INDY-02-PlaylistSyncer
SOFTWARE DESIGN DOCUMENT (SDD)

CS4850-02

Spring 2024

Professor Perry

2/13/2024

Team

Roles	**Name**	**Major responsibilities**	**Cell Phone / Alt Email**
Team leader	Nikita Smith	Developer, Manager	678-628-6076 Nikitasmith6@gmail.com
Team members	Ben Pallotti	Documentation, Developer	770-906-3367 ben.pallotti@gmail.com
	Josh Poore	Programmer, Developer	770-337-5477 joshua.poore@gmail.com
Advisor / Instructor	Sharon Perry	Facilitate project progress; advise on project planning and management.	770-329-3895

Table Of Contents

1. Introduction

The following is the software design specifications document for the PlaylistSyncer Android and iOS application. This document will outline the system's architecture and associated strategies utilized in developing this application given the complex goal of creating data interoperability between several streaming APIs.

1.1. Document Outline

- Introduction
- System Overview
- Design Considerations
    - Assumptions and Dependencies
    - General Constraints
    - Goals and Guidelines
    - Development Methods
- Architectural Strategies
    - 3.1: Programming Language/Framework
    - 3.2: API Integration
    - 3.3: Data Management
    - 3.4: User Interface
    - 3.5: Error Handling
    - 3.6: Concurrency/Synchronization
    - 3.7: Version Control
- System Architecture
    - 4.1: Subsystem Architecture
- Policies and Tactics
    - Policy 1: Complier
    - Policy 2: Coding Guidelines/Conventions
    - Policy 3: Ensuring Requirement Traceability
    - Policy 4: Software Testing
    - Policy 5: Software Maintenance
- Detailed System Design
    - 6.1: Classification
    - 6.2: Definition
    - 6.3: Responsibilities
    - 6.4: Constraints
    - 6.5: Composition
    - 6.6: Uses/Interactions
    - 6.7: Resources
    - 6.8: Processing
    - 6.9: Interface/Exports
    - 6.10: Detailed Subsystem Design
- Glossary
- Bibliography

1.2. Document Description

1.2.1. Introduction

This document will explain in detail our software architecture and design strategies for developing and managing the PlaylistSyncer mobile application. This explanation will include our strategies surrounding our usage of the Flutter framework, handling our development process in a remote-centric environment, and breaking down our system architecture. Our development team should use this document to develop the PlaylistSyncer application and related software products and services. Readers should be familiar with our Software Requirements and Specifications document before proceeding as many of the topics discussed in this report relate to the requirements and application specifications outlined in the Software Requirements and Specifications document.

The remainder of this report will discuss a set of topics that will act as the basis of our application architecture including items such as the essential design constraints, strategies, development policies, and system design. Our design constraints are primarily centered around the usage of external platform APIs to service core functionality as well as the reliance on mobile platforms for application execution. These considerations help inform our architectural strategies for constructing the app. We will use Google's Flutter framework for cross platform development, API integration, user interfaces, as well as handle potential errors and concurrency issues. Systematically we plan to use a set of API and user facing modules that handle the communication and servicing of user requests in conjunction with handling external API facing data handling commands. To maintain consistency and reliability in the development process we have established a set of policies centered around code uniformity, accurate commenting, and a focus on software testing.

1.2.2. System Overview

Our application is split into four specific subsections: authentication, playlist acquisition, playlist generation, and the UI module. The Authentication module should communicate with the user and streaming service servers to acquire access tokens to authenticate our requests in the subsequent playlist acquisition and generation modules. The playlist acquisition module uses user input and data from the authentication module to acquire playlist data from the specified streaming service. Once this data is acquired it is fed into the playlist generation module which handles all the necessary lookups and data processing to generate a playlist compatible with the specified destination streaming service. The UI module is responsible for communicating user input to all the other modules and displaying any necessary data to the user because of the processing done by other modules. These modules working together result in an app that allows users to authenticate their streaming account and then subsequently transfer playlist data between them as is specified by the user.

2. Design Considerations

2.1. Assumptions and Dependencies

Our primary dependency is on the access to music streaming services and the ability to accurately receive and send data to these platforms. This dependency also assumes that our

users will have accounts with at least 2 of the three supported streaming services. Additionally, we depend on mobile devices with Android or iOS operating systems for hosting our application and handling the transmission of application critical data through Wi-Fi and cellular data networks. As a result of our reliance on streaming services, in the future we may need to adjust our list of supported services to account for any listed services not allowing for the request and generation of playlists by third-party applications such as our app.

2.2. General Constraints

Given our usage of private user data, storing data securely is a key consideration along with working with streaming services to verify our user account authentication systems match their security standards. Additionally, as an extension of our application's dependance on Android and iOS, we are constrained by the policies and standards set by Google and Apple respectively for apps hosted on their OS platforms. The low processing overhead of our app should allow users of low-end mobile devices to still make usage of our app without any studders or potential unresponsive app behaviors. From a user experience standpoint another constraint to consider is the screen sizes used by our target mobile devices, as they dictate the quality and quantity of information that can be shown to the user.

2.3. Goals and Guidelines

The software should be simple to use, responsive, and transparent in what actions the software is performing at any given time. The UI should be accessible and purposeful, allowing users to quickly navigate and fully understand all the accessible UI functions immediately. This focus on simplification and transparency carries over to our system architecture as well, as every module, function, and class should be easy to understand and extend so developers can build the software efficiently. Furthermore, to remain in line with our software goals, our system architecture should be as responsive as possible even if it comes at a memory requirement cost.

2.4. Development Methods

Our team will utilize a scrum-based agile model with a focus of using stories, code sprints, and rapid iteration to develop our application. This development model aligns with our rapid development timeline and should allow our team members to get feedback on their work and integrate features as quickly and often as possible.

3. Architectural Strategies

3.1. Programming Language/Framework

We have chosen Dart as our preferred coding language because it is the corresponding coding language for Flutter. Flutter is a strong choice because it is an open-source codebase that allows for application development that are compatible with both iOS and Android. It is known for high performance and UI development tools that will allow us to freely develop our application.

3.2. API Integration

We can integrate Spotify, Apple Music, and YouTube APIs by connecting to each service and fetching playlist data. Getting the users data from these services will require permission from the user, and can be done with OAuth (An open-source authorization service that allows a user to share data with other sites, like how Gmail can be used to sign in to LinkedIn, etc.)

3.3. Data Management:

The goal is to ensure that data is securely withheld within the app to maintain playlists, user details, and login credentials whilst the app is open and running. Once the app is closed, all data on playlists and login credentials will be deleted for safety purposes. We plan to use local data stores to accomplish this.

3.4. User Interface:

Flutter can be used to create interfaces that are compatible with both iOS and Android. The goal is to keep the interface simple enough for even the least tech-savvy users to use. It must have minimal buttons and options, which will keep it simple and allow it to run without delays or crashes.

3.5. Error Handling:

Error handling is essential to ensure the application can overcome common issues that come with using APIs and/or data syncing. The errors that we are expecting to run into during development are failed API calls and issues converting data, though the converting data errors should be only an issue in early development. In case of a failed API call, the program will attempt to send a new call; if several call attempts are made without a response, we will notify the user with a UI message.

3.6. Concurrency/Synchronization:

Strategies for synchronization include making sure data stays the same across different platforms to avoid redundancy and mistakes. Dart uses an async/await feature in Flutter to allow for handling concurrent operations or tasks. We can also use data locking or other synchronization protocols.

3.7. Version Control:

GitHub is a reliable way to control versions and work on code between multiple people. It uses branching and merging to organize new code or integrate code. Our plan for ensuring that no merger issues occur is to have separate branches for each person and ensure developers communicate their progress and work consistently. When a project build needs to be made for testing or product release purposes, we will merge the separate branches to form the final product. This process will be conducted on a scheduled basis to ensure all team members are in sync with one another and the product's state.

4. System Architecture

There are several pieces of this system that we have decided to spilt into separate subsystems. The names of these subsystems are as follows: Login & Password Handling, Authentication, Displaying Screens, Transferring Data/Playlists, and the Handling of APIs. The reason for so many subsystems is for us to easily be able to compartmentalize our program. Each of these subsystems align with at least one of our functional requirements for this program and are covered individually below.

Commented [JP1]: Change

Login & Password Handling entails each user being able to login to each of their accounts, those being the Apple, YouTube, and Spotify music accounts. Currently, we plan to not store user passwords and login details to avoid any security issues with storing their data, though if we have time, it is something we plan to attempt to do. This subsystem is essential to the rest of the overall program, as without it, we would not be able to access each user's information on their music platforms, which would, therefore, make it impossible to accomplish the goal of this program, which is to transfer playlists from one music platform to another.

Authentication has a similar purpose as the last subsystem, because without it we would not be able to get the user's data. This subsystem's role is to establish communication with the servers of each music platform using the login information provided by our users. This will likely be done through each platform's API, as we can ensure safe transfer of user data this way, as our intended APIs use encryption for user data.

Displaying Screens is a subsystem that will allow us to display our application screens onto our devices, whether this be an Android or iOS device. We plan to have 3 main screens, those being the login screen, the transfer screen, and the settings screen. The login screen serves the role of allowing users to login into their music platforms so our app can access their data. The transfer screen serves the role of allowing users to select what platforms they are transferring to and from. The settings screen allows users to customize their experience within the app.

Transferring Data/Playlists is a subsystem that accomplishes our main goal for this program, which is to transfer playlists from one platform to another. Cooperation from the Handling of APIs and Login & Password subsection help to accomplish this. One of the main things that this subsection will have to do is format the data that it receives from the API subsection, i.e. the playlist data, to an acceptable form for the other APIs to handle as input for their respective platforms. Section 4.1 of this document covers this more in-depth.

Handling of APIs, a subsystem, gets and sends data from and to each respective music platform. This solely handles the retrieving and sending aspect of the data transfer process as the actual data conversion is done in the Transferring Data/Playlist subsystem. That said, this subsystem's role is to get the playlist data from the user's music platform account and to create playlists on the account they are transferring their data to.

4.1. Subsystem Architecture

Transferring Data/Playlists has a few subsystems within itself. The first subsystem is the conversion functions, which serve the role of converting the data we receive from the APIs into applicable data for the creation of playlists using the other APIs. We plan to have a master data class where we convert everything to and from, which will be of JSON format. This will require at least 6 functions in the Handling of APIs subsystem: two for every music platform, one for taking in data, and one for sending the data.

5. Policies and Tactics

5.1. Policy 1, Complier -

We plan to use Flutter's compiler through Visual Studio Code as our compiler and Integrated Development environment for the system's development. This is so we are more easily able to fluidly change our development if we are to run into any problems during the development cycle, for example one of our Dart APIs not working as intended. In a case like this, we may need to pivot using a python or web-based API instead, hence why we are using Visual Studio Code as it's extension feature will help us to more smoothly transition if need be.

5.2. Policy 2, Coding Guidelines/Conventions -

We want to make our code as readable as possible, so that if another one of the members of our group needs to understand it, we do not need to waste time explaining it in person. For this reason, we want to comment our code and compartmentalize as much of our code as we can. For example, each function will have a chunk of comments above the function definition itself, describing in at least surface level detail what the function does and what it requires to work properly. In a case-by-case basis, if the function is more complex, comments may be needed in the function code itself to clarify what is happening at certain lines of code. Outside of these guidelines we will follow standard Dart syntax and coding conventions as outlined in Dart's programming documentation.

5.3. Policy 3, Ensuring Requirements Traceability -

We plan to have a 4–5-line block of comments before and after each requirement is completed. Alternatively, developers may put comments at the top of each file, which states what requirements (Ideally notated by number) are being worked on or contributed to in the file. This could then be followed by a 1-line comment before each function that states what requirement it belongs to. In the main file, we could have a master comment that states whether a requirement has been completed in full or not and where all the pieces of said requirement are.

5.4. Policy 4, Software Testing -

To test the software, both during development and after its development, we plan to test it by using a few metrics. Firstly, we plan to test each of our modules after they are complete through the usage of test cases, to make sure their inclusion in the final product won't cause some unforeseen issue that was overlooked. We will also test the efficiency of each performance-critical function. This will help us gauge the best variation of said function when we eventually get to the optimization stage of development. Finally, we plan to attempt to have some black box testing by some of our fellow students, friends, and family members. We will have anyone who wants to test it fill out a report on what issues they found and what they liked about it as well.

5.5. Policy 5, Software Maintenance -

The software codebase will be maintained on a GitHub repository. This GitHub repository will be a separate repository from any of our personal accounts and will be a corporate account. This will allow us all to have a central location to access the program. Any and all changes to the repository must come from a verified team member and contain a concise description of what changes they have made.

6. Detailed System Design

6.1. Classification

Component	Classification
Playlist Sync App	Application/Main System
Playlist Retrieval	Subsystem
Data Storage	Subsystem
Synchronization	Module
User Interface	Subsystem
Authentication/Login	Subsystem
Concurrency Management	Module

6.2. Definition

Component	Definition
Playlist Sync App	Uses many components to synchronize applications from various platforms.
Playlist Retrieval	Manages retrieval of playlists from various platforms.
Data Storage	Stores and manages playlist data.
Synchronization	A module of data storage that ensures synchronization of local and user endpoint playlist/login data.
User Interface	Provides interactive and visual menu for user interaction.
Authentication/Login	Authenticate logins and manage APIs that allow for permission to login with another application.
Concurrency Management	Manage concurrent operations to keep app performance optimal.

6.3. Responsibilities

Component	Responsibilities
Playlist Sync App	Transfer of playlists between Spotify, Apple Music, and YouTube, ensuring data integrity and optimal user experience.
Playlist Retrieval	Fetches playlist data from Spotify, Apple Music, YouTube based on user API requests.
Data Storage	Manage storage of playlist data by caching and indexing for easy, efficient access.

Synchronization	Prevent and resolve conflicts that come with storing data both locally and remotely. Maintain consistency in data.
User Interface	Allow users to browse, select, and manage their playlists.
Authentication/Login	Securely verify user login and communicate with APIs of streaming services via OAuth to grant permission for data exchanges.
Concurrency Management	Ensure prop-er handling of concurrent operations so app is responsive and high-performance.

6.4. Constraints

Component	Constraints
Playlist Sync App	Potentially limited by the capabilities and constraints of underlying APIs. Requires internet connectivity for synchronization.
Playlist Retrieval	Rate limits and usage restrictions imposed by Spotify, Apple Music, and YouTube APIs. We must go by their API terms of service and authentication requirements.
Data Storage	Potential limits on data speed and storage capacity. Data must be stored with integrity.
Synchronization	Network bandwidth limitations, conflicts of going on and offline and synching, and requires efficient code for data resolution.
User Interface	Varying screen sizes can cause visual bugs or unappealing UI. Poor optimization could lead to lag or crashes.
Authentication/Login	Security best practices and encryption standards must be put in place to keep login data secure, especially if they are being used on multiple platforms.
Concurrency Management	Data corruption and race conditions can occur when handling concurrency, so data must be stored with appropriate CPU, memory, bandwidth, etc.

6.5. Composition

Component	Composition
Playlist Sync App	Playlist Retrieval, Data Storage, Synchronization, User Interface, Authentication/Login, and Concurrency Management.
Playlist Retrieval	Spotify, Apple Music, and YouTube APIs and code methods that process the transfers gracefully.
Data Storage	Use of Local memory hosting to maintain data. Endpoint devices will hold their data for synching.
Synchronization	Channels between local and remote data storages to make source data exchange maintains healthy synchronization.
User Interface	Uses Flutter UI components to make a clean interface.
Authentication/Login	Authorization protocols to establish secure connections with APIs.
Concurrency Management	Dart uses an asynchronous programming model that we can use to manage performance with ease.

6.6. Uses/Interactions

Component	Uses/Interactions
Playlist Sync App	Interactions with Playlist Retrieval, Data Storage, Synchronization, User Interface, Authentication/Login, and Concurrency Management.
Playlist Retrieval	Interacts with Spotify, Apple Music, and YouTube APIs to fetch playlist data.

Data Storage	Interacts with the playlist retrieval and synchronization components to store and retrieve playlist data.
Synchronization	Goes with the data storage component to maintain consistent data synch.
User Interface	Requires interaction with all components because all use interaction with the UI lead to a change in data.
Authentication/Login	Interacts with all platform APIs to validate user credentials and access playlist data.
Concurrency Management	Data fetching, processing, and synchronization keeps operations stable.

6.7. Resources

Component	Resources
Playlist Sync App	Resources include device memory and CPU usage during data sync and UI rendering.
Playlist Retrieval	Network bandwidth for making API requests to Spotify, Apple Music, and YouTube retrieve playlist data, which could be subject to API call limits.
Data Storage	Disk space and memory are local storages that are used for storing cached playlist data and user information.
Synchronization	Uses network resources to transfer playlist data between the app and the music platforms. These resources ensure efficient exchange and reduce lag/latency.
User Interface	Uses rendering resources for displaying UI visual elements and handling user interactions, and these resources will
Authentication/Login	Will require network resources for creating connections with authentication server to verify tokens and credentials.
Concurrency Management	Processor and memory resources are used to handle data processing tasks, which will avoid race conditions and deadlock situations with attentive management.

6.8. Processing

Component	Duties/Processes
Playlist Sync App	Using Synchronization algorithms, Flutter UI, concurrency, and error-handling, the app synchronizes playlists data that was fetched from APIs of YouTube, Spotify, and Apple Music.
Playlist Retrieval	Fetches data from APIs of YouTube, Spotify, and Apple Music. We can employ algorithms and HTTP requests

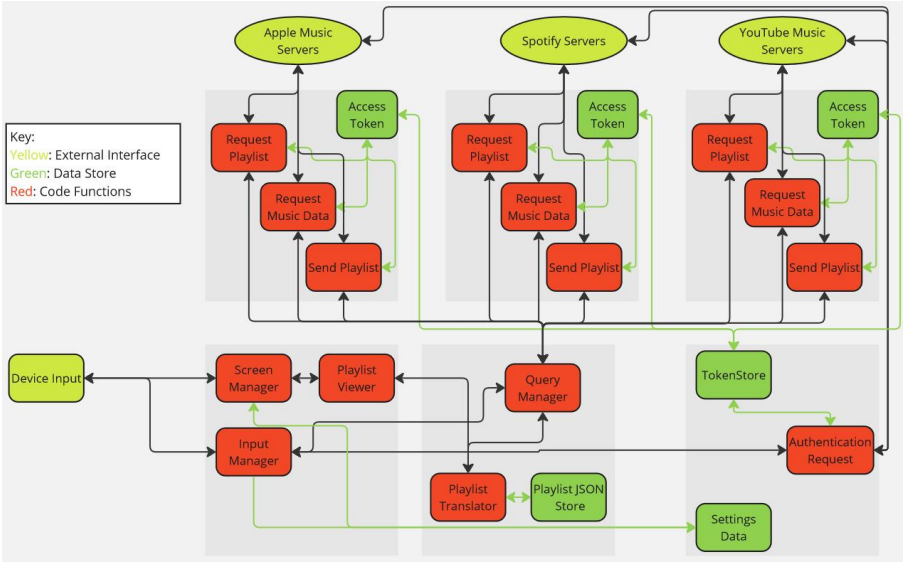
Due to the time at which this document was created, we did not feel as though we could give an accurate estimation of the algorithms and processes that would be used for the sections, which is why they are excluded here.

6.9. Interface/Exports

Component	Interface/Exports
Playlist Sync App	Classification: Application Definition: Playlist synchronization. Responsibilities: Manages components and stability. Constraints: Resource usage, stability. Composition: Flutter widgets and packages. Uses: Relies on Dart and Flutter SDK for implementation. Resources: Flutter's runtime and network resources.

	Processing: Dart async/await mechanism. Interface: UI from Flutter widgets.
Playlist Retrieval	Classification: Subsystem Definition: Retrieves playlist data from external platforms APIs using HTTP requests. Responsibilities: Handle data retrieval using Dart. Constraints: API requests can be limited. Network availability. Composition: HTTP client libraries and Dart. Uses: Dart uses HTTP client for API requests. Resources: Network Bandwidth and API accessing. Processing: Asynchronous data retrieval using Dart. Interface: Dart.
Data Storage	Classification: Subsystem Definition: Hold data with the use of a Local Data Store. Responsibilities: Use CRUD to maintain data. Constraints: Storage space and potential data corruption. Composition: Local Data Store. Uses: Stores and retrieves data within the app using Local Data Store APIs. Resources: Disk space and memory. Processing: Adding, altering, and removing data from databases in Local Data Store. Interface: Local Data Store API.

6.10. Detailed Subsystem Design



The above is a diagram of our intended modules and submodules including external and internal facing modules. Each module interaction is displayed as a black pointed line. This

diagram also includes our intended data stores as trace data interactions through the green pointed interaction lines.

7. Glossary

Flutter: An open-source framework that allows for easy integration with iOS and Android.
Concurrency/Synchronization: Management of tasks that are executed simultaneously.
Version Control: A system that can track changes in different versions of code.
Dart: The programming language that corresponds with the Flutter framework.
API: Also known as Application Programming Interface, it is a set of tools that facilitates communication between applications that otherwise would not be compatible.
Error Handling: The process of managing and responding to errors.
Subsystem: A smaller part of a larger system.
Module: A unit of software that performs a specific function. Made from functions, data structures, and other elements.
Stories: A few sentences in simple language that outline the desired outcome.

8. Bibliography

Flutter. “Flutter Documentation.” *Docs.flutter.dev*, docs.flutter.dev/.

“Music_kit Example | Flutter Package.” *Dart Packages*, pub.dev/packages/music_kit/example. Accessed 13 Feb. 2024.

“Spotify_sdk | Flutter Package.” *Dart Packages*, pub.dev/packages/spotify_sdk. Accessed 13 Feb. 2024.

“Youtube_data_api | Flutter Package.” *Dart Packages*, pub.dev/packages/youtube_data_api. Accessed 13 Feb. 2024.