Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Rosanda Potrebuješ

Mobilna aplikacija za podajanje konteksta uporabnika na prireditvah

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Veljko Pejović

Ljubljana, 2019

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V okviru diplomske naloge oblikujte idejno rešitev, implementirajte, ter testirajte mobilno aplikacijo za zaznavanje konteksta uporabnika na prireditvah. Aplikacija naj zaznava okvirno število ljudi v bližini uporabnika, uporabnikovo fizično aktivnost, glasbo, ki jo uporabnik posluša, ter ostale lastnosti konteksta, po potrebi. Aplikacija naj omogoči uporabnikom, da delijo svoje izkušnje s pomočjo objav. Implementacija naj bo narejena za Android okolje, s strežniško podporo za računsko bolj zahtevne storitve (kot so zaznavanje glasbe) in izmenjavo objav. Testiranje naj bo izvedeno z dejanskimi uporabniki.

Zahvaljujem se svojim bližnjim za vso podporo in vzpodbudo čez vsa leta študija, mentorju za njegove predloge, svetovanja in pomoč pri izdelavi diplomske naloge.

Kazalo

Povzetek

Abstract

1 Uvod in motivacija **1**

2 Arhitektura predlaganega sistema in primer uporabe **5**

3 Uporabniški vmesnik mobilne aplikacije **9**

3.1 Razporeditev aplikacije 9

3.2 Vpis in registracija 12

3.3 Domača stran 13

3.4 Navigacijski menu 13

3.5 Moj profil 13

3.6 Moji prijatelji 13

3.7 Nastavitve 14

3.8 O aplikaciji 14

4 Povezovanje podatkov v mobilni aplikaciji **15**

4.1 Arhitektura MVVM 17

4.2 Implementacija MVVM 18

4.3 Implementacija povezovanja podatkov s knjižnjico Data Binding 19

5 Spletni strežnik **23**

5.1 Komunikacija aplikacije s spletnim strežnikom 26
5.2 PHP podatkovni objekti 29

6 Baza podatkov **31**
6.1 Tabela user 32
6.2 Tabela login 32
6.3 Tabela friend 33
6.4 Tabela post 34

7 Shranjevanje podatkov v aplikaciji **35**
7.1 Objekti 36
7.2 Knjižnjica Room Persistence 38

8 Zaznavanje **43**
8.1 Razred UserActivity 45
8.2 Vmesnik MasterDetector 45
8.3 Zaznavanje števila ljudi 47
8.4 Zaznavanje lokacije 48
8.5 Zaznavanje gibanja 49
8.6 Zaznavanje glasbe 50

9 Testiranje **55**

10 Omejitve **59**
10.1 Velikost baze 60
10.2 Interakcija uporabnika z aplikacijo 60
10.3 Natančnost aplikacije 60
10.4 Hitrost aplikacije 61
10.5 Poraba energije baterije 62

11 Sklepne ugotovitve **63**

Literatura **66**

Slike

2.1 Glavne komponente aplikacije 5

3.1 Razporeditev aplikacije 9
3.2 Videz posameznih strani aplikacije 10
3.3 Videz navigacijskega menuja 11

4.1 Arhitekturni vzorec MVC 15
4.2 Arhitekturni vzorec MVP 16
4.3 Arhitekturni vzorec MVVM 16
4.4 Implementacija MVVM 18

5.1 Delovanje spletnega strežnika 23

6.1 Tabele v bazi 32
6.2 Tabela friends 33

9.1 Rezultati ankete 56

Seznam uporabljenih kratic

kratica	angleško	slovensko
MVVM	Model-View-ViewModel	arhitekturni pristop model-pogled-modelPogled
MVC	Model-View-Controller	arhitekturni pristop model-pogled-nadzor
MVP	Model-View-Presenter	arhitekturni pristop model-pogled-voditelj
PHP	hypertext preprocessor	skriptni jezik
PDO	PHP data objects	PHP podatkovni objekt
URL	uniform resource locator	enolični krajevnik vira
CSS	Cascading Style Sheet	kaskadne stilske podloge
HTML	Hypertext Markup Language	označevalni jezik za izdelavo spletnih strani
JSON	JavaScript Object Notation	preprost format za izmenjavo podatkov
HTTP	Hypertext Transfer Protocol	komunikacijski protokol za prenos podatkov v spletu
SQL	Structured Query Language	strukturirani povpraševalni jezik
DAO	Data Access Object	objekt z metodami za dostop do podatkov
AAC	Advanced Audio Coding	način kodiranja zvočnih datotek

M4A	Moving Picture Exprert Group 4 Audio	format zvočnih datotek
MP3	Moving Picture Expert Group Layer-3 Audio	format zvočnih datotek
FLAC	Free Lossless Audio Codec	format zvočnih datotek
KB	Kilo Bytes	kilobajt
ORM	Object-relational mapping	Objektno-relacijsko mapiranje

Povzetek

Naslov: Mobilna aplikacija za podajanje konteksta uporabnika na prireditvah

Avtor: Rosanda Potrebuješ

V naši diplomski nalogi razvijamo aplikacijo za sistem Android, ki zaznava kontekst uporabnika na prireditvah. Aplikacija zaznava uporabnikovo fizično aktivnost, lokacijo, število ljudi v okolici in glasbo, ki jo uporabnik posluša. To počne z uporabo več senzorjev, ki so na voljo na pametnih mobilnih telefonih, kot so mikrofon, pospeškometer, Bluetooth in drugi. Dobljeni rezultat prikaže v uporabniku prijazni obliki in mu dovoli, da ga deli z drugimi uporabniki.

Za implementacijo zgornjih funkcionalnosti naša aplikacija uporablja aplikacijski programski vmesnik Google Play za zaznavanje lokacije in aktivnosti uporabnika in Bluetooth za zaznavanje števila ljudi v bližini in Python knjižnjico Dejavu za zaznavanje glasbe. Poleg aplikacije naša rešitev vsebuje še strežnik in podatkovno bazo. Baza shranjuje vse podatke, ki jih naša aplikacija potrebuje, medtem ko strežnik skrbi za komunikacijo med aplikacijo in bazo.

Našo rešitev je testiralo deset uporabnikov, ki so ocenili njeno uporabnost in zmožnost natančnega zaznavanja konteksta. Njihove ocene nam omogočajo pridobitev uporabnih smernic za prihodnje izboljšave na področju mobilnega zaznavanja konteksta v družbenih / zabavnih aplikacijah.

Ključne besede: Mobilno zaznavanje, zaznavanje konteksta, Android.

Abstract

Title: Mobile application for providing user context on events

Author: Rosanda Potrebuješ

In thesis we develop an Android application for context inference at entertainment events. The application detects a user's physical activity and location, a number of people in the vicinity, and music that the user is listening to. The application uses multiple sensors available on the phone, such as microphone, accelerometer, Bluetooth, and others, and presents the inferred information in a user-friendly form. The application then allows the user to share these information with other users of the app.

For the implementation of the above functionalities, our app uses Google Play Application Programming Interface for detecting location and user activity; Bluetooth for detecting a number of people in vicinity; and Dejavu Python library for detecting music. Besides the Android app, our solution encompasses a server and a database. The mobile application is used for user interaction and detection, the database stores all the data that our application needs, while the server handles communication between the application and the database.

We test the developed solution among ten users who evaluated its usability and ability to correctly detect the context. The evaluation enables us to extract actionable guidelines for future improvements in mobile-based context detection for social/entertainment applications.

Keywords: Mobile sensing, Context detection, Android.

Poglavje 1

Uvod in motivacija

Zaradi naraščajoče uporabe socialnih omrežij in tendence uporabnikov, da objavljajo svojo udeležbo na dogodkih in prireditvah, smo se odločili, da izdelamo aplikacijo, ki to omogoča na preprost in hiter način z uporabo senzorjev.

Z aplikacijo, ki sama ugotavlja kontekst uporabnika, lahko na dogodku hitreje in z manj dela objavimo kje smo in kaj počnemo, kot če moramo to sami napisati. Z njo bi manj časa posvečali telefonu in bi se lahko bolj osredotočili na dogajanje okoli nas, medtem ko bi bil končni učinek – našim prijateljem sporočimo kje smo in kaj počnemo.

Glasbeni dogodki so zelo popularni in vse bolj prisotni tudi na socialnih omrežjih. Primeri: Slovenski festival Metaldays ima na socialnem omrežju sto deset tisoč sledilcev. Festival Brutal Assault ima dvainsedemdeset tisoč sledilcev. Wacken Open Air jih ima več kot milijon. Glastonbury jih ima osemsto dvainosemdeset tisoč. Primavera Sound jih ima tristo oseminpetdeset tisoč. Zaradi priljubljenosti in pogostosti obiskovanja glasbenih dogodkov bi bila naša aplikacija privlačna za veliko število ljudi.

Aplikacij, ki zaznavajo gibanje, je veliko. Obstajajo aplikacije, ki posnamejo fotografijo ob zaznavanju gibanja in take, ki delujejo v kombinaciji z raznimi Bluetooth napravami, preko katerih lahko zaznavamo in analiziramo naše gibanje, telovadbo, spanje. Projekt WISDM [19] zaznava gibanje

s pomočjo senzorja za merjenje pospeškov. Zaznavalo se je, ali uporabnik hodi, teče, hodi po stopnicah gor ali dol, sedi ali stoji. SleepBot je aplikacija, ki s pomočjo merilca pospeška in mikrofona spremlja in analizira naš spanec. Z aplikacijo Google Fit lahko spremljamo našo fizično aktivnost. Vse te aplikacije so za zelo specifično uporabo (primer: telovadba). Z našo aplikacijo omogočamo bolj splošno zaznavanje, vsakdanje gibanje uporabnika. S tem se približamo širšemu občinstvu. Hkrati je naše bolj splošno zaznavanje, pri katerem uporabnik ne potrebuje nastaviti ničesar, bolj enostavno za začetnike, ki so jim aplikacije z zaznavanjem gibanja tuje. Naša aplikacija je zato lep uvod v spoznavanje aplikacij z zaznavanjem gibanja.

Naša aplikacija vsebuje tudi prepoznavanje glasbe. Primera aplikacije, ki zaznavata glasbo sta Shazam in Soundhound. Naša aplikacija doda dodatno funkcionalnost zaznavanju glasbe. Če smo na festivalu in poslušamo nam neznane izvajalce, lahko z našo aplikacijo prepoznamo izvajalca, pesem in hkrati delimo objavo. Primer aplikacije, ki uporablja mikrofon za zaznavanje je tudi SoundSense [23]. SoundSense je sistem, ustvarjen za klasifikacijo zvoka na mobilnih telefonih, kjer imamo omejene vire. Je lahek (light-weight) in skalabilen. Klasifikacijo opravlja na mreži (online) z manjšo porabo virov pri računanju. Rezultati so primerljivi s sistemi, ki delujejo offline.

Primer aplikacije, podobne naši, je CenceMe [24], ki zaznava aktivnosti uporabnika, njegovo lokacijo, počutje in mu omogoča, da zaznavanja deli na družbenem omrežju. Je zastarela aplikacija, na voljo za iOS verzijo 1.2.0. EmotionSense [30] je aplikacija, ki zbira podatke o interakciji ljudi v družbi glede na mobilne naprave. Zaznava njihova čustva, bližino in vzorce pogovorov s pomočjo senzorjev pametnih telefonov. Naša aplikacija se razlikuje od njiju po tem, da mi ne zaznavamo počutja in čustev uporabnika in namesto tega zaznavamo glasbo, ki jo uporabnik posluša. Naša aplikacija poskuša dobiti podatke, ki bi opisali njegov obisk glasbene prireditve - kje je, kaj počne, kaj posluša, koliko ljudi je v okolici.

Socialna omrežja imajo na voljo vklop zaznavanja lokacije. Manjka jim večje število različnih zaznavanj, ki jih nudi naša aplikacija. Kot smo omenili

prej, že imamo veliko rešitev za vsako posamezno zaznavanje, ki ga opravlja naša aplikacije. Naša aplikacija združuje ta zaznavanja in jim doda priljubljen socialni del, ki ga ostale aplikacije nimajo. Aplikacija se tako razlikuje od drugih po tem, da smo združili zaznavanje aktivnosti in socialno omrežje. Hkrati je tudi enostavna za uporabo. Uporabnik ne potrebuje nobenega znanja o tem, kako deluje posamezno zaznavanje. Uporabniki tudi ne potrebujejo nobene dodatke opreme, ker vse delo opravljajo njihove mobilne naprave.

Za diplomsko nalogo smo se odločili, da bomo izdelali aplikacijo, ki bo z uporabo raznih senzorjev zaznala kontekst uporabnika, ki je relevanten pri obisku glasbenega dogodka. Aplikacija poda uporabnikovo lokacijo, število ljudi v bližini, gibanje uporabnika in naslov pesmi, ki jo uporabnik posluša. Te informacije predstavi v obliki besedila, ki ga lahko uporabnik nato objavi.

Rešitev smo razvili v Android okolju. Razvojno okolje, ki smo ga pri tem uporabljali, je bil Android Studio. Programski jezik, ki smo ga uporabljali, je Java. Za strežnik smo uporabili jezik PHP. Koda naše Android aplikacije [28] in koda strežnika [29] sta na voljo na spletni strani GitHub.

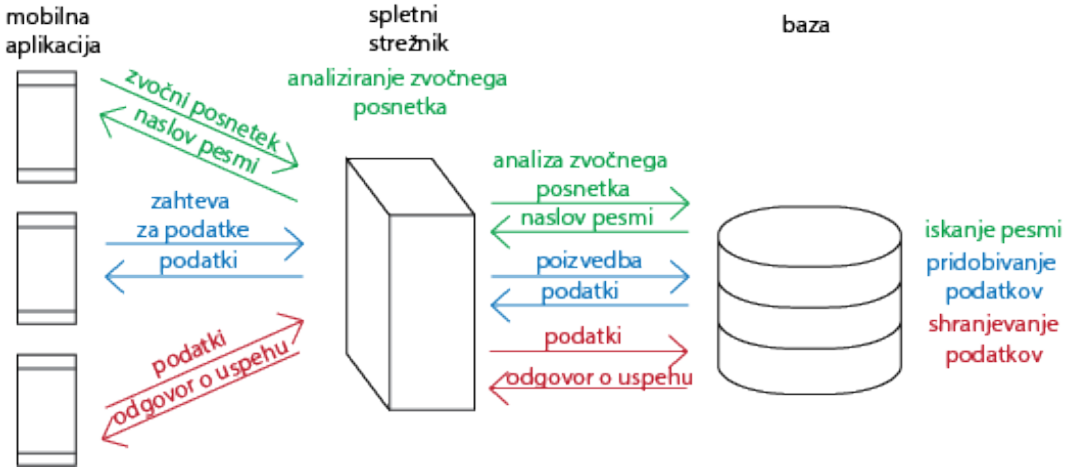
Testiranje je potekalo v lokalnem okolju na manjši skupini ljudi, ki so bili pripravljeni v razmiku nekaj ur preizkusiti aplikacijo. Na koncu so dobili še anketo z nekaj vprašanji o aplikaciji. Testiranje nam je omogočilo pregled naše aplikacije. Dobili smo prvi odziv ciljne publike in dobili informacije o funkcionalnostih, ki si jih še želi v aplikaciji. Pridobljene podatke smo primerjali tudi s tistimi, ki smo jih dobili, ko smo sami testirali aplikacijo. Najbolj natančno je bilo zaznavanje števila ljudi, ki je bilo v vseh primerih pravilno. Sledi zaznavanje lokacije, ki je imelo 96 % natančnost. Najslabše se je odrezalo naše zaznavanje glasbe, ki je imelo 78 % uspešnost. Aktivnost je bila pravilno zaznana v 90 % primerih.

V naslednjem poglavju (poglavje 2) bomo opisali arhitekturo predlaganega sistema. Opisali bomo glavne komponente naše aplikacije in opisali njihove naloge. V poglavju 3 bomo opisali uporabniški vmesnik, povedali, kako je razporejena aplikacija, in opisali posamezne strani. V poglavju 4

bomo opisali arhitekturni vzorec naše aplikacije, njegovo implementacijo in povezovanje podatkov v naši aplikaciji. Spletni strežnik in komunikacijo z njim opišemo v petem poglavju. V poglavju 6 bomo opisali našo bazo podatkov in nato bomo v poglavju 7 opisali, kako shranjujemo podatke v aplikaciji. Podrobnejši opis je v osmem poglavju, kjer bomo podrobneje razložili posamezno zaznavanje. V poglavju 9 bomo predstavili rezultate našega testiranja. V poglavju 10 bomo opisali omejitve naše aplikacije. V poglavju 11 bomo opisali naše sklepne ugotovitve.

Poglavje 2

Arhitektura predlaganega sistema in primer uporabe



Slika 2.1: Glavne komponente aplikacije

V tem poglavju bomo razložili posamezne dele aplikacije in primer uporabe. Glavne komponente diplomske naloge so mobilna aplikacija, spletni strežnik in baza. Mobilna aplikacija pošlje vse zahteve za podatke strežniku. Strežnik se poveže na bazo, izvede poizvedbo in vrne rezultat aplikaciji. Baza hrani vse podatke, ki jih aplikacija potrebuje, na primer podatke za avtentikacijo uporabnika in njihove objave. Aplikacija ima prvi del, ki skrbi za

registracijo in vpis uporabnika. Drugi (glavni) del aplikacije skrbi za zaznavanje in interakcijo z uporabnikom.

Glavni del aplikacije nudi pregled naših podatkov in objav, dodajanje drugih uporabnikov kot naših prijatelje, pregled njihovih objav in dodajanje novih objav s pomočjo zaznavanja konteksta uporabnika.

Za zaznavanje uporabljamo vmesnik `MasterDetector`, ki vsebuje vse metode, ki sprožijo naše zaznavanje. Te metode implementira `MainActivity`. V ozadju sproži zaznavanje in nato čaka, da dobimo rezultate posameznega zaznavanja. Zaznavanje poteka v ozadju, da lahko uporabnik nemoteno uporablja aplikacijo naprej. Vsa zaznavanja razen zaznavanja lokacije sprožimo z objekti Intent. Ti objekti so asinhrona sporočila, ki jih lahko pošiljamo znotraj aplikacije. Z njimi dostopamo do različnih funkcionalnosti aplikacije ali pa z njimi pošiljamo podatke, ki jih drug del aplikacije potrebuje. Primer tega je pošiljanje rezultata operacije, ki se je izvajala asinhrono.

Aplikacija zazna število ljudi v okolici s pomočjo protokola Bluetooth. Lokacijo in gibanje zazna s pomočjo Googlovega aplikacijskega programskega vmesnika za zaznavanje lokacije in aktivnosti. Pri zaznavanju glasbe, ki jo uporabnik posluša, je aplikacija tista, ki posname zvočni posnetek in ga posreduje strežniku.

Za zaznavanje glasbe uporabljamo razred `Service` [13], ki se uporablja za izvajanje daljših nalog v ozadju. Ni odvisna od življenskega cikla objekta `Activity`. Izvaja se, četudi uporabnik preklopi na drugo aplikacijo. Primer uporabe je nalaganje vira iz interneta, posodabljanje (tudi za predvajanje glasbe), če je prisotno še obvestilo uporabniku o njenem izvajanju. Za snemanje glasbe uporabljamo mikrofon telefona, ki posname 12-sekundni zvočni posnetek. Ta posnetek se s pomočjo HTTP protokola pošlje našemu strežniku, ki nam vrne naslov pesmi in izvajalca.

Za zaznavanje števila ljudi v okolici prav tako uporabljamo `Service`. Pripravimo objekt `BluetoothAdapter`, s katerim dostopamo do funkcionalnosti protokola Bluetooth. Za naše potrebe potrebujemo samo odkrivanje števila naprav okoli nas. Pri zaznavanju si zapomnimo število naprav, ki smo jih

zaznali, in to posredujemo naši aplikaciji.

Zaznavanje lokacije poteka s pomočjo Googlovega aplikacijskega vmesnika. Preko njega pošiljamo zahtevo za lokacijo. Določimo interval posodabljanja lokacije in natančnost lokacije. Zaznavanje nam vrne zemljepisno širino in dolžino, ki ju pretvorimo v fizični naslov, preden pošljemo naš rezultat aplikaciji. Uporabljamo pravico `ACCESS_FINE_LOCATION`, kar pomeni, da dobimo največjo možno natančnost lokacije glede na GPS, WiFi in mobilne podatke. Podrobnosti zaznavanja lokacije so v podpoglavju 8.4.

`IntentService` uporabljamo za zaznavanje gibanja v ozadju. Ko opravi svoje delo, se sama ustavi. Primer uporabe `IntentService` je nalaganje vira z interneta. Ima metodo `onHandleIntent`, ki prejme rezultat zaznavanja v obliki objekta `Intent`. Rezultat je gotovost, s katero se izvaja določeno gibanje. Ta rezultat nato posredujemo naši aplikaciji z objektom `Intent`.

Strežnik skrbi za pridobivanje podatkov, ki jih aplikacija zahteva in za prepoznavanje zvočnih posnetkov, ki so mu poslani. Strežnik prejme zahtevo, ki vsebuje parametre. Če ima vse parametre in so vsi ustrezni, se poveže z bazo, izvede poizvedbo, dobi rezultat in vrne odgovor. Če je prišlo do kakšne napake, vrne aplikaciji odgovor z informacijo o napaki. Za zaznavanje pesmi strežnik uporablja Python knjižnjico Dejavu, o kateri bomo podrobneje govorili v poglavju 8. Trenutno bomo samo omenili, da je to knjižnjica, s katero analiziramo glasbo, hranimo rezultate analize v bazi in nato naše posnetke glasbe primerjamo z njimi in iščemo zadetek. Strežnik je podrobneje opisan v poglavju 5.

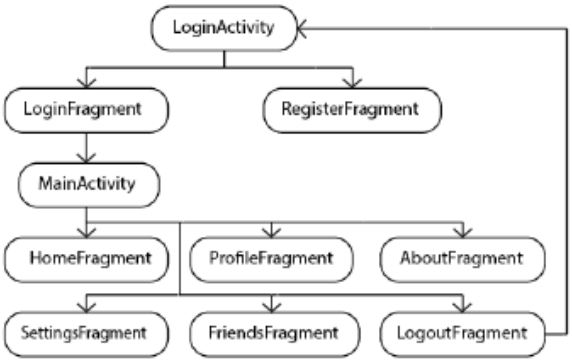
Baza hrani podatke za avtentikacijo uporabnikov, podatke o uporabnikih, njegove objave in rezultate analiz posameznih pesmi. Ko uporabnik pošlje posnetek pesmi, se ta posnetek analizira. Rezultat analize primerjamo z analizami, ki jih hranimo v bazi. Če pride do ujemanja, pomeni, da imamo v naši bazi že shranjeno pesem. Iz baze nato dobimo njen naslov in izvajalca.

Poglavje 3

Uporabniški vmesnik mobilne aplikacije

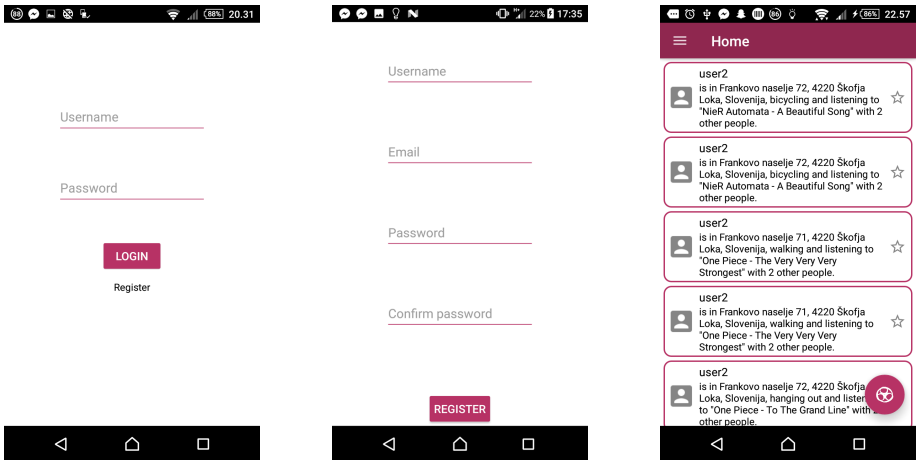
V tem poglavju bomo najprej v delu 3.1 predstavili razporeditev aplikacije in na splošno opisali posamezne strani. V naslednjih podpoglavjih opišemo posamezno stran.

3.1 Razporeditev aplikacije



Slika 3.1: Razporeditev aplikacije

V tem poglavju bomo opisali uporabniški vmesnik naše aplikacije. Našo aplikacijo smo razdelili na dva dela. Vsak od teh delov ima svoje strani, ki skrbijo za prikaz določene vsebine.



(a) Stran Login (b) Stran Register (c) Stran Home

(d) Stran Settings (e) Stran Friends (f) Stran About

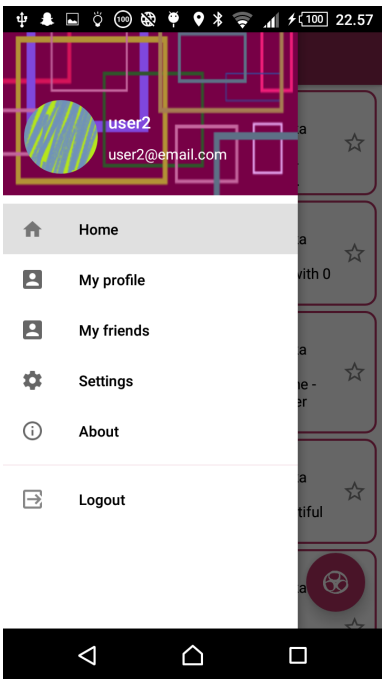
Slika 3.2: Videz posameznih strani aplikacije

Prvi del aplikacije ima dve strani. Na eni strani imamo vpis uporabnikov, na drugi pa imamo registracijo uporabnikov.

Drugi del ima pet strani. Imamo strani Home, My profile, My friends, Settings in About. Po vpisu se nam najprej prikaže domača stran Home.

Na njej imamo prikazane objave naših prijateljev. Preko navigacijskega menuja lahko obiščemo ostale strani. Stran My profile vsebuje podatke o uporabniškem računu in pregled objav uporabnika. Na strani My friend imamo pregled naših prijateljev. Stran About prikazuje logotip aplikacije in podatke o avtorici aplikacije in mentorju. Settings je stran, na kateri so na voljo nastavitve aplikacije.

Logotip in ozadji glave navigacijskega menija smo izdelali s pomočjo programa Adobe Illustrator.



Slika 3.3: Videz navigacijskega menuja

Aplikacijo smo želeleli smiselno ločiti na posamezne dele, ki bi tematsko spadali skupaj. Za vsak del smo ustvarili svoj `Activity`. Da bi bila delitev na dva dela bolj razvidna, smo se odločili, da bomo za posamezne dele aplikacije uporabili `Fragmente`.

Aplikacija ima tako dve aktivnosti: `LoginActivity` in `MainActivity`. `LoginActivity` je prva aktivnost, ki se požene ob zagonu aplikacije. Ima dva `Fragmenta`: `LoginFragment` in `RegisterFragment`. V `LoginFragmentu`

se lahko uporabnik vpiše v našo aplikacijo ali pa klikne gumb za registracijo, ki prikaže `RegisterFragment`, kjer se lahko registrira.

`MainActivity` ima pet `Fragmentov`: `HomeFragment`, ki prikazuje objave prijateljev uporabnika, `ProfileFragment`, ki vsebuje podatke o uporabniku in njegove objave, `FriendsFragment`, ki omogoča pregled nad prijatelji uporabnika in omogoča njihovo urejanje, `SettingsFragment`, ki vsebuje možnost menjave teme, in `AboutFragment`, ki prikazuje logotip aplikacije in podatke o avtorici in mentorju.

3.2 Vpis in registracija

Za vpis v aplikacijo mora uporabnik vnesti uporabniško ime in geslo. Pri registraciji pa vpiše uporabniško ime, e-pošto, geslo in potrditev gesla.

Uporabniško ime mora biti vsaj tri znake dolgo. Vsebuje lahko le male in velike črke, števila od 0 do 9 in podčrtaj. Geslo mora biti dolgo vsaj šest znakov. Vsebovati mora vsaj eno število, veliko črko in vsaj eno malo črko. Za preverjanje naslova e-pošte uporabljamo Androidov razred `Patterns`. Uporabniško ime in e-pošta morata biti unikatna. Dva registrirana uporabnika ne moreta imeti enakega uporabniškega imena ali e-pošte. Podvajanje le-teh preverjamo ob kliku na gumb `register`. Preden registriramo uporabnika, strežnik pogleda v bazo, če že imamo uporabnika s tem uporabniškim imenom ali e-pošto. Če to drži, o tem obvestimo uporabnika. Podvajanja ne preverjamo sproti, da zmanjšamo komunikacijo s strežnikom.

Za sprotno preverjanje vnosa v polja za registracijo uporabljamo `TextWatcher`. Tako uporabnik dobi odziv o pravilnosti vnosa, preden pritisne gumb register in lahko med vnosom dopolnjuje oziroma popravlja vnos. Ali sta uporabniško ime in e-pošta že zasedena, pa izve šele ob kliku na register, da zmanjšamo število strežniku poslanih zahtev.

3.3 Domača stran

Če je vpis uporabnika uspešen, se sproži `MainActivity`, medtem ko se `LoginActivity` zaključi. Prva stran, ki se nam prikaže po uspešnem vpisu, je domača stran - `HomeFragment`. Na tej strani se uporabniku prikažejo objave njegovih prijateljev. Osveževanje objav prijateljev ni avtomatsko, ker smo želeli dati uporabniku večji nadzor nad prikazovanjem objav in porabo njegovih mobilnih podatkov. Osvežitev se sproži z vertikalnim podrsavanjem po ekranu. Uporabnik lahko navigira po aplikaciji preko uporabniškega menuja. Do njega lahko dostopa s pritiskom na menu ikono ali s podrsavanjem s prstom iz leve proti desni.

3.4 Navigacijski menu

Navigacijski menu je sestavljen iz glave in iz samega menuja. Glava vsebuje uporabniško ime in e-pošto vpisanega uporabnika. Menu vsebuje vse strani, ki so na voljo uporabniku. Zadnja izbira je Log out, ki uporabnika izpiše iz aplikacije.

3.5 Moj profil

Moj profil omogoča uporabniku pregled svojega računa. Na zgornjem delu imamo podatke uporabnika (uporabniško ime, e-pošto, kratek opis). Pod tem pa imamo del My activities. Tu so prikazane objave vpisanega uporabnika.

3.6 Moji prijatelji

Sekcija Friends prikazuje seznam že potrjenih prijateljev uporabnika in seznam čakajočih prijateljev. Čakajoči prijatelji so uporabniki, ki so vpisanega uporabnika dodali kot prijatelja. Vpisani uporabnik lahko čakajoče prijatelje sprejme ali zavrne. Če jih sprejme, se dodajo na seznam prijateljev, zavr-

njeni pa se izbrišejo iz seznama čakajočih uporabnikov. Na vrhu strani ima uporabnik gumb za dodajanje uporabnikov med prijatelje. Ob kliku nanj se nam odpre dialog z vnosnim poljem in gumbom dodaj uporabnika.

3.7 Nastavitve

Sekcija Settings upravlja z nastavitvami aplikacije. Uporabnik lahko tu vklopi ali izklopi avtomatsko zaznavanje aktivnosti in spremeni temo aplikacije. Uporabnik lahko izbira med temno Drowner temo in svetlo Vine Red temo.

3.8 O aplikaciji

Sekcija About vsebuje logotip aplikacije, ime avtorice in ime mentorja.

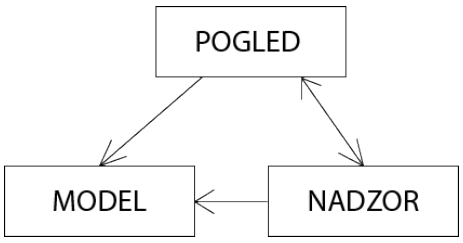
Poglavje 4

Povezovanje podatkov v mobilni aplikaciji

V tem poglavju bomo najprej opisali tri arhitekturne vzorce (MVC, MVP, MVVM). V 4.1 bomo podrobneje opisali naš izbrani arhitekturni vzorec (MVVM). V delu 4.2 bomo opisali implementacijo našega arhitekturnega vzorca, v delu 4.3 pa bomo govorili u povezovanju podatkov s pomočjo knjižnjice Data Binding.

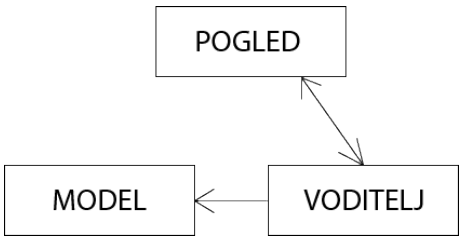
Arhitekturni vzorci nam pomagajo ločiti aplikacije na tri dele: podatke, videz in logiko [22]. Tako ločena aplikacija je bolj pregledna, testiranje je enostavnejše in več ljudi jo lahko hkrati razvija. Posamezne dele lahko po potrebi spremenimo ali ponovno uporabimo pri razvoju drugega projekta.

Vzorce, ki smo si pogledali, so Model-View-Controller (MVC), Model-View-Presenter (MVP) [20] in Model-View-ViewModel (MVVM) [31].



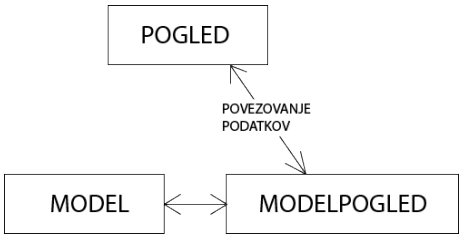
Slika 4.1: Arhitekturni vzorec MVC

MVC je sestavljen iz modela, pogleda in nadzora. Pri MVC lahko postane testiranje nadzora težje, ker se sam nadzor in njegova kompleksnost večata z rastjo aplikacije. Kodo naše aplikacije smo želeli bolj razporejeno, zato se nismo odločili zanj. Prav tako sta pri MVC pogled in model povezana. Pogled spremlja model za spremembe, ki jih nato prikaže uporabniku.



Slika 4.2: Arhitekturni vzorec MVP

MVP je sestavljen iz modela, pogleda in voditelja. MVP skoraj popolnoma loči pogled in model. Nimata več neposredne povezave. Voditelj opravlja komunikacijo med njima. Pogled obvesti voditelja o akciji uporabnika, voditelj pa ustrezno spremeni model in o spremembah obvesti pogled. Pogled se še vedno zaveda voditelja, kar otežuje testiranje.



Slika 4.3: Arhitekturni vzorec MVVM

MVVM je sestavljen iz modela (Model), pogleda (View) in modelPogleda (ViewModel). Pri MVVM si lahko več pogledov deli isti modelPogled. ModelPogled se ne zaveda pogleda. Testiranje je lažje; neodvisnost posameznih delov je večja. MVVM smo izbrali tudi zaradi povezovanja podatkov, ki nam olajša spreminjanje in posodabljanje prikazanih podatkov. S tem povezovanjem smo o spremembah podatkov obveščeni in podatki se posodobijo.

Imamo enosmerno in dvosmerno povezovanje podatkov. Pri enosmernem se nam podatki v pogledu posodobijo, ko se spremeni model. Pri dvosmernem povezovanju pa lahko tudi pogled spremeni podatke in se hkrati posodobi model.

Za MVVM smo se odločili, ker potegne najdebelejše črte med videzom, podatki in logiko. Uporabili smo ga v kombinaciji s knjižnjico Data Binding [5].

4.1 Arhitektura MVVM

4.1.1 Model

Model hrani podatke aplikacije. Model je lahko naša lokalna baza ali pa preprost objekt razreda, ki smo ga ustvarili za hranjenje podatkov.

4.1.2 View

Pogled (View) je videz aplikacije oziroma tisto, kar uporabnik vidi. Skrbi za prikaz podatkov. Podatke dobi preko ViewModela ali pa se na njih naroči s pomočjo knjižnjice Data Binding.

Zaznava uporabnikove akcije, kot so na primer pritisk na gumb, skrbi za prikaz obvestil in opozoril, spreminjanje strani oziroma preklapljanje med `Activity` oziroma `Fragmenti`, za dovoljenja, ki jih naša aplikacija potrebuje. `Activity` in `Fragment` sta primera pogleda.

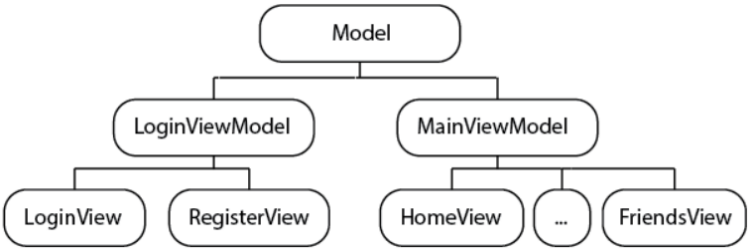
4.1.3 ViewModel

ViewModel [16] je logika aplikacije. ViewModel prejme obvestilo o akciji uporabnika in se odzove nanjo s klicanjem ustreznih metod. Če je pri tem model spremenjen, o tem obvesti ViewModel. Ta pa obvesti View. Primer: Uporabnik vnese novo uporabniško ime in pritisne gumb shrani. View zazna klik uporabnika. O akciji obvesti ViewModel. Ta sproži ustrezno akcijo, na primer pokliče metodo shrani, ki posodobi hranjeno uporabniško ime. Model

se je tako spremenil. O tem je obveščen ViewModel, ki o tem obvesti View in uporabniku je tako na zaslonu prikazano novo uporabniško ime.

4.2 Implementacija MVVM

V tem poglavju bomo predstavili, kako smo implementirali arhitekturni vzorec MVVM.



Slika 4.4: Implementacija MVVM

Odločili smo se, da si bodo posamezne `Activity` in njeni `Fragmenti` delili ViewModel. Za to smo se odločili, ker bo to omogočilo lažjo komunikacijo med posameznimi deli, kot če bi vsak `Fragment` (pogled) imel svoj ViewModel. `LoginActivity`, `LoginFragment`, `RegisterFragment` imajo `LoginActivityViewModel`. `MainActivity`, `HomeFragment`, `ProfileFragment`, `FriendsFragment`, `SettingsFragment`, `AboutFragment` imajo `MainActivityViewModel`.

ViewModel se ne sme zavedati Viewa. To pomeni, da ne sme vedeti ničesar o samem razredu `Activity` in njegovem kontekstu (`Context` [9]). `Context` je vmesnik, s katerim dostopamo do globalnih informacij o okolju naše aplikacije. Z njim dostopamo do virov in razredov naše aplikacije. Primer uporabe je menjava strani (`Fragmentov`) naše aplikacije in preklop iz `LoginActivity` na `MainActivity`, kjer s pomočjo `Contexta` poženemo novo `Activity`. Odločili smo se, da bomo `Context` ovili v nek drug objekt, preko

katerega bo lahko ViewModel izvajal vse stvari, ki potrebujejo kontekst. Zato smo ustvarili vmesnika `LoginNavigator` in `MainNavigator`.

Vmesnika `LoginNavigator` in `MainNavigator` skrbita za navigacijo v aplikaciji, menjavo oziroma izbiro posameznih strani in prikaz dialogov. Primer: metoda vmesnika `LoginNavigator` `goToRegisterFragment` se kliče ob kliku na gumb register in nam zamenja prikazan `LoginFragment` za `RegisterFragment`. Metoda `startMainActivity` prične `MainActivity` in zaključi `LoginActivity`. S klicanjem metode `lnAlertDialog` sprožimo `Alert Dialog`. Oba vmesnika imata metodo, ki vrne kontekst. Obe aktivnosti implementirata svoj `Navigator`. Ustvarita objekt `Navigator`, ki ga nato podata svojemu ViewModelu. ViewModel tako lahko kliče metode `Navigatorja` in po potrebi prikaže uporabniku opozorilo.

4.3 Implementacija povezovanja podatkov s knjižnjico Data Binding

Knjižnjica Data Binding [21] je vmesni sloj med Viewom in ViewModelom. Z njo se lahko povežemo z določenimi podatki in ko se ti podatki spremenijo, smo o tem obveščeni in se naši podatki posodobijo.

Da lahko uporabimo Data Binding moramo malce preurediti naš View. Vsebino naše xml datoteke prestavimo znotraj značke <`layout`>. Takoj pod značko <`layout`> in nad našo xml vsebino dodamo še značko <`data`>. Znotraj te značke definiramo podatke, ki bodo na voljo za povezovanje (binding). Tako lahko v delu data definiramo na primer objekt `Uporabnik` (`User`).

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/
    android">
    <data>
        <import type="android.view.View" />
        <variable name="user"
        type="com.example.rosa.diplomska.model.Entity.User"/>
```

```
        <variable name="mavm"
        type="com.example.rosa.diplomska.viewModel.
    MainActivityViewModel" />
    </data>
    <RelativeLayout
        android:focusableInTouchMode="true"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        ...
```

V naši xml vsebini nato lahko dostopamo do atributov našega objekta. Primer: če želimo v elementu `TextView` prikazati atribut uporabniško ime našega objekta `Uporabnik`:

```
<TextView
    ...
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@{user.username}" />
```

To bo prikazalo ime uporabnika, ki se bo posodobilo vsakič, ko se bo vrednost imena spremenila. To je enosmerno povezovanje podatkov. Na voljo imamo tudi dvosmerno povezovanje podatkov.

Pri dvosmernem povezovanju poteka sinhronizacija podatkov v obe smeri. Če se spremeni model, se bo sprememba prikazala uporabniku. Hkrati lahko uporabnik spremeni vrednost in se posodobi model. Primer uporabe tega načina povezovanja imamo v datoteki fragment_login.xml. V data delu imamo definiran objekt razreda `LoginCredentials`, ki se uporablja za hranjenje uporabnikovega vnosa. Dvosmerno povezovanje označimo tako, da namesto @{} uporabimo @={}. Na začetku je `loginUsername` tak, kot je definiran v modelu – prazen. Uporabniku se prikaže prazen vnos. Ko uporabnik vnese svoje uporabniško ime, se vrednost `loginUsername` v modelu posodobi na vneseno vrednost. Preureditev naše xml datoteke nam ustvari `DataBinding` razred. Ustvarjeni razred je ime xml datoteke brez podčrtajev. Vsaka beseda se začne z veliko začetnico. Na koncu dodamo še Binding. Primer razreda binding za `LoginActivity`, ki ima xml datoteko activity_login.xml:

```
ActivityLoginBinding binding = DataBindingUtil.setContentView(
    this, R.layout.activity_login);
```

Databinding nam omogoča tudi lažje dostopanje do posameznih grafičnih elementov v kodi. Namesto da pišemo

```
TextView tv =
(TextView) findViewById(R.id.text_view_login_username);
tv.setText("Uporabnik");
```

, dostopamo do elementa z

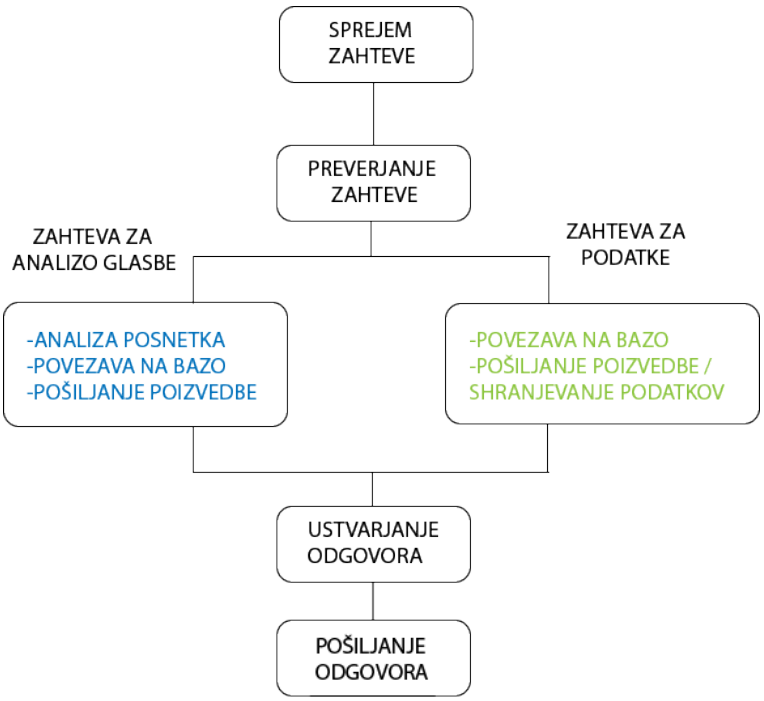
```
binding.textViewLoginUsername.setText("Uporabnik");
```

Zgoraj vidimo, da je ime našega elementa njegov identifikator, ki se začne z malo začetnico in nato ima vsaka naslednja beseda veliko začetnico.

Za uporabo Data Biding knjižnjice smo se odločili, ker nam je olajšala posodabljanje informacij, ki jih uporabnik vidi. Brez Data Binding knjižnjice bi morali sami implementirati zaznavanje sprememb v podatkih, posodabljanje in obveščanje vseh elementov, ki hranijo te podatke. Vse to nam Data Binding olajša. Razred, ki hrani podatke, mora podaljšati `BaseObservable`. Nad našo getter metodo dodamo anotacijo `@Bindable`. V setter metodi po nastavljanju vrednosti pokličemo metodo, ki obvesti naročnike na ta podatek, da se je vrednost spremenila. Primer te metode je `notifyPropertyChanged`.

Poglavje 5

Spletni strežnik



Slika 5.1: Delovanje spletnega strežnika

V tem poglavju bomo predstavili spletni strežnik. Opisali bomo njegovo vlogo, kako komuniciramo s strežnikom, kako strežnik izvede posamezno zahtevo, ki jo dobi. V delu 5.1 bomo predstavili, kako strežniku pošljemo zah-

tevo. V podpoglavju 5.2 opišemo, kako iz strežnika dostopamo do baze.

Naš spletni strežnik je vmesnik med našo aplikacijo in podatkovno bazo, ki hrani naše podatke. Kadar želi naša aplikacija pridobiti neke podatke ali pa jih shraniti, mora poslati strežniku zahtevo za te podatke oziroma za shranjevanje podatkov. Strežnik prejme zahtevo, izvede ustrezno akcijo in vrne aplikaciji odgovor. Odgovor strežnika vsebuje informacije o uspešnosti zahtevane akcije in podatke, če jih je aplikacija zahtevala in je bila zahtevana akcija uspešna.

Poleg pošiljanja in shranjevanja podatkov je naloga našega strežnika tudi prepoznavanje glasbe, ki jo uporabnik posluša. Za prepoznavanje moramo imeti podatkovno bazo, v kateri hranimo pesmi. Aplikacija pošlje posnetek glasbe, ki jo uporabnik posluša. Prejeti posnetek nato primerjamo s pesmimi v bazi in iščemo zadetek. Če v naši bazi najdemo pesem, pošlje strežnik aplikaciji podatke o izvajalcu in naslov pesmi, sicer sporoči, da je pesem neznana. Podrobnosti o tem so v poglavju 8.

Za strežnik smo uporabili Wamp Server [33], razvojno okolje za razvoj spletnih aplikacij na operacijskem sistemu Windows. Naložili smo verzijo WampServer 3.1.0 64bit, ki vsebuje Apache 2.4.27, PHP 5.6.31, MySQL 5.7.19. Samo kodo za strežnik smo pisali v jeziku PHP. PHP smo izbrali ker je odprtokoden, ima veliko podpore in dokumentacije.

Odločili smo se za uporabo brskalnika Chrome. Za pomoč pri razhroščevanju smo uporabili ChromePHP [3], ki je po novem znan kot Chrome Logger.

S telefonom smo do strežnika dostopali preko brezžičnega omrežja. Da smo lahko dostopali do vsebine na strežniku smo v httpd-vhosts.conf zamenjali „Require local“ za „Allow from local“ in „Allow from 192.168.1.0/8“.

Ker je večina interakcije z bazo pridobivanje podatkov o uporabnikih, smo za pošiljanje na strežnik uporabili metodo POST. Za POST smo se odločili, ker je ta metoda varnejša od GET [17]. Pri metodi GET so vsi naši poslani parametri vidni v naslovu URL. POST jih ne prikazuje in podatki se ne shranijo v zgodovino brskalnika. Pri GET lahko pošljemo omejeno število podatkov oziroma parametrov, ker je dolžina URL omejena. Pošljemo lahko

toliko parametrov, kot nam jih uspe dodati v URL. Pri POST teh omejitev ni.

Za komunikacijo med strežnikom in stranko uporabljamo HTTP knjižnjico Volley [15]. HTTP je protokol, ki uporablja model odjemalec-strežnik. Odjemalec pošlje zahtevo strežniku. Strežnik opravi delo in vrne odjemalcu odgovor. Odjemalec komunicira direktno s končno točko. Knjižnjica Volley nam omogoča enostavno in hitro implementacijo komunikacije [15]. Tako smo se odločili, ker sam upravlja z omrežnimi zahtevami. Lahko prekličemo zahtevo in jim dodajamo prioritete. Izvajamo lahko več zahtev sočasno. Izgledal je enostaven za implementacijo. Pričakovali smo, da bomo večinoma samo pridobivali in pošiljali String podatke, ki smo jih enostavno dodali med parametre kot par ključa in vrednosti. Nismo imeli predvidenih ogromnih prenosov podatkov. V literaturi je bilo navedeno, da je Volley priporočen za manjše zahteve HTTP. Četudi bi se nam nabralo na primer veliko podatkov v post tabeli, bi to lahko kontrolirali z omejevanjem pridobivanja števila objav, ki bi se nalagale in posodabljale postopoma, ko bi uporabnik prišel do dna seznama s podrsavanjem.

Za pošiljanje zvočne datoteke na strežnik smo najprej nameravali implementirati svojo Volley zahtevo [7]. Odločili smo se, da bo enostavneje, če serializiramo posneto datoteko, pretvorimo dobljeno tabelo bajtov v `String` in dodamo ta `String` k parametrom, ki jih nato pošiljamo strežniku. Tako imamo manj kode, ker nam ni bilo potrebno implementirati dodatne zahteve samo za eno funkcionalnost in smo lahko uporabili že obstoječo zahtevo.

Vse zahteve so tipa `JsonObjectRequest` [8], kar pomeni, da pri komunikaciji uporabljamo objekte `JSON`. Aplikacija in strežnik shranita vrednosti, ki jih želimo poslati v objekt v obliki ključa in vrednosti. Ta objekt nato zapakiramo v `JSON` in ga pošljemo. Ključ in vrednost sta tipa `String`.

Na strežniku imamo razred `UserFunctions`, ki vsebuje vse metode, ki jih aplikacija potrebuje. Strežnik prejme objekt `JSON`, ki vsebuje parametre. Eden od parametrov je ime funkcije, ki jo stranka strežnika želi izvesti. Objektu razreda `DB_Handler` se poda ime želene funkcije in parametre. Ta

preveri, ali so vsi parametri, ki jih potrebuje funkcija, nastavljeni. Če niso, vrne napako, sicer izvede želeno funkcijo razreda `UserFunctions`.

Vse funkcije so poleg zaznavanja glasbe v razredu `UserFunctions`. Izbrana funkcija se poveže z bazo, izvede določeno poizvedbo, pridobi podatke in vrne odgovor.

Odgovor strežnika je prav tako objekt `JSON`. Sestavljen je iz spremenljivke `success`, ki lahko zavzame vrednost 0, ki pomeni neuspeh, ali vrednost 1, ki pomeni uspeh, odvisno od uspešnosti izvedene funkcije. Drugi element je `message`, ki je kratko sporočilo o uspehu oziroma neuspehu. Tretji element je rezultat funkcije - podatki, ki jih je uporabnik želel pridobiti iz baze.

5.1 Komunikacija aplikacije s spletnim strežnikom

Razred `DataProvider` skrbi za komunikacijo aplikacije s strežnikom. Ima vse metode za pridobivanje podatkov, v katerih implementira vse naše Volley zahteve z izjemo zahteve, ki jo uporabljamo za pošiljanje zvočne datoteke na strežnik. Ta je implementirana v razredu `SongDetectionService`.

Primer ene od metod za vstavljanje objave v bazo:

```
public void insertUserPost(final Post post) {
    String insertPostUrl = serverAddr;
    String insertPostTag = "INSERT_POST_TAG";

    Map<String, String> params = new HashMap<>();
    params.put("fun","addPost");
    params.put("poster_id",Integer.toString(post
        .getPosterId()));
    params.put("content",post.getContent());
    params.put("timestamp",post.getTimestamp().toString());
    params.put("favourite_counter",Integer.toString(post
        .getFavouriteCounter()));

    JSONObject jp = new JSONObject(params);
```

```
JsonObjectRequest insertUserPostRequest =
    new JsonObjectRequest(Request.Method.POST,
        insertPostUrl, jp,
        new Response.Listener<JSONObject>() {
            @Override
            public void onResponse(JSONObject response) {
                try {
                    if(response.getInt("success") == 0) {
                        mainNavigator.
                        mnAlertDialog("Ooops!",
                        response.getString("message"));
                        mainNavigator.getUser()
                        .deletePost(post);
                        mainNavigator.getProfileFragment()
                        .notifyProfileAdapterOfChange();
                    }
                } catch (JSONException e) {
                    e.printStackTrace();
                    mainNavigator.mnAlertDialog("Ooops!",
                    "Error while parsing response.");
                    mainNavigator.getUser().deletePost(post)
;
                    mainNavigator.getProfileFragment()
                    .notifyProfileAdapterOfChange();
                }
            }
        }, new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError error) {
                VolleyLog.d(TAG, "Error: "
                + error.getMessage());
                mainNavigator.mnAlertDialog("Ooops!",
                "Something went wrong. Please try again"
                +" later.");
                mainNavigator.getUser().deletePost(post);
                mainNavigator.getProfileFragment()
                .notifyProfileAdapterOfChange();
```

```
                }
            });
    Context mContext = mainNavigator.getMNContext();
    AppSingleton appSingleton = AppSingleton.
        getInstance(mContext);
    appSingleton.addToRequestQueue(insertUserPostRequest,
        insertPostTag);
}
```

`InsertPostUrl` hrani naslov našega strežnika. `InsertPostTag` je oznaka naše zahteve. Najprej pripravimo naše parametre, ki jih bomo poslali. Shranimo jih v objekt `Map<String, String>`. Posamezen parameter je skupek ključa in vrednosti. Parameter `fun` hrani ime naše funkcije, ki jo želimo izvesti. `Poster_id` nam pove identifikator uporabnika, ki je lastnik objave. `Content` vsebuje vsebino objave. `Timestamp` nam pove čas, ko se je objava ustvarila, medtem ko `favourite_counter` pove, koliko ljudi je dodalo objavo kot priljubljeno. Naše parametre nato ovijemo v objekt `JSON`.

Ko imamo naše parametre, ustvarimo našo zahtevo, ki je tipa `JsonObjectRequest`. Povemo ji izbrano metodo HTTP, URL strežnika, parametre, objekt, ki posluša odgovor, in objekt, ki posluša napako. Objekt `Response.ErrorListener` posluša napake strežnika. `Response.Listener<JSONObject>` posluša odgovor. Ko dobimo naš `JSON` odgovor, ga poskušamo razčleniti. Med parametri imamo `success`, ki nam pove, ali je bila naša zahtevana funkcija uspešno izvedena. Če je bila neuspešna, je njena vrednost 0, sicer ima vrednost 1.

Če pride pri razčlenjevanju odgovora do napake, če je bila naša funkcija neuspešna ali pa če je naš `ErrorListener` prejel napako strežnika, o tem opozorimo uporabnika. Objava, ki smo jo poslali, se izbriše iz seznama objav in na koncu še obvestimo naš seznam o spremembi, da se ta posodobi.

5.2 PHP podatkovni objekti

Za dostop do baze iz strežnika smo uporabili PHP Data Objects (PDO) [4]. PDO doda abstraktni sloj oziroma nivo dostopanja do baze. Za PDO smo se odločili, ker uporablja tako imenovane pripravljene izjave [32]. Pri pripravljenih izjavah bazi najprej pošljemo našo poizvedbo brez parametrov. Baza poizvedbo pripravi za izvajanje tako, da jo razčleni, prevede in optimizira. Nato pošljemo bazi naše parametre, medtem ko baza izvede poizvedbo. Tako se poizvedba pripravi samo enkrat ne glede na število izvajanj. Zato imamo manj prometa in hitrejše izvajanje. Imamo tudi zaščito pred vrivanjem stavkov SQL, ker se parametri pošiljajo po drugem protokolu.

Bazi s funkcijo prepare pošljemo našo poizvedbo, da jo pripravi na izvajanje. Poizvedba se pošlje brez parametrov. Na mestih, kjer bomo podali naše parametre to označimo z podpičjem in nekim smiselnim imenom. Običajno je to ime atributa, ki ga v poizvedbi želimo nastaviti. Baza poizvedbo razčleni, prevede, optimizira in jo shrani. Vse, kar nam še ostane, je, da bazi pošljemo naše parametre.

Naše vrednosti moramo povezati z mesti, ki smo jih prej označili s podpičji. To naredimo s funkcijo `bindValue`. Podamo ji ime našega označenega mesta in vrednost, ki jo želimo poslati bazi. Poizvedbo izvedemo s klicanjem metode `execute`. Tako se poizvedba pripravi samo enkrat ne glede na število izvajanj. Pošljemo jo samo enkrat in ne vsakič, ko želimo izvesti poizvedbo. Zato imamo hitrejšo izvedbo in manj prometa, ker moramo bazi pošiljati zgolj naše parametre. Imamo tudi zaščito pred vrivanjem stavkov SQL, ker se parametri pošiljajo po drugem protokolu.

```
$stmt = $conn->prepare('INSERT INTO user '.
'(username, email, description) VALUES '.
'(:username, :email, :description)');
$stmt->bindValue(':username',$params["username"]);
$stmt->bindValue(':email',$params["email"]);
$stmt->bindValue(':description',"");
$insertedUser = $stmt->execute();
```

V zgornjem primeru je PDO uporabljen za vstavljanje novega uporabnika. S podpičji smo označili. da bomo vstavili uporabniško ime, e-pošto in opis. S `bindValue` povežemo uporabniško ime in e-pošto z vrednostmi iz naših parametrov, medtem ko za opis vstavimo prazno vrednost.

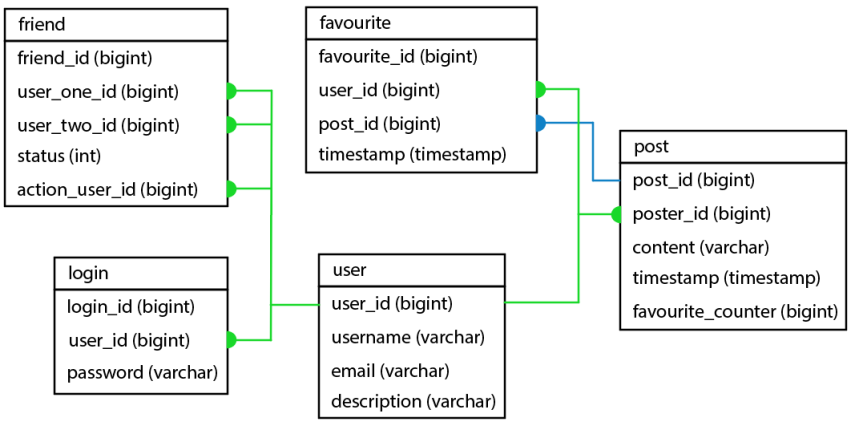
Poglavje 6

Baza podatkov

V tem poglavju bomo govorili o bazi, ki hrani podatke naše aplikacije. Najprej jo bomo na splošno opisali. V naslednjih podglavjih bomo opisali posamezno tabelo.

V eni bazi hranimo podatke o uporabnikih, njihove objave, prijateljstva. V drugi bazi hranimo podatke o pesmih, ki jih potrebujemo za prepoznavanje glasbe, ki jo uporabnik posluša. Prva baza ima tabele `user`, `post`, `login`, `friend`, `favourite`. Tabela `user` hrani podatke o uporabniku. Tabela `login` hrani geslo uporabnika. Tabela `friend` hrani informacije o prijateljstvih. Tabela `post` hrani objave uporabnikov. Tabelo `favourite` smo dodali v primeru, da bi implementirali funkcionalnost označevanja objav uporabnikov kot priljubljene. Hranila bi kateri uporabnik je označil katero objavo kot priljubljeno.

Za bazo smo uporabili MySQL (mysql 5.7.19), za administracijo baze pa smo uporabljali phpMyAdmin.



Slika 6.1: Tabele v bazi

6.1 Tabela user

Tabela `user` vsebuje:

- `user_id`, ki je identifikator uporabnikov,
- `username`, ki hrani uporabniško ime,
- `email`, ki hrani e-pošto uporabnika,
- `description`, ki hrani kratek opis uporabika.

Vsak identifikator uporabnikov je edinstven, kar pomeni, da si dva uporabnika ne moreta deliti iste številke. `Username` in `email` sta tudi edinstvena. Opis uporabnika je poljuben; ostali podatki v tej tabeli so obvezni.

6.2 Tabela login

Tabela `login` vsebuje identifikator `login_id`, hrani identifikator uporabnika (`user_id`) in uporabnikovo geslo (`password`).

6.3 Tabela friend

Tabela `friend` vsebuje `friend_id`, `user_one_id`, `user_two_id`, `action_user_id`. Pove nam, kateri uporabniki so prijatelji med seboj in kakšno je stanje njihovega prijateljstva. Ko eden od uporabnikov doda drugega kot prijatelja, se ta informacija shrani v bazi. Da postaneta prijatelja, mora drugi uporabnik potrditi prošnjo za prijateljstvo. Dokler se to ne zgodi, je prvi uporabnik čakajoči prijatelj drugega uporabnika. V bazi je to prikazano tako, da hranimo `user_one_id`, `user_two_id`, `status`, `action_user_id`. `User_one_id` je manjši identifikator od obeh uporabnikov. `User_two_id` je večji identifikator od obeh uporabnikov. Status opisuje stanje prijateljstva med njima. 0 pomeni, da prijateljstvo med njima še ni potrjeno. 1 pomeni, da je prijateljstvo med njima potrjeno. `Action_user_id` hrani identifikator uporabnika, ki je poslal prošnjo za prijateljstvo.

friend_id	user_one_id	user_two_id	status	action_user_id
1	1	2	0	1
2	1	3	1	1

Slika 6.2: Tabela friends

Na sliki vidimo dva primera prijateljstva. V prvi vrstici gledamo prijateljstvo med uporabnikoma z identifikatorjem 1 in 2. Status prijateljstva je 1, kar pomeni, da je bilo prijateljstvo potrjeno. `Action_user_id` je 1, kar pomeni, da je uporabnik z identifikatorjem 1 dodal uporabnika z identifikatorjem 2, ta pa je prijateljstvo potrdil. V drugi vrstici vidimo, da prijateljstvo med uporabnikoma z identifikatorjema 1 in 3 še ni potrjeno, ker je status prijateljstva 0. `Action_user_id` je 1, kar pomeni, da je uporabnik z identifikatorjem 1 dodal uporabnika z identifikatorjem 3 in sedaj čaka, da ga dodani uporabnik potrdi. `Friend_id` je identifikator prijateljstva.

6.4 Tabela post

`Post` tabela hrani objave uporabnikov. Vsebuje :

- `post_id`, ki je identifikator posamezne objave,
- `poster_id`, ki je je identifikator uporabnika, ki je objavo objavil,
- `timestamp`, ki je čas, ko je bila objava ustvarjena,
- `content`, ki je vsebina objave,
- `favourite_counter`, ki nam pove število ljudi, ki so označili objavo kot priljubljeno.

`Favourite_counter` je bil dodan zaradi primera, če bi bila implementirana možnost označevanja objave kot priljubljene. Zato je bila dodana tudi tabela `favourite`, ki hrani `favourite_id`, `user_id`, `post_id`, `timestamp`. `Favourite_id` je identifikator posamezne priljubljenosti, ki bi jo hranili v tabeli `favourite`. `User_id` je identifikator uporabnika, ki bi označil neko objavo kot priljubljeno. `Post_id` je identifikator objave, ki bi jo uporabnik označil kot priljubljeno. `Timestamp` je čas, ko bi uporabnik to naredil. `Favourite_counter` je bil dodan v tabelo `post` zato, da bi se sproti povečeval, ko bi uporabniki dodajali objavo med priljubljene. Tako nam ne bi bilo potrebno izvajati `COUNT` operacije nad tabelo `favourite` vsakič, ko bi nas zanimal podatek o številu uporabnikov, ki so dodali objavo med priljubljene.

Poglavje 7

Shranjevanje podatkov v aplikaciji

V tem poglavju bomo govorili o shranjevanju podatkov v sami mobilni aplikaciji. V 7.1 bomo opisali posamezne objekte, ki hranijo podatke v naši aplikaciji. V 7.2 bomo opisali knjižnjico Room Persistence, ki jo uporabljamo za bazo na uporabnikovi strani.

Določene podatke shranjujemo tudi v sami aplikaciji, da zmanjšamo komunikacijo s strežnikom in da lahko uporabljamo aplikacijo tudi, kadar ne moramo dostopati do strežnika. Hranimo informacijo o vpisu uporabnika, tako da se ni potrebno vpisati v aplikacijo vsakič, ko jo odpremo. Prav tako hranimo vsebino, ki smo jo že prejeli od strežnika - naše objave in objave prijateljev. Zapomnimo si tudi, katero temo aplikacije je uporabnik izbral. Aplikacije hrani tudi podatke o tem, kateri senzorji so nam na voljo za zaznavanje.

Za hranjenje podatkov v aplikaciji smo naredili več razredov, katerih objekti hranijo podatke, ki jih aplikacija potrebuje. Uporabili smo tudi knjižnjico Room Persistence [11] in objekt `SharedPreferences` [14].

7.1 Objekti

Razreda `Registration` in `LoginCredentials` uporabljamo zaradi knjižnjice Data Binding. Prvi hrani uporabnikov vnos pri registraciji. Drugi hrani uporabnikov vnos pri vpisovanju v aplikacijo. Razred `User` hrani podatke o uporabniku, razred `Post` pa podatke o posamezni objavi.

7.1.1 Razred LoginCredentials

Hrani vneseno uporabniško ime in geslo. Uporablja se na strani vpisa (`LoginFragment`). Razred je preprost in vsebuje le getterje in setterje za uporabniško ime in geslo.

7.1.2 Razred Registration

Poleg vnosa v polja registracije (uporabniško ime, e-pošta, geslo, potrditev gesla) hrani ta razred še posamezne napake, ki se lahko zgodijo pri vnosu. Zato imamo naslednje atribute:

- `registerEmailError`: hrani napake pri vnosu uporabniškega imena,
- `registerPasswordError`: hrani napake pri vnosu gesla,
- `registerConfirmPasswordError`: hrani napake pri vnosu potrditve gesla,
- `registerEmptyFieldsError`: hrani obvestilo o praznih vnosnih poljih.

7.1.3 Razred User

Razred `User` ima naslednje atribute: `userId`, `username`, `email`, `description`, `pending`, `posts`, `friends`, `favouritePosts`. `UserID` hrani identifikator uporabnika, `username` hrani uporabniško ime, `email` hrani e-pošto, `description` hrani opis uporabnika. `Posts` je tabela, ki hrani objave uporabnika. `Friends` je tabela, ki hrani prijatelje uporabnika. Ker se ta razred uporablja tudi za

prijatelje vpisanega uporabnika, imamo spremenljivko pending, ki pove, ali je naš objekt čakajoči prijatelj vpisanega uporabnika. Tabela `FavouritePosts` je bila dodana za primer implementacije označevanja objave kot priljubljene.

```
public class User extends BaseObservable {
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "userId")
    private int userID;
    @ColumnInfo(name = "username")
    private String username;
    @ColumnInfo(name = "email")
    private String email;
    @ColumnInfo(name = "description")
    private String description;
    @ColumnInfo(name = "pending")
    private boolean pending;

    @Ignore
    private ObservableArrayList<Post> posts;
    @Ignore
    private ObservableArrayList<User> friends;
    @Ignore
    private ObservableArrayList<Post> favouritePosts;
    ...
```

7.1.4 Razred Post

`Post` je razred, katerega objekti hranijo informacije o posamezni objavi. Hrani vse podatke, ki jih vsebuje tabela `post` v naši bazi (`post_id`, `poster_id`, `timestamp`, `content`, `favourite_counter`) in uporabniško ime uporabnika, ki je objavo objavil.

7.1.5 Objekt SharedPreferences

Objekt `SharedPreferences` se uporablja za shranjevanje manjše količine podatkov v obliki para ključa in vrednosti. Uporabljamo ga za shranjevanje podatkov vpisanega uporabnika in da si zapomnimo, ali je uporabnik vpisan. Tako se uporabniku ni potrebno akreditirati vsakič, ko požene aplikacijo, ampak se mu takoj prikaže domača stran aplikacije. `SharedPreferences` uporabljamo tudi za shranjevanje informacij o dovoljenjih in senzorjih. O tem bomo več povedali kasneje.

```
SharedPreferences pref = PreferenceManager.
    getDefaultSharedPreferences(loginNavigator.getLNContext());
SharedPreferences.Editor editor = pref.edit();
editor.putString("username",username);
editor.putInt("userId",userId);
editor.putString("email",email);
editor.putString("description",description);
editor.putBoolean("login",true);
editor.apply();
```

7.2 Knjižnjica Room Persistence

Za bazo na uporabnikovi strani (v aplikaciji) uporabljamo knjižnjico Room Persistence [12]. Za Room Persistence Library smo se odločili, ker doda abstraktni nivo k uporabi SQLite in nam olajša delo z bazo. Room je objektno-relacijsko mapiranje (ORM) med Java razredi in SQLite. ORM mapiranje pretvori podatke tako, da jih lahko uporabljamo v objektno orientiranih programskih jezikih. Z Room Persistence knjižnjico ustvarimo virtualno objektno podatkovno bazo, v kateri hranimo podatke naše aplikacije. Tako so lahko podatki na voljo uporabniku tudi, ko aplikacija ne bo imela dostopa do strežnika. Room Dao je sestavljen iz treh delov: baze, entitet, DAO.

7.2.1 Baza podatkov

Baza podatkov je v našem primeru razred `AppDatabase`. Podaljšati mora razred `RoomDatabase`. Nad razredom dodamo oznako `@Database`. Podamo tudi naše entitete in verzijo baze. V našem primeru je to `@Database(entitites = User.class,Post.class,version = 1)`.

```
@Database(entities = { User.class , Post.class}, version = 1)
@TypeConverters({RoomTypeConverters.class})
public abstract class AppDatabase extends RoomDatabase {

    private static final String DB_NAME = "localAdaDatabase.db";
    private static volatile AppDatabase instance;

    public static synchronized AppDatabase
    getInstance(Context context) {
        if (instance == null) {
            instance = create(context);
        }
        return instance;
    }

    private static AppDatabase create(final Context context) {
        return Room.databaseBuilder(
                context,
                AppDatabase.class,
                DB_NAME).fallbackToDestructiveMigration()
                .build();
    }

    public abstract UserDao getUserDao();
    public abstract PostDao getPostDao();
}
```

7.2.2 Entitete

Entitete predstavljajo posamezne tabele naše baze. V našem primeru sta to razreda `User` in `Post`. Entiteto označimo z oznako `@Entity`. Če želimo, da ima tabela kakšno drugo ime, to povemo tako, da nastavimo atribut `tableName`. Z oznako `@PrimeryKey` označimo naš primarni ključ.

```
@Entity(tableName = "user",
    indices = {
        @Index(value = "userId", unique = true),
        @Index("pending"),
        @Index(value = "username", unique = true)
    })
public class User extends BaseObservable {
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "userId")
    private int userID;
...
```

Če imamo več konstruktorjev v naši entiteti, označimo tiste, za katere ne želimo, da jih naša baza uporabi, z oznako `@Ignore`.

```
@Ignore
public Post(String username, String content) {
    this.posterUsername = username;
    this.content = content;
    this.timestamp = new Timestamp(System.currentTimeMillis());
}
```

7.2.3 DAO

DAO datoteke so vmesniki, ki vsebujejo vse naše interakcije s posameznimi tabelami. V našem primeru sta to `UserDao` in `PostDao`. Razred `DAO` označimo z oznako `@Dao`. Room Persistence Library zelo olajša delo z bazo. Namesto da pišemo celotno poizvedbo za vnos uporabnika, dodamo nad našo metodo za vnos oznako @Insert. Če želimo sami napisati poizvedbo, ki se bo izvedla

ko pokličemo metodo, uporabimo oznako `@Query` in v oklepajih podamo našo poizvedbo.

```
@Dao
public interface PostDao {
    @Query("SELECT * FROM post WHERE posterId=:posterId")
    List<Post> getUserPostsById(int posterId);
    @Query("SELECT * FROM post WHERE posterId!=:userId")
    List<Post> getHomePosts(int userId);
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    void insert(Post post);
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    void insert(ObservableArrayList<Post> posts);
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    void insert(List<Post> posts);
    @Delete
    void delete(Post post);
    @Delete
    void delete(ObservableArrayList<Post> posts);
}
```

Poglavje 8

Zaznavanje

V tem poglavju bomo govorili o zaznavanju, ki ga naša aplikacija opravlja. Na splošno bomo opisali potek zaznavanja. V 8.1 bomo opisali razred UserActivity, ki se uporablja za shranjevanje podatkov o posameznem zaznavanju. V 8.2 bomo opisali vmesnik MasterDetector, ki skrbi za začetek zaznavanja in za prikaz rezultata uporabniku. V 8.3 bomo opisali zaznavanje števila ljudi okoli nas. V 8.4 bomo govorili o zaznavanju lokacije. Podpoglavje 8.5 govori o zaznavanju gibanja, 8.6 pa o zaznavanju glasbe.

Naš kontekst uporabnika smo že definirali kot končni rezultat več zaznavanj. Zaznavamo število ljudi v okolici uporabnika, lokacijo uporabnika, gibanje uporabnika, glasbo, ki jo uporabnik posluša. Število ljudi zaznamo tako, da pogledamo, koliko naprav je v bližini. Za zaznavanje lokacije in gibanja uporabljamo aplikacijski programski vmesnik. Za zaznavanje glasbe uporabljamo knjižnjico, ki analizira posamezne pesmi. Rezultate teh analiz shrani v podatkovno bazo. Ko dobimo nek posnetek, ga najprej analiziramo in nato rezultat te analize primerjamo z rezultati analiz v naši bazi. Če imamo ujemanje v bazi, imamo podatke o izvajalcu in naslovu pesmi.

Uporabnik sproži zaznavanje s pritiskom na gumb ali pa z vklopom avtomatskega zaznavanja v nastavitvah. V drugem primeru aplikacija sproži zaznavanje vsake pol ure s pomočjo časovnika (timer). Če uporabnik sproži zaznavanje, preden poteče pol ure, se časovnik osveži in začne na novo odštevati.

Avtomatsko zaznavanje lahko vklopimo ali izklopimo v nastavitvah aplikacije.

Samo zaznavanje poteka tako, da najprej vprašamo uporabnika za vsa dovoljenja, ki jih potrebujemo za uporabo senzorjev. Če nimamo dovoljenja za senzor, ga ne uporabimo. Aplikacija počaka, da se vsa zaznavanja končajo. Nato pridobljene podatke prikaže z uporabniku prijaznim besedilom.

Za zaznavanje števila ljudi v okolici uporabljamo protokol Bluetooth. Za lokacijo in aktivnost uporabljamo Googlov aplikacijski programski vmesnik za zaznavanje lokacije in aktivnosti [26]. Za zaznavanje pesmi, ki jo uporabnik posluša uporabljamo Python knjižnjico Dejavu [34].

Za zaznavanje vseh teh stvari moramo uporabiti ustrezna dovoljenja, ki jih definiramo v manifestu naše aplikacije. Pri novejših verzijah Androida (6.0 in višje) uporabnik ne da več vseh dovoljenj pred nalaganjem aplikacije na telefon, ampak mora aplikacija zaprositi za dovoljenja sproti, preden uporabi funkcionalnost, ki potrebuje dovoljenje. Če uporabnik ne da dovoljenja, mu določena funkcionalnost ni na voljo. Dostopnost senzorjev in dovoljenj shranjujemo z objektom `SharedPreferences`.

Pridobljene informacije o uporabnikovem kontekstu oblikujemo v uporabniku prijazen tekst, ki ga uporabnik lahko objavi kot objavo. Informacije o kontekstu hrani objekt razreda `UserActivity`.

8.1 Razred UserActivity

Razred `UserActivity` uporabljamo za hranjene podatkov o posameznem zaznavanju. Vsebuje naslednje atribute: `userActivityDone`, `gotMusic`, `gotLocation`, `gotMovement`, `gotPeople`, `uaUsername`, `uaPeople`, `uaLocation`, `uaMusic`, `uaMovement`. Atribut `uaUsername` hrani uporabniško ime, `uaPeople` hrani število zaznanih ljudi, `uaLocation` hrani lokacijo, `uaMusic` hrani zaznano pesem, `uaMovement` pa zaznano gibanje.

Spremenljivke `gotMusic`, `gotLocation`, `gotMovement`, `gotPeople` so tipa boolean in nam povedo, ali je posamezno zaznavanje zaključeno. Ko prejmemo rezultat zaznavanja, nastavimo ustrezen atribut objekta tipa `UserActivity` in nastavimo njegovo got spremenljivko na `true`. Vsakič, ko se katera od teh spremenljivk nastavi, preverimo vrednost spremenljivke `userActivityDone`, ki je rezultat logične operacije in vseh teh got spremenljivk.

Če so vse nastavljene na drži (`true`), je naše zaznavanje končano in se pokliče metoda `onActivityDone`. To je metoda našega vmesnika `OnActivityDoneListener`. Naš `MainActivityViewModel` implementira ta vmesnik. V metodi `onActivityDone` iz pridobljenih podatkov sestavimo uporabniku prijazno besedilo, ki opiše kontekst uporabnika. Pokličemo metodo `activityDetectedDialog`, ki to besedilo prikaže uporabniku.

8.2 Vmesnik MasterDetector

Ustvarili smo vmesnik `MasterDetector`, ki ga implementira `MainActivity`. Večinoma vsebuje metode, ki preverijo, ali so vsi senzorji na voljo in ali smo dobili vsa potrebna dovoljenja od uporabnika. Metoda `startDetection` sproži vsa zaznavanja, medtem ko je `activityDetectedDialog` obvestilo, ki se prikaže uporabniku, ko je naše zaznavanje končano. Vsebuje rezultat zaznavanja. Uporabnik ima možnost objaviti rezultat kot svojo objavo, sicer zavrne obvestilo.

Ker se zaznavanja dogajajo v ozadju oziroma na svoji niti, imamo v `MainActivity` sprejemnike, ki prestrežejo rezultate posameznih zaznavanj.

Ti sprejemniki so `receiverP`, `receiverS`, `receiverM`, `receiverL`. `ReceiverP` prestreže rezultat zaznavanja števila ljudi, `receiverS` prestreže zaznano pesem, `receiverM` prestreže gibanje, `receiverL` pa lokacijo.

Posamezna zaznavanja pošljejo rezultat s klicem metode `sendBroadcast`. Tej metodi podajo namen – `Intent`. `Intent` je objekt, s katerim lahko pošiljamo sporočila iz enega dela aplikacije v drugega. Podamo mu akcijo in dodamo parameter, ki vsebuje naš rezultat.

```
Intent locationIntent =
    new Intent(DetectorConstants.ACTION_LOCATION_DETECTED);
locationIntent.putExtra(DetectorConstants.EXTRA_LOCATION,
    mAddress);
mMainActivity.sendBroadcast(locationIntent);
```

V zgornjem primeru ustvarimo `Intent`, ki ima akcijo `ACTION_LOCATION_DETECTED`. Z njo našemu sprejemniku sporočimo, da smo zaznali lokacijo. Lokacijo dodamo z metodo `putExtra`, kateri podamo ključ in vrednost. Nato pokličemo `sendBroadcast`, ki razpršeno odda naše sporočilo.

```
DetectionReceiver receiverL = new DetectionReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        switch(intent.getAction()) {
            case DetectorConstants.ACTION_LOCATION_DETECTED:
                String detectedLocation =
                intent.getStringExtra(DetectorConstants
                    .EXTRA_LOCATION);
                viewModel
                    .onLocationDetected(detectedLocation);
                break;
            default:
                break;
        }
    }
};
```

Sprejemnik na drugi strani posluša sporočila. Ko dobi sporočilo, katerega

akcija je `ACTION_LOCATION_DETECTED`, iz sporočila dobimo lokacijo s klicem metode `getStringExtra`, ki ji podamo ključ želenega parametra. Pokličemo metodo `onLocationDetected`, ki ustrezno nastavi atribute objekta `UserActivity`, ki se nanašajo na lokacijo.

8.3 Zaznavanje števila ljudi

Za zaznavanje števila ljudi s pomočjo protokola Bluetooth potrebujemo dve dovoljenji: `BLUETOOTH` in `BLUETOOTH_ADMIN`, ki ju dodamo v manifest naše aplikacije.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.rosa.diplomska">
    <uses-permission
        android:name="android.permission.BLUETOOTH" />
    <uses-permission
        android:name="android.permission.BLUETOOTH_ADMIN" />
```

Zaznavanje oseb opravlja `PeopleDetectorService`. Pridobimo primerek adapterja `BluetoothAdapter`. Ustvarimo sprejemnik `BroadcastReceiver`, ki bo zaznal določene akcije. Te akcije so začetek zaznavanja, konec zaznavanja in zaznavanje posamezne naprave. Ustvarimo objekt `IntentFilter`. Podamo mu vse akcije, ki jih bo zaznal naš sprejemnik. Pokličemo metodo `registerReceiver(sprejemnik,filter)`. Nato pokličemo metodo adapterja `startDiscovery`, ki prične z odkrivanjem naprav. Naš sprejemnik zazna začetek odkrivanja naprav. Število zaznanih oseb nastavimo na nič. Za vsako zaznano napravo povečamo število zaznanih oseb za eno število. Ko naš sprejemnik zazna konec odkrivanja, pošljemo razpršeno oddajanje (broadcast) s številom zaznanih oseb.

8.4 Zaznavanje lokacije

Zaznavanje oseb opravlja razred `LocationDetector`. Za pridobitev lokacije uporabljamo `FusedLocationProviderClient` [6]. Ta nam vrne objekt `Location`, ki vsebuje zemljepisno širino in dolžino o lokaciji našega uporabnika. Iz teh dveh podatkov nato pridobimo naslov lokacije. Za to funkcionalnost uporabljamo pravico `ACCESS_FINE_LOCATION`, ki jo dodamo v manifest naše aplikacije.

Najprej pripravimo zahtevo za lokacijo, ki je objekt `LocationRequest`. Nastavimo interval zaznavanja nove lokacije, najhitrejši interval zaznavanja lokacije in nastavimo prioriteto, ki je lahko:

- `PRIORITY_HIGH_ACCURACY`,
- `PRIORITY_BALANCED_POWER_ACCURACY`,
- `PRIORITY_LOW_POWER`,
- `PRIORITY_NO_POWER`.

`PRIORITY_LOW_POWER` nam poskusi vrniti mesto, v katerem smo. `PRIORITY_BALANCED_POWER` nam poskusi vrniti na ulico natančno zaznavanje lokacije. `PRIORITY_NO_POWER` nam poskusi vrniti najboljšo natančnost lokacije, ki je možna brez dodatne porabe energije baterije. `PRIORITY_HIGH_ACCURACY` nam poskusi vrniti lokacijo z največjo možno natančnostjo. Ker aplikacija uporabi to funkcionalnost le za kratek čas, smo se odločili za `PRIORITY_HIGH_ACCURACY`. Menimo pa, da bi bilo za naše potrebe dovolj tudi zgolj zaznavanje mesta, v katerem je uporabnik.

Ko imamo zahtevo za lokacijo, preverimo, ali imamo ustrezne nastavitve. Za to ustvarimo objekt razreda `SettingsClient` in pokličemo metodo `checkLocationSettings`. Poslušamo za uspeh oziroma neuspeh te metode. Če je metoda uspešna, lahko začnemo s pridobivanjem lokacije. Če je metoda neuspešna, pa preverimo, ali je težava rešljiva. To pomeni, da bomo prosili uporabnika za dovoljenje za spremembo nastavitev. Če dobimo dovoljenje,

ustrezno spremenimo nastavitve in začnemo s pridobivanjem lokacije sicer do zaznavanja lokacije ne pride.

Pridobivanje lokacije začnemo s klicem metode `requestLocationUpdates` objekta `FusedLocationClient`, ki vrne kot rezultat objekt `LocationResult`. Iz rezultata dobimo lokacijo tako, da kličemo metodo `getLastLocation`. Ko dobimo lokacijo, pokličemo metodo `getAddress`, ki nam s pomočjo objekta `Geocoder` na podlagi zemljepisne širine in dolžine vrne naslov naše lokacije. V določenih primerih se zgodi, da je rezultat `getLastLocation` metode `null`. Takrat nastavimo naslov lokacije na prazen String.

8.5 Zaznavanje gibanja

Zaznavanje gibanja opravlja razred `DetectedActivitiesIntentService` in potrebuje pravico `ACTIVITY_RECOGNITION`. `IntentService` se uporablja za daljše izvajanje kode v ozadju na lastni niti. Za implementacijo smo uporabili Googlov primer za zaznavanje aktivnosti uporabnika [18] in ga prilagodili svojim potrebam. Za zaznavanje gibanja uporabimo `ActivityRecognitionClient`. Zaznavanje začnemo s klicem metode `requestActivityUpdate`, ki ji podamo interval zaznavanja, in naš `Intent`, ki vsebuje naš `IntentService`. Rezultat zaznavanja je objekt `ActivityRecognitionResult`, ki ga dobimo s klicem metode `extractResult`. V rezultatu imamo seznam možnih aktivnosti. Vsaka aktivnost hrani tip aktivnosti in vrednost med nič in sto, ki nam pove gotovost, s katero se izvaja določena aktivnost. Imamo metodo `getActivityString`, ki nam na podlagi tipa aktivnosti vrne tekstovni opis aktivnosti. Aktivnosti, ki nas v našem primeru zanimajo, so:

- `IN_VEHICLE`,
- `ON_BICYCLE`,
- `RUNNING`,
- `WALKING`.

Vse ostale vrednosti (`STILL`, `TILTING`, `UNKNOWN`, `ON_FOOT`) se označijo kot druženje. Sprehodimo se čez tabelo možnih aktivnosti, in si zapomnimo tisto z največjo gotovostjo. Velikokrat se je zgodilo, da sta imeli dve aktivnosti enako gotovost, na primer `ON_FOOT` in `WALKING`. Zato se je kdaj zgodilo, da smo dobili kot aktivnost z največjo gotovostjo druženje namesto katero od zgoraj naštetih aktivnosti. To smo rešili tako, da smo dodali v `getActivityString` različne vrednosti samo tistim aktivnostim, ki nas zanimajo. Ostale so označene kot druženje. Zapomnimo si aktivnost z največjo gotovosto, ki ni druženje, in največjo gotovost aktivnosti, ki je druženje. Na koncu preverimo, če je največja gotovost aktivnosti druženje večja od gotovosti drugih aktivnosti. Če to drži, vrnemo aktivnost druženje, sicer vrnemo aktivnost z največjo gotovostjo.

Zavedamo se, da so aktivnosti, ki jih lahko trenutno zaznamo z Googlovim aplikacijskim vmesnikom, omejene, in niso najbolj ustrezne za našo aplikacijo. Naš cilj je bil narediti prototip in pokazati, kako lahko zaznavamo aktivnosti s pomočjo že narejenega modela strojnega učenja. Če bi v prihodnosti želeli tržiti našo aplikacijo, bi morali pridobiti veliko, označeno testno množico. Na njeni podlagi bi zgradili bolj ustrezen model, s katerim bi lahko prepoznali nam bolj pomembne aktivnosti (ali uporabnik pleše, sedi, stoji).

8.6 Zaznavanje glasbe

Zaznavanje glasbe opravlja razred `SongDetectorService`. Posname dvanajstsekundni zvočni posnetek, ki se pošlje strežniku. Strežnik ga analizira in vrne avtorja in naslov pesmi. Tu potrebujemo dovoljenji `RECORD_AUDIO` in `WRITE_EXTERNAL_STORAGE`.

Posnetek se posname z objektom `MediaRecorder` [10]. Povemo mu izvor zvoka, ki je v našem primeru mikrofon, format izhodne datoteke, ki je m4a, pot in ime izhodne datoteke, zvočni kodirnik, ki je AAC, število zvočnih kanalov, ki je 1, število vzorčenja zvočnega signala na sekundo (44100). Nastavimo lahko tudi, koliko bitov opiše kodirane podatke za 1 sekundo zvoka.

Nastavili smo najdaljše trajanje posnetka na dvanajst sekund.

```
mRecorder = new MediaRecorder();
mRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
mRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
mRecorder.setOutputFile(mAudioSavePathInDevice);
mRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
mRecorder.setAudioChannels(1);
mRecorder.setAudioSamplingRate(44100);
mRecorder.setAudioEncodingBitRate(192000);
mRecorder.setMaxDuration(12000);
mRecorder.setOnInfoListener(new MediaRecorder.OnInfoListener() {
    @Override
    public void onInfo(MediaRecorder mr, int what, int extra) {
        if(what == MediaRecorder.
   MEDIA_RECORDER_INFO_MAX_DURATION_REACHED){
            stopRecording();
        }
    }
});
```

S snemanjem pričnemo tako, da pokličemo metodo `start`, ki je metoda našega objekta `MediaRecorder`. Po dvanajstih sekundah ustavimo snemanje. Posneta datoteka se pretvori v tabelo bajtov, ki jo nato zakodiramo v `String`. Ta `String` dodamo kot enega od parametrov, ki jih pošljemo strežniku.

```
public void volleySendAudio() {
File f = new File(mAudioSavePathInDevice);
byte[] bytesArray = fileToBytes(f);

Map<String, String> parameters = new HashMap<>();
parameters.put("fun","detectAudio");
parameters.put("fileType","m4a");
parameters.put("filename",f.getName());
parameters.put("fileContent", Base64.encodeToString(bytesArray,
    Base64.DEFAULT));
JSONObject jp = new JSONObject(parameters);
```

Strežnik pretvori `String` nazaj v bajte in jih shrani v datoteko. Nato pa s pomočjo exec izvede Python skripto dip_detect_recorded_files.py, ki prepozna naš posnetek in nam vrne avtorja in naslov pesmi s pomočjo knjižnjice Dejavu.

8.6.1 Dejavu

Dejavu pretvori zvočni signal v prstni odtis. Ta odtis se potem shrani v bazo in uporabi za prepoznavanje vzorcev glasbe. Za pridobivanje prstnih odtisov vseh pesmi v mapi uporabimo funkcijo `fingeprint_directory`. Če želimo pridobiti odtise zgolj ene datoteke, uporabimo funkcijo `fingerprint_file`.

```
djv = Dejavu(config)
music_dir = "C:\Users\Dom\Desktop\dejavu_music"
djv.fingerprint_directory(music_dir, [".mp3",".flac"])
```

```
djv = Dejavu(config)
music_dir = "C:\Users\Dom\Desktop\dejavu_music\\"
djv.fingerprint_file(music_dir
    +"Mass-Effect-3--I'm-Proud-Of-You.mp3")
```

Prva funkcija sprejme tri argumente. Prvi argument je pot do mape, v kateri imamo datoteke. Drugi argument je končnica datotek, ki jih bomo analizirali. Tretji argument je izbirni in z njim podamo število procesov. Zgoraj je primer klicanja te funkcije. Podali smo mapo, v kateri imamo pesmi in našteli formate datotek (mp3, flac). Tretji argument smo izpustili. Druga funkcija sprejme en argument - ime datoteke, katere odtis želimo.

Za prepoznavanje uporabimo funkcijo `recognize`. Vzorce glasbe lahko Dejavu prejme preko datotek na disku ali pa posluša preko mikrofona na računalniku. Za prvi način prepoznavanja metodi `recognize` podamo `FileRecognizer` in datoteko. Za drugi način metodi podamo `MicrophoneRecognizer` in število sekund, ki povedo, koliko časa bo Dejavu poslušala preko mikrofona.

Na strežniku uporabljamo prvi način. Ko poženemo skripto dip_detect_recorded_file.py, ji podamo ime naše prejete datoteke kot argument. Skripta ustvari objekt `Dejavu`, pokliče njegovo recognize metodo in nam vrne atribut `song_name`, ki vsebuje ime avtorja in naslov pesmi.

```
djv = Dejavu(config)
filename = sys.argv[1]
song = djv.recognize(FileRecognizer,filename)
print song["song_name"]
```

Imeli smo problem s prepoznavanjem naših posnetkov v Dejavu. Sprva smo mislili, da je kvaliteta posnetka preslaba. Knjižnjico smo testirali še s posnetki, posnetimi z naključno izbrano aplikacijo za snemanje. Te je Dejavu lepo prepoznal in prišli smo do spoznanja, da je problem v nastavitvah `MediaRecorderja`. Preizkusili smo nekaj formatov in različnih nastavitev. Z zgoraj omenjenimi nastavitvami je Dejavu uspešno prepoznal pesmi. Z omejitvijo na dvanajst sekund so ustvarjene datoteke velikosti približno 144 KB .

Za poganjanje Python skripte smo se odločili preveriti, ali je možno pognati Python skripto znotraj Android aplikacije. Odkrili smo Jython in Scripting Layer for Android. Prvi omogoča razvoj Android aplikacij v jeziku Python; drugi omogoča izvajanje Python skript na Android napravi. Nismo našli primera, kjer bi znotraj Android aplikacije, ki uporablja Javo, poganjali Python skripte. Zato in ker so bile zvočne datoteke relativno majhne (144,1 KB), smo se odločili, da se bodo zvočne datoteke poslale na strežnik, ki bo pognal Python skripto za prepoznavanje.

8.6.2 Kako deluje Dejavu

Prstni odtisi so lokalne zgoščene vrednosti, ki jih pridobimo iz spektrograma zvoka. Vrhovi se prepoznajo z uporabo Hitre Fourierjeve transformacije.

Lokalni maksimumi se določijo z uporabo visokopasovnega sita in tehnike za iskanje vrhov pri obdelavi slik. Vrhove diskretiziramo. Vrhovi se kombinirajo s časom diskretizacije in vrednostjo ferkvence, da se ustvari edinstven

hash za tisti določeni trenutek v pesmi - to je naš prstni odtis.

Avtor knjižnice Dejavu se hvali s kar veliko točnostjo. Pri posnetkih, dolgih eno sekundo je bila natančnost 60,0 %. Že pri dveh sekundah dobimo 95,6 % natančnost. Od petih sekund naprej je natančnost 100 %.

Natančnost je odvisna tudi od tega, kako imamo nastavljene določene parametre. Več odtisov ko odvzamemo, bolj natančno bo zaznavanje, a bomo zato potrebovali več prostora za hranjenje podatkov. Velja tudi obratno - manj odtisov odvzamemo, manj prostora bomo potrebovali, vendar bo natančnost manjša.

Poglavje 9

Testiranje

V tem poglavju bomo najprej opisali našo testno skupino. Opisali bomo potek testiranja in prikazali rezultate ankete, ki so jo člani testne skupine izpolnili. Omenili bomo tudi funkcionalnosti, ki bi jih testerji želeli dodane v aplikaciji.

Aplikacijo smo dali testirati desetim ljudem. Pet ljudi je bilo žensk, pet moških. Stari so bili med 20 in 40 let. Uporabnikom uporaba telefona in raznih aplikacij ni bila tuja. Po testiranju smo jih prosili za izpolnitev kratke ankete. Anketa vsebuje osem trditev, za katere povemo, koliko se strinjamo z njimi, tako da izberemo število med ena in pet. Ena pomeni se popolnoma ne strinjam, pet pa pomeni popolnoma se strinjam. Trditve, ki jih anketa vsebuje, so:

- navigacija po aplikaciji je bila zame lahka,
- aplikacija je uspešno zaznala pesem,
- aplikacija je uspešno zaznala lokacijo,
- aplikacija je uspešno zaznala število ljudi,
- aplikacija je uspešno zaznala mojo aktivnost,
- aplikacija je delovala brez problemov, npr. zaustavitev,

– aplikacija porabi malo energije baterije,

– aplikacija se mi je zdela hitra.

Na koncu smo dodali še vprašanje, katere dodatne funkcionalnosti bi radi v aplikaciji. Natančnost posamezne funkcionalnosti pri testiranju smo merili tako, da je uporabnik desetkrat uporabil zaznavanje aplikacije. Vsako zaznavanje smo si zabeležili in primerjali s tem, kar je počel uporabnik. Natančnost zaznavanja aktivnosti nam pove, kolikokrat v teh desetih merjenjih je aplikacija pravilno zaznala uporabnikovo aktivnost. Opisali smo še rezultate našega testiranja. Opravili smo 30 zaznavanj in si zapisali njihovo natančnost.

vprašanje	ocena
navigacija po aplikaciji je bila zame lahka	5,0
aplikacija je uspešno zaznala pesem	3,9
aplikacija je uspešno zaznala lokacijo	4,8
aplikacija je uspešno zaznala število ljudi	5,0
aplikacija je uspešno zaznala mojo aktivnost	4,5
aplikacija je delovala brez problemov, npr. zaustavitev	4,9
aplikacija porabi malo energije baterije	5,0
aplikacija se mi je zdela hitra	5,0

Slika 9.1: Rezultati ankete

Z navigacijo uporabniki niso imeli problema. Vsem desetim uporabnikom je bila navigacija enostavna in so se hitro znašli v aplikaciji.

Natančnost zaznavanja, ki so jo dobili uporabniki, je 78 %. Pri naših testnih zaznavanjih smo dobili dobre rezultate - 97 % natančnost. Dejavu je zaznal glasbo tudi, ko se je v ozadju vrtela kakšna oddaja. Težave so bile le, ko je glasnost ozadja skoraj presegla glasnost predvajane glasbe.

Pri lokaciji smo preverjali čim večjo natančnost lokacije. Gledali smo, da sta kraj in ulica pravilno zaznana. Spremljali smo tudi spreminjanje hišnih številk. Natančnost zaznavanja lokacije je bila 96 %. Zaznavanje lokacije smo šteli za pravilno, če smo dobili natančen naslov do hišne številke. Natančnost je odvisna od kakovosti signala in čipa v našem telefonu. Pri našem testiranju je bila natančnost zaznavanja lokacije dobra. Vedno je vrnilo kraj pravilo, a je 40 % testnih primerov vrnilo napačno hišno številko. Menimo, da natančnost na hišno številko v našem primeru ni potrebna. Koncerti se sicer lahko dogajajo na točno določeni lokaciji, a lahko festivali zavzemajo veliko območje, zato bi bilo najbolje omejiti natančnost na kraj.

Natančnost zaznavanja ljudi je bila pri vseh uporabnikih dobra. Tudi pri lastnem testiranju je vedno pravilno prikazalo število naprav, ki so bile vidne preko protokola Bluetooth. Število ljudi v okolici ni vedno enako zaznanemu številu, ker imajo lahko nekateri izklopljen protokol Bluetooth ali pa imajo več naprav, ki so vidne. Nismo pa testirali z več kot petimi povezanimi telefoni naenkrat, ker jih več nismo imeli na voljo.

Natančnost zaznavanja aktivnosti je bila pri uporabnikih 90 %. Pri našem testiranju je v vseh 30 primerih vrnilo pravilno aktivnost. Testirali smo zaznavanje hoje, teka in statične pozicije, ki je označena kot druženje. Aplikacije nismo testirali med vožnjo z motornim vozilom ali s kolesom ker je server lokalen.

Težavo z zaustavitvijo je imela le ena uporabnica. Zaustavitev smo poskusili poustvariti na lastnem telefonu, ampak nam ni uspelo, ker nismo imeli dovolj informacij o tem, kaj je težave pri uporabnici povrzočilo. Pri naših 30 zaznavanjih do zaustavitev ni prišlo.

Vsi uporabniki so se strinjali, da aplikacija ne porabi veliko energije baterije. Žal so aplikacijo uporabljali le kratek čas, zato nimamo informacij o

porabi energije baterije pri daljši uporabi.

Vsi uporabniki so bili zadovoljni s hitrostjo aplikacije. Odgovore do strežnika prejmemo hitro, ker nimamo veliko podatkov na bazi in so posledično poizvedbe zelo hitre. Tudi brezžično omrežje, na katerem se je aplikacija testirala, je imelo odličen signal. Pri lastnem testiranju je bila hitrost aplikacije prav tako dobra.

Uporabniki so si želeli naslednje dodatne funkcionalnosti:

- možnost blokiranja drugih uporabnikov,
- več različnih tem in fontov pisave,
- možnost dodajanja uporabnikov, ki so v bližini,
- možnost določanja prioritet prijateljev,
- možnost dodajanja objav kot priljubljenih,
- možnost dodajanja fotografij,
- možnost pisanja komentarjev,
- možnost klepeta s prijatelji.

Kar trije od vseh uporabnikov, ki so izpolnili anketo, so napisali, da bi želeli imeti možnost dodajanja objav kot priljubljenih. Ker je bila ta funkcionalost največkrat omenjena, bi imela prioriteto pri dodajanju novih funkcionalnosti.

Poglavje 10

Omejitve

V tem poglavju si bomo ogledali omejitve, na katere naletimo pri naši aplikaciji. V 10.1 bomo govorili o tem, kako lahko omejimo velikost baze. V 10.2 bomo govorili o tem, kako bi lahko izboljšali aplikacijo z dodajanjem več funkcionalnosti. V 10.3 bomo govorili o natančnosti aplikacije. V 10.4 bomo govorili o hitrosti aplikacije. V 10.5 bomo govorili o tem, kako bi lahko zmanjšali porabo energije baterije.

Pri naši aplikaciji naletimo tudi na določene omejitve pri podatkovni bazi, interakciji z aplikacijo, njeno natančnostjo, hitrostjo in porabo energije baterije. Da bi bila naša aplikacija čim širše uporabna, bi morali imeti v bazi čim več različnih zvrsti glasbe in čim več različnih izvajalcev. Naša baza bi tako hitro rasla. Opisali bomo, kako bi lahko njeno rast omejili. Interakcijo z uporabnikom bi lahko izboljšali tako, da bi izboljšali trenutne funkcionalnosti, dodali nove, ki si jih uporabniki želijo in jim dali čim večji nadzor nad aplikacijo. Tudi pri natančnosti naše aplikacije smo omejeni. Bolj natančno zaznavanje bi lahko zagotovili z dodajanjem uporabe dodatnih senzorjev ali z izboljšanjem trenutnih zaznavanj.

10.1 Velikost baze

Knjižnjica Dejavu se uporablja pri prepoznavanju glasbe. Analizira naše pesmi in rezultate shrani v bazo. Največja ovira pri aplikaciji je shranjevanje teh analiz v naši bazi. Večjo natančnost analize imamo, več podatkov moramo hraniti. Da bi bila aplikacija zanimiva čim več ljudem, potrebujemo čim več različnih zvrsti glasbe, izvajalcev glasbe. Ker je aplikacija namenjena za uporabo na dogodkih, bi to težavo bi lahko omejili tako, da bi spremljali aktualne dogodke, koncerte, festivale, ki bodo v bližnji prihodnosti in bi imeli ti izvajalci višjo prioriteto pri analiziranju in dodajanju v bazo. Če bi želeli omejiti shranjevanje objav, bi lahko objavam dodali rok trajanja. Primer aplikacije, ki uporablja ta način, je Snapchat - slike in sporočila so na voljo samo omejen čas; po preteku tega časa se izbrišejo.

10.2 Interakcija uporabnika z aplikacijo

Ta del aplikacije bi izboljšali z dodajanjem novih funkcionalnosti, ki si jih uporabniki želijo, in z izboljšanjem že obstoječih. Primer tega bi bilo dodajanje možnosti označevanja objave kot priljubljene, saj je bila to funkcionalnost, ki si jo je naša testna skupina najbolj želela. Kot primer izboljšanja obstoječe funkcionalnosti bi lahko dodali večji nadzor nad tem, katere senzorje uporabljamo. V nastavitve bi lahko dodali možnost izklopa/vklopa posameznega zaznavanja.

10.3 Natančnost aplikacije

Natančnost zaznavanje aplikacije je odvisna od različnih faktorjev, zato je težko zagotoviti visoko natančnost v vsaki situaciji. Odvisna je od natančnosti same aplikacije, knjižnjic, ki jih uporablja, dostopa do storitev, ki jih aplikacija potrebuje, kvalitete senzorjev, ki jih ima ali nima posamezen telefon na voljo, in od kakovosti zvoka na samem dogodku. Zvok je še posebej problematičen, ker je lahko na koncertu veliko šuma (govor, kriki,

petje drugih ljudi). Kvaliteta ozvočenja je lahko slaba. Prav tako lahko vpliva pozicija samega telefona. Če imamo naš telefon na primer na dnu torbice ali pod kupom oblačil, ki bi dušila zvok, bomo imeli posnetek slabše kakovosti. Tu bi lahko poskusili izboljšati zaznavanje zvoka z dodatno obdelavo zvočnega posnetka, ki bi odstranila šume pred analizo. Lahko bi tudi zaznavali samo pozicijo telefona in če je telefon na neprimernem mestu (v torbici, kjer se zvok duši) zaznavanja ne sprožimo. Pozicijo telefona bi lahko zaznavali s senzorjema za bližino in svetlobo.

10.4 Hitrost aplikacije

Uporabniku moramo zagotoviti nemoteno uporabo aplikacije in mu podati čim večji občutek takojšnjega odziva aplikacije. Rešitev za to je, da stvari, ki bi ovirale uporabo aplikacije, izvajamo v ozadju ali na svoji niti, medtem ko uporabnika o samem izvajanju določenih funkcij obvestimo na takšen ali drugačen način. Primer je dodajanje nove objave – uporabniku dodamo objavo na seznam takoj, ko se odloči, da jo bo objavil. Nato pa pošljemo zahtevo za vstavljanje v bazo na strežnik. Uporabnik bo tako imel občutek, da se je dodajanje zgodilo takoj. Če je vnos v bazo neuspešen, o tem obvestimo uporabnika in naknadno izbrišemo objavo s seznama.

Druga možna nadgradnja bi bila menjava HTTP protokola s protokolom Message Queuing Telemetry Transport (MQTT[25]). HTTP uporablja model odjemalec-strežnik, MQTT model objavi/naroči se. Pri prvem odjemalec pošlje zahtevo strežniku, strežnik opravi delo in vrne odjemalcu odgovor. Odjemalec komunicira direktno s končno točko. Pri modelu objavi/naroči poteka komunikacija preko posrednika. Lahko objavljamo ali pa se naročamo na vsebine. Naprava, ki pošilja sporočilo, je tista, ki jo objavi. Naprava, ki sprejema sporočila, je naročnik. Posrednik razvršča vsa sporočila na podlagi njihove vsebine in naročil na to vsebino. MQTT je lahek, odprtokoden, enostaven za implementacijo. Ogledali smo si primerjavo protokola HTTP in MQTT [35, 2]. MQTT se je bolje izkazal. Z njim je bil prenos podatkov

hitrejši. Potreboval je manj prometa in manj energije baterije. Zaključili smo, da bi z uporabo MQTT namesto HTTP pohitrili našo aplikacijo. Uporabnik bi tako hitreje prejemal podatke in imel občutek bolj odzivne aplikacije.

10.5 Poraba energije baterije

Porabo energije baterije bi prav tako lahko zmanjšali z MQTT protokolom, ki porabi manj pasovne šire. Posledično bi z njim zmanjšali porabo energije baterije.

Če bi bila poraba energije baterije problem, bi to lahko omejili z zaznavanjem manjše količine podatkov. Posledično bi bila natančnost zaznavanja manjša. Primer – zaznavanje lokacije bi lahko nastavili na manjšo natančnost; snemanje zvoka bi lahko omejili na manjše časovno obdobje. Lahko bi dodali direktno pošiljanje zvoka na strežnik in bi se tako izognili shranjevanju zvočnega posnetka pred pošiljanjem. Izognili bi se tudi brisanju po pošiljanju. Lahko bi dodali tudi možnost izklopitve funkcionalnosti, ki porabijo več energije baterije. Če je možno, bi lahko dele aplikacije, ki porabijo največ energije baterije, prestavili na strežnik, kar pa bi povečalo komunikacijo med njima.

Poglavje 11

Sklepne ugotovitve

Namen naše diplomske naloge je bil ustvariti aplikacijo, ki bi združevala elemente socialnega omrežja in uporabo senzorjev našega telefona. Menim, da nam je to uspelo. Naša aplikacija je zanimiva, ker vsebuje zaznavanje bolj splošne, vsakodnevne aktivnosti. Tako ciljamo na širšo publiko kot na primer razne aplikacije za telovadbo, ki ciljajo na specifično skupino ljudi: na tiste, ki redno telovadijo.

Od drugih socialnih omrežij se razlikuje po tem, da objave ne piše uporabnik, ampak jih generira sama aplikacija. Menimo, da bi to lahko pritegnilo nove uporabnike, ki bi želeli poskusiti nekaj novega.

Zaznavanje glasbe je hiter in enostaven način, s katerim uporabnik opiše, na katerem dogodku je. Poleg tega bi lahko to funkcionalnost uporabili na festivalih za prepoznavanje nam neznanih izvajalcev in njihovih skladb.

Pri zaznavanju smo dobili dobro natančnost. Zaznavanje pesmi je imelo pri uporabnikih, ki so testirali aplikacijo, 78 % natančnost, naši testni primeri pa 97 % natančnost. Večino nepravilnih zaznavanj pripisujemo hrupu v ozadju ali slabemu zvočnemu posnetku. Zaznavanje lokacije je imelo 96 % natančnost pri uporabnikih. V okviru našega testiranja sta bili ulica in mesto vedno pravilna, a je v 40 % vrnilo napačno hišno številko. Zaznavanje ljudi smo testirali tako, da smo prižgali Bluetooth na določenem številu naprav in preverili, ali se je število teh naprav ujemalo s številom zaznanih naprav v

aplikaciji. Število zaznanih naprav se je vedno ujemalo s številom naprav, ki smo jim vključili Bluetooth. Tudi pri testni skupini je pravilno vedno zaznalo število naprav. Zaznavanje aktivnosti je bilo pri testnih uporabnikih 90 %; pri našem zaznavanju je vedno vrnilo pravilno aktivnost.

Naša aplikacija je sestavljena iz Android aplikacije, ki skrbi za interakcijo z uporabnikom, zaznavanje, bazo, ki hrani vse naše podatke, in strežnik, ki opravlja komunikacijo med njima. Vse te dele bi lahko dodatno izboljšali. Aplikaciji bi lahko dodali več funkcionalnosti. Nameravali smo dodati še možnost dodajanja objav kot priljubljenih, objavljanje objav na drugih socialnih omrežjih, spreminjanje podatkov uporabnika, potrjevanje registracije in e-pošte, možnost pozabljenega gesla, vendar nam je za te implementacije zmanjkalo časa. Sčasoma nameravamo aplikacijo še dodatno izpopolniti kot osebni projekt, dodati vse te funkcionalnosti in popraviti mogoče nepravilnosti oziroma hrošče, ki jih do sedaj nismo odkrili. Lokalni strežnik bi lahko zamenjali s spletnim strežnikom. Tako bi lahko aplikacijo bolje testirali, ker bi lažje dobili več uporabnikov za testiranje.

Aplikacijo bi lahko dopolnili z bolj natančnim zaznavanjem gibanja (sedenje, ležanje, ples). Za to bi lahko uporabili na primer Weko [1], ki je odprtokodna programska oprema za strojno učenje ali pa bi uporabili že obstoječe modele. Google Play Services Activity Recognition bi lahko zamenjali z orodjem PathSenseActivity [27], ki zaznava hitreje, bolj natančno in pri tem porabi manj energije baterije. Trenutno omogoča zaznavanje hoje, vožnje, držanja, mirovanja, tresenja in držanja v prevoznem sredstvu. Načrtujejo povečanje zaznavanja dodatnih aktivnostih oziroma dodatnega gibanja.

Literatura

[1] Machine Learning Group at the University of Waikato. Weka 3 - data mining with open source machine learning software in java. Dosegljivo: https://www.cs.waikato.ac.nz/ml/weka/. [Dostopano: 7.1.2018].

[2] Jan Bartnitsky. Http vs mqtt performance tests. Dosegljivo: https://flespi.com/blog/http-vs-mqtt-performance-tests. [Dostopano: 12.2.2018].

[3] Craig Campbell. Chrome logger - server side application debugging. Dosegljivo: https://craig.is/writing/chrome-logger. [Dostopano: 29.8.2017].

[4] Treating PHP Delusions. (the only proper) pdo tutorial - treating php delusions. Dosegljivo: https://phpdelusions.net/pdo. [Dostopano: 10.9.2017].

[5] Android Developers. Data binding library — android developers. Dosegljivo: https://developer.android.com/topic/libraries/data-binding/index.html. [Dostopano: 2.7.2017].

[6] Android Developers. Getting the last known location — android developers. Dosegljivo: https://developer.android.com/training/location/retrieve-current.html. [Dostopano: 6.3.2018].

[7] Android Developers. Implement a custom request — android developers. Dosegljivo: https://developer.android.com/training/volley/request-custom. [Dostopano: 9.1.2019].

[8] Android Developers. Making a standard request — android developers. Dosegljivo: https://developer.android.com/training/volley/request.html. [Dostopano: 13.9.2017].

[9] Android Developers. Mediarecorder — android developers. Dosegljivo: https://developer.android.com/reference/android/content/Context. [Dostopano: 7.1.2018].

[10] Android Developers. Mediarecorder — android developers. Dosegljivo: https://developer.android.com/reference/android/media/MediaRecorder.html. [Dostopano: 7.1.2018].

[11] Android Developers. Room persistence library — android developers. Dosegljivo: https://developer.android.com/topic/libraries/architecture/room.html. [Dostopano: 20.9.2017].

[12] Android Developers. Save data in a local database using room — android developers. Dosegljivo: https://developer.android.com/training/data-storage/room/index.html. [Dostopano: 20.9.2017].

[13] Android Developers. Services — android developers. Dosegljivo: https://developer.android.com/guide/components/services. [Dostopano: 6.3.2018].

[14] Android Developers. Sharedpreferences — android developers. Dosegljivo: https://developer.android.com/reference/android/content/SharedPreferences.html. [Dostopano: 20.8.2017].

[15] Android Developers. Transmitting network data using volley — android developers. Dosegljivo: https://developer.android.com/training/volley/index.html. [Dostopano: 13.9.2017].

[16] Android Developers. Viewmodel — android developers. Dosegljivo: https://developer.android.com/topic/libraries/architecture/viewmodel.html. [Dostopano: 2.7.2017].

[17] Diffen. Get vs post - difference and comparison — diffen. Dosegljivo: https://www.diffen.com/difference/GET-vs-POST-HTTP-Requests. [Dostopano: 29.8.2017].

[18] Google. android-play-location/activityrecognition at master · googlesamples/android-play-location. Dosegljivo: https://github.com/googlesamples/android-play-location/tree/master/ActivityRecognition. [Dostopano: 11.2.2018].

[19] Jennifer R. Kwapisz, Gary M. Weiss, and Samuel A. Moore. Activity recognition using cell phone accelerometers. *SIGKDD Explor. Newsl.*, 12(2):74–82, March 2011.

[20] 2017 vogella GmbH Lars Vogel, (c) 2015. Android architecture with mvp or mvvm - tutorial. Dosegljivo: http://www.vogella.com/tutorials/AndroidArchitecture/article.html. [Dostopano: 1.7.2017].

[21] 2017 vogella GmbH Lars Vogel (c) 2014. Using data binding in android - tutorial. Dosegljivo: http://www.vogella.com/tutorials/AndroidDatabinding/article.html. [Dostopano: 6.9.2017].

[22] Tian Lou. A comparison of android native app architecture - mvc, mvp and mvvm. Magistersko delo, univerza Aalto, 2006.

[23] Hong Lu, Wei Pan, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. Soundsense: Scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 165–178, New York, NY, USA, 2009. ACM.

[24] Emiliano Miluzzo, Nicholas D. Lane, Kristóf Fodor, Ronald Peterson, Hong Lu, Mirco Musolesi, Shane B. Eisenman, Xiao Zheng, and Andrew T. Campbell. Sensing meets mobile social networks: The design, implementation and evaluation of the cenceme application. In *Procee-*

dings of the 6th ACM Conference on Embedded Network Sensor Systems, SenSys '08, pages 337–350, New York, NY, USA, 2008. ACM.

[25] MQTT. Mqtt. Dosegljivo: `http://mqtt.org/`. [Dostopano: 11.2.2018].

[26] Ángel Torres Moreira. Design and implementation of an android application to anonymously analyse locations of the citizens in barcelona. Magistersko delo, Politehniška univerza Katalonije, 2015.

[27] PathSense. Pathsense. Dosegljivo: `https://pathsense.com/awesomeactivity`. [Dostopano: 19.1.2019].

[28] Rosanda Potrebuješ. Povezava do kode aplikacije. Dosegljivo: `https://github.com/RosaPotrebujes/dip`. [Dostopano: 16.9.2018].

[29] Rosanda Potrebuješ. Povezava do kode strežnika. Dosegljivo: `https://github.com/RosaPotrebujes/server`. [Dostopano: 16.9.2018].

[30] Kiran K. Rachuri, Mirco Musolesi, Cecilia Mascolo, Peter J. Rentfrow, Chris Longworth, and Andrius Aucinas. Emotionsense: A mobile phones based adaptive platform for experimental social psychology research. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing*, UbiComp '10, pages 281–290, New York, NY, USA, 2010. ACM.

[31] tutorialspoint.com. Mvvm introduction. Dosegljivo: `https://www.tutorialspoint.com/mvvm/mvvm_introduction.htm`. [Dostopano: 1.7.2017].

[32] w3schools. Php prepared statements. Dosegljivo: `https://www.w3schools.com/php/php_mysql_prepared_statements.asp`. [Dostopano: 10.9.2017].

[33] WampServer. Wampserver, la plate-forme de développement web sous windows - apache, mysql, php. Dosegljivo: `http://www.wampserver.com/en/`. [Dostopano: 27.8.2017].

[34] worldveil. worldveil/dejavu: Audio fingerprinting and recognition in python. Dosegljivo: `https://github.com/worldveil/dejavu`. [Dostopano: 7.1.2018].

[35] Tetsuya Yokotani and Yuya Sasaki. Comparison with http and mqtt on required network resources for iot. *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCE-REC)*, pages 1–6, 2016.