VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

IOS APLIKACE PRO SLEDOVÁNÍ SPORTOVNÍCH VÝKONŮ

SPORTS MANAGER FOR IOS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JAROSLAV HORT**
AUTHOR

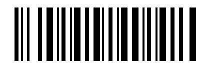
VEDOUCÍ PRÁCE **Ing. MARTIN HRUBÝ, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce



Student: **Hort Jaroslav**
Program: Informační technologie
Název: **iOS aplikace pro sledování sportovních výkonů**
Sports Manager for iOS
Kategorie: Uživatelská rozhraní

Zadání:

1. Prostudujte programování aplikací pro iOS (SwiftUI, CoreData, CloudKit). Prostudujte rozhraní pro sledování pohybu uživatele v zařízení s iOS.
2. Navrhněte aplikaci pro pořizování, ukládání a analyzování sportovních aktivit uživatele. Navrhněte vhodné propojení pouštění hudby při sportovních aktivitách.
3. Aplikaci implementujte. Implementuje jednoduché webové napojení na uživatelská data uložená v CloudKit.
4. Testujte aplikaci se zapojením několika uživatelů.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Hrubý Martin, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Práce se zabývá návrhem, implementací, propojením, testováním a praktickým využitím mobilní aplikace pro iOS a podpůrné webové aplikace.

Analyzoval jsem, jaký účinek má hudba na sport a navrhl a implementoval vhodnou mobilní aplikaci umožnující uživatelům získávat data ze svých aktivit, hudebních přehrávačů a sledovat, jaký vliv na ně měla hudba. V návrhu i implementaci byl kladen důraz na použitelnost, rozšířitelnost a synchronizaci s cloudovým úložištěm. Proto byla vytvořena i webová aplikace, umožnující sledování výsledků naměřených na mobilu odkudkoliv.

Abstract

This work discusses design, implementation, connection, testing and practical use of mobile application for iOS and supporting web application.

I have analyzed the effects of music to sport and designed and implemented a fitting mobile application allowing users to collect data from their activities, music players and see the effects of music on them. During design and implementation, I have emphasised useability, expandability and synchronization with cloud storage, for this reason I have created a web application, aloowing users to monitor their results captured from mobile anywhere.

Klíčová slova

sport, hudba, mobilní aplikace, iOS, webová aplikace, Vue.js, SwiftUI, synchronizace, cloudové úložiště, CloudKit

Keywords

sport, music, mobile application, iOS, web application, Vue.js, SwiftUI, synchronization, cloud storage, CloudKit

Citace

HORT, Jaroslav. *iOS aplikace pro sledování sportovních výkonů.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Martin Hrubý, Ph.D.

iOS aplikace pro sledování sportovních výkonů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Martina Hrubého, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Jaroslav Hort
5. května 2021

Obsah

1 Úvod **3**

2 Vliv sportu a hudby **4**
2.1 Sport . . . 4
2.2 Hudba . . . 4

3 Vývojové a datové technologie mobilních a webových aplikací **5**
3.1 Mobil . . . 5
3.1.1 React Native . . . 5
3.1.2 Flutter . . . 6
3.1.3 SwiftUI . . . 6
3.2 Web . . . 6
3.2.1 HTML . . . 6
3.2.2 CSS . . . 6
3.2.3 JavaScript . . . 7
3.2.4 jQuery . . . 7
3.2.5 React . . . 7
3.3 Datové technologie . . . 7
3.4 SQL Databáze . . . 7
3.5 NoSQL Databáze . . . 8
3.6 CloudKit . . . 8

4 Průzkum současných aplikací **9**
4.1 Apple Health . . . 9
4.2 Strava . . . 10

5 Návrh **13**
5.1 Použité technologie . . . 13
5.2 Mobilní aplikace . . . 13
5.3 Webová aplikace . . . 15
5.4 Databáze . . . 15

6 Implementace **17**
6.1 Vývoj mobilní aplikace . . . 17
6.1.1 Podpora verzí iOS . . . 17
6.1.2 Realizace návrhu uživatelského rozhraní . . . 18
6.1.3 Životní cyklus iOS aplikací . . . 25
6.1.4 Speciální funkcionality aplikace . . . 26

6.1.5 Databáze a propojení s rozhraním 28
6.1.6 Požadavky na databázi 29
6.1.7 Propojení s úložištěm CloudKit 31
6.1.8 Sběr GPS dat 32
6.1.9 Data z API hudebního přehrávače 34
6.2 Webová aplikace 35
6.2.1 Vizuální styl stránky 38
6.2.2 Integrace knihovny Vue.js 39
6.2.3 Data z knihovny CloudKitJS 41
6.2.4 Nástroje pro zajištění kompatibility prohlížečů 43

7 Sport a hudba v praxi 45
7.1 Zisk informací 45
7.2 Integrace v uživatelském rozhraní 45
7.3 Experiment 46

8 Testování 47

9 Závěr 48

Literatura 49

Kapitola 1

Úvod

V dnešní době tvoří mobilní zařízení nepostradatelnou část lidského života, mnozí si ho bez svého zařízení již nedokážou představit. Krom pracovních využití také spousta uživatelů využívá svá mobilní zařízení pro volno časové aktivity, jako je poslech hudby, nebo sport. Tyto dvě aktivity mají pozitivní dopad na lidské tělo a uživatelé je často kombinují poslechem své oblíbené hudby při sportovních aktivitách, také často požadují své sportovní výkony a výsledky monitorovat, ať už z personálních důvodů, nebo pro porovnání s přáteli.

Cílem bakalářské práce je spojit tyto dvě využití a vytvořit mobilní aplikaci, která bude prezentovat uživatelovy sportovní aktivity a jaký vliv na ně má poslech hudby, kterou poslouchal při výkonu aktivity a podpůrnou webovou aplikaci pro prezentaci uživatelských dat. Předpokladem pro to, je vytvoření řešení, které dokáže fungovat nezávisle na jeho akcích a data budou kdykoliv prezentovatelná na více zařízeních.

V následující části se konkrétněji zaměřím vliv sportu a hudby na lidské tělo. Dále prozkoumám technologie umožňující vývoj mobilních a webových aplikaci. Ve třetí kapitole popíšu návrh svého řešení a v poslední řadě se budu zabývat jeho implementací ve vybraných technologiích.

Kapitola 2

Vliv sportu a hudby

V této kapitole stručně popisuji jaký vliv mají tyto aktivity na lidské tělo a jak je možné je prolínat. Čerpám nejen z odborných textů, ale také článků publikovaných na internetu.

2.1 Sport

Sport má nejen pozitivní účinek na fyzickou formu lidského těla, ale také na lidskou psychiku. Například běh pomáhá tělu snížit cholesterol a krevní tlak. Běh zvyšuje srdeční tep, což napomáhá proudění krve následným zvýšením energie v těle. Z psychické stránky je běh pozitivní pro zlepšení mysli a snížením risku nemoci a stresu [8].

Podobný efekt má například i jízda na kole, u které se ukázalo, že je to jedna z nejlepších metod pro snížení kalorií v těle.

2.2 Hudba

Spousta lidí pravděpodobně poslouchá hudbu pouze pro zábavu, určitě také mají svůj oblíbený žánr, který rádi poslouchají. Co ale možná neví je, že hudba má pozitivní účinek na jejich život a zdraví.

Konkrétních efektů je spousta, v první řadě je jako jeden ze zdokumentovaných efektů poslechu hudby uvolnění od stresu. Bylo zjištěno experimentálním výzkumem, ve kterém bylo několik subjektů vystaveno stresu a poté jim byla puštěna hudba, což vedlo ke snížení stresových faktorů v těle, například srdečního tepu a krevního tlaku [16]. Další experimenty dokazují, že různé typy hudby mají větší vliv na uvolnění od stresu, pomalejší hudba se ukázala jako pozitivnější na snížení srdečního tepu, než rychlejší hudba [19].

Dalším efektem je snížení bolesti. Studie na toto téma s pacienty trpícími chronickou bolestí, Fibromyalgií, provedený v několika týdnech, kdy byli vystaveni hudbě každý den dokazuje, že tito pacienti cítili menší bolest na konci experimentu než pacienti, kteří nebyli hudbě vystaveni. Stejných výsledků bylo dosaženo při studii pacientů po operací, kdy byli také vystaveni hudbě a ukázalo se, že se u nich snížila bolest a úzkost.

V neposlední řadě je velmi důležitým vlivem hudby zvýšení motivace posluchače. Podle provedených výzkumů na 12 cyklistech bylo odhaleno, že poslech hudby zvýšil jejich rychlost až o několik procent v závislosti na aktuálním tempu hudby [17].

Dá se tedy předpokládat, že kombinace sportu s hudbou má pozitivní efekt na lidské tělo.

Kapitola 3

Vývojové a datové technologie mobilních a webových aplikací

Tato kapitola pojednává o aktuálních trendech ve vývoji aplikací pro mobilní zařízení a web a způsobech, jakými se mohou uchovávat data dostupná na těchto zařízeních. Uvedená data se opírají o každoroční průzkum provedený společností StackOverflow[1], ve kterém se různých cílových skupin dotazovali na jejich preferované programovací jazyky a technologie. Dalším důležitým zdrojem dat jsou statistiky vyhledávaní v internetovém vyhledávači Google[2].

3.1 Mobil

V současné době se na trhu dominantně vyskytují dvě mobilní platformy – iOS od společnosti Apple a systém Android od společnosti Google. iOS je dostupný exkluzivně na zařízeních Apple, jako jsou mobilní telefony iPhone, kdežto Android je volně dostupný operační systém používaný různými tvůrci mobilů.

Vývojáři se v dnešní době snaží své aplikace implementovat pomoci multiplatformních technologií s podporou více systému. Také se dnes objevuje cesta propojení vývoje mobilní aplikace s webovou aplikací, což má své plusy, například urychlení implementace a údržby jedné aplikace na více platformách, ale také mínusy, jako je komplikovanější návrh samotné aplikace, aby byla jednoduše přenositelná. Můžeme ale nalézt i jiné směry vývoje, kupříkladu společnost Apple upřednostňuje cestu unifikovaného vývoje aplikací pro své mobilní a počítačové zařízení.

3.1.1 React Native

React Native je mobilní technologie vyvíjená společností Facebook od roku 2015 [13]. Cílem je vyvíjet aplikace dostupné na různých zařízeních, jako jsou mobilní zařízení, chytré televize, webové aplikace a desktopové systémy. Vývoj začal s webovou technologií React, vývojáři potřebovali způsob, jak své aplikace adaptovat na více zařízení a využívat nativní vlastnosti a vzhled cílových systému, což React, jako webová technologie, neumožňoval. Aplikace jsou programovány v jazyce JavaScript a je kladen důraz na asynchronní chování a reaktivnost rozhraní. React Native je v současné době nejpopulárnější mobilní technologii.

[1] Průzkum za rok 2020 dostupný na https://insights.stackoverflow.com/survey/2020

[2] Trendy ve vyhledávání dostupné na https://trends.google.com/trends/explore?cat=31&date=2018-12-01%202020-12-01&q=SwiftUI,%2Fm%2F0hx0p

3.1.2 Flutter

Poprvé představen v roce 2015 pod názvem Sky [7], Flutter je mobilní technologie společnosti Google založená na jazyce Dart s podporou jak zařízení iOS a Android, tak i podporou webu. Technologie je založena na principu Widgetů, což znamená, že uživatelské rozhraní je složeno s více různých prvků, takzvaných Widgetů, které dokážou na základě uživatelských vstupů mezi sebou propagovat informace. Narozdíl ale od technologie React Native, není kladen důraz na nativní prvky systému, ale integrován designový jazyk Material Design, od společnosti Google. Popularita technologie každým rokem roste, dle průzkumu provedeným StackOverflow jeho popularita mezi roky 2019 a 2020 vzrostla více než dvojnásobně a stala se z něj druhá nejpopulárnější mobilní technologie.

3.1.3 SwiftUI

Nový způsob vývoje aplikací SwiftUI od společnosti Apple byl poprvé vydán v roce 2019, představuje velkou změnu ve vývoji aplikací pro zařízení Apple [3]. Cílem je umožnit vyvíjet aplikace, které budou kompatibilní jak s mobilními platformami Apple, jako jsou operační systémy iOS, iPadOS nebo watchOs, tak systémem pro chytré televize tvOs a systémem macOs pro laptopy a desktopové počítače. Velkým rozdílem oproti předchozímu vývoji aplikaci je způsob programování uživatelských rozhraní, který se přesunul z tvorby v grafickém editoru, takzvaný Storyboard, na deklarováni komponent rozhraní ve zdrojovém kódu v jazyce Swift.

3.2 Web

Technologií usnadňující vývoj webových aplikací je dnes mnoho. Spousta vývojářů si nedokáže představit tvořit takovou aplikaci bez některé z technologii, které nejen usnadňují práci, ale zvyšují produktivitu a dovolují vytvářet interaktivnější weby, než kdy dříve. Všechny ale mají několik společných vlastností, jsou založeny na jazyce JavaScript, používaném pro skriptování webových stránek a usilují po kompatibilitu mezi všemi dostupnými webovými prohlížeči na různých platformách.

3.2.1 HTML

HTML je jedna z nejstarších a stále používaných technologií na webu. První verze se datuje až do roku 1993 a s různými aktualizacemi se používá do dnes. Jazyk funguje na systému tagů, které jsou interpretovány prohlížečem a následně zobrazeny uživateli v grafické podobě. Dnes forma psaní webů v HTML jako takovém upadá a je preferováno psaní jednotlivých prvků stránky, takzvaných komponent, ve webových technologiích, jako je například React, které následně za běhu generují HTML tagy pro prohlížeč.

3.2.2 CSS

Jako doplněk k jazyku HTML, CSS bylo vytvořeno pro možnost stylování HTML souborů. Jednotlivé styly mohou měnit všechny grafické aspekty stránky, pozicování, fonty textů, barvy a další. Styly jsou aplikovány buď na celý dokument, na jednotlivé HTML tagy, nebo na takzvané třídy, které lze přiřadit tagům. V dnešní době je možné pomocí CSS vytvářet animace nebo dynamicky měnit vzhled stránky v závislosti na velikosti displaye zařízení.

3.2.3 JavaScript

Pro realizaci dynamických akcí na webu slouží jazyk JavaScript, díky kterému lze vytvářet skripty na webových stránkách. Jazyk je interpretovaný prohlížečem a mezi jeho hlavní využití patří například zpracovaní uživatelských dat, odeslání požadavků na vzdálený server, nebo změna struktury HTML dokumentu, což je realizováno pomocí aplikačního rozhraní Document Object Model. Vlastnosti jazyka jsou závislé na prohlížeči, ty jsou často aktualizované, aby podporovaly nové standardy jazyka. Dnes obsahuje funkcionality, jako je asynchronní vykonávání operací, nebo je adaptován jako programovací jazyk webových serverů prostřednictvím technologie Node.js.

Moderní webové technologie, jako například React, staví na jazyce JavaScript a užití čistého JavaScriptu je dnes méně populární[3].

3.2.4 jQuery

Knihovna jQuery byla vytvořena jako rozšíření jazyka JavaScript. Jejím cílem je usnadnit práci s jazykem a zvýšit kompatibilitu mezi různými prohlížeči. Problémem JavaScriptu je, že vývojář musí myslet na podporu uživatelů, kteří nemají aktuální webový prohlížeč s aktuálními funkcemi jazyka. Kód psaný v jQuery je interpretován jako kód v JavaScriptu s dostatečnou podporou starších prohlížečů. Knihovna také obsahuje funkce, které v jazyce chybí, nebo ulehčuje použití existujících funkcionalit.

Ačkoliv se stále jedná o nejpopulárnější webovou technologii, v dnešní době poptávka po této knihovně upadá, s nástupem nových technologií používající vlastní syntax a rychlejším vývojem JavaScriptu jsou již mnohé dříve chybějící funkcionality obsaženy a jiné usnadněny.

3.2.5 React

Vyvinuto společností Facebook, React je nejpoužívanější technologie pro vývoj webových stránek na popředí, cílem Reactu je usnadnit vývoj a interakci uživatelských akcí a prezentaci dat. Technologie je založena na principu deklarace komponent stránky, které tvoří jeden výsledný dokument, dále na asynchronnosti operací a jak již napovídá název, reaktivnosti webové stránky, což znamená, že jednotlivé komponenty spolu komunikují a na základě uživatelských akci si mezi sebou předávat data a instantně změnit komponentu na stránce.

3.3 Datové technologie

Existuje spousta možností, jak uchovávat data dostupné na více platformách. Nejpoužívanější metodou je vytvoření serveru s databází, ke kterému jsou všechny aplikace připojené, to ale může být pro menší vývojáře problémem, proto dnes vznikají komerční služby, na kterých lze ukládat data.

3.4 SQL Databáze

SQL Databáze vyžadují vytvoření serveru, na kterém bude tato databáze spuštěna. Data jsou ukládány do relačních tabulek a indexovány podle primárních klíčů, mezi tabulkami je možné vytvářet relace a data tak spojovat. Data pak lze zobrazovat, nebo s nimi manipulovat, pomocí dotazovacího jazyka SQL. Jedná se o jednu z nejstarších technologií,

[3]TypeScript je dle průzkumu StackOverflow druhý nejpopulárnější, JavaScript až desátý.

která je používaná do dnes ve formě různých systémů vycházejícího z tohoto jazyka. Dnes nejpoužívanější je systém MySQL.

3.5 NoSQL Databáze

Skupina databází, které nejsou založeny na použití jazyka SQL. Stejně, jako SQL databáze je třeba pro ně mít server, na kterém se ukládají data. Na rozdíl od SQL databázi ale nejsou data uloženy do relačních tabulek, ale jako takzvané dokumenty. Ty jsou modelovány jako objekty obsahující atributy, pro přístup a správu dat se používá objektového přístupu. Staly se populárními právě díky své jednoduchosti a modernímu objektovému přístupu, který se používá ve většině aktuálních programovacích jazycích. Nejrozšířenější takovou databázi je pak MongoDB používané často pro webové servery a její mobilní alternativa RealmDB.

3.6 CloudKit

Cloudová služba od společnosti Apple nabízí možnost realizace backendu mobilních a webových aplikací. Služba poskytuje prostředky pro uchování jak uživatelských, tak aplikačních dat v takzvaných veřejných a privátních kontejnerech a také možnost autentizace a přihlášení do aplikace. Uživatelé, kteří se do aplikací využívající CloudKit přihlásí pod svým Apple účtem mají možnost uchovávat svá aplikační data v prostředí iCloud a jsou tedy dostupná na všech Apple zařízení. Vývojářům jsou poskytovaná analytická data o uživatelích aplikací, ale nemají přístup k privátním datům uživatelů.

Technologie byla poprvé představena na konferenci v roce 2014 a je velkou součástí snahy společnosti Apple sjednotit vývoj aplikací pro jejich zařízení [10].

Kapitola 4

Průzkum současných aplikací

V kapitole se podívám na současné aplikace pro sledování sportovních výkonů. Pokusím se najít jejich silné a slabé stránky a inspirovat se jejich řešením do své práce. Konkrétně se zaměřím na aplikace dostupné na platformě iOS, podívám se na ty nejpopulárnější řazené dle žebříčků obchodu AppStore.

4.1 Apple Health

Apple Health od společnosti Apple je předinstalovaná na všech mobilních zařízeních Apple s operačním systémem iOS 8 a vyšším [18].

Aplikace disponuje přehledným uživatelským rozhraním perfektně zapadajícím do vizuálního stylu operačního systému iOS. Po otevření je uživatel přivítán úvodní stránkou sumarizující jeho výsledky. První panel na této stránce obsahuje oblíbené aktivity a jejich poslední výsledky. Je možnost nastavit, která data se v tomto okně budou objevovat. Po kliknutí na tlačítko "edit", se otevře seznam aktivit, aplikace aktuálně monitoruje a má pro ně dostupná data, ale je možnost přidat si i aktivity, které nebyly dosud naměřeny. Pro každou aktivitu, jako je například chůze, jízda na kole, nebo spánek, je dostupné několik atributů. Pro chůzi pak například spálené kalorie, počet kroků, nebo vystoupaných schodů. Rozkliknutím této sekce se zobrazí list všech aktivit, které aplikace vede se statistikami za poslední den, 30 dní nebo starších.

Domovská stránka obsahuje také vybrané důležité informace o statistikách za posledních několik dní v porovnání se staršími daty. Tato sekce pak dává uživateli rychlý přehled o jeho pravidelných výkonech a zlepšení, případně zhoršení výkonu. Dále zde lze nalézt fitness zprávy s tipy, jak se o své zdraví nejlépe starat a jak s tím může aplikace pomoci.

Obrázek 4.1: Rozhraní aplikace Apple Health

Po přepnutí do sekce Browse ve spodní liště aplikace se zobrazí seznam kategorii, do kterých jsou zařazeny jednotlivé aktivity a možnost textového vyhledávání dat a aktivit. Samotná stránka s aktivitou obsahuje textový popis aktivity a detailní informace ve formě grafu a možnosti zobrazení dat v intervalu jednoho dne, týdne, měsíce a roku.

Aplikace jako taková neměří aktivity, ale slouží jako centrální bod pro data z různých zdrojů. Je možnost připojit aplikace od jiných vývojářů, které mohou sdílet data s Apple Health, která je v přehledné formě podá uživateli. Samotné podání dat je ale velmi jednoduché, pro většinu aktivity je zobrazen pouze graf. V tomto ohledu vynikají jiná řešení, které dokážou prezentovat i detailní statistiky průběhu aktivit, jako například trasu běhu, nebo bicyklu.

4.2 Strava

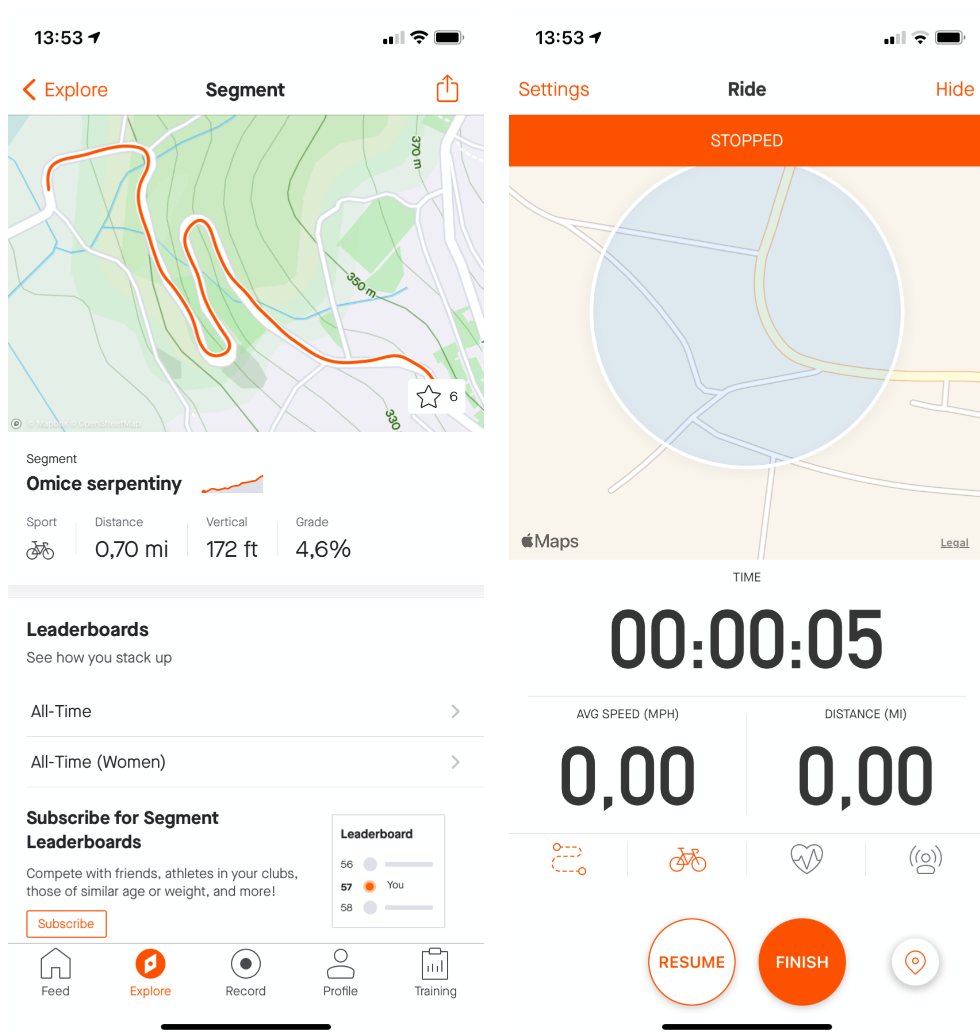
Strava je aplikace pro měření sportovních výkonů se zaměřením na pohybové aktivity, jako je chůze, běh a cyklistika. Vydána byla roce 2009 a dostupná je na mobilních zařízeních

Apple i Android, aplikace klade velký důraz na sociální stránku sportu a proto obsahuje spoustu vlastností umožňující sdílet a porovnávat výsledky s ostatními sportovci.

Po spuštění je v první řadě je po uživateli vyžadováno přihlášení, které lze provést pomocí existujících účtu, například Google, nebo Apple účtu. Díky přihlášení má aplikace možnost ukládat uživatelská data, a uživatelé mají možnost data publikovat na svých profilech, sledovat jiné uživatele, vytvářet a připojovat se do klubů a pořádat aktivity.

Na prvním okně aplikace zvaném Feed lze nalézt informace o aktivitách sledovaných uživatelů, své poslední aktivity, nebo aktivity klubů, do kterých uživatel patří. Tyto informace jsou obnovovány periodicky a řazeny od nejnovějších, je tedy možné jednoduše sledovat výkony ostatních. Aktivity lze otevřít a dostat tak detailnější informace, jako například mapu trasy, její délku a rychlost a čas uživatelova výkonu. Je možné také k aktivitám přidat komentář.

Další okno, Explore, obsahuje v horní liště čtyři možnosti zobrazení, v prvním z nich se nachází osobní doporučení na nové trasy, nabízené dle aktuální polohy a doposud dosažených výkonů, nebo kluby v blízké vzdálenosti, do kterých se lze připojit. Následující zobrazení Challenges nabízí pravidelné výzvy, které může uživatel aktivovat a pokusit se splnit, jako například výzva ujít několik kilometrů za měsíc, nebo splnit určitý počet aktivit měsíčně. V předposledním zobrazení určeném klubům je lze vyhledávat a filtrovat dle aktivity a následně zažádat o členství. Na posledním zobrazení je mapa zobrazující trasy poblíž pro inspiraci uživatele.



Obrázek 4.2: Rozhraní aplikace Strava

Strava neumí automaticky zaznamenávat pohyb, tudíž obsahuje okno zvané Record, ze kterého lze zahájit měření aktuální aktivity. Je na výběr z několika sportů a uživatel má možnost připojit ke svému mobilnímu zařízení doplňky, jako například chytré hodinky, nebo měřic srdečního tepu komunikující před protokol Bluetooth a aplikace data z těchto zařízení zahrne do měření aktivity. Po zahájení měření se na displayi objeví stopky a informace o dosažené vzdálenosti a aktuální rychlosti. Pokud se uživatel rozhodne skončit, signalizuje to aplikaci stiskem tlačítka a ta mu zobrazí shrnutí jeho výkonu a poskytne možnost přidat popis výkonu a zda chce tento výkon publikovat na svůj profil, uložit si ho soukromě, nebo smazat úplné.

Aplikace je z velké části zdarma, je však ale možné pořídit si předplatné, podpořit tak vývojáře a dostat přistup k detailnějšímu rozboru svých výkonů.

Kapitola 5

Návrh

V této kapitole představím svůj návrh řešení aplikace, zaměřím se na použité nástroje a programovací jazyky, návrh databáze a vhodného propojení a vzhled uživatelského rozhraní.

5.1 Použité technologie

Aplikace je primárně vyvíjena na zařízení iPhone s operačním systémem iOS. Na základě statistik[1] běžné publikovaných společností Apple k datu 15. prosince 2020 používá 81% uživatelů iPhone operační systém verze 14. Dalších 17% uživatelů používá verzi iOS 13 a zbylé 2% ostatní, starší verze. Jako minimální podporovanou verzi aplikace jsem tudíž zvolil verzi iOS 13, díky tomu bude aplikace dostupná až na 98% mobilních zařízení iPhone.

Jako vývojové prostředí bude sloužit program XCode, vyvinut Applem přímo pro vývoj mobilních aplikací na iPhone a iPad a dostupný na systému macOs. Pro vývoj rozhraní aplikace je zvolena technologie nová SwiftUI, umožňující lehký převod aplikace z mobilních zařízení na desktopová. Uložení dat bude provedeno pomocí technologií CoreData, starající se o data lokálně v uživatelově zařízení a CloudKit, pomocí kterého lze lokální data ukládat na Cloudové uložiště a mít je tak zálohována a následné dostupná na více zařízeních.

Webová aplikace bude převážně v jazyce JavaScript s použitím Apple knihoven a možností načtení dat z Cloudové uložiště pomoci CloudKitJS. Uživatelské rozhraní bude implementováno v HTML a CSS. Není zde potřeba mít vlastní server, který bz se staral o data, vše bude zprostředkováno pomocí CloudKit, který poskytuje i bezpečnou autentizaci uživatele.

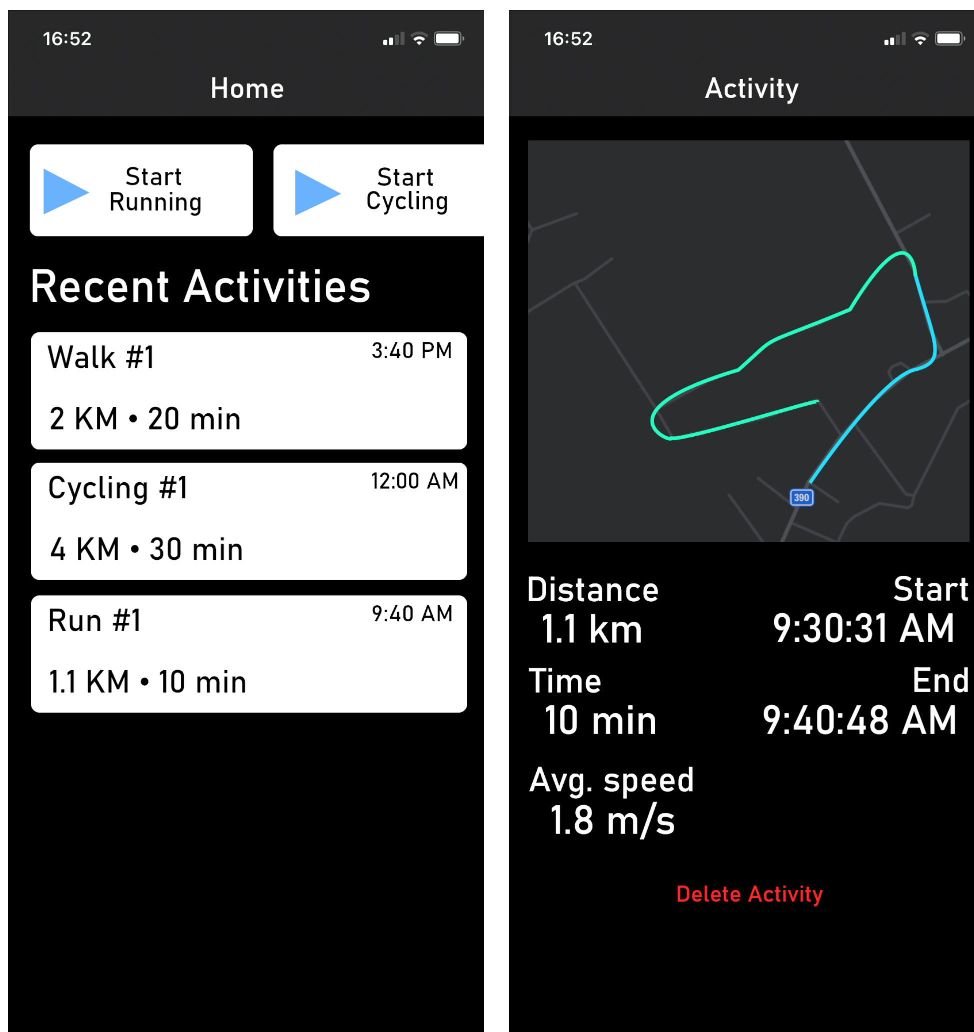
Pro realizaci propojení s hudbou bude využito programové rozhraní hudební aplikace Spotify, umožňující přistup k uživatelově hudbě z prostředí aplikace.

5.2 Mobilní aplikace

Mobilní aplikace se zaměřuje na přehlednost a jednoduchost použití na co největším počtu zařízení s různou velikostí obrazovky. Základem aplikace budou tři okna. První, výchozí okno, zobrazené v případě, že uživatel aktuálně nevykonává aktivitu, bude zobrazovat seznam posledních aktivit, které jsou byly uloženy. Na tomto okně bude pouze rychlý náhled obsahující pouze nejdůležitější informace, konkrétně typ aktivity, datum, kdy byla vykonána, celkovou dobu a vzdálenost, kterou uživatel absolvoval. V horní části okna budou také rychlé zkratky pro zahájení aktivit.

[1] https://developer.apple.com/support/app-store/

Po otevření některé z předchozích aktivit bude uživatel přesměrován na stránku s detailním zobrazením. Zde bude v první řadě vidět mapa aktivity, na níž bude barevně vyznačena trasa. Pokud bude mít uživatel aplikaci propojenou s hudebním přehrávačem Spotify, bude trasa vybarvena různě a to dle hudby, která hrála v určitém úseku. Bude možnost na jednotlivé úseky kliknout a tím zobrazit data o hudbě, která v ten moment hrála a informaci o uživatelově průměrné rychlosti v tomto úseku. Pod mapou lze nalézt další informace o aktivitě, krom dat, které byly zobrazeny v rychlém náhledu, zde budou i informace o průměrné rychlosti ve všech úsecích, hudba, při které byl uživatel nejrychlejší a čas zahájeni a konce aktivity.



Obrázek 5.1: Návrh rozhraní mobilní aplikace

Druhé okno bude sloužit pro monitorování aktuální aktivity. Dominantním prvkem zde bude mapa zobrazující aktuální polohu a při aktivitě aktuální trasu od bodu zahájení. V případě, že dosud nebyla zahájena aktivita, zde bude tlačítko, pomocí kterého ji lze zahájit. Pokud už aktivita běží, bude toto zobrazení výchozím zobrazením po otevření aplikace a tlačítko pro zahájení se změní na ukončení aktivity.

Samotné monitorování pohybu se bude dít na pozadí. V uživatelem nastavitelném intervalu od několika sekund až do minuty, si aplikace vezme současnou polohu zařízení a

aplikačního rozhraní Spotify se dotáže na aktuálně hrající hudbu. Aby se zabránilo příliš častému dotazování na Spotify, bude dotazování na službu provedeno pouze v každém druhém intervalu zisku lokace, ačkoliv se sníží přesnost, je nepravděpodobné, že by uživatel změnil hudbu vícekrát za minutu a tyto data by nebyly relevantní pro výsledky aktivity.

V posledním okně bude nastavení aplikace. Zde bude možné se přihlásit ke Spotify účtu a propojit ho tak s aplikací, nastavit intervaly měření, poskytnout aplikaci oprávnění ke sběru dat, nebo hromadně smazat všechna naměřená data.

5.3 Webová aplikace

Webová část aplikace bude sloužit pouze pro zobrazení dat vytvořených mobilní aplikací. Aplikace nebude přístupná bez přihlášení k Apple účtu a její vizuální styl bude z velké častí stejný, jako mobilní aplikace, pouze přizpůsobený pro web.

Po autentizaci bude uživatel přesměrován na výchozí stránku aplikace, v horní listě bude možnost vyhledat aktivitu dle jména a tlačítko se jménem uživatele, kde se bude moci z aplikace odhlásit.

V hlavní částí okna bude zobrazen seznam jeho aktivit, seřazen dle data od nejnovějších. Oproti mobilní verzi zde ale budou zobrazeny všechny datové informace, které jsou na mobilu dostupné pouze v detailním zobrazení. Pokud si uživatel vyhledá aktivity, budou zde místo toho zobrazeny výsledky hledání, společně s možnostmi pro řazení výsledku a to dle data, vzdálenosti, času, nebo rychlosti.

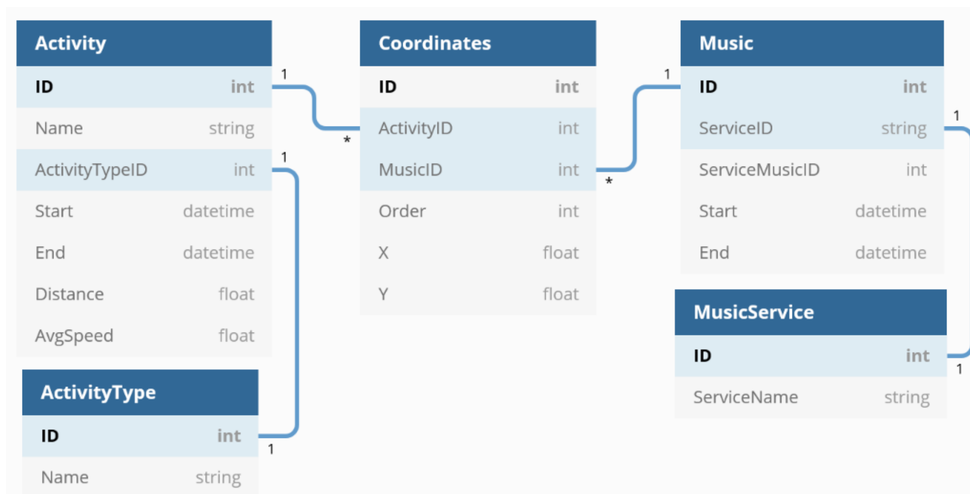
V detailním zobrazení aktivity, které se otevře jako modální okno, bude mapa s trasou aktivity, zobrazena stejně, jako na mobilním zařízení, obsahující barevně rozlišená data o hudbě.

5.4 Databáze

Databáze bude implementována pomoci CoreData. Data jsou ukládána do takzvaných entit, připomínající tabulky z klasických SQL databází, avšak na rozdíl od nich dokážou entity nést více informací a více datových typů. Nové záznamy budou nejprve ukládány lokálně v uživatelově zařízení a po dokončení aktivit budou nahrány do jeho cloudového úložiště a spravovány pomocí technologie CloudKit.

Pro každou novou aktivitu bude vytvořen jeden záznam entity Activity, po vytvoření obsahující informace o jméně aktivity, jejím typu, což bude buď běh, chůze, nebo cyklo, a datum a čas počátku aktivity. Další atributy budou doplňovány později. V průběhu aktivity, při periodickém sběru nových dat, jsou do tohoto záznamu ukládaný informace o poloze, zde modelováno jako pole entity Coordinates, která obsahuje GPS souřadnice a informaci o aktuálně hrající hudbě, které jsou uloženy v další entitě Music. Pro ušetření dat a eliminaci redundance jsou v této entitě pouze atributy o typu služby, ze které data pochází a ID hudby, pro zpětné nalezení ve službě.

Na konci aktivity je do tohoto záznamu vepsán datum a čas konce a vykonaná vzdálenost. Ostatní zobrazovaná data, jako je čas aktivity a průměrná rychlost, jsou dopočítána pouze při zobrazení a není třeba je uchovávat v entitě.



Obrázek 5.2: Digram entit

Mobilní a webová aplikace mají přístup ke stejným datům, ty jsou uloženy v uživatelově soukromém cloudovém úložišti, má k nim přístup pouze on a jsou přístupná pouze po autentizaci pomocí Apple účtu. Není tedy nutné entitám dávat atributy určující uživatele.

Kapitola 6

Implementace

V této kapitole se zabývám popisem vývoje jednotlivých částí projektu, jak mobilní, tak i webové aplikace, implementací návrhu uživatelského rozhraní, backendu a vhodného propojení mezi webovou částí, která je převážně implementována ve frameworku Vue.js, za použití podpůrných knihoven, a mobilní částí v jazyce Swift a uživatelským rozhraním ve SwiftUI. Dále zde proberu průběh testování a v poslední řadě možnosti rozšíření a publikaci projektu na platformu AppStore.

6.1 Vývoj mobilní aplikace

Jak již bylo zmíněno výše, mobilní aplikace je primární částí projektu, bez které je webová část zbytečná, bylo ji tedy věnováno více pozornosti. Po vytvoření návrhu bylo pro jeho realizaci důležité zvolit vhodné vývojové prostředí – Xcode od Apple, ve kterém byl vytvořen nový projekt s požadovanými vlastnostmi pro mobilní aplikaci s připojení na sdílené úložiště. Vývojové prostředí dokáže vygenerovat základní kód, pro uživatelské rozhraní a práci s daty, který lze spustit a stavět na něm dál vlastní aplikaci.

6.1.1 Podpora verzí iOS

Nastínil jsem, že plánem je podporovat zařízení se systémem iOS od verze iOS 13. Tento plán bohužel nevyšel. Snažil jsem se plně využít všech možností, které nabízí systém SwiftUI, jelikož je to ale relativně mladý framework, Apple ho často aktualizuje a přidává do něj novou funkcionalitu, nebo převádí vlastností starého frameworku UIKit do nového. Tyto aktualizace se dějí jak s příchody nových verzí jazyka Swift, tak celkových aktualizací vývojových prostředí a operačních systémů. Hlavní verze iOS často přináší největší změny, s příchodem iOS 14 přibyly do SwiftUI komponenty jako `LazyVStack`, nebo velmi důležitá komponenta aplikace, `MapView`. Doposud bylo třeba použít knihovny třetích stran, nebo si ho sám implementovat. Inkrementální aktualizace pak přináší menší změny, ale stále užitečné a využité v aplikaci. Z těchto důvodu bylo třeba povýšit minimální podporovanou verzi iOS na 14, což nebude představovat velký problém. Průměrná doba podpory nových verzí na zařízení ze strany Apple jsou 4 roky a počet zařízení běžících na této verzi se neustále zvyšuje.

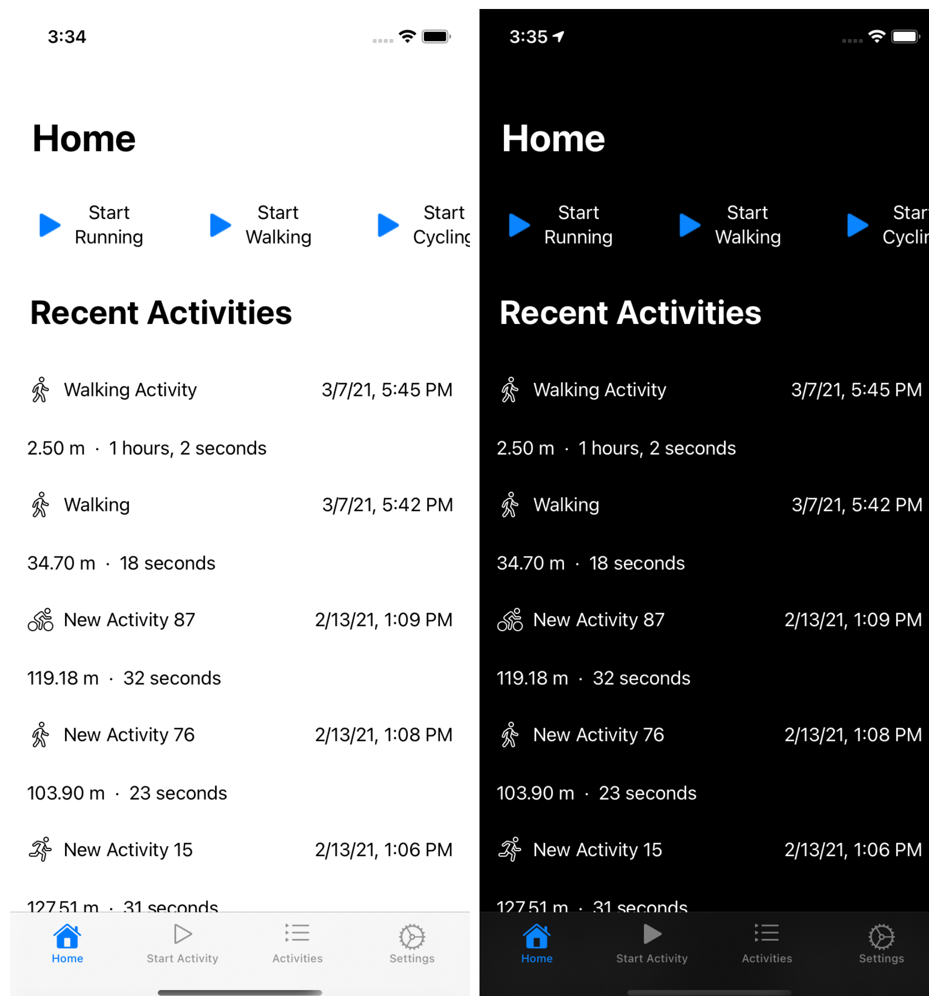
6.1.2 Realizace návrhu uživatelského rozhraní

V první řadě jsem tedy začal přetvářet návrh rozhraní do funkční podoby ve SwiftUI. Jednotlivé obrazovky jsou strukturovány do vlastních View souborů, které jsou dále rozloženy na další menší View pro zpřehlednění kódu a zjednodušení znuvupoužitelnosti. Na počátku je hlavní knihovní View `TabView`, který aplikaci rozčlení do takzvaných tabů, mezi kterými se lze přepínat pomoci navigační lišty, umístěné na spodku obrazovky. Každý tab má svoji ikonku a název pro reprezentaci obsahu. Byly použity převážně systémové ikonky od Apple, ale protože v systému nejsou vhodné ikonky pro všechny stavy, použil jsem nadále volně dostupné ikonky od společnosti `Icons8`[1] ve vhodném formátu, aby zapadly do celkového vzhledu aplikace.

TabView v sobě obsahuje vlastní views, mezi kterými se uživatel přepíná a všechny tyto Views jsou nové struktury v dalších souborech. Po startu aplikace jsou vytvořeny všechny instance všech views v TabView a inicializovány počáteční hodnoty.

Úvodní view, `HomeView`, obsahuje seznam posledních aktivit, který je načten z uživatelova úložiště a dále tlačítka pro rychlý start nové aktivity. Oba prvky jsou vytvořeny jako další view. Tlačítka pro start potřebují pro zobrazení název a jsou to prvky komponenty `ListView`, která je dokáže zobrazit jako jeden seznam v pořadí, v tomto případě horizontálně. Uživatel se mezi nimi může posouvat, v případě, že se mu na obrazovku nevejdou všechny naráz. Názvy aktivit jsou ve vlastním modelu a uloženy jako výčtový typ, tudíž pro zobrazení je nad tímto typem iterováno a v Listu jsou obsaženy všechny hodnoty, které tento typ obsahuje. Druhý prvek je také součástí `ListView` a obsahuje informace o vykonané aktivitě. O akce, které se vykonají pro ťuknutí na tyto komponenty, se stará událost `onTapGesture`. Lze ji navázat na většinu View struktur a tomto ohledu se komponenty chovají jako tlačítka. Aby uživatelovi bylo jasné, že dotykem na ně vyvolá operaci, mají obě grafické znázornění v podobě šipek, jak je tomu zvykem v běžných aplikacích na iOS. Vyvolání gesta na komponentu s uloženou aktivitou způsobí vytvoření nové instance detail okna s podrobnostmi aktivity a následnou navigaci na toto nové view.

[1] https://icons8.com/

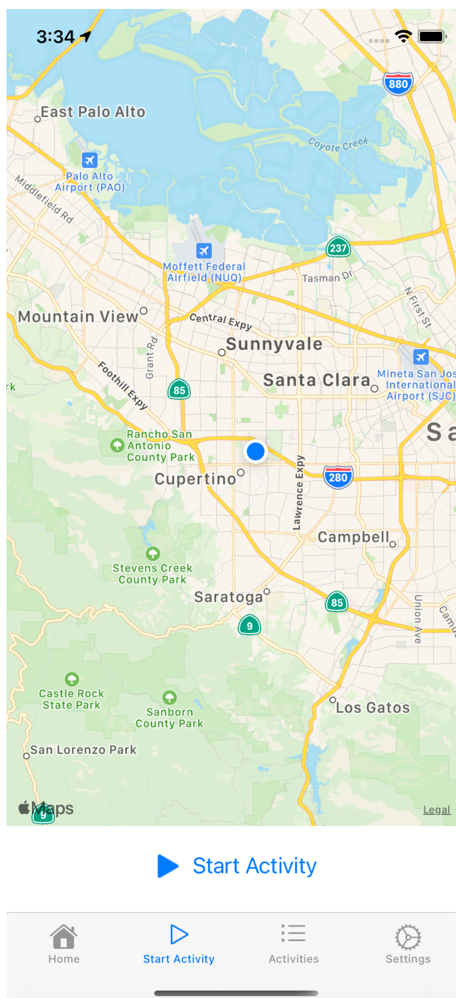


Obrázek 6.1: Domovská stránka ve světlé a tmavé variantě

Po stisku komponenty pro rychlé zahájení se nastaví patřičné stavové proměnné, aplikace se přepne na druhý view a zahájí sledování. Oba view sdílí jeden stavový objekt s logickou proměnnou pro inicializaci a proměnnou s výčtovou hodnotou typu aktivity, která se má spustit. Přepnutí je realizováno pomocí stavové proměnné `selectedIndex` struktury `TabView`. Uživatelské rozhraní SwiftUI reaguje na změny těchto proměnných hned a není třeba vyvolávat události pro změny v programu pro změnu TabView a tyto funkcionality SwiftUI ani nenabízí. Změny stavů způsobí nové renderování rozhraní a s tím se vykonají i události, jako je změna aktivního view.

Druhé view, nazvané `CurrentActivityView`, na obrázku 6.2, má na starosti sledování uživatelovy aktivity. Zde je potřeba, aby uživatel povolil aplikaci přístup k jeho aktuální lokaci. iOS nabízí několik způsobu pro sdíleni lokace. Mód Once dá aplikaci možnost dotázat se na lokaci pouze jednou, další požadavky budou zamítnuty. While In Use povolí všechny požadavky, pokud aplikace běží a je na popředí. Always povolí přistup vždy, i pokud je aplikace na pozadí v neaktivním stavu. Od verze iOS 14 Apple z důvodu větší bezpečnosti ztížil uživatelům možnost udělit povolení Always. Nově tato možnost není ve vyskakovacím okénku, na rozdíl od prvních dvou, a uživatel musí jít do systémového nastavení a udělit povolení ručně. Pro správnou funkčnost je třeba povolit ten nejvyšší Always. Uživatel je

dotázán na povolení sdílení, pokud tak neučiní, nebude mít možnost začínat nové aktivity a sbírat data.



Obrázek 6.2: Zobrazení stránky současné aktivity

Pro to, aby aplikace vůbec mohla požádat o povolení k lokaci, je nejprve třeba nastavit klíče s popisem důvodu v souboru `Info.plist`. Popisek je poté zobrazen v systémovém okénku vyžadující povolení. Existují čtyři možné klíče pro vyplnění a pokud aplikace žádá o povolení vyžadující některý z klíčů, jehož hodnota nebyla vyplněna, požadavek okamžitě selže, bez otevření okna. Přehled klíčů je zobrazen v tabulce 6.1 a byl převzat z dokumentace Apple.

Je zde vidět i mapa s aktuální polohou. Mapa je poskytnuta ze systémové knihovny `MapKit` od Apple a polohu si aktualizuje sama, je však také závislá na povolení práv aplikaci. Na spodní straně je umístěno tlačítko pro zahájeni, které otevře kontextovou nabídku s typem aktivity. Pokud aktivita již běží, je tlačítko nahrazeno možností zastavit sběr dat pro aktuální aktivitu.

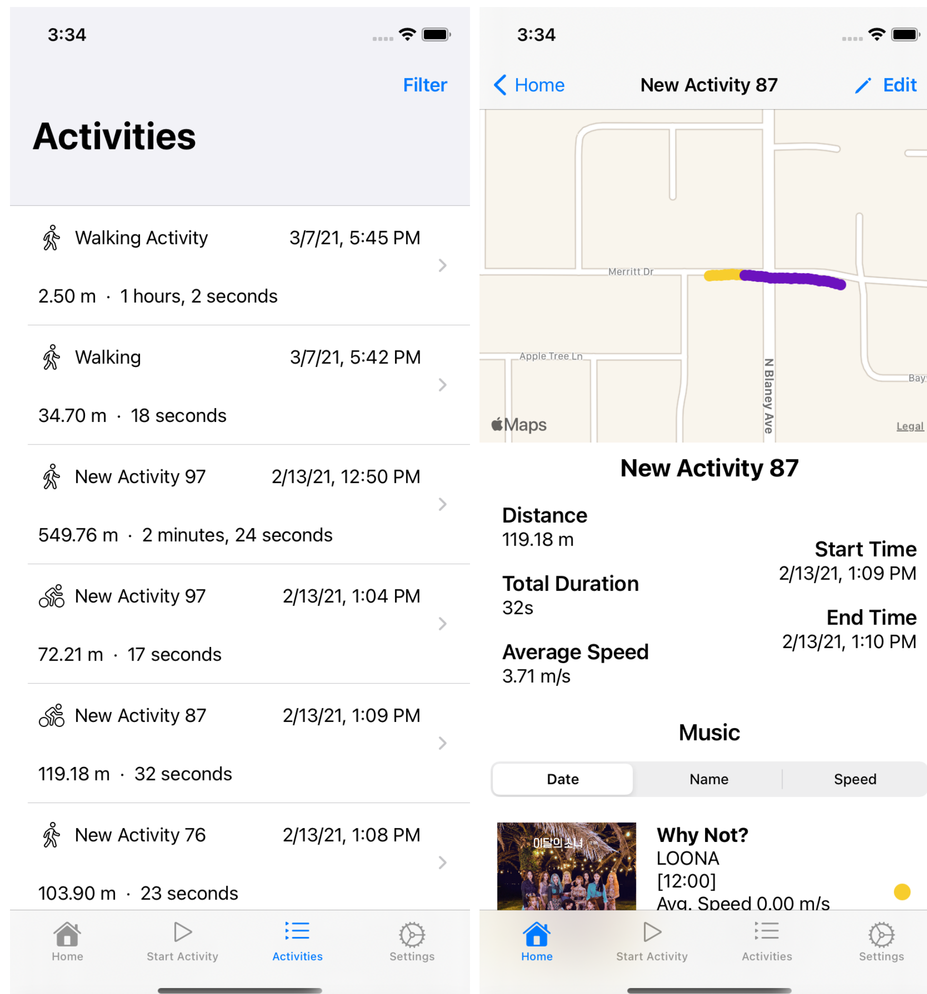
V neposlední řadě je `ActivityListView`, zobrazeno na obrázku 6.3, zobrazující větší seznam posledních aktivit. Pro jednotlivé zobrazení byla znovu použita komponenta z prvního view, aby byla zachována konzistence. Je možné aktivity filtrovat na základě dvou atributů – názvu a data začatku, a seřazovat dle návzu, data a rychlosti. Operace jsou realizovány

Klíč	Použití
NSLocationWhenInUseUsage	Aplikace vyžaduje přistup při používání, nebo kdykoliv
NSLocationAlwaysAndWhenInUseUsage	Aplikace vyžaduje přistup kdykoliv
NSLocationUsage	Aplikace běží na macOs a vyžaduje lokaci
NSLocationAlwaysUsage	Aplikace podporuje iOS 10 a nižší a vyžaduje lokaci kdykoliv

Tabulka 6.1: Přehled klíčů pro důvody použití lokační služby

pomoci predikátu a sort descriptorů FetchRequestu, podrobněji o implementaci v kapitole 6.1.6. Nad daty na pozadí, která jsou průběžně obnovována, pokud dojde k přidání nových, nebo ke změně, je vykonán dotaz, díky kterému lze vyfiltrovat pouze ty aktivity, které uživatel hledá. Samotný vyhledávací box není knihovní komponentou, jak jsem předpokládal, ale byl implementován od počátku. Ačkoliv je vyhledávání běžné a spousta systémových iOS aplikací ho nabízí jako součást listových zobrazení, tato komponenta v aktuálním stavu SwiftUI chybí.

Stejně jako na domovské stránce odkazují komponenty s aktivitou na detailní view, jež obsahuje podrobné informace o vykonané aktivitě. Souřadnice, které aplikace nasbírala, jsou umístěny na mapu v podobě bodů. Zde se vyskytuje další limitace knihovny MapKit ve SwiftUI a to ta, že není možné vykreslit body jako přímky. V předchozích způsobech vývoje iOS aplikací, UIKit, toto bylo možné. Pokusil jsem se obalit UIKit view novým SwiftUI, což mně umožnilo umístit na mapu body podobným způsobem, jako to bylo možné dříve, ale tento přistup vedl k dalšímu problému – zachytávání a reakce na události vyvolané uživatelem. Cílem bylo umožnit dotknout se jednotlivých bodů, nebo přímek, a zobrazit si další informace, což se nepodařilo. Vrátil jsem se proto ke knihovnímu MapKit view. Body se zobrazují dynamicky z pole souřadnic uložených společně s aktivitou, tudíž pokud se některý odebere, nebo přidá nový, je view překreslen.



Obrázek 6.3: Seznam aktivit a detailní zobrazení

Jednotlivé body na mapě jsou zbarveny podle aktuální hudby, kterou uživatel v té době poslouchal. Pokud je hudební integrace vypnuta, nebo uživatel nic neposlouchal, je bod vybarven výchozí barvou. Pro zobrazení mapy je zapotřebí jí předat oblast, kterou má zobrazovat, v parametru `coordinateRegion`. V aplikaci to pak je region dostatečně velký, aby obsáhl všechny souřadnice. Vykreslení probíhá předáním hodnot parametru `annotationItems`. Obsahem toho je pak předem připravená kolekce struktur obsahující souřadnice v typu `CLLocationCoordinate2D`, barvu `Color` s výchozí hodnotou reprezentující šedou a objekt s informacemi o hudbě. Ten může být null v případě, že taková data nejsou, pak je třeba kontrolovat na hodnotu před reakcí na akci ťuknutí.

Funkcí foreach je pak každá hodnota kolekce zobrazena pomocí MapAnnotation view, očekávající souřadnicí, na které se vykreslí, a vzhled, pro ten lze použít komponenty `MapMarker`, `MapPin`, nebo generické view, jako je `Circle`. Pokud má byt zobrazeno více souřadnic na malém prostoru přehledně a s různou barvou, první dva zde nejsou vhodné, proto byl použit `Circle`.

```
Map(coordinateRegion: $coordinateRegion,
    annotationItems: coordsToMark)
    { place in
```

```
        MapAnnotation(coordinate: place.coordinate, content: {
            Circle()
                .strokeBorder(place.color, lineWidth: 7)
                .onTapGesture {
                    if let _music = place.musicItem {
                        selectedMusic = _music
                        partialSheetShown = true
                    }
                }
        })
    }
```

Výpis 6.1: Zobrazení souřadnic na mapě

Po ťuknutí na kterýkoliv bod vysunut nový `PartialSheet` view. Zabírá pouze spodní polovinu obrazovky, uživatel tedy dále vidí na pozadí mapu, a na popředí jsou zobrazena data o hudbě, stažená ze zdroje, ze kterého hudbu poslouchal, obal alba, tvůrci, název, doba od kdy do kdy hudba hrála, a také informace o výkonu v tomto momentě. `PartialSheet` view lze zavřít stažením dolů, nebo ťuknutím mimo, například na pozadí.

Pod mapou je více informací o aktivitě. Její celkový začátek a konec, délka, doba a rychlost. Jako poslední komponenta je seznam hudby, seřazen od první puštěné do poslední. Každá položka je označena stejnou barvou, jako na mapě. V tomto seznamu je pouze jméno a čas kdy hrála, po ťuknutí se opět vysune nové view s detaily.

Aktivitu je možné editovat, konkrétně její jméno, nebo ji celou smazat. Pro editaci je třeba změnit mód aktivací editačního tlačítka v pravém horním rohu navigace. Komponenta, která zobrazuje název aktivity, je změněna z klasického textového Labelu na textové pole, do kterého lze zapisovat. Uložené změny jsou přepsány dynamicky ve všech částech aplikace a uloženy na úložiště.

Na posledním tabu je `SettingsView` pro uživatelská nastavení. Vzhledově jsem se snažil napodobit systémové nastavení v iOS. Na okna pro podrobné nastavení je odkazováno z `ListView`, které lze rozdělit na sekce a každé sekci dát záhlaví a zápatí, buď pouze jako text, nebo vlastní view. Řádky jsou udělány jako malé víew, obsahující ikonku, název a odkaz na view, které se otevře po kliknutí. Protože možností nastavení není mnoho, jsou pouze dvě sekce, jedna pro celkové možnosti a druhá pro zobrazení informací o aplikaci, jako je například verze, jméno autora a licence knihoven třetích stran.

Jako první možnost pro nastavení je interval sběru dat při aktivitě. Na výběr je několik možností v jednotkách stovek milisekund. Po provedení výběru se možnost ihned uloží a aktualizuje view, aby reflektoval novou možnost. Původně bylo cílem udělat tento výběr pomocí komponenty `Picker`, ale protože bylo příliš komplikované ji na data napojit tak, aby se při výběru ihned uložila a view aktualizovalo, vytvořil jsem tyto možnosti sám, jako pár tlačítek reagujících na událost.

Druhá možnost ovládá integraci hudebního přehrávače. V základu je možné integraci zapnout, nebo vypnout, po zapnutí se zobrazí další možnosti a indikátor stavu propojení. Samotné propojení probíhá pomoci autentizace s cílovým přehrávačem, v této době pouze Spotify. Uživatel inicializuje autentizaci přihlášením ve webovém prohlížecí v aplikaci, Spotify API vrátí token, který je uložen a dále používán při požadavcích na API.

Integrovaný webový prohlížeč v aplikaci je další věc, která v aktuálním stavu SwiftUI chybí. V UIKitu byl funkční prohlížeč `WebView`, který ale dosud nebyl zcela předělán do nového. Je možnost použít knihovnu `WKWebView`, díky níž lze zobrazit okno s vybranou

stránkou, chybí ale však ovládací prvky a jednoduchý přístup k událostem vyvolaným při navigaci v prohlížeči. Zvolil jsem proto volně dostupnou knihovnu, ve které autor implementoval potřebné navigační prvky a události nad systémovou knihovnou `WKWebView` aby fungovala podobně jako starší `UIKit` knihovna.

Ze stránky k ovládaní autentizace v nastavení aplikace je uživatel odkázán na nové view s prohlížečem, ten se otevře na stránce s přihlášením do Spotify a všechny práva, která jsou od Spotify požadována, jsou zapsána už v URL. Prohlížeč funguje tak, že při každé navigaci vyvolá událost `onNavigationAction`, ta může být několika různých typů. Mě zajímá pouze událost `didReceiveServerRedirectForProvisionalNavigation`, která je vyvolána, když byl uživatel přesměrován na jinou stránku vzdáleným serverem. Spotify vrací data o přihlášení v parametrech URL zadané při nastavení aplikace v jejich API, není ale možné událost vyvolat pouze na požadovaných URL, tudíž pokaždé, když proběhne přesměrování, je URL parsována a zkontrolováno, zda už je aplikace na správné URL. Pokud ano, jsou z ní vytažena data, ty uložena do databáze a prohlížeč je programově zavřen. Stav rozhraní se aktualizuje v reakci na úspěch, nebo neúspěch autentizace. Pokud je uživatel přihlášen, je na stránce v nastavení tato skutečnost signalizována.

Implementována byla i podpora světlého a tmavého režimu, varianta v obou režimech je na obrázku 6.1. Od iOS 12 lze přepnout vzhled celého systému do tmavého režimu, a pro to byla do SwiftUI přidána systémová proměnná výčtového typu ColorScheme, obsahující hodnoty `.light` nebo `.dark`. Pro to, aby aplikace reagovala na tyto změny v systému lze z proměnné číst a na každém view existuje stejnojmenná funkce, starající se o přepnutí barev. Není třeba ji volat na každém jednotlivém view, stačí pouze na tom hlavním zobrazujícím se po načtení aplikace, v mém případe `TabView`. Ukázka 6.2 obsahuje úryvek z kódu hlavního View a nastavení schématu pro podřazená view. Aplikaci lze nastavit různé barvy pro tmavý a světlý režim, ve výchozím stavu je to bíla a tmavě šedá, s třetí barvou pro zvýraznění textu. Vlastním ikonkám obsaženým ze souboru lze také nastavit barevná schémata, buď je třeba měnit ikonku manuálně, nebo jí nastavit systémovou kontrolu.

```
struct MainView: View {
    @Environment(\.colorScheme) var colorScheme

    var body: some View {
        TabView {
            ...
        }
        .colorScheme(colorScheme)
    }
}
```

Výpis 6.2: `ColorScheme` proměnná ve SwiftUI

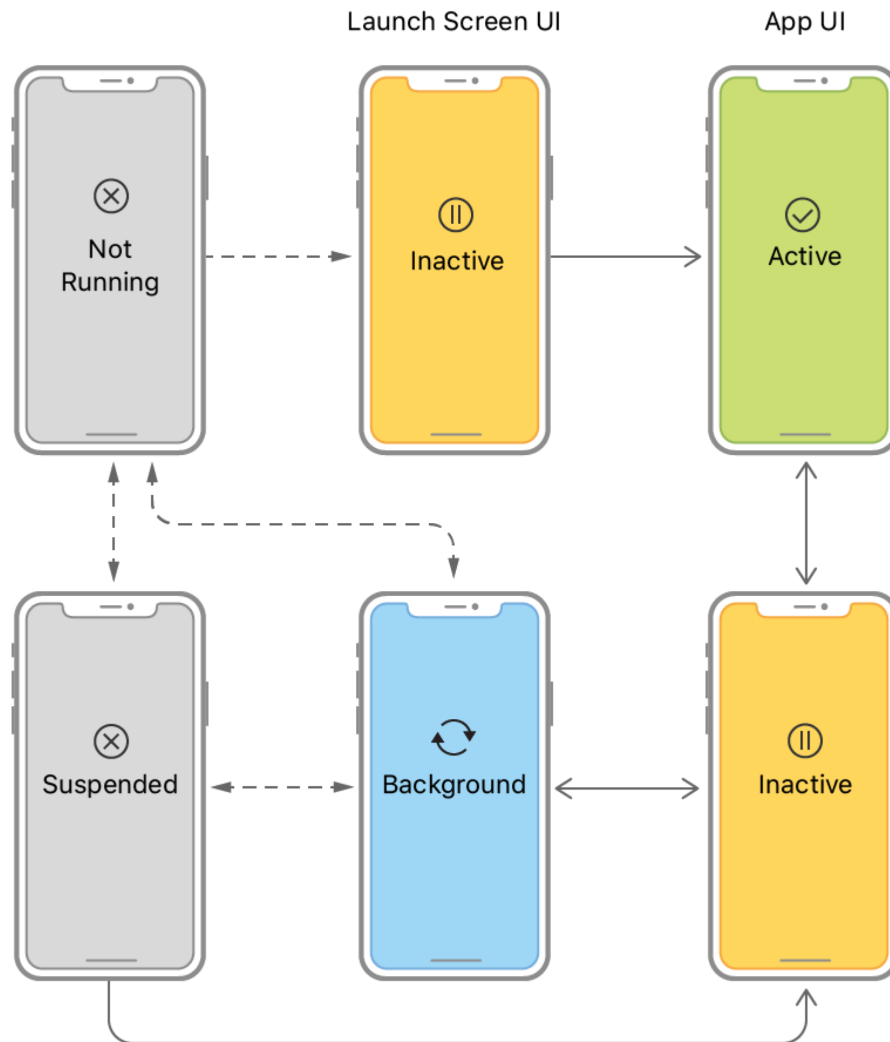
Je důležité uživateli oznamovat problémy, nebo úspěchy pomocí zpráv v aplikaci. Jednou možností, jak toho dosáhnout, by byly notifikace, avšak není potřeba mít takové zprávy na systémové úrovni. Pokud například selže ukládání, když se aplikace zrovna používá, je třeba zobrazit upozornění uvnitř aplikace. SwiftUI pro to možnost nemá, tudíž jsem se opět obrátil na knihovny třetích stran. Aktuálně existuje malé množství knihoven nativně podporujících SwiftUI, ale protože lze s trochou úsilí použít i komponenty UIKitu uvnitř SwiftUI, rozhodl jsem se použít jednu z dostupných UIKit knihoven, konkrétně knihovnu Loaf[2]. Umožnuje

[2] https://github.com/schmidyy/Loaf

zobrazení buď jednoho, ze čtyř standardních stylů, úspěch, varování, chyba a informace, nebo stylu s vlastní barvou a ikonkou. Tato upozornění pak lze zobrazit uvnitř aplikace, buď na horní, nebo spodní straně displaye, s několika možnostmi animací při zobrazení a skrytí. Zprávy jsou neblokující, uživatel může používat aplikaci i když je zpráva zobrazena, lze je skrýt přetažením v opačném směru, než se ukázaly a nebo počkat, než sama zmizí. Používají se například když uživatel začne aktivitu, pro oznámení, že vše proběhlo v pořádku a může začít, nebo pro oznámeni jakékoliv chyby, například při ukládání dat. Jak jsem již řekl, knihovna je vytvořena pro UIKit, lze ji ale použít na SwiftUI, zprávy se zobrazují na takzvaném rootViewController, což je controller prvního a hlavního view, která aplikace používá. Ve SwiftUI existuje, ale standardně se k němu programátor nepřistupuje, nachází se v kolekci windows na sdíleném objektu UIApplication

6.1.3 Životní cyklus iOS aplikací

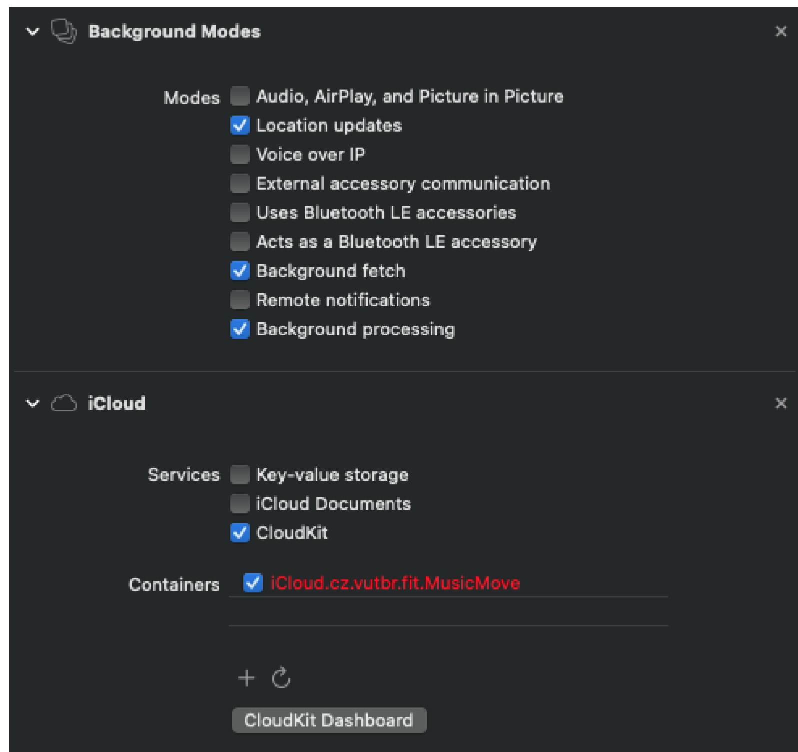
Každá iOS aplikace má 5 stavů ve svém životním cyklu. Ilustrováno na obrázku 6.4, který byl převzat z vývojářské dokumentace Apple [6]. Na počátku aplikace neběží, buď ji uživatel ještě nikdy nezapl, byla násilně vypnuta, nebo se systém nedávno nastartoval a nebyla po staru zapnuta. V tomto stavu nedokáže aplikace nic dělat a nemá přistup k žádným zdrojům. Po zapnutí přechází do stavu Inactive na popředí. Programátor může dát aplikaci Launch Screen, který je v tomto stavu zobrazen. Není možné aplikaci ovládat a posílat jí události, dalo by se říci, že tento stav je pouze načítací a aplikace si shromažďuje potřebné zdroje ke svému běhu. Až tento proces dokončí, přejde do aktivního stavu. V případě existence Launch Screenu je schován a místo něj zobrazeno hlavní View aplikace. Ve SwiftUI existují scény, lze vytvořit scénu pro každý operační systém, na kterém aplikace běží a zobrazit tak jiné View na jiných zařízeních. Hlavní scéna je pak startovním bodem aplikace při přechodu z neaktivního stavu po startu do aktivního stavu, ve scéně je vybráno relevantní view, to je následně zobrazeno uživateli. V případě mé aplikace existuje pouze jedna scéna a to pro systém iOS, která obsahuje MainView, to je první zobrazitelné View aplikace. Ve Storyboard návrhu lze pak designovat jakékoliv View jako hlavní. Avšak nelze vytvořit větvení na základě operačního systému. Až uživatel dokončí svoji práci a bude chtít aplikaci zavřít zmáčknutím domovského tlačítka, nebo na nových zařízeni, přetažením domovské lišty nahoru, aplikace bude uvedena opět do neaktivního stavu, avšak zde v pozadí, je uložena uživatelova činnost a aktuální pozice uvnitř aplikace. Například, pokud je aktuálně zobrazeno View s nastavením, systém si tuto pozici zapamatuje a při probuzení nezačíná cyklus od Launch Screen view a hlavního View, ale rovnou je uživatelovi prezentováno jeho poslední View. V tomto se tento stav liší od prvního neaktivního stavu, není zde zobrazeno žádné View, a aplikace má limitovaný čas provést poslední operace na pozadí, například uložení dat. Po vypršení Inactive časovače se aplikace nachází ve stavu Background. Opět nepřijímá žádné události od uživatele, ale může vykonávat události na pozadí, které byly dopředu naplánovány a nastaveny. Aby to ale mohla aplikace dělat, je třeba ji nejprve dát tuto pravomoc. Aplikace běžící příliš dlouho na pozadí, bez vykonávání akci, jsou uvedeny do Suspended stavu, ze nemohou vykonávat ani úkoly na pozadí, i kdyby na to měly pravomoci. Je nutné podotknout, že aplikace není sama ukončena natvrdo, to musí učinit uživatel manuálně, například vyhozením z listu aplikací v "task manageru", nebo restartem systému, jinak je aplikaci vždy přidělena určitá paměť. Když se uživatel rozhodne k aplikaci vrátit, je opět nejprve uvedena do neaktivního stavu, pokud už dříve běžela a je ve stavu Suspended, nebo Background, není vyvolán Launch Screen, ale opět se přejde na poslední aktivní View.



Obrázek 6.4: Životní cyklus aplikace

6.1.4 Speciální funkcionality aplikace

V základu má každá aplikace přistup pouze k nejdůležitějším vlastnostem potřebným k chodu. Může přijímat vstupy od uživatele, ukládat data lokálně do zařízení, nebo být připojena k internetu, kupříkladu. Ne vždy tyto základní funkcionality stačí, některé vlastnosti lze aktivovat importem příslušné knihovny a správnou inicializací jejich tříd, jedna z takových je možnost sledování lokace zařízení a zasílání požadavků na modul GPS. Jiné je třeba první řadě povolit ve vývojovém prostředí v sekci Entitlements, takzvané nároky aplikace.



Obrázek 6.5: Příklad nastavení Entitlements

Na obrázku 6.5 je zobrazeno nastavení mé aplikace. Jsou zde zobrazeny pouze nastavení vlastností, na které má aplikace nárok. Pro přidání nového je nutné ho vybrat z nabídky. Ta je příliš obsáhlá na to, abych ji zde popisoval, celý seznam je k dispozici na stránkách Apple[3], zaměřím se pouze na ty, které jsem použil a některé další důležité.

Background Modes vlastnost umožnuje vykonávat zvolené činnosti když je aplikace ve stavu `background`. Je třeba zvolit typ činnosti, která má být povolena a následně implementovat chování. V mé aplikaci jde o povolení vykonávání požadavků na sběr GPS dat v pozadí pomocí módu Location Updates, více o průběhu sběru dat je pak popsáno v následující podkapitole 6.1.8. Následné, pro zisk a zpracování dat dalších dat, například ze Spotify API, jsou povoleny módy Background Fetch a Background Processing. Názvy dalších módů efektivně reprezentují, co aplikaci dovolí.

Pro připojení ke CloudKit databázi je nutné přidat nárok iCloud. Zde se vybere, jakým způsobem má aplikace komunikovat s cloudovým úložištěm. Na uložení CoreData databáze do CloudKitu je třeba povolit službu CloudKit, následně se zobrazí nabídka kontejneru, do kterých bude databáze uložena. Lze vybrat jakýkoliv kontejner ke kterému má vývojář přístup. Pro vytvoření nového musí zajít na CloudKit Dashboard, kam se dá dostat kliknutím z nastavení, zde se pak pomocí tlačítka + vybere kontejner a přidá do aplikace. Je možnost synchronizovat data s více kontejnery.

Mezi další pravomoci, které může aplikace získat, patří třeba možnost přihlášení se do aplikace pomocí Apple účtu. Vlastnost, kterou Apple do systému iOS přidal teprve nedávno, umožnuje uživateli alternativu k vyplnění registračních údajů pro vytvoření účtu, nebo přihlášení uvnitř aplikace. Třetí strana tímto způsobem nezíská žádné heslo, pouze informace, které potřebuje k identifikaci uživatele a ty dopředu oznámí. Uživatel si může

[3] https://developer.apple.com/documentation/bundleresources/entitlements

vybrat, zda třetí straně sdělí svůj email asociovaný s Apple účtem, nebo zda jí bude poskytnut proxy email od Apple, který přeposílá emaily na uživatelovu adresu. Ačkoliv tato funkcionalita není použita v mé aplikaci, protože používám účet, který je přihlášen v systému, je tato nová funkcionalita důležitým snahy společnosti Apple o zvýšení bezpečnosti a soukromí svých uživatelů.

6.1.5 Databáze a propojení s rozhraním

Jak již bylo zmíněno v předchozí kapitole, databáze byla implementována v CoreData s využitím technologie CloudKit. Implementace navrženého diagramu tabulek byla v celku jednoduchá, vývojové prostředí XCode disponuje vizuálním editorem entit, ve kterém lze realizovat všechna propojení a nastavení integrace CloudKit. Výsledné entity se od návrhu mírně liší. Pro integraci s CloudKitem bylo nutné všem vazbám dát i vazbu zpětnou, což znamenalo doplnit entity o nové atributy. Při standardním lokálním použití CoreData tento požadavek není.

Je možné vytvářet One-To-One, nebo One-To-Many vazby, ekvivalentně vazbám 1:1 a 1:N v v SQL návrhu. Pro realizaci vazeb N:N je třeba ručně vytvořit novou pomocnou tabulku, CoreData tento přistup automaticky nedovoluje. Vazby mohou mít pravidla co se má stát s podřazenou entitou, pokud se nadřazená entita smaže, toto pravidlo se nazývá Delete Rule a je podobné, jako ON DELETE trigger v SQL. V CoreData jsou čtyři možnosti, No Action, neprovede s podřazenou entitou nic, ta si nadále mysli, že nadřazená existuje a je ve vazbě. Nullify pravidlo vynuluje cíl vazby, podřazená entita nadále existuje, ale již není provázána s nadřazenou. Cascade je ekvivalentní ON DELETE CASCADE, všechny podřazené entity ve vazbě s nadřazenou budou smazány také. Toto je ideální, pokud existence podřazené entity nemá smysl bez nadřazené. Poslední pravidlo, Deny, zakáže smazat nadřazenou entitu, dokud se buď neodstraní vazba mezi entitami, nebo nesmaže podřazená entita úplně, až poté je možné nadřazenou smazat [9].

Další změnou bylo vynecháni entity ActivityType. Vytvářet entitu s pevně danými daty se ukázalo jako nepraktické a nepřinášelo mnoho benefitů, vynechal jsem proto celou entitu a tento vztah a typ aktivity přidal jako atribut entity Activity, jehož hodnota je nyní řetězcová a je naplněna z výčtového typu.

Pro realizaci nastavení aplikace a možnou přenositelnost těchto nastavení jsem zvolil vytvoření nové entity v CoreData. Alternativním přístupem by bylo použití třídy `UserDefaults` pro uložení jednoduchých nastavení pouze lokálně a na každém zařízení zvlášť. Protože bylo ale cílem přistupovat k datum z více zařízení a mít je synchronizována, dal jsem nastavení do vlastní entity, která obsahuje pouze jednu relaci a ta se aktualizuje v případě změny. Při prvním zapnutí aplikace je provedena kontrola, zda tato entita existuje, a pokud ne, tak se vytvoří nová, se kterou se dále pracuje. Je zde uloženo vše, co uživatel může změnit v nastavení, včetně klíče pro přistup k API hudebního přehrávače.

Nová data jsou lokálně uložena ihned, po vyvolání metody `ViewContext.Save()` a na cloudové uložiště se uloží, pokud má zařízení přístup k internetu. Načtení dat proběhne instantně v případě načítání lokálních a poté se synchronizují s cloudovými. Ve ViewController strukturách je možné přistoupit ke sdílené aplikační proměnné viewContext typu NSManagedViewContext. Ta se stará o správu CoreData, je možné s ní vytvářet nové entity, načítat existující data, upravovat a ukládat stávající, nebo je mazat. Přistup probíhá stejně, jako k proměnné pro vzhled systému, přes klíčové slovo @Environment. Pokud je třeba uložit mimo strukturu implementující View, není v tomto kontextu proměnná dostupná jako systémová proměnná, ale je třeba k ní přistoupit přímo přes předem vygenerovanou třídu

Persistence, obsahující singleton objekt viewContext. Tato třída se inicializuje při zapnutí aplikace, tudíž je to také dobré místo pro kontrolu existence dat entity Settings. Při inicializaci se pokusí získat data, a pokud žádná neexistují, jsou vytvořena nová. Uživatel sám záznamy Settings smazat nemůže, tudíž je nepravděpodobné, že by se tento proces prováděl vícekrát, ale aplikace je na to připravena. Protože je SwiftUI reaktivní knihovnou, jakákoliv změna dat, které jsou zobrazeny v uživatelském rozhraní, vyvolá přepsaní rozhraní a aktualizaci. Načtení a navázaní dat na rozhraní jsem z velké části ponechal na systémových metodách. Ve Views je k proměnné viewContext přistoupeno a vyvolán požadavek na potřebná data, ty jsou načtena do kolekce, která je následně zobrazena v některém z knihovních View. Detailní view pro editaci, nebo zobrazení více informací z entity, nenačítají data, ale jsou jim předány z předchozích view. Při uložení po editaci, nebo po vytvoření nové entity, stačí pouze vyvolat uložení ViewContextu, ten se již postará o to, aby byla entita zařazena správné do kolekce. Pro entitu Settings bylo potřeba tento přistup pozměnit, Data jsou standardně načtena jako kolekce, avšak v uživatelském rozhraní se pracuje pouze s první položkou kolekce.

6.1.6 Požadavky na databázi

Pro získání dat se používá generické třídy `NSFetchRequest`. Lze pomocí ní vykonávat komplexnější query jednodušeji, než v jazyce SQL. Na začátku je třeba vytvořit novou instanci s typem a názvem entity, kterou chceme získat, následně je možné na objektu specifikovat query nastavením vlastností. K limitování počtu dat je celočíselná proměnná `fetchLimit`, ekvivalentní SQL klazuli `LIMIT [n]`, ta vrátí prvních N řádku požadavku. Pokud bychom chtěli přeskočit určitý počet relací, je pro to další celočíselná proměnná `fetchOffset`, toto je první z proměnných, pro které v některé z verzí jazyka SQL nemá ekvivalent. v SQLite to však je klauzule `OFFSET [n]`. Při implementaci stránkováni, nebo načtení pouze aktuálně zobrazitelného počtu dat v seznamu se hodí celočíselná vlastnost `fetchBatchSize`.

Seřazení entit je jednoduché a provádí se vytvořením instancí třídy `NSSortDescriptor`. Každému objektu lze nastavit klíč, odpovídající názvu atributu relace, následně, zda má být podle tohoto klíče řazeno vzestupně, nebo sestupně. Je možné nastavit i vlastní komparátor, který se použije pro porovnání a seřazení hodnot v dotazu a určení pořadí, ve kterém budou. Pro objekty SortDescriptor je na třídě FetchRequest kolekce `sortDescriptors`, do které je možné vložit více objektů, ty jsou poté vyhodnocovány od prvního vloženého. SortDescriptors odpovídají klauzulím `ORDER BY` v SQL, ale mohou být komplexnější.

Bližší upřesnění požadavku je možné pomoci predikátu v proměnné `predicate` typu `NSPredicate` [12]. Zhruba odpovídají klazuli `WHERE`, ale umožňují jednodušší tvorbu větších dotazů. Predikáty používají vlastní jazyk, založený na logických podmínkách, výsledek každého predikátu musí být pravda, nebo nepravda. Podobně jako SQL WHERE umožnují i operace nad daty při vyhodnocováni. Například při práci s řetězci v SQL je třeba použít klíčové slovo LIKE, následované podmínkou, jak má řetězec vypadat. V predikátech existují klíčová slova pro různé operace nad řetězci, každé z nich má levou a pravou stranu výrazu, na levé straně je většinou název atributu, který je porovnáván, ale může to být libovolný výraz a lze tak kombinovat více operací, na pravé straně je pak výraz, se kterým je levá strana porovnána. Tabulka 6.2 obsahuje seznam možných řetězcových operaci a jejích ekvivalentní zápis v SQL. V operacích LIKE je možné použít wildcard symboly ? a *. Operace MATCHES pak podporuje regulární výrazy.

Predikáty se zapisují do řetězce, do atributu `format` při inicializaci `NSPredicate`. Pokud má predikát používat data z proměnné, je třeba ve formátu použít klíčová slova uvozená

Predicate	SQL
name `BEGINSWITH` foo	LIKE 'foo%'
name `CONTAINS` foo	LIKE '%foo%'
name `ENDSWITH` foo	LIKE '%foo'
name `LIKE` foo	LIKE 'foo'
name `MATCHES` foo\d{1,3}	LIKE 'foo\d{1,3}'

Tabulka 6.2: Predicate operace nad řetězci a SQL ekvivalenty

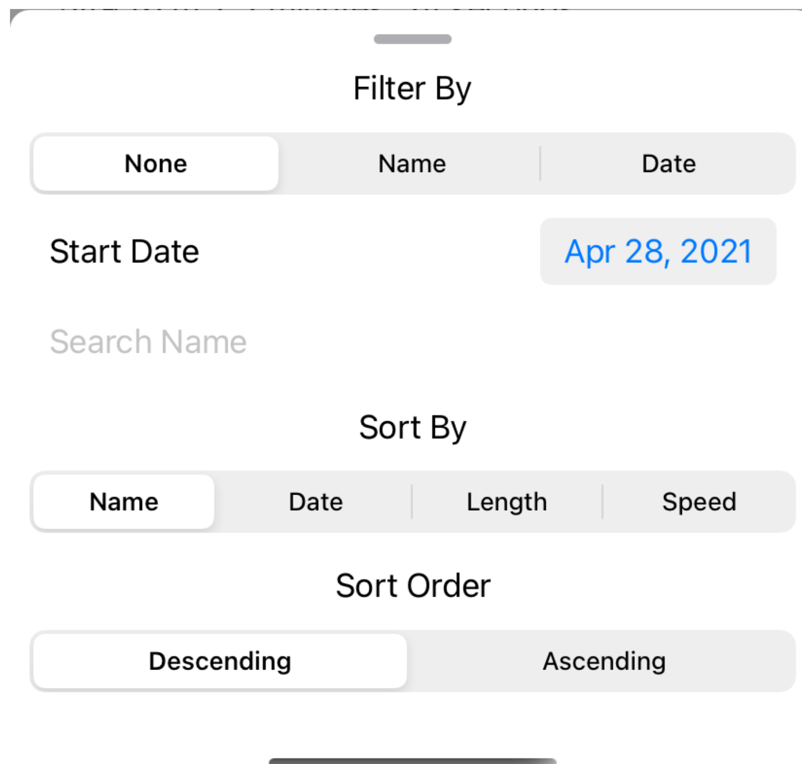
procentem, následně proměnné dodat konstruktoru jako další argumenty. V případě, že by byl obsah proměnné zadán do formátu přímo, mohlo by dojít k ekvivalentu SQL Injection. Pokud by se jednalo o proměnnou obsahující uživatelský vstup, uživatel by jí mohl naplnit vlastním predikátem a ovlivnit tak výsledky. Predikáty je možné použít i mimo CoreData, třeba pro filtrováni obsahu lokálních kolekcí.

Řetězcové operace nejsou jediné, v čem se liší. Predikáty nabízí operace nad polem hodnot, Například, pokud bude výsledkem po vyhodnocení výrazu pole hodnot, lze ho indexovat jako pole v programovacích jazycích v hranatých závorkách. Krom indexů nabízí operace tři klíčová slova, FIRST, pro první prvek, LAST, pro poslední prvek a SIZE pro velikost pole. Operace nad polem patři do skupiny agregačních operaci, dalšími příklady z této skupiny jsou relační operace ANY, ALL, NONE a IN. Na pravé straně těchto operaci musí být logická podmínka, výsledkem výrazu jsou pak hodnoty splňující podmínku.

Ve Swiftu a Objective-C pro Apple je možnost kombinovat predikáty buď přímo ve formátu a následným spojením logickou spojkou, nebo pomocí třídy NSCompoundPredicate, té se specifikuje typ spojení a pole predikátu. Každý predikát z pole je poté spojen stejnou spojkou do jednoho.

FetchRequest vrací generickou kolekci FetchResults s typem entity, která byla dotazována. Je však možnost FetchRequest upravit tak, aby vrátil pouze některé atributy entity, názvy atributu se pak zadávají do pole `propertiesToFetch`. Požadavek vrátí vždy objekty zadané entity, proto musí být typ nastaven. Dotaz lze vyvolat manuálně metodou `execute`. Ve SwiftUI se tento přistup však nevyužívá. Pokud je potřeba data získat ve View. Jsou v podstatě dva přístupy, jak o data požádat. Buď se vytvoří funkce, která vrací FetchRequest dopředu, poté o něj stačí ve View požádat, nebo se FetchRequest píše přímo. Pokaždé, když chceme, aby ve View byla kolekce zobrazující data přímo z databáze, je potřeba proměnnou uvodit klíčovým slovem `@FetchRequest`. Následně v parametru lze předat náš předem vytvořený FetchRequest, nebo ho vytvořit přímo. Aby byla zajištěna konzistence dat, je nutno každému požadavku dát alespoň jeden NSSortDescriptor, bez něj by se mohlo stát, že data budou při každém obnovení načtena v jiném pořadí.

Bylo zmíněno, že uložení změn probíhá pouhým zavoláním metody Save na ViewContextu. Při tvorbě nové entity je třeba ji vytvořit za pomoci tohoto ViewContextu, aby o ní věděl. Ten si pak vede správu všech entit, které aktuálně existují, a umí rozpoznat, zda se na některé provedla změna. Tyto entity jsou v modifikovaném stavu, při uložení jsou uloženy všechny změny na všech entitách, které dosud nebyly uloženy. Pokud je databáze připojena na CloudKit, jsou synchronizovány change tags. Nově vytvořené entity jsou také uloženy všechny naráz. Pokud zařízení aktuálně nemá přístup k internetu, jsou uloženy entity nejdříve lokálně a při připojení je provedena synchronizace s CloudKitem.



Obrázek 6.6: Popup s filtery

Praktické využití v predikátu a seřazení je pro filtrování a řazení seznamu aktivit. Ve View obsahující uživatelovy uložené aktivity je na horní liště tlačítko, které otevře vysouvací view, jako je tomu na obrázku 6.6. Zde jsou dvě sekce, výběr ze tří možností filtrování, hodnoty pro filtrování a pole, podle kterého se mají výsledky seřadit. Programově je list aktivit realizován jako další view, které má parametry pro filtrovanou položku, hodnotu a řazení. Při změně hodnoty výběru je View aktualizováno a výsledky znovu nahrány. Data pro datum začátku aktivity jsou od uživatele získány pomocí komponenty DatePicker, která je omezena pouze na výběr dne měsíce a roku. Komponenta však vrací přesný datum a čas, tudíž při vzniku predikátu jsou vytvořeny dva datové objekty, počáteční datum, nastaveno na začátek dne, a koncové datum nastaveno na konec dne. Predikátem jsou poté filtrovány výsledky mezi těmito daty. Filtrování názvu je case insensitive a je použit predikát CONTAINS na atribut `name`. Kód pro `FilterView` byl z části převzat ze stránky *Hacking With Swift*[4]. V konstruktoru je vytvořen nový FetchRequest, pro filtrování je mu předán predikát, na základě parametru. Řazení probíhá přes SortDescriptor, opět při konstrukci FetchRequestu. Získaná data jsou zobrazena v listu, pomocí cykle ForEach, a každá položka odkazuje na svoje konkretní detailní zobrazení.

6.1.7 Propojení s úložištěm CloudKit

Všechna data, která aplikace ukládá, jsou pouze v rukou uživatele. Systém CloudKit umožnuje vývojářům ukládat data do takzvaných kontejnerů [4]. Na výběr je buď kontejner veřejný, jehož data vidí jak vývojář, tak všichni uživatelé, je sdílen všem uživatelům a mohou

[4] `https://www.hackingwithswift.com/books/ios-swiftui/dynamically-filtering-fetchrequest-with-swiftui`

z něj číst, nebo do něj zapisovat, bez nutnosti přihlášeni, nebo druhý soukromý kontejner, do kterého už vývojář nevidí. Uživatel se musí autentizovat svým Apple účtem a následně získá do něj přístup. Vývojář aplikace se nikdy nedozví uživatelovy údaje, správa kontejneru je totiž v režii operačního systému iOS a jeho synchronizace probíhá na systémové úrovni. Díky tomu je aplikace bezpečná a uživatel nemusí mít strach, že jeho citlivá data o poloze zneužije třetí strana.

Pro přidání podpory CloudKitu je nejprve potřeba, aby autor aplikace měl plnohodnotný vývojářský učet u Apple. V současné době jsou dva typy vývojářských účtu. První je zdarma a stačí mít Apple účet, který pak lze povýšit na vývojářský. S ním je možné vyvíjet aplikace, avšak bez možnosti propojení se spoustou systému v Apple ekosystému, jako je tomu CloudKit. Také nelze aplikace publikovat na AppStore, tudíž je tento typ účtu určen spíše pro začínající vývojáře, nebo pro vývoj soukromých aplikací. Druhý, plnohodnotný typ účtu je placený. Za roční poplatek získá vývojář plný přistup ke všem systémům a může své aplikace publikovat. Vývojář si přistup buď koupí, nebo se lze zapojit do teamu, který vlastní tuto plnohodnotnou licenci a jeho účet bude pak povýšen.

Následně je třeba vytvořit novou databázi na portálu pro správu CloudKit kontejnerů, dále v nastavení aplikace ve vývojovém prostředí se přidá možnost propojení s CloudKit databázi. Z nabídky se vybere ta správná a zvolí mód, buď development, pro vývoj a testování, nebo production, pro data po vydání aplikace. Existující CoreData data a schéma databáze v aplikaci lze převést na CloudKit, avšak je třeba splnit podmínku, že všechna propojení entit budou mít i zpětné vazby. V návrhu schématu CoreData je možné zvolit, které entity budou nahrány na cloud a které ne, a tak je možnost mít í lokální data v jedné aplikaci.

Záznamům v CloudKit databázi se říká Records, každý z nich má svoje ID, pokud není specifikován, je to náhodné vytvořené UUID. Toto pole se nazývá `recordName`, a lze podle něj databázi indexovat. Záznamy mají i časová data o vytvoření a poslední editaci záznamu, tyto atributy nelze editovat a jejich správa je v rukou databázového systému. Jednotlivé tabulky jsou pak `recordType`.

Samotná synchronizace pak probíhá tak, že se nejprve načtou data lokální, poté se pokusí stáhnout data z cloudu. Každý záznam má svůj tag indikující, zda byl záznam změněn, a časové razítko o době změny. Pokud se tyto údaje neshodují, a záznam v cloudu je novější, pak je lokální záznam nahrazen. Pokud však proběhly změny jak lokálně, tak v cloudu, tak jsou záznamy sjednoceny do jednoho, aby lokální byl novější.

6.1.8 Sběr GPS dat

V sekci 6.1.1 jsem popsal, že aby aplikace mohla přistupovat ke GPS datům a získat tak lokaci zařízení, je potřeba jí udělit povolení. Pro práci s GPS lze použít systémové knihovny CoreLocation, které obsahují všechny potřebné funkce, od vyvoláni požadavku na přistup k datům, po samotné rozhraní s GPS modulem a zisk dat. Knihovna je dostačující pro práci, avšak je třeba si pomocí poskytnutých metod implementovat vlastní požadavky na modul. Protože tyto problémy již řešilo spoustu vývojářů přede mnou obrátil jsem se na knihovnu třetí strany SwiftLocation, usnadňující práci s GPS modulem. Stačí zde pouze nastavit, jaká data chci získat a s jakou přesností a metoda knihovny dokáže vrátit objekt obsahující souřadnice a pohybovou rychlost. Další důležité vlastnosti, kterou jsem využil, je možnost sběru dat pokud je aplikace na pozadí. Zde bylo třeba nastavit aplikaci a knihovnu, aby vyvolávala přerušení a na chvílí se probudila ze spánku a provedla potřebnou činnost. iOS

nedává aplikacím na pozadí plný přístup, je však možné pro účely přístupu k lokaci, nebo rychlé manipulaci s daty, aplikaci periodicky probouzet ze spánku.

```
SwiftLocation.gpsLocationWith {
    $0.subscription = .continous // do zastaveni
    $0.accuracy = .house
    $0.minDistance = 300 // vzdalenost mezi aktualizaci
    $0.minInterval = 30
    $0.activityType = .automotiveNavigation
    $0.timeout = .delayed(5)
}.then { result in
    switch result {
    case .success(let newData):
        print("New location: \(newData)")
    case .failure(let error):
        print("An error has occurred: \(error.localizedDescription)")
    }
}
```

Výpis 6.3: Nastavení `SwiftLocation`

Periodický sběr probíhá v metodě `SwiftLocation.gpsLocationWith`. Předá se jí nastavení o přesnosti lokace, typ lokačních dat a periodu obnovy a přihlásí se jí metoda, která se volá po uplynutí intervalu. Ukázkový příklad 6.3 byl přezván z dokumentace knihovny SwiftLocation [11].

Nastavení možnosti activityType má vliv na výsledky odpovědi dotazu na GPS lokaci. Jednotlivé možnosti pro nastavení výčtového typu `CLActivityType`, společně s jejich rozdíly při použití, jsou vypsány v tabulce 6.3. Aplikace používá typ fitness. Typy pro navigaci, včetně fitness, mají možnost přerušit snímaní, pokud se zařízení nepohne o vzdálenost určenou možností `minDistance`. Ta je v jednotkách metrů. Vyvoláni dotazu na novou lokaci pak probíhá na základě dvou nastavení, minDistance a minInterval. Pokud jsou splněny obě podmínky, to znamená, že od poslední aktualizace se zařízení pohnulo o určený počet metrů a uplynula určená doba v sekundách, je provedena aktualizace.

Typ	Určení
other	Ostatní určení, nevyhovující jiným možnostem
automotiveNavigation	Specificky pro automobilovou navigaci
fitness	Pro aktivity typu běh, cyklo a chůze, vypnuto snímaní uvnitř budov
otherNavigation	Jiná vozidla, krom automobilů, například čluny
airborne	Pokud se předpokládá pohyb v podstatné výši nad zemí

Tabulka 6.3: Přehled `CLActivityType` a jejich určení

Další možnost je přesnost lokací, v Apple knihovně se nastavuje přesná hodnota na metry v datovém typu float. Knihovna obsahuje předem připravený výčtový typ. V aplikaci je použit typ `house`, odpovídající přesnosti na 50 metrů. Další typy jsou `city`, `neighborhood`, `block` a `room`.

Nastavením subscription lze určit, zda chceme data o lokaci průběžně, nebo pouze jednou. Jedná se opět o výčtový typ, pro jednotný zisk lokace je typ `single` a pro průběžné je na výběr buď z `continous`, nebo `significant`. Použil jsem continous, který poskytuje data splňující podmínky dokud není sběr manuálně zastaven. Typ significant je nevhodný pro aplikaci tohoto typu, poskytuje aktualizace pouze pokud se lokace mezi dvěma aktualizacemi podstatně změní, a ačkoliv díky tomuto šetří baterii, jeho určení je pro sběr nevyžadující tak přesná data o lokaci, jako je fitness aplikace.

Je možné přihlásit více metod s nastaveními. Po zadaném čase se zavolají všechny přihlášené funkce s parametrem obsahující buď výsledek, nebo chybovou hlášku, pokud došlo k chybě. V tom případě se sběr zastaví a uživatel je na tuto skutečnost upozorněn pomocí notifikace v horní části obrazovky. Když se požadavek na data vykoná úspěšně, přejde se k uložení. Metoda má přistup k entitě aktuální aktivity, tudíž se vytvoří pouze nová entita souřadnic, vloží se do ní získaná data o poloze a rychlosti a do průměrné rychlosti aktivity se započítá aktuální rychlost. V případě, že uživatel povolil integraci hudebního přehrávače, dojde i k zisku dat z API.

6.1.9 Data z API hudebního přehrávače

Pro správnou funkčnost je nutné integraci nejdříve povolit a přihlásit se svým účtem ve zvolené službě. Aktuálně aplikace podporuje pouze integraci se službou Spotify, avšak je programována tak, aby bylo případné pozdější rozšíření o více platforem jednoduché jak pro uživatele, tak z pohledu vývojáře.

Autentizace proběhne v nastavení, kde se v integrovaném webovém prohlížecí v aplikaci uživatel přihlásí a vzdálené API následně odpoví s dvěma tokeny, jeden pro přístup a druhý, obnovovací, pro obnovu přístupového po vypršení. Odpověď ve formátu JSON je zpracována při události navigaci prohlížeče na adresu začínající vymyšleným protokolem `musicmove://`.

Aplikaci bylo nejprve třeba zaregistrovat na vývojářském portálu Spotify [14]. Registrace je bezplatná a API je možno používat v libovolném rozsahu v případě, že cílová aplikace je nekomerčního typu a nebude publikována na AppStore, což v mém případě je. Po registraci lze získat aplikační token a heslo, které se používají pro autentizaci aplikace při zisku nového přístupového tokenu. Tyto údaje se nemění a proto jsou uloženy v aplikaci a nikdo jiny krom vývojáře, k nim nemá přistup. Při požadavku o přihlášení uživatele je třeba deklarovat i typy dat, které budou požadovány. Celý seznam je relativně dlouhý a je k nalezení v dokumentaci Spotify API[5], ty jsou vloženy do URL pro autorizaci jako URI komponenty, nahrazující nevhodné znaky jejich UTF-8 reprezentací. Pro správný chod aplikace je použito typů `user-read-playback-state`, díky kterému lze získat aktuálně hrající hudbu, včetně času a pozice přehrávače.

Pokud je integrace povolena a uživatel přihlášen, je při průběhu aktivity periodicky posílán požadavek na API pro zisk aktuálně hrající hudby. Narozdíl od sběru GPS dat je interval mezi požadavky různý. Z důvodů limitace, a protože se s největší pravděpodobností nebudou data měnit v řadu stovek milisekund, jako tomu je u lokačních dat, začíná interval sběru na 30 sekundách. Po odeslání prvního požadavku po startu aktivity se zjistí, zda uživatel hudbu v tento moment poslouchá, a pokud ne, tak se dotaz opakuje za 30 sekund. V opačném případě je z API vytažen zbývající čas do konce hudby a ten je použit jako interval, za který se pošle další požadavek. Pro uložení je vytvořena nová entita Music, která je v relaci s aktivitou a aktuální lokací, je uloženo pouze ID hudby z API a jméno služby a čas. Abych redukoval počet duplicitních dat, nová entita Music je vytvořena a uložena

[5] https://developer.spotify.com/documentation/general/guides/scopes/

pouze tehdy, pokud se hudba změní. V případě, že zasláni požadavku selže, aplikace zareaguje. Pokud je chyba z důvodu přenosu dat, například pro nedostupnost služby, je chyba oznámena a požadavek se opakuje po výchozím intervalu. Stejně tak je požadavek opakován, pokud vyprší platnost přístupového tokenu. V tom případě Spotify API skutečnost oznámí a aplikace zašle požadavek s refresh tokenem, API v případě úspěchu vrátí nový token, který přepíše v databází starý a požadavek na data se opakuje s použitím nového tokenu.

Protože nejsou uložena všechna data o hudbě, je třeba při zobrazení předešlých aktivit načíst tyto data z API. Spotify API umožnuje získat informace o několika položkách v jednom požadavku a vrátit je jako pole JSON objektů, které jsou dále naparsovaný a použity jako zdroj informací v rozhraní.

Každá skladba je součástí nějakého alba, i když je to pouhý single, pro to má i svůj obrázek. Spotify poskytuje tři velikosti obrázků, a protože je třeba je zobrazit na mobilní aplikaci, zvolil jsem pro ušetření dat ten nejmenší. Obrázky ale nejsou přeneseny jako data, ale pouze jako URL odkazující na úložiště s obrázkem. SwiftUI nedokáže zobrazovat data z URL, tudíž je třeba ho nejprve stáhnout. Mohl jsem zvolit standardní způsob komunikace přes HTTP ve Swiftu, ale existuje knihovna, která umožní zobrazení obrázku z URL a zároveň ho uloží do interní cache, aby se nemusely opakovat požadavky na stejný obrázek, pokud je zobrazen na více místech.

Při zobrazení mapy jsou data mezi dvěma záznamy o hudbě dopočítána, dokud se nedojde na souřadnici v relaci s jinou položkou hudby, mapa předpokládá, že v ten okamžik hrála stejná hudba. Každý úsek je barevně odlišen a jeho barva je závislá na externím ID hudby, jenž je rozděleno na tři stejně dlouhé řetězce. Z každého je získána jeho číselná reprezentace, která je následně modulována na hodnotu 255. Tímto algoritmem lze dopočítat barvu stejnou pro každé ID, bez nutnosti napevno vymýšlet různé barvy.

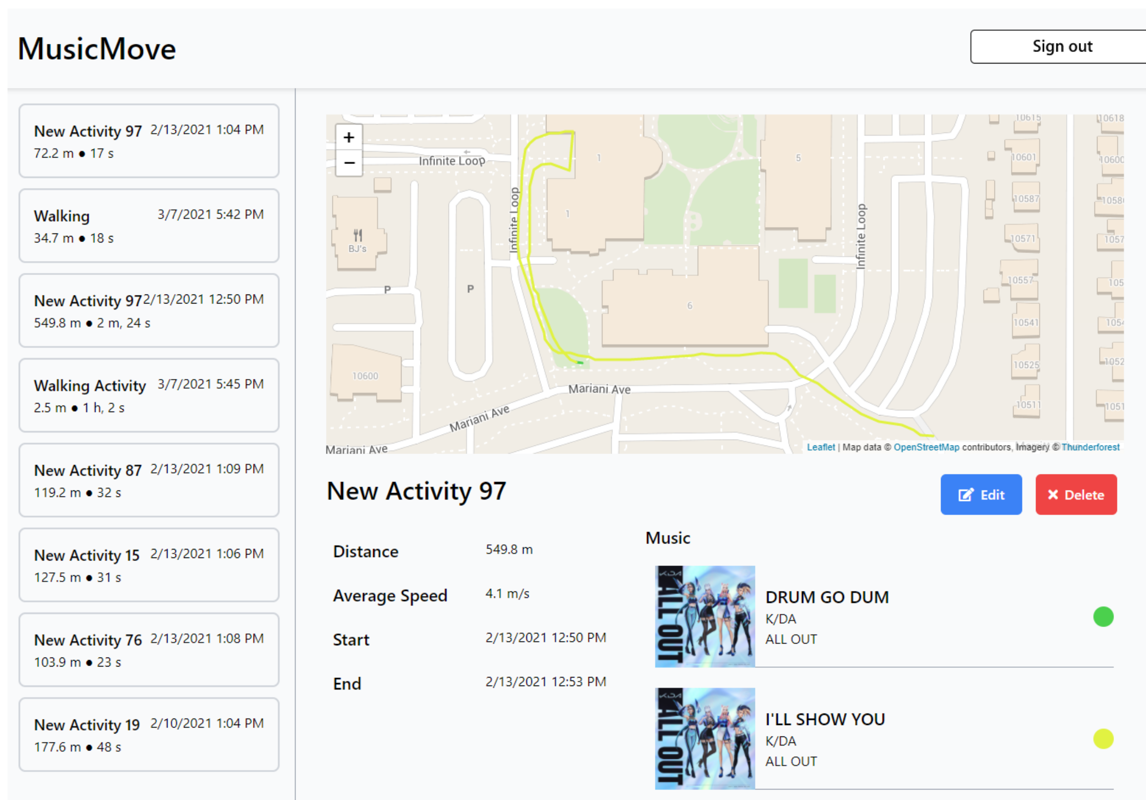
6.2 Webová aplikace

Cílem bylo vytvořit jednoduchou webovou aplikaci s rozhraním podobnému tomu, které je na mobilní aplikaci, aby si uživatel mohl odkudkoliv zobrazit svoje data přehledněji a na větší obrazovce, než je ta mobilní. Protože žádná data nejsou uložena na vlastních serverech a komunikace probíhá výhradně se servery Apple, nebo API Spotify, rozhodl jsem se aplikaci koncipovat jako klientskou, tudíž veškerá činnost probíhá na straně prohlížeče klienta. Pro rozhraní jsem využil standardních nástrojů HTML a CSS, s aplikací knihoven třetích stran. Pro interakci a vykreslení byla zvolena JavaScriptová knihovna Vue.js a pro vzhled css framework `TailwindCSS`.

Celé rozhraní je koncipováno do tři části, příklad ve světlé variantě je na obrázku 6.7. Na horní straně je navigační lišta s názvem aplikace, jménem uživatele a tlačítkem pro přihlášeni, nebo odhlášení. Pokud není uživatel přihlášen, otevře se při vstupu na stránku vyskakovací okno oznamující tuto skutečnost. Bez toho nelze aplikaci použít, tudíž není možnost toto okno zavřít jinak, než úspěšným přihlášením. Jak bylo zmíněno výše, všechna data jsou uložena na straně Apple pod účtem uživatele. Do aplikace se tedy autentizuje pomocí Apple účtu. Tento proces je realizován knihovnou CloudKitJS. Pro autentizaci je odkázáno na stránky Apple, ty následně vrátí token, který si knihovna uloží a dále s ním pracuje.

Po přihlášení jsou natažena uživatelova data. Zde je aplikace rozdělena na dva panely. Levý obsahuje seznam aktivit, seřazen dle času zahájení od poslední uložené. Pravý panel pak detail aktivity. Zisk dat proběhne odesláním dotazu na CloudKit API. Pokud proběhne

úspěšně, je výsledná kolekce dat uložena do datového zdroje Vue aplikace. Ta následně dynamicky vloží nové HTML elementy na stránku. Z pohledu vývojáře není třeba vytvářet elementy ručně. V HTML struktuře jsou tagy, kde se má kolekce zobrazit, a které parametry se mají zobrazit kde. Stejně jako SwiftUI, Vue.js využívá znovupoužitelných komponent. Každá položka je tedy realizována jako komponenta, které se předá objekt s daty, ty jsou následně dosazeny do HTML.



Obrázek 6.7: Webová stránka ve světlé variantě

Vue obsahuje i systém pro zpracování událostí a uživatelských interakcí, po kliku na položku v levé časti stránky se otevře detailní pohled na aktivitu. Data z iCloud API už jsou v této době načtena a pouze předána komponentě detailního zobrazení, avšak v zájmu ušetření objemu přenesených dat, se data z hudebního API se načítají až při otevření. Pro toto se použije stejný token jako ten, který nastaví uživatel v mobilní aplikaci, zde je tedy další využití entity Settings. Z nově získaných dat se poskládá zobrazení. Stejně, jako na mobilu, je hlavní komponentou mapa. Zde byla použita knihovna Leaflet.

Knihovna jako taková umožnuje pouze interakci s mapou, neposkytuje vlastní data, a tak je potřeba využít vlastní zdroj. Nejprve se vybere plocha, na které se má mapa zobrazit, Leaflet této ploše dodá ovládací prvky, jako na běžných mapách. Následně lze vytvářet nové vrstvy, je možnost vykreslit například předpřipravené UI prvky, jako je tomu v komponentě MapKit v mobilní aplikaci, nebo vytvořit vrstvu vektorových objektů. Tento typ vrstvy byl použit pro vykreslení trasy aktivity. Leaflet byl použít z důvodu dostupnosti a jednoduchosti nastavení. První možnost, kterou jsem zvažoval, byla knihovna Apple MapKitJS. Jejím problémem však bylo, že vyžaduje existenci serveru a implementaci autentizace promocí JWT tokenu, což nebylo zcela ideální, pokud mým cílem bylo vyhnout se tvorbě autentizace

a správy zabezpečení serveru. Leafet je veřejně dostupná, stačí ji importovat jako node package, nebo pouze jako JavaScriptový soubor. Pro vrstvu s mapovými daty jsem zvolil službu Thunderforest, nabízející API s geografickými daty pro veřejné použití.

Stejně jako Spotify API, je u Thunderforest třeba aplikaci zaregistrovat a požádat si o přistup k datům. V aplikaci je využíván bezplatná úroveň přístupu, limitována na 150 000 požadavků na mapový čtverec za měsíc. Při porovnání konkurenčních mapových API mně tato možnost přišla nejvíce fér a v případě přesažení limitu není vývojáři účtováno extra. Například jedna z nejpoužívanějších služeb MapBox, nabízí pouze 50 000 požadavků měsíčné, i když je nepravděpodobné, že by moje aplikace tyto limity v nejbližší době přesáhla. Případná výměna API je v knihovně Leaflet jednoduchá. Knihovně se předá vzorová URL, jako tomu je v argumentu metody `tileLayer` na příkladě 6.4, přezvaném z dokumentace Leaflet.js [1], na ni se posílají požadavky o část mapy na aktuálně zobrazovaných souřadnicích. Většina API používá stejný vzor se souřadnicemi x, y, z. Pokud by tomu bylo jinak, lze vzor nastavit při inicializaci nové mapy. Místa, na která se mají doplnit data, se ve vzoru obalí složenými závorkami. Thunderforest také vyžaduje, aby v URL na data byl API klíč, který vývojář dostane po registraci. Protože je klíč uložen pouze v JavaScriptu, ke kterému má uživatel přistup, mohla by zde být bezpečnostní díra a uživatel by mohl klíč zneužít. Naštěstí, API podporuje limitování příchozích požadavků dle URL adres, aby se tomuto předešlo.

```
L.tileLayer('http://foo.xyz/{z}/{x}/{y}/{id}.png?token={accessToken}',
    {
        attribution: '... copyright ...',
        maxZoom: 18,
        id: 'streets',
        tileSize: 512,
        zoomOffset: -1,
        accessToken: 'your.access.token'
    }
).addTo(mymap);
```

Výpis 6.4: Inicializace mapy v leaflet.js

Vykreslení mapy probíhá podobně, jako v mobilní aplikaci, jednotlivé úseky jsou opět barevné rozděleny na základě hudby, a pokud žádná hudební data nejsou, je použita výchozí, šedá barva. Je zde podstatný rozdíl v tom, že SwiftUI neumožňovalo jednoduše vykreslovat čáry na mapě, Leaflet to dokáže vcelku jednoduše. V kolekci souřadnic stačí nalézt počáteční a koncový bod a knihovna mapu vykreslí linii mezi nimi. Takto jsem sestrojil pole objektů o hodnotách latitude a longtitude a následně je položil na mapu. Každá linie se dá následně upravovat a navázat na ní události. V tomto místě je čáře nastavena barva, dle hudby. Mapu takto třeba lze inicializovat jednou. Všechny objekty nakreslené na mapě jsou uloženy do proměnných a při znovuotevření, nebo rozkliknutí nového detailu, jsou objekty z mapy odstraněny a nahrazeny novými.

Pod mapou jsou opět stejné detailní informace a seznam hudby, kterou uživatel poslouchal. Uživatel má možnost u každé aktivity editovat její název. Po povolení editačního módu se opět textové pole změní na vstup, ve kterém lze změnit název a ten poté uložit. Lokálně se Vue.js postará o to, aby byl text lokálně přepsán instantně. Na úložiště je poté uložen požadavkem na změnu objektu.

Stránka byla vytvořena pouze pro užití na desktopu, pro mobilní uživatele je zbytečná, protože mohou k datům přistoupit z aplikace. Bylo proto třeba uživatelům, kteří se pokusí na stránku přistoupit z mobilní sdělit tuto skutečnost. Aktuálně JavaScript nedisponuje žádným systémem, nebo jednoduchou cestou, jak zjistit, že byl vyvolán požadavek z mobilu. Jedinou cestou, jak toho docílit je analyzovat user agent string[6], který posílá každý prohlížeč v hlavičce požadavku. Tento řetězec obsahuje veškeré potřebné informace o prohlížeči, zařízení a operačním systému, na kterém běží. Pro referenci, řetězec pro mobil iPhone v prohlížeči Safari je na příkladu 6.5.

```
Mozilla/5.0 (iPhone; CPU iPhone OS 14_4_2 like Mac OS X)
    AppleWebKit/605.1.15 (KHTML, like Gecko)
    Version/14.0.3 Mobile/15E148 Safari/604.1
```

Výpis 6.5: User-Agent pro iPhone v Safari

Každý User-Agent začíná názvem produktu. Z historických důvodů se dochovalo[7], že velká část prohlížečů zasílá řetězec jako produkt Mozilla, i když tomu tak ve skutečnosti nemusí být. Následují informace o systému, na příkladu to je zařízeni iPhone se systémem iPhone OS (iOS) ve verzi 14.4.2, podobné jako Mac OS X (dnes už pouze macOs). Dále platforma prohlížeče, neboli vykreslovací jádro, na kterém běží, pro Safari to pak je `AppleWebKit`, jeho verze a případné detaily, zde to pak je open source jádro KHTML. Poté verze prohlížeče, platforma, na které běží, a samotný název prohlížeče a jeho číslo sestavení, název paradoxně stojící na samotném konci, až za svojí verzí.

Je zde patrné, že z User-Agent řetězce lze zjistit mnoho o uživateli a prohlížecí, včetně dožadované vlastnosti rozpoznaní mobilního, nebo desktopového zařízení. Zvolil jsem proto velice jednoduchou knihovnu `mobile-detect.js`[8], které stačí při inicializaci předat User-Agent řetězec a ona z něj dokáže získat relevantní data. Při načtení stránky je tedy zjištěn prohlížeč uživatele a v případě, že se jedná o mobilní, je mu místo stránky zobrazeno varování a v případě, že jde o iOS zařízení, je odkázán na mobilní aplikaci.

Přestože je jednoduché takto zjistit informace o prohlížeči, není tento způsob 100% spolehlivý. Uživatel si totiž může nastavit vlastní User-Agent, a předstírat, že se jedná o jiný prohlížeč, nebo systém, než je skutečně pravdou. Tento proces se nazývá user-agent spoofing a není dobré ho praktikovat, protože jak již bylo zmíněno, každý prohlížeč může podporovat jiné vlastnosti a funkce, a spousta stránek může spoléhat na tento řetězec, aby rozeznala jakou verzi stránky může zobrazit a jaké funkce použít. Ve skutečnosti drtivá většina běžných uživatelů nemá potřebu předstírat, že používají jiný prohlížeč, proto je tento přistup dostačující, i když ne zcela ideální.

6.2.1 Vizuální styl stránky

Zmínil jsem, že pro vzhled jsem zvolil knihovnu `TailwindCSS`. Je nabízena ve dvou verzích. Ta základní je zadarmo a nabízí pouze předpřipravené třídy stylu, které je třeba aplikovat na vlastní elementy. Druhá verze je již placena a obsahuje i hotové komponenty. V zájmu větší volnosti z hlediska vzhledu a ušetření používám základní stylovací knihovnu. Je nainstalována jako node package a je třeba ji importovat do vlastního souboru se styly.

[6] Více o User-Agent na https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent
[7] První prohlížeč Netscape měl agenta Mozilla/1.0, ostatní prohlížeče toto přejaly [2].
[8] https://github.com/hgoebl/mobile-detect.js

Specifikace CSS import modulů nenabízí, tudíž bylo třeba použít nástroje WebPack a PostCSS, které node package transpilují do souborů použitelných .css souborů pro prohlížeč. Od nedávné verze CSS specifikace existuje media query, které rozpozná, zda uživatelovo zařízení je aktuálně v tmavém, nebo světlém režimu, a tuto vlastnost lze povolit i v knihovně Tailwind. Uživatelské rozhraní je tedy dostupné ve dvou modech, tmavém a světlýém Z hlediska vývoje byl tento proces lehčí, než se zdálo. Tailwind disponuje modifikátory, které aplikuji styly v závislosti na podmínce, jako je například podmínka dark pro tmavý režim. Každý element má tedy třídu pro světlý a tmavý režim.

Občas základní styly knihovny nevyhovují a i když je zcela validní možností napsat si styly vlastní do vlastního CSS souboru, Tailwind pro přizpůsobení preferuje psát pravidla do JavaScriptového souboru s nastavením. Zde je pak možné používat více funkcí pro vytvoření stylu, například javascriptové výpočty, a také lze vlastní styly navázat na předpřipravené modifikátory Tailwind, jako jsou hover, focus, a další. Další výhodou je úprava již existujících tříd, ať už je to změna pravidla, nebo doplnění nových. Styly, které jsou takto zapsány, jsou při generování CSS souboru brány v potaz a doplněny do souborů.

6.2.2 Integrace knihovny Vue.js

Tvořit webové stránky s dynamickým obsahem je za použití čistého JavaScriptu v dnešní době nepraktické, naštěstí existuje spousta knihoven usnadňující práci. Vue.js je jednou z nich [15], a lze ji importovat do jakékoliv stránky pouhým přidáním souboru se skriptem.

Na začátku je potřeba vytvořit takzvanou Vue instanci. v HTML musí existovat unikátní element, například div, s id tagem, na který se bude odkazovat nová Vue instance. Ta následně vidí a dokáže manipulovat s celou DOM strukturu pod tímto tagem. Na instanci lze uchovávat proměnné ve vlastnosti data, s nimi lze následně pracovat v celé instanci a v HTML. Pomocí Vue atributu je možné data zobrazovat, například atribut v-bind zajistí jednosměrnou komunikaci mezi elementem a daty. Při změně proměnné, která je nabindovaná na element je zajištěno, že se hodnota v HTML instantně změní. Atributem v-model pak lze zajistit obousměrné vázání. Běžné použití je pro input elementy, například textové pole, v průběhu editace pole uživatelem se hodnota proměnné mění jak v instanci, tak na všech místech, kde je použita v HTML. Toho je využito pro změnu názvu aktivity.

Dynamická změna DOM struktury je provedena na základě Vue atributů nastavených na HTML elementech. Vue nepočítá s možností vytvoření vlastního nového elementu v Javascriptu a jeho následném přidáni do DOM stromu. Naopak, pokud má stránka obsahovat elementy zobrazující se dynamicky, na základě podmínek, musí být jejich vzory předem zapsány v HTML a Vue je následně zobrazuje na základě podmínek, nebo existenci dat. Pro podmínečné zobrazení jsou dvě možnosti, `v-if`, která elementy přidá, nebo odstraní, dle podmínky uvedené v hodnotě atributu, v-show má podobný účinek, avšak všechny elementy vždy existují v DOM. Pouze je jim nastaven CSS styl display pro skrytí, nebo zobrazení. Pokud má být přidán element na základě akcí při běhu aplikace, je třeba ho buď připravit předem v HTML a poté podmínečné skrýt na základě existence dat, nebo použít atribut `v-for` pro zobrazení kolekcí, jako je tomu na ukázce 6.6. Ten pracuje na principu cyklu foreach. Všechny elementy z kolekce postupně vezme a dle vzoru vytvoří, nebo smaže, elementy na určeném místě. Renderovací atributy lze kombinovat, například podmínečně zobrazovat prvky kolekce uvnitř cyklu.

```
<div class="...">
    <music-info
```

```
        v-for="(track, index) in selectedActivityMusic"
        :key="index"
        :title="track.name">
    </music-info>
</div>
```

Výpis 6.6: Zobrazení dat ve sloupci

Pro zpřehlednění kódu a logické rozčlenění znovupoužitelných částí je možné vytvářet komponenty. Ty jsou buď součástí původní instance, a pak sdílí veškeré datové proměnné a metody zaregistrované na instanci, nebo jsou vytvořeny mimo, do vlastního JavaScriptového souboru. Pak má každá komponenta svoje datové úložiště a funkce. Pokud je vytvořena v JavaScriptovém souboru, pak je třeba jí pojmenovat a vytvořit vzorový HTML kód, který pak zobrazuje. Komponenta však nemůže stát samostatně a je třeba ji referencovat v instanci, po registraci ji pak lze použít v HTML kódu stejně, jako tradiční elementy. Lze předávat komponentám data, takzvané props, jejich názvy nejprve třeba registrovat jako hodnoty pole v proměnné props v komponentě. Každá prop pak může mít i svůj datový typ, Vue se pak za běhu postará o kontrolu typu a v případě neshody vyvolá chybu. Pro jednoduchost uvádím v ukázce 6.7 pouze názvy. Data komponentám avšak lze předávat pouze jednosměrně, směrem dolů z rodičovské instance, do komponenty. Uvnitř je možné s daty pracovat stejně, jako v původní instanci, ale změny se nepromítnou zpět nahoru. Proměnné pro props, které chceme poslat do komponenty, je třeba zapsat do atributu v-bind, nebo v jeho zkrácené verzi stačí použít znak : před názvem vlastnosti. Bind atributy očekávají název proměnné, ze které data vezmou, v případě nutnosti předat komponentě konstantní hodnotu se atribut neuvozuje klíčovým slovem v-bind a pouze se zapíše samotná hodnota. Na ukázce 6.6 je demonstrováno použití dynamického zobrazení kolekce pomocí v-for, každá položka je v lokální proměnné track a její pořadí v index. Ten je použit jako klíč elementu a z dat položek je hodnota jména poslána komponentě v prop title.

```
Vue.component('music-info', {
    props: [ 'title', ... ],
    methods: {
        ...
        // vlastni metody komponenty nezavisle na rodici
    },
    data: {
        ...
        // vlastni data komponenty nezavisle na rodici
    },
    template: `...` // vzorovy HTML kod pro vykresleni
})
```

Výpis 6.7: Komponenta ve Vue.js

Každému elementu lze přidělit události. Atributem v-on, následovaným názvem události, na kterou má reagovat, a metodou, jenž se zavolá po aktivaci události, lze takto jednoduše vytvořit interakce s prvky na stránce. Metody jsou opět na Vue instanci ve vlastnosti `methods`, lze jim dát libovolný počet atributu a zavolat je odkudkoliv uvnitř instance. Obdobně pak existují `computed` funkce, připomínající spíše get vlastnosti s objektových

jazyků. Jsou používány jako proměnné pro čtení, avšak při použití jsou volány jako funkce vracející libovolnou hodnotu, ať už je to například upravená hodnota z dat, nebo výsledek voláni některé z metod.

Lze pracovat i s událostmi Vue instance jako takové. Je možné deklarovat funkce, které se vyvolají před, nebo po klíčových událostech životního cyklu Vue.js, například před vytvořením nové instance, po kompletním načtení a vykreslení všech HTML elementů, nebo po každé aktualizaci. Pro realizaci detekce, zda je stránka otevřena na mobilním prohlížeči, byla použita událost `mounted`. Ta se vyvolá poté, co byla vytvořena instance, nasadila se na specifikovaný element a je připravena naslouchat změnám a vykonávat operace. Kompletní seznam události je dostupný v dokumentaci Vue.js[9].

6.2.3 Data z knihovny CloudKitJS

Apple nabízí vlastní knihovnu pro realizaci propojení aplikace s úložištěm iCloud [5]. Na stránkách pro správu CloudKit databázi je třeba nejprve vytvořit API klíč, pomocí kterého se bude z webové aplikace přistupovat do databáze, následně se nastaví název databáze a produkční, nebo vývojářský mód. Po dokončení inicializace je vyvolán proces autentizace, knihovna na předem určená místa v HTML kódu dosadí tlačítka pro přihlášení, nebo odhlášení. Na základě události přihlášení a úspěšné autentizace je možné přistupovat ke kontejnerům s daty.

Stejně jako v mobilní aplikaci toto probíhá pomocí zasílání dotazů. Všechny operace jsou asynchronní a po jejich dokončení je vrácena buď kolekce s daty nebo chybové hlášení. Nejprve se vytvoří query objekt, jehož atributy tvoři popis toho, která data a jak se mají získat. Ukázka 6.8 je z části převzata z dokumentace Apple[10] a ukazuje příklad tvorby a odeslání query.

Query objekt může mít tři atributy, `recordType`, `filterBy` a `sortBy`. Povinný je pouze `recordType`, což je název entity, kterou chceme získat. Ve výchozím stavu jsou entity, které byly vytvořeny z CoreData v mobilní aplikaci uvozeny přeponou CD, následované samotným názvem. Každá entita, ke které chceme takto přistupovat, musí být indexovatelná, což je třeba nastavit manuálně ve správci kontejnerů CloudKit. Požadavek selže, pokud požadovaný `recordType` neexistuje. Další dva atributy jsou nepovinné a používají se pro specifikaci požadavku na entitu. `filterBy` je pole predikátů. Tvorba predikátu se podstatně liší, od verze ve Swiftu. V JavaScriptu má predikátový objekt tři atributy. Atribut `comparator` určuje, jaká operace se má provést, následně `fieldName` specifikuje atribut entity, který se porovnává. Ve Swift predicate to je levá strana výrazu. `fieldValue` pak představuje pravou stranu tohoto výrazu neboli hodnotu, se kterou se levá strana operace porovná. `sortBy` je stejně jako ve Swift FetchRequestu pole. Jeho položky jsou objekty obsahující název atributu, dle kterého se výsledky řadí, a zda jsou podle něj sestupně, nebo vzestupně. Opět záleží na pořadí položek a je řazeno od první.

Následně se vybere databáze z kontejneru, buď privátní, nebo veřejná a na ní se vykoná query. Protože je metod asynchronní, vrací typ Promise. Knihovna nebyla prozatím dostatečně modernizována, aby tyto funkce podporovaly novější způsob práce s asynchronním voláním v JavaScriptu, takzvaný `async/await`, tento přístup by kód zpřehlednil vynecháním callback funkcí. Je tedy nutné na `performQuery` následně navázat funkcí `.then()`, ve

[9] https://vuejs.org/v2/api/#Options-Lifecycle-Hooks

[10] https://developer.apple.com/documentation/cloudkitjs/cloudkit/database/1628596-performquery

které bude výsledek v případě úspěchu, a funkci `.catch()` pro výsledek v případě selhání. S úspěšným výsledkem pak lze pracovat, v ukázkovém případě je uložen do dat Vue instance.

```
const query = {
    recordType: 'Artwork',
    filterBy: [{
        comparator: 'EQUALS',
        fieldName: 'address',
        fieldValue: { value: 'Fort Bragg, CA' }
    }]
}

var db = this.container.privateCloudDatabase;
db.performQuery(activityQuery).then((response) => {
    // ulozeni ziskanych dat
}).catch((error) => {
    // zpracovani chyb
});
```

Výpis 6.8: Příklad query pro CloudKitJS

`performQuery` lze ještě upřesnit volitelným parametrem Options. Zde se nachází například možnost limitovat počet výsledků dotazu pomocí vlastnosti `resultsLimit`, Obdobně, jako ve Swiftu, je možnost dotázat se pouze na některé atributy entity, kolekce `desiredKeys` může obsahovat názvy atributů, které chce, opět to ale musí být atributy náležící požadované entitě, pokud je pole prázdné, jsou vráceny všechny. Zajímavostí je binární vlastnost `numbersAsStrings`. Ta zajistí, že všechny číselné hodnoty ve výsledku dotazu budou převedeny na řetězce.

Datové položky jsou pak uloženy do Vue instance pro zobrazení, a v metodách Vue instancí je realizována jejich modifikace. Abych zbytečně nezatěžoval databázi, jsou modifikace odesílány pouze po vyvoláni uložení, ne při každé změně provázaných dat. Při požadavku na změnu je třeba správně zadat datovou zónu. Pro jednoduchost v aplikaci existuje pouze jedna, avšak je možné data v jedné databázi třídit do více zón a ty pak nezávisle na sobě měnit. Datové objekty z Cloudu na sobě obsahují tag se změnou, ten je třeba odeslat při každém požadavku na změnu, aby správně proběhla synchronizace. Při změně API vrátí nový tag, kterým je třeba se následně řídit.

Uložení změn, nebo vytvoření nového záznamu, probíhá metodou `saveRecords`. Stejně, jako u dotazu, je třeba nejdříve vytvořit objekt specifikující co chceme udělat. Protože každý záznam v CloudKit databází má vlastní unikátní ID UUID, je třeba ho při požadavku na změnu hodnot specifikovat ve vlastnosti `recordName`, v případě, že vytváříme nová data, není třeba ho specifikovat, pak bude vytvořeno nové UUID pro tento záznam. Následně, pole `recordType`, je povinné, pokud má požadavek vytvořit nový záznam v tabulce, zde se pak určí, v jaké tabulce to má být. Při editaci existujících dat je nutné uvést `recordChangeTag`. V poslední řadě vlastnost `fields`, do které se vloží objekty reprezentující jednotlivé atributy entity. Při změně stačí pouze atribut, který chceme změnit, když vytváříme nový, tak je třeba zadat všechny povinné atributy.

Pro smazání dat lze použít metodu `deleteRecords`. Je třeba specifikovat záznam, který chceme smazat. Narozdíl od save, zde ale stačí pouze unikátní ID záznamu v parametru

volání metody. Alternativně je možné vytvořit Record objekt, avšak zde se nemusí specifikovat vlastnosti, jako název tabulky, nebo hodnoty. Smaže se vždy celý záznam pod zadaným ID. Lze také smazat více záznamů naráz. Oba přístupy, buď přes Record, nebo řetězec, podporují zadání kolekce s více záznamy.

Alternativou pro `performQuery` je metoda `fetchRecords`. Její použití se mírně liší a podobá se více použití `saveRecords`, nebo `deleteRecords`. Parametrem metody může být objekt Record, ve kterém je nutné zadat pouze `recordName`, také je možnost zadat ID záznamu jako řetězec. Rozdílem oproti `performQuery`, je že tato metoda je vhodná na získání záznamu, jehož ID už je známo. Nelze získat celou tabulku záznamů, ani specifikovat parametry, podle kterých se záznamy získají. Metoda je ekvivalentní tomu, kdybychom v SQL chtěli získat záznam, podle jeho primárního klíče, nebo ve `performQuery` vytvořili filter objekt, limitující výsledky pouze shodné ID.

Webová aplikace umožnuje pouze zobrazení, nebo změnu dat, vytvoření vlastních dat, například vyplněním souřadnic a informací o aktivitě a hudbě, nebylo pro jednoduchost a konzistenci dat zamýšleno.

6.2.4 Nástroje pro zajištění kompatibility prohlížečů

Dnes existuje mnoho prohlížečů, a i když je situace lepší, než byla dříve, ne všechny mezi sebou sdílí stejné vlastnosti a stejnou úroveň kompatibility JavaScriptových a CSS funkcí. Existují nástroje na zajištění správné funkčnosti napříč většinou moderních prohlížečů a některé z nich jsou zde využity.

První z nich, Babel[11] , zajištuje kompatibilitu mezi předem určenými prohlížeči. Při použití rozšíření `@babel/preset-env` lze určit cílové verze a typy prohlížečů, ve kterých má být kód funkční. Buď se nastaví přesné názvy a verze, nebo se použije dynamické nastavení dle popularity. Využil jsem možnosti kompatibility cílené na 99% aktuálně populárních prohlížečů. 100% kompatibilitu nikdy zaručit nelze, vždy se najde uživatel zastaralého webového prohlížeče, který podporuje příliš málo funkcí. Zdrojem dat o užívání prohlížečů je pak další balíček `browserlist`, v něm lze nastavit i vlastní data odlišná od globální databáze a jimi se pak další balíčky řídí. To je však nad rámec aplikace. Samotný proces poté probíhá tak, že se analyzuje zdrojový JavaScriptový kód a pokud obsahuje funkce a vlastnosti nedostupné na některých prohlížečích, jsou implementace těchto věcí doplněny do výsledného kódu, takzvaný polyfill. Výsledkem je sice o něco větší zdrojový soubor, ale je zaručena požadovaná kompatibilita.

Až na CloudKitJS, jsou všechny knihovny staženy pomocí správce balíčků npm a ačkoliv spousta vývojářů knihoven nabízí verze pouze s přímým odkazem na skript, nebo pro stažení lokálně, jsou i takové, které je třeba importovat v kódu a ten poté transpilovat do souboru použitelného ve webovém prohlížecí. Proto existuje nástroj WebPack. V kódu lze používat možnosti, jako je import tříd a knihoven, které standardní JavaScript v prohlížecí nepodporuje. Stejně, jako Babel, WebPack pak tyto knihovny vloží do jednoho JavaScript souboru a zajistí, aby všechna volání tříd a funkcí jiných knihoven fungovala správně. WebPack také referencuje další nástroje používané v aplikaci, stačí je zahrnout v JavaScriptovém objektu pro nastavení a on se pak postará o to, aby byly spuštěny ve správném pořadí. Z těchto důvodů je třeba po každé změně spustit příkaz pro transpilaci všech knihoven do příslušných souborů.

Knihovna Tailwind je také nainstalována jako npm balíček a ačkoliv to je knihovna kaskádových stylů, a ne JavaScriptu. Pro to je třeba použít i preprocesor PostCSS. Tailwind

[11] https://babeljs.io/

obsahuje nové direktivy, které základní CSS nepodporuje, například direktivu `@tailwind` pro import stylů do souboru. PostCSS dokáže tyto nové direktivy spravovat tak, aby bylo výsledné CSS možno použít v prohlížečích. PostCSS má svůj vlastní ekosystém pluginů rozšiřující jeho funkčnost, v základním stavu ho lze použít pouze jako preprocesor, doplňující CSS o pár nových funkcí, jedním takovým pluginem je Autoprefixer. Různé prohlížeče od různých vývojářů často implementují vlastní CSS pravidla, která ještě nejsou standardem CSS. Tato pravidla mají pak vlastní prefixy na základě renderovacího jádra prohlížeče. Taková pravidla lze kombinovat s ostatními a každý prohlížeč si vybere to své, které podporuje. Autoprefixer se stará o doplnění těchto pravidel do CSS kódu. Například, programátorovi může použít ve zdrojovém kódu na ukázce 6.9 pouze pravidlo `user-select`, a při zpracování kódu v PostCSS s pluginem Autoprefixer, je výsledný kód doplněn o další ekvivalentní pravidla.

```
.example {
    display: -ms-grid;
    display: grid;
    -webkit-user-select: none;
       -moz-user-select: none;
        -ms-user-select: none;
            user-select: none;
}
```

Výpis 6.9: Ukázkový výstup Autoprefixer

Kapitola 7

Sport a hudba v praxi

Cílem práce nebylo pouze vytvořit funkční aplikaci, ale zároveň prozkoumat vliv hudby na sport a tyto informace předat uživateli. V této kapitole tedy proberu, jak jsem informace o hudbě integroval do rozhraní, a v následující se zaměřím na praktické testování a jeho výsledky.

7.1 Zisk informací

V předchozích kapitolách jsem se zabýval technickým aspektem zisku informací, jak jsou staženy a jak jsou uloženy. Rychlost uživatele je nejdůležitější informace, kterou k analýze výsledku potřebuje. Měl jsem zde v podstatě dvě možnosti, buď rychlost přepočítat z páru souřadnic, nebo si ji vzít přímo ze zařízení. Protože zisk GPS dat probíhá v řadu stovek milisekund, ruční výpočet by znamenal zjištění vzdálenosti mezi souřadnicemi v určitých jednotkách, následně stanovení rozdílu mezi časovými značkami a převod do standardizované jednotky rychlosti, v tomto případě metry za sekundu. Druhým přístupem je získat informaci přímo z akcelerometru zařízeni. Rychlost je měřena v metrech za sekundu a je relativní k aktuálnímu směru. Tato informace také není úplně přesná, tudíž existuje další proměnná, obsahující přesnost rychlosti, se kterou je třeba počítat.

7.2 Integrace v uživatelském rozhraní

Rychlosti jsou postupně průměrovány. Každá hudební položka obsahuje vlastní údaje o uživatelově rychlosti, také aktivita jako celek má svůj atribut s rychlosti. U náhledu každé aktivity je tedy zobrazena její celková rychlost. V tomto smyslu to znamená průměrnou rychlost napříč celou aktivitou. Dále zde nalezne začátek aktivity, absolvovanou délku a čas, za který ji absolvoval. Aby bylo možné své výsledky porovnat, nabízí se na prohlížecím view možnost výsledky seřazovat podle rychlosti a délky a filtrovat podle data. Jednoduše tak zde uvidí, jak si vedl v jaký den, nebo se může rychle podívat na své nejlepší a nejhorší výkony. V detailním zobrazení aktivity nalezne pak více informací a konkretní hudbu. Ta je seřazena podle data, kdy byla puštěna a opět se nabízí možnost ji seřadit dle rychlosti uživatele. Hudba, u které byl nejrychlejší, je zapsána zvlášť. Tudíž, stejně jako aktivity, může porovnávat hudbu a zjistit si, u které byl neproduktivnější.

7.3 Experiment

Práce je založena na hypotéze, že sport a hudba jsou spolu provázané a mají na sebe pozitivní efekt, jak bylo nastíněno v kapitole 2. Cílem bylo tedy mimo jiné pokusit se ověřit, že toto platí. Protože ale byla práce vytvořena v nelehké době pandemie a restrikcí, byla moje možnost experimentování s větším počtem subjektů značně omezena. I přes to jsem se rozhodl udělat experiment alespoň s pár lidmi, kterým byla aplikace předána na mobilní zařízení, byla vytvořena trajektorie, které se měli držet a zvoleny hudební stopy.

Stejně, jako v článku [17], ze kterého vychází předpoklady uvedené na začátku práce, byla zvolena hudba s různým tempem. Protože aplikace je založena na propojení se Spotify, využil jsem playlistů vytvořených editory platformy, pro hudbu s vyšším tempem jsem zvolil playlist orientovaný na běh, obsahující rychlejší hudbu a pro nižší tempo oddechový playlist. Dva uživatelé absolvovali předurčenou trať dvakrát, pokaždé s jiným playlistem.

Protože byl můj dataset značné limitovaný a nemohl jsem vyzkoušet tolik kombinací, jako v publikacích zabývající se přímo výzkumem a experimentováním vztahu hudby a sportu, i tak jsem dosáhl příznivých výsledků. Průměrná rychlost při poslechu rychlejší hudby byla 4.0 metrů za sekundu, rychlost u pomalejší hudby byla o něco málo nižší – 3.7 metrů za sekundu. Procentuálně bylo v tomto testu zaznamenáno 8%, což je o něco menši, než práce, ze které vycházela moje hypotéza. Věřím, že kdybych měl k dispozici více testovacích dat, že by výsledky byly přesnější. Je navíc možné, že rozdíly mohly být způsobeny jinými faktory, se kterými se v testu nepočítalo, ale i tak jsem s výsledky spokojen.

Kapitola 8

Testování

Téměr výhradně jsem aplikaci testoval za použití vývojového prostředí a simulátorů pro platformu iOS. S prostředím XCode mají vývojáři přistup k plnohodnotnému simulátoru jakéhokoliv zařízení iOS a možnost ladit, trasovat kód a číst konzolové výpisy. Dalším důležitým důvodem je jednoduchost a rychlost provedení změn v aplikaci. Po změně kódu stačí novou verzi sestavit a ta je nahrána na cílové simulované zařízení. Je možné připojit k počítači i vlastní iOS zařízení a aplikaci ladit na něm, avšak to se pro mě ukázalo ne zcela výhodné. Simulátory běží na stejném PC, jako vývojové prostředí, a komunikace je proto rychlejší, než se zařízením připojeným přes USB. Dalšími důvody byla možnost spustit si více simulátorů naráz a testovat tak stejnou aplikaci na různých zařízeních. V mém případě toto bylo důležité pro otestování cloudové synchronizace a v poslední řadě lze v simulátoru simulovat lokační data. Protože je moje aplikace zaměřena na sledováni pohybu a sběr dat, bylo by zcela nepraktické se po každé změně a nutnosti otestovat funkčnost aplikace vypravit ven nasbírat nová data. V simulátoru lze nastavit trasu, která se nasimuluje, aniž by se musela změnit fyzicky lokace zařízení.

Testováni funkcionality aplikace bylo tedy prováděno mnou, formou manuálního testování kódu. Po každé důležité změně jsem provedl operace, které by vykonal uživatel a ověřil, že vše funguje jak má. Protože se jedná o aplikaci s důrazem na uživatelské vstupy, přišla mi tato forma testování, oproti vytvoření automatizovaných testů, výhodnější.

Návrh a funkčnost uživatelského rozhraní jsem sám otestovat nemohl, zvolil jsem proto několik přátel s různou úrovní znalosti systému iOS a mobilních aplikací jako takových. Předložil jsem jim svoji aplikaci s jednoduchým cílem, zkusit se v ní zorientovat a zvládnout základní úkony, na které byla navržena. Výsledek byl v celku uspokojivý. Ukázalo se, že některé částí rozhraní nebyly v původní verzi příliš intuitivní a lehce rozpoznatelné, primárně pak v listech aktivit nešlo lehce rozeznat, s kterou část UI lze ovládat. Zde jsem provedl lehkou revizi návrhu, všechna tlačítka a prvky, na které může uživatel ťuknout jsou buď ohraničena okrajem a vyniknou tak z pozadí, nebo prostého textu, nebo jim byla přidána šipka signalizující existenci více dat po rozevření. Druhým problémem byl první koncept oznamovaní zpráv uživateli. Ten spočíval v oznámení pomocí vyskakovacího okna, Alert view, avšak prakticky byl příliš rušivý. Rozhodl jsem se ho proto vyměnit za knihovnu, díky které se zobrazí nerušivá, ale stále viditelná zpráva na straně obrazovky.

Kapitola 9

Závěr

V práci jsem se zabýval vztahem mezi hudbou a aktivitou a zjišťoval, jaké efekty má různá hudba na uživatele. Proto byla vytvořena mobilní aplikace, umožnující sledování aktivity a propojení s hudebním přehrávačem. Dále jsem navrhl jednoduchou webovou aplikaci, umožnující sledování dat z jiných zařízení. Při vývoji jsem si rozšířil obzory tvorby mobilních aplikací a seznámil se s vývojem iOS aplikací pomoci nového frameworku SwiftUI, následně vytvoření datového úložiště CloudKit a propojení s více zařízeními a synchronizací dat mezi mobilní a webovou aplikací.

Analyzoval jsem aktuální trendy vývoje aplikací a zvolil takové vývojové prostředky, které by nejlépe vyhovovaly pro vývoj cílového produktu. Následně bylo navrženo uživatelské rozhraní, u kterého byl kladen důraz na znovupoužitelnost, jednoduchost ovládaní a podobnost s nativními aplikacemi platformy. Se stejným předpokladem jsem navrhl a implementoval i webovou aplikaci, která je lehčí na funkcionalitě, ale dodá uživateli možnost sledování výsledku na jakékoliv platformě s přístupem k internetu. Za tímto účelem bylo též nutné prozkoumat možnosti synchronizace dat a uložení na cloudovém úložišti a byl zvolen takový systém, který jednoduše zapadne do zbytku aplikace. Výsledná aplikace byla testována prakticky na reálných datech, na nich byly provedeny experimenty a vyvozeny závěry odpovídající teoretickým předpokladům, na kterých je práce postavena.

Výslednou aplikaci je možné vydat na platformu AppStore a publikovat ji tak většímu spektru uživatelů, kdykoliv je také možné aplikaci rozšířit a obohatit o další vlastnosti. Protože byl pro mě vývoj aplikací ve SwiftUI a připojení cloudových úložišt novinkou, musel jsem řešit unikátní problémy. Doufám, že jak moje výsledná aplikace, tak práce jako taková, nabídne budoucím čtenářům užitečné informace.

Literatura

[1] Agafonkin, V. *Leaflet Quick Start Guide* [online]. 2021 [cit. 2021-04-30]. Dostupné z: https://leafletjs.com/examples/quick-start/.

[2] Andersen, A. *History of the browser user-agent string* [online]. Září 2009 [cit. 2021-04-30]. Dostupné z: https://webaim.org/blog/user-agent-string-history/.

[3] Apple. *Apple unveils groundbreaking new technologies for app development* [online]. červen 2019 [cit. 2021-04-30]. Dostupné z: https://www.apple.com/newsroom/2019/06/apple-unveils-groundbreaking-new-technologies-for-app-development/.

[4] Apple. *Framework CloudKit* [online]. 2021 [cit. 2021-04-30]. Dostupné z: https://developer.apple.com/documentation/cloudkit.

[5] Apple. *Framework CloudKit JS* [online]. 2021 [cit. 2021-04-30]. Dostupné z: https://developer.apple.com/documentation/cloudkitjs/.

[6] Apple. *Managing Your App's Life Cycle* [online]. 2021 [cit. 2021-04-30]. Dostupné z: https://developer.apple.com/documentation/uikit/app_and_environment/managing_your_app_s_life_cycle.

[7] Bush, A. *Dart Summit 2015* [online]. Květen 2015 [cit. 2021-04-30]. Dostupné z: https://www.sm-cloud.com/dart-summit-2015/.

[8] Cherry, K. *How Listening to Music Can Have Psychological Benefits* [online]. Prosinec 2019 [cit. 2021-04-30]. Dostupné z: https://www.verywellmind.com/surprising-psychological-benefits-of-music-4126866.

[9] Jacobs, B. *Core Data Relationships and Delete Rules* [online]. 2021 [cit. 2021-04-30]. Dostupné z: https://cocoacasts.com/core-data-relationships-and-delete-rules/.

[10] Kar, S. *Apple's CloudKit is the MBaaS Answer to AWS and Azure Cloud Services* [online]. červen 2014 [cit. 2021-04-30]. Dostupné z: http://cloudtimes.org/2014/06/05/apples-cloudkit-is-the-mbaas-answer-to-aws-and-azure-cloud-services/.

[11] Margutti, D. *SwiftLocation Readme* [online]. únor 2021 [cit. 2021-04-30]. Dostupné z: https://github.com/malcommac/SwiftLocation/blob/master/README.md.

[12] Mattt. *NSPredicate* [online]. červenec 2013 [cit. 2021-04-30]. Dostupné z: https://nshipster.com/nspredicate/.

[13] Occhino, T. *React Native: Bringing modern web techniques to mobile* [online]. Březen 2015 [cit. 2021-04-30]. Dostupné z: https://engineering.fb.com/2015/03/26/android/react-native-bringing-modern-web-techniques-to-mobile/.

[14] Spotify. *Authorization Guide* [online]. 2021 [cit. 2021-04-30]. Dostupné z: https://developer.spotify.com/documentation/general/guides/authorization-guide/.

[15] Team, V. *What is Vue.js?* [onine]. říjen 2020 [cit. 2021-04-30]. Dostupné z: https://vuejs.org/v2/guide/index.html.

[16] Thoma, M. V., La Marca, R., Brönnimann, R., Finkel, L., Ehlert, U. et al. The Effect of Music on the Human Stress Response. *PLOS ONE*. 1. vyd. Public Library of Science. Srpen 2013, sv. 8, č. 1. DOI: 10.1371/journal.pone.0070156. Dostupné z: https://doi.org/10.1371/journal.pone.0070156.

[17] Waterhouse, J., Hudson, P. a Edwards, B. Effects of music tempo upon submaximal cycling performance. *Scandinavian Journal of Medicine & Science in Sports*. 1. vyd. Červenec 2010, sv. 20, č. 4, s. 662–669. DOI: https://doi.org/10.1111/j.1600-0838.2009.00948.x. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1600-0838.2009.00948.x.

[18] Welch, C. *Apple HealthKit announced: a hub for all your iOS fitness tracking needs* [online]. červen 2014 [cit. 2021-04-30]. Dostupné z: https://www.theverge.com/2014/6/2/5772074/apple-healthkit-ios-8-announcement.

[19] Yamamoto, M., Naga, S. a Shimizu, J. Positive musical effects on two types of negative stressful conditions. *Psychology of Music*. 1. vyd. Duben 2007, sv. 35, č. 2, s. 249–275. DOI: 10.1177/0305735607070375. Dostupné z: https://doi.org/10.1177/0305735607070375.