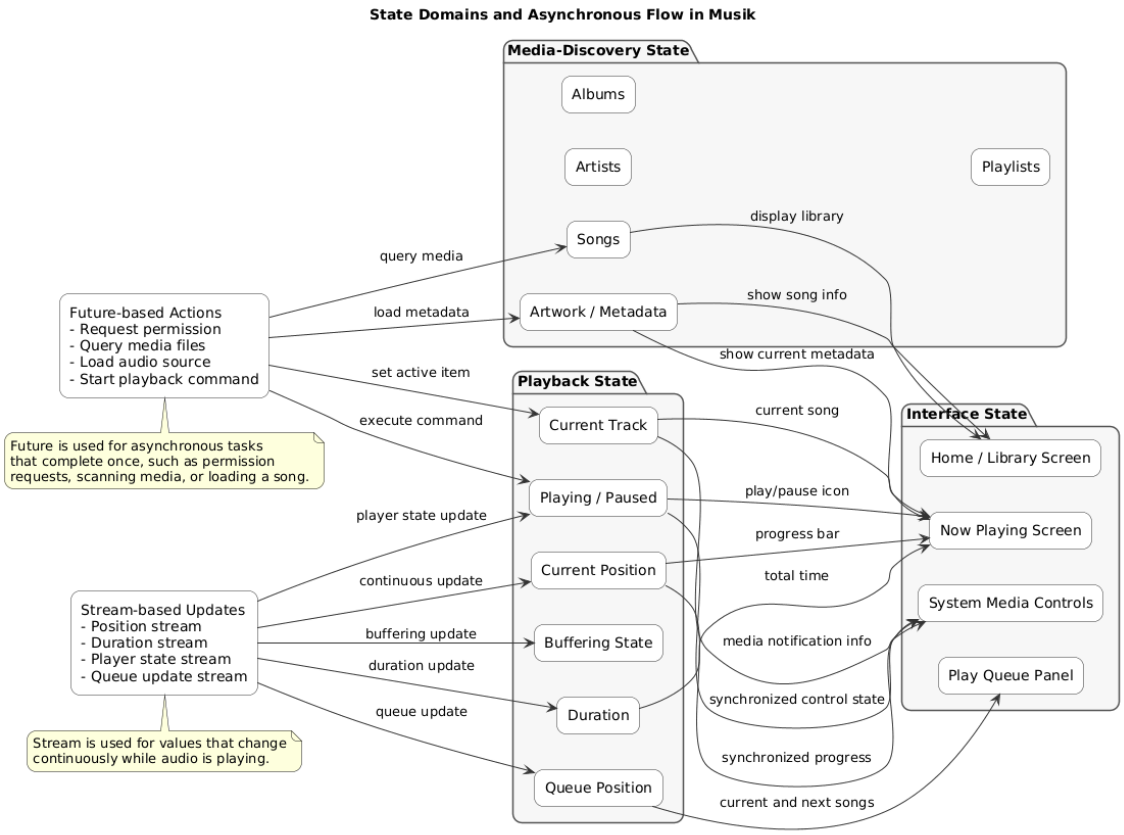
This separation is important for Musik because placing all media, playback, and interface logic into one screen class would make the application difficult to maintain and extend. A music player may appear simple from the user's perspective, but internally it must handle asynchronous loading, permission requests, playback commands, progress updates, queue changes, and UI rebuilding. If these responsibilities are mixed together, small changes in one feature can easily break another feature. Therefore, our application is organized so that the user interface focuses on rendering and interaction, view models or controllers manage UI-related state and commands, repositories provide access to data, and services handle platform or package-level operations such as media querying and audio playback. [5]

Audio-related state is especially suitable for this kind of separation because it is inherently asynchronous. Dart provides two important abstractions for this situation: Future and Stream. A Future is appropriate when an operation produces one result later, such as requesting media permission, loading an audio source, or initializing a service. A Stream is appropriate when values are produced continuously over time, such as playback position, buffered position, duration changes, and player-state updates. This distinction maps naturally to the behavior of a music player: some operations happen once and complete, while others continue to emit updates while the song is playing. [8]

In Musik, this asynchronous model can be seen most clearly in the playback flow. When a user selects a song, the application may need to prepare the audio source, update the active media item, and start playback. These actions are usually handled through Future-based asynchronous calls. After playback starts, the interface must continue to receive changing values such as current position, duration, buffering state, and play/pause status. These values are better represented through streams because they change repeatedly during the listening session. The Now Playing screen, progress bar, queue panel, and system media controls all depend on this continuous flow of playback state.

The just_audio package provides a concrete example of this model. A typical audio player initializes asynchronously, configures an audio source, listens for playback errors, and exposes playback-related values through streams. The user interface can then rebuild selected parts of the screen whenever position, duration, or player state changes. This behavior shows how Dart's asynchronous and reactive programming model becomes practical in a real media application, rather than remaining only a theoretical language feature. [17]

Therefore, the main technical challenge of Musik is the coordination of media data, playback behavior, and user interface state in a clean architecture. The application must allow users to browse songs, start playback, view metadata, monitor progress, manage the queue, and control playback from both the app interface and the system media interface. To support these requirements, the implementation needs clear state boundaries, asynchronous operations, reactive updates, and modular components. This is why Musik is a suitable case study for analyzing Dart and Flutter in the context of programming language concepts, especially object-oriented design, asynchronous programming, reactive state updates, and maintainable software structure.



This figure shows the internal state structure behind the Musik application. The diagram divides the application into three main state domains: media-discovery state, playback state, and interface state. Media-discovery state contains the information obtained from local or available media sources, such as songs, artists, albums, playlists, artwork, and metadata. Playback state contains the values controlled by the audio player, such as the current track, duration, position, buffering state, playing or paused status, and queue position. Interface state contains the visible screens and controls that users interact with, including the Home screen, Now Playing screen, Play Queue panel, and system-level media controls. The state domains are connected through asynchronous operations. Future-based actions are used for tasks that complete once, such as requesting media permission, querying media files, loading an audio source, or starting a playback command. Stream-based updates are used for values that change continuously while audio is playing, such as playback progress, duration, buffering status, player state, and queue position. This separation is important because it prevents the user interface from directly owning all playback and media logic.

Through this structure, Musik demonstrates several programming language concepts in a realistic application. Typed models help represent media and playback data safely. Object-oriented organization helps separate responsibilities between UI components, controllers, repositories, and services. Future and Stream support asynchronous and reactive behavior. The result is an application design in which the interface can remain responsive while the audio player continues to update state in the background. This makes Musik a suitable case study for analyzing Dart and Flutter from both a practical and programming-language perspective.

3. Programming Language Selection Criteria

The selection of Dart for Musik is justified by both the academic purpose of the project and the technical requirements of a mobile music player. From the academic perspective, Dart is suitable because it allows

Troy University Resources

the project to demonstrate several important programming language concepts in one realistic application, including strong typing, null safety, object-oriented organization, asynchronous programming, reactive data flow, generic collections, and multi-paradigm development. From the technical perspective, Dart works directly with Flutter, which makes it possible to build a cross-platform mobile application from a single codebase while maintaining a responsive and declarative user interface.

First, Dart provides a strong but practical type system. The language is strongly typed, but it also supports type inference, allowing developers to write concise code without losing the benefits of static checking. This is useful in Musik because the application needs to work with structured data such as songs, artists, albums, playlists, queue items, and playback state. Instead of relying on unstructured maps or loosely typed values, the application can represent these concepts through typed models and collections. As a result, the code becomes easier to read, easier to maintain, and less likely to produce type-related runtime errors. [7]

Second, Dart's sound null safety is especially valuable in a media application. In a music player, many values may be unavailable at certain moments. For example, album artwork may be missing, a song duration may not be known before the audio source is loaded, and there may be no current song before the user selects a track. Dart makes these cases explicit by separating nullable and non-nullable types. This helps the developer handle missing data intentionally instead of allowing null-related errors to appear unexpectedly during runtime. [9]

Third, Dart's asynchronous programming model fits the behavior of a music player very well. Musik needs to perform operations such as requesting media permissions, scanning local songs, loading metadata, preparing audio sources, and starting playback commands. These operations do not always complete immediately, so they are naturally represented using Future and async/await. At the same time, playback-related values such as current position, duration, buffering state, player state, and queue changes continue to update while the song is playing. These values are naturally represented through Stream-based updates. Therefore, Dart provides both one-time asynchronous operations and continuous reactive data flow, which are essential for audio applications. [8]

Fourth, Dart integrates closely with Flutter's declarative UI model. In Flutter, the interface is rebuilt according to the current application state instead of being manually updated element by element. This model is suitable for Musik because a single playback-state change may affect several parts of the interface at the same time, such as the play/pause button, progress bar, current song title, queue highlight, and system media controls. By using Flutter with Dart, the application can present changing playback state in a consistent and responsive way. [6], [11]

Fifth, Dart and Flutter are supported by a mature package ecosystem for music-player development. Packages such as just_audio, audio_service, on_audio_query, provider, and go_router provide practical support for audio playback, background media control, local media querying, state management, and navigation. These packages make it possible to build a course-scale application with realistic functionality while still keeping the project manageable. just_audio supports playback and playlist-related behavior, audio_service supports background playback and system media controls, on_audio_query supports querying songs, albums, artists, playlists, artwork, and media permissions from device storage, provider helps expose and update application state, and go_router supports structured navigation between screens. [16]–[21]

Overall, Dart was selected because it matches both the educational and technical goals of Musik. It allows the project to demonstrate programming language concepts in a concrete software system, while also providing the tools needed to implement a responsive and maintainable mobile music player. The

combination of strong typing, null safety, asynchronous programming, reactive streams, object-oriented structure, and Flutter integration makes Dart an appropriate language choice for this project.

Table 3. Dart and Flutter Suitability Assessment for Musik

Criterion	Why it matters for Musik	Assessment
Static correctness	Musik uses structured data such as songs, artists, playlists, queue items, and playback state, so type-related errors should be detected early.	Strong, because Dart supports static typing, type inference, generic collections, and sound null safety. [7], [9]
Null-safety support	Media data may be incomplete, such as missing artwork, unknown duration, or no currently selected song before playback starts.	Strong, because Dart separates nullable and non-nullable values, making missing data explicit in the code. [9]
Asynchronous programming	Music playback requires permission requests, media loading, playback commands, progress updates, and state synchronization.	Strong, because Dart provides Future for one-time asynchronous tasks and Stream for continuous updates. [8]
Reactive UI development	The interface must update when playback state changes, including progress, play/pause status, current song, and queue position.	Strong, because Flutter uses a declarative UI model in which the interface is rebuilt from application state. [6], [11]
Cross-platform reuse	A mobile music player should reuse as much code as possible across Android and iOS to reduce development effort.	Strong, because Flutter supports a single-codebase workflow for cross-platform mobile development. [15]
Package ecosystem	Musik needs practical support for audio playback, background media control, metadata querying, state management, and routing.	Strong, because packages such as just_audio, audio_service, on_audio_query, provider, and go_router support these needs. [16]–[21]
Testability	The project needs a structure that can be explained, tested, and maintained beyond a simple visual demo.	Strong when combined with layered architecture, separated responsibilities, and appropriate unit, widget, or integration testing. [5], [13], [14]
Language-feature demonstration	Musik should clearly show how Dart language features are used in a realistic application.	Strong, because the project demonstrates object-oriented modeling, null safety, generics, asynchronous programming, streams, and multi-paradigm development. [7]–[10]

Troy University Resources

Methodology

The methodology of Musik combines software architecture, language-level design, and an iterative implementation process. The project was not developed only as a visual mobile application, but as a practical system for demonstrating how Dart and Flutter can be used to organize data, handle asynchronous behavior, update the user interface reactively, and separate responsibilities between components. Therefore, the methodology focuses on both the structure of the application and the programming language features used to implement that structure.

From the architectural perspective, Musik follows a layered organization. The user interface layer is responsible for displaying screens and receiving user interactions, such as selecting a song, pressing playback buttons, opening the queue, or navigating between screens. The state-management or controller layer is responsible for coordinating UI state and user commands. The repository or data-access layer is responsible for obtaining music-related data, such as songs, artists, albums, playlists, and metadata. The service layer is responsible for interacting with external packages or platform-level functions, such as querying local media, controlling audio playback, and supporting system media controls. This separation helps reduce coupling between the visual interface and the technical logic behind it. [4], [5]

From the language-level perspective, the methodology emphasizes how Dart features are applied in the implementation of Musik. Typed classes and generic collections are used to represent structured data such as songs, playlists, queue items, and playback state. Null safety is used to handle incomplete media information, such as missing artwork or unavailable duration. Future-based asynchronous operations are used for tasks that complete once, such as requesting permissions, loading media data, or preparing an audio source. Stream-based updates are used for values that change continuously during playback, such as playback position, duration, buffering state, and player status. These features allow the application to remain responsive while managing complex media behavior. [7]–[10]

From the implementation-process perspective, Musik was developed through incremental feature construction. The initial stage focused on defining the project scope, setting up the Flutter environment, creating the repository structure, and identifying the main application screens. The next stage focused on core music-player behavior, including media loading, metadata display, playback controls, and the Now Playing interface. Later stages added queue or playlist behavior, search or browsing support, state synchronization, system-level playback control, testing, bug fixing, screenshot collection, and report preparation. This process allowed the team to build the application gradually while keeping the main technical goals clear.

The methodology also considers trade-offs. A very simple implementation could place most logic directly inside screen widgets, but that approach would make the application harder to maintain as features grow. A highly complex architecture could introduce too many abstractions for a course-scale project. Musik therefore uses a balanced structure: the application separates UI, state coordination, data access, and playback services, while still keeping the overall design understandable and suitable for a student project. This balance supports both practical development and academic analysis.

Overall, the methodology of Musik is based on connecting implementation decisions with programming language concepts. Each major technical decision is related to a specific need of the application: typed models support reliable media data handling, null safety supports incomplete metadata, Future supports delayed operations, Stream supports continuous playback updates, and Flutter's declarative UI model supports reactive interface rebuilding. As a result, the project methodology demonstrates not only how Musik was built, but also why Dart and Flutter are appropriate tools for implementing a mobile music player.

1. Architecture Options Considered

Before selecting the implementation architecture for Musik, three realistic design options were considered. The first option was to keep most state inside individual widgets using widget-local setState(). This approach is simple and fast for early prototyping, but it becomes difficult to maintain when playback state, media data, queue information, and system media controls must be shared across multiple screens. The second option was to use Provider with ChangeNotifier and an MVVM-style layered structure. This approach separates user interface code from state coordination, repositories, and services, making it more suitable for a music player with asynchronous playback behavior. The third option was to adopt a heavier Clean Architecture-style design with more formal layers such as entities, use cases, repositories, and controllers. Although this approach provides strong separation, it also introduces more complexity than necessary for a course-scale project.

The table below compares these options and explains why Provider with ChangeNotifier and MVVM-style layering was selected as the most appropriate architecture for Musik.

Table 4. Comparison of Architecture Options for the Musik Application

Option	Description	Strengths	Weaknesses	Suitability for Musik
Widget-local setState()	State is stored and updated directly inside individual widgets or screens.	Very simple to implement, fast for prototyping, and suitable for short-lived interface state such as button presses, selected tabs, or temporary UI changes. [10], [12]	Does not scale well when state must be shared across multiple screens, playback services, repositories, and system media controls.	Suitable only for a very small prototype or isolated UI interactions.
Provider + ChangeNotifier + MVVM-style layering	Shared state is exposed through providers, UI-related logic is handled by view models or controllers, and data or playback logic is separated into repositories and services.	Provides a clear separation of responsibilities, reduces unnecessary coupling, supports dependency injection, improves testability, and remains understandable for a course-scale project. [5], [11], [19]	Requires more files, clearer naming conventions, and consistent architectural discipline from all team members.	Best fit for Musik because the application needs shared playback state, media data access, queue handling, and reactive UI updates.
Heavier state frameworks or Clean Architecture variants	The application is divided into more formal layers such as entities, use cases, repositories, controllers, and event/state classes.	Provides strong separation of concerns, long-term extensibility, and clearer boundaries for large applications.	Adds more complexity, boilerplate, and learning overhead than necessary for a student music-player project.	Useful only if Musik grows into a larger production-scale application with many more features and contributors.

Flutter's state-management model supports this architectural decision because it does not require every type of state to be handled in the same way. Simple widget-specific state can remain inside individual

Troy University Resources

widgets, while shared application state should be managed through a more systematic structure as the application grows. This distinction is important for Musik because not all state has the same scope. For example, a temporary button animation or selected tab may belong only to one screen, but playback state, current song information, queue position, and media metadata must be shared across several screens and sometimes with system-level media controls. [10]–[12]

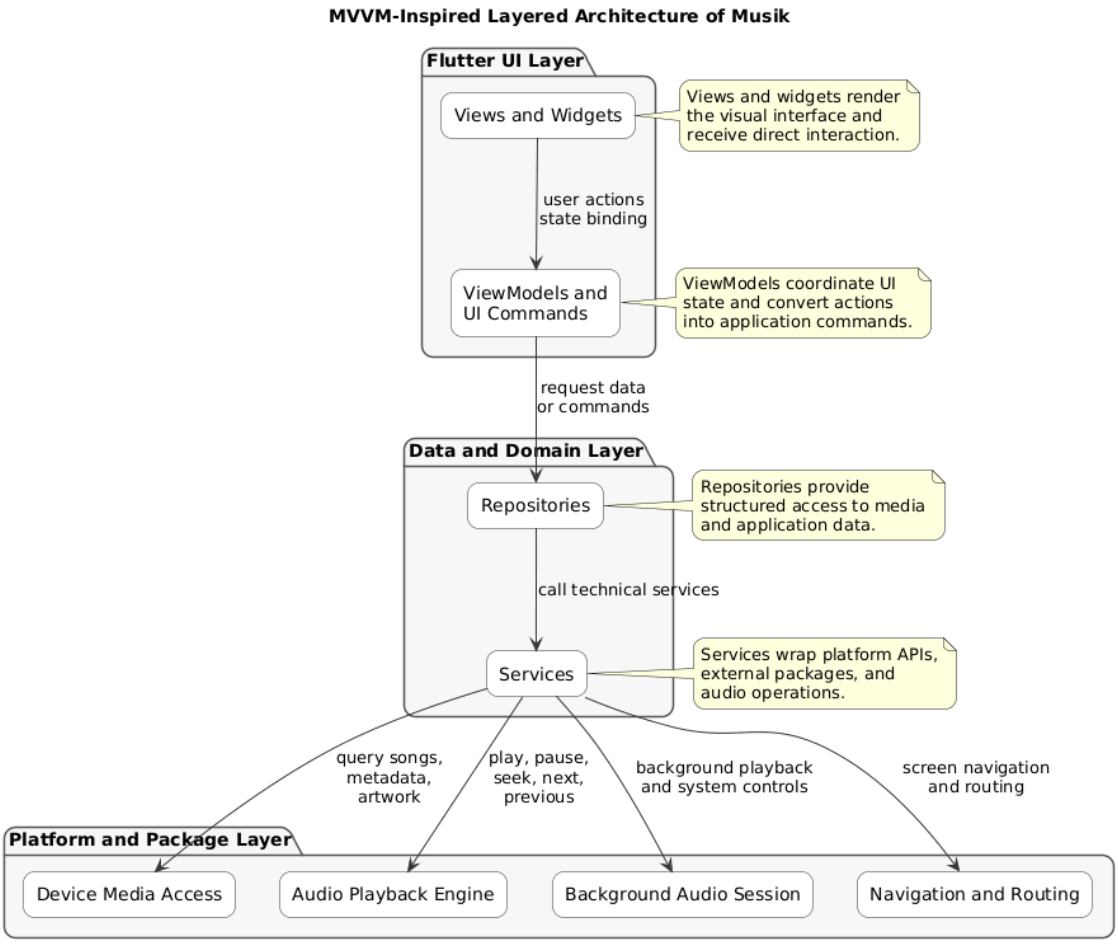
For this reason, Musik adopts a middle-ground architecture based on Provider, ChangeNotifier, and MVVM-style layering. Provider is suitable for this project because it offers a practical way to expose shared state and dependencies to the widget tree without introducing excessive boilerplate. It also supports clearer allocation and disposal of objects, lazy initialization, and state propagation through the application. This makes Provider more scalable than using only widget-local setState(), while still being simpler than heavier architectural frameworks. [19]

In the selected structure, the user interface layer focuses on rendering screens and receiving user interactions. View models or controllers manage UI-related state and translate user actions into application commands. Repositories provide access to media data, while services wrap technical operations such as media querying, audio playback, and background or system-level media control. This separation is especially important in Musik because playback behavior is asynchronous and must remain synchronized across the Home screen, Now Playing screen, queue panel, and system media controls.

The MVVM-inspired structure is also appropriate from an academic perspective because it makes the relationship between software design and programming language features easier to explain. Dart's typed models can be used to represent songs, playlists, queue items, and playback state. Future-based methods can be used for one-time asynchronous operations such as permission requests, media loading, and playback commands. Stream-based updates can be used for continuously changing playback values such as position, duration, buffering state, and player status. As a result, the architecture helps make Dart's language features visible in the implementation rather than hiding them inside a single large screen class.

This architectural choice also reflects a reasonable trade-off for a student-developed music player. A simpler setState()-only design would be faster to implement at the beginning, but it would become harder to maintain once the application needs shared playback state, queue management, and background control. A heavier Clean Architecture-style design would provide stronger formal separation, but it could introduce unnecessary complexity for the scale of this project. Provider with ChangeNotifier and MVVM-style layering therefore provides the best balance between simplicity, maintainability, testability, and academic clarity for Musik.

2. Chosen Component Model



The architecture shown in the figure follows the layered separation recommended for Flutter applications. In this structure, the view layer is responsible for rendering the user interface and handling only simple presentation-related behavior. The view model or controller layer manages UI state, prepares data for display, and exposes commands that the view can call. The repository layer provides stable access to application data and acts as the source of truth for specific data types. The service layer wraps external packages, platform APIs, and technical operations, usually through asynchronous interfaces such as Future and Stream. This separation is important because it prevents the user interface from directly depending on low-level media access or playback implementation details. [5]

In Musik, the view layer includes the visible screens and widgets of the application, such as the Home or Library screen, song list, Now Playing screen, Play Queue panel, search interface, and settings or permission-related screens. These screens are responsible for presenting information to users and receiving interactions such as selecting a song, pressing the play or pause button, opening the queue, or navigating to another screen. However, they do not contain complex business logic such as filtering media data, managing playback sessions, updating the queue, or synchronizing background media controls. Instead, the views receive prepared state from the view model layer and send user actions back through commands or callback methods.

The view model or controller layer acts as the connection between the user interface and the deeper application logic. For example, a library-related view model can hold the current list of songs, loading state, permission state, selected filter, and possible error message. A player-related view model can hold the current media item, playing or paused state, playback position, buffered duration, total duration,

Troy University Resources

and queue information. A search-related view model can maintain the current query and filtered search results. This layer is important because it allows the UI to stay simple while still responding to changing application state. When the playback state changes, the view model can notify the interface so that the Now Playing screen, mini-player, queue panel, and system media controls remain consistent.

The repository layer provides a more stable boundary for accessing and organizing application data. In Musik, repositories can be used for media data, playback-related data, playlist data, and settings or preferences. A MediaRepository can provide access to songs, artists, albums, playlists, and artwork. A PlaybackRepository can coordinate information about the current track, queue, and playback session. A SettingsRepository can manage user preferences or configuration values if needed. By placing this logic in repositories, the application avoids scattering data-access behavior across many screens. This also makes the application easier to test, because repositories can be replaced with mock or fake implementations during unit testing.

The service layer contains the technical integrations with external packages and platform-specific features. In a music-player application, this layer may include a media-query service for reading songs, albums, artists, playlists, artwork, and media permissions from the device; an audio playback service for loading audio sources and controlling play, pause, seek, next, and previous actions; a background audio or media-session service for notification and lock-screen controls; and a routing service or navigation configuration for moving between screens. These services are placed below repositories because they deal directly with package APIs and platform behavior. They expose clean methods or streams to the upper layers instead of forcing the UI to interact with package-specific details. [5], [8], [18], [21]

This structure is especially suitable for Musik because music playback is both stateful and asynchronous. Some operations, such as requesting permissions, scanning songs, loading an audio source, or starting playback, are one-time asynchronous tasks and can be represented using Future. Other values, such as playback position, duration, buffering status, player state, and queue changes, update continuously while the song is playing and are better represented using Stream. By placing these asynchronous operations in services and exposing organized state through repositories and view models, Musik can keep its interface responsive while maintaining a clear separation between presentation, data, and playback behavior. [8]

Overall, the selected architecture helps Musik remain understandable, maintainable, and testable. Views focus on displaying the application, view models coordinate screen state and user commands, repositories organize application data, and services handle platform or package-level operations. This organization also supports the academic purpose of the project because it makes Dart and Flutter concepts visible in the implementation. Typed models, null safety, Future-based operations, Stream-based updates, reactive UI rebuilding, and modular object-oriented structure all appear naturally within this layered design.

3. Implementation of Selected Packages in the Software

Table 5. Packages selected for Musik

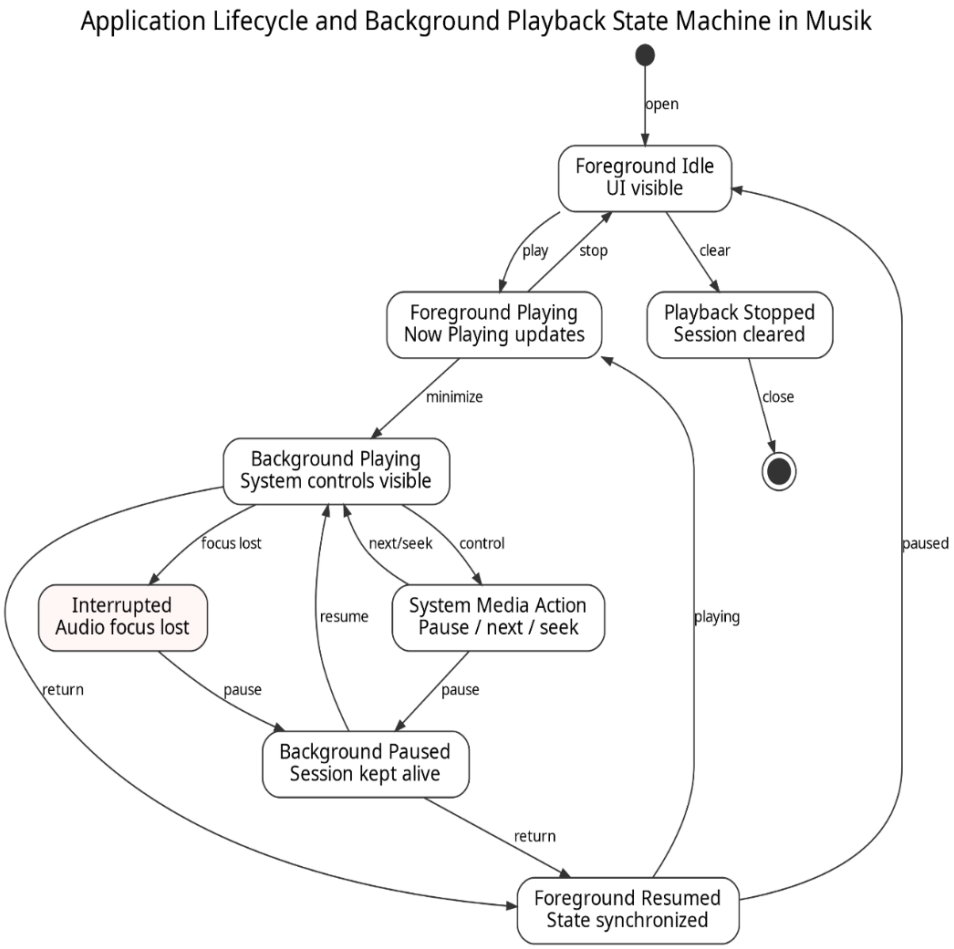
Package	Primary role in Musik	Technical justification
on_audio_query	Discover songs, albums, artists, playlists, artwork, and media permissions	Designed specifically for media querying from device storage and documents permissions/platform support [21]
just_audio	Playback engine, playlists, seeking, progress streams	Feature-rich audio player with playlist support and stream-oriented player state [16], [17]
audio_service	Background audio, system media controls, queue/session integration	Exposes an audio handler as a single source of truth for all clients and supports notifications/lock screen/media controls [18]
provider	Dependency injection and state exposure	Reduces boilerplate and simplifies allocation/disposal and consumption of shared state [19]
go_router	Declarative navigation across app screens	Provides a URL-based declarative router with deep-link and nested-route support [20]

on_audio_query plays an important role in Musik because the application needs to access and display local music information from the device. The package provides methods for querying songs, albums, artists, playlists, genres, artwork, and related media information. It also supports permission-related requirements on mobile platforms, such as audio media access on Android and media usage descriptions on iOS. In the architecture of Musik, on_audio_query acts as the boundary between the application logic and the device media store. This allows the UI and data models to work with structured media information without directly handling platform-specific media access details. [21]

just_audio provides the core playback engine of Musik. It supports loading and playing audio from different sources and provides common playback operations such as play, pause, seek, next, previous, and playlist-based playback. This package is suitable for Musik because music playback is not a static operation; it involves continuous updates such as current position, duration, buffering state, player state, and playback completion. The stream-oriented design of just_audio fits Dart's asynchronous programming model and allows the application to update the Now Playing screen, progress bar, queue panel, and playback controls reactively. [16], [17]

audio_service extends the playback experience beyond the foreground application interface. In a real mobile music player, users expect music to continue playing when the app is minimized and to remain controllable through notification controls, lock-screen controls, or system media buttons. audio_service supports this requirement by providing a background audio handler and system-facing media control integration. For Musik, this package helps connect the internal playback state of the application with the operating system's media interface. It also supports the idea of a single source of truth for playback state, which is important when the same song information and controls must appear consistently in the app UI and the system media controls. [18]

Troy University Resources



This diagram illustrates how Musik manages playback across foreground and background states. The session begins in the foreground idle state. When the user starts playback, Musik enters the foreground playing state and updates the Now Playing interface. If the application is minimized, playback continues in the background and system media controls remain available. These controls allow users to pause, resume, seek, or move between tracks without reopening the application.

The diagram also represents interruption and recovery behavior. Audio focus loss moves playback into an interrupted state, while pause actions keep the background session alive. When the user returns to the application, Musik synchronizes the foreground interface with the current playback session. When the session is stopped or cleared, playback returns to an idle or stopped state. This state machine shows how audio_service supports background playback and system media controls, while just_audio supplies the playback state and stream updates used by the application interface. [17], [18]

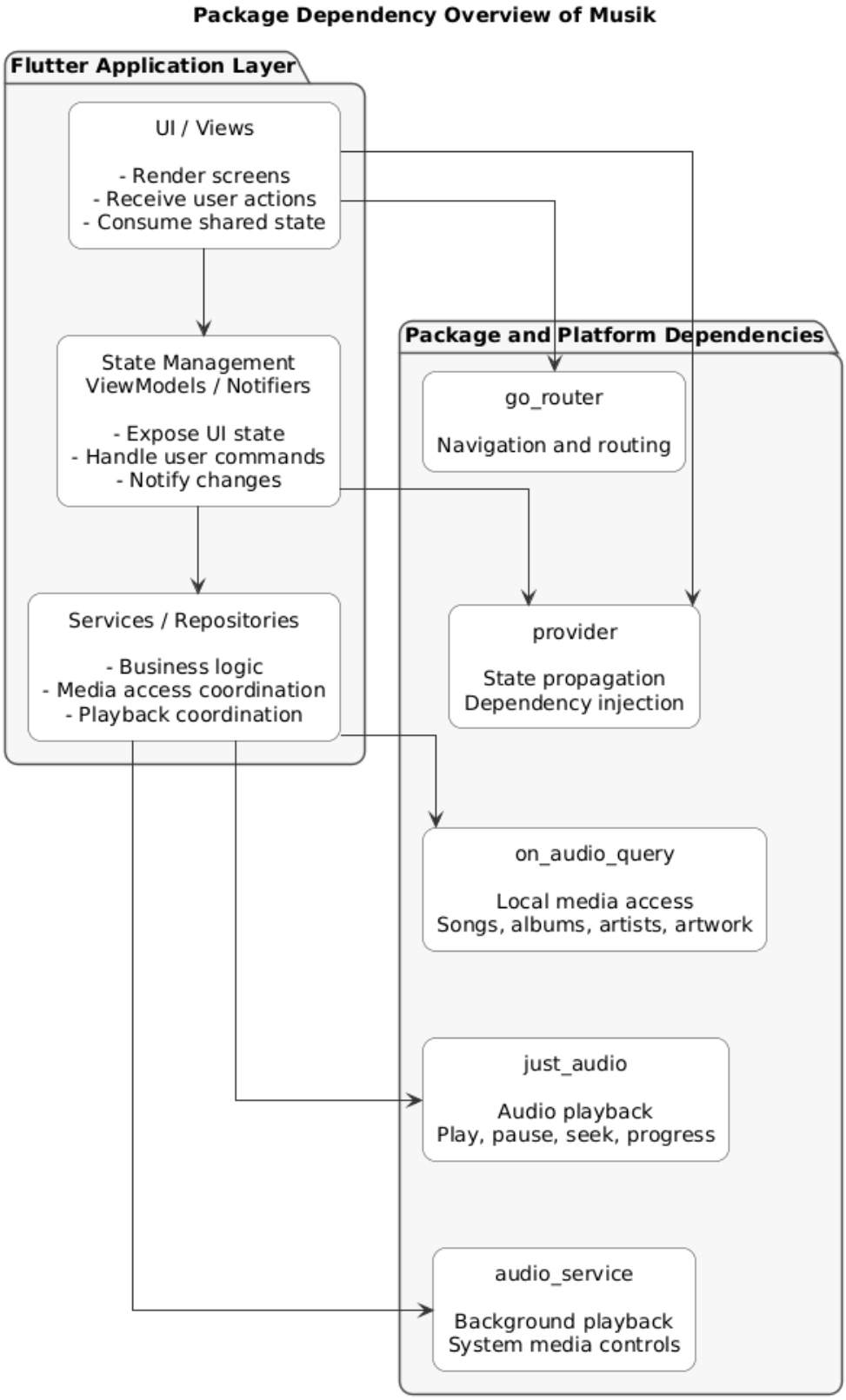
provider is used as supporting infrastructure for dependency injection and shared state exposure. Although it is not a music-specific package, it is important for the maintainability of Musik because playback state, media data, queue information, and user interactions need to be shared across different screens. provider allows view models, controllers, repositories, or services to be made available to the widget tree in a structured way. This reduces unnecessary boilerplate and avoids passing dependencies manually through many widget constructors. As a result, Musik can keep its UI components cleaner while still allowing them to react to changes in shared application state. [19]

go_router is used to organize navigation between screens in the application. Musik contains several user-facing areas, such as the Home or Library screen, Now Playing screen, Play Queue panel, Search screen, and Settings or permission-related screens. A declarative routing approach helps define these

navigation paths more clearly and makes screen transitions easier to manage as the application grows. Although routing is not part of the audio domain itself, it supports the overall user experience by making movement between browsing, playback, queue management, and supporting screens more structured. [20]

These packages form the practical implementation foundation of Musik. on_audio_query handles media discovery and metadata access, just_audio handles the core playback engine, audio_service connects playback with background and system-level controls, provider supports shared state and dependency management, and go_router structures navigation between screens. This package selection shows that the implementation is not built around isolated libraries, but around a layered design in which each package has a clear responsibility. It also reinforces the programming-language analysis of the project, because these packages make direct use of Dart and Flutter concepts such as typed models, asynchronous operations, streams, reactive UI updates, and modular object-oriented structure.

Troy University Resources



The diagram illustrates how the main packages in Musik are connected to the application layers. The UI layer uses ‘go_router‘ for screen navigation and ‘provider‘ to consume shared application state. The state-management layer exposes view-model or notifier state through ‘provider‘, while the services and repositories layer uses ‘on_audio_query‘ for local media access, ‘just_audio‘ for playback control, and ‘audio_service‘ for background playback and system media integration. This structure keeps the user

interface independent from low-level package APIs and helps the application remain easier to maintain, test, and extend.

4. Repository-Driven Code Structure

The code structure of Musik can be presented according to both visible features and architectural responsibilities. Instead of treating the repository as a flat collection of screens and utility files, the project is better understood as a layered Flutter application. The main user-facing features are represented through views and widgets, while shared state, media access, playback behavior, navigation, and platform integration are separated into view models or controllers, repositories, and services.

This organization helps connect the source code structure with the architecture discussed in the previous section. Screens such as Home, Search, Now Playing, Queue, Library, and Settings belong to the presentation layer. State-management classes coordinate UI state and user commands. Repositories organize access to media and application data. Services wrap package-level or platform-level operations such as querying local songs, controlling playback, and supporting background media controls. The table below summarizes the recommended repository decomposition and explains the purpose of each part in the Musik application.

Table 6. Repository Structure and Implementation Evidence for Musik

Repository area	Responsibility	Evidence in Musik
pubspec.yaml	Defines package dependencies, SDK constraints, assets, fonts, and application metadata.	Provides the dependency foundation for media access, audio playback, state management, routing, fonts, and visual assets used by the Flutter application.
lib/main.dart	Serves as the application entry point and initializes the main Flutter application structure.	Connects the startup flow of Musik with the widget tree, application configuration, and initial service or controller setup.
lib/screens/	Contains user-facing interface screens such as Home, Search, Now Playing, Playlist, Queue, Library, and related views.	Shows that the presentation layer is separated from data access and playback-service logic.
lib/controllers/	Coordinates application logic, shared state, playback commands, and interaction flow.	Supports the role of a controller or view-model layer by connecting UI actions with media and playback behavior.
lib/repositories/	Provides structured access to application data and separates data handling from screen widgets.	Helps isolate media-library data, playlist information, and related data operations from the visual interface.
lib/services/	Wraps technical operations such as audio playback, background behavior, media access, permissions, or package-level integration.	Keeps low-level package and platform behavior outside the UI layer, supporting maintainability and testability.

Troy University Resources

Repository area	Responsibility	Evidence in Musik
lib/models/	Defines typed data entities used by the application.	Provides object-oriented and null-safe representations of music-related data such as songs, artists, playlists, history items, queue items, or playback state.
lib/methods/	Stores reusable helper methods and feature-specific operations.	Reduces duplication by keeping repeated logic outside individual screen files.
lib/utils/	Contains utility code such as constants, formatting helpers, theme values, or shared configuration.	Supports consistency across the user interface and application behavior.
fonts/	Stores custom font files used by the application.	Supports the visual style and branding of Musik.
Root image assets	Stores visual assets such as background.png, logo_foreground.png, and splash1_padded.png.	Provides application branding, splash-screen materials, and static visual resources.
android/ and ios/	Contain platform-specific configuration for mobile builds.	Support device-level behavior such as permissions, media access, app metadata, and native build configuration.
linux/, macos/, web/, and windows/	Contain platform folders generated by the Flutter project structure.	Show that Musik follows Flutter's cross-platform project organization, even though the main product focus is mobile playback.
test/	Contains test files or test scaffolding for unit, widget, or application-level checks.	Provides a location for validating models, controllers, widgets, and playback-related behavior through Flutter's testing workflow.
README.md	Provides project-level documentation.	Gives a human-readable entry point for understanding the repository purpose, setup, and usage context.

The table presents the repository structure of Musik as a layered Flutter application. Each repository area is connected to a specific responsibility in the system, from application bootstrap and user interface screens to state management, repositories, services, models, assets, platform configuration, and testing. This organization makes the codebase easier to understand because each part has a clear role in supporting the music-player workflow.

The table also shows how the implementation structure follows the architectural decisions discussed earlier. Views and screens are responsible for the visible user interface, view models or controllers manage UI state and user commands, repositories organize access to application data, and services handle technical integrations with media-query, playback, background audio, and routing packages. Platform folders such as android/ and ios/ support permissions and media-session behavior, while configuration files such as pubspec.yaml define the package ecosystem used by the application.

Overall, this repository organization supports the maintainability and academic clarity of Musik. It allows the project to be explained not only as a working music player, but also as an example of layered software design in Dart and Flutter. The structure makes important programming language concepts

visible in the codebase, including typed models, null-safety decisions, asynchronous service methods, Stream-based playback updates, modular object-oriented design, and separation of responsibilities across application layers. [4], [5], [15]–[21]

The actual repository structure of Musik shows that the project is organized as a cross-platform Flutter application. The root directory contains platform-specific folders such as android/, ios/, linux/, macos/, web/, and windows/, which indicates that the project can be built for multiple platforms through the Flutter toolchain. The main Dart implementation is placed inside the lib/ directory, while assets, fonts, tests, and configuration files are stored separately.

```
d:\MusicPlayer\
├── android/                # Android platform source and configuration
├── ios/                    # iOS platform source and configuration
├── linux/                  # Linux platform source and configuration
├── macos/                  # macOS platform source and configuration
├── web/                    # Web platform source and configuration
├── windows/                # Windows platform source and configuration
├── fonts/                  # Custom font files used by the application
├── lib/                    # Main Dart source code of the application
│   ├── controllers/        # Application logic and state coordination
│   ├── methods/            # Helper methods and reusable functions
│   ├── models/             # Data models and typed entities
│   ├── repositories/       # Data-access layer and media repositories
│   ├── screens/            # User interface screens
│   ├── services/           # Background services and external package wrappers
│   ├── utils/              # Utilities, constants, and theme configuration
│   └── main.dart           # Application entry point
├── test/                   # Unit tests, widget tests, or test scaffolding
├── background.png          # Static image asset
├── logo_foreground.png     # Application logo asset
├── splash1_padded.png      # Application splash screen asset
├── pubspec.yaml            # Dependencies, SDK constraints, assets, and metadata
└── README.md               # Project documentation
```

The structure in the list replaces the earlier abstract repository framework with evidence from the real codebase. The lib/ directory contains the core implementation of the application and is divided into several responsibility-based folders. The screens/ folder represents the presentation layer, including user-facing pages such as the home screen, library screen, current-playing screen, and playlist-related screens. The controllers/ folder contains application logic and state-management code, including files such as audio_handler.dart and main_controller.dart. The models/ folder stores typed data structures, while repositories/ and services/ separate data access, media handling, playback behavior, and external package integration from the visual interface.

This organization supports the layered architecture discussed in the previous section. Screens focus on rendering the user interface and receiving user interactions. Controllers coordinate state and commands. Repositories provide structured access to media or application data. Services wrap technical operations such as audio playback, background behavior, and platform or package-level features. The presence of android/, ios/, linux/, macos/, web/, and windows/ also shows that Musik follows the standard Flutter project structure for cross-platform development.

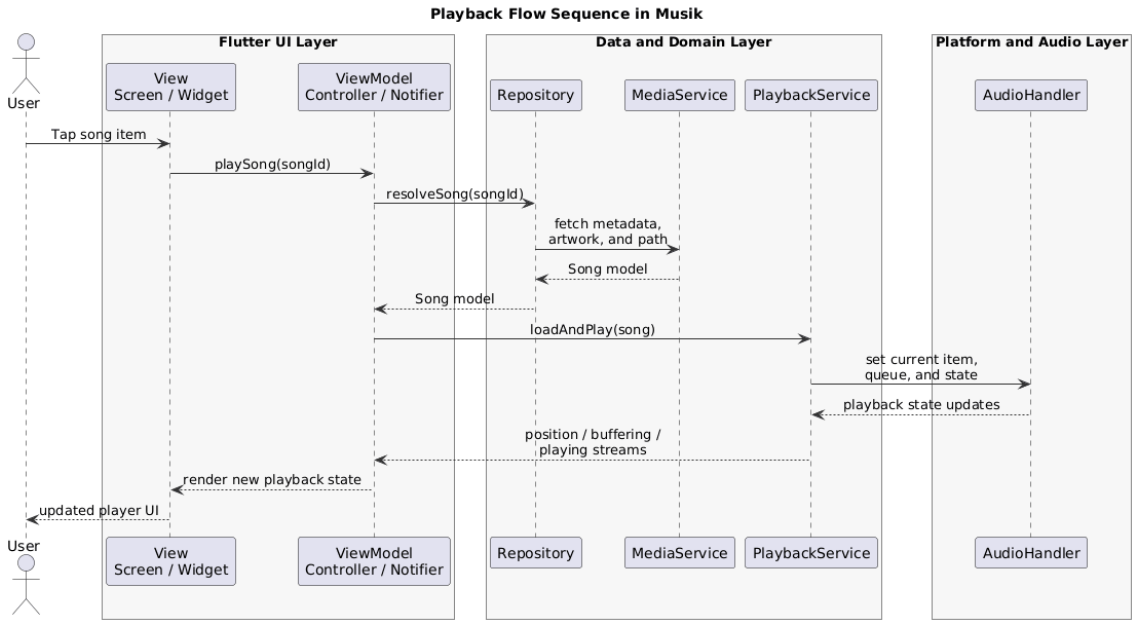
So, the repository structure provides concrete implementation evidence for the design of Musik. It shows that the application is not organized as a single monolithic file, but as a modular Flutter project with separated folders for UI, state management, data models, repositories, services, utilities, tests, assets,

Troy University Resources

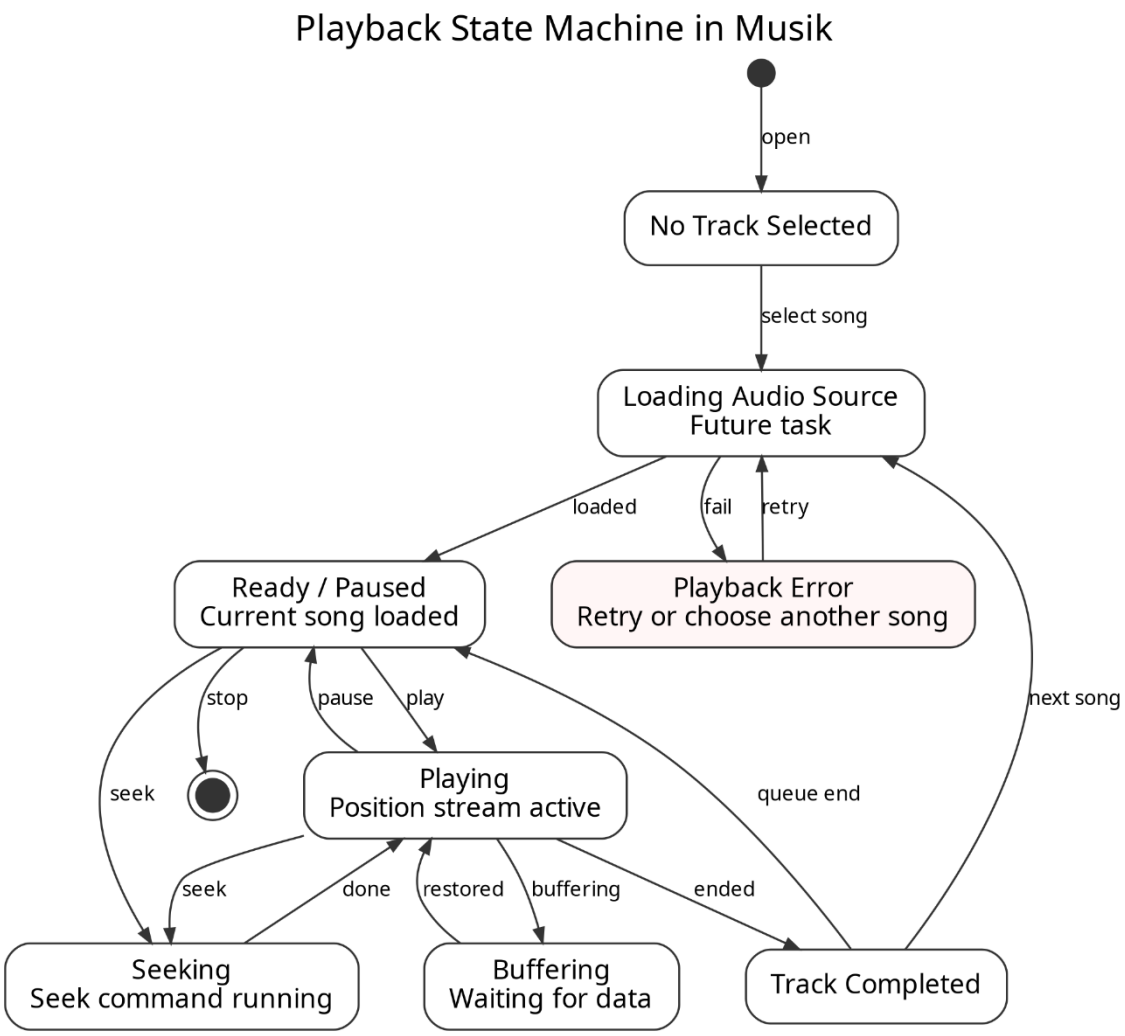
and platform configuration. This structure makes the project easier to understand, maintain, extend, and relate to Dart programming concepts such as object-oriented decomposition, typed models, asynchronous services, and separation of responsibilities.

5. Process Flow of a Typical Playback Action

The runtime flow for selecting and playing a song in *Musik* should look like the figure below.



The sequence in the figure is consistent with the package structure and layered architecture used in Musik. The media-resolution step is supported by ‘on_audio_query‘, which provides access to local audio metadata, artwork, and file information. The playback step is handled by ‘just_audio‘, which provides the core audio player and exposes playback-related values through asynchronous streams. The background and system-control step is supported by ‘audio_service‘, where the audio handler manages media-session behavior, queue information, and playback-state updates. At the architectural level, the sequence also follows Flutter’s recommended separation of responsibilities: the view receives user interaction, the view model coordinates UI-facing commands, the repository resolves application data, and services handle package-level or platform-level operations. This design prevents screens from directly controlling low-level audio APIs and keeps the playback flow easier to maintain, test, and extend. [5], [17], [18], [21]



This diagram shows the main playback states of Musik. The user starts with no track selected, chooses a song, waits for the audio source to load, and then moves into ready, paused, or playing behavior. While playing, the system may temporarily enter seeking or buffering states before returning to playback. When a track ends, the application either loads the next song or returns to a ready state if the queue has ended. The error state shows how the application can recover by retrying or allowing the user to choose another song. This diagram supports the report discussion of Future-based playback commands and Stream-based playback updates. [8], [17], [18]

6. Use of Selected Programming Language Features in Implementation

The methodology of Musik is also a methodology of language use. The project does not only apply a software architecture, but also demonstrates how Dart language features support the implementation of a realistic mobile application. Therefore, the design of Musik connects architectural decisions with concrete language-level mechanisms such as typed classes, generic collections, information hiding, null safety, asynchronous operations, streams, and state management.

A suitable pattern for entity modeling in Musik is the use of typed classes to represent important application concepts such as Song, AlbumGroup, PlaylistSummary, PlaybackSnapshot, and PermissionState. These entities help transform raw media information into structured application data. Dart is appropriate for this purpose because it supports object-oriented programming, generic types, and type inference. As a result, Musik can represent songs, playlists, queue items, and playback information through clear

Troy University Resources

data models instead of relying on loosely typed maps or unstructured values. This improves readability, reduces type-related mistakes, and makes the code easier to maintain. [7]

Information hiding is another important language-level design choice. In Dart, identifiers that begin with an underscore are private to their library. This convention can be used in Musik to hide low-level implementation details and expose only the public methods or properties that other layers need. For example, a playback service can keep its internal audio player object private while exposing public methods such as play, pause, seek, or loadAndPlay. Similarly, a repository can hide raw package responses and return clean application models to the view-model layer. This prevents UI components from bypassing repositories or directly mutating playback internals. [7]

Asynchronous programming is central to the implementation of Musik. Some operations, such as requesting media permission, scanning local songs, initializing services, or loading an audio source, are one-time tasks that complete later. These operations are naturally represented using Future and async/await. Other values, such as playback position, duration, buffering status, player state, and queue changes, update continuously while music is playing. These values are better represented using Stream. This distinction allows Musik to separate one-time asynchronous actions from continuous reactive updates, which is essential for a responsive music-player interface. [8], [17]

View-state management also follows the distinction between local widget state and shared application state. Strictly local state, such as a temporary animation, selected tab, or small widget interaction, can remain inside a StatefulWidget. However, session-relevant state such as the current song, queue, play/pause status, playback position, and media library was moved into shared view models, controllers, repositories, or services exposed through Provider. This separation prevents individual screens from owning state that must remain consistent across the Home screen, Now Playing screen, queue panel, and system media controls. [10], [11]

The following pseudo-code sketch illustrates how these language-level ideas can be expressed in the design of Musik. It shows the relationship between typed models, private fields, Future-based commands, Stream-based playback updates, and shared application state.

```
class PlayerViewModel extends ChangeNotifier {
  PlayerViewModel(this._playbackRepository);

  final PlaybackRepository _playbackRepository;

  MediaItemModel? _current;
  Duration _position = Duration.zero;
  Duration _buffered = Duration.zero;
  bool _isPlaying = false;

  MediaItemModel? get current => _current;
  Duration get position => _position;
  Duration get buffered => _buffered;
  bool get isPlaying => _isPlaying;

  Future<void> initialize() async {
    await _playbackRepository.initialize();
    _playbackRepository.mediaItemStream.listen((item) {
      _current = item;
      notifyListeners();
    });
    _playbackRepository.playbackStateStream.listen((state) {
      _position = state.position;
```

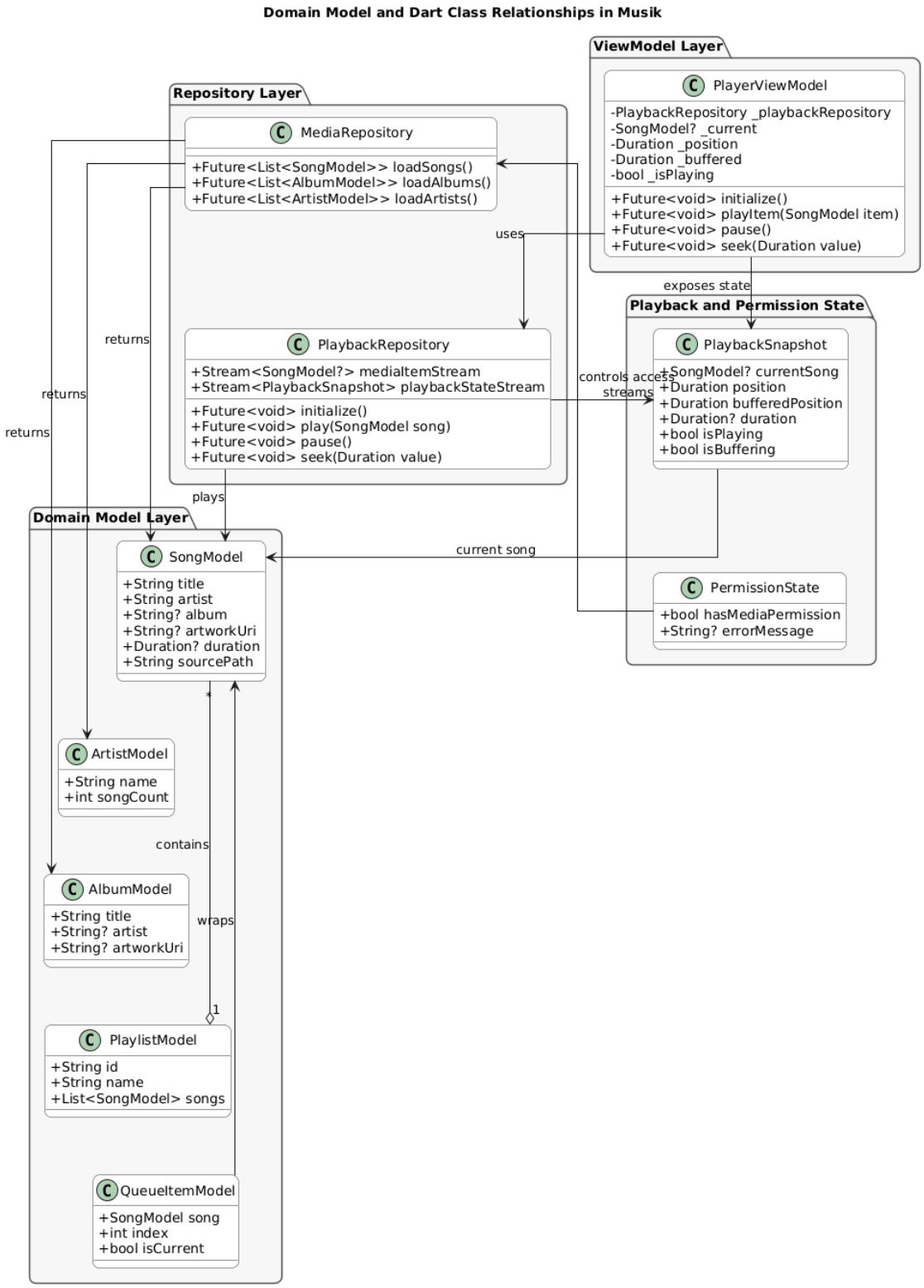
```
      _buffered = state.bufferedPosition;
      _isPlaying = state.playing;
      notifyListeners();
    });
  }

  Future<void> playItem(MediaItemModel item) async {
    await _playbackRepository.play(item);
  }

  Future<void> pause() => _playbackRepository.pause();
  Future<void> seek(Duration value) => _playbackRepository.seek(value);
}
```

The design is consistent with the architecture and package choices of Musik. The use of ChangeNotifier allows the view model to notify the Flutter interface when playback-related values change. The view model does not control the audio engine directly, but communicates with a PlaybackRepository, which follows the layered responsibility between views, view models, repositories, and services. The stream-based updates also match the playback model of packages such as just_audio and audio_service, where media item changes, player state, playback position, and queue information are exposed over time. Therefore, the pseudo-code demonstrates a realistic way to connect Flutter state management with Dart's asynchronous programming model in a music-player application. [5], [11], [17]–[19]

Troy University Resources



The figure presents the domain model and Dart class relationships used to explain the Musik implementation. The diagram focuses on typed entities such as SongModel, ArtistModel, AlbumModel, PlaylistModel, QueueItemModel, PlaybackSnapshot, and PermissionState. These classes represent the main data concepts of a music-player application, including songs, albums, artists, playlists, queue items, playback state, and media permission state. Object-oriented modeling also supports separation of responsibility. MediaRepository is responsible for loading structured media data, while PlaybackRepository

provides playback-related commands and streams. PlayerViewModel depends on PlaybackRepository instead of controlling the audio engine directly. This keeps the user interface separate from playback internals and makes the code easier to maintain and test.

This class diagram is important for the programming-language analysis because it makes Dart features visible at the design level. Typed fields show static modeling, nullable fields such as String?, Duration?, and SongModel? show null-safety-conscious design, List<SongModel> demonstrates generic collections, Future<void> demonstrates one-time asynchronous operations, and Stream<PlaybackSnapshot> demonstrates reactive playback-state updates. Therefore, the diagram connects the structure of Musik directly to Dart concepts such as object-oriented design, type safety, null safety, generics, Future, and Stream.

7. Build, Run, and Test Method

The build and run process of Musik follows the standard workflow of a Flutter application. The repository is opened from the project root directory, where the pubspec.yaml file defines dependencies, assets, SDK constraints, and project metadata. Before running the application, dependencies are resolved using flutter pub get. The source code is then checked with flutter analyze to detect syntax, type, and style-related issues. If tests are available, flutter test is used to execute unit tests or widget tests. Finally, flutter run is used to launch the application on a connected emulator or physical device.

The following command block summarizes the build and run workflow used for Musik:

```
cd d:\MusicPlayer

flutter pub get
flutter analyze
flutter test
flutter run
```

The command flutter pub get resolves the packages declared in pubspec.yaml, including the dependencies required for media access, playback, state management, and routing. The command flutter analyze checks the Dart source code for potential errors and ensures that the project follows Flutter and Dart analysis rules. The command flutter test runs the available test files in the test/ directory. Finally, flutter run builds and launches Musik on the selected target device. This workflow provides a clear and repeatable way to verify that the repository can be installed, analyzed, tested, and executed from the project root.

This subsection presents the build, run, and test workflow of Musik as part of the project methodology. By including dependency installation, static analysis, testing, and application execution, the report treats Musik as a reproducible software artifact rather than only a described product. The Flutter command-line tool supports this workflow through standard commands for analyzing, testing, building, and running Flutter applications. Therefore, the inclusion of these commands strengthens the technical reliability of the report and shows that the project can be checked through the normal Flutter development process. [15]

Troy University Resources

```
PS D:\MusicPlayer> flutter pub get
Resolving dependencies...
Downloading packages...
  audio_session 0.1.25 (0.2.3 available)
  bloc 8.1.4 (9.2.1 available)
  cli_util 0.4.2 (0.5.1 available)
  code_assets 1.0.0 (1.2.1 available)
  flutter_bloc 8.1.6 (9.1.1 available)
  flutter_launcher_icons 0.13.1 (0.14.4 available)
  flutter_lints 2.0.3 (6.0.0 available)
  flutter_native_splash 2.4.7 (2.4.8 available)
  freezed_annotation 2.4.4 (3.1.0 available)
  hooks 1.0.3 (2.0.1 available)
  image 4.8.0 (4.9.1 available)
```

```
PS D:\MusicPlayer> flutter run
Launching lib\main.dart on ANY NX1 in debug mode...
Running Gradle task 'assembleDebug'...
```

Results and Discussion

The results of Musik can be discussed at two levels: the product level and the language-and-architecture level. At the product level, Musik demonstrates the main behavior of a mobile music player, including browsing available songs, displaying media metadata, starting playback, showing the current track, updating playback progress, managing the play queue, and supporting system-level playback controls. These results show that the application is not only a static interface, but a functional media application with an interactive playback flow. At the language-and-architecture level, Musik demonstrates how Dart and Flutter can be used to build a responsive and maintainable mobile application. The project applies typed models to represent music-related entities, null safety to handle incomplete media information, Future-based asynchronous methods for one-time operations, Stream-based updates for continuous playback state, and ChangeNotifier or Provider-based state propagation for reactive user interface updates. These features connect the implementation directly to the programming language concepts discussed in the course.

The product results are visible through the application screenshots and interaction flow. The Home or Library screen shows that users can browse music content and view basic metadata such as title, artist, artwork, and duration. The Now Playing screen shows the current media item, playback controls, progress bar, and play/pause state. The Play Queue panel shows the current and upcoming songs, making the playback order visible to the user. The system media control screen shows that playback can be controlled outside the main application interface, which is important for a realistic mobile music-player experience.

The architectural results are visible through the layered organization of the project. Screens and widgets are responsible for rendering the user interface, while controllers or view models coordinate UI state and user commands. Repositories and services handle media access, playback behavior, background audio integration, and other technical operations. This separation reduces the risk of placing all logic inside screen widgets and makes the project easier to maintain, test, and extend.

Overall, Musik successfully demonstrates the relationship between product functionality and programming language design. The application provides a practical music-player workflow while also showing how Dart features support real software development. The results confirm that Dart and Flutter are suitable for this type of project because they combine cross-platform UI development, strong typing, null safety, asynchronous programming, reactive updates, and a package ecosystem that supports media access and playback integration.

1. Obtained Functional Results

Table 7. Functional results of Musik

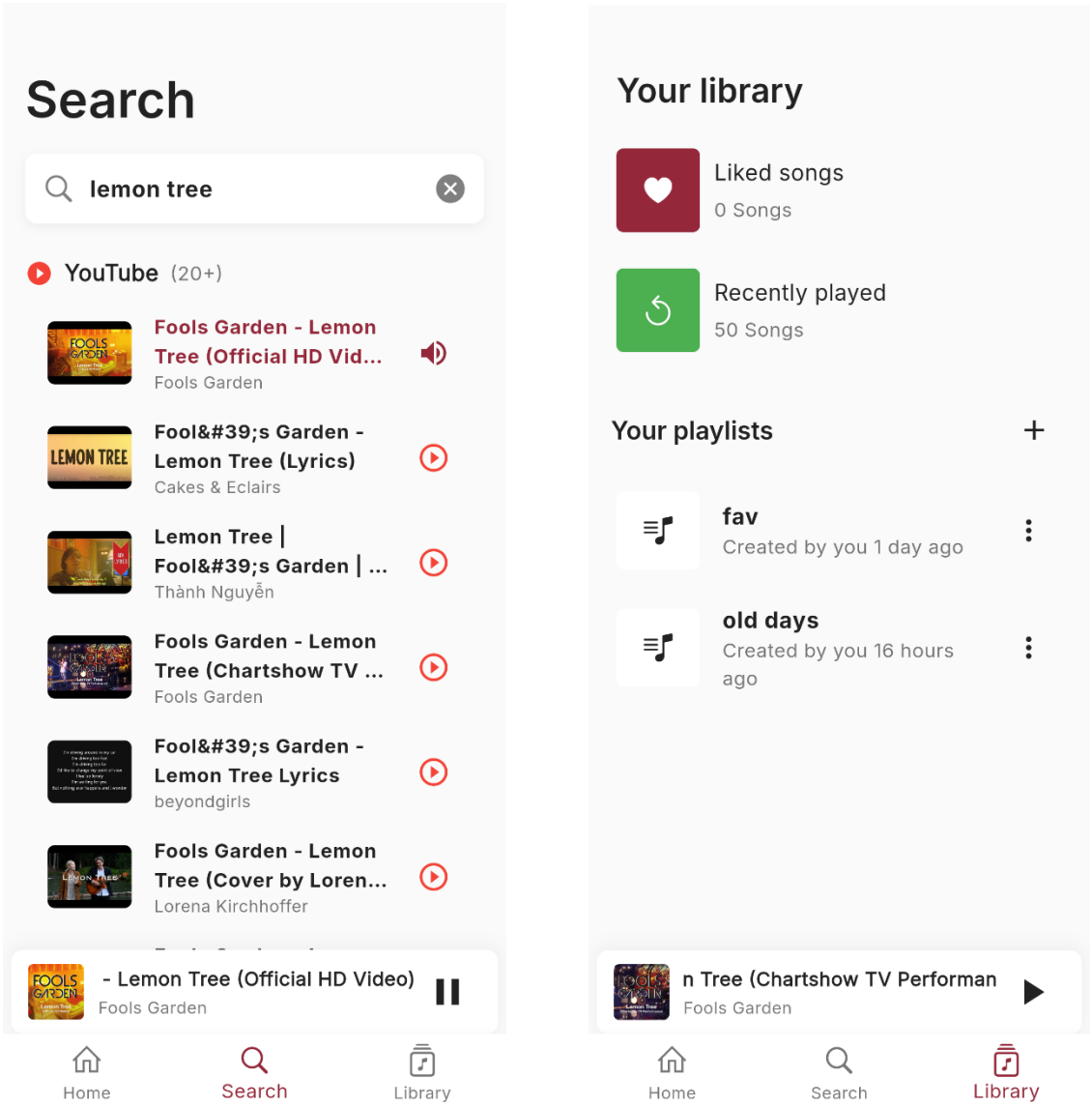
Functional area	Result in Musik	Technical basis	Evidence in the report
Media discovery	Musik provides access to songs, albums, artists, playlists, and artwork from device storage.	on_audio_query supports local media querying and artwork access. [21]	Home / Library screen, permission and library loading state machine, package discussion.
Playback	Musik supports loading audio and controlling playback through play, pause, stop, seek, previous, and next actions.	just_audio provides the core playback engine and playback-control APIs. [16], [17]	Now Playing screen, playback sequence diagram, playback state machine.

Troy University Resources

Functional area	Result in Musik	Technical basis	Evidence in the report
Queue behavior	Musik supports playback order through queue display and next/previous navigation.	just_audio supports playlist-style playback, while audio_service supports queue and media-session behavior. [16]–[18]	Play Queue screen, requirement-to-feature mapping, playback flow discussion.
Background operation	Musik supports playback behavior beyond the foreground screen through system media controls and background media-session integration.	audio_service supports notification controls, lock-screen controls, media buttons, and background audio handling. [18]	System media control screenshot, application lifecycle and background playback state machine.
Reactive progress UI	Musik updates playback position, buffered position, duration, and play/pause state during playback.	just_audio exposes playback-related values through streams, which can be connected to Flutter's reactive UI model. [17]	Now Playing screen, state-domain diagram, PlayerViewModel discussion.
Permissions and media access	Musik handles media-read permission before loading local audio content.	on_audio_query documents platform permission requirements for local media access. [21]	Requirement table, permission and library loading state machine, Android/iOS platform discussion.

The table above summarizes the functional results of Musik by connecting each major feature area with both technical support and report evidence. The purpose of the table is to show that the application results are not described only in general terms, but are linked to observable screens, diagrams, code excerpts, and package responsibilities. Media discovery and metadata display are supported by the Home or Library screen, playback control is supported by the Now Playing screen and playback sequence diagram, queue behavior is supported by the Play Queue panel, and background or system-level playback is supported by the system media control screenshot.

The table also connects these visible results with the technical foundation of the project. Packages such as on_audio_query, just_audio, and audio_service provide support for media querying, playback control, playback-state streams, queue behavior, and system media integration. These packages help explain why the observed features are technically realistic in a Flutter application. At the same time, the repository structure, package dependency diagram, and PlayerViewModel pseudo-code show how the implementation separates UI, state management, repositories, and services.



2. Analysis of Language Use and Software Design Ability

The strongest part of Musik as a programming-language project is that it naturally demonstrates object-oriented design. The project domain contains many concepts that can be represented as structured entities and responsibilities, including songs, albums, artists, playlists, queue items, playback state, repositories, services, and view models. Dart supports this decomposition through object-oriented programming, strong typing, generic collections, and library-level privacy. As a result, Musik can be explained not only as a collection of app features, but also as a system of cooperating objects with clear responsibilities. [7]

The second major strength is asynchronous programming. Audio applications are a suitable environment for demonstrating Future, async/await, Stream, and reactive updates. In Musik, operations such as service initialization, permission requests, media loading, and playback commands can be represented as Future-based tasks because they complete once after some delay. In contrast, playback position, buffering state, player status, current media item, and queue changes are naturally represented as Stream-based values because they continue to change while the user listens to music. This shows how Dart's asynchronous model supports a realistic media application in a readable and structured way. [8], [17]

Troy University Resources

The third major strength is null-safety-conscious design. Media applications often work with incomplete or temporarily unavailable information. Album artwork may be missing, duration may not be known before the audio source is loaded, and there may be no selected song when the application first starts. Dart's sound null safety allows these cases to be represented explicitly in the type system. Nullable types such as ‘MediaItemModel?‘ make the absence of a current item visible in the code, while safe default values such as ‘Duration.zero‘ help keep the initial playback state predictable. This improves reliability and reduces the risk of unexpected null-related runtime errors. [9]

The fourth major strength is declarative interface thinking. In Flutter, the user interface is rebuilt from state rather than being manually mutated element by element. This model is well suited to a music player because one playback-state change may affect several visible elements at the same time, such as the play/pause icon, progress bar, current song title, queue highlight, and system media controls. By using shared state and reactive notification mechanisms, Musik can keep different parts of the interface synchronized with the playback engine. This demonstrates not only Flutter syntax, but also Flutter's state-driven programming model. [6], [10], [11]

The fifth major strength is architecture-aware package composition. Instead of wiring audio and media packages directly into screen widgets, Musik can organize them behind services, repositories, and view models. In this structure, ‘on_audio_query‘ is responsible for media access, ‘just_audio‘ provides playback control, ‘audio_service‘ supports background and system-level media integration, and ‘provider‘ exposes shared state to the widget tree. This separation helps prevent the user interface from depending directly on low-level package APIs. It also makes the project easier to maintain, test, and extend. [5], [18], [19], [21], [22]

Overall, these strengths show that Musik is a suitable project for analyzing Dart as a programming language. The application demonstrates object-oriented decomposition, strong typing, null safety, asynchronous programming, Stream-based reactive updates, declarative UI development, and modular architecture. These features are not only discussed theoretically, but are connected to the actual needs of a music-player application, such as media discovery, playback control, queue management, progress updates, and system media integration.

3. Design Pattern Discussion

The primary architectural pattern applied in Musik is an MVVM-inspired structure with repository and service boundaries. This structure fits the application because Musik contains several responsibilities that need clear separation, including user interface rendering, playback state coordination, media-library access, queue management, and platform-level audio control. In this architecture, views render the interface and receive user actions, view models or controllers manage UI-facing state and expose commands, repositories organize access to application data, and services wrap lower-level package or platform operations such as media querying, audio playback, and background media control. This structure gives Musik a clearer separation of responsibility than a screen-based implementation. [5]

In Musik, this architecture maps directly onto the music-player workflow. The Home/Library screen displays songs and metadata, while media querying is handled through the media repository and service layer. The Now Playing screen presents playback state, while playback commands and player-state updates are coordinated through the player controller and playback service. The Play Queue panel shows the current and upcoming songs, while queue updates are managed outside the UI layer. This separation keeps package-specific code away from screen widgets and makes the interface easier to maintain.

The MVVM-inspired structure is applied flexibly rather than as a rigid template. Its value in Musik comes from reducing coupling between the interface, media data, and playback logic. Temporary presen-

tation state remains inside individual widgets when it only affects a local UI element. Shared state, such as the current song, playback progress, queue state, media-library state, and playing status, is managed at the application level through view models or controllers. This approach follows Flutter's distinction between local widget state and broader application state. [10]–[12], [22]

For a course-scale application, Provider-based MVVM gives Musik a balanced architecture. A setState()-only design is simple for isolated screens, but it becomes difficult to maintain when multiple screens depend on the same playback state. A heavier Clean Architecture-style design provides stronger formal separation, but it adds unnecessary complexity for the scope of this project. Provider with ChangeNotifier, combined with repository and service boundaries, gives Musik enough structure to support shared state, asynchronous playback, media access, and reactive UI updates without making the implementation overly complex. [10]–[12], [19]

This architectural choice also demonstrates how the project applies design patterns according to the needs of the music-player domain. Session-level state such as the current track, playback position, queue, and playing status is shared consistently across the Home screen, Now Playing screen, queue panel, and system media controls. Local interface details remain closer to the widgets that use them. As a result, Musik applies MVVM as a practical organization strategy rather than as a mechanical pattern, making the codebase more maintainable, testable, and easier to explain from a programming-language perspective.

4. Testing Results and Quality Discussion

Testing is an important part of the methodology of Musik because the project uses a layered architecture. When the application separates views, view models, repositories, and services, each layer can be tested at an appropriate level. Flutter distinguishes between unit tests, widget tests, and integration tests. Unit tests are suitable for checking small pieces of logic, widget tests are suitable for checking interface behavior, and integration tests are suitable for verifying complete user flows across the application. [14]

This testing model is relevant to Musik because a music player contains both visible UI behavior and hidden asynchronous logic. A LibraryViewModel can be tested with a fake media repository to verify loading states, permission states, and song-list updates. A PlayerViewModel can be tested with a fake playback repository to verify current song changes, play/pause state, playback position, buffering state, and queue updates. Repository tests can use mocked services to check media-query behavior, data transformation, and error handling. Widget tests can verify that screens such as Home, Now Playing, and Play Queue display the correct state when predefined view-model data is supplied. [13]

The main purpose of the testing strategy is to ensure that Musik is not evaluated only through screenshots or manual demonstration. Testing connects the implementation to verifiable behavior. Unit tests provide fast feedback for models, view models, and repositories. Widget tests check whether the interface reacts correctly to state changes. Integration tests, when included, verify complete user flows such as opening the app, loading songs, selecting a track, starting playback, viewing the queue, and controlling playback. This structure supports the architectural decision to keep UI code separate from business logic and package-level services.

Troy University Resources

Table 8. Quality-assurance strategy for Musik

Test layer	What should be tested in *Musik*	Expected evidence
Unit tests	Song filtering, queue logic, playback-state mapping, permission-state transitions	test/ files, flutter test output
Widget tests	Song list rendering, play/pause button state, seek bar display, navigation actions	Widget-test screenshots or terminal output
Integration tests	End-to-end playback flow, permission grant flow, queue navigation, lifecycle behavior	Emulator/device run logs, integration-test output
Static analysis	Lints, type issues, dead code, import mistakes	flutter analyze output
Manual demo validation	Browse library, select song, play/pause/seek, next/previous, artwork loads	Screen captures or presentation demo checklist

This is methodologically sound because it follows Flutter's official testing model and CLI toolchain. [14], [15]

Beyond functional correctness, Musik was also evaluated in terms of maintainability. A music-player application becomes difficult to extend when the user interface, playback engine, media access, and state management are tightly coupled. For this reason, the maintainability of Musik is considered through the separation between major system responsibilities. The interface, playback logic, media access layer, background playback support, and state-management components are organized so that each part can be modified with limited impact on the others.

Several indicators were used to assess this maintainability. First, playback logic is separated from screen widgets, so the Now Playing screen focuses on displaying playback state and sending user commands instead of directly managing the audio engine. Second, media access is placed behind repositories or services, allowing songs, metadata, artwork, and playlists to be retrieved without exposing platform-specific details to the UI layer. Third, media-session and background playback logic are centralized so that system media controls remain consistent with the application's internal playback state. Fourth, nullable media information is modeled explicitly, because fields such as artwork, duration, or current song can be unavailable at certain moments. Finally, the architecture supports testing by allowing playback logic, view-model state, and interface rendering to be considered separately.

These maintainability indicators are consistent with the layered architecture discussed earlier. Views focus on rendering and user interaction, view models or controllers coordinate state and commands, repositories organize data access, and services wrap platform or package-level behavior. This separation helps reduce coupling, improves testability, and makes future modification safer. As a result, Musik can be evaluated not only by whether it plays music correctly, but also by whether its internal structure supports reliable development, debugging, and extension over time. [5], [22], [23]

5. Final Technical Judgment

As a final judgment, Musik is a strong final-project topic because it combines a product that is easy to demonstrate with several technical concerns that are meaningful for programming-language analysis.

From the user's perspective, Musik is a mobile music player with familiar functions such as browsing songs, viewing metadata, controlling playback, managing the queue, and interacting with system-level media controls. From the implementation perspective, however, these functions require typed data models, asynchronous operations, Stream-based updates, reactive UI rebuilding, state management, package integration, and layered software architecture.

The project demonstrates that Dart is not only syntactically approachable, but also technically capable for building a realistic mobile application. Strong typing and generic collections support clear domain modeling for songs, playlists, queue items, and playback state. Sound null safety helps represent incomplete media information safely, such as missing artwork, unknown duration, or no current song before playback starts. Future and async/await support one-time asynchronous operations such as initialization, permission requests, media loading, and playback commands. Stream supports continuously changing values such as playback position, buffering state, player status, and queue updates.

Flutter complements these Dart features through a declarative UI model and a cross-platform development workflow. In Musik, state changes can be reflected consistently across the Home screen, Now Playing screen, Play Queue panel, and system media controls. The layered architecture also makes the application easier to understand because screens, view models or controllers, repositories, and services have separate responsibilities. This structure improves maintainability, supports testing, and prevents the user interface from depending directly on low-level audio or platform APIs.

Overall, Musik is a suitable and academically valuable project for CS3370 because it connects programming language concepts with a practical software system. The project shows how Dart and Flutter can be used to implement object-oriented decomposition, null-safe data handling, asynchronous programming, reactive state updates, declarative interface design, and modular architecture. Therefore, the value of Musik is not limited to whether the application can play music; its stronger contribution is that it demonstrates how modern programming-language features can support the design and implementation of a maintainable mobile media application.

References

[1] Musik Project Team, "MUSIK – Project Work Plan, Tracking, and Contribution Summary," internal project spreadsheet, 2026.

[2] Faculty of Mathematics and Informatics, "Final Project Report Sample," course reference document, Troy University – Hanoi University of Science and Technology, 2026. Accessed: May 15, 2026.

[3] Purdue Online Writing Lab, "General Format," Purdue OWL. Accessed: May 15, 2026.

[4] Google, "Architecting Flutter Apps," Flutter Documentation. Accessed: May 15, 2026.

[5] Google, "Guide to App Architecture," "Architecture Recommendations and Resources," and "Testing Each Layer," Flutter Documentation. Accessed: May 15, 2026.

[6] Google, "Introduction to Declarative UI," Flutter Documentation. Accessed: May 15, 2026.

[7] Google, "Introduction to Dart," Dart Documentation. Accessed: May 15, 2026.

[8] Google, "Asynchronous Programming," Dart Documentation. Accessed: May 15, 2026.

[9] Google, "Sound Null Safety," Dart Documentation. Accessed: May 15, 2026.

[10] Google, "Differentiate Between Ephemeral State and App State," Flutter Documentation. Accessed: May 15, 2026.

[11] Google, "Simple App State Management," Flutter Documentation. Accessed: May 15, 2026.

[12] Google, "Approaches to State Management," Flutter Documentation. Accessed: May 15, 2026.

Troy University Resources

[13] Google, “Testing Each Layer,” Flutter Documentation. Accessed: May 15, 2026.

[14] Google, “Testing Flutter Apps,” Flutter Documentation. Accessed: May 15, 2026.

[15] Google, “Flutter Command-Line Tool,” Flutter Documentation. Accessed: May 15, 2026.

[16] pub.dev, “just_audio Package,” Flutter package documentation. Accessed: May 15, 2026.

[17] pub.dev, “just_audio Example,” Flutter package documentation. Accessed: May 15, 2026.

[18] pub.dev, “audio_service Package,” Flutter package documentation. Accessed: May 15, 2026.

[19] pub.dev, “provider Package,” Flutter package documentation. Accessed: May 15, 2026.

[20] pub.dev, “go_router Package,” Flutter package documentation. Accessed: May 15, 2026.

[21] pub.dev, “on_audio_query Package,” Flutter package documentation. Accessed: May 15, 2026.

[22] M. Fuksa, S. Speth, and S. Becker, “MVVM Revisited: Exploring Design Variants of the Model-View-ViewModel Pattern,” arXiv:2504.18191, 2025, doi: 10.48550/arXiv.2504.18191.

[23] W. Sheikh and N. Sheikh, “A Model-View-ViewModel (MVVM) Application Framework for Hearing Impairment Diagnosis,” arXiv:1911.08289, 2019, doi: 10.48550/arXiv.1911.08289.

[24] Bladz1, “MusicPlayer,” GitHub repository. Accessed: May 15, 2026.

Appendices

The appendices collect project artifacts used to support the requirements, architecture, programming-language features, implementation methodology, testing strategy, including the project work plan, contribution tracking, repository structure, dependency evidence, testing records, and manual validation checklist.

Appendix A. Project Work Plan and Contribution Tracking

This appendix supports the contribution summary in the main report by showing how each member participated in the project and how the final application was built through an incremental team process.

- **Project planning and scope definition**
  - The team defined Musik as a Dart and Flutter-based mobile music player.
  - The main project scope included music browsing, metadata display, playback control, queue management, playlist support, search, and background playback.
  - [STUDENT_1] coordinated the initial planning process and helped align the project scope with the course requirements.
  - [STUDENT_3] supported requirement clarification, MVP definition, and architecture planning.
  - [STUDENT_2], [STUDENT_4], and [STUDENT_5] also contributed ideas related to user interface, playlist behavior, search flow, and final demonstration requirements.
- **Environment setup and architecture planning**
  - The Flutter development environment was configured for the project.
  - The initial repository structure and coding conventions were discussed.
  - [STUDENT_3] helped configure the Flutter project and establish the navigation and architecture foundation.
  - [STUDENT_1] supported architecture discussion, technical coordination, and integration planning.
  - The team selected the main packages used in the application, including just_audio, audio_service, on_audio_query, provider, and go_router.
- **Core playback and application flow**
  - [STUDENT_3] contributed to the local music scanning foundation, core playback engine, seek bar logic, playback modes, and background playback.
  - [STUDENT_1] supported playback implementation, seek bar and timing display, shuffle and repeat states, playlist actions, and background playback behavior.
  - These tasks helped connect the music library, Now Playing screen, playback controls, queue panel, and system media controls into a consistent listening flow.
- **Data, playlist, and storage features**
  - [STUDENT_2] focused on the data-oriented and user-library features of Musik.
  - He worked on the music catalog foundation, song model, artist model, playlist model, history item model, and local cache mapping.
  - He implemented playlist management features such as creating playlists, renaming playlists, deleting playlists, adding songs to playlists, and removing songs from playlists.

Troy University Resources

  - He also supported local storage, listening history, liked songs, genre/language filtering, and unified history behavior.

- **Search, API integration, and optimization**
  - [STUDENT_4] supported the technical infrastructure, search-related features, API integration, and optimization work.
  - He implemented search by song name and artist.
  - He improved catalog loading behavior and supported thumbnail/cache optimization.
  - He worked on YouTube Data API v3 integration, API response parsing, empty-state handling, failed-request handling, and connecting YouTube-related data into the listening flow.
  - He also supported regression testing, bug fixing, performance improvement, and final testing for mixed local and online content flows.

- **User interface and visual interaction flow**
  - [STUDENT_5] was responsible for user interface implementation and visual interaction flow.
  - He transformed the planned screen structure into usable Flutter interfaces.
  - His work included the Home screen, Search screen, Now Playing screen, Playlist-related screens, song list display, artist information display, and mini-player layout.
  - He also supported bottom navigation, application theme, catalog UI, seek bar presentation, playlist interface, search UI, autocomplete behavior, album views, and genre views.
  - In the final stage, he helped collect screenshots, prepare documentation materials, run functional tests, and rehearse the group presentation.

- **Integration, testing, and final documentation**
  - [STUDENT_1] acted as the main coordinator for integration, testing evidence, final documentation, and quality control.
  - [STUDENT_3] supported integration testing, bug fixing, final build consolidation, and report preparation.
  - [STUDENT_4] supported regression testing, bug fixing, optimization, and final testing.
  - [STUDENT_5] supported screenshot collection, UI checking, documentation materials, and presentation preparation.
  - [STUDENT_2] supported verification of data, playlist, local storage, history, and liked-song behavior.
  - The team tested the main functional flows, including browsing songs, playing music, controlling playback, managing playlists, searching music, and using background playback controls.

- **Individual contribution evidence**
  - [STUDENT_1] contributed as the team leader, integration coordinator, playback supporter, testing organizer, documentation editor, and final quality controller.
  - [STUDENT_2] contributed to music catalog data, playlist management, local storage, listening history, liked songs, and data model support.
  - [STUDENT_3] contributed to project setup, architecture planning, navigation, local scanning, playback engine, seek bar logic, playback modes, and background playback.

- [STUDENT_4] contributed to search features, YouTube API integration, API response handling, cache optimization, performance improvement, and mixed-content testing.
- [STUDENT_5] contributed to the main UI screens, navigation-facing components, playlist interface, Now Playing interface, search UI, visual refinements, screenshots, and presentation materials.

Appendix B. Repository Structure and Dependency Evidence

The project follows the standard structure of a cross-platform Flutter application, with platform folders for Android, iOS, Linux, macOS, web, and Windows. The main Dart implementation is located in the lib/ directory, while tests, fonts, assets, configuration files, and documentation are stored separately. This organization supports the layered architecture discussed in the main report. [24]

```
d:\MusicPlayer\
├── android/                # Android platform source and configuration
├── ios/                    # iOS platform source and configuration
├── linux/                  # Linux platform source and configuration
├── macos/                  # macOS platform source and configuration
├── web/                    # Web platform source and configuration
├── windows/                # Windows platform source and configuration
├── fonts/                  # Custom font files used by the application
├── lib/                    # Main Dart source code of the application
│   ├── controllers/        # Application logic and state coordination
│   ├── methods/            # Helper methods and reusable functions
│   ├── models/             # Data models and typed entities
│   ├── repositories/       # Data-access layer and media repositories
│   ├── screens/            # User interface screens
│   ├── services/           # Background services and external package wrappers
│   ├── utils/              # Utilities, constants, and theme configuration
│   └── main.dart           # Application entry point
├── test/                   # Unit tests, widget tests, or test scaffolding
├── background.png          # Static image asset
├── logo_foreground.png     # Application logo asset
├── splash1_padded.png      # Application splash screen asset
├── pubspec.yaml            # Dependencies, SDK constraints, assets, and metadata
└── README.md               # Project documentation
```

Repository area	Role in the architecture	Evidence value
lib/screens/	Presentation layer	Contains user-facing screens such as Home, Search, Now Playing, Queue, Library, and Settings.
lib/controllers/	State coordination and commands	Coordinates UI state, playback commands, and interaction logic.
lib/models/	Typed domain models	Represents songs, playlists, queue items, playback state, and other structured data.
lib/repositories/	Data-access boundary	Organizes access to media data and hides raw source details from the UI layer.
lib/services/	Technical integration layer	Wraps platform or package-level behavior such as playback, media query, and background control.

Troy University Resources

Repository area	Role in the architecture	Evidence value
pubspec.yaml	Dependency and asset configuration	Lists Flutter packages, fonts, assets, SDK constraints, and project metadata.
android/ and ios/	Mobile platform configuration	Stores platform-specific files such as permissions and media-session-related configuration.
test/	Testing area	Stores unit tests, widget tests, or other test scaffolding for project validation.

The package dependency evidence is primarily taken from pubspec.yaml. The most important packages for Musik are related to media discovery, audio playback, background media control, state management, and navigation. These dependencies support the technical explanation in the main report because each package is connected to a specific architectural responsibility.

Dependency	Primary role in Musik	Evidence value
on_audio_query	Local media discovery and metadata access	Supports song, album, artist, playlist, artwork, and media-permission workflows.
just_audio	Core audio playback engine	Supports loading audio sources, playback control, seeking, and playback-state streams.
audio_service	Background playback and system media controls	Supports notification controls, lock-screen controls, media-session behavior, and queue integration.
provider	State exposure and dependency injection	Supports shared view-model or controller state across the widget tree.
go_router	Declarative navigation	Supports structured movement between Home, Search, Now Playing, Queue, and supporting screens.

Appendix C. Testing and Manual Validation Evidence

Because the application uses a layered architecture, testing can be divided across unit tests, widget tests, integration tests, static analysis, and manual demo validation. [14]

Test layer	Validated behavior in Musik	Evidence record
Unit tests	Song filtering, queue logic, playback-state mapping, and permission-state transitions are checked at the logic level.	test/ files and flutter test terminal output.
Widget tests	Song-list rendering, play/pause button state, seek bar display, queue display, and navigation actions are checked at the interface level.	Widget-test files, screenshots, or terminal output.
Integration tests	End-to-end playback flow, permission-grant flow, queue navigation, and application lifecycle behavior are checked across multiple layers.	Emulator/device run logs or integration-test output.
Static analysis	Lints, type issues, dead code, invalid imports, and nullable-value warnings are checked before execution.	flutter analyze terminal output.

Test layer	Validated behavior in Musik	Evidence record
Manual demo validation	Library browsing, song selection, play/pause/seek, next/previous controls, queue opening, artwork display, and system media controls are verified through the running application.	Screen captures, demo checklist, or presentation evidence.

The following command block summarizes the repeatable verification workflow for the Musik repository.

```
cd d:\MusicPlayer

flutter pub get
flutter analyze
flutter test
flutter run
```

Manual validation item	Expected result	Status
Open application	Application launches without crash and displays the main library or home screen.	Pass
Grant media permission	Permission flow is handled correctly and the song library can be loaded.	Pass
Browse library	Song titles, artists, artwork, and duration are shown where available.	Pass
Select a song	Selected song opens or updates the Now Playing interface.	Pass
Play and pause	Play/pause state changes correctly in the UI and audio behavior.	Pass
Seek progress	Seek bar can move playback position and UI progress updates afterward.	Pass
Next and previous	Playback moves to the expected queue item and metadata updates correctly.	Pass
Open play queue	Current and upcoming songs are shown in the queue panel.	Pass
System media controls	Playback information and controls appear outside the foreground application.	Pass
App lifecycle behavior	Playback and UI state remain reasonable when the app is minimized and reopened.	Pass

The testing evidence supports the maintainability argument made in the main report. Clear separation between views, controllers, repositories, and services allows playback logic to be validated separately from interface rendering, while screens can be checked with predefined state.

Troy University Resources