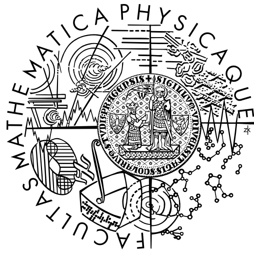


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

BACHELOR THESIS

Michal Jurčo

Remotely controlled multi-platform music player

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Martin Kruliš, Ph.D.
Study programme: Computer Science
Study branch: Software and Data Engineering

Prague 2020

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

This way I would like to thank to my supervisor, RNDr. Martin Kruliš, Ph.D., for his advice and help whenever I needed anything.
Thanks should also go to Ján Hoffman, who created a demo mobile application to help demostrate how this project works.
I'm also extremely grateful to my partner Kristýna and my parents, relatives and friends for their support throughout all my studies.

Title: Remotely controlled multi-platform music player

Author: Michal Jurčo

Department: Department of Software Engineering

Supervisor: RNDr. Martin Kruliš, Ph.D., Department of Software Engineering

Abstract: Music plays on many places as a primary or secondary tool to make people feel relaxed and in good mood. Most currently available applications focus on what is being played but do not adapt music choice based on who is currently listening to. The most precise way is to let the listeners decide which is what jukeboxes do but they have not caught up with the progress of technology yet.
The goal of this thesis was to design a music player where playback can be remotely controlled by multiple users. Application operator may create a music library from songs stored in local files (such as MP3) and organize these songs in playlists. Regular users may explore playlists and enqueue songs into playback. The application is ready for integration with mobile technologies, so regular users may control the playback via a mobile application.

Keywords: music player web

Contents

Introduction **3**
Target Group . . . 3
Legal Perspective . . . 3
Thesis Structure . . . 4

1 Analysis and General Design Proposal **5**
1.1 Use Case Scenarios And Requirements . . . 5
1.1.1 User Requirements and Functionality . . . 6
1.2 Related Work . . . 7
1.2.1 BarBox . . . 7
1.2.2 Noispot . . . 7
1.2.3 Mood Mix . . . 8
1.2.4 Streaming Services . . . 8
1.2.5 Youtube . . . 8
1.2.6 Music players supporting remote control . . . 8
1.2.7 Conclusion . . . 8
1.3 Modular Application . . . 9
1.4 Definition of Modules . . . 9
1.4.1 Reasons for Module Division . . . 11
1.5 Manager Module . . . 12
1.5.1 GUI options . . . 12
1.6 Fileserver Module . . . 13
1.6.1 API design . . . 14
1.7 Player Module . . . 15
1.7.1 API design . . . 16
1.8 Data Management Module . . . 17
1.8.1 Database requirements . . . 18
1.8.2 Platform Choice . . . 18
1.8.3 Firebase Platform . . . 20
1.9 Guest Application Module . . . 20

2 Design **22**
2.1 Target use-case and extent of features . . . 22
2.1.1 Modular Design and Ease of Use . . . 22
2.1.2 Multi-platform Application . . . 22
2.2 Programming Language . . . 23
2.2.1 Manager Module . . . 24
2.2.2 Fileserver Module . . . 24
2.2.3 Player Module . . . 25
2.2.4 Data Management Module . . . 25
2.2.5 Final Decision . . . 25
2.3 Module Design . . . 26
2.3.1 Manager Module . . . 26
2.3.2 Fileserver Module . . . 31
2.3.3 Player Module . . . 34

2.3.4 Data Management Module 39

3 Implementation **43**
3.1 Manager Module 43
3.1.1 Web Application Part 43
3.1.2 CEF Wrapper and Integration of Other Modules 46
3.2 Fileserver Module 47
3.2.1 Data Storage 47
3.2.2 REST API 48
3.3 Player Module 48
3.3.1 File Caching 49
3.3.2 Audio Decoding and Playback 49
3.3.3 WebSocket API 49

Conclusion **51**

Bibliography **52**

List of Figures **54**

A User manual **55**
A.1 Juke It 55
A.1.1 Sign in and sign up 55
A.1.2 Music spot creation 56
A.1.3 User interface 56
A.1.4 Devices 57
A.1.5 Setting up JukeIt 57
A.1.6 Settings 58
A.2 Player Module 59
A.2.1 Parameters 59
A.3 Fileserver Module 59
A.3.1 Parameters 59

B Instructions for Building the Project **61**
B.1 Prerequisities 61
B.1.1 Building Boost 61
B.2 Cloning and Setting Up Working Directory 62
B.3 Building and Using Web Application 63

C Attachments **64**

Introduction

Listening to music is one of the most popular ways to relax for many people and music playing in the background is one of the most common things that makes people feel better and more relaxed at any place, whether it is in a shopping centre, in a restaurant, in a waiting room at hospital or in your car.

While you can choose what you want to listen to in your car, it is impossible in most of the other places. The only option for people to choose what they want to listen to in a public place are jukeboxes in bars. The technology has moved in a fast pace since they were introduced, but jukeboxes remain big boxes taking up space, eating coins when most of the people pay by card or by smart phone. Many places and venues therefore offer no choice of what is being played.

The goal of this thesis is to design a music player where playback can be remotely controlled by multiple users. Application operator may create a music library from songs stored in local files (such as MP3) and organize these songs in playlists. Regular users may explore playlists and enqueue songs into the playback queue. The application is ready for integration with mobile technologies, so regular users may control the playback via a mobile application frontend.

Target Group

As with every piece of software, we would like to make our target group the widest possible to attract a lot of potential users. This is reflected in some of the decisions we had to make during design and implementation steps.

There are many places that would benefit from features that this application offers. From small private home parties where anyone can become a DJ, through waiting rooms in hospitals or hairdressers where people often spend a lot of time waiting without any other entertainment besides music, to restaurants and bars where music is an essential part of a good atmosphere and completes the whole feel of the place.

We would not like to disallow anyone from this range of potential users to use this application. The final product should offer platform independence, so everyone can choose what suits them best, modularity, so everyone can customize it according to their needs, and scalability, so no one will be held back once they need to grow.

Legal Perspective

The complexity of laws in Czech Republic might have slowed down the creation of similar applications and certainly influences what services are offered by commercially available applications.

Music is subject to copyright ownership. Getting a copy of a sound recording means acquiring a permission or license from the copyright owner. Getting or distributing such copies without permission of owner is (except certain circumstances) considered a copyright infringement. Moreover, there are certain

requirements around the public use of copyright works. Usually another licence is required for public reproduction of copyright works.

This thesis focuses on the process of designing and implementing a music application. For simplicity, we assume that users of the application have done all steps required by the law and legal directives that apply on this matter as it goes beyond the topic of the thesis.

Thesis Structure

This thesis is split into several chapters. In the introduction we briefly describe the goal of the thesis. In chapter 1 we analyze the problem and discuss the general application design. In chapter 2 we explain the design of our implementation of the application, used programming languages and libraries. In chapter 3 we talk about specifics of our implementation and interesting problems we faced.

1. Analysis and General Design Proposal

In this chapter we will define user requirements. We will analyze requirements for individual parts of our application and propose a general design for the parts that our application consists of.

1.1 Use Case Scenarios And Requirements

We would like to start with describing a few real life use case scenarios of our application. These will help us to get a better understanding of the issue and to define user requirements.

- An entrepreneur is establishing a new bar. The entrepreneur has not solved the sound system yet. A classic jukebox does not match the interior design and, additionally, the pursued clientele does not carry cash or change anymore.
- The owner of an establishment, where customers spend a lot of time waiting (e.g., hairdresser), would like to make the waiting more pleasant by playing music. The owner has no particular knowledge about the music preference of their customers and would like to allow the customers to choose the music themselves while they wait.
- Peter throws a New Year's Eve party for his friends. He has a good collection of music on his computer. The music preference of his guests is varied so he would like to allow his guests to choose their songs. However, Peter has a lot of private things on his computer and he would not like to allow his guests to use his computer without his supervision nor would he like to spend the whole evening sitting at his computer instead of spending time with his guests.

More formally, the user of the application (further referenced to as *spot admin*) would like to let other people (further referenced to as *guests*) to choose what songs are being played in the application in their *music spot*. Spot admin can define what songs are currently available to choose from and has control over music playback. Guests can browse available songs on a different device (e.g., mobile application on their phones) and add their ordered songs in a queue.

This thesis focuses on designing and developing an application for music playback used by spot admins. The design of application for guests is not part of this thesis. Such application considers slightly different user requirements and requires a different skill set outside of the expertise of the author of this thesis. A prototype of mobile app for guests is under development that integrates with our application via a common data management system available on the internet. The details of the integration of the mobile app will be explained later.

1.1.1 User Requirements and Functionality

Based on the previously mentioned scenarios we are now able to formulate user requirements that our application tries to fulfill and which influence the design. These requirements are split into functional and non-functional.

Functional requirements

- Playing music is the core functionality of the application. It is important to support decoding of multiple popular audio formats and outputting the audio to an audio device.
- A song queue is a common thing known from jukeboxes. Multiple guests may order songs simultaneously or before a previous song has finished. Ordered songs are held in a queue and played one after the other based on when they were ordered.
- Music playback should be continuous. Music should play even when no songs are ordered by guests. The application should select song order by itself in such situations to limit required user supervision.
- Another important requirement is music library. A music library stores and manages a collection of music files. A varied music library may contain thousands of music files. These music files will be offered to guests to choose from. Spot admins may also create playlists. Playlists allow to create a custom subset of music files within a music library. These can be used to offer different sets of songs on different occasions.
- User accounts and user authentication. Spot admins may provide a description of their music spot as part of their user account. Guests may discover music spots available close to them. It also lays the basis for potential payment system for ordered songs.
- Graphical user interface is necessary to allow spot admins to easily interact with the application.

Non-functional requirements

- The solution needs to be scalable to handle increasing number of guests within a music spot and to handle increasing size of music library. These two numbers should not limit the usability of the application.
- The application should be platform independent. Different use cases and users may prefer or require different platforms to run the application. Platform independence provides more potential users.
- Internationalization of the application is a necessity. Even within Czech Republic potential spot admins might speak different languages (e.g., foreign owners of establishments, exchange students, minorities etc.). The application should be designed to support localization to different languages.

- The application should offer means of customization to accommodate a wide range of use cases.
- The deployment of the application and music spot setup should be simple enough so it can be performed by users with basic computer skills.

1.2 Related Work

There are currently available several other applications and solutions that are used for music playback in public places.

1.2.1 BarBox

BarBox is advertised as next generation jukebox. It provides music database with millions of songs and businesses may restrict and specify what songs they wish to play in their establishment and offer to their guests. Guests may interact with establishment by using a mobile application that connects to local wi-fi. They may select their preferred songs and artists and BarBox then customizes the playback based on preferences of current guests. Additionally, guests may push their songs higher in order by purchasing and spending BBcoins [1].

Current state of services provided by BarBox is close to the desired functionality of our application. Their licensed music database is convenient for businesses as they do not need to purchase their own. Mobile applications allows guests to choose what they want to listen to. They support creation of custom playlists and allow remote access to admin interface. BarBox does not support playing music from own music library, guests are limited to use only local wi-fi that businesses have to set up and their services are limited to businesses as they require a monthly payment. It is important to note that the application has developed significantly in past two years. Before (when we started developing our application), the services it offered were much more limited.

1.2.2 Noispot

Noispot is a solution for businesses that uses custom hardware for music playback that is easy to setup and install and offers its own music database. Businesses may choose from playlists selected by Noispot DJs or create their own playlists. Additionally, it provides a mobile application that guests may use to interact with the music being played. They may discover what is playing at the moment or what will be playing next and they may vote for their favourite songs or artists and Noispot plays songs that are preferred by most of them [2].

This solutions does not offer a way for guests to directly play their desired songs, but they can vote for their preferred ones to influence the playback. Custom player hardware allows to put it on the most suitable location for the audio system and admin console is accessible through web application. Similarly to BarBox, businesses are limited to their music database.

1.2.3 Mood Mix

Mood Mix is music streaming solution focused on small and medium businesses. They offer playback of music selected by experts to match various types of businesses. Business owners can choose from their range of playlists to create specific mood and atmosphere on different occasions, but there is no interaction with guests. This service is paid but is fully licensed and legal. The streaming application can run on any device (computer or mobile device) [3].

1.2.4 Streaming Services

This category includes various services such as Spotify, Apple Music or Google Play Music. These services provide access to their licensed music database containing millions of songs for a monthly subscription fee. They do not offer a way for guests to influence music playback, but it would be an interesting option if it could be integrated into our application as a substitute Fileserver module. Unfortunately, their terms of use limit the usage only for private purposes.

1.2.5 Youtube

The most basic way that some businesses do is playing music from Youtube. Youtube is a platform focused on sharing video content and contains a lot of music videos. It is very popular and provides an easy access to free music which makes it attractive and appealing to users with lowest demands. It offers no options for guests to influence what is being played, contains a lot of ads (without using ad-block browser extensions) and should not be used for public playback due to copyrights.

1.2.6 Music players supporting remote control

There are several music players that support remote control from smartphones, such as Winamp [4] and Ampwifi Winamp Remote [5] or VLC [6] and VLC Mobile Remote [7]. Besides providing remote control of music playback from smartphones, these apps also support interacting with current playlist and access to media files and folders on the computer over LAN. While it may be useful on a small home party for friends and family, these apps are intended to offer users remote access to their private music player. They do not distinguish between admin and guests and give every connected user the same level of control over playback and media file access.

1.2.7 Conclusion

Current commercial solutions for businesses limit their services to using their music library, which may be convenient for users with no music library or whose music preference is contained within it. Users interested in music not contained within the music libraries (e.g., local bands, unlicensed production) have no options to allow guests to choose from this music. Furthermore, we did not come across any solution for private/personal use that would recognize guest access that only allows ordering songs from playlist. They either offer no way to browse

music library and select songs from multiple devices or they give each guest the same level of access as the admin.

1.3 Modular Application

The range of possible use-cases of our application is wide. It can be seen earlier on the real life scenarios. Each of them have different requirements. The users would therefore appreciate the option to customize the application based on their needs. Application design should be able to accommodate these needs.

One of the possible differences is the required complexity of provided services. Some users might prefer simple *Plug & Play* type of application that is simple to set up, simple to start and the music is playing. Others might require complex audio settings and/or complex playlist creation and management tools.

Additionally, our application uses wide range of hardware. Whether it is speakers and audio devices, which may require complex wiring in larger buildings, internet connection to communicate with mobile applications or data storage devices containing music libraries. Users might appreciate if they would not have to provide access to all of these resources on one device.

The idea is to split the application into multiple separable modules. A module represents a separable process. Each module has specific role and defines unique functionality and services that it should provide. Modules communicate with each other via network and together they provide all functionality that the application offers. They have predefined application programming interfaces that define how their services can be accessed and how to communicate with them. These modules may then run on one or more devices owned by user or on a server.

By adopting this approach it is possible to create multiple implementations of certain modules. Each implementation of module might offer different level or approach of required functionality and services or support different platforms and devices. Users might select the most suitable implementation for them.

Additionally, modules can be separated from each other based on specific local conditions. Each of them can run on a different machine, or all on one. The only requirement is network access for communication. That should not be a limiting factor thanks to capabilities of current wireless network technologies.

Last, but not least, not all modules would have to be run by the user. There can be modules providing services to multiple users such as a shared music library server.

1.4 Definition of Modules

Upon reviewing the user requirements and analyzing target use cases we have decided to specify five separate modules. Each module then specifies what its implementations need to do and how they communicate with other modules.

Figure 1.1 shows the relationship between modules. An arrow leads from module that provides an API to module that uses the API and describes what services are used.

1. Manager module

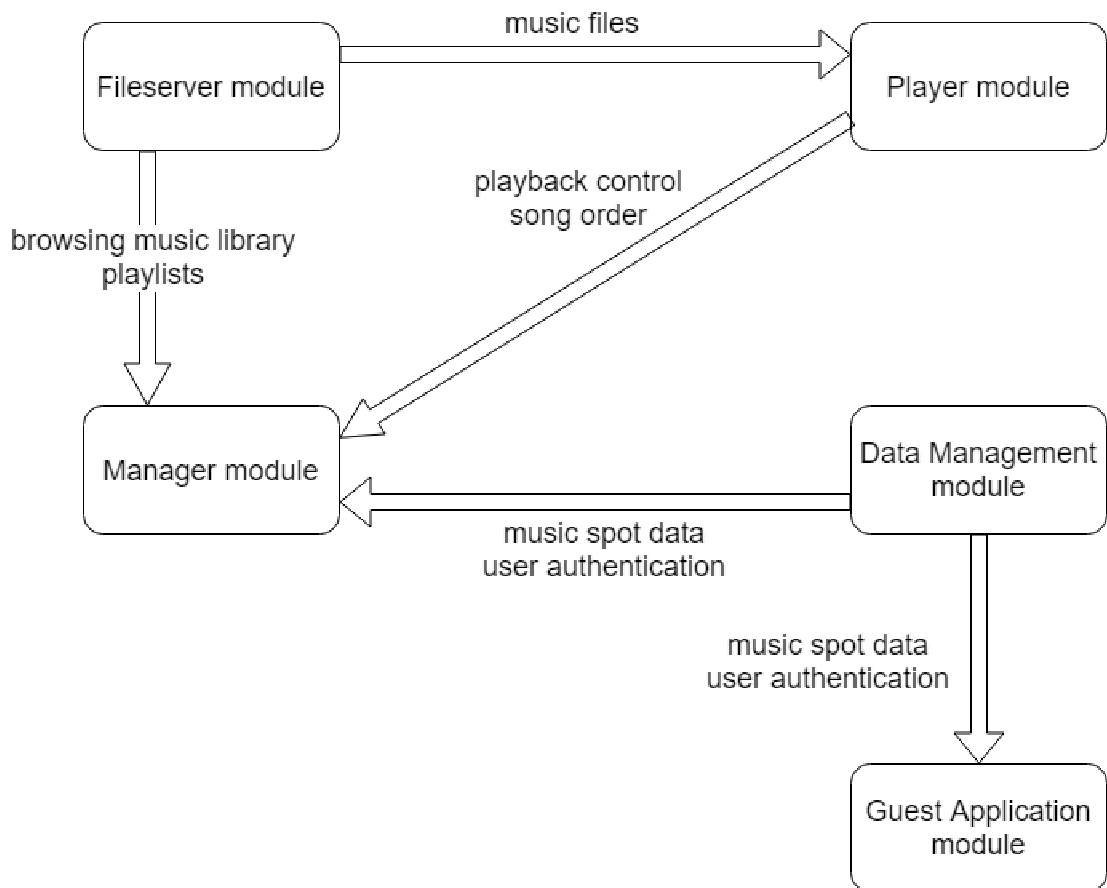


Figure 1.1: Modules and relationship between them

- Used by spot admins to operate the application
- Provides graphical user interface (GUI)
- Manages the state of the application

2. Fileserver module
   - Stores a music library and supports browsing through it and distribution of music files
   - Stores playlists (custom subsets of its music files) created by spot admins
3. Player module
   - Handles the process of playing music - decoding music files and outputting audio to audio device
4. Data Management module
   - Manages data exchange between spot admins and guests
   - Stores user accounts and provides user authentication
   - Stores music spot data (e.g., list of available songs for guests, song queue) and settings

5. Guest Application module

    - Provides front end for guests to discover music spots, browse available songs and order them

1.4.1 Reasons for Module Division

An application could be divided into modules based on different criteria. We proceeded by separating certain functionality into modules to solve problems that spot admins might face. We tried to keep the number of modules low to prevent unnecessary complexity in design and user experience.

Music library may require a considerable amount of disk storage. It might be appropriate to have this storage moved on a different device (e.g., because of its physical size, safety or cooling requirements) or in a cloud storage. Furthermore, music is protected by copyrights. Acquiring a large amount of music files might cost a lot of money. Therefore it might be useful to use a public Fileserver instead which offers its service for a monthly fee similarly to music streaming services such as Spotify. These are the two reason why Fileserver is a separate module.

Player is a separate module to allow users to remotely control playback. It is possible to put the device (e.g., a miniature PC) running this module to the most suitable location. As we mentioned earlier, in larger buildings the speakers may require complex wiring and this wiring does not have to lead to the device from which the player is controlled.

Data Management module provides global services and is separated because it is not run by spot admins. User authentication and music spot data should be accessible from anywhere on the internet so all users can login to their accounts and guests can discover nearby music spots wherever they are. Furthermore, it separates Guest Application and Manager modules. This way, Manager module does not need to provide any communication interface accessible from the internet (no safety risk) and Guest Application module does not require any setup to communicate with the right Manager module. It however adds additional requirement on Data Management module to support notifying Manager module about changes (song orders).

Manager and Guest Application module provide the remaining functionality required by their respective users which does not need to be split further.

Guest Application is a simple front end for browsing available songs within a music spot and sending song orders.

Manager module requires front end layer to process user input and back end layer to manage state of the application. Back end layer communicates with other modules to synchronize the state of the application (e.g., sending current order of songs to Player, reacting to song orders from guests). It would be possible to separate back end and front end layers into separate modules so that multiple people could control a single music spot (e.g., multiple waiters working with one application instance in a bar). That is, one application state managed by one back end module could be controlled from multiple front end modules. However, we decided not to require this feature from all implementations of Manager module. It would introduce additional security demands and complexity of state management to the general design.

1.5 Manager Module

Manager module provides graphical user interface (GUI) that allows spot admins to interact with the application. This GUI is used to control most of the application functionality. It is expected to provide these controls:

- Login and registration controls.
- Controls for selecting/connecting to Player and Fileserver modules.
- Controls for browsing through music library.
- Controls to create and manage playlists. Playlists consist of songs within a music library.
- Music player controls.
- Controls for editing music spot details.
- Controls for managing order of songs in queue.
- Application and user settings.

Spot admins need to be able to browse through music library of the connected Fileserver module to create their custom playlists and to choose what songs they would like to offer to their guests. To do that they need to see information about songs like title, author or album. Additionally, this information is further offered to guests. The location of music files and music files themselves are irrelevant in this process. They only need to be available for Player when the file is about to be played.

Before guests can access the list of available songs in a music spot, Manager module has to upload it to Data Management module so guests can access it. Since Manager module already has access to these data there is no problem with that. Manager also retrieves song orders from guests from Data Management module.

This module also allows changing playback state of Player module by sending it playback control commands issued by spot admins and providing it with an updated sequence of songs to be played. This sequence is constructed in Manager module from songs ordered by guests, songs selected by spot admin or generated by Manager module. The GUI should support modifying this sequence up to preferences of spot admin.

1.5.1 GUI options

There are multiple ways to create a GUI - classic desktop applications, web applications and mobile applications.

We would like the GUI to be modern and visually attractive. Common GUI frameworks for desktop applications tend to struggle when it comes to supporting multiple platforms as they usually either offer a limited set of features, limit customization of them or are difficult to work with. Current useful frameworks are Swing or JavaFX for Java, WPF for .NET, Qt for C/C++ and GTK that is supported in multiple languages.

In the past few years, web applications have become very popular. Increasing performance of personal computers and development in web technologies as well as rise of single-page application frameworks allowed the creation of applications running almost entirely in a web browser. Following this trend there are some desktop applications written entirely using web technologies, enclosed in a web browser wrapper such as Electron [8] or Chromium Embedded Framework (CEF) [9]. Web GUI can be written once and then only be slightly modified for different use cases such as web application running on any web browser even on mobile devices or be wrapped as desktop application. This approach could be a convenient way to provide a cross-platform GUI for Manager module.

Web applications are, however, limited in the range of network technologies they can use. Since Manager module utilizes API of all other modules, we should be careful to use technologies supported by both web and desktop applications when we create these APIs.

1.6 Fileserver Module

Fileserver module manages a collection of music files. A music file is a file that contains audio data (raw or encoded) and usually also metadata about the content such as title, author or album name. These files are grouped in a music library that is offered to other modules as a service.

In Manager module, spot admin can browse through this music library to create playlists and select songs for playback. For humans, the metadata of music files are more important and comprehensible during this process than music files themselves or file structure. Therefore, the API should support browsing through metadata rather than music files. We support four most common metadata:

- Song title
- Artist name
- Album name
- Genre name

The structure of data in music library should reflect the logical structure among songs - they have an author, belong to an album or genre, optionally a playlist. This module therefore extracts metadata from music files and sorts the songs based on that. These data are then offered as music library. We consider that to be a more natural and intuitive structure for users.

The music library can range from a small private one containing tens or hundreds of songs to a large shared one containing millions of songs. The file structure can vary for each use case or platform. Moreover, direct file system access is most likely unwanted in case of a public Fileserver. That is why file management is excluded from the API that this module offers. Instead, Fileserver implementations are free to choose their way of managing the contents of the library.

Player module still requires to access music files to be able to play music. However, it does not require any additional information apart from the music file itself. It is given a list songs to be played from Manager module. These can be

song identifiers which Player module then uses to retrieve associated music files for playback.

As Fileserver, the source of music files, can run remotely from Player module, the delay in music file transfer can reach several seconds. There are two ways to solve music file transfers. They can be streamed continuously or downloaded fully before they are played.

Streaming has low time overhead and low space requirements as small bits of music files are sent on demand for playback. However, it relies heavily on stable network to provide continuous playback.

On the other hand, downloading full songs takes non-negligible amount of time (but this can be done in advance) and downloaded songs have to be stored before they are played. This method guarantees continuous playback of a song once it begins even with unstable network connection.

The application is expected to provide smooth and seamless music playback as music is intended to play for many people. Therefore we have chosen to support downloading full songs as we find it to be a more suitable solution.

1.6.1 API design

Fileserver works as a server in a client-server model where it responds to client requests for music files by Player module and music file metadata by Manager module.

Music file metadata should be extracted once when music file is added to library and stored in a database or data structure as sometimes a whole file needs to be inspected. Keeping metadata away from files allows the owner of Fileserver to keep files structured up to their liking while providing required logical metadata structure. The API should also support ordering and filtering on server side so the entire music library does not need to be loaded to perform these tasks on client. The communication is stateless as no information needs to be retained between requests.

FTP protocol would work well for music file transfers and it would be possible to allow browsing through metadata, e.g. by storing them in JSON file. This solution however does not allow a way to filter and order metadata on the server and it would be harder to scale it up with more songs in library. Moreover, support for this protocol in web browsers is limited with plans to be deprecated soon, which would be a problem for web application implementations of Manager module. This problem is similar for other protocols for file transfer and file sharing.

HTTP protocol is a suitable protocol for transferring both music files in binary format and metadata in JSON. A RESTful API and SOAP are similar approaches to get resources over the internet using this protocol. SOAP is more robust than REST but this robustness is not required as it introduces more strict interface. RESTful API use URI to identify its resources and HTTP methods to specify requested operations on them. URI can contain query parameters to filter and order requested metadata so client (spot admin) can transfer only data they need. RESTful API design defines additional constraints, especially HATEOAS[1], which

[1]A REST service is not an endpoint but a web of interconnected resources, with an underlying hypermedia model that determines not only the relationships among resources but also the

we do not need to include. Such API requires backend to keep track of where the music files are located and store metadata in a convenient structure to allow filtering and sorting quickly.

A simple RESTful API is sufficient to provide required level of services. It contains 6 different resources:

- Songs
- Artists
- Albums
- Genres
- Playlists
- Music files

Each resource is defined by the endpoint (URI) that it can be accessed by and query parameters are used to modify the result. HTTP request method determines the action performed when accessing a resource. GET method is used to retrieve resources. POST, PUT and DELETE methods are used to create, modify or delete playlist resource as all others are immutable. JSON file format is used for transferring data and music files are transferred in binary format.

1.7 Player Module

Player module is responsible for decoding and playing music files. All music files are encoded using a specific codec[2]. Player module provides tools for decoding these music files utilising multiple popular audio codecs to produce data that can be sent to connected audio device to reproduce the audio.

Since music files can be located on a remote Fileserver it is important to support file caching, so that next file is always ready to play without a pause for buffering after the previous one finished. To achieve that, Player module needs to know which songs to cache in advance. It has an internal queue of files to be played. Instead of telling it what song to play after the current one, Manager module is required to provide a list of songs to be played next in advance so the files can be cached. It is important to note that this module can not decide what songs are played as it has no knowledge of currently available songs in music spot.

During playback of a song there are several events that Manager module needs to know about, such as beginning of playback of a new song, end of playback of

possible net of resource state transitions. REST clients discover and decide which links/controls to follow/execute at runtime. This constraint is known as HATEOAS (Hypermedia As The Engine Of Application State) [10].

[2]A typical digital audio coder, or codec for encoder-decoder, is a device that takes analogue audio signals as input and transforms them temporarily into a convenient digital representation. This transformation process takes place in the encoder stage of the coder. Once we have the signal represented as a series of numbers then we can store it, process it, or transmit it. At some point, we would like to be able to listen again to the sound. To do so we need to transform the signal from its digital representation back to an analogue signal so that the human ear can detect and enjoy it. This inverse transformation from digital back to analogue takes place in the decoder stage of the coder [11].

a song or error during caching or decoding of a song. Because the playback is performed on Player module, it needs to be able to notify Manager module when these events occur. It means that the communication between Player and Manager modules either has to be bi-directional or Manager would have to check for these events in periodic intervals. We prefer the first option, because it allows reacting to these events when they occur, not when Manager finds out about them and lowers the amount of exchanged messages between both modules to the necessary amount.

1.7.1 API design

API requirements for Player module are different from Fileserver module. Fileserver is stateless and should be able to support multiple users. On the other hand, Player needs to preserve its state, e.g. playback progress or cache, it can only support one session at a time as there is usually only one audio device available. Additionally, it needs to be able to notify Manager module about changes of its state without a request (e.g., errors, playback state, not enough songs to cache), so communication is bi-directional.

In direction from Manager to Player the API should support calling playback control commands. These commands have to follow request-response pattern. Manager module needs to know the outcome of the commands it sent, because they usually result in a change of playback state when they succeed. We should consider using any existing Remote Procedure Call (RPC) protocol[3] as we simply want to call procedures on a remote device.

In direction from Player to Manager we send notifications when an event occurs during playback. These notifications do not need confirmation that they were processed. They only inform that a change of playback state occurred. We just have to ensure that these notifications are delivered and they are delivered in correct order. This suggest that we can use a TCP connection or a protocol built on top of it.

Since we want to support Manager module implementations using web technologies, we are more or less restricted to use HTTP or WebSocket protocols supported by browsers. If we used HTTP protocol, we could set up a REST or SOAP API in direction from Manager to Player, similar to Fileserver module. Sending requests in such API would run a command on the Player and response would contain its outcome. The opposite direction however would not be feasible because it would either require a wrapper around Manager module that supports receiving HTTP requests or we would have to cancel this direction and make Manager module poll for notifications.

Using WebSocket protocol we are able to send messages both ways once the connection is established. WebSocket is a common protocol used by web applications that require a full-duplex communication and has implementations available in many programming languages including C++, C# or Java, so it does not limit

[3]Using this technique, we can call Player function stubs as if it was a local component. These stubs then pack function name with parameters into a message and send it to a remote Player instead of calling the actual function locally. Upon receiving the message, the Player unpacks it and calls the desired function with provided parameters. On completion, the response is sent similarly back to sender.

Player module implementations. A proper message format needs to be defined to easily distinguish between messages sent in either direction.

Looking at existing RPC protocols, we decided to use JSON-RPC, which is a light-weight RPC protocol utilising JSON format to serialize messages exchanged between server and client. JSON-RPC is transport agnostic so it works well over WebSocket. It defines several data structures, format of messages and the rules around their processing. Its simplicity was the reason we chose it as it can be used in any programming language or platform. The procedure calls for playback control are very simple and easily defined in this protocol. Additionally, it supports notification messages which allows us to also use this message format for notifications sent form Player to Manager.

1.8 Data Management Module

Data Management module is an important module that connects spot admins using our application with guests using mobile application. It provides services used by both groups of users as well as handles interaction between them.

This module provides user authentication and stores user accounts with their preferences and settings. One user account database is sufficient for spot admins and guests as one account can be used to host a music spot or as a guest.

Each user account can own a single music spot. A music spot is a place where music is played from our application. Each music spot can add its name and description. Before music spot is activated, spot admins upload from Manager module a list of songs (music file metadata) that are currently available. If spot uses private Fileserver, it has to provide full music metadata. In case of public Fileserver we would already have all metadata, so just a list of song identifiers or playlist identifier would be enough. Guests may browse active music spots and when they enter them, they can browse available songs and order them. Ordered songs are then added to queue.

This module therefore also stores music spot data - its basic information such as name or description, list of available songs and song queue.

The requirements for this module lead us to creating only one global implementation of this module as opposed to other modules. Those can be run by user as they provide user-specific functions. This module provides services for all users of the application (both spot admins and guests) and therefore can not be run by one of them.

Alternatively we can only support user authentication and basic music spot information in this module and handle the interaction between guests and spot admins (providing list of available songs and ordering songs) in a new separate module or add this functionality to Manager module. That would, however, make it more difficult to set up a music spot as spot admin would have to provide a way for guests to connect to the new module (either restricting to local wi-fi network or setting up a server).

We chose the first option. It simplifies the deployment of music spot and guests can browse available songs and make orders using any internet connection (e.g., wi-fi, cellular data etc.). The problem might be the amount of song metadata that spots upload. We expect private music libraries to contain up to several thousands of songs on average. The uploaded list of available songs would then

have up to a couple of megabytes. One instance should be able to support couple of thousands of music spots at once.

There are two disadvantages of this solution:

- Anyone in the world may order certain songs to cause discomfort to people present in the music spot. To deal with that, application for guests can provide countermeasures to ensure that songs can be ordered only when guests are present in music spot. There are some possibilities such as passphrase, scanning a QR code available in the music spot or determining by location. Spot admins may then select and combine supported countermeasures to find balance between comfort for guests and required level of security. In addition to that, spot admins may cancel any song orders or ban users from entering their music spot.
- There are running costs for operating this module. Those can be covered by donations, fee for running a music spot or share of the revenue from sold songs in a music spot. This depends on the business plan if the application was to be released.

1.8.1 Database requirements

The core of the module is a database that stores the data. Spot admins need to be able to read and modify their user settings and spot information, upload list of available songs and be notified on song queue changes.

Guests need to be able to read and modify their user settings, read spot data (information and list of available songs) and add songs to queue.

List of available songs is uploaded when the spot is activated and removed when spot deactivates. This minimizes the amount of storage as only data of active spots need to be stored. When the spot is inactive, guests can not browse or order songs so there is no point in keeping them stored. Furthermore, available songs may differ entirely between two uploads for the same spot which would result in deleting old and inserting new list anyway. It is also unknown when/if the spot activates again.

The requirements on database are minimal as the only concurrent modification of the same data occurs in the queue. Other data are modified just by a single user that *owns* them. All other users just read the data. Our application also does not need to run any complex queries. Guests require querying music spots (finding spot by name or location) and browsing through available songs in a music spot.

1.8.2 Platform Choice

When choosing platform for Data Management module, we need to consider several criteria:

- User authentication and authorization and security of user information
- Database - data structure, performance, scalability, how difficult it is to set it up, configure and maintain
- Communication interface for Manager module, how to notify about song orders

- Communication interface for Guest Application
- What we can use out-of-the-box and what we need to implement ourselves

For user authentication we would like to use an already existing solution that implements all necessary up-to-date security rules and standards, such as password hashing and salting, encryption of sensitive data and protection from different types of attacks. If we implemented it ourselves, we would need to hire a security expert to guarantee that it is secure, which would be expensive and time-consuming.

Our data are not too complex, so besides traditional relational databases we could also use a suitable NoSQL database. The advantages of relational databases are ACID properties and strong query language. The advantages of NoSQL databases are schema flexibility and data structure closely mapped to data structures of popular programming languages [12]. The examples of open-source relational databases are PostgreSQL and MySql, for NoSQL databases there are MongoDB, Cassandra and others.

Recently, there is a third option called Backend-as-a-service (BaaS) that is popular with mobile application developers. Backend-as-a-Service (BaaS) is a cloud service model in which developers outsource all the behind-the-scenes aspects of a web or mobile application so that they only have to write and maintain the frontend. BaaS vendors provide pre-written software for activities that take place on servers, such as user authentication, database management, remote updating, and push notifications (for mobile apps), as well as cloud storage and hosting [13]. BaaS solution usually use NoSQL databases for data storage.

Using standard relational or NoSQL database, we would need to implement API to access database, integrate it with a user authentication solution, figure out how to notify Manager module about song queue changes and maintain the server that it would run on (whether it would be self-hosted or cloud-hosted). BaaS solutions provide most or all of these features out-of-the-box. Current popular BaaS solutions are Firebase, Parse, Backendless, AWS Amplify and others.

In cooperation with developers of Guest Application we agreed that BaaS solution would suit our needs the most as it saves a lot of effort. These solutions are rather new and in recent years experienced rapid development. We were choosing the platform in 2016 when possibilities were different. If we were choosing it now, we would probably choose differently.

We chose Firebase Platform [14] to host Data Management module. This choice was supported and preferred by Guest Application developers who already had experience with it. Essentially, we were choosing between Firebase, Parse and Backendless. AWS Amplify was not launched yet. All three of them offer required features, so any choice would be fine. At the time, Parse was experiencing changes as it was being transferred from proprietary technology to open-source and the result was unclear, so we ruled it out. Although Firebase might be less flexible and more expensive than Backendless, it is fast and easy to setup and offers great analytics, which are appreciated by Guest Application developers.

We would like to note that if we were choosing platform now, we would choose Parse. Transferring to it however would not be easy as a lot of work from both our and Guest Application developer sides was already done using Firebase, so we are using Firebase for now. Parse is now fully open-source, which is advan-

tageous to vendor-locked Firebase in case we wanted to make any modifications. Furthermore, it can be both self-hosted or cloud-hosted as it can be deployed to any infrastructure that can run Node.js.

1.8.3 Firebase Platform

Firebase provides Realtime NoSQL JSON-based tree-like database. It supports subscribing to notifications for changes where clients may listen to updates on any node within the database. On update, client is notified and callback function is called with modified data. This feature is important to listen to song queue changes in a music spot.

The interaction between spot admins and guests is performed by modifying data in realtime database. Spot admins modify their music spot information and list of available songs that guests can see and browse. Guests add song orders to song queue. To deal with concurrent writes, records in queue are identified by unique identifier generated by database. These identifiers can be ordered in lexicographical order based on time when they were created. Manager module used by spot admins listens to changes of queue to play ordered songs in the music spot. Realtime database also supports transactions in case multiple nodes have to be modified at once or if a node needs to be modified concurrently.

Firebase Platform is hosted in its own cloud with pricing for using its services. It starts for free for testing purposes and offers multiple pricing plans according to expected usage. Additionally, Firebase provides SDKs for both main mobile platforms Android and iOS along with JavaScript and C++ support and REST API. It also provides crash reporting and analytics that are important for mobile application developers.

The realtime database allows defining rules that determine who has read and write access to the database, how the data is structured, and what indexes exist. Indexes can be defined for nodes that need to be queried by value, for example when guests look for a song by its name instead of its identifier.

Authentication service provides secure way to authenticate users. It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook, Twitter and more [14].

1.9 Guest Application Module

This module provides an application used by guests. It allows them to interact with music spots.

As we mentioned earlier, application for guests is not part of the thesis, but a mobile application is being developed. A mobile application was chosen since smartphone is a common device owned by many people nowadays and mobile applications are a familiar way to interact with the surrounding digital world. However, other implementations are feasible as long as they are able to consume services provided by Data Management module.

The application allows guests to log into their accounts. Then they can discover available music spots (by name or by location). They may enter a music spot, which may require verification selected by spot admin. This is not yet implemented but reviewed methods are by location (being in a selected radius around

music spot), entering a passphrase chosen by spot admin or scanning a QR code with the passphrase available at the music spot. When guests successfully enter a music spot, they get access to list of available songs which they may browse and from this list they may order songs. Song orders may require payment so the application should support that. Additionally, guests can see what songs have already been ordered and what song is currently playing in music spot.

2. Design

In the first chapter we proposed a general design that allows multiple implementations for each module so that we can accommodate various use-cases. In this chapter we will explain what use-cases our implementation is intended for and specify the extent of implemented features. We will explain our programming language choice and take a look at libraries and frameworks we use and their alternatives.

2.1 Target use-case and extent of features

The general design described in the first chapter only specifies the topology of modules, how they communicate with each other and what they do. It says nothing about the extent of individual features that they are supposed to implement. It is clear that creating an implementation for all meaningful use-cases is beyond the scope of this thesis, not only by the sheer amount of work required, but also other resources such as marketing research, infrastructure and funding.

Our goal was to create a minimum viable product (MVP), a product that contains essential features to satisfy early users and provide feedback for further development. The MVP should satisfy simpler use-cases, such as private home parties and eventually small and medium businesses (bars or restaurants). The general design then allows us to further improve existing module implementations or provide new implementations to eventually cover all meaningful use-cases.

2.1.1 Modular Design and Ease of Use

Being able to use modules separately is definitely one of the essential features that the MVP must provide. However, running three separate programs is not the most pleasant user experience. In a lot of cases early users will not need to separate Player and Fileserver modules and instead would prefer if they were integrated in Manager module. The MVP should be able to integrate or mimic integration of these modules so that users do not need to perform any configuration when all three modules run on one device. In other cases it should be simple enough to configure, run and connect modules when they are separated.

When speaking about module integration, it is important to mention that for the purposes of MVP we focused on private Fileserver module implementation, which is operated, managed and music files are supplied by users. An implementation of public Fileserver with licensed music library would require infrastructure and funding beyond the scope of this thesis.

2.1.2 Multi-platform Application

One of user requirements is to support multiple platforms. There are three main platforms, Windows by Microsoft leading the market share with a big margin to OS X by Apple then followed by Linux (see Fig. 2.1).

For simplicity we could assume that vast majority of users use Windows and MVP could target this single platform and support other platforms by further

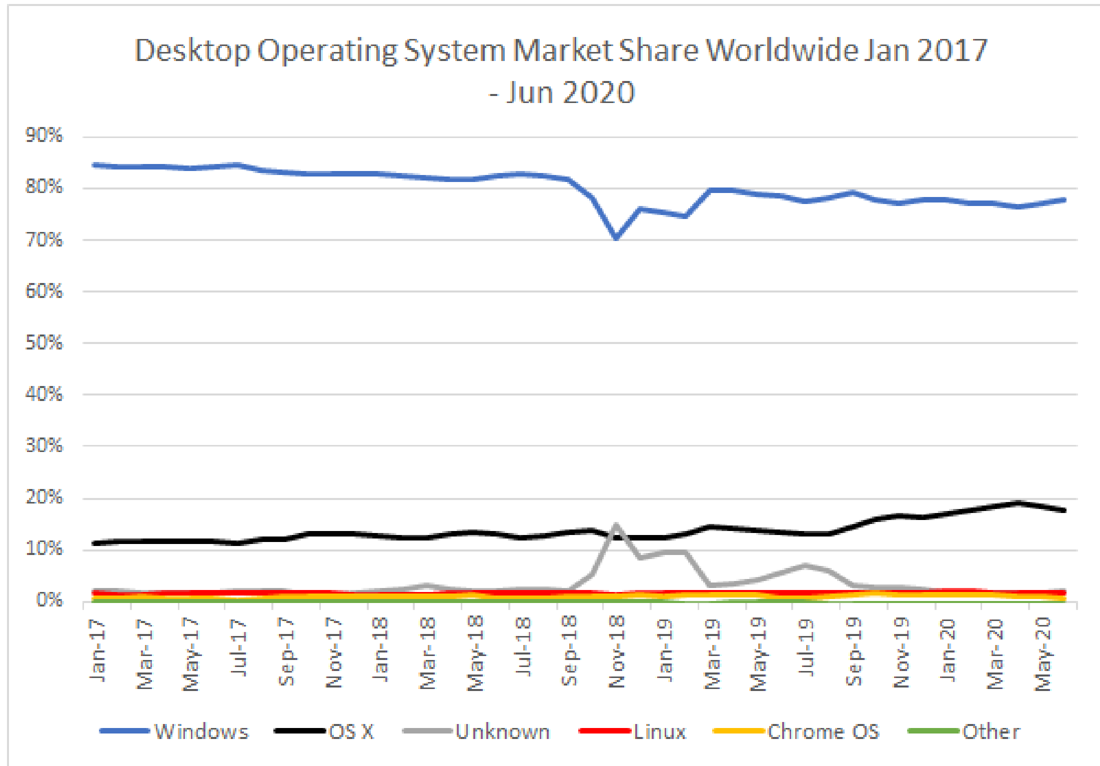


Figure 2.1: Desktop Operating System Market Share Worldwide from January 2017 - June 2020 [15]

implementations. But we do not know the exact market share in our target group and focusing on a single platform may be a major mistake.

Another reason to support multiple platforms is the recent progress of miniature PCs. The modular design allows to run each module on a different device and these computers have enough computing power to run Fileserver or Player module that can be separated. They require just a small initial investment into hardware. These PCs usually run or at least support some versions of Linux, which has the smallest market share among major operating systems.

The downside compared to creating platform-specific implementation is some extra amount of work that needs to be done both during designing and programming. However, we consider the benefits of adopting this approach now to be significant enough compared to having to create implementations for each platform from scratch in the future. While the MVP itself can be released just for one platform at first, the other platforms can be supported soon after as only a small amount of work is required compared to creating a whole new implementation.

2.2 Programming Language

The first big decision we had to take was to select suitable programming languages. We had to follow the main guidelines that we have set earlier. That is, we had to find a way to support multiple platforms, while at the same time we needed language flexible enough to handle our modular design.

Even though our modules are separable and therefore each of them could be written in a different language, we wanted to reduce the amount of used languages to minimum, ideally just one or two for the sake of simplicity. One programming language would allow us to integrate Fileserver and Player modules into Manager modules directly. Optionally, we could satisfy the condition for integration by choosing languages that can interoperate, e.g., by providing bindings. Furthermore, we focused on languages that would allow us to write one implementation of a module for all platforms with the minimum of tweaks. Last but not least, we wanted a language that we had at least some experience with. These were C++, C#, Java and JavaScript.

2.2.1 Manager Module

The main part that influenced language choice for Manager module is GUI. The back end part which communicates with other modules using HTTP and WebSocket APIs can be implemented in many languages. The more problematic part is GUI. We would like the GUI to be modern and visually attractive. Common cross-platform GUI frameworks usually either offer a limited set of features, limit customization of them or are difficult to work with. Frameworks for C# (WinForms, GTK#) and Java (Swing, JavaFX) are good for creating native-looking apps with various forms, but we would like to have more control over graphic design. C++ frameworks have similar issues, the exception might be Qt [16], which is a very capable GUI toolkit written in C/C++, plus there is a Java wrapper for this library available.

Another important aspect is that we had already had an earlier implementation written using web technologies, mostly JavaScript, and it would be very convenient for us to reuse this code. The advantage of a web-based GUI is that it looks the same on all platforms, requires only one codebase and offers good options for customization and styling. The downside is that it requires a web browser to run. While Manager module running in a web browser is a viable and meaningful option, for the purposes of MVP it would be difficult to integrate Player and Fileserver modules unless they were also written for web browser.

Alternative approach to dependency on web browser is to use web integratable desktop application frameworks. These frameworks allow web applications to make system calls, so a web application can work similarly to native desktop applications. The examples of such frameworks are Electron [8] and Chromium Embedded Framework [9] (CEF). Both rely on Chromium rendering engine, Electron combines it with Node.JS runtime, so only JavaScript, HTML and CSS are required to build a desktop application. CEF uses IPC[1] to communicate between main application written in native code and web application. It has many language bindings including C, C++, C#, Go, Java, and Python. These frameworks would allow us to integrate a web-based Manager module with other modules.

2.2.2 Fileserver Module

In Fileserver module we had to implement music file management, which includes accessing music files and reading and indexing their metadata for quick browsing.

[1]Inter-process communication

It also needs to implement a HTTP server with REST API. The most popular current languages for writing REST API server code are JavaScript and Python that make it quite simple to launch such server. Other languages also support creating these servers, albeit not as straightforward. This REST API, although it needs to be fast, does not have to handle too many concurrent users and requests. It serves one or a few spot admins, so instead of choosing language to create a high performance server, we focused on choosing a language suitable for music file management and preferred to implement server in that language.

The complexity of music file management features required by MVP is pretty low, there are libraries for reading metadata in all common programming languages and indexing them is a matter of selecting right data structures. However we also have to consider further improvements, such as editing metadata or possibly encrypting files that Player module downloads[2] that we think can be difficult to achieve using JavaScript.

2.2.3 Player Module

A Player module implementation can solve playing music in two ways - using a ready-to-use music player library or handle decoding and outputting the audio itself. The first approach would be easier, but we would not have enough control over playback process during future improvements such as custom equalizers or observing custom events during music playback. It might result in the need to implement a custom music player anyway. On the other hand, implementing own music player increases workload at the beginning, but allows us to better integrate music playback and caching of files downloaded from Fileserver module and to have full control over the playback process to make implementing new features easier. With regards to these thoughts, we regard C++ to be advantageous to other languages as a lot of libraries that work with audio are written directly in C/C++ with bindings to other languages.

WebSocket has implementations in all common programming languages, so we were not concerned about API implementation when we were choosing programming language.

2.2.4 Data Management Module

Data Management module uses Firebase framework, so we do not need to choose its programming language. Manager module uses its SDK to communicate with it and manipulate its database and its SDKs are available for JavaScript, C++ and via REST API, so it also does not demand the use of certain language.

2.2.5 Final Decision

With regards to arguments for each module we have decided to use C++ as programming language for Fileserver and Player modules and C++ with CEF frame-

[2] A separate Fileserver module will have REST API accessible from the local network, if an attacker gets access to this API, they may steal all music files. Encrypting transferred files ensures that they can be only used to be played by Player module that implements decryption. Similar approach is used in Spotify to store offline files on user devices.

work for Manager module wrapper with web-based GUI (JavaScript, HTML, CSS).

We found reusing an earlier implementation of Manager module to be a good starting point. Manager module general design was created with a web-based solution in mind. We preferred CEF to Electron for browser wrapper framework to avoid possible limitations in further development due to being restricted to using only Node.JS. Using Electron would be easier, but Node.JS libraries often use C++ library wrappers and it is necessary for some functionality to be written in C/C++ anyway, so we decided to use it right away.

When considering different implementations of CEF, we chose C++ as its main project is written using C/C++. C# and Java implementations just provide a binding to the main project. C# multi-platform projects use Mono or .NET Core, both of which a C# implementation of CEF, CefSharp struggles with. We preferred C++ to Java because of its dependency on its virtual machine and personal preferences.

For Player and Fileserver modules, we considered C++ to be the best choice in the long run, even though it might be more difficult to work with at the beginning. It also matches the programming language of CEF implementation, so we can easily integrate these modules.

2.3 Module Design

Having chosen programming languages, next we need to design our modules and select libraries and frameworks.

2.3.1 Manager Module

The module itself consists of two separate parts - web application and browser wrapper. Web application implements most of the features while wrapper handles rendering and integration with Player and Fileserver modules. As both of them use different programming languages and technologies, different considerations were needed during design and so we will describe them separately.

Web Application

A common way to design a web application for CEF is a single-page web application. The reason to build it that way is simple - usual web pages are served from the server and running a server for this purpose would make little sense. Single-page applications (SPA) are loaded once at the beginning (HTML, JavaScript and CSS code) and instead of loading new pages from server they dynamically rebuild the current page. These applications have become popular recently and provide a solution to some problems web applications have compared to traditional web page design. Namely, stateless nature of web pages and wait time due to loading. On the other hand, SPA brought their own problems but most of them can be solved by some useful JavaScript libraries, packages and frameworks. One of the main negatives - speed of initial load when all required code is loaded at once in the beginning is irrelevant for us. The whole application is stored locally on the device.

Framework

There are a few frameworks for building single-page applications. They help handle page re-rendering so developers can focus on designing the content of the application. They all provide their way of rebuilding the page and data management. Most of them are labeled as MVC[3] frameworks. It is important to note that we chose our framework early in 2017 and since then a lot has changed. We were effectively choosing between Angular.JS and React.JS. The rest of the frameworks did not have large enough community or support at that time so we did not take them into consideration.

To be precise, React is not labeled as MVC framework but as a JavaScript library. Its role is to make creating interactive user interfaces easier, it provides only the View part. It has to be coupled with other libraries to supply Model and Controller roles to reach the capabilities of MVC framework. In such form, we think both Angular and React can be used to do the same work, it is just a matter of personal choice and preference.

It is difficult to write a comparison between React and Angular as they release new features on frequent and regular basis. Many up-to-date articles comparing them and also other new frameworks such as Vue.js can be found online. Here is a list of arguments that we took into consideration:

- Angular is a complete framework that provides more features out of the box compared to React, which has to supply them using various community-provided packages, making React more flexible but harder to maintain up-to-date
- Angular more strictly enforces the way the applications should be build which makes it more difficult to learn it initially
- React is considered to be faster based on various benchmarks due to its refined *Virtual DOM*[4] rendering mechanism
- We consider React JSX syntax allowing HTML markup and JavaScript code to be merged in a single file to be a good tool for increasing productivity
- React has wider community of developers and supporters

We chose to use React as our framework for web application. We find the syntax and concept more natural and simple to use as we had no previous experience with either of them. The performance advantage is also a nice-to-have feature as Manager module needs to handle more tasks than just rendering the GUI.

[3] Model-View-Controller

[4] React stores a virtual representation of UI in memory and synchronizes it with real DOM. Rendering process is first done on virtual representation, which is faster compared to DOM changes and browser rendering. After that the minimal required changes are applied to synchronise both representations.

React Introduction

React is a JavaScript library for building user interfaces. It lets us compose complex UIs from small and isolated pieces of code called *components*. React has a few different kinds of components, we can create our own by subclassing them.

Components tell React what we want to see on the screen. When our data changes, React will efficiently update and re-render our components. A component takes in parameters, called *props* (short for properties) that can be passed to modify its behavior and appearance, and returns description of what to render. React takes the description and displays the result. Most React developers use a special HTML-like syntax called *JSX* (see Fig. 2.2 for an example) which makes these structures easier to write.

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}

// Example usage: <ShoppingList name="Mark" />
```

Figure 2.2: JSX syntax example [17]

React components can store their state. React has one-way data binding, meaning that when state changes, React renders the change in component. However, when a component is changed, the state does not automatically change. When the two-way binding is required, a callback can be used to project this change onto state.

Redux

Preserving state in React components is sufficient in small projects, but for bigger projects with deep nested component structure maintaining the state is an issue. State changes can be propagated only downwards to nested components through props, so the global state would have to be stored in topmost component, which can become hard to manage.

There are several libraries that solve this issue of which we chose Redux [18]. Redux is a state container for JavaScript apps. Instead of storing the state in the

topmost component, Redux provides a single source of truth for the state that components can read from or send updates to. Redux is not the most lightweight library for that case. It requires to follow specific constraints in the application: store application's state as plain data, describe changes as plain objects, and handle those changes with pure functions that apply updates immutably. These constraints require effort to implement, but reward us with useful features, such as store persistence and predictability (useful for testing/debugging).

The reason that we need to use such library is that besides providing just a UI our Manager module also manages the state of the whole application. It needs to persist this state in advance to the state of the UI, so it becomes rather complex.

In short, Redux preserves *state* in one immutable object. State can be changed only by dispatching *actions*. An action is a plain JavaScript object that describes the change of state. These actions are processed by *reducers*. Reducer is a pure function that, given previous state object and an action, returns new immutable state object after applying the change described by action.

Design Principles

JavaScript is more forgiving compared to strong-typed compiled object-oriented programming languages so good code structure is important to maintain readability of code. Application design therefore follows several recommended design principles for React and Redux. The code becomes structured and is split into several types of code files. Every type contains specific functionality and extracts it from the others so similar functionality is grouped together. Below is the list of code file types and their description.

- **Components** are React components that contain layout of the page – through props, they display data supplied to them and link supplied actions to layout elements. They are usually stateless. We used two types of components, layout components that represent a whole page and visual components that represent a reusable functional element within a page
- **Containers** are React components that contain functionality and business logic – functions, event callbacks, state management and data preparation methods passed to *Components*. They encapsulate *Components*, therefore they are called *Containers*.
- Redux **store** is used to preserve shared state of the application.
- **Reducers** react to actions and change the data stored in the store.
- **Actions** trigger changes on the store. For every reducer there is a set of actions and action creators. Every action creator may create zero, one or more actions depending on the conditions.

The flow of React + Redux application can be seen in figure 2.3.

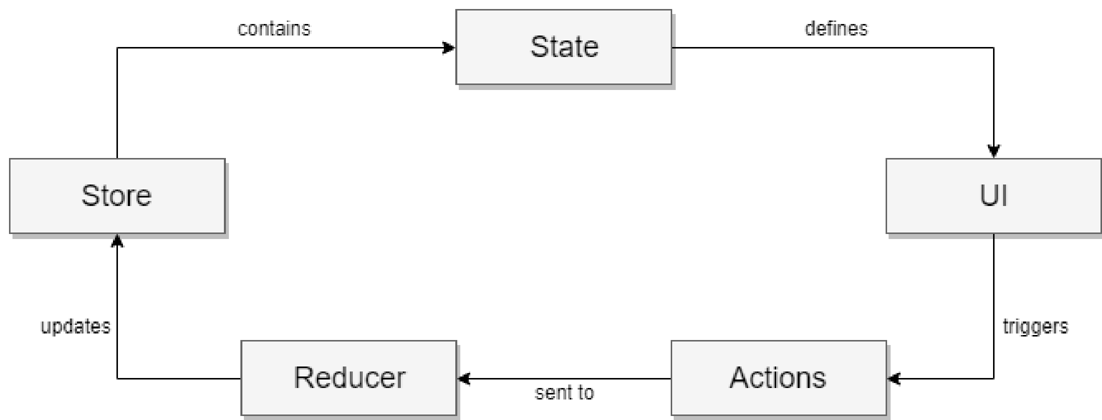


Figure 2.3: React + Redux flow

Reducers

Reducers are the only place where store state can be changed. They process *state change objects* and return a new state object. State objects are modified always in an immutable manner. When a property in state object has to be changed, instead of modifying that property we replace it with a new instance with modified values. Immutability of state object allows us to better debug and test an application as well as maintain a predictable behavior. For example, we can observe state objects after changes and compare them to see how they changed. For better code readability, a store consists of multiple state objects and every state object is handled by its reducer.

Actions

When an event that changes state occurs in application, an action is dispatched. Basic Redux actions are *state change objects* that are passed to reducers. In our application these objects contain two properties: `type` property contains a constant string that identifies the action and `payload` property that contains action parameters. Redux provides a dispatch function that can be injected to components to send these actions to Redux store.

To extend Redux functionality it supports adding middlewares. Middleware can intercept anything we dispatch, and in turn, can pass actions to the next middleware in the chain. When the last middleware in the chain dispatches an action, it has to be a plain *state change object*. We are using redux-thunk middleware to also allow dispatching *thunks*. A thunk is a function that wraps an expression to delay its evaluation. These thunks can read the store and dispatch other actions or thunks to create more complex actions and express asynchronous actions in a convenient way.

CEF Wrapper and Integration of Other Modules

CEF wraps our web application in an executable and provides a browser that renders it. It allowed us to integrate Player and Fileserver module so we can also ship a compact application for users. Web application always uses defined API to

communicate with the modules, however through IPX[5] that CEF offers we were able to add controls to run integrated modules into the GUI. Separate module implementations only offer a command line user interface.

2.3.2 Fileserver Module

The MVP implementation of Fileserver module allows creating private music databases on computers and devices of spot admins. It consists of two main functional parts - data storage for music file metadata and REST API.

REST API

First we took a look at REST API, its requirements to design its resources in a way that cooperates well with other modules. After that we can formulate data storage requirements to create a responsive service.

We tried to design an API structure that could not only be used for the purposes of MVP but also could be reused in the future with minimum modifications for a public Fileserver containing unlimited number of songs, so Manager module can use them using same code. We assumed that the music library may be too big to be loaded by the client at once. Therefore it must support queries for just parts of its content (pagination) as well as implement filtering and ordering on server-side. Client is then free to choose what amount of data will remain cached.

A *resource* is a primary data representation in REST. A resource can be a singleton or a collection. For example, *songs* is a collection resource and *song* is a singleton resource. We can identify *songs* collection resource using the URI `/songs`. We can identify a single *song* resource using the URI `/songs/{songId}`. A resource may also further contain sub-collection resources. To enable sorting, filtering and pagination capabilities in resource collection we pass the input parameters as query parameters in URI [19].

Resources and URI

General design defines that we should support 6 resources: songs, artists, albums, genres, playlists and music files. Below is the list of URIs that are used to access these resources[6]:

- `/songs` is a collection of all songs in a music library
- `/songs/{songId}` identifies a single song
- `/albums` is a collection of all albums
- `/albums/{albumId}` identifies a single album
- `/albums/{albumId}/songs` is a collection of songs within an album
- `/artists` is a collection of all artists

[5] inter-process communication

[6] A more detailed and interactive documentation can be found at `https://yolloz.docs.apiary.io/`

- `/artists/{artistId}` identifies a single artist
- `/artists/{artistId}/songs` is a collection of songs of an artist
- `/genres` is a collection of all genres
- `/genres/{genreId}` identifies a single genre
- `/genres/{genreId}/songs` is a collection of songs of a genre
- `/playlists/{userId}` is a collection of playlists of one user
- `/playlists/{userId}/{playlistId}` identifies a single playlist of a certain user
- `/playlists/{userId}/{playlistId}/songs` is a collection of songs within one playlist
- `/download/songs/{songId}` identifies a music file related to a certain song
- `/ping` is an empty resource, it can be used to check if the server is running

The URI definition is pretty straight-forward. *Playlists* resource uses userId identifier to separate lists of playlists for different users. `Music files` does not exist as a collection as it would not make sense to download all songs. Instead we defined a `download` resource to distinguish between metadata resources and music file resources. A music file can then be identified using its metadata identifier from *songs* resource.

Collection resources can be parameterized by query parameters. `limit` parameter limits number of resources in a collection, `page` parameter specifies n-th page of results. An `orderby` parameter specifies property by which resources in a collection should be sorted, `desc` specifies, whether they are sorted in ascending or descending order. Finally, `filter` parameter filters out resources that do not match filter value.

Endpoints and Versioning

This API will surely evolve through time as new features will be added. Since we can not make all users use the most updated version of the application, we need to ensure that the changes maintain backwards compatibility. Many changes can be made easily so, for example adding data fields to a resource or adding new resource. Older application versions do not use new features, so they are not bothered by additional functionality. However, in case backwards compatibility can not be maintained we would like to be able to define a new version of API. For those cases we added a version identifier to endpoint URIs. Fileserver API endpoints are then located at *server_address*`/api/v1/`*resource_URI*, where *server_address* is URI to locate where HTTP server of Fileserver module is running, `/api` just informs, that is a call to API, `/v1` specifies API version and *resource_URI* specifies resource that clients wants to use.

Resource Representation

Resources in a REST API can be represented in various formats, such as JSON, XML or even HTML. We chose JSON format, because its simple to use both in JavaScipt, which we use in Manager module, but also in other languages. A singleton resource is represented as a JSON object, a collection resource is represented as an array of JSON objects. Music files are a specific resource, they are represented as a binary file.

Data Storage

Data storage is required to store music file metadata of music library. Extracting metadata takes time as sometimes whole files have to be scanned, so storing once extracted metadata increases performance.

We also want to store location of these music files. They can be located anywhere across the file system of the host device. Storing the location allows spot admins to preserve their preferred file structure, they just need to specify which files they want to include in music library.

We expect that target users may have a couple of thousands of music files. Such data sets are large enough that storing them in a database can be useful, however it is still possible to store them in a structured file. Data change rarely, only when music files are being added or removed and when playlists are being modified. Furthermore, no high concurrency during data modification is expected. ACID properties, while appreciated are not absolutely necessary, if we can secure data consistency.

Saving data in a custom structured file would require a lot of coding to maintain efficiency and consistency with growing number of music files. This solution requires a lot of effort compared to using an existing solution. Although we could optimise it to work reliably, we do not consider it worth the effort.

A common approach to enable data queries would be to store it in a relational database to make use of the power of SQL. Most relational database management systems (RDBMS) require a server to access the database. It is inconvenient to set up such server in target use cases. Similarly, most of NoSQL databases are inappropriate due to server deployment.

We decided to use SQLite library that implements a self-contained, serverless, zero-configuration SQL database engine [20]. It can easily store thousands of records, which is sufficient. It can be embedded in an application and stores its data in an ordinary disk file. It is cross-platform, written in C, but offers bindings in many programming languages including C++. SQLite guarantees ACID properties and fast data queries, that are both very beneficial. SQLite is a widely used library, its developers claim that it is one of the most deployed software modules, has good support, is well tested and continuously developed.

Database Schema

The database schema can be seen in figure 2.4.

The database schema we use is a normal form of a model containing metadata metadata for music files. We have added a table for playlists with a one-to-many

relationship with songs. We also added two views that simplify common queries for songs and albums from REST API.

SQLite supports 5 main data types: `NULL`, `INTEGER`, `REAL`, `TEXT` and `BLOB`. Other data types commonly known from other SQL implementations can be used, but will be stored as the most appropriate one of them. All tables use `INTEGER` surrogate keys as primary keys.

2.3.3 Player Module

The main feature of Player module is playing music. Besides that, it is important to implement caching functionality and API for communication over WebSocket.

Audio Decoding and Playback

Digital audio files contain digital description of analog sound. In general, pulse-code modulation (PCM) method is used that samples the amplitude of analog signals in regular intervals. Each sample is quantized - mapped to a nearest digital value. The two basic properties of a PCM data stream are sampling rate, which defines how many samples are taken per second, and bit depth, which determines the number of possible digital values that can be used to represent each sample [21].

Digital audio data can be stored in multiple formats. These formats can be split into three categories [22]:

- *Uncompressed or raw formats* contain original uncompressed digital audio data. Raw formats (such as PCM data) contain pure data without any headers (sampling rate, bit depth etc.), uncompressed formats (such as WAV) may contain headers and metadata about the audio file.

- *Lossless compressed formats* compress original data to take up less space on disk without losing any information, so that the original data can be recreated after decompression. Commonly used are FLAC or ALAC formats.

- *Lossy compressed formats* enable even greater reduction in file size at the expense of losing some of audio information and simplifying the data. Various techniques are used to minimize perceived quality loss for a human listener. Most formats offer various levels of compression. Higher levels of compression result in smaller file sizes but lower sound quality. Most well-known formats are MP3, AAC or Ogg Vorbis.

Output audio devices and their low-level APIs usually only support pulse-code modulation (PCM) data. To retrieve these data from a music file we need to use an appropriate decoder that can decode its file format. A good music player should support many formats, our MVP must support at least the main and most popular ones. We consider these to be MP3, AAC and OGG Vorbis for lossy formats and WAV and FLAC for lossless formats. MP3 format does not need to be explained, it is the most common format on the internet. AAC format was developed as a successor to MP3 and is used in Apple's iTunes service, but it is also a popular audio format for HD videos. Ogg (container format) and Vorbis (audio coding format), very often used together represent an open-source

alternative to the previous two formats. It is used by Spotify, among others, for audio streaming. In case spot admins want to use high-quality audio formats and have the infrastructure that supports it, they may use WAV and FLAC formats. These formats require several times more disk space and bandwidth, but with adequate setup may provide high-fidelity sound reproduction for music enthusiasts.

Library Choice

There are various libraries for audio decoding with different ranges of target use cases and codecs. There are libraries dedicated to one file format, small libraries offering just a couple of codecs for simple purposes and finally, big heavyweight ones.

We have decided to use FFmpeg library [23] for audio decoding. It is a leading multimedia framework able to decode, encode, transcode, mux, demux and play pretty much anything that humans and machines have created. While we do not need to use all its capabilities, the amount of available decoders is astonishing. It allows us to play many more formats that we require. The drawback of its capabilities is its complexity. Its not as easy to use and embed into the application compared to smaller libraries, but offers good possibilities for future development. FFmpeg is cross-platform and written in C/C++, so it integrates well with our module. An important note is that it is licensed under GNU GPL or LGPL as some codecs may be subject to licensing in proprietary software.

Many small libraries are preferred for simpler cases, such as playing music in video games, where music is only secondary feature and music files are managed by developers. Using multiple dedicated libraries would require implementing their different APIs, which seemed much more labour-intensive than using ffmpeg. Lastly, we have not found any libraries that offered a complete music player that would allow control over the process. They focused on providing music player as a ready-to-use service.

FFmpeg allowed us to decode music files into raw PCM audio data, which is most of the work, we just needed to output them to speakers. Unfortunately, all major platforms provide a number of various APIs to work with sound, so we looked for a library that would encapsulate them and provide one uniform API that we can use across all the platforms. Other capabilities of such a library are not too important for us as long as it can handle outputting audio in real-time.

We have chosen PortAudio library [24]. It is a well-know cross-platform C library for audio input and output. RtAudio is a similar library with similar capabilities, but we preferred PortAudio as we have used it briefly in the past on a different project. It works on an asynchronous basis, an output stream is provided a callback function that is periodically called when it needs more data. Within this function we can decode a small part of a music file using FFmpeg. This way we do not need to store an entire decoded PCM file in memory but we decode just small chunks on-demand.

File Caching

We need to download and cache music files in advance to have them ready for playback.

Cached files can be stored in storage or in memory. This decision is important for miniature PCs that operate with limited resources. Storage option would allow us to cache more songs and provide larger cache for big files. However, these devices usually use flash drives for storage which do not handle frequent writes very well and usage of our application would wear these memories down too early.

Storing cached music files in memory is limited by memory capacity which is usually between 1 and 4 GB on these devices. To provide a smooth music playback we would like to have at least three songs cached - one being played, one ready to be played next and one being loaded. An average size of a music file using compressed format can be around 10MB, so three files would take up 30 MB of space. This should not cause any problems. Even using lossless file formats where file sizes are around 70MB would require just 210MB which should be possible to handle.

We decided to store cached files in memory. It gives us some more performance as I/O operations do not have to be performed while downloading songs and when they are being played. This approach is also advantageous with regards to how a public Fileserver would work. It would not be convenient to allow users to save songs directly on their devices during playback as these files could then be redistributed further. Storing them in memory makes it much harder to retrieve these files.

WebSocket API

Player module implements a WebSocket server that a Manager module may connect to to take over control. They exchange RPC messages and notifications encoded in JSON-RPC format via a WebSocket connection.

We have designed simple communication rules based on an idea of a *session*. A session begins when a Manager module establishes a WebSocket connection with a Player module. One Player can support at most one session at a time. When the session begins, it is in an uninitialized state. To get to an initialized state, Manager module must send a request with an address of a Fileserver module that it uses. When the Player verifies that it can connect to that Fileserver, communication can move to an initialized state. Now Player can start playing music and Manager module can start sending playback control commands. When events occur on Player, it sends notifications back to Manager module. When Manager module resets the session or when the connection is terminated the session ends and Player awaits for a new connection.

This schema is very simple and suitable for cases when a stable connection can be guaranteed, such as when both modules run on the same device or are on a wired LAN connection, because it is easy to implement. While we think it should be sufficient for the purposes of MVP, it sure needs to be improved in the future to provide more robustness and adaptation in case the connection is unstable. However, it would require not only upgrading the rules but would also require a lot of other features to be developed.

For example, when a connection is lost, after reconnecting both modules need to be able to reliably identify themselves mutually to ensure that they are continuing a previous session and its state can be synchronized. Also there is a question how to reliably notify a spot admin that a connection was lost and can not be

re-established. They might not be near the device, the application may be running in a background. Playing a notification sound can be disruptive for guests. Without a proper notification a silence would occur anyway when Player runs out of files to play. A solution might be to implement a fallback cache on Player. It would store some selected music files on a local drive and when a connection is lost to either module, it would use the fallback cache. This would give time to spot admins to react to connection loss.

Considering these options, we concluded that we do not yet have enough information, experience and feedback to effectively design more stable communication rules. The proposed features require a lot of work to be done and may or may not be desired by users so we decided to wait until we get some feedback to revise these rules.

WebSocket Commands and Notifications

The JSON-RPC messages follow a strict structure, to send such messages we have to specify method/command name and parameters. Parameters are either empty or a JSON object. Below is the list of currently supported commands:

- *Play* command begins the playback at current location.
  - method: `PLAY`
  - params: none
- *Pause* command pauses the playback at current location.
  - method: `PAUSE`
  - params: none
- *Next* command moves playback to next item in queue.
  - method: `NEXT`
  - params: none
- *Reset* command resets the player and moves it to uninitialized state. To begin playing an *Initialize* command must be sent.
  - method: `RESET`
  - params: none
- *Update queue* command updates the queue of Player module to the sent order. Cache will be updated accordingly.
  - method: `UPDATE_QUEUE`
  - params: an array of songId/itemId pairs.

```
{
    "queue":
        [
            {
                "songId": "125a",
```

```
                "itemId": "d15ca"
            },
            {
                "songId": "ad23",
                "itemId": "d187b"
            },
            ...
        ]
}
```

- *Initialize* command sets the Fileserver address for Player module and initializes session, making it ready for playback.
  - method: `INITIALIZE`
  - params: Fileserver url

```
{
    "url": "http://localhost:26500"
}
```

- *Volume* command sets the volume of playback.
  - method: `VOLUME`
  - params: desired volume level in the range 0-100

```
{
    "volume": 87
}
```

Besides commands sent by Manager module there are also notifications sent by Player module. These follow the same principles, the difference between them is that notifications do not have id parameter in JSON-RPC message structure.

- *Song failed* notification informs that Player was unable to play the song.
  - method: `SONGFAILED`
  - params: identification of song that failed.

```
{
    "songId": "125a",
    "itemId": "d15ca"
}
```

- *Song started* notification informs that Player started playing a new song.
  - method: `SONGSTARTED`
  - params: identification of song that failed.

```
{
    "itemId": "d15ca"
}
```

- *Request playlist* command informs that Player does not have enough songs in cache and requires more to be generated.
  - method: `REQUESTPLAYLIST`
  - params: none
- *Status* command informs about the change of playback status.
  - method: `STATUS`
  - params: current status

```
{
    "timestamp": 26500,
    "playing": false
}
```

- *Fileserver disconnected* command informs, that Player can not connect to Fileserver and that it was reset to uninitialized state.
  - method: `FILESERVERDISCONNECTED`
  - params: none

WebSocket Library

We wanted the library to offer asynchronous message processing. Some of available libraries offered only WebSocket client, most of the feasible libraries relied on ASIO / Boost.Asio library. We have decided to use another one of Boost libraries, Boost Beast library [25]. It is one of the newer Boost libraries and its design closely follows that of Boost.Asio, so those who already worked with it should get used to it quickly.

2.3.4 Data Management Module

Data Management module utilizes firebase platform. It uses its Authentication feature and Realtime Database.

Authentication feature is used to authenticate both spot admins and guests. They can use e-mail and password to sign in. Authentication tool manages registered users and safely stores their credentials and provides authentication tokens to signed in users. This token is required to access Realtime Database.

Realtime Database stores data that are exchaged between spot admins and guests. This is a NoSQL database in JSON format with a tree-like structure. Applications access this database using Firebase SDK. It provides a set of functions for data manipulation. To read or modify the data, application has to specify which node within the JSON structure it is interested in. These functions are then applied to this node.

In Realtime Database, clients implement logic for working with it. Realtime Database is schemaless, so it is important to ensure data consistency since data manipulation is distributed among clients. It supports defining a set of rules to validate consistency of database structure and to restrict access to data. Validation rules ensure that writes preserve consistent structure. Authorization rules

restrict read and write access to database. Authorization rules cascade, so granting read/write access to a node grants same access to all its child nodes [26].

It is advised to design the database to have shallower and wider structure compared to SQL schemas due to how data access and access rules work. When a node is loaded, all its child nodes are loaded as well, so a deep narrow structure would result in a lot of transferred data even when only a part of the information is required. Data redundancy is also not an entirely discouraged concept. When used cautiously the database can offer better performance and less reads are required to get all data. Firebase provides transactions that can be used for consistent updates of redundant data and other cases.

Records in lists are usually identified by a unique key that Realtime database can automatically generate. Besides being unique globally, these keys can also be easily sorted in lexicographic order that is the same order which they were created in. We rely on this feature to determine the exact order of ordered songs.

Lastly, Firebase supports subscribing to receive notifications about various events that occur at a specific location in the tree structure like direct changes on a node or creation and removal of child nodes. This is another feature that we use on various places.

Database Structure

Database has a nested tree structure. Every field can either be an elementary data type, an object or a list of objects. A list of objects is just an object containing key-value pairs. To be able to describe our database structure we use custom notation that describes types of objects in our database. Normal text represents exact field key, text in brackets represents an identifier key. An arrow (`->`) represents a level of nesting. Expressions enclosed in braces represent a list of objects each with a defined key : value format. We will describe value formats later.

```
libraries -> { [spotId] : Library }
que -> { [spotId] : QueItem }
spots -> public -> { [spotId] : PublicSpotInfo }
      -> private -> { [spotId] : PrivateSpotInfo }
users -> public -> { [userId] : PublicUserInfo }
      -> private -> {[userId] : PrivateUserInfo }
```

Top-most objects contains four fields: libraries, que, spots and users. *Libraries* field is a list of music spot libraries. Each library is identified by `[spodId]` key and has a value in `Library` format. Similarly, *que* field contains a list of music spot queues. *Spots* and *users* fields contain two child fields, *private* and *public*. *Public* fields contain publicly readable spot/user info, *private* fields contain private portion of data visible only to each user/spot. This division is necessary due to how access rules work in Firebase - giving access to a node means having access to all child nodes.

In order to keep the previous schema readable we introduced some value formats. Below is a description of value formats. It starts with a name of the format and key type separated by hyphen. All keys are of string type, key type just distinguishes between context of keys, e.g., if two keys of different types have

different value, they are still different keys. Below the name, indented, is a list of child fields and their type separated by colon. A type can be either String, Boolean, Number or another value format.

Entities Structure

```
PublicUserInfo - [userId]
    name : String

PrivateUserInfo - [userId]
    credits : Number @Optional
    adminForSpot : [spotId] @Optional

QueItem - [pushId]
    songId : String
    userId : String @Optional # if missing then item was added by spot

PublicSpotInfo - [spotId]
    name : String
    active : Boolean
    description : String
    address : String

PrivateSpotInfo - [spotId]
    billingInfo : BillingInfo

BillingInfo
    bankAccount : String
    variableSymbol : String

Library - [spotId]
    artists : [artistId] : MusicEntity
    albums : [albumId] : MusicEntity
    genres : [genreId] : MusicEntity
    songs : [songId] : Song
    lists : [songListId] : List

List - [songListId]
    [songId] : Boolean # always true

MusicEntity - [id]
    songListId : [songListId]
    name : String

Song - [songId]
    name : String
    length : Number
```

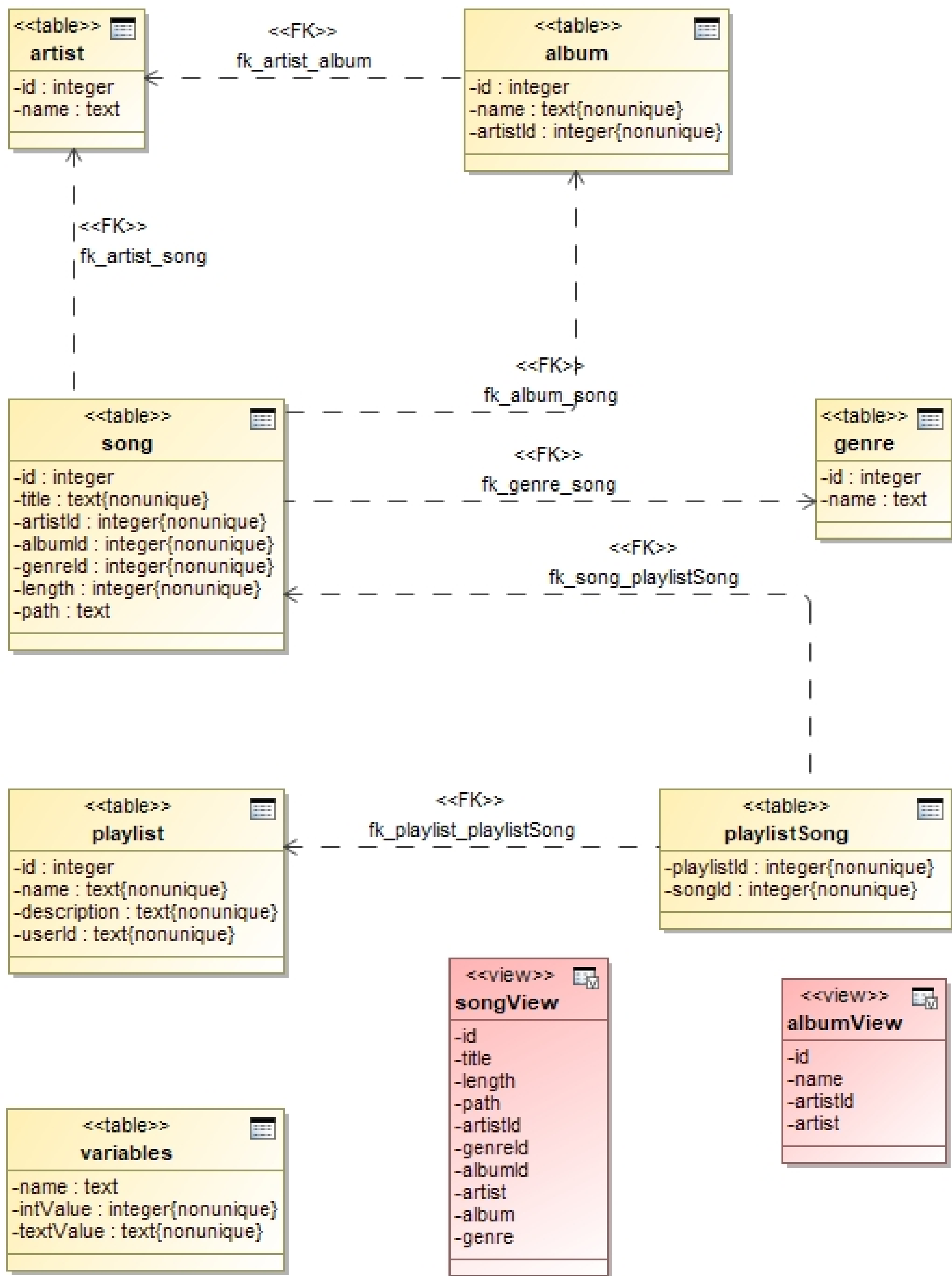


Figure 2.4: SQLite database schema in Filesrver module

3. Implementation

We will describe technical details of our implementation, interesting problems we faced and their solutions in this chapter.

3.1 Manager Module

Manager module provides user interface for our application. It is written as a single-page web application and runs in an embedded Chromium browser. Other modules are managed from this module by utilising their communication interfaces. Our implementation supports running its own Player and Fileserver module to provide an all-in-one product as well as connecting to remote modules.

3.1.1 Web Application Part

Developing an SPA web application is different to traditional web pages. We will describe the specifics of our project compared to a traditional web page development so the reader can get a better understanding of the implementation.

JSX Syntax

A traditional web page contains markup in HTML files, JavaScript scripts in JS files (or script tags when they are simple) and styling in CSS files. React provides JSX syntax so that markup can be used in JS files. To define a visual component, we may write a single JavaScript code that describes how the component looks, how it behaves and how it reacts to events. The JSX syntax is similar to HTML, component structure is described using nested tags. In advance to HTML tags it may contain tags representing user-defined React component, e.g., a component `BlueButton` can be added to layout by adding a tag `<BlueButton />`.

Application Entry Point

A traditional web page entry point is `index.html` file that contains hyperlinks to other pages. Our application entry point is also a `index.html` file. However, our file is practically empty, contains only the basic HTML structure and serves as a placeholder. The real entry point is `app.js` file. In there we set up React and Redux and tell ReactDOM[1] which element from our `index.html` file it should render the application into. At this point ReactDOM renders our top-most component, forcing it to render its sub-components and the application starts running.

Routing

Similarly to common web pages the page layout is determined by path/URI, in React called *route*. However, since this is a single page application, we have nowhere to navigate. Instead of loading web page from URI location, React apps use routing libraries to implement virtual routing. We are using React Router

[1]ReactDOM is a React library that handles virtual DOM rendering.

library. A route is only symbolic, the real URI location is always `index.html`. Route definitions specify what components are rendered at which location. Navigating to different routes forces application to re-render it's content. Nesting the route can make only a part of the page to re-render.

Components and Styling

As mentioned in the previous chapter, there are two main categories of our React components. *Components* contain mostly layout and styling while *Containers* contain most of the logic and behavior, but no markup beyond what is necessary. To style the application we used Material-UI library. It offers styled React components in popular material design developed by Google. We used these components to create a modern and appealing UI which should look familiar to users. Material-UI provides a theme that sets up what its components look like.

Outside that we only needed little visual styling of our components, most of the styling was related to positioning. Due to that we do not use any CSS files apart from one that sets up fonts. Instead, required component styles are defined within *Component* files as JavaScript objects. While React natively supports CSS, it also allows to define styles in JavaScript and JSX, which we preferred to using CSS locators.

Actions and Action Creators

Default Redux actions are plain JavaScript objects, in our case containing `type` data field with string constant defining type of action and `payload` data field containing action parameters. These actions are usually created using *action creators*, which are functions that create actions. Basic action creators create only basic actions by providing correct `type` constant and packing parameters into `payload`.

We also extensively use *thunks*. A thunk is a function that wraps an expression to delay its evaluation. A typical use case of actions is to dispatch its action creator in response to an event. Using thunks we can write action creators than are conditional, e.g., an action may or may not be dispatched depending on the current state. Additionally, we often use asynchronous functions and JavaScript Promises. Thunks are the way to implement asynchronous actions, e.g., so that an action is dispatched only after the promise was resolved or an error action is dispatched when a promise failed.

Reducers

Our Redux store is divided into multiple objects each managed by one reducer. This layout reduces the amount of code within one reducer to a manageable amount. Reducer performs code changes in an immutable way. When state is being changed, new object is created using `...` (three dots) spread operator. This approach is inconvenient for deep nested objects, because new objects have to be created at each level of nesting, therefore we designed state objects to be shallow.

Communication With Other Modules

To communicate with Fileserver module, we use axios library. It is an asynchronous HTTP client for the browser and Node.JS [27]. We use it to send asynchronous HTTP requests to Fileserver. It simplifies sending AJAX requests and provides modern Promise API.

WebSocket communication with Player module utilizes WebSocket object from WEB API in browser. Upon establishing connection with Player, the created socket is stored in store. The messages sent to Player are in JSON-RPC format. We use json-rpc-protocol library to create and deconstruct messages. To implement asynchronous request-response behavior of RPC messages we store success and failure callback functions for each request. When a response arrives, a corresponding callback function is invoked to resolve the asynchronous request.

Firebase provides software development kit (SDK) in JavaScript. Every communication with Data Management module utilizes this SDK. It provides methods for reading and manipulation of data as well as for registering callbacks. When the user signs in, an authentication token is received and all subsequent calls to firebase use this authentication. Module initially loads music spot information. When spot admin selects a list of songs for playback, these are processed to match firebase structure, uploaded and it subscribes to queue change notifications. When the playback ends, list of songs and queue are removed to clean up the database.

Boilerplate and Other Libraries

We used react boilerplate[2] as a basis for our web application project. This boilerplate contains basic project setup, configuration files, scripts, useful tools, code samples and libraries. A simple React project can be created without it, for larger projects it is convenient to use updated maintained boilerplate and customize it.

Boilerplate provides configuration for Webpack. It is a bundler commonly used by modern web applications. It internally builds a dependency graph which maps every module a project needs and generates one or more bundles. It ensures that only code that is used is imported from libraries and that projects can be organized in a developer-friendly way while a bundle works well in a browser. It also supports Hot Module Replacement (HRM). During development, it replaces modules while application is running without the need for full reload so the changes we make are instantly visible. In order to run the web application we either need to build it using build command to create a bundle or run the HRM feature.

React-intl library is used to create a localized application. All strings in components are grouped in *messages.js* file next to the code files, marked with unique identifier and a default translation is supplied. When a string is needed its identifier is passed to a function from react-intl library which takes care of finding the right translation. When the application is built all strings are extracted to a JSON file. Multiple JSON files can be provided with different translations to support multiple languages.

React-virtualised library provides infinite and virtual scroll components. The data in music library are loaded in small amounts using AJAX requests. Infinite

[2] `https://www.reactboilerplate.com/`

scroll allows to dynamically load more data when they need to be displayed to user. Virtual scroll is necessary to deal with displaying long lists of data. With growing number of items in a list, more DOM elements are required to render it. The more DOM elements there are, the slower the rendering becomes. Virtual scroll renders only a few items that the user can see on screen and dynamically changes contents of these elements as the user scrolls to simulate scrolling.

GUI Design

The GUI is divided into multiple boards that the user can switch between using navigation bar at the top of the screen. Each board offers controls for different tasks. At the bottom of the screen is player bar offering access to player controls across all boards. There are 5 boards:

- Library
- Establishment
- Playback
- Devices
- Settings

Library board displays contents of the music library of connected Fileserver module. It offers controls for filtering available songs, browsing genres, albums and artists and creating and modifying playlists defined by the spot admin. Any list of songs in the library can then be selected for music playback. If the application uses local integrated Fileserver, it also provides controls for adding and removing music files in the library.

Establishment board displays general information about the music spot that guests can see in mobile application.

Playback board is used to manage music playback. It displays list of currently available songs and details of queues. There are three available queues: priority, order and playlist queue. Priority queue can be used by spot admin to select songs to play with highest priority. Order queue is filled with songs ordered by guests. Playlist queue contains songs that will play when there are no songs in other two queues. Songs generated by Manager module are added to this queue.

Devices board is used to manage connections to Fileserver and Player modules. Users may choose to use a local one integrated in this module or connect to a remote one.

Settings board is used for editing application settings.

3.1.2 CEF Wrapper and Integration of Other Modules

For the browser support and desktop application wrapper we used Chromium Embedded Framework (CEF).

CEF Structure

CEF runs using multiple processes. The main process which handles window creation, painting and network or file system access is called the *browser* process. This is generally the same process as the host application and the majority of wrapper logic will run in the browser process. DOM rendering and JavaScript execution occur in a separate *render* process [28]. CEF behavior can be controlled and modified by overriding methods provided by classes of each process. CEF provides default implementation to most of the methods.

We used one of examples that CEF provides, *cefclient*, as a template for our project. Our classes subclass several CEF interfaces and classes and override only methods that we needed to modify.

CEF Integration

When the application starts the browser process (implemented by `Client` class) is created and initialized. Then a new render process is spawned for each unique origin (scheme + domain). In our case there will be a single render process for the web application part. Since CEF runs in multiple processes it is necessary to provide mechanisms for communicating between them [28]. For this CEF provides its own inter-process communication mechanism. Besides exchanging process startup and runtime messages it also supports custom message routing. CEF provides a generic implementation for routing asynchronous messages between JavaScript running in the render process and C++ running in the browser process. The renderer-side router supports generic JavaScript callback registration and execution. We registered a callback function `cefQuery` in JavaScript that allows sending messages to browser code. The renderer-side router registers success and failure callbacks passed to `cefQuery` function and sends a JSON-encoded message to the browser-side router. The browser-side router supports application-specific logic via one or more provided Handler instances [28]. We provide four Handlers: `MsgHandler_MusicPlayer` integrates Player module within Manager module and handles starting and stopping it. `MsgHandler_FileServer` integrates Fileserver module within Manager module and handles Fileserver management commands. This way we were able to add Fileserver management UI into Manager module GUI. Utilisation of this UI is restricted to using integrated Fileserver module. `MsgHandler_Configuration` handles loading configuration and settings of web application from file stored in file system. `MsgHandler_WebLogger` logs messages and errors from web application to log file located in file system.

3.2 Fileserver Module

Fileserver consists of two parts, data storage that stores information about music library and REST API that allows other modules to access the library.

3.2.1 Data Storage

Music library structure is stored in SQLite database. `SqliteAPI` class takes care of operations on it. SQLite provides rich SQL syntax, but each SQL statement

has to be provided in a string. A typical query consists of several steps: first, a generic SQL statement string with parameter placeholders is constructed. Next, it is compiled against a database file into a binary form. We can inject parameter values to a compiled statement, one compiled statement can be used multiple times with different parameters, but we do not use this feature. The advantage of injecting parameters is a protection against SQL injection. Finally, we can step such statement to receive all rows in query result.

When new files are added to the database, they are first scanned by `AudioInspector` class for metadata. It uses FFmpeg library, which we already use in Player module.

It is possible that file paths stored in database may get outdated as files are moved or deleted in file system. In case a file is not found when it is requested we store a flag that can be used to notify Fileserver module operator about missing files so the database can be kept consistent.

`SqliteAPI` public interface is separated on two parts, one is used by REST API to get resources from database, the other is used by Fileserver module operator to manage its contents. This is either a command line wrapper that we implemented in `FileserverModule` project or Handler in CEF Wrapper.

3.2.2 REST API

To receive and send HTTP requests we used C++ REST SDK library [29]. It provides an HTTP listener that listens for incoming requests and a set of asynchronous methods to handle these requests in a modern and efficient way. It provides its own implementation of asynchronous tasks. Tasks are executed in a managed thread pool.

To create an abstraction between the HTTP listener and data storage we utilized the bridge design pattern. It allowed us to create HTTP listener independent from data storage implementation. Coupling it with another data storage would just require implementing the abstract class `AbstractFileServerHandler` in a new bridge.

The `FileServerAPI` class wraps the HTTP listener. Upon receiving a request from HTTP listener a corresponding HTTP method handler parses URI and parameters and validates the request. Then it passes the parameters to a corresponding method of the associated bridge which executes the action on underlying data storage and returns data for the response. This response is then sent back to the client.

`FileServerHandler` class is a bridge implementation for SQLite data storage. It owns an `SqliteAPI` class instance and calls its methods to retrieve data for requests. It then transforms the data to JSON format supported by REST API.

3.3 Player Module

The main feature of Player module is playing music. Besides that, it is important to implement caching functionality and API for communication over WebSocket.

3.3.1 File Caching

Caching is implemented in `SongCache` class. This class contains HTTP client from C++ REST SDK library that it uses to download music files into cache. On input it receives a sequence of songs and outputs next music file to be played when it is ready.

Internally, `SongCache` utilizes `SongCacheItem` class for storing each queued item. This class holds an asynchronous buffer that is used to download the data asynchronously and upon completion it provides a `std::basic_istream` wrapper for reading cached file contents. Before accessing it, it is necessary to check for errors.

To provide a little more robustness for song skipping or sudden queue changes we always try to cache 3 songs and one additional song is being played. In case there are less than 3 songs provided by Manager module, this implementation notifies it using API notification to supply more songs until the cache is full. The cache never contains more than 5 songs at one. When already cached songs are pushed down in order then only first 5 songs are allowed to stay, the rest is removed and would be cached again later.

3.3.2 Audio Decoding and Playback

Playing audio files is done in two steps. First, music file has to be decoded to produce pulse-code modulation (PCM) data which are then sent to the output audio device.

We utilize FFmpeg library [23] for audio decoding and PortAudio library [24] for working with audio device. First, audio file is inspected by FFmpeg library to detect the right decoder. Then a PortAudio output stream is started. A PortAudio stream works like a sink. It is filled by an asynchronous callback function running in a real-time thread and sends data to audio device. This function is called every time the stream's buffer is getting empty. Within this function a required amount of music samples are decoded using FFmpeg library and written to buffer of audio device. This functionality is contained within `MusicPlayer` class.

Although we support only MP3, AAC, OGG, WAV and FLAC formats, our implementation should support decoding all formats that FFmpeg contains a codec for. However we were only able to test the application with the supported formats. We would not like to claim that we support others without testing.

3.3.3 WebSocket API

This module implements a WebSocket server that awaits connection from a Manager module. We used Boost Beast library [25] that takes care of asynchronous communication over WebSocket.

The functionality of the server is layered into multiple classes. At the top is the `API` class. This class wraps the entire server and offers methods to start and stop the server.

When the server is started, `API` class instance runs an instance of `Listener` class. This class provides functionality to listen for and accept incoming connections asynchronously.

After a new connection is accepted, an instance of `Session` class takes over the created socket and handles the incoming ant outgoing communication. It represents a session of Player module during which the music playback occurs. This instance owns song cache and music player and takes care of cooperation between them. Reads and writes on WebSocket are performed asynchronously.

Conclusion

In this thesis we designed and implemented a music application that can play music files stored on the same or a different device and the order in which the songs play can be influenced by multiple people using a mobile application. The application supports two levels of access, administrator and guest. The administrator has direct control over music files and music playback, guests may browse available songs on their mobile phones and enqueue them into playback.

Before we implemented our project we have analyzed user requirements and designed a concept of an application that can be used in both public and private places. We created a design that clearly defines how can the application be steadily improved to support new features and attract new users. We split the application into several modules which allows users to customize the application for their specific needs and offers more freedom in application development.

Our application contains essential features and aims to be a minimum viable product, so that the first users can start using it and provide valuable feedback required for further development. We used technologies that should allow us to build new features on top of the existing ones without risking to hit their limitations.

In the near future the application should be made available for users. One of the first possible improvements is to simplify configuration even more by implementing a network discovery of modules or creating a file system watcher that automatically synchronizes contents of music library with file structure in a dedicated folder on disk. Eventually, music library can be connected to an API that allows downloading richer song metadata or finds metadata for files which miss it.

Bibliography

[1] BarBox. Barbox. http://barboxapp.com/main, 2019. [Online; version 2019-06-19].

[2] Noispot. Noispot. http://noispot.com/, 2019. [Online; version 2019-06-19].

[3] s.r.o Mood Media Group CZ. Mood mix. https://www.moodmedia.cz/mood-mix/, 2019. [Online; version 2019-06-19].

[4] Winamp. Winamp. https://www.winamp.com/, 2020. [Online; version 2020-07-17].

[5] Bitterhead. Ampwifi. http://www.blitterhead.com/ampwifi-android-app, 2020. [Online; version 2020-07-17].

[6] VideoLAN. Vlc media player. https://www.videolan.org/index.html, 2020. [Online; version 2020-07-17].

[7] VLC Mobile Remote. Vlc mobile remote. https://vlcmobileremote.com/, 2020. [Online; version 2020-07-17].

[8] Electron. Electron. https://electronjs.org/, 2018. [Online; version 2018-12-16].

[9] Chromium Embedded Framework. Chromium Embedded Framework. https://bitbucket.org/chromiumembedded/cef, 2018. [Online; version 2018-12-16].

[10] Rosa Alarcon, Erik Wilde, and Jesus Bellido. Hypermedia-driven restful service composition. In E. Michael Maximilien, Gustavo Rossi, Soe-Tsyr Yuan, Heiko Ludwig, and Marcelo Fantinato, editors, *Service-Oriented Computing*, pages 111–120, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[11] M. Bosi and R.E. Goldberg. *Introduction to Digital Audio Coding and Standards*. The Springer International Series in Engineering and Computer Science. Springer US, 2012.

[12] Inc. MongoDB. NoSQL vs SQL databases. https://www.mongodb.com/nosql-explained/nosql-vs-sql, 2020. [Online; version 2020-07-21].

[13] Inc. Cloudflare. What is BaaS? https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/, 2020. [Online; version 2020-07-21].

[14] Google. Firebase. https://firebase.google.com, 2019. [Online; version 2019-01-27].

[15] StatCounter. Desktop Operating System Market Share Worldwide. http://gs.statcounter.com/os-market-share/desktop/worldwide, 2020. [Online; version 2020-07-25].

[16] The Qt Company. Qt. https://www.qt.io/, 2019. [Online; version 2019-04-27].

[17] Facebook Inc. Introducing jsx. https://reactjs.org/docs/introducing-jsx.html, 2019. [Online; version 2019-05-22].

[18] Dan Abramov and the Redux documentation authors. Redux. https://redux.js.org/, 2020. [Online; version 2020-07-21].

[19] restfulapi.net. Rest resource naming guide. https://restfulapi.net/resource-naming/, 2020. [Online; version 2020-07-21].

[20] SQLite. Sqlite. https://www.sqlite.org, 2019. [Online; version 2019-04-27].

[21] Wikipedia contributors. Pulse-code modulation — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Pulse-code_modulation&oldid=968950618, 2020. [Online; accessed 28-July-2020].

[22] Wikipedia contributors. Audio file format — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Audio_file_format&oldid=967154520, 2020. [Online; accessed 28-July-2020].

[23] FFmpeg. Ffmpeg. https://ffmpeg.org/about, 2019. [Online; version 2019-04-27].

[24] PortAudio community. Portaudio. http://www.portaudio.com/, 2019. [Online; version 2019-04-27].

[25] Vinnie Falco. Boost.beast. https://www.boost.org/doc/libs/1_70_0/libs/beast/doc/html/beast/introduction.html, 2019. [Online; version 2019-04-27].

[26] Google. Firebase. https://firebase.google.com/docs/database, 2019. [Online; version 2019-01-27].

[27] axios. axios library. https://www.npmjs.com/package/axios, 2019. [Online; version 2019-05-22].

[28] Chromium Embedded Framework. Chromium Embedded Framework. https://bitbucket.org/chromiumembedded/cef/wiki/GeneralUsage, 2019. [Online; version 2019-05-22].

[29] Microsoft. C++ rest sdk. https://github.com/Microsoft/cpprestsdk, 2019. [Online; version 2019-04-27].

List of Figures

1.1 Modules and relationship between them 10

2.1 Desktop Operating System Market Share Worldwide from January 2017 - June 2020 [15] 23
2.2 JSX syntax example [17] 28
2.3 React + Redux flow 30
2.4 SQLite database schema in Filesrver module 42

A.1 Login screen 55
A.2 Spot creation wizard 56
A.3 Adding songs to playlist 58
A.4 Playback screen 60

A. User manual

In this part we provide a user manual for our application.

A.1 Juke It

JukeIt is an application for running and managing a music spot. In a music spot, you play music for your visitors and offer them the option to interact with what is being played. You can create your playlists, manage your music, control the playback and other related actions.

JukeIt application is ready to use after unzipping the provided zip file in the attachments. It runs on Windows 7 or newer, requires at least 1GB RAM and 250MB free disc space.

A.1.1 Sign in and sign up

To enter the application you need to sign in with your user account (see Fig. A.1. If you don't have one, continue to sign up page and create one by entering your e-mail address, password and name, which will be visible to other users.

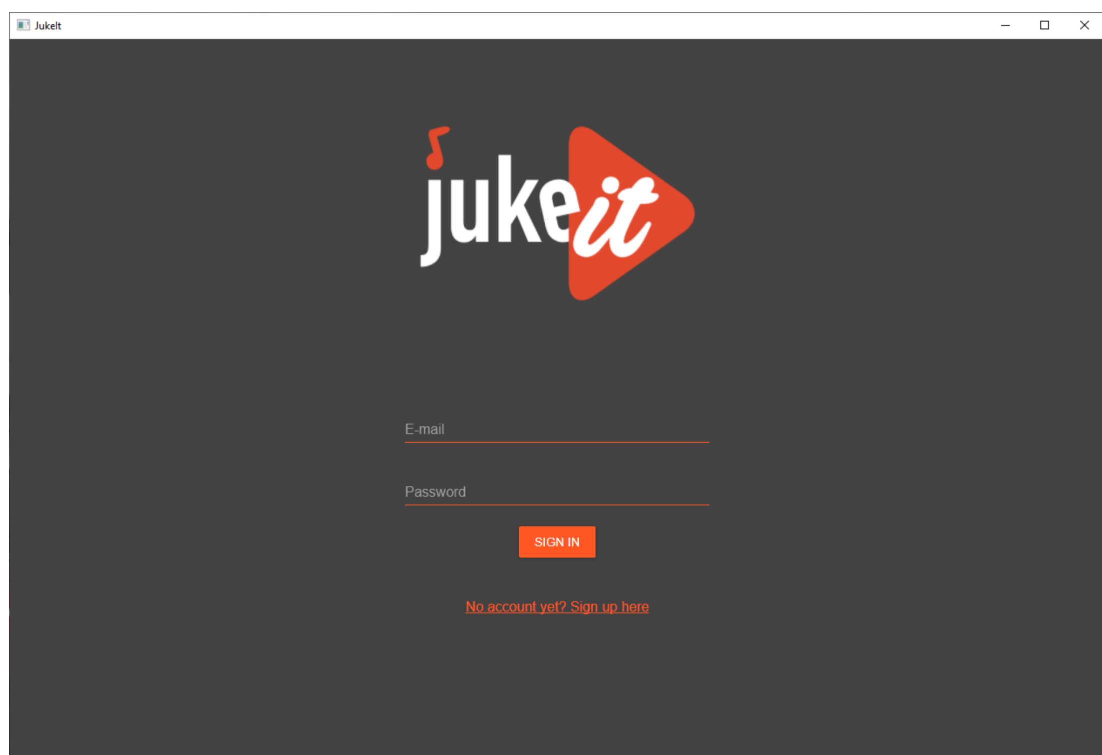


Figure A.1: Login screen

A.1.2 Music spot creation

In JukeIt, every music spot is tied to one user account. If you don't have one yet, after signing in you will be taken to *music spot creation wizard* (see Fig. A.2), where you enter the necessary information for your new music spot. This information can be further modified after the creation in the application.

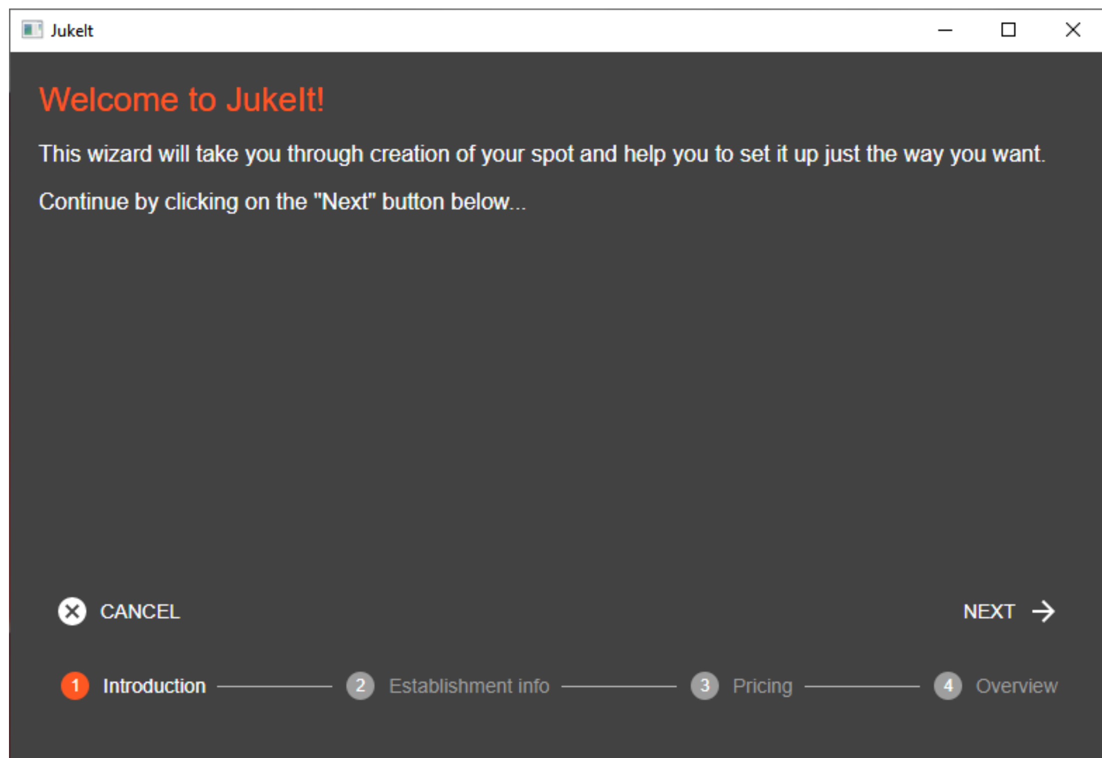


Figure A.2: Spot creation wizard

A.1.3 User interface

JukeIt provides several different screens accessible from tabs at the top of the application:

- **Library** - allows you to browse the music library you are connected to and manage your playlists
- **Establishment** - displays information about your music spot and allows you to edit it
- **Playback** - provides interface for managing music spot playback and song queues
- **Devices** - you can manage connections to Fileserver and Player from this screen
- **Settings** - allows you to change several preferences in the application

Additionally, at the bottom of the application is the music player widget that allows you to control music playback across all screens.

A.1.4 Devices

JukeIt consists of three parts - Manager, Fileserver and Player. Manager application provides user interface for managing your music spot. Fileserver provides access to a music library while Player handles playing and outputting audio. With Fileserver and Player you can opt to use local or remote device. Local is running directly from your Manager application. Remote runs on a different device and you connect to it via network. This option allows you to customize the usage of JukeIt based on your needs, for example running a remote Player on a device, that is right next to speakers in your music spot while running Manager application on a PC in your office.

A.1.5 Setting up JukeIt

In this section we will describe how to start playing music with JukeIt step by step.

Preparing your playlist

First we need to prepare a playlist that will be played on our music spot. This can be done from *Library* screen.

If you're using local Fileserver, you can add new music files to your library by clicking on *Add files* button in top right corner. Remote Fileserver does not support file management and should already be set up. By default, JukeIt connects to local Fileserver.

To create a playlist, go to *Playlists* tab on *Library* screen. Click on *Add playlist* button and fill the name and description fields in the dialog window that appears. After clicking *Save*, a new blank playlist is created. To fill it with songs, navigate through library and select songs you wish to add. We can add them either by right-clicking the desired song and selecting option *Add to playlist* from context menu or selecting multiple songs at once by clicking *three dots* button located above the list on the right side and selection option *Add to playlist* (see Fig. A.3). Then we can select songs we wish to add and confirm by clicking *Add* button that appears above the list on the right side.

Publishing a playlist

Once the playlist is ready we need to publish it so the guests may see it from their mobile apps. This is done by clicking *Play* button in top right corner. Besides playlists, this action can be done on any list of songs in library, e.g. all songs from an artist.

When the playlist is published, you will be redirected to *Playback* screen (see Fig. A.4). There are three important buttons in top right corner. *Activate spot* changes the state of the music spot to active so the guests may start selecting songs. *Remove playlist* unpublishes current playlist in case we would like to select a different playlist. *Start playing* begins playing songs from current playlist (in case the playback hasn't started yet). At the center of the screen are located cards with available songs in current playlist, priority queue, order queue and playlist queue.

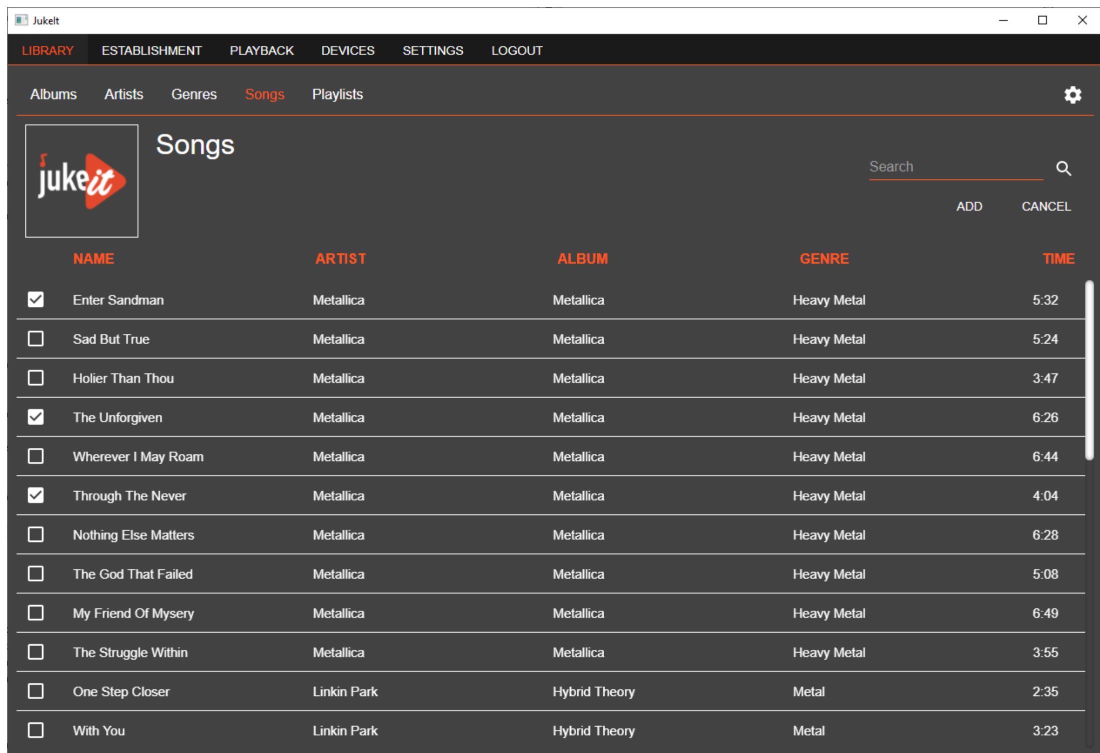


Figure A.3: Adding songs to playlist

Song queues

JukeIt utilizes three types of queues to determine song order. Each queue has set priority and songs in it are played only if there are no songs in a queue with higher priority.

Priority queue has the highest priority and songs in there are played first. Only you can add songs to this queue and it is intended to be used for songs that you wish to play immediately.

Order queue has medium priority. It contains songs ordered by guests in the order as they have done so from the mobile application. You can not add songs to this queue.

Playlist queue has the lowest priority and is used as a fallback option for when there are no ordered songs. You may add songs that you wish to play during blank intervals without orders, but in case this queue does not contain enough songs, JukeIt will randomly generate new songs for you. Note that Player requires to have always a couple of songs ready so there will always be more than one song based on the Player requirements.

A.1.6 Settings

On *Settings* screen you may set up your device preferences. You may set whether you want to use local or remote Fileserver and Player, how to connect to remote, whether to connect to them immediately on application start up etc. It allows you to save these settings without the necessity to enter this information every time.

A.2 Player Module

Player module provides a command line tool. To run it call *JukeIt Player* and specify parameters. When the module starts running, it is awaiting connections. To stop the module type `end` and press enter.

A.2.1 Parameters

Player module supports these parameters:

- *Address* (`--address`, `-a`, `/address`, `/a`) specifies address, where Player module will be accessible by application, e.g., `localhost`, `192.168.0.105`.
- *Port* (`--port`, `-p`, `/port`, `/p`) specifies port, on which Player module will be accessible by application, e.g., `26500`.

A.3 Fileserver Module

Fileserver module provides a command line tool. To run it call *JukeIt FileServer* and specify parameters. When the module starts running, it is awaiting connections. To stop the module type `end` and press enter. If you want to run Fileserver on different address than loopback/localhost, you need to run it with elevated privileges.

A.3.1 Parameters

Fileserver module supports these parameters:

- *Address* (`--address`, `-a`, `/address`, `/a`) specifies address, where Fileserver module will be accessible by application, e.g., `-a localhost`, `-a 192.168.0.105`.
- *Port* (`--port`, `-p`, `/port`, `/p`) specifies port, on which Fileserver module will be accessible by application, e.g., `-p 26550`.

Figure A.4: Playback screen

B. Instructions for Building the Project

In this part we explain how to set up environment and provide instructions on how to build the application on your own.

B.1 Prerequisities

In order to be able to build the application you are expected to have these applications and libraries ready to use:

- CMake 3.14 or higher (`https://cmake.org/download/`)
- NodeJS 8.0 or higher (`https://nodejs.org/en/download/`)
- Visual Studio 2017 or newer (`https://visualstudio.microsoft.com/downloads/`)
- Boost v 1.67 with pre-built binaries (`https://www.boost.org/doc/libs/1_67_0/more/getting_started/windows.html`)

B.1.1 Building Boost

To build boost libraries, follow these steps

1. Download boost v 1.67 from the Boost link above
2. Unpack, then place unpacked folder (boost_1_67_0) on desired location (e.g., `C:\Program Files\Boost`)
3. Create a temporary folder for intermediate boost build files
4. Open command line (if you use Visual Studio, open *Developer Command Prompt for Visual Studio* as Administrator) and navigate to boost (boost_1_67_0) folder
5. From the command line, navigate to unpacked directory and run following commands:
   - `bootstrap`
   - `.\b2 --build-dir=build-directory toolset=toolset-name --build-type=complete stage --with-system --with-date_time` where build-directory is a temporary directory for intermediate build files you've created earlier and toolset-name is name of build tool (use `msvc-14.1` for building with Visual Studio 2017/2019)
   - Add two environment variables to help CMake find Boost libraries:
     - `BOOST_ROOT` variable should contain path to root folder (e.g., `C:\Program Files\Boost\boost_1_67_0`)

  - BOOST_LIBRARYDIR variable should contain path to pre-built libraries in stage\lib folder (e.g., C:\Program Files\Boost \boost_1_67_0\stage\lib)

Refer to the Get started manual available via Boost link in case of additional questions.

B.2 Cloning and Setting Up Working Directory

The project repository is available on GitHub (https://github.com/janhofman/juke-it-cef) or in the attachment of this thesis. Some of the contents are too big to be stored at either place, but are available online. Follow these steps to prepare the working directory and build the application:

- (GitHub only) Clone the repository
- Download and unpack Standard Distribution of CEF for your OS, **version 3.3202.1683** (use version filter) from http://opensource.spotify.com/cefbuilds/index.html, then copy **Debug** and **Release** directories to project root folder
- (GitHub Only) Download and unpack additional external libraries from https://www.dropbox.com/s/13az32lsxirskrf/libraries.zip?dl=0, copy the libraries folder to project root folder
- On command line, navigate to root folder of your working directory
  - cd /path/to/project-root-folder
- Create and enter the build directory.
  - mkdir build
  - cd build
- Follow steps for your OS:
  - To perform a Windows build using a 32-bit CEF binary distribution:
    - Visual Studio 2017:
      - cmake -G "Visual Studio 15" ..
      - Then, open build\JukeIt.sln in Visual Studio 2017 and select *Build* → *Build Solution.*
    - Visual Studio 2019:
      - cmake -G "Visual Studio 16 2019" -A Win32 ..
      - Then, open build\JukeIt.slnin Visual Studio 2019 and select *Build* → *Build Solution.*
  - To perform a Windows build using a 64-bit CEF binary distribution:
    - Visual Studio 2017:
      - make -G "Visual Studio 15 Win64" ..

        - Then, open `build\JukeIt.sln` in Visual Studio 2017 and select *Build* → *Build Solution.*
      - Visual Studio 2019:
        - `cmake -G "Visual Studio 16 2019" -A x64 ..`
        - Then, open `build\JukeIt.sln`in Visual Studio 2019 and select *Build* → *Build Solution.*

- **Note**: sometimes CMake fails to read environment variables to locate Boost directories. If that is the case, the path to Boost directory can be supplied as a parameter to cmake command
  - `-DBOOST_ROOT="path_to_boost_root_directory"` specifies location of Boost root directory
  - `-DBOOST_LIBRARYDIR="path_to_boost_lib_directory"` specifies location of directory containing Boost libraries
  - `cmake -G "Visual Studio 16 2019" -A x64 -DBOOST_ROOT= "C:\Program Files\Boost\boost_1_67_0" -DBOOST_LIBRARYDIR= "C:\Program Files\Boost\boost_1_67_0\stage\lib" ..`
    is an example of a command containing these parameters

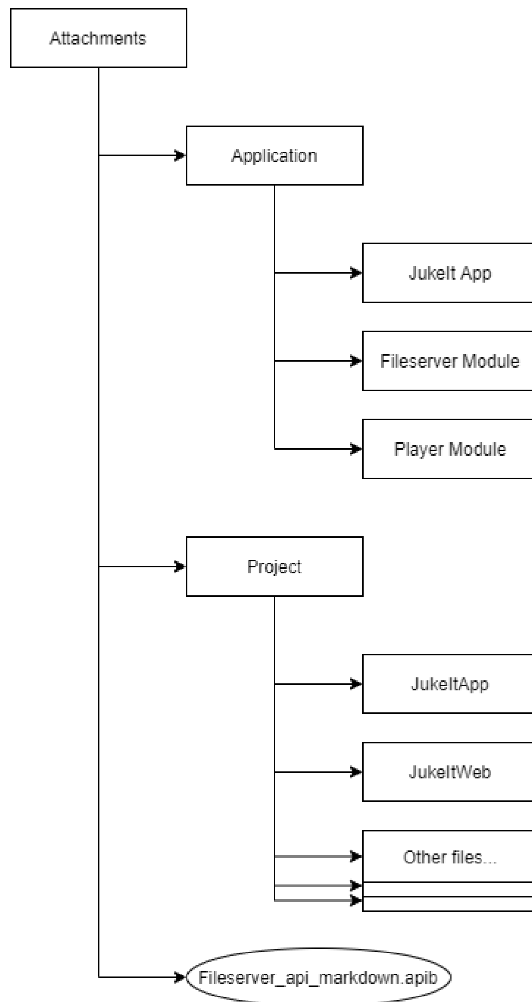
B.3 Building and Using Web Application

Desktop application uses web application as GUI. In Debug configuration, application accesses `localhost:3000` and supports hot reloading, i.e. on code changes in web application code it is reloaded without the need for rebuilding/reloading. In Release configuration it accesses built web application in *app* folder.

1. From command line, go to *JukeItWeb* directory
   - `cd path/to/project/JukeItWeb`
2. First time, you will need to install packages - run `npm install` command
3. In Debug environment, to run hot reloading server for debugging, run `npm start`. The web application will be accessible at `localhost:3000`.
4. To build the web application in Release environment, run `npm run build`. After that, copy *build* directory into root directory of built *Release* version of *JukeIt App* and rename it to *app*.
   - `xcopy build ..\build\JukeItApp\Release\app\ /E /Q`

C. Attachments

The attachment comprises of the following items:



- The *Application* folder, which contains built applications ready to use.
- The *Project* folder, which contains the source code of the project.
  - The *JukeItApp* folder contains C++ code for our application.
  - The *JukeItWeb* folder contains code for the web application.
- The *Fileserver_api_markdown.apib* file, which contains API Blueprint markdown documentation of Fileserver API.