VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

CHYTRÝ REPRODUKTOR S RASPBERRY PI

SMART LOUDSPEAKER WITH RASPBERRY PI

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **TOMÁŠ VONDRÁČEK**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. ADAM HEROUT, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Vondráček Tomáš**
Program: Informační technologie
Název: **Chytrý reproduktor s Raspberry Pi**
Smart Loudspeaker with Raspberry Pi
Kategorie: Uživatelská rozhraní

Zadání:

1. Seznamte se s možnostmi vývoje na Raspberry Pi, zaměřte se na síťovou komunikaci a přehrávání multimédií.
2. Prostudujte problematiku přehrávání hudby na mobilních zařízeních a jiných počítačích, zaměřte se na externí reproduktory a streamování a přenos hudby.
3. Navrhněte způsob komunikace a ovládání reproduktoru pracujícího v režimu IoT.
4. Prototypujte dílčí prvky konstruovaného řešení a vyhodnocujte jejich vlastnosti.
5. Navrhněte a implementujte prototyp IoT reproduktoru založeného na Raspberry Pi.
6. Iterativně vylepšujte vytvářené řešení.
7. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek a krátké video pro prezentování projektu.

Literatura:

- Steve Krug: Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability, ISBN-13: 978-0321965516
- Steve Krug: Rocket Surgery Made Easy: The Do-It-Yourself Guide to Finding and Fixing Usability, ISBN-13: 978-0321657299
- Susan M. Weinschenk: 100 věcí, které by měl každý designér vědět o lidech, Computer Press, Brno 2012

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3, značné rozpracování bodů 4 a 5.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Herout Adam, prof. Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 6. listopadu 2018

Abstrakt

Bakalářská práce se zabývá tvorbou hudebního systému, který umožní uživatelům organizovat a přehrávat hudbu v reálném čase. Hudební systém je založen na architektuře klient-server a skládá se ze serverové, webové a mobilní aplikace. Serverová aplikace vykonává roli přehrávače hudby a zprostředkovatele komunikace mezi klienty. Klient slouží jako uživatelské rozhraní pro server, ale umožňuje organizovat a přehrávat hudbu plně ve vlastní režii. Hudební systém je implementován v jazyce JavaScript (ECMAScript 2018). Serverová aplikace je postavena na Node.js a implementuje Socket.IO server včetně REST API. Webová aplikace je založena na knihovně React a prezentovaná jako SPA. Mobilní aplikace je implementována v React Native se zaměřením na systém Android. Serverová a webová aplikace je nasazena na serverech Heroku a počítači Raspberry Pi. Mobilní aplikace je publikována na Google Play. Hudební systém lze využít pro soukromé přehrávání hudby, nebo jako prostředek organizování hudby mezi více uživateli.

Abstract

The bachelor thesis deals with the creation of music system that allows users to organize and play music in real time. The music system is based on client-server architecture and consists of server, web and mobile application. The server application serves as a music player and a connection broker among clients. The client serves as a user interface for the server, but can organize and play music on its own. The music system is implemented in JavaScript (ECMAScript 2018). The server application is built on the Node.js and implements Socket.IO server with REST API. The web application is based on React and presented as SPA. The mobile application is implemented in React Native with focus on Android system. The server and web applications are deployed on Heroku servers and Raspberry Pi computer. Mobile application is published on Google Play. The music system can be used for private music playback or as a means of organizing music among multiple users.

Klíčová slova

React, React Native, Node.js, Socket.IO, Redux, Redux-Saga, Immutable.js, Express, Raspberry Pi, klient-server, sdílení kódu mezi platformami

Keywords

React, React Native, Node.js, Socket.IO, Redux, Redux-Saga, Immutable.js, Express, Raspberry Pi, client-server, code sharing among platforms

Citace

VONDRÁČEK, Tomáš. *Chytrý reproduktor s Raspberry Pi*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Adam Herout, Ph.D.

Chytrý reproduktor s Raspberry Pi

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Adama Herouta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Vondráček
14. května 2019

Poděkování

Velké díky patří zejména vedoucímu prof. Ing. Adamu Heroutovi, Ph.D. za řízení směru vývoje aplikace po celý zimní i letní semestr.

Obsah

1 Úvod **3**

2 Vývoj aplikací v jazyce JavaScript **4**
2.1 Monolitické repozitáře 4
2.2 Sdílení kódu mezi platformami 5
2.3 Redux 6
2.4 Neměnná data 8
2.5 Normalizace dat 9
2.6 Redux-Saga 10
2.7 React 11
2.8 React Native 13
2.9 Raspberry Pi 14
2.10 Obousměrná komunikace mezi zařízeními 15

3 Internet věcí a komunikace zařízení v rámci lokální sítě **17**
3.1 Zařízení se statickou IP adresou 17
3.2 Přidělení doménového jména 18

4 Analýza existujících aplikací **21**
4.1 Mubo 21
4.2 Volumio 22
4.3 Závěr 23

5 Návrh systému **24**
5.1 Funkční požadavky na systém 24
5.2 Uživatelské rozhraní 25
5.3 Stav aplikace 30

6 Implementace systému **34**
6.1 Struktura repozitáře 34
6.2 Společné aspekty vývoje webové a mobilní aplikace 35
6.3 Navigace v aplikacích 39
6.4 Uživatelské rozhraní 40
6.5 Přehrávání hudby 41
6.6 Sdílení kódu a multiplatformní komponenty 44
6.7 Nalezení serveru na lokální síti 46
6.8 Typy spojení mezi klientem a serverem 47
6.9 Real-time komunikace 50

6.10 Přenos hudby na server 52

7 Testování a nasazení systému **55**
7.1 Webová a serverová aplikace 55
7.2 Mobilní aplikace 56
7.3 Raspberry Pi 57

8 Závěr **59**

Literatura **60**

A Obsah přiloženého DVD **62**

B Instalační manuál **63**
B.1 Společná instalace pro všechny aplikace 63
B.2 Webová aplikace 63
B.3 Mobilní aplikace 63
B.4 Serverová aplikace 64

C Překlad a spuštění aplikací **65**
C.1 Serverová a webová aplikace 65
C.2 Mobilní aplikace 65

D Plakát **67**

Kapitola 1

Úvod

Práce popisuje návrh a proces tvorby hudebního systému. Součástí každého hudebního systému je reproduktor, který tato práce realizuje prostřednictvím serverové aplikace ve spojení s počítačem Raspberry Pi. Ovládání hudebního systému lze realizovat klientem, který je ve formě webové a mobilní aplikace. Uživatelé libovolného klienta mohou ovládat server a komunikovat s ostatními uživateli. Komunikace mezi klienty zahrnuje sdílení playlistů, fronty a skladeb.

Existující řešení tohoto problému nabízí server sloužící jako přehrávač, který lze ovládat klienty. Cílem této práce je vytvořit server, který lze použít i jinak než pouze jako přehrávač, a klienty, kteří neslouží pouze pro ovládání serveru, ale i jako přehrávače. Pokud server nebude z nějakého důvodu schopný přehrát hudbu, bude stále fungovat jako zprostředkovatel spojení mezi klienty. Klienti připojení k takovému serveru budou stále schopni komunikovat mezi sebou.

Hudební systém je kompletně naprogramován v jazyce JavaScript, což nabízí možnost sdílení kódu mezi všemi aplikacemi. Způsoby sdílní kódu a organizace repozitáře jsou popsány zpočátku druhé kapitoly spolu se základním výčtem klíčových technologií pro tuto práci. Třetí kapitola se zabývá způsoby nalezení serveru na lokální síti. Čtvrtá kapitola popisuje podobná řešení, která dovolují sdílet a organizovat hudbu mezi uživateli. Pátá kapitola specifikuje požadavky na aplikace, navrhuje uživatelské rozhraní a stav aplikace. Šestá kapitola popisuje konkrétní specifika implementace hudebního systému. Zabývá se zejména architekturou aplikace, přehráváním hudby a komunikací mezi aplikacemi. Poslední kapitola popisuje testování a nasazení systému do produkční verze.

Kapitola 2

Vývoj aplikací v jazyce JavaScript

JavaScript je programovací jazyk, který vznikl již před 29 lety. Z počátku byl využíván velmi zřídka, protože jeho specifikace nebyla řádně implementována napříč webovými prohlížeči. Až v roce 2009 s příchodem nové specifikace ECMAScript 5, dostal JavaScript větší dávku pozornosti od vývojářů a webové aplikace tak začaly více implementovat interaktivním prvky. V dnešní době lze JavaScript použít pro tvorbu nejen webových aplikací, ale také aplikací, které běží na mobilních zařízeních nebo počítačích.

Multiplatformní vlastnosti toho jazyka tak umožňují použít stejný kód, který poběží na více platformách a který bude na vykonávat stejnou věc. Aspekty vývoje multiplatformních aplikací jsou popsány v kapitole 2.1 a 2.2. Dále je popsána tvorba multiplatformních aplikací pomocí konkrétních knihoven v jazyce JavaScript.

2.1 Monolitické repozitáře

Monolitické repozitáře (dále jen „monorepo“) [11] jsou zejména definovány v kontextu verzovacích systémů. Monorepo pak obsahuje kód, který by jinak byl organizován ve více repozitářích. Tato praktika je používaná velkými firmami jako je Google, Microsoft či Facebook.

Monorepo má význam ve vývoji aplikací, které jsou určené pro více platforem a které sdílí kód. Avšak více typickým příkladem jednoduchého monorepa je aplikace, která obsahuje backend i frontend v rámci jednoho repozitáře. Monorepo může být spravováno pomocí nástrojů jako jsou Lerna[1] nebo Bit[2]. Monorepo může být velice výhodné, ale také v některých případech svazující. Typické monorepo má následující vlastnosti:

Výhody:

- **Jednodušší organizace** – Mít veškerý kód na jednom místě je nejen lepší pro navigaci a případné porozumění problému, ale také má velmi kladný vliv na udržení konzistence struktury napříč jednotlivými projekty.
- **Změny ve více projektech** – Taková změna by znamenala v každém projektu udělat commit[3], který by je nutné vzájemně prolinkovat. Monorepo umožňuje udělat veškeré změny v jednom commitu, který je pak daleko snáze dohledatelný.

[1] Lerna – https://github.com/lerna/lerna
[2] Bit – https://github.com/teambit/bit
[3] Git commit – https://git-scm.com/docs/git-commit

- **Sdílení kódu** – Kód, který je pro více projektů stejný, stačí extrahovat do sdíleného adresáře. Techniky sdílení kódu jsou více rozebrány v kapitole 2.2.

Nevýhody:

- **CI/CD**[4] – Pro složitější monorepo je obtížnější správné nastavení.
- **Udělování přístupu** – Udělování přístupu nemusí být problém pro menší projekt, kdy na projektu pracuje uzavřena skupina lidí. V případě projektu, na kterém se podílí velká skupina lidí, nemusí být vždy žádané, aby všichni vývojáři měli přístup k celému repozitáři.

2.2 Sdílení kódu mezi platformami

U projektů, ve kterých je sdílená značná část funkcionality mezi všemi platformami, jistě není žádoucí psát stejnou funkcionalitu pro každou platformu zvlášť. Jako příklad sdílené funkcionality si lze představit scénář, ve kterém uživatel zahájí proces výběru další skladby kliknutím na tlačítko. Po vyhodnocení nadcházející skladby se musí aktualizovat stav aplikace a předat informace všem připojeným klientům. Tento proces musí být realizovatelný nejen v mobilní a webové aplikaci, ale také z pozice serveru, na které tento proces může začít v momentě, kdy aktuálně přehrávaná skladba skončí.

Myšlenka sdílení kódu je jistě velice atraktivní, protože zefektivní nejen programování, ale i opravu chyb v aplikacích. Avšak je nutné přemýšlet nad způsobem organizace takového kódu a také nad tím, který kód je dostatečně abstraktní na to, aby byl vhodný pro sdílení.

Organizace sdíleného kódu:

- **npm balíček**[5] – Takový balíček poté má jasně definované API a nabízí kvalitní prostředí pro jeho distribuci.
- **Adresář v projektu** – Není nutné specifikovat API. Současně tento způsob zapadá do konceptu monorepo, protože je vše soustředěno na jednom místě.

Abstraktní kód, který je vhodný pro sdílení:

- **Knihovna Redux** – Respektive její části, ze kterých je formovaná. Tyto části je vždy dobré mít natolik abstraktní, aby jejich použití dávalo smysl na všech platformách. Konkrétně se jedná o části: `reducers`, `actions`, `selectors`, `middlewares` a případně `containers`. Více v kapitole 2.3.
- **Knihovna Redux-Saga** – Nabízející generátor funkce, které reagují na změny stavu aplikace a pokud neobsahují platformně specifický kód, jako třeba zápis do paměti prohlížeče či telefonu, je možné je bez problému sdílet napříč platformami. Více v kapitole 2.6.
- **Koncept React Hooks** – Jedná se o způsob abstrakce lokálního stavu prezentačních komponent, který je více rozebrán v kapitole 2.7.

[4] CI/CD – https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

[5] npm – https://docs.npmjs.com/packages-and-modules

Platformně specifický kód, který není vhodné sdílet:

- **Prezentační komponenty** – Prezentační komponenty jednotlivých platforem jsou příliš odlišné ve svojí deklaraci i stylování, aby mohly být pokládány za dostatečně abstraktní.
- **Stav aplikace** – Vytvořením instance stavu aplikace ve sdílené složce je odebrána možnost přidávání platformně specifických podstavů. Přesto, že instance tedy musí být vytvořena na každé platformně zvlášť, je požadováno, aby webová i mobilní aplikace implementovala stejný postup vytvoření instance stavu.

2.3 Redux

Redux je knihovna, která se stará o stav aplikace [16]. Redux byl navržen pro knihovnou React[6], ale lze použít i s ostatními knihovnami jako jsou Vue.js[7], Angular[8] nebo React Native[9]. Redux je založený na 3 principech:

- **Jeden zdroj pravdy** – Stav celé aplikace je uložen ve stromu objektů v rámci jednoho skladu, nazývaný *store*. Z pohledu implementace se jedná o jeden objekt, který obsahuje další objekty reprezentující jednotlivé části stavu aplikace.
- **Stav je pouze ke čtení** – Jediný způsob, jak změnit stav aplikace, je zavoláním metody `dispatch` s parametrem akce, která popisuje, jaká změna stavu bude provedena.
- **Změny jsou provedeny *„čistou funkcí“***[10] – Akce, která definuje změnu, je dále předána do funkce zvané `reducer`, která provede konkrétní transformaci stavu.

Redux v částech

Actions a Actions Creators

Akce (Actions), které jsou reprezentovány objekty, jsou častěji vytvářeny funkcemi (Actions Creators). Tyto funkce umožní parametrizaci akcí, které poté kromě jejich typu, nesou i libovolná data, použitá pro následující transformaci stavu. Akce však nepopisují proces transformace stavu, od toho jsou reducers funkce, které pracují s daty, které jim akce poskytnou.

```
const TRACKS_SAVE = 'TRACKS_SAVE'; // Action type

const tracksSave = (tracks) => ({  // Action creator which creates Action
    type: TRACKS_SAVE,
    tracks,
});
```

[6] React – https://reactjs.org
[7] Vue.js – https://vuejs.org
[8] Angular – https://angular.io
[9] React Native – https://facebook.github.io/react-native
[10] Čistá funkce (Pure function) – Funkce, jejíž návratová hodnota je ovlivněna pouze jejímy vstupy. Tato funkce zároveň nemá žádné vedlejší účinky.

Reducers

Reducer je opět čistá funkce, která provede konkrétní transformaci stavu popsanou danou akcí. Transformací stavu se rozumí vytvoření nového stavu, který se poté transformuje a následně nahradí stav předešlý. V následujícím kódu je ukázku jednoduchého konceptu reducer funkce, který pracuje s čistými JavaScriptovými objekty.

```
const tracksReducer = (state, action) => {
    switch(action.type){
        case: 'SAVE_TRACKS':
            return {...state, tracks: action.tracks}
        default:
            return state;
    }
}
```

Práce s čistými JavaScriptovými objekty nemusí být ve více zanořené struktuře čitelná. Proto se používá technika `combineReducers`[11], která nabízí efektivní strukturování. Dále se používá koncept neměnných dat (Immutable data), který je dále popsaný v kapitole 2.4. Reducer funkce používající neměnná data vypadá následovně:

```
const tracksReducer = (state, action) => {
    switch(action.type){
        case: 'SAVE_TRACKS':
            return state.set('tracks', action.tracks)
        default:
            return state;
    }
}
```

Naneštěstí funkce `combineReducers`, kterou Redux implementuje, neumí pracovat s neměnnými daty, a proto je nutné instalovat balíček `redux-immutable`[12], který exportuje stejnojmennou funkci `combineReducers`.

Store

Je objekt, ve kterém je uložen stav celé aplikace. Store definuje několik funkcí pro manipulaci s jeho stavem. Získání aktuálního stavu se provádí funkcí `getState()` a jeho změna pomocí funkce `dispatch(Action)`. Pokud se aplikace dostane do stavu, ve kterém havaruje, je možné ji obnovit do bodu, ve kterém se dříve nacházela. Aby tato obnova byla možná, je nutné mít stav aplikace serializovatelný. Serializovatelný stav tedy nemůže obsahovat neserializovatelná data, kterými jsou například funkce nebo třídy.

Middleware

Middleware je pokročilejší koncept knihovny Redux. Dříve než akci dostane cílový reducer, projde tato akce skrze registrovaný middleware. Reakce middlewaru na akci tedy může být její modifikace, zrušení, zavolání více akcí, logování nebo například navázání spojení se serverem [16]. Příkladem knihovny, která používá Redux middleware je Redux-Saga[13], která je popsána v kapitole 2.6.

[11] combineReducers – https://redux.js.org/recipes/structuring-reducers
[12] redux-immutable – https://github.com/gajus/redux-immutable
[13] Redux-Saga – https://redux-saga.js.org

Selectors

Selectors jsou funkce, které definují API pro přístup ke stavu aplikace. Tyto funkce si lze představit jako příkazy `SELECT` v jazyce SQL. Jedná se tedy o funkce, které pouze čtou části stavu. Části stavu jsou většinou dále zpracovávány a některé operace mohou být výpočetně náročné. V případě, že existuje více funkcí, které provádějí často výpočetně náročné operace, je nutné tyto funkce optimalizovat takovým způsobem, aby se jejich zpracování omezilo pouze na případy, kdy je jejich zpracování opravdu nezbytné. Problematika napojení selector funkcí na Redux, včetně jejich optimalizace, je popsána v kapitole 2.7.

Kontejnerové komponenty

Kontejnerové komponenty jsou speciálním případem komponent, které přímo komunikují se stavem aplikace a předávají potřebná data prezentačním komponentám, které jsou tak odstíněny od implementačních detailů konkrétní aplikace. Vytvoření kontejnerových komponent a jejich napojení na Redux je popsáno v kapitole 2.7.

2.4 Neměnná data

Struktura, která je tvořena pomocí neměnných (immutable) dat, nemůže být již po vytvoření změněna. Každá změna takové struktury vytvoří její kopii, na které jsou změny provedeny. Současně je původní struktura zachována beze změny. Struktura vykazující takové chování, je nazývaná jako o perzistentní datová struktura. [5]

Immutable.js

Immutable.js[14] je knihovna, které implementuje perzistentní datové struktury v jazyce JavaScript. Tato knihovna pochází od společnosti Facebook Inc.

Mezi základní kolekce této knihovny patří:

- **List** – Uspořádaná kolekce, která je podobná implementaci pole v jazyce JavaScript.
- **Map** – Neuspořádaná kolekce, která je podobná implementaci objektu v jazyce JavaScript.
- **Record** – Speciální případ implementace Mapy, u které je nutné při instancianci specifikovat strukturu dat, která je závazná po celou dobu života dané kolekce.

Použití knihovny jako je Immutable.js přináší jednotný způsob práce s daty, který je řízen plně v rámci API dané knihovny. Tento přístup je mnohem efektivnější a méně náchylnější na chyby, než-li specifikace konvence, kterou by programátoři museli znát a dodržovat pro dodržení principu neměnných dat. Immutable.js nabízí velice přímočaré API pro práci se zanořenými strukturami pomocí funkcí `getIn`, `setIn`, `mergeIn`, `updateIn` nebo `removeIn`. Optimalizace, které Immutable.js přináší s knihovnou Redux jsou popsané v knize The Complete Redux Book [16].

[14] Immutable.js – https://immutable-js.github.io/immutable-js

2.5 Normalizace dat

Normalizace dat z pohledu webových aplikací se velmi podobá normalizaci ve světě databází. V obou případech je snaha o:

- **Odstranění duplikace dat** – Za účelem zvýšení konzistence systému, kdy se změna provede pouze na jednom místě a všude jinde je pak tato změna automaticky reflektována.
- **Izolování závislosti dat** – Data jsou strukturována tak, že každý logický celek je extrahován do vlastní „tabulky“ – v případě webu se tedy jedná o objekt.
- **Ukládání pouze nezbytných dat** – Veškeré hodnoty, které lze spočítat, by neměly být uloženy v databázi.

Nenormalizovaný stav:

```
const state = {
    playlists: [
        {
            id: 'pl1',
            playlistName: 'laylist 1',
            tracks: [{id: 'tr1', url: 'url1'}, {id: 'tr2', url: 'url2'}, ...]
        },
        ...
    ]
}
```

Příznaky špatně navrženého stavu:

- **Výběr playlistu podle id** – V nejhorším případě to znamená průchod celého pole playlistů.
- **Vytvoření záznamu o tracku, který není v playlistu** – Nelze.
- **Editace tracku** – Nutná ve všech playlistech, kde se takový track vyskytuje.

Normalizovaný stav:

S takto normalizovaným stavem se lépe pracuje a přístupové doby k zanořeným strukturám jsou kratší. Tím, že objekt je neuspořádaná kolekce, která nezaručuje pořadí, je pořadí možné zařídit buď přidáním pole `order`, jak je uvedeno v případě objektu `playlists`, nebo přidáním atributu `createdAt`, jak je ukázáno v případě objektu `tracks`:

```
const state = {
    playlists: {
        pl1: {
            id: 'pl1',
            name: 'playlist 1',
            tracks: ['tr1', 'tr2', ...]
        },
        ...,
        order: ['pl1', ...],
    }
```

```
    tracks: {
        tr1: {
            id: 'tr1',
            url: 'url1',
            createdAt: 1554626935401, // Unix time
        },
        tr2: {
            id: 'tr2',
            url: 'url2',
            createdAt: 1554626950801, // Unix time
        },
        ...
    }
}
```

2.6 Redux-Saga

Redux-Saga [24] je knihovna, která si dává za cíl správu vedlejších efektů aplikace (asynchronní operace, přístup do paměti prohlížeče) zjednodušit pro vývojáře. Současně jsou tyto vedlejší efekty testovatelné a efektivní na vykonání. Knihovna je implementována jako middleware pro knihovnu Redux. Využívá přitom funkce typu generátor[15], které jsou specifikované v ES6[16].

Invokace ságy:

- **Uživatelem** – Taková sága je volána pomocí blokujícího volání `call` a `apply`, nebo neblokujících `fork` a `spawn`.
- **Jako reakce na akci** – Reakce lze implementovat pomocí blokujícího volání `take` a `takeMaybe`, nebo neblokujících `takeEvery`, `takeLeading` a `takeLatest`.

Konstrukce, které jsou definovány jako blokující pouze blokují (čekají) v kontextu dané ságy, nikoliv v rámci aplikace. Nedochází tak k efektu „zamrznutí“ aplikace.

Napojení na externí zdroje

Pokud aplikace vyžaduje práci s externími zdroji například ve formě protokolu WebSocket (kapitola 2.10), knihovna nabízí funkci `eventChannel`.

eventChannel

`eventChannel` je „Higher Order function“[17]. První argument této funkce je tedy funkce, která zpracovává asynchronní zprávy přijaté z externího zdroje. Tyto zprávy jsou následně zaslány volajícímu, který na ně čeká pomocí blokující funkce `take`. Volající má zodpovědnost za zpracování či zastavení příjmu zpráv, které `eventChannel` vytváří. Zavření kanálu se může realizovat i přímo uvnitř `eventChannel`, a to zasláním zprávy, jež obsahuje konstantu `END`, kterou zpřístupňuje Redux-Saga. Zavřením kanálu se zavolá funkce, kterou

[15] Generátor – http://es6-features.org/#GeneratorFunctionIteratorProtocol
[16] ECMAScript 6 – https://www.ecma-international.org/ecma-262/6.0
[17] Higher Order function – funkce, která přijímá funkci jako argument nebo ji vrací.

`eventChannel` vrací. Této funkci se říká „unsubscribe function“, která by měla ukončit komunikaci s externím zdrojem, aby nedocházelo k únikům paměti (memory leak).

Napojení na externí zdroje tedy řeší případy, kdy aplikace potřebuje přijímat a zpracovávat zprávy, které jsou zaslány v rámci `callback` funkcí. Příkladem takového externího zdroje může být WebSocket spojení, nebo funkce `setInterval`.

2.7 React

React [3] je knihovna pro tvorbu znovupoužitelných komponent, které slouží jako základní stavební bloky celého uživatelského rozhraní. React pochází od společnosti Facebook Inc. a sám o sobě není dostačující nástroj pro tvorbu celé aplikace jako třeba framework Angular[18]. React se pouze soustředí na tvorbu komponent a problémy týkající se navigace nebo správy stavu aplikace nechává na programátorovi aplikace.

Komponenty

Každá komponenta reprezentuje část aplikace. Příkladem komponenty může být menu, které jistě obsahuje i položky (komponenty). Komponenty, které React vytváří, lze také nalézt pojmenované jako prezentační komponenty. Toto označení se často používá v kontextu s kontejnerovými komponentami pro odlišení významu daných komponent. Slovo komponenta v kontextu knihovny React je v následujícím textu myšlena vždy jako prezentační komponenta.

Syntaxe, kterými jsou komponenty tvořeny, je nazvána JSX. JSX je v podstatě syntaxe podobná XML/HTML. Nejpoužívanějšími atributy jsou `onClick`, který lze v JavaScriptu implementovat jako `element.addEventListener('click', callback)` a `className`, který lze implementovat jako `element.className` či `element.classList.add('cls')`. Komponenty tvořené JSX zápisem prohlížeč nedokáže interpretovat, a proto je nutné použít nástroj jako je Webpack[19] nebo Gulp[20], který zjednoduší transpilaci jednotlivých komponent do podoby, která je interpretovatelná prohlížeči.

Bezstavové komponenty

Struktura bezstavové komponenty je vždy plně závislá na argumentech, které jsou definovány uživatelem při jejím použití. Příklad jednoduché bezstavové komponenty je následující:

```
const Track = ({name, isPlaying}) => {              // API
    const state = isPlaying ? 'playing': 'paused';  // Business logic

    return (
        <div>{name} -- is {state}</div>             // Component structure
    )
};
```

Stavové komponenty

Struktura stavové komponenty není plně závislá pouze na argumentech, protože může být změněna i jako výsledek akce provedené uživatelem v rámci dané komponenty. Typickým

[18] Anuglar – https://angular.io
[19] Webpack – https://webpack.js.org
[20] Gulp – https://gulpjs.com

příkladem takové komponenty může být rozbalovací menu, které při kliknutí na nadřazený element vykreslí celou nabídku.

Implementace stavové komponenty byla před verzí `16.8.0`[21] možná pouze vytvořením třídy, která dědila ze třídy `React.Component` a svůj stav inicializovala pomocí konstruktoru s třídní proměnnou `this.state`. Od této verze je dostupné React Hooks API, které umožňuje psát stavové komponenty jako funkce namísto tříd. Takové funkce inicializují svůj stav funkcemi `React.useState`. Příklad jednoduché stavové komponenty využívající React Hooks je následující:

```
const Track = ({name, isPlaying}) => {
    const [playing, toggle] = React.useState(isPlaying);
    const state = playing ? 'playing': 'paused';

    return (
        <div onClick={() => toggle(!playing) }>{name} -- is {state}</div>
    )
};
```

V případě, že je nutné, aby více komponent sdílelo stejnou logiku, pak je možné abstrakci provést následovně:

```
const usePlay = ({isPlaying}) => {
    const [playing, toggle] = React.useState(isPlaying);
    const state = playing ? 'playing': 'paused';
    const toggle = () => toggle(!playing);

    return [state, toggle]
};

const Track = ({name, isPlaying}) => {
    const [state, toggle] = usePlay(isPlaying);

    return (
        <div onClick={toggle}>{name} -- is {state}</div>
    )
};
```

Funkce `usePlay`, která pracuje se stavem komponenty, je nyní použitelná v jakékoliv komponentě se stejným výsledkem. Tím bylo dosaženo abstrakce stavové logiky, kterou lze lépe testovat a sdílet mezi komponentami, což před příchodem React Hooks nebylo možné. Další zajímavé vzory, které nabízí zvýšení úrovně abstrakce, jsou Render Props[22] a HOC[23].

React a Redux

Udržování lokálního stavu v prezentačních komponentách je dostačující, není-li třeba tento stav reflektovat mezi více prezentačními komponentami. Pokud se jedná o menší projekt, React nabízí `Context API`[24], avšak je-li projekt rozsáhlejší, je použití knihovny Redux výhodnější, protože lze stav efektivněji strukturovat. O propojení mezi knihovnami React a Redux se stará další knihovna, která se jmenuje `React Redux`[25]. Tato knihovna, kromě

[21] React changelog – https://github.com/facebook/react/blob/master/CHANGELOG.md
[22] Render Props, více viz https://reactjs.org/docs/render-props.html
[23] HOC – High-Order Components, více viz https://reactjs.org/docs/higher-order-components.html
[24] Context API – https://reactjs.org/docs/context.html
[25] React Redux – https://react-redux.js.org

propojení, implementuje optimalizační techniky, které zabraňují zbytečnému vykreslování komponent. Propojení knihoven je realizováno pomocí API, které má základní 2 části:

- **Provider** – Komponenta, která umožní napojit všechny její potomky na Redux sklad.
- **connect** – Je Higher Order function, která vytváří kontejnerové komponenty z prezentačních komponent tím, že jim zpřístupní komunikaci se skladem.

Příklad použití funkce `connect`:

```
// Presentational Component
const Track = ({isPlaying}) => isPlaying ? 'playing' : 'paused';

const mapStateToProps = (state) => (
    // Simple selector
    isPlaying: selectPlayerIsPlaying(state),
    // Composed selector, which derives data from different parts of state
    similarArtists: selectSimilarArtists(selectCurrentTrack(state)),
)
// Container Component
const ConnectedTrack = connect(mapStateToProps)(Track);
```

Na této ukázce lze, kromě vytvoření kontejnerové komponenty, pozorovat také selektor `selectSimilarArtists`, který by bylo vhodné vytvořit pomocí knihovny Reselect[26]. V tomto příkladu jsou oba selektory volány při změně proměnné `isPlaying`. Avšak nedává příliš smysl provádět náročné operace pomocí selektoru `selectSimilarArtists` pro stále stejnou skladbu, protože výsledek bude vždy stejný. Knihovna Reselect v tomto případě zabezpečí, že při stejné skladbě jednoduše vrátí předešlý výsledek. Tato optimalizační technika se nazývá memoization[27].

2.8 React Native

React Native [13] je framework, který nabízí vývoj skutečně nativních mobilních aplikací pomocí JavaScriptu. React Native je založen na knihovně React, ze které využívá paradigma pro tvorbu komponent. Stejně jako React pochází od společnosti Facebook Inc.

React Native, narozdíl od knihovny React, neumožňuje použití elementů jako je `div` nebo `span`. Základními stavebními bloky jsou zde již vytvořené komponenty, které jsou definovány v knihovně. Tyto komponenty se na rozdíl od konkurenčních řešení, jako jsou Ionic[28], PhoneGap[29] nebo Cordova[30], překládají do nativních komponent dané platformy a nejsou tak pouze emulovány.

V době psaní této práce je React Native ve verzi `0.59.5`. Nové minoritní vydání knihovny často představuje nekompatibilní změny a označení komponent jako zastaralých. Přechod na novou verzi tedy znamená zkontrolovat changelog[31] a v případě nekompatibilních změn je nutné zkontrolovat i použité knihovny v projektu, kterých by se tato změna mohla dotýkat.

[26] Reselect – https://github.com/reduxjs/reselect
[27] memoization –https://www.geeksforgeeks.org/memoization-1d-2d-and-3d
[28] Ionic – https://ionicframework.com
[29] PhoneGap – https://phonegap.com
[30] Cordova – https://cordova.apache.org
[31] Changelog – https://github.com/react-native-community/releases/blob/master/CHANGELOG.md

Vývoj aplikace pro:

- **Android** – Může být vyvíjena nezávisle na operačním systému.
- **iOS** – Pro vývoj je nutné prostředí Xcode[32], které běží pouze na operačním systému macOS.

Princip fungování

React Native funguje na principu propojení vlákna naší aplikace, které spouští kód napsaný v JavaScriptu s nativním vláknem dané platformy. O propojení těchto vláken se stará bridge „dále jen most“. Most umožňuje obousměrnou asynchronní komunikaci mezi oběma vlákny. Komunikace probíhá na základě fronty zpráv, které jsou serializované ve formátu JSON. Most tyto zprávy následně deleguje cílovému vláknu, které je dále zpracovává. Bridge je implementován v jazyce C/C++ a pracuje s JavaScriptovým jádrem dané platformy, které dovoluje spouštět JavaScript kód v jazycích jako jsou C, Objective-C nebo Java. [15]

React native aktuálně používá dvě různé JavaScript jádra, kdy jejich použití se liší na základě prostředí, ve kterém se aplikace právě nachází. Pokud se aplikace nachází v debug režimu v prohlížeči Chrome, pak React Native požívá V8[33], jinak se používá JavaScript-Core[34]. [23]

2.9 Raspberry Pi

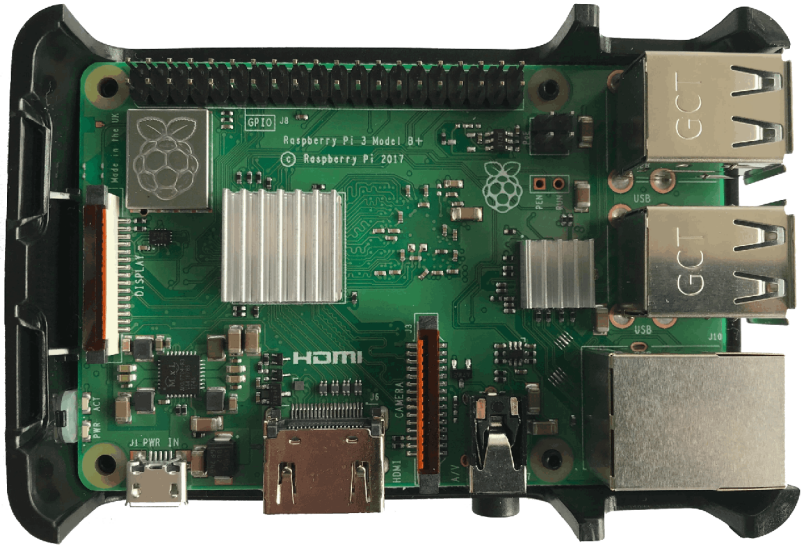
Raspberry Pi [22] je malý počítač o velikosti kreditní karty, který se pyšní nízkou pořizovací cenou a značnou popularitou. Nový počítač nemá nainstalovaný žádný operační systém. Oficiální operační systém pro všechny modely Raspberry Pi je Raspbian[35], avšak existují i alternativy jako třeba Windows 10 IoT Core [36] nebo Ubuntu Mate[37]. Instalace systému lze snadno provést pomocí nástroje Etcher[38]. Připojení k internetu je možné pomocí Wi-Fi nebo Ethernetu.

Raspberry pi nabízí velkou řadu modelů, ze které je nejvýkonnější model `Raspberry Pi 3 Model B+`, který je na obrázku 7.1. Mezi jeho důležité parametry patří:

- **Architektura** – ARMv8-A (64bit)
- **Procesor** – 1.4 GHz 64-bit quad-core ARM Cortex-A53
- **Operační pamět** – 1 GB

Raspberry Pi umožňuje vývoj aplikací v nejznámějších programovacích v jazycích, jako jsou Python, Java, JavaScript, C nebo C++. Pro tuto práci je však důležitý programovací jazyk JavaScript, který na této platformně funguje díky běhovému prostředí Node.js[39].

[32] Xcode – https://developer.apple.com/xcode
[33] V8 – https://v8.dev
[34] JavaScriptCore – https://developer.apple.com/documentation/javascriptcore
[35] Raspbian – https://www.raspberrypi.org/downloads/raspbian
[36] Windows 10 IoT Core – https://docs.microsoft.com/en-us/windows/iot-core/downloads
[37] Ubuntu Mate – https://ubuntu-mate.org/raspberry-pi
[38] Etcher – https://www.balena.io/etcher
[39] Node.js – https://nodejs.org/en



Obrázek 2.1: Raspberry Pi 3 Model B+ s přidaným pasivním chlazením umístěný v oficiální Raspberry Pi krabičce.

2.10 Obousměrná komunikace mezi zařízeními

Obousměrná komunikace je druh komunikace, při které spolu klient a server asynchronně komunikují. Iniciátorem komunikace může být jak server tak i klient, přičemž protistrana na danou zprávu může zareagovat odpovědí. Tento druh komunikace je tedy velmi důležitý pro efektivní zajištění zasílání zpráv klientům z pozice serveru.

Před příchodem protokolu WebSocket [14] byla obousměrná komunikace implementována pomocí techniky HTTP polling [20], která nejen zbytečně zatěžuje server a čerpá jeho zdroje, ale také nutí klienta tvořit kolerace mezi odchozími a příchozími spojeními.

V době psání této bakalářské práce je podpora protokolu WebSocket globálně u uživatelů na 95.7% a v České republice dokonce na 95.8% [12].

Protokol WebSocket

Protokol WebSocket [14] je navržen tak, aby umožnil obousměrnou komunikaci ve verzi protokolu HTTP/1.1, a to na standardních portech 80 a 443. Protokol WebSocket funguje na transportní vrstvě pomocí protokolu TCP. Protokol HTTP zde figuruje pouze ve fázi navázání spojení, která se nazývá jako fáze handshake. V rámci provedení fáze handshake klient pošle GET request, ve kterém je specifikován nový typ spojení. Handshake, který je tvořen touto webovou aplikací pomocí knihovny Socket.IO vypadá následovně:

```
GET ws://player-be.herokuapp.com/socket.io/?transport=websocket HTTP/1.1
Connection: Upgrade
Upgrade: websocket
Host: player-be.herokuapp.com
Origin: http://player-fe.herokuapp.com
Sec-WebSocket-Key: uYbaFqVII7Aqp9FDc9Trkw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Odpověď ze serveru, která indikuje úspěšné navázané spojení:

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: oBJnknkm0d1Vu7hzKUIa7QkgljY=
```

HTTP/2

Protokol HTTP/2 [4], který byl specifikován v roce 2015, je další verzí protokolu HTTP, která nabízí kompresi hlaviček a souběžnou výměnu dat ve formě streamu využívající pouze jedno navázané spojení mezi klientem a serverem, této technice se říká multiplexing. Ovšem tato verze protokolu nikterak nespecifikuje chování Websocket Protokolu.

Protokol WebSocket spolu s protokolem HTTP/2 byl specifikován až v roce 2018 [21]. Přičemž klient pro navázání spojení pošle HTTP CONNECT, který definuje nový atribut `:protocol` s hodnotou `websocket`. Otevřené WebSocket spojení již není realizováno nezávisle na serveru, ale je spravováno protokolem HTTP/2 jako stream v rámci jednoho spojení technikou multiplexing.

Socket.IO

Socket.IO[40] [2] je knihovna, která zprostředkovává v reálném čase obousměrnou komunikaci mezi klientem a serverem. Implementace komunikace je založená na registrování callback funkcí k vysílaným událostem.

Jádrem Socket.IO knihovny je knihovna Engine.IO[41], která jako první používá techniku HTTP polling, která sice není optimální, ale je spolehlivější, a pokud je to technicky možné, je využito protokolu WebSocket. Socket.IO dovoluje tuto fázi přeskočit a rovnou začít používat WebSocket protokol. Mezi další důležité funkce knihovny patří:

- **Automatické opětovné připojení** – Pokud spojení selže, je automaticky obnoveno, je-li to možné.
- **Multiplexing** – V rámci jednoho spojení lze vytvořit další jmenné prostory, ve kterých lze komunikovat.
- **Místnosti** – Jmenné prostory lze dále strukturovat do místností.

[40] Socket.IO – https://socket.io
[41] Engine.IO – https://github.com/socketio/engine.io

Kapitola 3

Internet věcí a komunikace zařízení v rámci lokální sítě

Internet věcí (Internet of Things) [18] popisuje síť fyzických zařízení, kde každé zařízení má zabudovanou elektroniku, senzory a připojení k internetu. Tyto zařízení jsou také schopny komunikovat mezi sebou nebo být ovládány člověkem.

Aby fyzické zařízení mohlo komunikovat s ostatními zařízeními, musí být adresovatelné. Adresovatelné zařízení je takové, které má přiřazenou platnou IP adresu v rámci dané sítě. IP adresa a další konfigurační parametry jsou dnes dynamicky přiřazovány protokolem DHCP. Důsledek tohoto přiřazování je takový, že se nelze spoléhat na přidělení stejné IP adresy stejnému zařízení pokaždé, když se do sítě připojí. Poté tedy vyvstává přirozená otázka, jak se k takovému zařízení připojit, když nelze s jistotou říci, jakou bude mít IP adresu. Naivní řešení, které se nabízí, je nastavit zařízení *statickou IP adresu*. Další řešení je přidělit zařízení *doménové jméno*. Obě možnosti jsou v následující kapitole rozebrány.

3.1 Zařízení se statickou IP adresou

Statická IP adresa na zařízení zaručí, že zařízení bude mít vždy stejnou IP adresu. Konfigurace statické IP adresy na zařízení je vždy prováděná vůči síti, do které je právě zařízení připojeno. Následující konfigurace statické IP adresy čerpá z knihy Raspberry Pi Cookbook [22]:

- **Nalezení rozsahu adres, které mohou být přiděleny v rámci sítě** – `ip -4 addr show | grep global`
- **Nalezení adresy směrovače nebo brány** – `ip route | grep default | awk '{print $3}'`
- **Nalezení adresy DNS serveru** – `cat /etc/resolv.conf | grep nameserver | awk '{print $2}'`

Po zjištění předchozího nastavení je nutné zjistit rozsah adres, které má DHCP server k dispozici, aby bylo možné nastavit statickou IP adresu mimo tento rozsah, avšak stále ve stejné podsíti. Pokud tohle není zabezpečeno, může dojít ke konfliktu IP adres. Zařízení, které jsou v takovém konfliktu, nemají přístup do Internetu. Poslední krok je tedy samotné nastavení zařízení v souboru `/etc/dhcpcd.conf`, kde je nutné nastavit statickou adresu IP, směrovače a DNS.

Je tedy zřejmé, že adresování zařízení pomocí statické IP adresy není jenom problematické na nastavení, ale není ani realizovatelné, pokud má být zařízení adresovatelné v rámci sítí, o kterých nic nevíme.

3.2 Přidělení doménového jména

Přidělení doménového jména nabízí možnost připojení k zařízení pomocí dobře zapamatovatelného doménového jména (např. `player.local`) namísto IP adresy. O překlad doménového jména na IP adresu se stará systém DNS. Proces překladu, který systém DNS provádí, se nazývá DNS rezoluce. Ještě před příchodem systému DNS bylo mapování doménových jmen na IP adresy definováno v souboru `/etc/hosts`[1], a protože tedy existuje více zdrojů, odkud je možné provést překlad, jsou zde soubory `/etc/host.conf`[2] a `/etc/nsswitch.conf`[3], ve kterých je toto pořadí specifikováno. Následně budou představeny techniky přiřazení doménového jména zařízení v rámci lokální sítě.

soubory `hosts`

Tato technika je založená na statickém mapování IP adresa-doménové jméno ve všech uzlech dané sítě, které chtějí doménové jméno použít. Dále je nutné nakonfigurovat statickou IP adresu na zařízení, která budou dostupná pod daným doménovým jménem.

Výhody:

- **Abstrakce IP adresy** – Není nutné překládat aplikace, které běží na zařízeních pro každou síť, kde by mohla být dostupná jiná IP adresa. Zodpovědnost, že komunikace bude fungovat, je tedy přenesena na konfiguraci sítě.
- **Jednodušší implementace** – Pro překlad doménových jmen není třeba konfigurovat DNS.

Neýhody:

- **Změna doménového jména** – V této situaci přestane celý systém fungovat do doby, než proběhne rekonfigurace ve všech sítích a jejich uzlech.
- **Konfigurace** – Konfigurace takového systému není flexibilní a je náchylná na chyby.

Závěr

Konfigurace `hosts` souborů může být vhodná, pokud se systém nachází v rámci vývoje. Pro reálné nasazení je však tato technika nedostačující.

DNS server

Systém DNS zodpovídá za chod mnoha služeb, bez kterých by Internet byl pouze těžko představitelný. Aby uživatel mohl těchto služeb využít, musí mít přiřazenou IP adresu

[1] hosts – http://man7.org/linux/man-pages/man5/hosts.5.html
[2] host.conf – http://man7.org/linux/man-pages/man5/host.conf.5.html
[3] nsswitch.conf – http://man7.org/linux/man-pages/man5/nsswitch.conf.5.html

DNS serveru, který dále využívá protokol DNS. Doménová jména, které klient nedokáže přeložit pomocí `hosts` souborů, tak putují do DNS serveru, který následně provede buďto rekurzivní nebo iterativní dotaz na další DNS servery. V případě, že je nutné mapovat doménová jména, která existují pouze v rámci lokální sítě, je nezbytné zabezpečit, aby DNS server dále nekomunikoval s ostatními servery a přímo vrátil odpověď.

Konfigurace:

- **Na směrovači** – Pokud to směrovač umožňuje, je možné přímo na něm přidat vlastní mapování doménových jmen na IP adresy.
- **Vytvořením vlastního DNS serveru** – Dnsmasq[4] je software, který nabízí možnost vytvoření vlastní síťové infrastruktury, která zahrnuje: DNS, DHCP, Router advertisement a network boot.

V obou případech je nutné zajistit, aby uzly věděly o IP adrese DNS serveru, kde mají hledat překlad doménového jména. Tato informace je buď nastavena přímo na uzlech nebo distribuována prostřednictvím DHCP.

Závěr

Možnost konfigurovat DNS server je mnohem flexibilnější než konfigurace `hosts` souborů, avšak vytvoření nebo správa takového serveru je již náročnější. Použití tohoto řešení je výhodné, pokud víme, že zařízení, na kterém poběží systém, bude pracovat pouze v rámci dané sítě a současně se do sítě často připojují/odpojují zařízení, které není možné nebo chtěné přímo konfigurovat.

Zeroconf

Zeroconf neboli Zero Configuration Networking [9] je soubor tří technik, které slouží k automatické konfiguraci sítě bez nutnosti znalosti její topologie a dalšího nastavení DHCP nebo DNS. Myšlenka Zeroconf je taková, že nastavení síťového zařízení, by nemělo být těžší než nastavení fungování nové lampičky. V obou případech by mělo stačit připojit zařízení do elektrické sítě a následné ho zapnout. Techniky, které Zeroconf používá jsou: Link-Local Addressing, Multicast DNS a DNS Service Discovery.

Link-Local Addressing

Link-Local Addressing [6] zabezpečuje adresovatelnost zařízení v rámci sítě. Jak již bylo zmíněno, tuto konfiguraci obstarává DHCP. Pokud však zařízení z nějakého důvodu nedostane informace od DHCP, je vhodné, aby takové zařízení bylo schopné si IPv4 adresu přiřadit samo. Taková IPv4 adresa je poté v rozsahu `169.254/16`.

Multicast DNS

Multicast DNS (mDNS) je protokol [8], který umožňuje provádět dotazy podobné dotazům Unicast DNS na lokální síti. Fungování mDNS není nijak ovlivněno aktuálním stavem sítě ani její infrastrukturou. Multicast DNS specifikuje novou doménu nejvyššího řádu `.local.`,

[4] Dnsmasq – http://thekelleys.org.uk/dnsmasq/doc.html

která je platná pouze v rámci lokální sítě. Veškeré dotazy na domény, které končí `.local.` musí být poslány na IPv4 adresu `224.0.0.251`, nebo IPv6 adresu `FF02::FB` protokolem UDP s cílovým portem 53.

DNS Service Discovery

DNS-SD [7] specifikuje strukturu dotazů, které je možné použít prostřednictvím standardního rozhraní DNS, pro vyhledávání služeb na lokální síti.

Podpora Zeroconf:

- **Apple** – Nevyžaduje žádnou instalaci, je implementováno pomocí Bonjour[5].
- **Windows** – Windows 10 již implementuje Zeroconf techniky. Některé programy jako Skype nebo iTunes instalují Zeroconf pro jejich fungování. Pro systémy s verzí Windows menší než 10, které nemají nainstalovaný takový program, lze podpora doinstalovat.
- **Linux včetně Raspberry Pi** – Implementováno pomocí Avahi[6]. Avahi je již ve mnoha distribucích instalován, případně lze jednoduše doinstalovat.
- **Android** – Android nativně nedokáže přeložit domény s nejvyšším řadem `.local.`, nabízí ale DNS-SD API [7], které má umožnit programátorovi vyhledat služby na lokální síti. O implementaci této funkcionality do prohlížeče již existuje od roku 2014 „issue ticket“[8].

[5] Bonjour – https://developer.apple.com/bonjour
[6] Avahi – https://www.avahi.org
[7] Android DNS-SD – https://developer.android.com/training/connect-devices-wirelessly/nsd
[8] Podpora mDNS v Chrome pro systém Android – https://bugs.chromium.org/p/chromium/issues/detail?id=405925

Kapitola 4

Analýza existujících aplikací

Analýza zahrnuje aplikace, které se slouží skupinám lidí pro organizaci a přehrání hudby. Zároveň lze aplikace použit jako zdroj inspirace pro návrh uživatelského rozhraní pro mobilní a webovou verzi. Součástí analýzy byla i anketa na sociální síti, která se zaměřila na zjištění, jestli lidé znají aplikaci, která by jim umožnila spolupracovat na tvorbě playlistů a přehrávání hudby. Závěr této ankety byl takový, že lidé znají pouze Spotify, které nabízí pouze kolaborativní playlisty, které však dále nejde přehrávat ve skupině, ale pouze soukromě.

4.1 Mubo

Mubo[1] je mobilní aplikace, která nabízí vytvoření sdílené fronty, které se říká MusicBox. Uživatelé, kteří se k MusicBoxu připojí pomocí mobilní aplikace mají 3 hlasy, které mohou rozdělit mezi jednotlivé skladby. Skladby s nejvíce hlasy se řadí na vrchol fronty a jsou tak přehrány dříve. Po určité době nazývané „cooldown“, která lze nastavit, se hlasy opět resetují a hlasování může začít znovu.

Výhody:

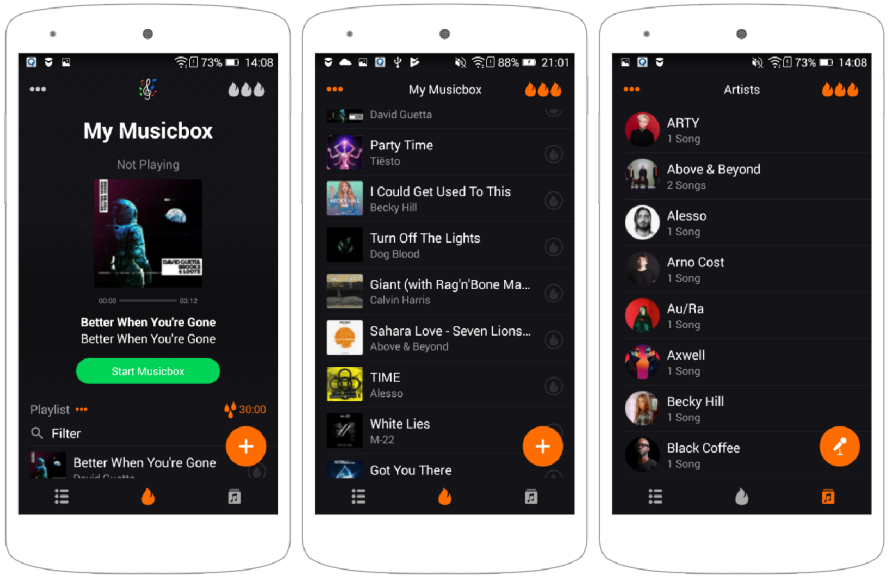
- **Propojení s aplikacemi třetích stran** – Mubo umožňuje propojení se Spotify a Apple Music. Pomocí importovaných skladeb lze tvořit frontu.
- **Hlasování** – Možnost hlasování je opravdu zajímavá a ve své podstatě je to prvek, který tvoří tuto aplikaci.

Neýhody:

- **Sdílení skladeb** – Uživatelé mohou pracovat pouze s hudbou, která je přidána tvůrcem MusicBoxu.
- **Nutnost instalace aplikace** – Pro použití MusicBoxu je nutné instalovat aplikaci.
- **Offline přehrávač** – Aplikace není dobře použitelná jako offline přehrávač.

Aplikace je vyvíjena malým týmem skládající se ze 3 lidí, a to v jejich volném čase. Mubo má stále několik nedostatků jako třeba nemožnost dát hlas poslední skladbě ve frontě, protože je v daném místě překrytá možností přidaní skladeb, jak lze pozorovat na prostředním

[1] Mubo – https://www.mubo-app.com



Obrázek 4.1: Uživatelské rozhraní aplikace Mubo. Na levém zařízení je pohled na vršek fronty, kde lze pozorovat aktuálně přehrávaná hudba. Uprostřed je pohled na frontu, kde lidé mohou vpravo hlasovat o dané skladbě. Na pravém zařízení je ukázka knihovny, která je seskupena podle umělce.

zařízení v obrázku C.1. Funkce zamíchání fronty má sice svoje tlačítko, ale zatím nefunguje. Osobně mě zarazil/zklamal odstavec při registraci MusicBoxu, který popisuje jak má vypadat silné heslo, které je akceptovatelné aplikací.

4.2 Volumio

Volumio[2] je hudební přehrávač, který lze instalovat na Raspberry Pi, Odroid, osobní počítač nebo notebook. Aplikaci je možné zadarmo stáhnout a instalovat na zmíněné platformy. Ovládání přehrávače je umožněno zadarmo pomocí webového prohlížeče nebo prostřednictvím mobilní aplikace pro systém na Android za 59.9 Kč nebo iOS za 49.0 Kč. Hodnocení aplikace na pro systém Android je 3.4/5 a pro iOS není hodnocení dostupné z důvodu nedostatečného množství hlasů.

Volumio si na platformně Raspberry Pi neuchovává žádnou hudbu v nevolatilní paměti. Z tohoto důvodu je nutné mít veškerou hudbu dostupnou z externího zařízení jako Flash disk nebo NAS[3] komunikující prostřednictvím protokolu NFS či CIFS. Volumio implementuje systém pluginů, které lze v průběhu používání instalovat a získat tak propojení například se Spotify.

Volumio nabízí cloudové služby, které umožňují kontrolu přehrávače i mimo lokální síť. Tyto služby jsou však placené podle počtu kontrolovaných zařízení, kterých může být v tento moment maximálně 6.

[2]Volumio – https://volumio.org
[3]NAS – „Network Attached Storage“ je datové úložiště dostupné na LAN síti.

Obrázek 4.2: Webové rozhraní aplikace Volumio, která je přístupné na adrese `http://volumio.local` za předpokladu, že operační systém podporuje protokoly tomu potřebné (kapitola 3.2).

4.3 Závěr

Mubo je moderní aplikace, ze které lze čerpat inspiraci na tvoření uživatelského rozhraní pro mobilni zařízení, zároveň její hlavní funkcionalita ve formě hlasování je jistě zajímavým prvkem, který by mohl v budoucnosti figurovat v podobě tzv. „Párty módu“ tohoto systému.

Volumio je již vyspělejší produkt, jehož aplikace pro Android má za sebou přes 10 000 stažení v Google Play. Vzhled uživatelského rozhraní je možné změnit pomocí obrázku, který je po celou část aplikace stejný a celkově tak spíše snižuje úroveň aplikace. Webová aplikace dokáže konfigurovat síť, budík, spánek, aktualizace a spoustu dalšího. Co však Volumio neumí, je přehrávání hudby u uživatele a upload hudby přímo z prostředí uživatelského rozhraní.

Kapitola 5

Návrh systému

Hudební systém pro organizaci a přehrávání hudby je založen na architektuře klient-server a bude se skládat z webové, mobilní a serverové aplikace. Návrh systému se sestává z bližších specifikací na systém, návrhu uživatelského rozhraní a návrhu stavu aplikace.

5.1 Funkční požadavky na systém

- **Ovládání přehrávání aktuální skladby** – Přehrávání je možné ovládat klasickými funkcemi, ke kterým patří: přehrát, zastavit, pozastavit, přeskočit na daný čas, změnit hlasitost přehrávání.
- **Nahrání hudby** – Webová i mobilní aplikace umožní nahrát skladby na server. Informace o nově nahrané skladbě musí byt zaslána všem připojeným uživatelům. Aplikace také ihned po nahrání skladby umožní provést akci přehrávání nebo zařazení do fronty.
- **Playlisty** – Aplikace musí umožnit správu playlistů. Tato správa zahrnuje vytvoření a mazání playlistů. Do každého playlistu je možné přidat či odebrat jakoukoliv skladbu.
- **Fronta** – Uživatelé mohou přidávat skladby či celé playlisty do fronty, které budou následně přehrány.
- **Reflektování změn** – Každý playlist, který je zařazen ve frontě, musí reflektovat změny, které jsou na něm provedeny. Konkrétně se jedná o akce jako jsou přejmenovaní a přidání či odebrání skladby.
- **Přehrávání hudby** – Webová a mobilní aplikace musí umožnit přehrávání hudby za jakýchkoliv situací. Serverová aplikace umí přehrát hudbu na zařízeních, které jsou podporovány. Serverová aplikace musí fungovat i bez možnost přehrávání, kdy slouží pouze jako zprostředkovatel komunikace mezi uživateli.
- **Správa serverů** – Webová i mobilní aplikace nabídne vytvoření spojení se serverem na základně doménového jména či IPv4 adresy a specifikovaného portu. V případě, že port nebude uvedný, uvažuje se port `80`. Uživatel pak bude mít možnost mezi přidanými spojeními libovolně přepínat.

5.2 Uživatelské rozhraní

Návrh uživatelského rozhraní je pro multiplatformní aplikace komplikovanější. Apple definuje „Human Interface Guidlines“[1] a design Android aplikací je zase založen na Material designu[2]. Rozdíly designu mezi platformami lze pozorovat například na komponentě tlačítka, které má na Androidu text vždy velkým písmem, zatímco Apple v tomto případě není příliš konzistentní a lze pozorovat celý text napsaný velkým písmem, ale i text, který má velká pouze první písmena slov [19]. Material design však není vázán pouze na systém Android, nabízí design pokyny i pro vývoj aplikací pro iOS, web nebo Flutter[3].

Konzistence uživatelského rozhraní napříč aplikacemi

Návrh uživatelského rozhraní se skládá z komponent, kde je každá komponenta v každé aplikaci tvořená pomocí odlišných knihoven. V momentě, kdy chce uživatel provést akci, automaticky a často podvědomě pátrá po vodítkách, které by mu pomohly [25]. Velká část těchto vodítek je zajištěna použitím Material Design knihoven.

Prvky konzistence uživatelského rozhraní:

- **Barvy** – Obě aplikace musí používat stejnou paletu barev. Počet výrazně odlišných barev je proto dobré držet na nízkém počtu zhruba 3 barev. Stejnou barvu musí mít nosné prvky aplikace, kterými jsou navigace, komponenty skladeb/playlistů a tlačítka.
- **Rodina ikon** – Stejná rodina ikon je velice důležitá pro zajištění dobré orientace napříč aplikacemi. Proto je nutné při výběru knihovny pro tvorbu UI klást důraz i na podporu ikon. Z této analýzy se jevila nejlepší knihovna Material Design Icons[4].
- **Komponenty** – Komponenty musí na obou platformách sdělovat stejné informace, které by navíc měly být stejně pozicované. Například komponenta playlistu musí stejně zobrazovat jméno a počet skladeb.

Techniky zvyšující uživatelskou přívětivost

Uživatelská přivětivost (UX - User Experience) nezávisí pouze na hezky vytvořeném uživatelském rozhraní. Figuruje zde i chování uživatelského rozhraní, jež můžeme optimalizovat technikami jako jsou Skeleton screens nebo Optimistic UI.

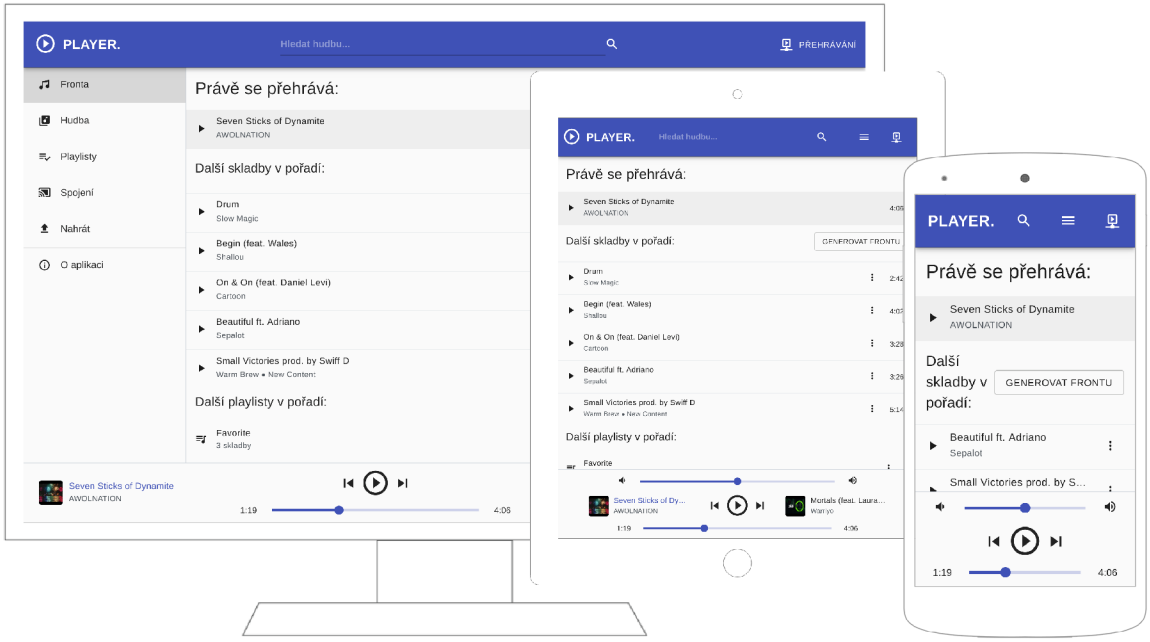
- **Skeleton screens** [10] – Slouží k zobrazení kostry stránky v momentě, kdy ještě není dostupný obsah. Uživatel si tak může lepe formulovat představu o tom, jak bude obsah na stránce rozmístěn, což mu po načtení obsahu urychlí navigaci na takové stránce. Tento typ načítání obrazovky je vlastně náhrada různých ukazatelů načítání, které uživateli nedávají žádnou informaci o budoucím obsahu. Princip skeleton screens jsem aplikoval pro komponentu skladby (obrázek 5.4 a 5.5).
- **Optimistic UI** [1] – Je technika zrychlující interakce s aplikací, které by jinak trvaly delší dobu. Tento princip je aplikovatelný na všechny komponenty ovládající hudbu.

[1] Human Interface Guidlines – https://developer.apple.com/design/human-interface-guidelines
[2] Material design – https://material.io
[3] Flutter – https://flutter.dev
[4] Material Design Icons – https://materialdesignicons.com

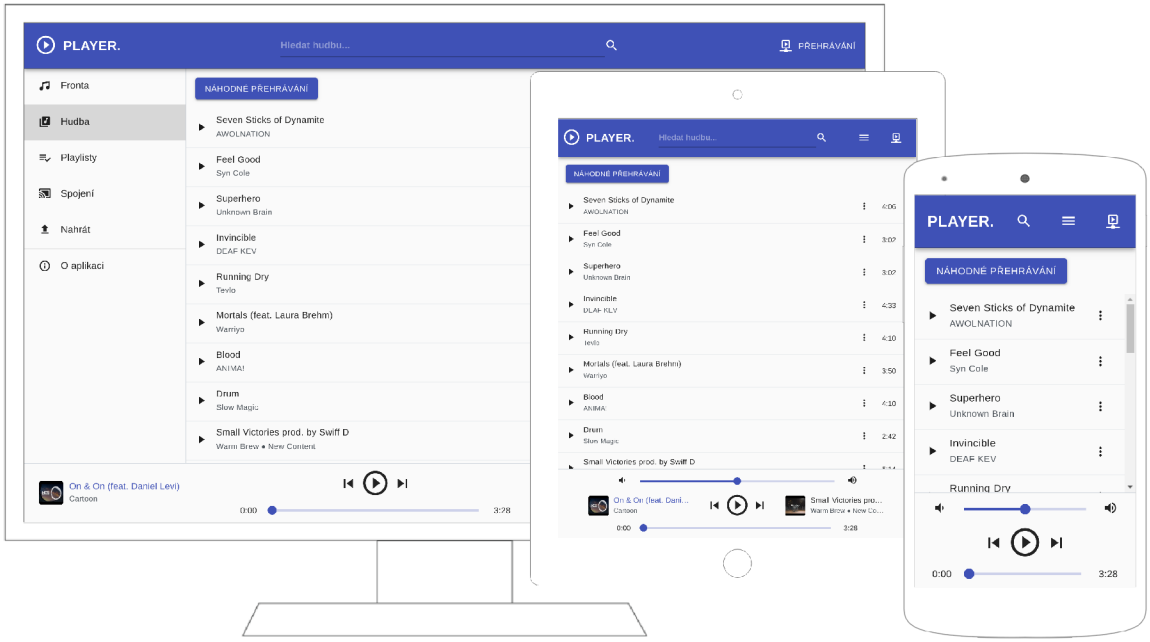
Pokud uživatel provede akci, akce je nejdříve provedena u něj a ihned potom je zaslána na server, který akci pošle všem uživatelům kromě odesílatele. Pokud by se akce nejdříve zaslala na server a až poté všem uživatelům, tak by se uživateli, který tuto akci prováděl, jevila jako zpožděná.

Uživatelské rozhraní webové aplikace

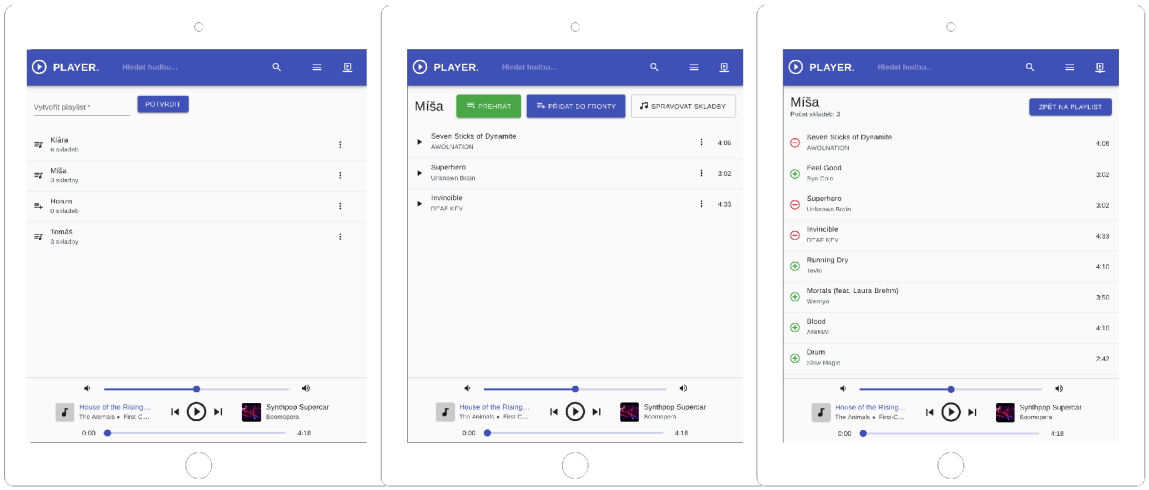
Uživatelské rozhraní je navrženo tak, aby bylo plně responzivní a uživatel tak mohl používat své oblíbené zařízení. Webová aplikace je složena z 6 hlavních obrazovek, přičemž položka „Playlisty“ obsahuje další 2 vnořené obrazovky.



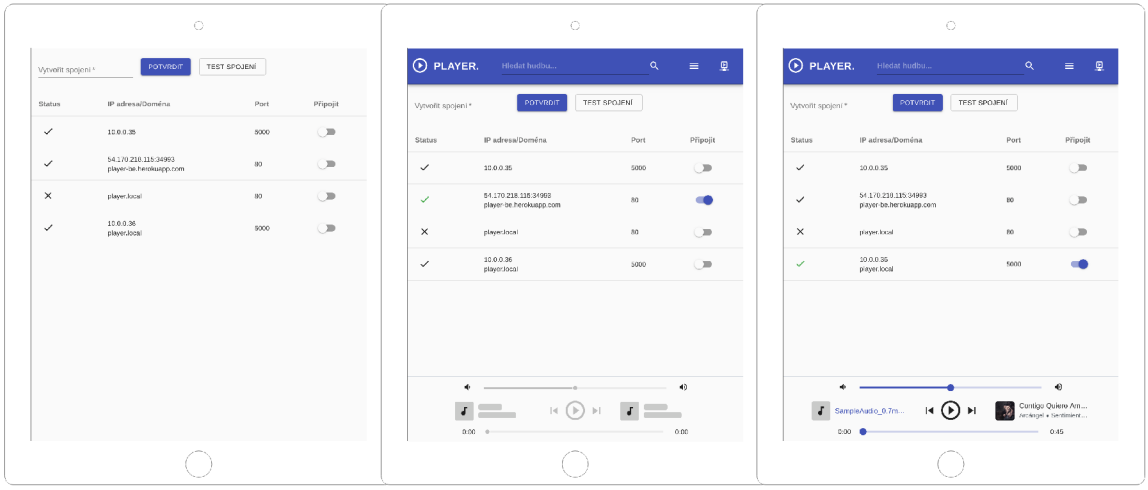
Obrázek 5.1: **Obrazovka fronty** slouží jako úvodní obrazovka. Obsahuje aktuálně přehrávanou skladbu, následující skladby a playlisty. Uživatel může generovat nadcházející skladby a odebírat či přeskakovat ve skladbách a playlistech.



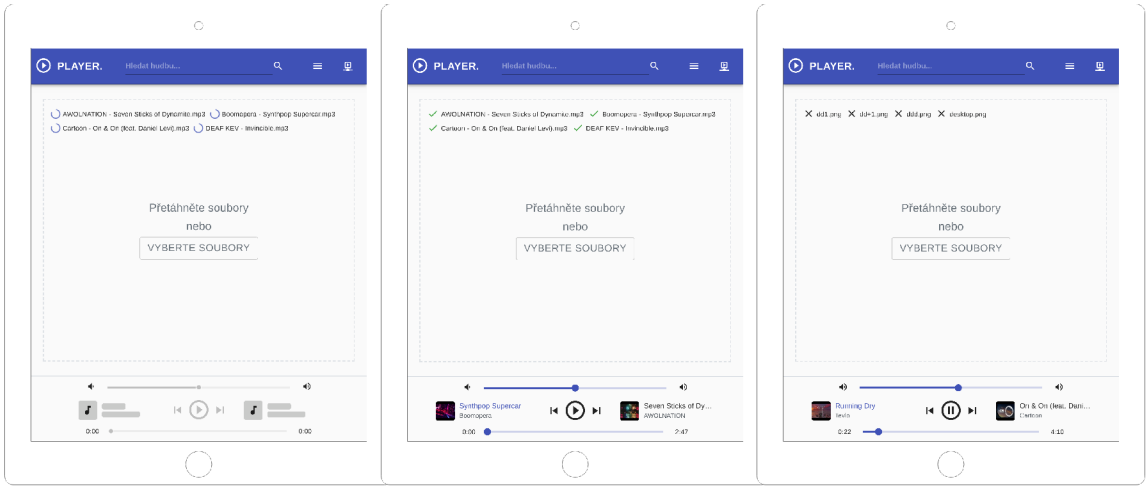
Obrázek 5.2: **Obrazovka hudby** obsahuje všechny dostupné skladby. Z této obrazovky lze náhodně přehrát všechny skladby. Jednotlivé skladby lze přehrávat/pozastavovat při kliknutí. Každá skladba obsahuje kontextové menu, které nabízí možnost přidání skladby do fronty a její přehrání.



Obrázek 5.3: **Obrazovky playlistů**. První obrazovka slouží pro vytvoření playlistů. Každý playlist obsahuje na pravé straně kontextové menu, které nabízí možnosti přehrát, přidat skladby, přidat do fronty nebo smazat. Pokud playlist neobsahuje žádné skladby, je uživatel při kliknutí na playlist přesměrován na poslední obrazovku, která slouží pro přidání/odebrání skladeb v rámci daného playlistu. Jestliže se uživatel nachází na první obrazovce a playlist již obsahuje alespoň jednu skladbu, je uživatel přesměrován na druhou obrazovku, která slouží jako přehled aktuálního playlistu. Skladby, na druhé obrazovce, obsahují také kontextové menu, ve kterém lze daná skladba přehrát, nebo odebrat z playlistu.



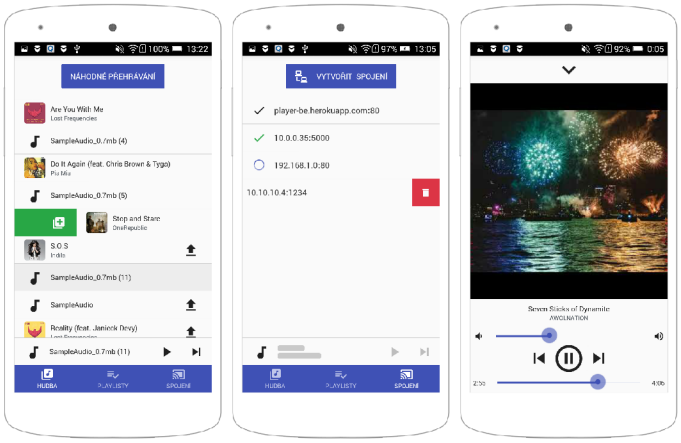
Obrázek 5.4: **Obrazovky spojení**. První obrazovka spojení je zobrazena, pokud není aplikace připojena k žádnému serveru. Prostřední obrazovka ukazuje, že je aplikace připojena k serveru, který nedisponuje žádnými skladbami. Na této obrazovce je ukázáno použití principu Skeleton screens v podobě koster skladeb v dolní části obrazovky. Současně nelze manipulovat s posuvníky, protože se nachází v `disabled` stavu. Poslední obrazovka je připojena k lokálnímu serveru, který disponuje skladbami. Mezi dostupnými servery lze libovolně přepínat.



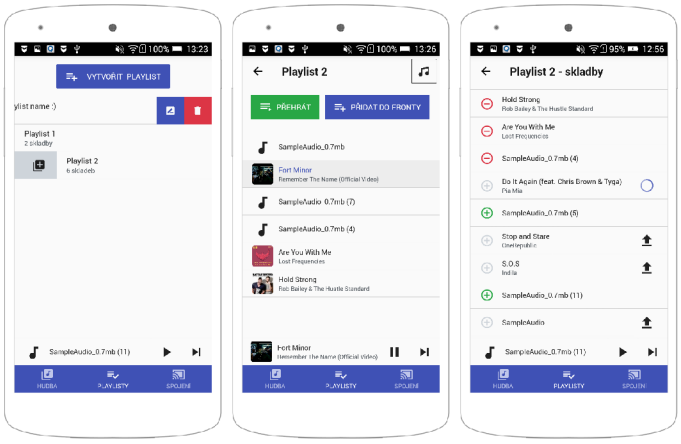
Obrázek 5.5: **Obrazovka nahrát** umožňuje nahrát skladby přímo na server. První obrazovka naznačuje přenos skladeb. Prostřední obrazovka ukazuje, že skladby byly úspěšně nahrané. Poslední obrazovka ukazuje neúspěšný přenos souborů.

Uživatelské rozhraní mobilní aplikace

Uživatelské rozhraní je složeno z 6 obrazovek, přičemž obrazovka přehrávače je společná pro všechny obrazovky, a to v minimalizované nebo v maximalizované podobě.



Obrázek 5.6: První obrazovka zobrazuje hudbu v daném zařízení současně s hudbou na serveru. Skladby mohou být nahrány na server kliknutím na ikony nahrávání, přehráním nebo přidáním do fronty. Prostřední obrazovka zobrazuje spojení, která lze přirovnat k obrazovce webové aplikace 5.4. Současně je demonstrován princip Skeleton screens na komponentě skladby. Poslední obrazovka zobrazuje maximalizovaný přehrávač. Obrazovka přehrávače lze minimalizovat tažením směrem dolů, nebo kliknutím na ikonu v horní části obrazovky.



Obrázek 5.7: **Obrazovky playlistů**, jsou podobné obrazovkám ve webové aplikaci 5.3. Mobilní aplikace nabízí gesta tažení (swipe). Při překročení určité hranice v pohybu zleva do prava, je playlist přidán do fronty. Pohyb zprava nabízí přejmenování a odstranění playlistu.

5.3 Stav aplikace

Návrh stavu aplikace vychází z kapitoly o strukturování stavu pomocí reducers funkcí (kapitola 2.3) a normalizaci dat (kapitola 2.5). Nelze tvrdit, že existuje pouze jeden správný způsob, jak strukturovat a normalizovat stav aplikace, proto je nutné přizpůsobit stav tak, aby prováděné operace byly co možná nejefektivnější. Nejvíce typická akce je výběr části stavu, která vyžaduje spojit více částí stavu v jeden.

Každá část stavu, která bude následně navržena, je vždy ve formátu JavaScriptového objektu, a to z důvodu čitelnosti. Implementace bude však realizována pomocí neměnných dat a knihovny Immutable.js, která byla představena v kapitole 2.4. Takto navržený stav lze předat funkci `fromJS(...state)`, kterou Immutable.js exportuje a vytvořit reducer funkci, která bude tuto část stavu zpracovávat.

Skladby

Struktura skladeb udržuje informace o všech skladbách, které jsou dostupné aplikaci. Tato struktura současně musí být natolik univerzální, aby její použití dávalo smysl ve všech aplikacích. Mobilní aplikace musí rozlišovat mezi skladbami, které má uložené v paměti telefonu a skladbami, které jsou přístupné ze serveru. Webová aplikace má veškeré skladby přístupné pouze ze serveru. Serverová aplikace se zase ke skladbám přistupuje lokálně. Současně je třeba brát v potaz možnost nahrání skladeb na server z mobilní aplikace, kdy je nutné indikovat, že skladba již byla synchronizována se serverem.

```
const trackAttributes = {
    title: '',
    artist: '',
    album: '',
    filename: '',
    artwork: '',
    url: '',
    duration: 0,
}

state['tracks'] = {
    fetching: false,
    tracks: {
        id1: {
            id: 'id1',
            ...trackAttributes,
            meta: {
                type: 'local' | 'remote',
                synced: false
            }
        }
    }
}
```

- **`trackAttributes`** – Parsování skladeb bude na serverové aplikaci realizováno pomocí knihovny music-metadata[5] a v mobilní aplikaci pomocí knihovny react-native-

[5] music-metadata – https://www.npmjs.com/package/music-metadata

get-music-files[6]. Obě knihovny mají odlišný výstup, a proto je nutné zajistit stejnou strukturu skladeb napříč aplikacemi.

- `meta` – Serverová aplikace při startu označí veškeré skladby jako `{type: 'remote'}` a mobilní aplikace zase `{type: 'local'}`. Uživatel mobilní aplikace může nahrát skladby typu `{type: 'local', synced: false}` na server. Pokud server obdrží novou skladbu, zašle informaci o nové skladbě všem uživatelům. Každý uživatel, který nově nahranou skladbu již má u sebe, si tuto informaci poznačí jako `{synced: true}`. Analogickým způsobem by bylo realizováno stahování skladeb ze serveru pro skladby, které mají nastavený `meta` objekt jako `{type: 'remote', synced: false}`. V kapitole 4.1 byl zmíněn „Party mode“ aplikace. V tomto módu by přibyl v `meta` objektu další atribut `votes`, podle něhož by se řadily skladby ve frontě.

Playlisty

Pořadí výpisu playlistů je zajištěno pomocí atributu `createdAt`, jehož hodnota je Unix time[7]. Pole skladeb obsahuje pouze identifikátory skladeb. Generování identifikátorů playlistů je realizováno na základě atributu `name` a `createdAt`. Atribut `type` je opět důležitý pro rozlišení, zda je playlist vytvořen v době, kdy je aplikace připojena k serveru.

```
state['playlists'] = {
    378336043: {
        id: '378336043'
        name: 'Name',
        createdAt: 1556278370844,
        type: 'remote',
        tracks: ['1018765506', '581514383', ...]
    }
}
```

Přehrávač

Struktura přehrávače je velice jednoduchá a snaží se udržovat co možná nejmenší počet atributů, nutných k přehrávání. Počáteční struktura je následující:

```
state['player'] = {
    isPlaying: false,
    trackId: null,
    seek: 0,
    volume: 50,
}
```

Fronta

Pokaždé, když se má přehrávat předchozí/další skladba je struktura fronty použita pro zjištění informace, jaká skladba to bude. Atribut `activePlaylist` slouží jako index pro přístup k aktuálnímu playlistu v poli `playlists`. Pokud je hodnota atributu `activePlaylist` rovna -1, pak se přehrávají skladby z pole `next`, jinak se berou skladby z playlistu na daném indexu.

[6] react-native-get-music-files – https://www.npmjs.com/package/react-native-get-music-files
[7] Unix time – https://en.wikipedia.org/wiki/Unix_time

```
state['queue'] = {
    activePlaylist: -1,
    prev: ['1018765506', '581514383', ...],
    next: ['1018765506', '581514383', ...],
    playlists: [{id: '378336043', name: 'Name', tracks: ['1018765506']}, ...],
}
```

Spojení

Struktura spojení udržuje informace o spojeních. Každé spojení má uložené informace o serveru a jeho dostupnosti. Navázání spojení se serverem je rozděleno do několika fází. Z počátku je aplikace ve stavu `offline`. V momentě, kdy si uživatel vybere dostupné spojení, začne se k němu aplikace připojovat. Aplikace u uživatele nastavuje stavy v rozmezí `<offline, connected>`, server pak nastavuje stavy `synchronizing` a `online`.

```
state['connection'] = {
    activeConnection: '1321751923',
    status: 'offline' | 'searching' | 'searching/done' | 'connecting'
            | 'connected' | 'synchronizing' | 'online'
    connections: {
        1321751923 : {
            id: '1321751923',
            ip: '10.0.0.36',
            domain: 'player.local',
            port: 5000,
            status: 'unavailable' | 'testing' | 'available'
        }
    }
}
```

Nastavení

Struktura nastaveni aplikace je složena ze dvou částí. Je zde tak využito funkce `combineReducers`, která byla zmíněna v kapitole 2.3.

```
state['config'] = {
    app: {
        port: 8080,
        localIP: '10.0.0.35'
        publicIP: '',
        env: 'SERVER' | 'MOBIL' | 'WEB'
    },
    playback: {
        playlists: 'waterfall' | 'jukebox',
        interactions: 'public' | 'private',
    }
}
```

- `app` – Táto část uchovává informace o tom, na jakém běží aplikace portu nebo jakou má lokální a veřejnou IP adresu. Atribut `env` je důležitý pro to, aby bylo možné mířit s určitou funkcionalitou pouze na konkrétní platformy.

- `playback` – Atribut `playlists` specifikuje způsob, jakým jsou playlisty přehrávány. Hodnota `waterfall` přehraje vždy jednu skladbu z aktuálního playlistu, a pokud je ve frontě více playlistů, přehraje se poté jedna skladba z dalšího playlistu. Atribut `jukebox` přehraje vždy celý playlist a po jeho skončení se začne přehrávat další playlist. Atribut `interactions` specifikuje způsob chování aplikace. Hodnota `public` zpřístupňuje uživateli kontrolu přehrávání na serveru. Hodnota `private` umožňuje přehrávání přímo v aplikaci u uživatele. S těmito hodnotami souvisí i sdílení akcí. V `public` režimu sdílí uživatel veškeré akce s ostatními uživateli, kteří se také nachází v `public` režimu. V režimu `private` jsou veškeré akce provedeny pouze u uživatele a ostatní uživatelé jimi nejsou nijak ovlivněni.

Kapitola 6

Implementace systému

Serverová, webová i mobilní aplikace bude implementovaná v jazyce JavaScript, a to v nejnovější specifikaci ECMAScript 2018[1], která vyšla v roce 2018. Hlavní důvod pro použití této verze jsou nové `rest` a `spread` operátory. Využití našla i nová konstrukce asynchronních iterací v serverové aplikaci. Podpora specifikace ES9 je v případě serverové i mobilní aplikace plně v režii vývojáře, avšak u uživatelů, kteří využívají webovou aplikaci, je dobré počítat s tím, že nepoužívají nejmodernější prohlížeče, které tyto specifikace implementují. Pro podporu těchto nových specifikací se používají různé polyfilly[2]. Nejznámější knihovna takových pollyfilů je core-js[3], která je implementovaná v rámci `@babel/polyfill`, což je modul pro JavaScript kompiler Babel[4].

Serverová i webová aplikace využívá pro sestavení nástroj Webpack[5]. Mobilní aplikace, která je postavená na frameworku React Native, využívá pro sestavení nástroj Metro[6]. Všechny aplikace využívají kompilátor Babel.

6.1 Struktura repozitáře

Struktura repozitáře vychází z kapitol o monolitických repozitářích 2.1 a sdílení kódu prostřednictvím společného adresáře v projektu, jak bylo představeno v kapitole 2.2. Na základě těchto znalostí jsem navrhl následující strukturu repozitáře:

```
/
|-- data    // contains tracks and generated artworks from track's files
|-- mobile  // mobile application
|-- server  // server application
|-- web     // web application
+-- shared  // modules which are being shared among applications
```

Nástroj Metro, který je lokalizován v adresáři `mobile` nedokáže bez dalšího nastavení reagovat na změny vytvořené mimo jeho adresář. Nastavení není složité, ale stalo se mi, že se s novou verzí nepatrně změnilo. Aktuální nastavení je realizováno v souboru `metro.config.js` a přidáním záznamu do pole `watchFolders`.

[1] ECMAScript 2018 – https://www.ecma-international.org/ecma-262/9.0
[2] Polyfill - Kód, který implementuje funkcionalitu, která není podporovaná.
[3] core-js – https://github.com/zloirock/core-js
[4] Babel – https://babeljs.io
[5] Webpack – https://webpack.js.org
[6] Metro – https://facebook.github.io/metro

Dalším důležitým prvkem optimalizace je importování funkcionality ze sdíleného adresáře do aplikací. Jistě není žádoucí importovat funkce ze sdíleného adresáře uvnitř aplikace tímto způsobem:

```
import {modul} from '../../../shared/modul';
```

Pokud se soubor, který importuje takový modul, kamkoliv přesune, je třeba myslet i na případnou opravu veškerých importů. Řešení tohoto problému představuje v případě webové i serverové aplikace plugin `Resolve`[7], který pochází od nástroje Webpack. Pro nástroj Metro jsem nenašel žádnou alternativní funkcionalitu. Existuje však plugin `babel-plugin-module-resolver`[8] pro Babel, který daný problém řeší. Po nastavení pluginů je výsledek importování ve všech aplikacích následující:

```
import {modul} from 'shared/modul';
```

6.2 Společné aspekty vývoje webové a mobilní aplikace

Po ustanovení struktury repozitáře je nutné definovat strukturu uvnitř webové a mobilní aplikace. Na tuto strukturu jsou kladeny vysoké nároky, co se týče modularity a škálovatelnosti. Obecná strukturu pro vývoj takových aplikací je již skvěle popsána v článku „The Anatomy Of A React & Redux Module“ [17], ze které tento konkrétní návrh čerpá.

Struktura adresářů v `/mobile` a `/web`:

```
/
|-- components  // Presentational React/React Native components
|-- containers  // Container components - Connected Presentational componnets
|-- core        // Specific application code
|-- screens     // Contains application screens
+-- utils       // Extracted code
```

Prezentační komponenty obsažené v `/components` jsou dále strukturovány do adresářů se jmény komponenty. Každý takový adresář poté definuje soubor index.js, který exportuje komponenty pod jejich jmény, jedná se o tzv. „named exports“. V případě webu, může být v komponentách přítomný i soubor `index.scss`, který definuje styly pro danou komponentu. Tyto styly jsou definovány pomocí BEM[9] konvencí, které napomáhají k lepší udržovatelnosti a pochopení významu stylů.

Struktura v adresáři `/shared`:

Obsahuje abstraktní moduly, které jsou sdílené mezi všemi aplikacemi. Struktura modulu je složena z následujících částí:

```
/module
|-- constants.js // Required always
|-- actions.js   // Required if Redux parts are present otherwise optional
|-- reducer.js   // Required if Redux parts are present otherwise optional
|-- saga.js      // Optional
|-- selectors.js // Required if Redux parts are present otherwise optional
+-- index.js     // Required always
```

[7] Resolve – https://webpack.js.org/configuration/resolve
[8] babel-plugin-module-resolver – https://github.com/tleunen/babel-plugin-module-resolver
[9] BEM – Block Element Modifier, více viz http://getbem.com

Tyto soubory jsou součástí téměř každého modulu. Dále se však mohou vyskytovat i adresáře s Redux middlewarem, React Hooks, kontejnerové komponenty nebo dokonce prezentační komponenty využívající abstrakci, která jim umožní nasazení v mobilní i webové aplikaci.

Soubor `constants.js` obsahuje pojmenování modulu, které je podle konvence vždy realizováno konstantou se jménem `NAME`. Použití této konstanty je platné pouze uvnitř daného modulu. Důvody použití této konvence jsou následující:

- Definuje zanoření modulu v rámci stromové struktury stavu
- Slouží jako prefix všech akcí, které jsou hodnotou atributu `type` pro funkce vytvářející akce tzv. `Actions creators`
- Je součástí cesty pro přístup k dané části stavu v případě selektorů.

Soubor `actions.js` definuje a exportuje typy akcí včetně funkcí, které tyto akce vytváří. Teoreticky není nutné, aby veškeré typy akcí byly vytvářené pomocí funkcí, protože existují případy, kdy není třeba pracovat s žádným dalším argumentem. V rámci zachování konzistence je ale lepší, pokud veškeré akce jsou vytvářeny a existuje tedy jednotné API, o kterém není třeba při použití příliš přemýšlet.

Základní struktura akcí a funkcí, které tyto akce vytvářejí je popsána v kapitole 2.3. Nyní, po představení souboru `constants.js`, lze však zajít dále a příklad z teorie abstrahovat následovně:

```
// constants.js
export const NAME = 'tracks'; // By convention valid only in module scope

// actions.js
import {NAME} from './constants';

export const TRACKS_SAVE = NAME + '/SAVE'; // Composed action

export const tracksSave = (tracks) => ({  // Action creator
    type: TRACKS_SAVE,
    tracks,
});
```

Výhoda této abstrakce spočívá v zajištění unikátnosti jména akce v rámci celé aplikace, protože každý modul musí mít přiřazené unikátní jméno. Další výhoda je při použití debug nástroje pro React Native[10] nebo pro React[11], protože hodnota konstanty `NAME`, která figuruje jako prefix akce, přesně odpovídá danému modulu.

Soubor `reducer.js` definuje a exportuje reducer funkci pro daný modul. Struktura reducer funkce je popsána v kapitole 2.3. Jediná změna je v nahrazení hodnoty `case` příkazu za konstanty definované v souboru `actions.js`. Je-li to nutné, tento soubor může exportovat funkci `combineReducers`, která umožňuje kombinovat více reducer funkcí.

[10] React Native debug – https://github.com/jhen0409/react-native-debugger
[11] React debug – https://github.com/zalmoxisus/redux-devtools-extension

Soubor `saga.js` definuje a exportuje generátor funkci pro daný modul. Tato funkce registruje generátor funkce, které čekají na akce provedené v rámci stavu aplikace. Tomuto typu exportované ságy se říká „watcher saga“ a její příklad je následující:

```
import {takeEvery} from 'redux-saga/effects';
import {TRACKS_SAVE} from './actions';

function *trackSaveSaga(action){
    const tracks = action.tracks;
    // Process tracks
}

export default function* tracksSaga() { // Watcher saga
  yield takeEvery(TRACKS_SAVE, trackSaveSaga);
}
```

Callback funkce `trackSaveSaga` je standardně zavolána až poté, co je provedena reducer funkce. Standardně znamená, že se na toto chování lze spolehnout, pokud v Redux middlewaru není představeno asynchronní zpracování.

Soubor `selectors.js` definuje a exportuje selektor funkce pro daný modul. Tyto funkce jsou představeny v kapitole 2.3. Implementace `selectors.js` je následující:

```
import {NAME} from './constants';
const getModel = (state) => state.get(NAME);

export const selectTracks = (state) => getModel(state).get('tracks');
```

Soubor `index.js` je chápán jako API daného modulu a jeho podoba je následovná:

```
import {NAME} from './constants';
import saga from './saga';
import reducer from './reducer';

export default {
  NAME,       // required
  reducer,    // required
  saga        // optional
};
```

Spojení návrhu adresáře `/shared` s jeho realizací

Moduly, který se skládají z několika souborů, je nutné spojit dohromady. Pro tento účel existují další 3 soubory.

Soubor `createReducer.js` se zodpovědností spojit veškeré reducer funkce všech modulů.

```
import {combineReducers} from 'redux-immutable';

export const createReducer = (modules) => combineReducers(
        modules.map(({NAME, reducer}) => ({[NAME]: reducer}))
               .reduce((combinedReducers, moduleReducer) =>
                           ({...combinedReducers, ...moduleReducer}), {}),
);
```

Soubor `createSaga.js` se zodpovědností nastartovat veškeré ságy.

```
import {all, fork} from 'redux-saga/effects';

export const createSaga = (sagas) =>
        function* rootSaga() {
            return yield all(sagas.map(fork));
        };
```

Soubor `createStore.js` jehož implementace je složitější a velice klíčová pro správné fungování celé aplikace. Důležitá specifika funkce `createStore`, která je exportována, jsou následující:

- **Registrace Redux částí** – Registruje reducers funkce, které jsou vytvořeny funkcí `createReducers` a také Redux middleware, který jednak může registrovat vývojář, ale i aplikace sama, jako je tomu v případě knihovny Redux-Saga.
- **Povolení Redux devtools pouze při vývoji aplikace** – Pokud webová aplikace nezakáže devtools v produkční verzi, uživatel této aplikace má přístup k celému stavu aplikace, ve kterém je schopný nalézt informace, ke kterým by neměl mít přístup.
- **HMR**[12] – Hot Module Replacement je důležitá vlastnost pro vývojáře aplikace při jejím vývoji. Tato vlastnost rapidně zvýší jeho produktivitu, protože nemusí ručně obnovovat stránku při každé změně. Stejně tak je důležité zachování aktuálního stavu během změn. Avšak správné nastavení HMR vyžaduje korektní obnovu reducers a sága funkcí. V dřívějších verzích knihovny Redux byla obnova automatická, nyní je tato zodpovědnost přenesena na vývojáře.

Struktura v adresáři `/screens`:

Struktura tohoto adresáře je závislá na různorodosti případných navigačních knihoven konkrétní platformy, proto je velmi těžké definovat přesnou strukturu. Výběr navigační knihovny je rozebrán v kapitole 6.3. Avšak společný základ pro každou obrazovku je následovný:

```
/screen
|-- constants.js
|-- Screen.js
+-- index.js
```

Soubor `constants.js` exportuje alespoň jednu konstantu pojmenovanou `ROUTE`, která definuje endpoint pro danou obrazovku.

Soubor `Screen.js` exportuje komponentu, která je logicky propojena s konstantou `ROUTE` daného modulu.

Soubor `index.js` exportuje informace o modulu obrazovky. Přesný formát je dále nedefinován.

[12] Hot Module Replacement – https://webpack.js.org/concepts/hot-module-replacement

6.3 Navigace v aplikacích

Navigace v aplikacích je třeba řešit, pokud aplikace obsahuje více než 1 obrazovku. Cílem je zobrazení vyžádaných obrazovek uživatelem. Tyto obrazovky jsou v aplikaci mapovány na tzv. „endpointy“, ke kterým uživatel přistupuje. Knihovny řešící navigaci jsou někdy nazývané jako „routovací“ či „navigační“ knihovny.

Webové aplikace

Mezi nejznámější navigační knihovny patří: React Router[13], Router5[14] nebo Universal Router[15]. Jako kandidáta pro implementaci jsem vybral knihovnu React Router, protože je nejpopulárnější a nejvíce aktivně udržovaná. React Router také nabízí implementaci navigace i pro mobilní aplikace, což zní velice slibně.

Rozhodl jsem se tedy využít techniku PoC (Proof of Concept), abych zjistil, jak si React Router stojí při implementaci ve webové i mobilní aplikaci. Ukázalo se, že implementace byla funkční a přímočará v obou případech, avšak implementace pro mobilní zařízení nepůsobila příliš „nativním“ dojmem. Proto jsem se rozhodl hledat další knihovny pro React Native, které řeší problém navigace.

Mobilní aplikace

React Native nabízí větší množství navigačních knihoven. Jako kandidáty pro implementaci jsem vybral knihovny:

- **React Native Navigation**[16] – Tato knihovna slibuje velice dobrý výkon, protože z velké části využívá nativní vlákno. Proces integrace této knihovny je při nejmenším nelehký. Tato knihovna nenabízí automatické linkování, a proto je nutné vše udělat ručně. Ruční úprava zahrnuje úpravu několika souborů v obou platformách. Knihovna je podporována pro systém Android od `minSdkVersion=19`, což odpovídá operačnímu systému verze 4.4 a výše.
- **React Navigation**[17] – Integrace této knihovny je automatická až na úpravu jednoho souboru pro systém Android. API této knihovny je v porovnání s webovými knihovnami trochu složitější, ale stále velice přívětivé. Podpora této knihovny je pro systém Android od `minSdkVersion=16`.

Závěr

Konečný výběr knihovny je však závislý na možnosti implementovat komponentu, která je sdílena napříč všemi obrazovkami a přitom musí podporovat navigaci. A tedy tato komponenta nesmí být překreslena při žádné změně obrazovky. To se v případě React Native Navigation ukázalo jako nemožné. Ani React Navigation pro tohle řešení nemá připravenou žádnou implementaci, ale existuje vlákno na platformně GitHub[18], které nabízí řešení tohoto specifického případu.

[13] React Router – https://github.com/ReactTraining/react-router
[14] Router5 – https://github.com/router5/router5
[15] Universal Router – https://github.com/kriasoft/universal-router
[16] React Native Navigation – https://github.com/wix/react-native-navigation
[17] React Navigation – https://github.com/react-navigation/react-navigation
[18] React Navigation a sdílená komponenta – https://github.com/react-navigation/react-navigation/issues/1439#issuecomment-303661539

Navigace ve webové aplikaci bude realizována pomocí **React Router** a v mobilní aplikaci pomocí **React Navigation**.

6.4 Uživatelské rozhraní

Implementace uživatelského rozhraní je založena na principech Material Designu. Webové aplikace budou implementovány knihovnou React, která byla představena v kapitole 2.7. Mobilní aplikace zase pomocí knihovny React Native, která je více rozebrána v kapitole 2.8. Výběr knihoven React a React Native je založen na myšlence sdílení kódu mezi platformami. Po ustanovení základních technologií je třeba vybrat knihovny, které implementují principy Material Designu pro webové a mobilní aplikace.

Webové aplikace

- **Materialize**[19] – Je framework, který implementuje komponenty v HTML kódu, které dodržují přesnou hierarchii elementů s jejich CSS třídami. Pro tento projekt, který je založený na knihovně React existuje knihovna React-Materialize[20], která slouží jako obálka neboli „wrapper“ pro Materialize knihovnu. Materialize disponuje dobrou dokumentací a velkým množstvím komponent, které tento projekt vyžaduje. React-Materialize jakožto wrapper nabízí stránku s dokumentací, která je založena na prezentaci komponent knihovnou Storybook[21], avšak tato dokumentace působí nedodělaným dojmem, protože nejsou implementované Knobs a Actions pluginy, které přesto zabírají značkou část obrazovky. Počet týdenních stažení knihovny se pohybuje kolem 5 000[22].

- **Material-UI**[23] – Je knihovna, která přímo implementuje Material Design v knihovně React. Knihovna uvolňuje odladěné komponenty ve svém jádru a komponenty, které jsou aktuálně v rámci vývoje zase prostřednictvím speciálního balíčku `Lab`. Konfigurovat styly je možné v rámci celé aplikace, ale i na úrovní jednotlivých komponent. Je zde i podpora pro módy uživatelského rozhraní ve smyslu tmavý/světlý mód. Některé ukázky komponent zahrnují integrace s knihovnami třetích stran, což je dobrým znamením. Počet týdenních stažení knihovny se pohybuje kolem 1 000 000[24].

Závěr

Uživatelské rozhraní pro webovou aplikaci bude realizované pomocí knihovny **Material-UI**, protože je z dlouhodobého hlediska perspektivnější díky své popularitě mezi komunitou a protože nabízí propracovaný systém konfigurace komponent.

Mobilní aplikace

- **React Native Material UI**[25] – Je knihovna nabízející 18 komponent implementující Material Design. Konfigurace jednotlivých komponent je realizována objektem, který

[19] Materialize – https://materializecss.com
[20] React-Materialize – https://www.npmjs.com/package/react-materialize
[21] Storybook – https://github.com/storybooks/storybook
[22] React-Materialize stažení – https://www.npmjs.com/package/react-materialize
[23] Material-UI – https://material-ui.com
[24] Material-UI stažení – https://www.npmjs.com/package/@material-ui/core
[25] React Native Material UI – https://github.com/xotahal/react-native-material-ui

je nazván `uiTheme`. Knihovna nabízí ukázky a API komponent prostřednictvím webu GitHub. Počet týdenních stažení knihovny se pohybuje kolem 3 500[26].

- **NativeBase**[27] – Je knihovna, které implementuje 27 komponent. Knihovna nabízí konfigurace založené na Material Designu a platformně specifickém vzhledu. Oba tyto módy nabízejí vlastní konfiguraci, která je realizována vlastními soubory obsahující konfigurační objekt. Tyto soubory jsou překopírovány z `node_modules` adresáře přímo do aplikace. Dokumentace je vytvořena jako webové aplikace, což vždy působí seriózněji dojmem než dokumentace na platformně GitHub. Dalším skvělý počin této knihovny je vlastní konfigurátor[28] pro vlastní mód specifikující vzhled komponent. Počet týdenních stažení knihovny se pohybuje kolem 37 000[29].

Závěr

Pro výběr knihovny jsem se rozhodl udělal PoC[30], který se skládal z implementace základních komponent: `Button`, `ListItem`, `BottomNavigation` a `Icon`, na kterých aplikace stojí. Při stylování komponent jsem často musel nahlédnout do dokumentace, která se v případě React Native Material UI, ukázala jako zastaralá a neúplná, proto jsem musel často pro aktuální API komponent nahlédnout přímo do kódu knihovny. Dokumentace knihovny NativeBase je aktuální a přehledná.

Přepisování stylů u knihovny React Native Material UI se realizuje na základě jasné struktury, což ovšem vyžaduje velké množství uživatelského kódu přímo v aplikaci. NativeBase nedefinuje žádné speciální struktury jako předešla knihovna, proto je třeba mnohem méně kódu k dosažení stejného efektu.

Ukázalo se, že **NativeBase** je mnohem pokročilejší knihovna, která je vhodná k implementaci uživatelského rozhraní pro mobilní aplikaci.

6.5 Přehrávání hudby

Uživatel jakékoliv aplikace by měl být schopný přehrát si hudbu. Každá aplikace přitom přehrávání realizuje jiným způsobem. Proto jsem navrhl rozhraní (obrázek 6.1), které musí každá aplikace implementovat, aby mohla přehrávat hudbu.

Obrázek 6.1: Rozhraní přehrávače, které je implementováno specifickými aplikacemi.

[26] React Native Material UI stažení – https://www.npmjs.com/package/react-native-material-ui
[27] NativeBase – https://nativebase.io
[28] NativeBase UI konfigurátor – https://nativebase.io/customizer
[29] NativeBase stažení – https://www.npmjs.com/package/native-base
[30] PoC – Proof of Concept, více viz https://en.wikipedia.org/wiki/Proof_of_concept

Třída, která implementuje rozhraní (obrázek 6.1), je dále předána do middlewaru knihovny Redux, který je stejný pro všechny aplikace. Zjednodušená implementace middlewaru přehrávače je následující:

```
export const createPlayerMiddleware = (SpecificPlayer) => {
  return store => {
    const player = new SpecificPlayer(store);

      return (next) => (action) => {
        next(action);

     /* Decide whether media player should be playing
      *       - Server    play if player is installed on the system
      *       - Web       play if user is in private mode
      *       - Mobil     play if user is not connected to the server
      */
      switch (action.type) {
        case PLAYER_PLAY : {
          // Add track to queue.prev if track is not taken from queue.prev
          return player.play();
        }
        case PLAYER_PAUSE : return player.pause();
        case PLAYER_STOP : return player.stop();
        case PLAYER_RESUME : return player.resume();
        case PLAYER_RESET : return player.reset();
        case PLAYER_SET_SEEK: return player.seek(action.seconds);
        case PLAYER_SET_VOLUME: return player.volume(action.volume);
      }
    };
  };
};
```

Webová aplikace

Webové aplikace nabízejí přehrávání hudby pomocí `Web Audio API`[31]. Toto API není vůbec podporováno v prohlížečích Internet Explorer, a proto jsem našel knihovnu Howler.js[32], která má velice přímočaré API, a navíc je kompatibilní s velkou škálou prohlížečů.

Mobilní aplikace

Po analýze knihoven, které React Native nabízí, jsem se rozhodl vybrat knihovnu React Native Track Player[33], která disponuje dobrým API a aktuální dokumentací. Tato knihovna podporuje přehrávání na systémech Android, iOS i Windows včetně přehrávání na pozadí. Zároveň také podporuje přehrávání z lokálních souborů, ale také souborů dostupných z Internetu.

Serverová aplikace

Vzhledem k tomu, že primární cíl této práce je přehrávání na Raspberry Pi, zaměřil jsem se proto na nalezení přehrávače, který je pro tuto platformu optimalizovaný.

[31] Web Audio API – https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
[32] Howler.js – https://howlerjs.com
[33] React Native Track Player – https://react-native-kit.github.io/react-native-track-player

- **OMXPlayer**[34] – Je oficiálně podporovaný program pro přehrávání hudby na Raspberry Pi, který využívá HW akceleraci. Tento přehrávač běží jako aplikace v systému Raspbian, a proto je nutné nalézt způsob, jak s tímto programem komunikovat. Jeden způsob komunikace je prostřednictvím klávesnice, kdy je možné spustit další proces, který bude zasílat zprávy aktivnímu oknu o stisku klávesy. Dalším způsob, jak komunikovat, je zasílání zpráv prostřednictvím D-Bus[35]. Kommunikaci skrze D-Bus implementuje knihovna OMX-interface[36].

Ukázalo se, že implementace pomocí přehrávače OMXPlayer nebyla vůbec efektivní. Trvalo vždy zhruba 3s, než došlo k přehrání hudby. Knihovna OMX-interface je, dle mého, velmi špatně napsaná. Při rychlé změně skladby se často stávalo, že knihovna nedokázala rušit vytvořené procesy a některé skladby se tak přehrávaly, i když neměly. Nebylo ani možné nastavit výchozí hlasitost přehrávání, a protože knihovna nenabízí žádný `callback`, kterým by se hlasitost dala změnit ihned po startu, byl poslech hudby značně nekomfortní.

Po další analýze možných přehrávačů jsem nalezl multiplatformní hudební přehrávač mpv[37].

- **mpv** – Je přehrávač, který řeší veškeré problémy, které byly s spojené s přehrávačem OMXPlayer. Komunikace s mpv je realizována protokolem založeným na JSON IPC[38]. Existuje knihovna Node-MPV[39] implementující komunikaci skrze tento protokol. Další zajímavá vlastnost přehrávače mpv je možnost přehrávání hudby přímo ze služeb jako jsou YouTube nebo SoundCloud. Pro toto přehrávání je však nutné doinstalovat knihovnu youtube-dl[40].

Přehrávání na serveru je tedy možné nejen na Raspberry Pi, ale potažmo na všech počítačích, jež disponují přehrávačem mpv a prostředím Node.js. Funkčnost přehrávání je však aktuálně zaměřena na operační systém Linux v distribucích Raspbian a Ubuntu 18.04.

Multimediální soubory a podpora v aplikacích

Serverová i mobilní aplikace analyzuje každý hudební soubor pro zjištění jeho metadat. Struktura metadat, která je uložena o každé skladbě, je definována v kapitole 5.3.

Serverová aplikace používá mpv přehrávač, který umí přehrávat mnohem více formátů, než je uvedeno v tabulce 6.1. Problém je spíše s analýzou skladeb. Při testu formátu jsem zjistil, že při analýze formátu `.aac`, chybí `duration` atribut. Napsal jsem proto „bug report[41]“ na stránku knihovny, kde jsem se dozvěděl, že implementace tohoto atributu byla záměrně přeskočena, protože výpočet je příliš náročný, Avšak autor řekl, že výpočet délky skladby bude podmíněn nastavením `{duration: true}` při začátku analýzy.

Mobilní aplikace používá knihovnu react-native-get-music-files[42], která najde všechny hudební soubory, které následně analyzuje a předá aplikaci. Přehrávání hudby je zajištěno knihovnou react-native-track-player. Bohužel ani jedna knihovna nespecifikuje podporované formáty multimediálních souborů, a proto jsou v tabulce 6.1 použité znaky **?**.

[34] OMXPlayer – https://www.raspberrypi.org/documentation/raspbian/applications/omxplayer.md
[35] OMXPlayer D-Bus – https://github.com/popcornmix/omxplayer/blob/master/dbuscontrol.sh
[36] OMX-interface – https://github.com/diederikfeilzer/node-omx-interface
[37] mpv – https://mpv.io
[38] JSON IPC – https://github.com/mpv-player/mpv/blob/master/DOCS/man/ipc.rst
[39] Node-MPV – https://www.npmjs.com/package/node-mpv
[40] youtube-dl – https://github.com/ytdl-org/youtube-dl
[41] https://github.com/Borewit/music-metadata/issues/207
[42] react-native-get-music-files – https://www.npmjs.com/package/react-native-get-music-files

Tabulka 6.1: Tabulka ukazuje stav podpory konkrétních multimediálních souborů podle jejich přípony. Přičemž znak ✓ značí plnou podporu a znak ✗ říká, že daný formát není podporován. Znak ∼ značí částečnou podporu, kdy při zpracování skladby nejsou dostupné všechny metadata. Znak **?** říká, že podpora nemohla byt otestována.

	Server		**Mobil**		**Web**
	Analyzátor	Přehrávač	Analyzátor	Přehrávač	Přehrávač
`.3gp`	✓	✓	✗	?	✗
`.aac`	∼	✓	✓	✓	✓
`.flac`	✓	✓	✓	✗	✓
`.m4a`	✓	✓	✓	✓	✓
`.mp3`	✓	✓	✓	✓	✓
`.mp4`	✓	✓	✗	?	✓
`.ogg`	✓	✓	✓	✓	✓
`.opus`	✓	✓	✗	?	✓
`.wav`	✓	✓	✓	✓	✓
`.wma`	✓	✓	✓	✗	✗

Přehrávání na Raspberry Pi

Raspberry Pi nemá žádný integrovaný reproduktor, ale na desce disponuje 3.5 *mm* Jack konektorem a HDMI výstupem verze 1.4. Kvalitnější reprodukce zvuku lze dosáhnout připojením externí zvukové karty. Ceny těchto zvukových karet se pohybují od stovek až po několik tisíc korun. Výsledná kvalita zvuku je však znatelně lepší, protože jsou dodrženy standardy TOSLINK[43], nebo S/PDIF[44].

6.6 Sdílení kódu a multiplatformní komponenty

Následující statistiky se týkají pouze kódu napsaného v jazyce JavaScript. Měření bylo provedeno pomocí nástrojem Cloc[45], který měří počet řádků kódu. Výsledky tohoto měření jsou spíše orientační, protože se kód může kdykoliv změnit.

Tabulka 6.2: Statistiky kódu

	Počet řádků	**% z celkového počtu**	**% sdíleného kódu v adresáři**
`web`	3 500	34 %	40 %
`mobil`	3 000	29 %	43 %
`server`	1 500	15 %	50 %
`shared`	2 300	22 %	100 %

Komponenty tvoří minoritní část kódu, který je sdílen. Avšak velice zajímavá je abstrakce komponenty, která se stará o zobrazení a posun času aktuálně přehrávané skladby. Tato komponenta je až na implementační detaily v prezentační komponentě totožná pro mobilní i webovou aplikaci. Její implementaci jsem proto rozložil do 4 úrovní, které následně představím.

[43] Toshiba Link – https://en.wikipedia.org/wiki/TOSLINK
[44] Sony/Philips Digital Interface – https://en.wikipedia.org/wiki/S/PDIF
[45] Count Lines of Code – https://www.npmjs.com/package/cloc

Hook

Koncept React Hooks byl popsán v kapitole 2.7. Použití Hooks funkcí je vhodné pro změnu hlasitosti (volume) a posun přehrávání (seek). Chování obou prvků je totožné v mobilní i webové aplikaci. Funkci `useVolume` zde nebudu uvádět, protože je velmi jednoduchá a implementuje Fuzzy logiku na převod úrovně hlasitosti. Zajímavější je funkce `useSeek`, která uživateli zobrazuje čas, po který se aktuální skladba přehrává.

```
export function useSeek(seek, isActive) {
  const [time, setTime] = useState(seek);
  let interval = null;

  useEffect(() => {
    if (!isActive) return clearInterval(interval);

    interval = setInterval(() => {
      setTime(time + 1);
    }, 1000);

    return () => { clearInterval(interval); };
  });

  return [time, setTime];
}
```

Funkce `useSeek` každou sekundu inkrementuje proměnnou `time`, pokud tedy není atribut `isActive` pravdivý. Pravdivost atributu `isActive` je definována na základě toho, zda uživatel manipuluje s komponentou posuvníku (slider).

Render Props

Funkce `useSeek` je implementovaná jako `hook` v komponentě `Seek`, která vytváří abstrakci nad manipulací s posuvníkem.

```
export function Seek({isPlaying, seek, children}) {
  const [isDragging, setDragging] = useState(false);
  const [time, setTime] = useSeek(seek, isPlaying && !isDragging);

  const onChange = (value) => {
    if (!isDragging) setDragging(true);
    setTime(value);
  };

  const onDragEnd = () => {
    setDragging(false);
    if (time !== seek) setTime(time);
  };

  const onDragStart = () => { setDragging(true); };

  return children({value: time, onChange, onDragStart, onDragEnd});
}
```

Prezentační komponenta

Tato komponenta je jiná pro každou aplikaci, ale používá předchozí abstrakce. Zjednodušená implementace vypadá následovně:

```
export const SeekControl = ({isPlaying, seek}) => (
    <Seek seek={seek} isPlaying={isPlaying}>
      { (props) => ( <Slider {...props} /> ) }
    </Seek>
);
```

Kontejnerová komponenta

Prezentační komponenta je následně připojená k Redux skladu pomocí kontejnerové funkce `SeekContainer`, která je opět stejná pro obě aplikace.

```
const SeekContainer = ({children, seek, playingId, ...rest}) => (
    React.cloneElement(children, {
      key: seek + playingId,
      seek,
      ...rest
  })
);
```

Důležitý atribut je `key`, který při změně vykreslí úplně novou komponentu. To je důležité, protože komponenta udržuje svůj lokální stav, který je nutné přepsat zvenku. Tahle situace nastává, buď při změně skladby, nebo při změně posunu času skladby od jiného uživatele.

6.7 Nalezení serveru na lokální síti

Tato kapitola vychází z teorie o Zeroconf (kapitola 3.2). Na Raspberry Pi jsem zprovoznil službu Avahi, která nabízí informace o službách, které běží na zařízení. Avahi také rozesílá jednu speciální službu `_workstation_.tcp`, která nese informace o zařízení jako MAC, IPv4 a IPv6 adresa. Nastavení Avahi je realizováno v souboru `/etc/avahi/avahi-daemon.conf` a definování služby v souboru `/etc/avahi/services/XXX.service`, kde `XXX` je jméno služby. Definice služby je založena na XML a je možné v jednom souboru definováno i více služeb současně. Ukázku nastavení služby prostřednictvím Avahi lze nalézt v knize Raspberry Pi Cookbook [22], která dále popisuje i způsob aplikování změn.

Kontrola nastavení Avahi na Raspberry Pi z externích zařízení:

- **Linux** – Nabízí se programy avahi-discover[46] nebo avahi-browse[47], kterými lze nalézt služby.
- **iOS** – Aplikace Discovery[48], která nachází lokální Bonjour služby. Avahi a Bonjour však obě implementují stejné protokoly a jejich použití je tak zaměnitelné.

[46] avahi-discover – https://linux.die.net/man/1/avahi-discover
[47] avahi-browse – https://linux.die.net/man/1/avahi-browse
[48] Discovery – https://itunes.apple.com/us/app/discovery-dns-sd-browser/id305441017

- **Android** – Aplikace NetX Network tools[49], která nabízí sken lokální sítě a informace o všech připojených zařízeních.

Raspberry Pi má z počátku jméno počítače `raspberrypi`. Toto jméno lze změnit v souborech `/etc/hostname` a `/etc/hosts` [22].

Implementace Zeroconf v mobilních zařízeních

- **iOS** – Pro systém iOS není třeba žádná implementace ze strany vývojáře, protože Apple instaluje Bonjour do všech jeho zařízení. Apple tak překládá lokální doménové jména v jádru systému, se kterým aplikaci komunikují. Není tedy žádný problém v jakémkoliv prohlížeči pracovat s lokálními doménami.
- **Android** – Android na druhou stranu neimplementuje Bonjour ani Avahi. Otestoval jsem knihovny react-native-dnssd[50] a react-native-zeroconf[51], avšak ani jedna knihovna nedokázala nalézt žádné služby, které byly nabízené na lokální síti.

Řešení by mohla představovat knihovna mDNSResponder[52], která je zatím pouze pro Android a neexistuje tak její implementace pro React Native. Avšak knihovna react-native-zeroconf se již zabývá implementaci této knihovny namísto modulů NsdManager.

Aby se mobilní aplikace mohla připojit k lokálnímu zařízení jinak než doménovým jménem, implementoval jsem na serveru proces, který do webové aplikace vrátí IPv4 adresu serveru, na kterou se poté uživatel mobilní aplikace může připojit.

Pokud se webová aplikace připojuje na server pomocí lokálního doménového jména, je webové aplikaci vrácena lokální IPv4 adresa serveru, pokud se však nejedná o lokální doménové jméno je vrácena veřejná IPv4 adresa serveru.

6.8 Typy spojení mezi klientem a serverem

Spojení mezi klientem a serverem je ustanoveno pomocí knihovny Socket.IO (kapitola 2.10). Systém rozlišuje 3 typy spojení:

- `Stable` – Typ spojení, ve kterém se nachází všichni klienti, kteří pracují se systémem. Potřebná data jsou tak zasílána mezi klienty. V tomto spojení je podpora pro opětovné ustanovení spojení.
- `Discover` – Tento typ spojení slouží výhradně pro zjištění zda je daný server dostupný. Klientovi nejsou zaslána žádná aplikační data a spojení nepodporuje opětovné připojení v případě selhání.
- `Unspecified` – Spojení, které existuje pouze jako „fallback“, pokud předchozí spojení neuvedou správně jejich typ.

Klient podporuje nastavení počátečního spojení, které se při startu otestuje. Webová aplikace se k dostupnému připojení dokáže ihned připojit. Nastavení počátečních spojení lze provést v souboru `shared/connection/constants.js` v poli `DEFAULT_HOSTS`. Podporovaný

[49] NetX – https://play.google.com/store/apps/details?id=com.tools.netgel.netx
[50] react-native-dnssd – https://github.com/kopera/react-native-dnssd
[51] react-native-zeroconf – https://github.com/balthazar/react-native-zeroconf
[52] mDNSResponder – https://github.com/andriydruk/RxDNSSD

je formát `{domain|IPv4}<:{port}>`, pokud `port` není definován, použije se číslo portu z konstanty `DEFAULT_PORT_MIN`. Spojení lze provozovat na protokolu `http` i `https`. Tento protokol lze specifikovat v témže souboru pomocí konstanty `PROTOCOL`.

Aplikace operuje buď ve veřejném, nebo v privátním režimu (kapitola 5.3). Tento režim je relevantní zejména pro webovou aplikaci. Mobilní aplikace, pokud je připojená k serveru, operuje vždy ve veřejném režimu. Proto každé spojení musí nést i informaci o tom, v jakém režimu se nachází, aby data byly distribuována správným aplikacím. Současně webová aplikace může mezi těmito režimy přepínat a server musí konkrétní nastavení na klientovi respektovat.

Klient

Klient navazuje, vytváří a testuje spojeni se serverem. Vytvoření instance schránky a identifikace o jaký typ spojení se jedná, je implementováno v rámci `query.handshake` objektu knihovny Socket.IO.

Navázání spojení je implementováno v rámci Redux middlewaru. Připojení je vyvoláno zavoláním akce `connectionsConnect(connectionId)`, kde parametr `connectionId` představuje identifikátor spojení, které je uložené ve stavu aplikace jako aktivní.

```
socket = createSocket(host, port, {
    query: {
        [HANDSHAKE_CONNECTION_TYPE]: HANDSHAKE_CONNECTION_STABLE,
        [HANDSHAKE_INTERACTIONS]: selectPlaybackInteractions(store.getState()),
    },
});
```

Spojení definuje o jaký typ se jedná pomocí atributu `HANDSHAKE_INTERACTIONS`. V případě, že klient změní režim, pošle se na server informaci o této změně pomocí funkce `connectionUpdateHandshake(name, value)`, kde atribut `name` slouží jako klíč v objektu `query.handshake`. Server si ukládá všechny aktivní spojení, a pokud přijde tato akce, upraví si informace o tomto spojení. Pokud server dostane pokyn pro distribuci dat, vždy kontroluje v jakém režimu se dané spojení nachází. Akce pro uložení či smazání skladby jsou poslány všem klientům bez výjimek.

Nalezení serveru je implementováno pomocí knihovny Redux-Saga za pomocí funkcí, které dovolují připojení k externím zdrojům (kapitola 2.6). Zjednodušená implementace vypadá následovně:

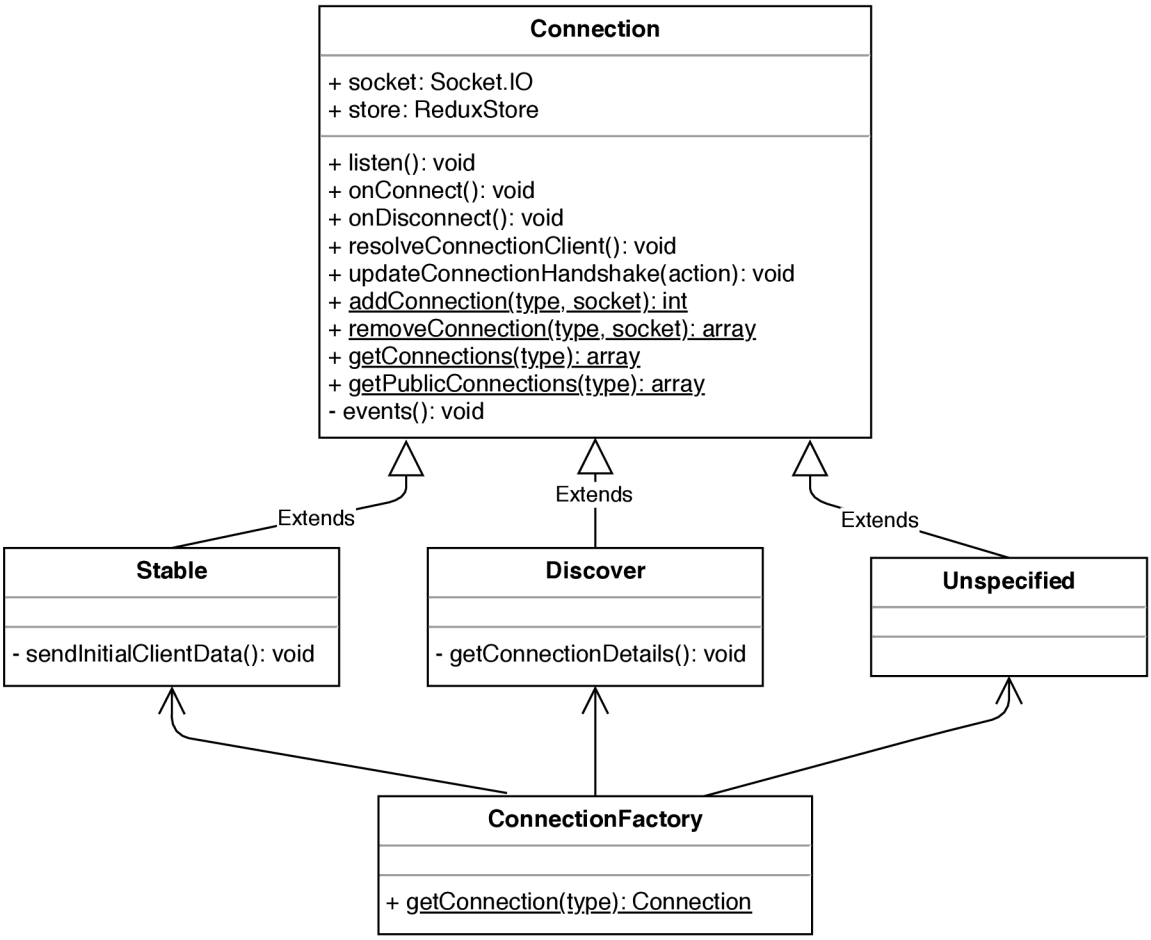
```
function createConnectionTestChannel(socket) {
  return eventChannel((emit) => {
    const handler = (status) => (data) => {
      emit({status, data});
      emit(END);
    };
    socket.on(HANDSHAKE_CONNECTION_DISCOVER, handler(CONNECTION_AVAILABLE));
    socket.on(SOCKET_CONNECT_ERROR, handler(CONNECTION_UNAVAILABLE));
    socket.on(SOCKET_CONNECT_TIMEOUT, handler(CONNECTION_UNAVAILABLE));

    return () => { socket.close(); };
  });
}
```

```
function* connectionTestChannel(socket) {
  const channel = yield call(createConnectionTestChannel, socket);
  return yield take(channel);
}

function connectionTest(id){
  // select connection data from store by its `id`
  const socket = createSocket(host, port, {
    query: {[HANDSHAKE_CONNECTION_TYPE]: HANDSHAKE_CONNECTION_DISCOVER},
    reconnection: false,
  });

  const {status, data} = yield call(connectionTestChannel, socket);
  // process connection related data ...
}
```

Události `SOCKET_CONNECT_ERROR` a `SOCKET_CONNECT_TIMEOUT` pochází z knihovny Socket.IO a indikují neúspěšné ověření dostupnosti serveru. Událost `HANDSHAKE_CONNECTION_DISCOVER` definuje úspěšné nalezení serveru a tedy potvrzení jeho dostupnosti. Argument `data`, který je získán ze serveru může obsahovat například jméno serveru, jeho lokální či veřejnou IPv4 adresu a další relevantní informace o serveru.

Server

Server reaguje na příchozí spojení od klienta. Rozmanitost spojení jsem se na serveru rozhodl implementovat pomocí tovární metody, která je zobrazena na obrázku 6.2. Navázání spojení je implementováno v Redux middlewaru. Server však při startu musí načíst skladby a příjem spojení je odložen. Instanciance Socket.IO serveru je povolena až po zavolání akce `connectionAllow()`.



Obrázek 6.2: Diagram tříd pro realizaci spojení na serveru. Parametr `type` v třídě `ConnectionFactory` odpovídá hodnotě `query.handshake[HANDSHAKE_CONNECTION_TYPE]`. Spojení typu `Unspecified` okamžitě odpojí klienta a žádná data tak nejsou zaslána.

6.9 Real-time komunikace

Po ustanovení spojení mezi klientem se serverem přichází na řadu komunikace. Server vždy komunikuje se všemi připojenými klienty, avšak ne všem zasílá všechna data. Klienti ve veřejném režimu mezi sebou prostřednictvím serveru komunikují a ovládají přehrávání na serveru. Celá implementace komunikace stojí na třech základních principech, které jsem si vytyčil.

Komunikace musí být:

- **Flexibilní** – Protože klient nabízí komunikaci ve veřejném a privátním režimu, musí být veškeré akce kompatibilní s oběma režimy. To znamená, že klient musí vědět, zda může akci zaslat serveru nebo ne.
- **Robustní** – Implementace by neměla spoléhat na konvence, které každá akce v aplikaci musí dodržovat. Současně musí být implementace použitelná pro naprosto libovolnou akci.

- **Rychlá** – Zasílání akci by nemělo trvat více než je nutné. Tento bod je prakticky nejednoduší na řešení, protože se nabízí implementace v rámci knihovny Redux-Saga, nebo middlewaru knihovny Redux. Jelikož Redux-Saga je jenom nadstavba nad middlewarem, bude proto efektivnější použít pouze middleware, který nabízí i větší flexibilitu.

Flexibilita a robustnost implementace je založená na principu funkce vyššího řádu. Definoval jsem proto několik takových funkcí. Nejdůležitější funkce je `actionDispatchBroadcast`.

Implementace `actionDispatchBroadcast`

```
export const actionDispatchBroadcast = (action) => ({
  type: ACTION_DISPATCH_BROADCAST,
  action,
});
```

Použití `actionDispatchBroadcast`

```
dispatch(actionDispatchBroadcast(playlistsRemove(playlistId)))
```

Použití je demonstrováno na konkrétní funkci `playlistsRemove`, avšak může se jednat o naprosto libovolnou funkci, která vytváří akci.

Zpracování `actionDispatchBroadcast`

```
case ACTION_DISPATCH_BROADCAST: {
    const finalAction = resolveSocketAction(action.action, store);

    store.dispatch(finalAction);

    const isPublicMode = socketEmitAllowed(store);

    return isPublicMode && socket.emit(ACTION_BROADCAST, finalAction);
}
```

Funkce `resolveSocketAction` je pouze další úroveň abstrakce pro přidání informace o spojení dané akci. Tuto abstrakci využívá akce pro tvorbu playlistů. Tato abstrakce by šla ušetřit, přičemž by však playlist musel mít při vytvoření i informace, které se spíš než jeho, týkají aplikace. Finální akce je předána dané aplikaci pomocí `store.dispatch`, a pokud se klient nachází ve veřejném režimu, je akce poslána na server.

Zpracování akce na serveru

```
this.socket.on(ACTION_BROADCAST, (action) => {
    Connection.getConnections(Connection.STABLE).forEach((socket) => {
        const interactions = socket.handshake.query[HANDSHAKE_INTERACTIONS];
        const isPublicSocket = interactions === PLAYBACK_INTERACTIONS_PUBLIC;
        const isNotSender = this.socket.id !== socket.id;

        if (isPublicSocket && isNotSender) socket.emit(ACTION_BROADCAST, action);
    });

  this.store.dispatch(action);
});
```

Knihovna Socket.IO implementuje broadcast funkci, která všem klientům zašle danou zprávu, avšak kvůli podpoře režimů jsem spíše než broadcast, potřeboval multicast. Další způsob, jak tento problém řešit, je využití místností, které Socket.IO nabízí. Protože však existují akce, které jsou sdíleny všem klientům, rozhodl jsem se místnosti zatím nepoužít. Do budoucna však mají místnosti potenciál nahradit stávající řešení multicastu a funkce `connectionUpdateHandshake`.

Zpracování akce u klienta

```
socket.on(ACTION_BROADCAST, (action) => {
    store.dispatch(action);
});
```

Klient pouze obdrženou akci promítne do stavu aplikace, čímž je dokončeno sdílení informací v reálném čase mezi uživateli. Zpracování této akce je opět provedeno v middlewaru knihovny Redux.

Závěr

Implementace splnila vytyčené cíle a libovolná akce, může být distribuovaná mezi ostatní klienty. Současně jsou všechny akce čisté a neobsahují žádné příznaky, kterými by se měnilo jejich chování, a tím pádem se zachovala jejich jednoduchá testovatelnost. Z hlediska implementace rozlišení spojení, je prostor pro přechod na způsob místností, jež jsou nadstavbou knihovny Socket.IO.

6.10 Přenos hudby na server

Přenos hudby na server je realizován pomocí knihovny rn-fetch-blob[53], která umožňuje stream protokolem WebSocket a upload protokolem HTTP. Přenos skladby na server může probíhat ve 3 režimech, kdy se režimy liší v tom, jaké akce jsou provedeny po přenosu skladby na server:

- **`Upload`** – Žádná akce není provedena.
- **`UploadQueue`** – Skladba je přidána do fronty.
- **`UploadPlay`** – Skladba je přehrána.

Stream

Soubor je čtený po částech knihovní funkcí `readStream`. Funkce dovoluje nastavit kódování, velikost bufferu a interval, po kterém jsou části poslány přes React Native bridge aplikaci. Výběr kódování souboru musí být podporován v Node.js a současně musí být kompatibilní s protokolem WebSocket a knihovnou Socket.IO. Takové kódování poté jsou `UTF8`, `BASE64` a `ASCII`. Podle dokumentace knihovny rn-fetch-blob má kódování `ASCII` největší režii a je tedy velmi neefektivní. Rozdíly mezi kódováními `UTF8` a `BASE64` jsou zanedbatelné. Další faktor ovlivňující efektivitu přenosu je velikost bufferu, který je přenášen. Velikost bufferu přímo udává počet přenosů přes React Native bridge do aplikace. Pokud však přes bridge proudí v krátkých intervalech mnoho dat, aplikace přestává reagovat na akce provedené

[53] rn-fetch-blob – https://github.com/joltup/rn-fetch-blob

uživatelem a uživatelské rozhraní „zamrzá“. Efekt zamrznutí lze částečně vyřešit snížením frekvence zasílání dat přes bridge, avšak tento krok negativně ovlivní rychlost přenosu a spíš než jako řešení, působí jako „hack“. Pro podporu streamu souboru jsem vytvořil nový typ spojení `Stream`.

- **`Stream`** – Motivace pro vytvoření nového typu spojení je zisk lepší kontroly nad tokem dat na server. Nové spojení umožňuje limitovat počet navázaných spojení, zrušení spojení nebo i zakázání vytvoření spojení pro určité klienty na základně IP adresy.

Přenos souboru z klienta probíhá na základě fronty, do které uživatel přidává skladby, které nejsou nahrané na serveru. První přidaná skladba do fronty spustí proces, který vytvoří `Stream` spojení a zavolá funkci pro zaslání skladby na server. Tato funkce využívá koncové rekurze[54], která bere přidané skladby z fronty a sekvenčně streamuje skladby na server. Tímto je využito jednoho spojení pro přenos více souborů. Více souboru nelze přenášet současně, protože by tím aplikace byla nepoužitelná.

Důvod použití streamu, jako způsobu přenosu dat, je umožnit přehrávání skladby již po přenosu prvních několika kilobajtů dat. Taková optimalizace však vyžaduje analýzu hudebních formátů a zabezpečení, že aktuálně přenášená skladba nebude moci být uživatelem přetáčena.

Příjem souboru na serveru je realizován třídou `Stream`, která registruje `callback` funkce na události zaslané klientem. Data jsou zapisována funkcí `createWriteStream`, která pochází z `fs` modulu.

HTTP

Přenos souborů pomocí HTTP protokolu je realizován knihovní funkcí `fetch`. Tato funkce realizuje potřebné kódování mimo vlákno JavaScript aplikace a uživatelské rozhraní tak není blokováno. Při použití HTTP protokolu není třeba vytvářet žádné další spojení. Režimy přenosu jsou implementovány pomocí URL[55] parametrů (query string).

Přenos souboru z klienta je v principu stejný jako u přenosu pomocí streamu, proto lze velká část logiky jednoduše převzít.

Příjem souboru na serveru je realizován na základě REST[56] API. Jelikož tvorba REST API v knihovně Express[57] nepočítá s asynchronními `callbacky` použil jsem knihovnu express-async-handler[58], která tuto funkcionalitu přidává.

Je žádoucí, aby klient začal přehrávat skladbu ihned potom, co je uživatelem vybrána. REST API proto podporuje dotazy pro částečný obsah[59] souboru v hlavičce odpovědi.

[54] Koncová rekurze – https://medium.com/functional-javascript/recursion-282a6abbf3c5
[55] Uniform Resource Locator – https://en.wikipedia.org/wiki/URL
[56] Representational State Transfer – Definuje pravidla a způsoby vytváření webových služeb.
[57] Express – https://expressjs.com
[58] express-async-handler – https://github.com/Abazhenov/express-async-handler
[59] Range Request – https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Range

Tabulka 6.3: REST API pro správu skladeb. Metoda POST a DELETE je doprovázena zasláním dané informace všem připojeným uživatelům.

HTTP metoda	**Endpoint**	**Popis**
GET	/tracks	Vrátí všechny skladby v JSON formátu.
GET	/tracks/{id}	Zpřístupní skladbu pro použití přehrávačem.
GET	/tracks/{id}/download	Začne stahovat soubor v prohlížeči.
POST	/tracks	Vytvoří skladbu na serveru.
DELETE	/tracks/{id}	Smaže skladbu ze serveru.

Závěr

Po implementaci obou variant a testování jejich výkonu, se přenos souborů pomocí streamu projevil jako 3× pomalejší než přenos souborů pomocí protokolu HTTP. Vytvořené REST API je navržené tak, aby umožnilo přenos i souborů z webové aplikace.

Kapitola 7

Testování a nasazení systému

Proces testování a nasazení systému je automatizován prostřednictvím GitLab CI[1]. Tento proces je rozdělen do 4 etap (stages).

- **`Build`** – Etapa testuje, zda je aplikace přeložitelná. Přeložitelnost aplikace je testována příkazem **`npm run build`**, který je stejný pro všechny aplikace. Aktuálně je automaticky testováno přeložení serverové a webové aplikace.
- **`Test`** – Etapa, která testuje kód. Testování je založeno na knihovně Jest[2] a testy jsou spuštěny příkazem **`npm run test`**. Aktuálně jsou testovány klíčové moduly aplikace v **`shared`** adresáři.
- **`Lint`** – Etapa kontrolující kvalitu kódu. Kvalita kódu je testována nástrojem ESLint[3] pro všechny adresáře obsahující aplikační kód. Testování kvality v daném adresáři je spuštěno příkazem **`npm run lint`**.
- **`Deploy`** – Etapa nasazení systému je dostupná pro webovou a serverovou aplikaci. Aplikace jsou nasazovány na servery Heroku[4].

7.1 Webová a serverová aplikace

Obě aplikace jsou nasazeny prostřednictvím bezplatného plánu[5] na Heroku. Protože je repozitář organizován jako monorepo je proces nasazení obou aplikací výrazněji obtížnější na automatizaci. Heroku dovoluje vytvořit aplikace, ale v každé aplikaci lze vytvořit pouze 1 HTTP server. Proto je nutné mít na Heroku vytvořené 2 aplikace, avšak pro „neověřený účet“[6] není povolen překlad více než 1 aplikace současně. Přenos aplikace z Gitlab CI na Heroku je realizován pomocí Dpl[7]. Pro přenos je vyžadován Heroku API klíč, který je na GitLabu uložen v bezpečném prostředí. Problém monorepa spočívá ve specifikaci aplikace, která se má po přenosu na Heroku překládat.

[1] GitLab CI – https://docs.gitlab.com/ee/ci
[2] Jest – https://jestjs.io
[3] ESLint – https://eslint.org
[4] Heroku – https://www.heroku.com
[5] Heroku ceník – https://www.heroku.com/pricing
[6] Neověřený účet je typ účtu, který nemá uložené informace o kreditní kartě.
[7] Dpl – https://docs.gitlab.com/ee/ci/examples/deployment

Nasazení je tedy zautomatizováno pro jednu aplikaci. Pokud je potřeba nasadit i druhá aplikace je nutné v kořenovém adresáři upravit soubor `package.json`, kde je nutná změna adresáře web/server a dále je třeba změnit cíl nasazení aplikace v souboru `.gitlab-ci.yml`.

Testování

Na serveru běží hlavně Socket.IO server, jehož testování je prováděno interakcí s aplikací. Server ovšem nabízí i REST API, které bylo testováno službou Postman[8].

Interakce s webovou aplikací byly testovány desítkami lidí, kteří k ní měli přístup prostřednictvím Heroku. Nejdůležitější aspekt, který působil problém, byl veřejný režim, ve kterém byla aplikace nasazena. Uživatelé nerozuměli tomu, proč jim nehraje žádná hudba. Proto jsem se rozhodl nasadit aplikaci v soukromém režimu, aby každý uživatel mohl přehrávat hudbu ihned po jejím zapnutí. Pro použití aplikace jako ovladače serveru, je nutné přepnutí do veřejného režimu.

Nasazení serveru pro webovou a serverovou aplikaci

Obě aplikace běží nad knihovnou Express. Webová aplikace je realizována jako SPA[9], a proto je potřeba realizovat veškerou navigaci na straně klienta. Tento typ navigace nabízí middleware knihovna pro Express express-history-api-fallback[10], která veškeré požadavaky na aplikaci přesměrovává na soubor `index.html`.

Pro nasazení obou aplikací na Heroku je nutné umožnit spuštění aplikace pod portem, který jí Heroku přidělí. Port pro serverovou i webovou aplikaci je tedy definován následovně:

```
const port = process.env.PORT || SOME_INTERNAL_PORT;
```

7.2 Mobilní aplikace

Aplikace je vyvíjena pouze pro systém Android, protože nedisponuji Apple zařízením pro překlad aplikace na systém iOS. Přesto jsou použité knihovny kompatibilní i se systémem iOS. Aby aplikaci bylo možné nasadit na Google Play[11], je nutné založit účet v Google Play Console[12]. Založení účtu je podmíněno jednorázovým poplatkem 25 $. Z webového prostředí lze poté nahrát přeložené `*.apk` soubory přímo na Google Play.

Testování

Aplikace je aktuálně dostupná v Alfa verzi a stáhnout ji tak mohou pouze uživatelé, kteří jsou jejími testery. Do testování aplikace se zapojilo celkem 6 uživatelů. V prvních verzích aplikace jsem tak zaznamenal několik hlášení o špatném výkonu aplikace, protože uživatelé používali větší množství skladeb než-li já při vývoji. Hlavní problém byl na stránce s hudbou a použitím nevhodných komponent, které vykreslovaly všechny skladby. Nedostatky se podařilo odstranit využitím virtualizovaných komponent, které umí pracovat s neměnnými daty. Virtualizované komponenty také vykreslují pouze tolik elementů, aby jimi zaplnily obrazovku. S tímto výkonnostním problémem souvisela i knihovna React Navigation, která

[8] Postman – https://www.getpostman.com
[9] SPA – Single Page Application, více viz https://en.wikipedia.org/wiki/Single-page_application
[10] express-history-api-fallback – https://www.npmjs.com/package/express-history-api-fallback
[11] Google Play – https://play.google.com/store
[12] Google Play Console – https://developer.android.com/distribute/console

při přechodu na jinou stránku neruší komponenty z jiných stránek, což ve spojení se nevirtualizovanou komponentou učinilo aplikaci s více skladbami nepoužitelnou.

7.3 Raspberry Pi

Počítač Raspberry Pi je použit pro nasazení serverové i webové aplikace. Raspberry Pi jsem zakoupil ze svých zdrojů a cena všech komponent se vyšplhala na 2962 Kč.

Tabulka 7.1: Soupis zakoupených komponent pro Raspberry Pi.

Komponenta	**Cena v Kč**
Raspberry Pi	1 096
4"rezistivní IPS displej	855
MicroSDHC karta 32GB	730
Pouzdro	198
Pasivní chladiče	83

V kapitole o Raspberry Pi 2.9 je obrázek počítače bez displeje. Dotykový displej jsem se rozhodl přikoupit, abych vytvořil samonosné zařízení, které je schopné pracovat nezávisle na externí klientské aplikaci.

Obrázek 7.1: Raspberry Pi s webovou aplikací.

Přenos přeložených souborů na zařízení je aktuálně realizován přes protokol SSH. Jejich spuštění však musí být realizováno robustněji než pouhým zavoláním příkazu `npm run start`, který spustí HTTP servery. Důvody pro robustnější řešení jsou monitorování prostředků využívaných aplikací a znovuspuštění aplikace, pokud by došlo k jejímu pádu.

Dva nejznámější nástroje pro Node.js jsou forever[13] a pm2[14]. Prvně jsem vyzkoušel forever, ale práce s ním nebyla tak příjemná jako s nástrojem pm2, který nabízí více funkcí, které se intuitivněji používají. Konfigurace pm2 se nachází v souboru `bootstrap/pm2.json`.

Po konfiguraci pm2 a spuštění aplikace je třeba zařídit, aby aplikace byla dostupná na portu `80` s doménou `player.local`. Nastavení lokální domény je a jména počítače je rozebráno v kapitole 6.7. Dostupnost aplikace na portu 80 lze zařídit následujícími způsoby:

- **Privilegovaný režim** – Aplikace může být spuštěna na portu 80, pokud je spuštěna v privilegovaném režimu (příkaz `sudo`). Tento způsob však není optimální, protože představuje bezpečnostní rizika.
- **Nginx Reverse Proxy** – Aplikace je spuštěna na libovolném portu bez speciálních oprávnění. Nginx Reverse Proxy nabízí přesměrování komunikace z portu 80 na libovolný port. Konfigurace a spuštění Nginx se nachází v souboru `bootstrap/Nginx.md`.

Spuštění aplikace po startu zařízení

Při startu Raspberry Pi se musí pro úspěšný start aplikace spustit: Avahi daemon, Nginx, pm2 (server pro serverovou a webovou aplikaci a Chromium prohlížeč s adresou `player.local`. Avahi daemon a Nginx jsou spuštěny v rámci procesu `systemd`. Spuštění aplikaci přes pm2 jsem ještě zabalil do skriptu `berry.sh`, abych abstrahoval specifické pm2 funkce. Výsledný skript je již vhodný pro spuštění aplikace při startu systému.

Možnosti spuštění skriptu `berry.sh`:

- **systemd** – Nabízí poměrně jednoduchou konfiguraci, kterou jsem vytvořil v souboru `bootstrap/berry.service`. Hlavní problém použití tohoto způsobu je v detekci inicializace grafického prostředí a spuštění prohlížeče.
- **SysVinit** – Je předchůdce služby systemd, který spouští služby sekvenčně při startu počítače. Nastavení tohoto způsobu je popsáno v knize Raspberry Pi Cookbook [22]. Nevýhoda tohoto způsob je opět detekce inicializace grafického prostředí. Tento způsob je také označován jako za zastaralý.
- **LXDE[15] autostart** – Operační systém Raspbian je modifikovanou verzí prostředí LXDE. LXDE nabízí soubor `/etc/xdg/lxsession/LXDE-pi/autostart`, který je spuštěn po tom, co je inicializováno grafické prostředí. Příkaz pro spuštění aplikace v tomto souboru je `@sh /home/pi/Desktop/berry/bootstrap/berry.sh start`.

Závěr

Po testu všech možností jsem použil spuštění aplikace pomocí souboru `autostart`. Systemd a SysVinit jsou vhodnější pro spuštění programů, které nevyžadují start aplikace s uživatelským rozhraním.

[13] forever – https://github.com/foreverjs/forever
[14] pm2 – https://github.com/Unitech/PM2
[15] Lightweight X11 Desktop Environment – https://en.wikipedia.org/wiki/LXDE

Kapitola 8

Závěr

Cílem této práce bylo vytvořit hudební systém, který umožní uživatelům organizovat a přehrávat hudbu. Nad rámec zadání je řešena komunikace v reálném čase mezi klienty a vytvořena webová aplikace.

- **Serverová aplikace** – Nachází využití v podobě přenosného přehrávače nebo online serveru pro sdílení hudby. Serverová aplikace je otestována na operačním systému Linux v distribucích Raspbian a Ubuntu 18.04 a operačním systému Windows 8.
- **Webová aplikace** – Nabízí soukromý a veřejný režim, ve kterém je schopna fungovat. Ve veřejném režimu slouží pouze jako uživatelské rozhraní serveru, kde jsou veškeré akce sdíleny v reálném čase mezi ostatními uživateli, kteří se také nachází ve veřejném režimu. Soukromý režim na druhou stranu nabízí odpojení od vnějšího hudebního světa, kde si může uživatel nerušeně organizovat a přehrávat hudbu dle vlastního uvážení. Provedené akce v soukromém režimu nejsou sdíleny s ostatními uživateli. Webová aplikace je otestována v prohlížečích Chrome, Firefox a Edge.
- **Mobilní aplikace** – Funguje jako offline přehrávač nezávisle na serveru. Pokud se uživatel rozhodne připojit aplikaci k serveru, stane se z mobilní aplikace uživatelské rozhraní serveru a uživateli je tak umožněna organizace hudby s ostatními uživateli. První publikace mobilní aplikace na Google Play byla uskutečněna 15.3.2019 a od té doby aplikace získala řadu aktualizací.

V budoucnu může aplikace dostat tzv. „Párty mód“, ve kterém uživatelé dávají své hlasy jednotlivým skladbám, které se následně řadí ve frontě podle počtu svých hlasů. Vývoj se také může ubírat více komerčním směrem ve formě zpoplatnění přehrávání skladeb na serveru. Server by pak sloužil jako moderní Jukebox.

Literatura

[1] Andrzejewski, P.: The optimistic UI with React. *Medium.com*, Listopad 2018.

[2] Arrachequesne, D.; Little, E.: Socket.IO documentation. [Online; navštíveno 07.05.2019].
URL https://socket.io/docs/

[3] Banks, A.; Porcello, E.: *Learning React*. O'Reilly Media, Inc., Květen 2017, ISBN 978-1-491-95462-1.

[4] Belshe, M.; Peon, R.; M. Thomson, E.: Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, Září 2018, doi:10.17487/RFC7540.

[5] Buddhika, D.: How Immutable Data Structures (E.g. Immutable.js) are Optimized. *Medium.com*, Únor 2018.

[6] Cheshire, S.; Aboba, B.; Guttman, E.: Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927, Květen 2005, doi:10.17487/RFC3927.

[7] Cheshire, S.; Krochmal, M.: DNS-Based Service Discovery. RFC 6763, Únor 2013, doi:10.17487/RFC6763.

[8] Cheshire, S.; Krochmal, M.: Multicast DNS. RFC 6762, Únor 2013, doi:10.17487/RFC6762.

[9] Chesire, S.; Steinberg, D.: *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., Prosinec 2005, ISBN 0-596-10100-7.

[10] Chung, B.: Everything you need to know about skeleton screens. *Medium.com*, Říjen 2018.

[11] contributors, W.: Monorepo. [Online; navštíveno 05.05.2019].
URL https://en.wikipedia.org/wiki/Monorepo

[12] Deveria, A.: Can I use. [Online; navštíveno 06.05.2019].
URL https://caniuse.com/#search=WebSocket

[13] Eisenman, B.: *Learning React Native*. O'Reilly Media, Inc., Prosinec 2016, ISBN 978-1-491-92900-1.

[14] Fette, I.; Melnikov, A.: The WebSocket Protocol. RFC 6455, Prosinec 2011, doi:10.17487/RFC6455.

[15] Frachet, M.: Understanding the React Native bridge concept. *Medium.com*, Červen 2018.

[16] Gelman, I.; Dinkevich, B.: *The Complete Redux Book*. Leanpub, Duben 2018, ISBN 978-965-92642-0-9.

[17] Hsu, J.: The Anatomy Of A React & Redux Module (Applying The Three Rules). [Online; navštíveno 06.05.2019].
URL https://jaysoo.ca/2016/02/28/applying-code-organization-rules-to-concrete-redux-code

[18] The Internet of Things. [Online; navštíveno 08.05.2019].
URL https://www.ietf.org/topics/iot/

[19] Lin, C.-C.: Android VS. iOS: Compare 20 UI Components & Patterns (Part 1). *Medium.com*, Prosinec 2017.

[20] Loreto, S.; Saint-Andre, P.; Salsano, S.; aj.: Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. RFC 6202, Duben 2011, doi:10.17487/RFC6202.

[21] McManus, P.: Bootstrapping WebSockets with HTTP/2. RFC 8441, Září 2018, doi:10.17487/RFC8441.

[22] Monk, S.: *Raspberry Pi Cookbook*. O'Reilly Media, Inc., Prosinec 2014, ISBN 978-1-449-36522-6.

[23] React Native Runtime. [Online; navštíveno 07.05.2019].
URL https://facebook.github.io/react-native/docs/javascript-environment

[24] Richardson, A.: What is Redux-Saga? *Medium.com*, Červen 2017.

[25] Weinschenk, S.: *100 Things Every Designer Needs to Know About People*. New Riders, Květen 2011, ISBN 0-321-76753-5.

Příloha A

Obsah přiloženého DVD

- `bootstrap/` – Návody a skripty pro spuštění serveru na Raspberry Pi.
- `thesis/` – Zdrojové soubory k bakalářské práci včetně výsledného `.pdf` souboru.
- `graphics/` – Obsahuje použitou a vytvořeno grafiku při práci. Grafika se skládá z rámů zařízení, plakátu a videa.
- `mobile/` – Zdrojové soubory mobilní aplikace.
- `server/` – Zdrojové soubory serverové aplikace.
- `shared/` – Zdrojové soubory, které jsou sdíleny mezi aplikacemi.
- `web/` – Zdrojové soubory webové aplikace.

Příloha B

Instalační manuál

Všechny následující instalace vyžadují mít nainstalovaný Node.js[1] v minimální verzi `10.x` včetně nástroje npm[2], který je aktuálně instalován automaticky společně s Node.js.

B.1 Společná instalace pro všechny aplikace

- `/` – Babel od verze 7 představuje koncept monorepo pro repozitáře ve formě konfiguračního souboru `babel.config.js`[3], který se nachází v kořenovém adresáři. Je tedy nutné v kořenovém adresáři spustit příkaz `npm run install`.
- `shared/` – Všechny aplikace používají zdrojové soubory z adresáře `shared`, proto je nutné se přesunout do tohoto balíčku a spustit příkaz `npm run install`.

Potom, co jsou instalovány sdílené balíčky, je možné přejít k instalaci balíčku pro jednotlivé aplikace. Instalace ve všech aplikacích je provedena pomocí příkazu `npm run install` v příslušných adresářích dané aplikace.

B.2 Webová aplikace

Pro vývoj aplikace je dobré instalovat následující rozšíření do prohlížeče Chrome:

- `Redux DevTools`[4] – Debug nástroj pro vizualizaci Redux stavu aplikace.
- `React Developer Tools`[5] – Debug nástroj pro React komponenty.

B.3 Mobilní aplikace

Po instalaci balíčků v mobilní aplikace je nutné provést příkaz `react-native link`, který linkuje nativní kód instalovaných knihoven s mobilní aplikací.

[1] Node.js – https://nodejs.org/en/download
[2] npm – https://www.npmjs.com/get-npm
[3] Babel monorepo – https://babeljs.io/docs/en/config-files#monorepos
[4] https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd
[5] https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi

Externí aplikace důležité pro vývoj

- `react-native-debugger`[6] – Nabízí debug a vizualizaci Redux stavu, konzoly pro výpisy z aplikace, profilování aplikace a další užitečné možnosti.
- `Android Studio`[7] – Po instalaci tohoto IDE, je nutné doinstalovat konkrétní zařízení do emulátoru, na kterém později bude spuštěna mobilní aplikace.

Po instalaci Android Studia je dále nutné přidat do souboru `~/.bashrc` následující cesty:

```
export ANDROID_HOME=$HOME/Android/Sdk
export PATH=$PATH:$ANDROID_HOME/tools
export PATH=$PATH:$ANDROID_HOME/emulator
export PATH=$PATH:$ANDROID_HOME/tools/bin
export PATH=$PATH:$ANDROID_HOME/platform-tools
```

B.4 Serverová aplikace

Přehrávání hudby na serverové aplikace spoléhá na přítomnost instalovaného mpv[8] přehrávače. Pokud však mpv přehrávač není k dispozici, server bude fungovat mimo přehrávání hudby.

Instalace přehrávače je možná několik způsoby pro Linux i Windows. Následující možnosti jsou mnou ověřené jako funkční.

Linux

- `sudo add-apt-repository ppa:mc3man/mpv-tests` – Přidání balíčku.
- `sudo apt-get install mpv` – Samotná instalace přehrávače. Před instalací je možno spustit příkaz `sudo apt-get update`.

Windows

- https://mpv.srsfckn.biz/mpv-x86_64-20181002.7z – Po překopírování této adresy do prohlížeče, začne stahování archivu přehrávače. Více možností instalace lze nalézt na stránce instalace[9] přehrávače mpv.

Po stažení a extrakci archivu přehrávače, je k dispozici spustitelný soubor `mpv.exe`. Následně je nutné v souboru `server/core/player/mpv/MpvPlayer.js` definovat v objektu `options` následující atributy:

```
const options = {
    ... // Some other default options
    binary: 'C:\\User\\Desktop\\mpv.exe',   // Change path to downloaded .exe file
    socket: '\\\\.\\pipe\\mpvserver',       // Path to socket, default to Unix one
};
```

Konfigurace objektu `options` vychází z knihovny Node-MPV[10].

[6] react-native-debugger – https://github.com/jhen0409/react-native-debugger
[7] Android Studio – https://developer.android.com/studio
[8] mpv – https://mpv.io
[9] mpv instalace – https://mpv.io/installation
[10] Node-MPV – https://github.com/00SteinsGate00/Node-MPV/#usage

Příloha C

Překlad a spuštění aplikací

C.1 Serverová a webová aplikace

Je přeložena příkazem `npm run build`. Spuštění je následně realizováno buď v módu pro vývoj (development), nebo v produkčním módu (production), ve kterém je možné aplikaci nasadit.

- `Vývojový mód` – Spuštěn příkazem `npm run dev`, který nabízí automatickou obnovu aplikace při uložených změnách.
- `Produkční mód` – Spuštěn příkazem `npm run start`. Pro úspěšné dokončení tohoto příkazu je nutné mít danou aplikace přeloženou.

C.2 Mobilní aplikace

Její překlad a spuštění se výrazně odlišuje od předchozích aplikací a výrazně se liší pro vývojový a produkční mód.

Vývojový mód pro Android

Vyžaduje spuštění třech terminálů, ve kterých jsou spuštěné následující příkazy:

- `Emulator` – Spuštění emulátoru lze realizovat příkazem `cd $ANDROID_HOME/emulator && emulator -avd Berry -no-snapshot-load`. Nastavení a pojmenování emulátoru je nastavené prostřednictvím Android Studia.
- `Překlad JavaScriptu` – Spuštěn příkazem `npm run start`.
- `Překlad nativních modulů` – Spuštěn příkazem `react-native run-android`.

Produkční mód pro Android

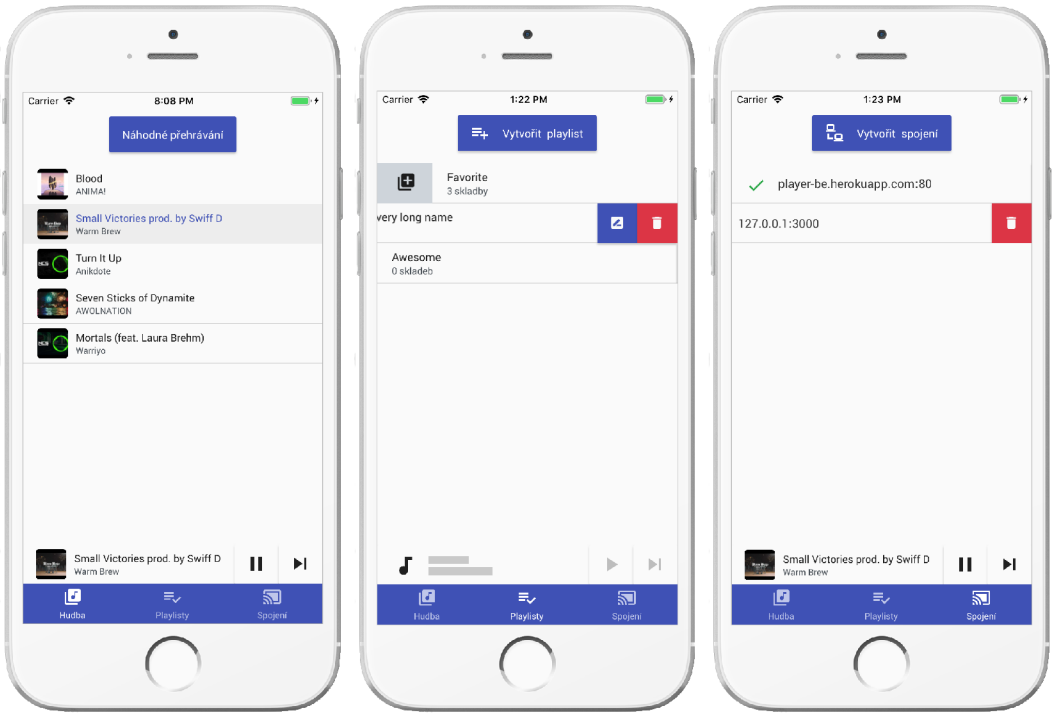
Produkční verze pro Android je přeložena příkazem `./gradlew assembleRelease` v adresáři `mobile/android`. Výsledkem kompilace jsou tři `apk` balíčky, které lze nalézt v adresáři `mobile/android/app/build/outputs/apk/release`. Jednotlivé balíčky je optimalizovány pro různé mobilní telefony se systémem Android. Produkční verze aplikace lze testovat na zařízení příkazem `react-native run-android --variant=release`.

Vývojový mód pro iOS

Překlad vyžaduje prostředí Xcode[1], které nabízí emulátory mobilních zařízení. V tomto prostředí je nutné nastavit nový překládací systém (New Build Sytem) a přidat závislost `JavaScriptCore.framework`[2] do knihoven používaných prostředím xCode.

Před samotným spuštěním překladu je nutné instalovat knihovnu `react-native-swift`[3], kterou vyžaduje knihovna `react-native-track-player`[4].

Spuštění je oproti systému Androidu jednodušší a lze provést buď přímo v prostředí Xcode, nebo spuštěním příkazu `react-native run-ios` v adresáři `mobile/`.



Obrázek C.1: Na prvním obrazovce jsou vždy všechny dostupné skladby. Prostřední obrazovka zobrazuje tři vytvořené playlisty a umožňuje jejich správu. Na poslední obrazovce je demonstrováno připojení aplikace k serveru na doméně `player-be.herokuapp.com`.

[1] xCode – https://developer.apple.com/xcode

[2] JavaScriptCore.framework – http://docs.onemobilesdk.aol.com/ios-ad-sdk/adding-frameworks-xcode.html

[3] react-native-swift – https://www.npmjs.com/package/react-native-swift

[4] Swift pro přehrávač – https://react-native-kit.github.io/react-native-track-player/install

Příloha D

Plakát

Obrázek D.1: Plakát zahrnuje části, ze kterých se skládá hudební systém.