Usporedna analiza programskih pristupa u razvoju mobilnih aplikacija za Android i iOS

Marić, Petar

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:885760

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-02-28**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

USPOREDNA ANALIZA PROGRAMSKIH PRISTUPA U RAZVOJU MOBILNIH APLIKACIJA ZA ANDROID I IOS

Diplomski rad

Petar Marić

Osijek, 2020.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 12.07.2020.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime studenta:	Petar Marić
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1000R, 19.09.2019.
OIB studenta:	33866030341
Mentor:	Prof.dr.sc. Goran Martinović
Sumentor:	Dino Kurtagić
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Krešimir Nenadić
Član Povjerenstva 1:	Prof.dr.sc. Goran Martinović
Član Povjerenstva 2:	Izv. prof. dr. sc. Alfonzo Baumgartner
Naslov diplomskog rada:	Usporedna analiza programskih pristupa u razvoju mobilnih aplikacija za Android i iOS
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U diplomskom radu treba s gledišta načela programskog inženjerstva, programskih jezika, tehnologija i alata, programskih arhitektura i testiranja, analizirati i objasniti različite pristupe u razvoju mobilnih aplikacija. Posebno treba razmotriti korištenje sljedećih programskih jezika i alata: Java i Kotlin za Android, te Objective C i Swift za iOS. Na projektu mobilne aplikacije za obavješćivanje korisnika o njemu zanimljivim događajima, te praćenje njegove vjernosti i dodjeljivanje popusta koja je izvorno pisana u Javi i u Objective C-u, pokazati postupak razvoja takve aplikacije u Kotlinu za Android i Swiftu za iOS. Na temelju toga, analizirati postupak razvoja, odgovarajuću programsku arhitekturu
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	12.07.2020.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 24.07.2020.

Ime i prezime studenta:	Petar Marić
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1000R, 19.09.2019.
Turnitin podudaranje [%]:	4

Ovom izjavom izjavljujem da je rad pod nazivom**: Usporedna analiza programskih pristupa u razvoju mobilnih aplikacija za Android i iOS**

izrađen pod vodstvom mentora Prof.dr.sc. Goran Martinović

i sumentora Dino Kurtagić

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD 1
1.1 Zadatak diplomskog rada 1
2. PREGLED TRENUTNOG STANJA U PODRUČJU PROGRAMSKIH PRISTUPA ZA RAZVOJ MOBILNIH APLIKACIJA 3
3. OPIS MOBILNIH PLATFORMI I PROGRAMSKIH JEZIKA 5
3.1 Mobilne platforme 5
3.1.1 Android 5
3.1.2 iOS 6
3.2 Programski jezici 7
3.2.1 Java 7
3.2.2 Objective-C 8
3.2.3 Kotlin 10
3.2.4 Swift 13
3.3 Arhitektura MVP 16
3.4 Testiranje programske podrške 18
3.5 Načela SOLID 19
4. RAZVOJ APLIKACIJE CITYALARM 21
4.1 Postman 22
4.2 Prikaz rada poslužitelja 24
4.3 Pregled razvoja mobilne aplikacije koristeći programski jezik Kotlin 28
4.3.1 Korištene biblioteke 28
4.3.2 Biblioteka Card View 29
4.3.3 Biblioteka Material 30
4.3.4 Biblioteka Swipe Refresh Layout 30
4.3.5 Biblioteka Glide 30
4.3.6 Biblioteka Open Street Map 31
4.3.7 Biblioteka Koin 31
4.3.8 Biblioteka Retrofit 31
4.4 Razvoj Android aplikacije 32
4.4.1 Postavljanje i karakteristike Android projekta 32
4.4.2 Opis glavnih komponenti sučelja Android aplikacija 36
4.4.3 Razvoj zaslona Android aplikacije 41
4.5 Pregled razvoja mobilne aplikacije koristeći programski jezik Swift 49
4.5.1 Upravitelji biblioteka 49
4.5.2 Biblioteka Alamofire 53
4.5.3 Biblioteka Swinject 53

4.5.4 Biblioteka Kingfisher 54
4.5.5 Biblioteka Toast-Swift 54
4.5.6 Biblioteka SWRevealController 54
4.6 Razvoj iOS aplikacije 54
4.6.1 Postavljanje i karakteristike iOS projekta 54
4.6.2 Opis glavnih komponenti sučelja iOS aplikacija 57
4.6.3 Razvoj zaslona iOS aplikacije 62
4.7 Testiranje iOS aplikacije 70
5. USPOREDBA PLAFORMI ANDROID I IOS S NAGLASOM NA POSLUŽITELJSKU STRANU RJEŠENJA 72
5.1.1 Opis razvijenog mrežnog sloja na Android platformi 72
5.1.2 Opis razvijenog mrežnog sloja na platformi iOS 76
6. ZAKLJUČAK 81
SAŽETAK 83
ABSTRACT 84
LITERATURA 85
PRILOZI 91
ŽIVOTOPIS 95

1. UVOD

Problematika diplomskog rada jest pregled različitih programskih jezika, alata, razvojne arhitekture, pristupa testiranju te razvoj mobilnih aplikacija. Usporedbom razvoja čitatelja se informira o pojedinostima koje odlikuju mobilne platforme kako bi mogao donijeti ispravnu odluku o razvoju budućih aplikacija za određenu mobilnu platformu.

Stvaranje programskih rješenja za mobilne operacijske sustave Android i iOS obavljeno je korištenjem modernih programskih jezika nastalih u prethodnom desetljeću. Riječ je o programskim jezicima Kotlin i Swift koje odlikuje jednostavnost pisanja programskog koda te sintaksna sličnost. Prije usporedbe razvoja mobilnih aplikacija izložene su putanje, parametri upita te rezultati upita usmjerenih na putanje korištenog poslužitelja. Za prikaz razvoja mrežne arhitekturalne razine korištene su biblioteke Retrofit za Android mobilnu aplikaciju te biblioteka Alamofire za iOS mobilnu aplikaciju uz prikaz programskog koda kojim je ostvarena na svakoj od platformi.

Drugo poglavlje prikazuje trenutno stanje u području razvoja mobilnih aplikacija s gledišta mobilnih platformi, programskih jezika, razvojnih alata te razvojnih arhitektura. Trećim poglavljem izloženi su detalji mobilnih platformi Android i iOS na kojima se pokreću mobilne aplikacije. Prikazana je arhitektura platformi s pojašnjenim elementima koji pružaju potporu pokretanju i radu aplikacija. Također je dan pregled programskih jezika koji se koriste za razvoj mobilnih aplikacija. U nastavku trećeg poglavlja opisana je popularna razvojna arhitektura programske podrške MVP, načela SOLID te glavne karakteristike testiranja programske podrške. U četvrtom poglavlju dan je uvid u korišteni poslužitelj te opis razvoja mobilnih aplikacija za operacijske sustave Android i iOS. Peto poglavlje daje usporedbu korištenih elemenata korisničkih sučelja, hardverske zahtjeve te usporedbu razvoja mrežne arhitekturalne razine kreiranih aplikacija uporabom Retrofit i Alamofire biblioteka.

1.1 Zadatak diplomskog rada

U diplomskom radu treba s gledišta načela programskog inženjerstva, programskih jezika, tehnologija i alata, programskih arhitektura i testiranja, analizirati i objasniti različite pristupe u razvoju mobilnih aplikacija. Posebno treba razmotriti korištenje sljedećih programskih jezika i alata: Java i Kotlin za Android, te Objective C i Swift za iOS. Na projektu mobilne aplikacije za obavješćivanje korisnika o njemu zanimljivim događajima, te praćenje njegove vjernosti i dodjeljivanje popusta koja je izvorno pisana u Javi i u Objective C-u, pokazati postupak razvoja

takve aplikacije u Kotlinu za Android i Swiftu za iOS. Na temelju toga, analizirati postupak razvoja, odgovarajuću programsku arhitekturu, primijenjena načela SOLID i načela pisanja čistog koda, korištenja dodatnih biblioteka, te alata za svaku od platformi, a zatim napraviti usporedbu obje platforme s naglaskom na poslužiteljsku stranu rješenja. Također, obaviti potrebno testiranje programske podrške za iOS aplikaciju.

2. PREGLED TRENUTNOG STANJA U PODRUČJU PROGRAMSKIH PRISTUPA ZA RAZVOJ MOBILNIH APLIKACIJA

Robert C. Martin programski je inženjer s višegodišnjim iskustvom te je autor knjiga u kojima bilježi bitne stavke programerske struke. U [1] i [2] prikazuje se pisanje čitljivog i održivog koda zajedno s načelima SOLID primjenjive na razvoj bilo kojeg oblika programske podrške, a koji su bili primijenjeni u razvoju praktičnog dijela ovoga rada.

Funkcionalno slične aplikacije razvijenim aplikacijama u ovome radu raspoložive na aplikacijskim trgovinama Android i iOS platforme jesu *Eventbrite* [3], *Meetup* [4] te *All Events in City* [5]. Navedene aplikacije nude oblik odabira određene interesne kategorije te okvirne lokacije oko koje se traže aktualni događaji ili okupljanja. Razlike u odnosu na korišten poslužitelj za razvoj vlastitih aplikacija jest u tome da navedene aplikacije sličnu funkcionalnost nude u obliku web stranica. Omogućavaju dijeljenje, pohranu korisniku interesantnih događaja, kontaktiranje emailom te geolokaciju. Aplikacije također nude mogućnost korištenja ograničavanja geografskog područja (engl. *geofencing*) ukoliko korisnik dođe u blizinu nekog događaja koji odgovara njegovim preferencijama.

Za razvoj mobilnih aplikacija na Android i iOS platformama koriste se moderna programska okruženja. Najpoznatiji alat za razvoj Android mobilnih aplikacija jest Android Studio [6] razvijen od tvrtke JetBrains te najpoznatiji alat za razvoj iOS mobilnih aplikacija jest Xcode [7] razvijen od tvrtke Apple. Oba programska okruženja kao osnovu pružaju uređivač teksta te integrirane alate za prevođenje i pokretanje aplikacija na pridruženim emulatorima. Android emulator ima sve pogodnosti pravog mobilnog uređaja dok se iOS emulatorom može testirati samo pokretanje aplikacija i navigacija po korisničkom sučelju, odnosno, nedostaje podrška za testiranje ispravnosti rada razvijene programske podrške za korištenje GPS-a, kamere, jačine signala u slučaju podatkovne veze itd. Neovisno o navedenom, najbolja je uporaba fizičkog mobilnog uređaja prilikom testiranja ispravnosti rada mobilnih aplikacija. Bitno je napomenuti da se okruženja razlikuju u rukovanju ovisnostima. Tako na Android platformi postoji jedinstven alat za rukovanje ovisnostima zvan Gradle dok na iOS platformi postoje centralizirana i necentralizirana rješenja za rukovanje ovisnostima poput Cocoa Pods [8] te Carthage [9] koje dolaze sa svojim pojedinostima uporabe. Prednost Gradle alata upravljanja ovisnosti u usporedbi s alatima dostupnim na iOS platformi jest automatiziranost dohvaćanja i uvrštavanja u projekt. Odgovornost programera jest pronaći putanju tražene ovisnosti te specificiranje inačice ovsinosti koju želi uvrstiti u projekt.

Razvoj aplikacija u počecima Android platforme bio je u Java programskom jeziku dok je na iOS platformi to bio programski jezik Objective-C. Navedene programske jezike iz uporabe u novim razvojnim projektima zamijenili su Kotlin programski jezik na Android platformi te Swift programski jezik na iOS mobilnoj platformi. Oba programska jezika na sličan način definiraju varijable, konstante, funkcije, klase, sučelja te nasljeđivanje. Također, oba programska jezika uklanjaju potrebu za specificiranjem tipova podataka varijabli ili konstanti jer prevoditelji imaju ugrađenu podršku za zaključivanjem o tipu podatka.

Najkorištenije arhitekture primijenjene u razvoju mobilnih aplikacija jesu MVC, MVP te MVVM. Razvojna arhitektura MVC koristi se u slučaju manjih aplikacija u kojima se ne očekuje velika promjena zahtjeva jer u protivnom može doći do nakupljanja koda unutar klase upravljača. Upravljač također direktno upravlja sadržajem pogleda. Arhitektura MVP dolazi kao nadogradnja arhitekture MVC s povećanjem sadržaja jedne klase na povećanje broja klasa te dodavanjem sučelja kako bi se ograničilo saznanje prezentera o razini pogleda. Razvojna arhitektura MVP očekuje promjenu zahtjeva te dodavanje funkcionalnosti pod cijenu povećanja metoda sučelja prezentera te pogleda koje je potrebno implementirati u prikladnim klasama. Razvojna arhitektura MVVM eliminira sučelja između pogleda i modela pogleda uz povećanje složenosti i završnu veličinu aplikacije uporabom potrebnih biblioteka RxKotlin [10] ili RxSwift [11]. Isplativost korištenja MVVM razvojne arhitekture vidljiva je kod većih projekata.

Odabir mobilne platforme treba temeljiti na ciljanoj populaciji na određenoj mobilnoj platformi te tehnološkim mogućnostima izvedbe rješenja u vidu dostupnih biblioteka ili raspoloživim ljudskim resursima za razvoj jedinstvenog rješenja.

3. OPIS MOBILNIH PLATFORMI I PROGRAMSKIH JEZIKA

Poglavlje 3 prikazuje povijest trenutno najzastupljenijih mobilnih platformi, te daje opis svake od platformi zajedno s pojedinostima. Također, dan je uvid u programske jezike koji su se prvotno koristili za razvoj mobilnih aplikacija te u novije programske jezike koji preuzimaju vodstvo u razvoju aplikacija zbog svoje jednostavnosti.

3.1 Mobilne platforme

3.1.1 Android

Operacijski sustav Android na najnižoj razini koristi Linux jezgru za izvršavanje zadataka. Linux jezgra koja se koristi u mobilnim uređajima modificirana je grana Linux jezgre kojom upravlja Linus Torvalds. Inačica jezgre koristi se za pružanje sigurnosti memorijskih prostora te pokretanju aplikacija, upravljanje procesima i memorijom, upravljanje mrežnom komunikacijom te upravljanje višenitnošću [12]. Dodatne nadogradnje jezgre potrebne za mobilne uređaje uključuju upravljanje potrošnjom električne energije, uključivanjem zaslona na određene događaje (engl. *wakelock*) te rukovanje memorijom u slučaju velikog zauzeća memorije. Android okolina za pokretanje aplikacija (engl. *Android Runtime*, ART) koristi navedene karakteristike jezgre za pokretanje aplikacija.

Povrh Linux jezgre nalazi se razina apstrakcije hardvera (engl. *hardware abstraction level,* HAL) čije se programsko sučelje prema gornjim razinama arhitekture ne mijenja. HAL razina sastoji se od nekolicine modula koji predstavljaju određenu funkcionalnost hardvera. Na primjer, postoje moduli zaduženi za upravljanje bežičnom konekcijom, kamerom mobilnog uređaja, senzorima, zvukom itd. Navedeni moduli pisani su C i C++ programskim jezicima. Postojanjem HAL-a omogućuje se proizvođačima mobilnih uređaja kreiranje pogonskih programa neovisno o Android operacijskom sustavu te moraju osigurati implementaciju sučelja koja se pozivaju s HAL razine na zahtjev viših arhitekturalnih razina [13].

ART te nativne C/C++ biblioteke dijele sljedeću arhitekturalnu razinu. Prema [14], ART je zamijenio Dalvik *runtime* okolinu s inačicom Android operacijskog sustava 5.0. Dalvik je za razliku od ART-a koristio prevođenje tijekom izvođenja (engl. *just in time, JIT)* dok ART koristi prevođenje prije pokretanja (engl. *ahead of time, AOT)*. Oba *runtimea* služe kako bi se izvršni dalvik bajtkod (engl. *dalvik executable bytecode, DEX*), dobivenog od prevedenog Java bajtkoda, mogao izvoditi na hardveru mobilnog uređaja. Nedostaci korištenja Dalvika bili su vidljivi u

zastajkivanju aplikacija jer je JIT prevođenje neadekvatno za mobilne uređaje s manjkom obradne snage. Zbog korištenja AOT prevođenja ART-om se eliminira nedostatak Dalvika te se prevođenje odvija prilikom instalacije aplikacije na mobilni uređaj. Dodatna prednost jest u poboljšanom otklanjanju grešaka te preciznijim porukama iznimki. Nativnim C/C++ bibliotekama zaobilazi se Java bajtkod te se dobiva na ubrzanju aplikacija jer se prevode direktno u strojni kod spreman za izvršavanje. Nativnim bibliotekama moguće je rukovati grafikom, zvukom, kamerom te multimedijom.

Na razini iznad nalaze se Java biblioteke koje se koriste za razvoj mobilnih aplikacija. Sadržava biblioteke za kreiranje korisničkih sučelja, biblioteke za rukovanjem obavijestima, telefonskim pozivima, lokaciji, pristupu datotekama.

Vršnom razinom arhitekture predstavljene su korisničke aplikacije instalirane na mobilnom uređaju. Arhitektura Android platforme vidljiva je iz [12].

3.1.2 iOS

Arhitektura iOS operacijskog sustava slična je arhitekturi Android operacijskog sustava. Arhitektura se sastoji od pet razina od kojih viša razina koristi usluge omogućene od niže razine. Na najnižoj razini nalazi se jezgra i pogonski programi potrebni za ispravan rad hardvera. Prema [15], jezgra korištena za iOS operacijski sustav jest XNU *(X is not Unix)* jezgra korištena iz Darwin operacijskog sustava. XNU predstavlja hibridnu jezgru koja je izgrađena od dvije komponente. Koristi komponente Mach jezgre razvijene na Carnegie Mellon sveučilištu te komponente FreeBSD-a. Mach komponenta XNU jezgre zadužena je za osiguravanje sigurnosti memorije, interprocesnu komunikaciju, rukovanje prekidima te višenitnošću [16]. FreeBSD komponenta jezgre zadužena je za izradu modela procesa, dozvolama, sigurnošću, datotečnim sustavom, TCP/IP protokolima te vatrozidom. Razina povrh jezgre jest CoreOS koja koristi sučelje jezgre za izvršavanje zadataka. Sadržava programske okvire za sigurnost, ključeve te kreiranje certifikata. *Accelerate* okvirom [17] ubrzavaju se složeni matematički izračuni kao i manipulacija slika korištenjem optimiziranih algoritama. Također se može koristiti za obradu signala te neuronske mreže. Povrh Core OS razine nalazi se razina usluga jezgre (engl. *Core Services*) koja apstrahira korištenje niže razine. Na ovoj razini nalaze se biblioteke potrebne za rukovanje mrežom, pristupom bazama podataka i datotečnom sustavu, lokaciji uređaja, višenitnosti te mrežnim zahtjevima. Sljedeća razina jest Media razina kojom se omogućuje korištenje audio i video datoteka, reprodukcija zvuka, animacije te grafike. Zadnja razina arhitekture nazvana je Cocoa Touch. Na navedenoj razini nalaze se biblioteke potrebne za razvoj mobilnih aplikacija. Biblioteke

obuhvaćaju kod kojim se generiraju komponente korisničkih sučelja, reagira na korisnikove radnje, omogućava prikaz mapa itd. Prikaz arhitekture iOS platforme vidljiv je iz [18].

3.2 Programski jezici

3.2.1 Java

Java programski jezik prvu inačicu dobio je 1996. godine, a razvila ga je američka tvrtka Sun Microsystems [19]. James Gosling bio je glavni razvojni inženjer zadužen za razvoj programskog jezika. Karakteristika Java programskog jezika jest u platformskoj neovisnosti. Tako je programe pisane u Javi moguće pokretati na Windows, Linux, macOS te Solaris operacijskim sustavima. Pokretanje na operacijskim sustavima omogućeno je koristeći JVM koji nije platformski neovisan, a prevodi Java *bytecode* u strojni jezik koji se zatim izvršava na određenom procesoru. Prema [20]:

- Osnovni tipovi podataka unutar Java programskog jezika su cijeli brojevi *int,* brojevi s pomičnim zarezom *float,* brojevi s pomičnim zarezom dvostruke preciznosti *double,* logička vrijednost *boolean* te znakovni *char*. Podržane su također operacije nad bitovima. Podržani relacijski operatori su: jednakost ==; različitost !=; manje i manje ili jednako <, <=; veće i veće ili jednako >, >=. Podržani logički operatori jesu disjunkcija ||, konjunkcija && te negacija !. Od aritmetičkih operatora podržani su +, -, *, / te %. Za kreiranje varijable potrebno je eksplicitno navođenje tipa ispred naziva varijable. Deklaraciju i dodjeljivanje vrijednosti moguće je skraćeno napisati jednim izrazom poput „*int number = 47;“* te je vidljivo da se cjelobrojna vrijednost 47 pridružuje varijabli *number*. Također je potrebno svaki izraz zaključiti interpunkcijskim znakom ';'. Deklaracija polja elemenata ostvaruje se dodavanjem uglatih zagrada nakon tipa podataka koje će polje sadržavati, a prije njegova naziva. Primjerice, deklaracija i kreiranje polja glasi: „*double[] doubleArray = new double [] {2.712, 3.14}*“. Prethodnim isječkom kreira se polje koje pohranjuje *double* tip podatka te se pridružuju vrijednosti varijabli polja.
- Kontrola toka programa odvija se koristeći *if, for, while, do-while* izraze koji određuju granu izvršavanja provjerom uvjeta. Primjer grananja dan je u nastavku: „*if (number > 3) System.out.println(number);*“. Ukoliko prethodna deklarirana varijabla *number* ne sadržava vrijednost veću od broja tri, na zaslon se ne bi ispisala vrijednost pohranjena unutar varijable.
- Klasu se unutar Java programskog jezika definira ključnom riječi *class*. Konstruktorom je moguće definirati vrijednosti varijabli budućeg objekta. Konstruktor mora imati isti naziv kao klasa te mora imati definiran tip i naziv parametara koji se zatim dodjeljuju varijablama. Unutar Jave moguće je naslijediti najviše jednu dodatnu klasu i proizvoljno mnogo sučelja. Klasama je

također moguće definirati metode kojima se manipulira varijablama. Podržano je definiranje apstraktnih klasa koje sadržavaju deklaracije metoda nužnih za implementaciju kao i varijabli u deriviranim tipovima podataka. Ukoliko klasa nasljeđuje drugu klasu, nakon imena klase potrebno je navesti ključnu riječ *extends* te naziv klase koju se nasljeđuje.

- Sučeljima je moguće ostvariti višestruko nasljeđivanje definiranjem konstantni te deklaracija funkcija. Sučelja se definiraju ključnom riječi *interface* te je potrebno specificirati naziv sučelja. U slučaju da sučelje nasljeđuje drugo sučelje, potrebno je navesti ključnu riječ *extends* te naziv sučelja kojeg se nasljeđuje. Prilikom adaptacije sučelja u klasi, potrebno je navesti ključnu riječ *implements* u deklaraciji klase te naziv sučelja kojeg klasa implementira.
- Funkcije unutar Java programskog jezika definiraju se navodeći tip podatka koji se vraća iz funkcije, naziv funkcije te listu parametara unutar zagrada. Listu parametara definira se slično definiranju varijabli bez dodjeljivanja vrijednosti. Funkcije unutar Jave ne podržavaju definiranje podrazumijevanih parametara što je karakteristika mnogih novijih programskih jezika. Moguće je definirati funkcije koje nisu vezane za određeni objekt, nego na klasu, a navedeno se ostvaruje ključnom riječi *static* ispred povratnog tipa funkcije.

3.2.2 Objective-C

Prema [21], Objective-C programski jezik direktan je nadskup programskog jezika C. Svaki program pisan C programskim jezikom može se napisati unutar Objective-C programskog jezika bez sintaksnih promjena. Objektno orijentirane karakteristike uvedene su iz Small Talk programskog jezika. Objective-C inicijalno je razvijan od tvrtke NeXT, sve dok Apple nije preuzeo navedenu tvrtku. Koristio se za razvoj aplikacijskih okvira i aplikacija za iOS i macOS uređaje. Glavnu ulogu u razvoju aplikacija imao je do pojave novog programskog jezika Swift koji zadržava kompatibilnost s Objective-C programskim jezikom te uvodi značajne promjene u načinu pisanja programskog koda. Za korištenje Objective-C programskog jezika potrebno je definirati datoteku koja sadržava deklaracije metoda, poznatiju kao datoteku zaglavlja, te datoteku u kojoj su implementirane metode, takozvanu implementacijsku datoteku. Prema [22], karakteristike Objective-C su:

- Od programskog jezika C nasljeđuje primitivne *int, float, double* te *char* tipove podataka. Objektne verzije navedenih tipova pohranjuju se unutar *NSNumber, NSString* te *NSValue* objekata. Kolekcije elemenata unutar Objective-C programskog jezika potrebno je definirati koristeći *NSArray*, *NSSet*, *NSDictionary* klase. U navedenim kolekcijama moguća je pohrana

isključivo objekata iz Objective-C programskog jezika. Ključne riječi za kontrolu toka programa preuzete su iz C programskog jezika te glase *if, switch, for, while, do-while*.

- Za definiranje klasa potrebno je definirati sučelje, API, kojim je potrebno opisati varijable klase (svojstva, engl. *property*) te metode koje se implementiraju danom klasom. Pomoću navedenog sučelja definira se popis *poruka* na koje klasa može reagirati, tj. definiraju se deklaracije metoda. U klasi se zatim moraju dati implementacije metoda kako bi se mogla izvršiti nekakva obrada primitkom poruke. Klasa koju svaka druga klasa mora naslijediti jest NSObject u kojoj se definiraju zajedničke karakteristike interakcije dvaju objekata. Definicija sučelja klase definira se ključnom riječi *@interface* iza koje slijedi naziv klase te dvotočkom odvojeno nasljeđivanje klase NSObject. Definiciju sučelja potrebno je zatvoriti ključnom riječi *@end*. Validna definicija sučelja izgledala bi: „*@interface Car : NSObject @end*“. Za definiranje svojstava unutar klase potrebno je navesti ključnu riječ *@property* prije definiranja tipa podatka. Funkcije na instanci određene klase, odnosno metode, definiraju se sa značajnim izmjenama u usporedbi s C funkcijama. Tako je unutar javnog sučelja *Car* klase moguće navesti „*- (void)drive:(int)distance;*“ ako se klasi želi omogućiti pozivanje *drive* funkcije iz druge klase. Nakon definiranja metode unutar sučelja, potrebno je dati implementaciju metode unutar klase *Car* koja glasi: „*- (void)drive:(int)distance { NSLog(@"Driving for: %d km.", distance); }*“. Zatim je unutar koda potrebno stvoriti objekt (engl. *instantiate*) što se odrađuje s „*Car *car = [[Car alloc] init];*“, te poziv implementirane funkcije „*[car drive:1000];*“. Metode nad tipom određene klase moguće je kreirati unutar sučelja s prefiksom ‘+’ te specificiranjem deklaracije funkcije koju je zatim potrebno implementirati. Funkcije koje nisu vezane niti za jednu klasu definiraju se i pozivaju ekvivalentno C funkcijama.
- Objective-C uveo je pojam kategorije koji se u novijim programskim jezicima zove proširenje. Kategorijama se omogućava dodavanje neke funkcionalnosti u postojeće klase koje također mogu biti iz biblioteka. Kategoriju se definira ekvivalentno definiranju sučelja klase izuzev liste nasljeđivanja te specificiranjem naziva kategorije unutar zagrada. Nakon definiranja kategorije, nužno je dati implementaciju.
- Protokole koje određene klase moraju implementirati moguće je definirati ključnom riječi *@protocol* nakon koje je potrebno definirati naziv protokola. Unutar tijela protokola zatim se navode svojstva i metode koje određena klasa treba implementirati. Definicija protokola završava ključnom riječi *@end*. Implementaciju protokola potrebno je odraditi unutar sučelja te implementacije klase. Unutar sučelja klase potrebno je navesti samo oznaku prihvaćanja (konformaciju) na protokol, tj. nije potrebno navoditi iste deklaracije metoda navedenih protokolom. Deklariranje prihvaćanja određenog protokola navodi se nakon liste nasljeđivanja

unutar sučelja klase. Nazivi protokola navode se unutar „< >“ te se odvajaju zarezom ukoliko je riječ o višestrukom nasljeđivanju protokola.

3.2.3 Kotlin

Programski jezik Kotlin inačicu 1.0 dobio je 2016. godine [23]. Razvijen je s ciljem razvoja platformski neovisnog koda koji je siguran te jednostavan za uporabu. Razvoj je započela ruska tvrtka JetBrains koja je cijeli programski jezik napravila otvorenim kodom dostupnim na [24]. Trenutno podržava razvoj poslužiteljske strane, kreiranje web stranica, analizu podataka, *Kotlin/Native*, razvoj Android te međuplatformski (engl. *cross-platform*) razvoj mobilnih aplikacija [25]. Za razvoj poslužiteljske strane postoji nekoliko razvojnih okvira od kojih su najpoznatiji *Ktor*, *Micronaut*, *Javalin*. Kreiranje web stranica omogućava se prijevodom Kotlin koda u ECMAScript 5.1 standard za JavaScript. Korištenje Kotlina za razvoj Android mobilnih aplikacija rezultira prevođenje Kotlin koda u Java *bytecode* kojeg razumije JVM. *Kotlin/Native* omogućava razvoj nativnih aplikacija za operacijske sustave koji za pokretanje aplikacija ne koriste *runtime* okolinu ili virtualni stroj što je slučaj kod Android platforme. *Kotlin/Native* koristi LLVM prevoditelj koji je zadužen za kreiranje izvršnih datoteka za određene računalne platforme. Podržani operacijski sustavi za razvoj su Linux, Windows, Android, iOS, macOS [26]. Prema [27] vrijedi:

- Nad bilo kojim ugrađenim tipom podatka unutar Kotlin programskog jezika moguće je pozivanje metoda i svojstava. Osnovni podržani podaci su brojevi, znakovi, *boolean* vrijednosti, polja te nizovi znakova. Osnovni tipovi brojeva su *Byte*, *Short*, *Int*, *Long*, *Float* te *Double*. Prilikom pisanja većih brojeva moguće je koristiti ‘_‘ kao separator znamenki bez mijenjanja vrijednosti broja. Podržane su također operacije nad bitovima. Za definiranje raspona brojeva koristi se '..' operator. Znakovi predstavljaju oznaku koja također može sadržavati i izlaznu sekvencu (engl. *escape sequence*) poznate iz C programskog jezika. Kreiranje polja ostvaruje se pozivom funkcije *arrayOf* te predavanjem nekog od podržanih tipova ili čak vlastito kreiranih tipova te će Kotlin prevoditelj prepoznati da je riječ o polju određenog tipa. Tipovi elemenata polja ne smiju biti različiti od onih korištenih prilikom deklaracije polja. Ostale podržane kolekcije jesu skupovi i imenici (engl. *dictionary*). Znakovne nizove definira se unutar para navodnika te je moguće koristiti interpolaciju znakovnih nizova korištenjem ${} gdje se unutar vitičastih zagrada specificira varijabla, poziv neke funkcije koja vraća vrijednost ili vraća vrijednost koristeći *if* ili *when* izraze.

- Kontrola toka programa modificira se korištenjem *if, when, for, while* te *do-while* ključnih riječi. *If* i *when* ključne riječi su izrazi, odnosno omogućeno je vraćanje vrijednosti iz određenih grana kako bi se vratila vrijednost u varijablu ili dobio izlaz iz funkcije. *For* petljom moguće je iterirati preko liste elemenata ili preko raspona brojeva. Primjer *for* petlje *for (number in 1..6) print(number).*
- Definiranje klasa izvršava se korištenjem ključne riječi *class* te definiranja naziva klase. Klase imaju jedan primarni konstruktor kojeg se može definirati ključnom riječi *constructor* te je moguće definirati više sekundarnih konstruktora. Za definiranje primarnog konstruktora nije potrebno specificiranje ključne riječi ukoliko se ne mijenja vidljivost konstruktora te je moguće definirati svojstva unutar para zagrada nakon naziva klase bez korištenja *klasičnog* konstruktora gdje se svakoj varijabli mora dodijeliti vrijednost već se automatski predanim parametrima mogu postaviti svojstva klase. Korištenjem sekundarnih konstruktora mora se osigurati delegacija na primarni konstruktor kako bi inicijalizacija objekta bila uspješna. Instance, odnosno objekte kreira se definiranjem varijable te poziva konstruktora nad određenim tipom s predanim parametrima. Tako bi inicijalizacija proizvoljne klase bila *val rectangle = Rectangle()*. Nasljeđivanje određene klase specificira se dodavanjem dvotočke nakon definicije klase te specificiranjem bazne klase i ako je potrebno, prosljeđivanje određenih parametara unutar zagrada. Podržavaju se i apstraktne klase kojima se osigurava zajednička implementacija u deriviranim tipovima. Za definiranje konstanti ili funkcija nad tipom, potrebno je koristiti *companion object* konstrukt. U slučaju da se treba kreirati klasa koja samo pohranjuje vrijednosti, moguće je dodati ključnu riječ *data* ispred definicije klase. Tada je klasi omogućeno samo definiranje svojstava primarnim konstruktorom.
- Svojstva unutar Kotlin programskog jezika mogu biti definirana kao promjenjiva (engl. *mutable*) te one kojima je moguće samo dobiti pohranjenu vrijednost (engl. *read-only*). Promjenjiva svojstva definiraju se koristeći ključnu riječ *var*, dok se za definiranje *read-only* svojstva koristi ključna riječ *val*. Za specificiranje svojstva nije potrebno specificirati tip podatka jer će prevoditelj moći zaključiti tip iz vrijednosti koja se pridružuje svojstvu. Moguće je također definirati vlastite metode dohvaćanja (engl. *get*) i postavljanja (engl. *set*) nad svojstvom. Ključnom riječju *lateinit* moguće je definirati promjenjivo svojstvo kojemu se vrijednost dodjeljuje negdje drugdje, izvan konstruktora.
- Sučelja unutar Kotlin programskog jezika služe za definiranje apstraktnih metoda, podrazumijevanih implementacija metoda te apstraktnih svojstava. Sučelje se definira ključnom riječi *interface* nakon koje je potrebno definirati naziv sučelja. Sadržaj sučelja piše se unutar

vitičastih zagrada. Klasama nije ograničen broj sučelja koje mogu naslijediti te sučelja mogu nasljeđivati druga sučelja.

- Proširenjima se definiraju nove metode i svojstva nad postojećim tipovima bez potrebe za njihovim nasljeđivanjem te dodavanjem specifičnosti. Primjerice, proširenje za definiranje metode nad tipom *Customer* koja ispisuje dostavnu adresu kupca glasilo bi *fun Customer.printAddress() { print(this.address) }*. Ključnom riječi *this* pristupa se objektu zvanog *receiver object* tipa *Customer* nad kojim se poziva metoda *printAddress* te se pristupa svojstvu *address* unutar *Customer* objekta. Metode proširenja moguće je definirati na *nullable* tipovima, tj. tipovima koji mogu pohranjivati vrijednost *null*. Kako bi se izbjegle greške unutar koda koji poziva proširenje na *nullable* tipovima podataka, unutar proširenja omogućava se provjera sadržava li *receiver object null* vrijednost, te ako sadržava moguće je vratiti podrazumijevanu ne *null* vrijednost kojom se neće prekinuti program podizanjem iznimke.
- Enumeracije unutar Kotlina označavaju se ključnim riječima *enum class* te nazivom enumeracije. Moguće je definirati zajednički tip podatka svakom slučaju enumeracije te je moguće definirati sučelje s metodama koje zatim svaki slučaj enumeracije mora implementirati.
- Funkcije se definiraju nešto drugačije od tradicionalnih programskih jezika. Prvo se definira funkcija ključnom riječi *fun*, a zatim se definira naziv, parametri te povratni tip funkcije. Tako bi primjer jednostavne funkcije bez parametara i povratnog tipa izgledao „*fun printGreeting() { print(„Hello“) }*“. U slučaju da je potrebno vratiti vrijednost iz funkcije, potrebno je napisati „*fun square(number: Double): Double { return number*number}*“. Prilikom definiranja parametara funkcije, potrebno je eksplicitno navesti tip parametra. Mogućnost prilikom korištenja funkcija jest također u definiranju podrazumijevanih vrijednosti parametara (engl. *default parameter values*) kojima se omogućava poziv funkcije bez prosljeđivanja parametra ukoliko nije potrebno promijeniti podrazumijevanu vrijednost. U slučaju da postoji kombinacija podrazumijevanih i obveznih parametara, podrazumijevane parametre treba pomaknuti na kraj definicije funkcije. U slučaju većeg broja parametara funkcije, moguće je prilikom poziva funkcije imenovati parametre funkcije te proslijediti vrijednost za odabrani parametar. Imenovane parametre moguće je koristiti različitim redoslijedom od onog definiranog u deklaraciji funkcije. Ako se tijelo funkcije unutar vitičastih zagrada sastoji samo od *return* izraza, moguće je cijelo tijelo funkcije skraćeno napisati pomoću znaka jednakosti i izraza koji se nalazio u *return* izrazu.
- Podržava kreiranje zamjenskih naziva tipova podataka (engl. *type alias*), odnosno drugačijih naziva za tipove podataka korištenih unutar aplikacije. Tako se primjerice može specificirati

zamjenski naziv tipa podatka koji će predstavljati složeni funkcijski tip jednostavnim nazivom od jedne riječi. Zamjenski naziv tipa kreira se korištenjem ključne riječi *typealias*. Tako se primjerice zamjenski naziv tipa podatka za funkcijski tip „*(Int) -> Unit*“ definira s „*typealias CustomName = (Int) -> Unit*“.

3.2.4 Swift

Začetnik Swift programskog jezika jest Chris Lattner, inženjer razvojnih alata unutar tvrtke Apple [28]. Izvorni kod dostupan je na [29]. Swift programski jezik za razvoj aplikacija za macOS, watchOS te tvOS može se koristiti od 2015. godine. Za pokretanje Swift programskog koda na hardveru koristi se LLVM prevoditelj. Obilježja programskog jezika Swift prema [30] su:

- Osnovni podaci koji se mogu koristiti unutar Swift programskog jezika su brojevi, *boolean* vrijednosti te znakovni nizovi. Brojevi se reprezentiraju pomoću cijelih brojeva *Int, Float, Double* tipova podataka, *boolean* pomoću *Bool*, a znakovni nizovi pomoću *String*. Podržane kolekcije jesu polja, skupovi te imenici koji mogu pohranjivati kolekciju elemenata istog tipa. Swift uvodi i novi tip podatka koji predstavlja uređenu n-torku zvanu *tuple*. Tipove podataka nije potrebno specificirati dokle god prevoditelj može zaključiti o tipu podatka koji se dodjeljuje određenoj varijabli. Kreiranje konstanti ili varijabli izvršava se korištenjem *let* ili *var* ključnih riječi nakon kojih se specificira naziv konstante ili varijable. Tako će se primjerice „*let name = „John“*“ *name* konstanti dodijeliti znakovni niz *John* te će prevoditelj zaključiti da je konstanta *name* tipa *String*. U slučaju da se definirala varijabla, a prevoditelj prepozna da se postavljanje vrijednosti izvršava samo jedanput, obavijestit će programera da promijeni varijablu u konstantu. Interpolaciju znakovnih nizova moguće je ostvariti unutar Swift programskog jezika korištenjem izlaznog (engl. *escape*) operatora '\' te zagrada. Unutar zagrada se zatim definira varijabla ili konstanta čija se vrijednost želi uvrstiti u znakovni niz. Kao i Kotlin, podržava odvajanje znamenki većih brojeva koristeći _ radi poboljšanja čitljivosti. Uređenom n-torkom moguće je vratiti više srodnih podataka u jednom te se može koristiti prilikom vraćanja vrijednosti iz funkcije. Kreiranje polja odvija se specificiranjem istovrsnih podataka unutar uglatih zagrada. Primjerice “*let integerArray = [1,2,3,4,5]*“ prevoditelj će prepoznati da je riječ o polju cijelih brojeva te će zaključiti tip [Int]. Aritmetički, logički i relacijski operatori identični su do sada navedenim programskim jezicima.
- Kontrola toka programa odvija se koristeći *if, switch, for-in, while, repeat while* ključne riječi. Navedeni operatori razlikuju se od klasičnih operatora istog ili sličnih naziva po tome što se uvjeti ne pišu unutar zagrada nego se od ključnih riječi odvajaju razmakom. Za korištenje petlji

moguće je definiranje numeričkih iteratora koristeći „…“ za zatvoreni interval te “..<” za polu otvoreni interval. Primjer *for* petlje na zatvorenom intervalu te pridruživanje trenutne iteracije varijabli, dan je u nastavku: „*for iteration in 1…5 { print(“Trenutna iteracija \(iteration)”) }*“. Navedenom for petljom pet puta će se ispisati vrijednost iteracije koristeći interpolaciju znakovnog niza. Unutar *switch* izraza moguće je dodjeljivanje pojedine vrijednosti u lokalnu varijablu koja se koristi u određenom slučaju. Izrazito je korisno kada se rukuje enumeracijom koja sadržava pridružene vrijednosti (engl. *associated value*).

- Swift podržava kreiranje instanci struktura i klasa. Glavna razlika između navedenih tipova podataka jest što su klase referentni tip (engl. *reference type*) podatka, dok su strukture vrijednosni tip (engl. *value type*) podatka. Navedeno znači da se dodjeljivanjem instance klase jednoj varijabli te pridruživanjem te varijable drugoj varijabli omogućuje pristup istom objektu, tj. referiraju na istu memorijsku lokaciju. Ako bi se navedeno napravilo na vrijednosnom tipu podatka, u drugu varijablu bila bi pohranjena kopija prve varijable te promjenom bilo koje od varijabli ne bi imalo utjecaj na vrijednost druge varijable. Klase i strukture omogućavaju definiranje svojstava, metoda, konstruktora, implementaciju protokola te njihovo proširivanje. Klasama je moguće naslijediti najviše jednu klasu te je moguće imati više referenci na isti objekt prilikom rada programa. Strukturu se definira koristeći ključnu riječ *struct*, a klasu ključnom riječi *class* nakon kojih je potrebno specificirati njihove nazive. Ukoliko se svim svojstvima unutar strukture dodijeli vrijednost, prevoditelj ne zahtijeva eksplicitno kreiranje konstruktora pomoću ključne riječi *init*. U protivnom je potrebno određeni parametar proslijediti prilikom kreiranja instance strukture. Za kreiranje instance klase također je potrebno je koristiti konstruktor zvan glavni konstruktor (engl. *designated constructor*), a omogućava se kreiranje pomoćnih konstruktora (engl. *convenience constructor*) koji moraju u sebi pozvati glavni konstruktor. Pomoćnim konstruktorima potrebno je ispred *init* ključne riječi dodati *convenience* ključnu riječ. Instance klase ili strukture kreira se bez ključnih riječi navođenjem imena klase ili strukture te prosljeđivanja potrebnih parametara u konstruktor. Bitno je napomenuti da su vrijednosni tipovi podataka uz strukture također svi osnovni tipovi podataka unutar Swift programskog jezika te enumeracije.
- Svojstva se unutar Swift programskog jezika dijele na dvije kategorije – one koje pohranjuju vrijednost, pohranjivo svojstvo (engl. *stored property*); te one koji vrše obradu te vraćaju vrijednost, obradno svojstvo (engl. *computed property*). Unutar struktura i klasa moguće je postojanje oba tipa svojstava¸ dok su enumeracijama dostupna samo obradna svojstva. Pohranjivim svojstvima moguće je dodati promatrače promjene vrijednosti (engl. *observere*) koji se pozivaju nakon dodjeljivanja vrijednosti te je moguće započeti nekakvu radnju unutar

programa. Pohranjiva svojstva mogu biti pohranjena unutar varijabli ili konstanti, dok obradna svojstva moraju biti definirana nad varijablama. Također je moguće definiranje svojstava nad tipom podatka.

- Sučelja, odnosno protokole, definira se ključnom riječi *protocol* te specificiranjem njegova naziva. Protokolima se definiraju deklaracije metoda, svojstva te mogućih konstruktora koje određeni tip treba implementirati. Protokole mogu implementirati klase, strukture te enumeracije, a protokolima je omogućeno nasljeđivanje drugih protokola kako bi se omogućile složenije kompozicije. Implementaciju određenih protokola moguće je limitirati na klase ključnom riječi *class* u listi nasljeđivanja protokola.
- Proširenjima se unutar Swift programskog jezika omogućava definiranje obradnih svojstava, metoda, konstruktora te implementacija sučelja. Proširenje se definira koristeći *extension* ključnu riječ te specificiranjem tipa kojeg se proširuje. Nakon naziva proširenja moguće je definirati protokole čija će se implementacija dati unutar tijela proširenja.
- Enumeracijama se omogućava definiranje vrijednosti koje će svaki slučaj enumeracije sadržavati u sebi te se tu vrijednost naziva osnovna vrijednost (engl. *raw value*). Osnovna vrijednost može se definirati kao cijeli broj ili kao znakovni niz. U slučaju definiranja osnovne vrijednosti kao cijelog broja, prvi slučaj ima vrijednost 0, drugi 1 itd. Moguće je prvom slučaju postaviti drugačiju vrijednost te će drugi imati uvećanu vrijednost prvog slučaja. U slučaju definiranja znakovnim nizom, pridružena vrijednost bit će naziv samog slučaja unutar enumeracije. Enumeracijama je moguće definirati pridružene vrijednosti prilikom kreiranja instance pojedinog slučaja. Za ostvarenje navedenog, potrebno je unutar definicije enumeracije pored pojedinog slučaja navesti tipove podataka koje je potrebno proslijediti prilikom inicijalizacije slučaja. Unutar *switch* konstrukta je zatim moguće vršiti pretragu po određenom slučaju te dobiti pridružene vrijednosti korištene u slučaju.
- Funkcije se definiraju ključnom riječi *func* te specificiranjem naziva nakon kojeg slijedi popis parametara i povratni tip funkcije. Razlika od prethodnih programskih jezika jest što unutar Swifta postoje različiti nazivi parametara prilikom poziva funkcije, te naziva parametara koji se koristi unutar funkcije za određene kalkulacije. Prosljeđivanjem vrijednosti u funkciju definiraju se argumenti koji moraju odgovarati parametrima koje funkcija koristi. Time se kreira poseban API kojim je omogućeno definiranje adekvatnijih naziva što će biti demonstrirano u nastavku. Deklaracija proizvoljne funkcije glasi: „*func printCapitalized(valueOf value: String) -> Void*“, a za API navedene funkcije piše se *printCapitalized(valueOf:)*. Vidljivo je da je naziv argumenta definiran nazivom *valueOf* te da se unutar funkcije koristi naziv parametra *value* čime se s mjesta poziva omogućava sintaksa

slična govornom jeziku. Može se vidjeti da se povratni tip funkcije definira nakon „->". Moguće je izostaviti argument funkcije te se može napisati '_' te se prilikom poziva funkcije ne navodi naziv argumenta. Naziv parametra mora biti prisutan u svakoj definiciji funkcije. Ukoliko je unutar tijela funkcije prisutan samo jedan izraz, moguće je povratnu vrijednost funkcije vratiti bez ključne riječi *return*. Parametrima funkcije moguće je dodjeljivati podrazumijevane vrijednosti unutar definicije te ih je moguće izostaviti prilikom poziva.

- Zamjenski nazivi tipova kreiraju se identično kako je pokazano za Kotlin programski jezik.

3.3 Arhitektura MVP

Prema [31], arhitektura MVP oblikovanja aplikacija jedna je od niza arhitektura koje se mogu koristiti za razvoj. Arhitektura treba služiti kao niz uputa pomoću kojih se oblikuje aplikacija te poslovna logika. Arhitektura je također platformski i jezično neovisna, odnosno, primjenjiva je na razvoj bilo koje programske podrške – od konzolnih aplikacija pa do računarstva u oblaku. Praćenjem arhitekturalnih uputa osigurava se bolje razdvajanje odgovornosti komponenti, olakšava testiranje te održavanje aplikacije. Komponentama se kreira konceptualna cjelina aplikacije s jednom odgovornošću. Tako je primjerice moguće postojanje odvojenih komponenti za prijavu, registraciju te rukovanjem korisnikovog profila. Razdvajanjem aplikacije na komponente može se osigurati paralelan razvoj nekih dijelova aplikacije što u konačnici može dovesti do skraćivanja vremena razvoja projekta ili povećanje vremena koje je moguće provesti na testiranje aplikacije. Također se time omogućava dodjela pojedine komponente manjem timu ili pojedinačnim programerima koji zatim mogu razvijati komponentu neovisno o drugim komponentama. Dobrim odabirom arhitekture povećava se broj komponenti koje je moguće ponovno iskoristiti na drugim projektima, osigurava se lakše održavanje koda, pronalazak grešaka, izmjena ili dodavanje novih funkcionalnosti te olakšava testiranje razvijenog rješenja.

Tri glavne cjeline na kojima se zasniva arhitektura MVP su model (engl. *model*), pogled (engl. *view*), te prezenter (engl. *presenter*) [32]. Cilj je razdvojiti odgovornosti pojedine razine na samostalne cjeline ne radeći pri tome čvrsto vezanje između razina, tj. razine ovise o apstrakcijama. Apstrakcije je moguće ostvariti korištenjem sučelja ili protokola određenih programskih jezika, kako je pojašnjeno sljedećim poglavljem. Tako je za svaku razinu potrebno definirati skup sučelja te implementirati navedena sučelja kako bi ostvarila arhitektura MVP.

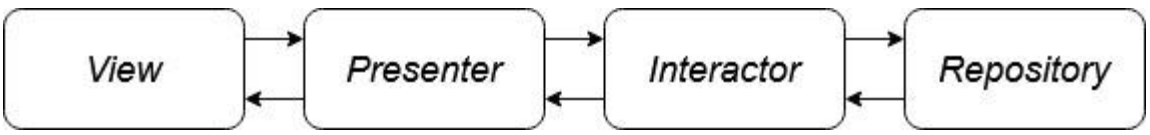
U razinu pogleda unutar arhitekture MVP pripadaju one klase koje imaju direktan utjecaj na korisničko sučelje. U radu razinu pogleda predstavljaju klase aktivnosti i fragmenata na Android platformi te upravljač pogleda (engl. view *controller*) na iOS platformi. Navedena razina također

ne bi trebala znati o dohvaćanju podataka s interneta te pretvaranju i oblikovanju objekata nužnih za rad programskog rješenja. Zadaća jest reagiranje i obavještavanje prezentera o korisnikovim zahtjevima te primati naredbe od prezentera, tj. zadaća je prosljeđivanje i prikazivanje poruka. Unutar sučelja zaduženog za apstrakciju navedene razine potrebno je navesti deklaracije metoda kojima će prezenter obavještavati klasu korisničkog sučelja.

Prezenter predstavlja razinu u kojoj se vrše sve značajnije odluke vezane za određeno programsko rješenje. Odlučuje se o reakciji na korisnikove zahtjeve dobivene s razine pogleda, dohvaćanju podataka s razine modela te obavještavanju pogleda. Sučelje prezentera treba sadržavati deklaracije metoda koje će biti moguće pozivati s razine pogleda, a koje će biti implementirane u klasi prezentera. Klasa zadužena za implementaciju sučelja prezentera ne smije imati ovisnosti klasa iz biblioteka ili okvira zaduženih za generiranje korisničkog sučelja te ta odgovornost mora u potpunosti biti ostvarena od razine pogleda. Prezenter će poruke razini pogleda slati putem mehanizma povratnog poziva (engl. *callback*) gdje će prezenter znati samo o onim metodama navedenim unutar sučelja pogleda, tj. neće se ostvariti čvrsto vezivanje razina. Programskim putem moguće je kreirati instancu klase koja nasljeđuje sučelje pogleda kako bi se omogućilo testiranje funkcionalnosti prezentera.

Razina modela arhitekture MVP uobičajeno se razdvaja na dvije dodatne razine. Riječ je o razini slučajeva korištenja te razini repozitorija. Razina slučajeva korištenja zadužena je za apstrahiranje potrebnih radnji i manipulaciju podataka prije nego se prezenteru daju na korištenje. Sučelja slučajeva korištenja sastoje se od deklaracije funkcija koje koristi razina prezentera. Razina repozitorija zadužena je za dohvaćanje potrebnih podataka. Tako se ovom razinom može apstrahirati dohvaćanje potrebnih objekata putem mrežnog pristupa, pristupa iz baze podataka ili datotečnog sustava. Sučelje ove razine može imati mnogo deklaracija metoda koje koristi sloj slučajeva korištenja. Cilj razine repozitorija je pružanje stalnog API-ja prema slučajevima korištenjima što omogućava lake izmjene samog načina dohvaćanja podataka bez promjene slučajeva korištenja i ponovnim prevođenjem slučajeva korištenja, odnosno prezentera.

Skica dane arhitekture dana je slikom 3.1.



Slika 3.1 Prikaz razina arhitekture MVP

Kao glavni nedostatak ističe se količina potrebnog koda da se dobije vidljiv rezultat rada. Prednosti korištenja arhitekture MVP jesu brza reakcija na promjene zahtjeva te odvajanje odgovornosti.

3.4 Testiranje programske podrške

Razvojem programske podrške pojavljuje se potreba za poboljšanjem kvalitete. Poboljšanje kvalitete može doći u obliku programa koji je pouzdaniji, brži te sigurniji. Jedan od načina kojim se poboljšava kvaliteta programa jest testiranje. Testiranje programa dolazi u raznim oblicima u ovisnosti u kojoj fazi razvoja se nalazi projekt. Tako je primjerice moguće definirati specifikacije testova, potrebne ulaze i izlaze programa u fazi analize zahtjeva za programskom podrškom. Dvije glavne razine testiranja programske podrške predstavljaju funkcionalno i nefunkcionalno testiranje [33]. U funkcionalno testiranje pripada testiranje jedinica koda (engl. *unit testing*), integracijsko testiranje (engl. *integration testing*), testiranje sustava (engl. *system testing*) te prihvatno testiranje (engl. *acceptance testing*). Nefunkcionalno testiranje obuhvaća testiranje performansi, sigurnosti i uporabljivosti programske podrške. Testiranjem jedinica koda testira se manji skup funkcionalnosti uvedenih u programsko rješenje. Uobičajeno obuhvaća testiranje rezultata poziva funkcija te inicijalizacije objekata. Testove jedinica koda mogu pisati programeri i testeri. Integracijskim testiranjem testira se programsko sučelje između različitih modula unutar programskog rješenja. Testira se ispravnost poslanih podataka u drugi modul te ispravnost rezultata. Poželjno je testirati cjelokupno sučelje između modula. Sistemsko testiranje predstavlja veći napor u testiranju jer se testira cjelokupni sustav. Prihvatno testiranje je završni proces u kojem se vrši završna provjera programskog rješenja te se analizira ispunjenje korisničkih zahtjeva. Nakon prolaska prihvatnog testiranja, programsko rješenje se može smatrati spremnim za isporuku. Pristupi testiranju mogu biti pristup bijele kutije (engl. *white box*), sive kutije (engl. *gray box*) te crne kutije (engl. *black box*). Testiranjem bijelom kutijom tester ima uvid u cijeli izvorni kod programa kojeg testira te pomoću tih informacija modelira podatke i testove. Testiranjem sivom kutijom programeru se omogućuje ograničen pogled na programski kod te se daje uvid u baze podataka. Testiranjem crnom kutijom testiranjem tester nema uvid u programski kod, nego se cijeli sustav predstavlja crnom kutijom. Tester tada može samo dobro formatirati ulazne podatke i dobiveni izlaz usporediti s očekivanim ili zahtijevanim izlazom. U slučaju podudaranja izlaza, testni slučaj smatra se uspješno položenim. Nefunkcionalnim testiranjem testiraju se karakteristike programa koje ne proizlaze direktno iz korisnikovih zahtjeva. Unatoč tome, ovakav oblik testiranja ne bi trebao ostati neiskorišten. Testiranjem performansi pokušava se dobiti uvid u ponašanje programskog rješenja u nekoliko situacija različitih opterećenja. Tako se na primjer web usluga može testirati na određeni broj korisnika. Navedenim testom potrebno je utvrditi normalan rad usluga prema specifikaciji zahtjeva. Dodatno postoji i testiranje izdržljivosti kojim se sustav izlaže većem broju korisnika od predviđenog kroz veći period kako bi se dobio

uvid u ponašanje sustava. Postoji i test nagle pojave korisničkih zahtjeva kako bi se otkrilo ponašanje programskog rješenja na naglu pojavu zahtjeva i njihovo nestajanje. Testom sigurnosti provjeravaju se zahtjevi programske podrške, a uobičajeno se provode testovi integriteta, povjerljivosti te autentifikacije. Testiranjem uporabljivosti pokušava se dobiti uvid u probleme korisnikove interakcije s kreiranim programskim rješenjem. U slučaju detekcije problema, moguće je izvršavati A/B testove korisničkih sučelja kako bi se dobile informacije o korištenju novog korisničkog sučelja. Test uporabljivosti uobičajeno se koristi prilikom razvoja web stranica, mobilnih aplikacija te aplikacija vezanih za određenu računalnu platformu.

Unutar određenih tvrtki testeri se trebaju pridržavati određenih pravila testiranja. Primjerice od testera se može zahtijevati izrada dokumentacije testiranja. Navedenu dokumentaciju moguće je početi razvijati nakon definiranja zahtjeva na programsku podršku. Moguće je definirati opis testova za validaciju korisničkih zahtjeva te izrada skice potrebnih testova. Također se testerima nalaže kreiranje detaljne dokumentacije u kojoj su sadržani testovi te rezultati testiranja. Unutar takve dokumentacije potrebno je definirati naziv testa, područje koda koje se testira, datum i vrijeme testiranja, ulazne podatke, koraci postavljanja okoline za moguće repliciranje testa, očekivani izlaz te izlazni podaci. Ukoliko izvršavanje testnog slučaja nije uspješno, uz test je potrebno dodati opis greške te ozbiljnost kvara kako bi se otklanjanje grešaka moglo prioritizirati. Menadžment, glavni tester te osoba za razvoj moraju neprestano komunicirati o načinu razvoja proizvoda. Menadžment i glavni tester tako moraju nadzirati i kontrolirati proces testiranja programske podrške, pratiti promjene programskog koda te dokumentirati proces kako bi se pristup testiranju mogao poboljšati na drugom projektu, a u konačnici razviti model koji se primjenjuje u cijeloj tvrtki.

3.5 Načela SOLID

Načela razvoja SOLID predstavljaju način na koji bi se trebao organizirati programski kod unutar projekta neovisno o platformi i programskom jeziku. Načela su prepoznale različite osobe, no Robert C. Martin odlučio ih je predstaviti pod nazivom SOLID te je predstavio pet osnovnih načela pisanja programske podrške. Principi redom glase princip jedne odgovornosti (engl. *single responsibility principle*, SRP), princip otvorenosti-zatvorenosti (engl. *open-closed principle*, OCP), liskovljev princip zamjene (engl. *liskov substitution principle*, LSP), princip razdvajanja sučelja (engl. *interface segregation principle*, ISP) te princip zamjene orijentacije ovisnosti (engl. *dependency inversion principle*, DIP). Prema [2], prihvaćanjem principa ostvaruje s toleriranje promjena unutar projekta, kod postaje jednostavniji za čitanje i razumijevanje te se omogućuje

prenosivost razvijenih komponenti u druge projekte. SR princip nalaže da svojstva i metode klasa trebaju imati jedinstveni izvor promjena. Točnije, u klasi ne trebaju postojati metode i svojstva koja ispunjavaju zahtjeve više od jednog aktera jer postoji mogućnost da prvi akter traži promjene koda koje bi rezultirale greškom za drugog aktera. Stoga je rješenje odvajanje funkcionalnosti u odvojene cjeline koje ispunjavaju ciljeve točno jednog aktera. OC principom osigurava se proširivanje postojećih funkcionalnosti programa uz minimalne modifikacije postojećeg izvornog koda programa. Primjenom principa izbjegavaju se skupe promjene na projektu ukoliko se trebaju implementirati dodatni zahtjevi. Također, ako dodavanje novog zahtjeva traži izmjenu cijelog sustava, taj sustav nije adekvatno razvijen, odnosno, opire se promjeni. Ostvaruje se primjenom SR i DI principa. LS princip nalaže da program mora dati jednake rezultate kada se na nekom mjestu unutar koda bazni tip podatka zamijeni deriviranim tipom podatka. Navedeni princip također se iskorištava prilikom testiranja razvijenog programa gdje se može kreirati nova instanca klase koja nasljeđuje određeno sučelje te provjerava ispravnost programa. IS princip specificira da se unutar programa ne bi trebalo kreirati bazno sučelje ili bazna klasa koju zatim implementiraju derivirani tipovi podatka. IS princip služi za izbjegavanje kreiranja sučelja i baznih klasa koje imaju mnoštvo definiranih odgovornosti. Štoviše, takvim pristupom mnogo deriviranih tipova moralo bi imati implementacije metoda koje uopće ne koriste za rad te bi se promjenom sučelja morali mijenjati svi derivirani tipovi. Kao rješenje naglašava se kreiranje manjih sučelja koja se zatim implementiraju na točno određenim mjestima. Kao zadnji princip preostaje DI princip. DI principom nastoji se vlastitu programsku podršku osigurati od čestih promjena implementacija određenih klasa. Stoga se za kreiranje funkcionalnih dijelova koda koriste apstrakcije poput sučelja ili apstraktnih klasa jer onda dio koda koji koristi navedene apstrakcije ne ovisi o njihovoj konkretnoj implementaciji.

4. RAZVOJ APLIKACIJE CITYALARM

CityAlarm projekt kombinacija je poslužiteljske (engl. *server, backend*) i klijentske strane koje međusobno komuniciraju internetom kako bi se korisnicima omogućio pregled ponuda. Za potrebe izrade ovog diplomskog rada koristilo se gotovo poslužiteljsko rješenje pisano u PHP skriptnom jeziku jer je fokus ovoga rada na razvoju mobilnih aplikacija za Android i iOS platformu. Poslužitelj se udomljava unutar PAAS Microsoft Azure okoline oblaka kojom se omogućava korištenje računalnih resursa. Oblak računala omogućuje dinamičnu skalabilnost u ovisnosti o opterećenju što uvelike može smanjiti troškove u situacijama kada bi se plaćalo mnogo zahtjevnije resurse, a da zapravo nisu adekvatno iskorišteni. Također je u takvom obliku računarstva omogućeno definiranje vlastitih podataka koji će u ovome slučaju predstavljati informacije o korisnicima, ponudama, kategorijama, državama i lokacijama.

CityAlarm predstavlja sustav koji omogućava gospodarskim subjektima reklamiranje, odnosno kreiranje ponuda određenog sadržaja. Aktivnosti koje se mogu oglašavati uključuju koncerte, promocije, kulturna zbivanja, razna okupljanja ljudi, popuste na razne ulaznice ili proizvode. Kreiranje ponuda na poslužitelju omogućeno je putem web stranice kojoj pristup imaju administrator i registrirani promotori koje treba ručno autorizirati na adekvatnu razinu ovlasti. Unutar sučelja omogućen je pregled ukupnog broja korisnika, promotora, ponuda i kategorija. U jednostavnoj tablici prikazan je popis postavljenih ponuda s njihovim nazivom, promotorom, lokacijom, kategorijom ponude te datumom stvaranja. Ponude je moguće izmjenjivati nakon njihove kreacije, te se takvo obnovljeno stanje može odmah dobiti kao rezultat poziva na određenu krajnju točku (engl. *endpoint*) web usluga.

Prilikom kreiranja nove ponude potrebno je odabrati korisnika koji je zatražio kreiranje, tip ponude, naziv ponude te kratki opis. Potrebno je također definirati adresu, a po potrebi i riječima specificirati gdje se nalazi objekt u kojem se nudi određena ponuda u slučaju da jedna adresa pokriva veliku kvadraturu poput Zagrebačkog velesajma ili većeg trgovačkog centra. Radi omogućavanja korištenja različitih implementacija mapa u mobilnim aplikacijama, u ponudi je također poželjno specificirati geografske koordinate kako bi se korisnicima poboljšala pristupačnost smanjenjem potrebnog vremena za potpuno informiranje o ponudi. Svakoj ponudi definira se slika, poruka za dijeljenje te poveznice na web stranice ponude, Facebook, Instagram i YouTube poveznice kao i kontakt broj kako bi korisnici lakše mogli dijeliti sadržaj. Ponudi se specificira cijena, valuta, datumi početka i završetka promocije kao i datumi početka i završetka

samog događaja. Naposljetku je nužno izvršiti odabir države, grada i kategorije za određenu ponudu.

Za unošenje nove države u sustav, lokacije i kategorije postupak je jednostavniji od dodavanja ponude jer se uobičajeno te stavke konfiguriraju prilikom pokretanja web usluga. Za unos nove države potrebno je specificirati naziv države te skraćeni dvoznačni ISO zapis. Lokaciju se specificira odabirom države, dodjeljivanjem naziva i specificiranjem geografskih koordinata. Novu kategoriju specificira se riječima. Odgovornost za dodjeljivanje jedinstvenih identifikatora dodijeljena je sustavu prilikom kreiranja bilo koje od stavki.

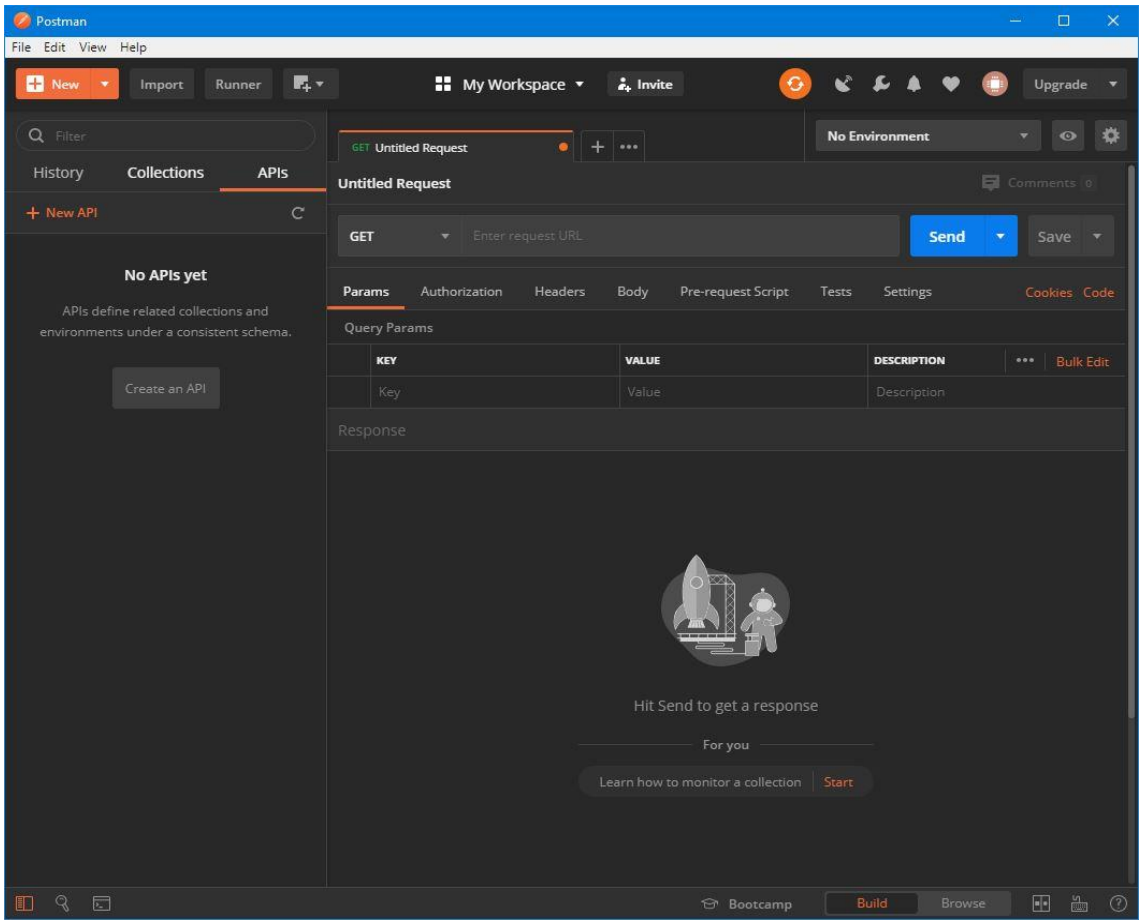
Prije prikazivanja usporedbi razvoja programskim jezicima Kotlin i Swift prikazane su pojedinosti alata *Postman* koji se koristi za testiranje krajnjih točaka web usluga kako bi se moglo upoznati s poslužiteljskom stranom rješenja. Također pokazane su karakteristike nekih krajnjih točaka poslužitelja kojima se vrši dohvaćanje podataka za rad aplikacija.

4.1 Postman

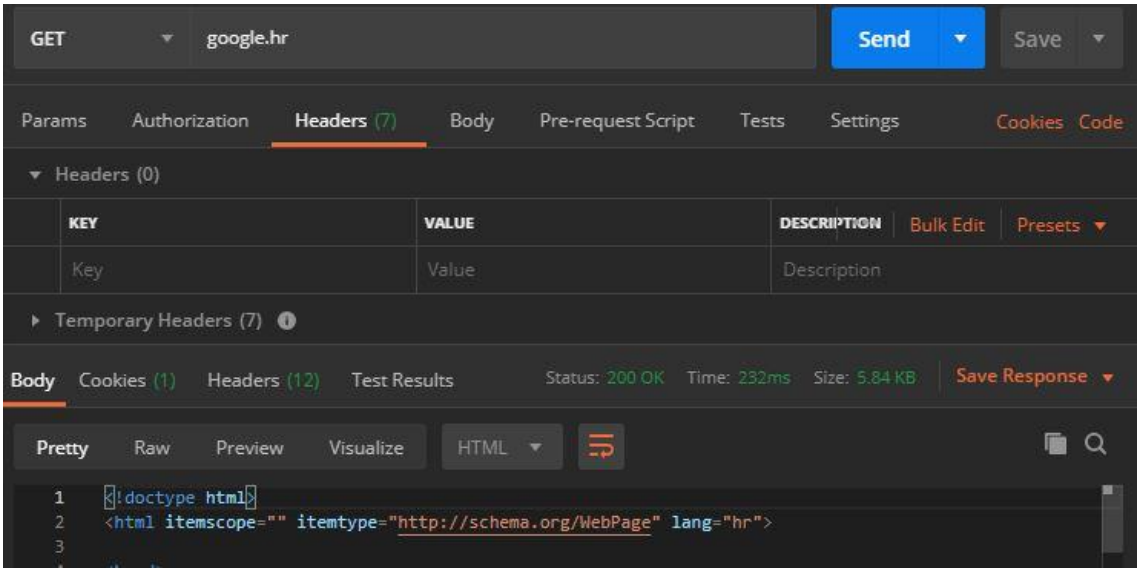
Postman je alat za testiranje aplikacijskog programskog sučelja web usluga raspoloživo za Linux, Windows i MacOS operacijske sustave. Prema [34], Abhinav Asthana 2012. godine objavio je Postman proširenje za internet preglednik Google Chrome.

Na slici 4.2 prikazano je sučelje alata Postman. Klikom na narančastu tipku *New* otvara se prozor u kojem se može kreirati novi zahtjev, kolekcija zahtjeva, nova okolina, API dokumentacija ili lažni poslužitelj (engl. *mock server*). Odabirom novog zahtjeva dodjeljuje se naziv, opcionalni kratki opis zahtjeva te kolekcija u koju će se dodati zahtjev. Kratki opis podržava upotrebu posebnog *Markdown* jezika označavanja za uređivanje teksta [35]. Novom zahtjevu potrebno je dodijeliti kolekciju bilo odabirom postojeće ili kreiranjem nove kolekcije opcijom u otvorenom prozoru. Pohranom zahtjeva fokus se premješta na glavni prozor gdje se otvara kartica s nazivom prethodno definiranog zahtjeva. Trenutno je moguće unijeti nekakav URL u adresnu traku te poslati zahtjev. Na slici 4.1 prikazan je poziv GET metodom na *google.hr* te dio odgovora poslužitelja. Ostale podržane metode su POST, PUT, DELETE, PATCH.

Nakon poslanog zahtjeva moguće je vidjeti status odgovora poslužitelja, potrebno vrijeme za dohvat odgovora te veličinu zahtjeva. U slučaju da se odgovor želi pohraniti, potrebno je odabrati *Save Response* te dodijeliti ime pohranjenom zahtjevu. Dodavanje komentara na određeni zahtjev također je moguće odabirom *Comments* labele vidljive na slici 4.2 . Postman također podržava specificiranje parametara zahtjeva kao i dodatnih zaglavlja. Ako je potrebno na više zahtjeva



Slika 4.2 Prikaz sučelja alata Postman



Slika 4.1 Prikaz sučelja nakon slanja zahtjeva

koristiti isti skup zaglavlja, omogućeno je kreiranje predloška koji se može odabrati odabirom na gumb *Presets* vidljiv na slici 4.1.

U slučaju potrebe slanja podataka poslužitelju za POST ili PUT zahtjeve, sadržaj tijela zahtjeva može se specificirati odlaskom na karticu *Body*. Ondje se definira tip podatka te se u predstavljen prostor mogu upisati podaci koje se žele slati poslužitelju. Ukoliko korisnik ispuni registracijski obrazac te se prijavi u Postman, omogućava mu se kreiranje radnih prostora u koje može spremati zahtjeve, kolekcije, te varijable koje koristi u zahtjevima. Odabirom naziva *My Workspace* pri vrhu slike 4.2 , korisnik može kreirati osobni ili timski radni prostor. Odabirom bilo koje opcije ima mogućnost dodavanja suradnika putem emaila. Također, korisnik će imati pristup istim radnim prostorima i kolekcijama na bilo kojem računalu koje ima pristup internetu.

Za uvid u sveobuhvatne mogućnosti Postman alata, potrebno je istražiti službenu dokumentaciju dostupnu na [36].

4.2 Prikaz rada poslužitelja

Ovim potpoglavljem prikazan je korišteni poslužitelj na okolini oblaka računala Azure kroz nekoliko zahtjeva. Potrebno je pokazati odgovore poslužitelja za određene krajnje točke jer se time najbolje opravdavaju kreirani modeli objekata u daljnjim poglavljima. Odgovori poslužitelja izloženi su u obliku tablice. Svaka tablica sadržava relativnu putanju do krajnje točke, potrebne parametre upita, no izostavlja zaglavlja i sadržaje zaglavlja koji se koriste za autorizaciju. Sva komunikacija s poslužiteljem odvija se putem HTTPS protokola koji vrši enkripciju zaglavlja i tijela zahtjeva. Bazni URL, na kojeg se nadovezuju relativne putanje zahtjeva poslužitelja glasi: *https://cityalarm.azurewebsites.net/*. Svi odgovori poslužitelja vraćaju se u JSON transportnom formatu. Prvi bitan odgovor poslužitelja sadržava popis ponuda. Za svrhu izrade ovoga rada kreiran je testni skup ponuda za razvoj mobilne aplikacije. Prvi upit prikazan je tablicom 4.1. Iz tablice 4.1 može se vidjeti da svaka ponuda sadržava mnoštvo potrebnih stavki. Većina vrijednosti za određene ključeve JSON reprezentacije ponude može biti *null* što predstavlja odsutnost vrijednosti. To znači da se prilikom specificiranja nove ponude na poslužitelju mogu birati željene informacije koje će ponuda sadržavati. U mobilnim aplikacijama se stoga mora odlučiti o iscrtavanju i ispunjavaju sadržaja ukoliko on postoji u odgovoru poslužitelja. Osnovne informacije koje svaka nova ponuda mora sadržavati jesu naziv, kategorija i lokacija. Za svaku ponudu poželjno je imati sliku kako bi se na atraktivan način zadobila korisnikova pozornost te kako bi korisnik odabrao ponudu te pogledao detalje ponude. Također se time povećavaju izgledi da će korisnik podijeliti odabranu ponudu s krugom svojih prijatelja te da će oni učiniti isto.

Tablica 4.1 Prikaz odgovora poslužitelja na krajnju točku ponuda

Relativna putanja	**admin/api/offers**
Metoda	GET

Odgovor poslužitelja	[
	{
	"id": 1,
	"user": null,
	"type": "general",
	"location": "Zagreb",
	"categories": "Zabava",
	"share_url": "https://cityalarm.azurewebsites.net//ponuda/prva-promocija",
	"name": "Prva promocija",
	"safe_name": "prva-promocija",
	"promoter": "Kristijan",
	"description": "Opis",
	"place": null,
	"address": "Zagreb",
	"longitude": 45,
	"latitude": 16,
	"image": "https://i.ytimg.com/vi/2UbscgZ9hpw/maxresdefault.jpg",
	"price": 100,
	"discount": 20,
	"url": "http://google.hr",
	"facebook": "https://www.facebook.com/",
	"twitter": "https://twitter.com/",
	"instagram": "https://www.instagram.com/",
	"youtube": "https://www.youtube.com/",
	"google_plus": null,
	"email": null,
	"date_start": "2019-08-26 13:41:41",
	"date_end": "2020-07-31 13:40:00",
	"date_event_start": null,
	"date_event_end": null,
	"contact_phone": "0911234567",
	"currency": "HRK",
	"social_message": "Dođi"
	}
	]

Pristupačnost ponude također se povećava korištenjem poveznica na različite društvene mreže gdje korisnici mogu vidjeti aktualnu objavu, prethodne objave i recenzije. Poslužiteljem je također omogućeno specificiranje geografske širine i dužine kako bi se unutar aplikacija za detalje ponuda mogla prikazati adekvatna karta s lokacijom ponude. U ovisnosti o prirodi ponude, postoji mogućnost specificiranja cijene i valute.

Tablicom 4.2 prikazan je odgovor poslužitelja u slučaju kada je potrebno dohvatiti listu kategorija raspoloživih na poslužitelju. Riječ je o vrlo jednostavnom JSON formatu u kojem su najvažniji podaci jedinstveni identifikator te naziv kategorije. Na poslužitelju je prilikom kreiranja kategorije nužno specificirati samo naziv kategorije, a za ostalu funkcionalnost brine se upravljač. Ova

krajnja točka koristi se u postupku registracije, a nužno ju je pokazati radi ažuriranja korisničkih kategorija.

Tablica 4.2 Prikaz odgovora poslužitelja na krajnju točku kategorija

Relativna putanja	**admin/api/categories**
Metoda	GET
Odgovor poslužitelja	[{ "id": 1, "name": "Zabava", "safe_name": "zabava" "active": true }, { "id": 2, "name": "Glazba", "safe_name": "glazba" "active": true }, …]

Krajnjom točkom u tablici 4.3 korisniku se omogućava odabir njemu zanimljivih kategorija. Za primjer se poslužitelju kao jedina odabrana kategorija šalje kategorija s jedinstvenim identifikatorom 1. Nakon obrade, poslužitelj kao odgovor vraća stanje trenutno odabranih kategorija kako bi se novo stanje moglo reprezentirati u aplikacijama.

Tablica 4.3 Prikaz odgovora poslužitelja na krajnju točku ažuriranja kategorija

Relativna putanja	**admin/api/users/update-categories?id=<id>**
Metoda	POST
Parametar zahtjeva	Id korisnika

Sadržaj zahtjeva	{
	"categories": [1]
	}

Odgovor poslužitelja	[{ "id": 1, "name": "Zabava", "safe_name": "zabava", "active": true, "selected": true }, { "id": 2, "name": "Glazba", "safe_name": "glazba", "active": true, "selected": false }, ...]

Tablicom 4.4 prikazan je odgovor poslužitelja koji sadržava trenutne gradove pohranjene na poslužitelju koje korisnik može odabrati tijekom postupka registracije. Također svaki zapis grada sastoji se od jedinstvenog identifikatora te nazivom grada.

Tablica 4.4 Prikaz odgovora poslužitelja na krajnju točku gradova

Relativna putanja	**admin/api/locations**
Metoda	GET
Odgovor poslužitelja	[{ "id": 1, "name": "Osijek", "safe_name": "osijek", "country": "HR", "active": true }, { "id": 2, "name": "Donji Miholjac", "safe_name": "donji-miholjac", "country": "HR", "active": true }, ...]

Iz tablice 4.5 vidljivo je da se ažurira trenutno odabrana lokacija korisnika na onu lokaciju koja je poslana unutar polja zahtjeva.

Tablica 4.5 Prikaz odgovora poslužitelja na krajnju točku ažuriranja gradova

Relativna putanja	**api/users/update-locations?id=<id>**
Metoda	POST
Parametar zahtjeva	Id korisnika

Sadržaj zahtjeva	{
	"locations": [2]
	}

Odgovor poslužitelja	[{ "id": 1, "name": "Osijek", "safe_name": "osijek", "country": "HR", "active": true, "selected": false }, { "id": 2, "name": "Donji Miholjac", "safe_name": "donji-miholjac", "country": "HR", "active": true, "selected": true }, ...]

4.3 Pregled razvoja mobilne aplikacije koristeći programski jezik Kotlin

4.3.1 Korištene biblioteke

Biblioteku predstavljaju skupovi klasa, funkcija i dokumentacije koji programerima olakšavaju razvoj vlastitih aplikacija. Prednosti korištenja biblioteka su u tome što mogu drastično ubrzati razvoj novih programskih rješenja. Na primjer, na jednom projektu može se pojaviti zahtjev iz kojeg slijedi potreba za razvoj složene funkcionalnosti poput rukovanja mrežnim zahtjevima. Rukovanje mrežnim zahtjevima obuhvaćalo bi kreiranje potpore za definiranje metoda zaglavlja, slanja sadržaja, primanja odgovora, serijalizacije i deserijalizacije podataka, rukovanjem blokiranjem niti i istovremenosti više zahtjeva. Također, potrebno je rukovati priručnom memorijom (engl. *cache*), omogućiti slanje slika, videa koji ne se ne mogu poslati na klasičan način jednostavnim zahtjevom. Iz primjera je dakle vidljivo da je za samo jedan segment projekta potrebno uložiti veliku količinu vremena bez da se projekt stvarno pomaknuo prema cilju. Takav problem rješiv je jedino velikim tvrtkama koje imaju dovoljno ljudskih resursa u vidu znanja, raspoloživog vremena te novčanih resursa da bi se mogli upustiti u takav pothvat. Za ostale tvrtke to jednostavno nije opcija jer budžet ne dozvoljava potrošnju novca, a poslovanje im direktno ovisi o kreiranom proizvodu. Ostalim tvrtkama preostaje korištenje vanjskih biblioteka koje im omogućavaju fokus na razvoj proizvoda. Glavna razlika između biblioteka koje kreira velika tvrtka za poboljšanje svoga razvoja te biblioteka korištenih od ostalih tvrtki jest pouzdanost takvih biblioteka. Velika tvrtka ima svu kontrolu nad razvojem biblioteke, dodavanjem novih funkcionalnosti, dok ostali ovise o tuđim projektima. Veliki je pothvat dati toliko povjerenje ljudima koji u bilo kojem trenutku mogu promijeniti svoje biblioteke u tolikoj mjeri da se zahtijevaju veće izmjene projekta koji ovisi o tim bibliotekama. Također se može dogoditi da se otkrije sigurnosni propust ili da autori jednostavno odustanu od održavanja biblioteke. Stoga se prije svakog projekta trebaju dobro ocijeniti prednosti i rizici korištenja raznih biblioteka.

Za potrebe razvoja Android aplikacije koristile su se *Retrofit*, *Koin*, *Card View*, *Swipe Refresh Layout*, *Glide* te *Material* biblioteke. Navedene biblioteke potrebno je dodati unutar *build.gradle* datoteke projekta. Navedena datoteka može se pisati Kotlinom ili Groovyjem jer *Gradle* alat za izgradnju projekta može raditi s oba jezika prema [37]. Za kreiranje aplikacije korišten je zapis Groovyjem. Ovisnosti se u *build.gradle* datoteci zapisuju pod blok *dependencies* ključnom riječi *implementation*. Nakon ključne riječi *implementation*, potrebno je unutar para dvostrukih navodnika navesti globalni jedinstveni identifikator u obliku reverznog URL-a zajedno s inačicom biblioteke (Slika 4.4).

```
// UI
implementation "androidx.cardview:cardview:$cardViewVersion"
implementation "com.google.android.material:material:$materialVersion"
// Networking
implementation "com.squareup.retrofit2:retrofit:$retrofitVersion"
implementation "com.squareup.retrofit2:converter-gson:$retrofitVersion"
implementation "com.squareup.okhttp3:logging-interceptor:$okHttpLoggingInterceptorVersion"
// DI
implementation "org.koin:koin-android:$koinVersion"
// Glide
implementation "com.github.bumptech.glide:glide:$glideVersion"
annotationProcessor "com.github.bumptech.glide:compiler:$glideVersion"
// SwipeRefreshLayout
implementation "androidx.swiperefreshlayout:swiperefreshlayout:$swipeRefreshLayoutVersion"
// OpenStreetMap
implementation "org.osmdroid:osmdroid-android:$openStreetMapVersion"
```

Slika 4.4 Korištene biblioteke za razvoj Android mobilne aplikacije

Kao što je vidljivo na slici 4.4, svakoj biblioteci potrebno je specificirati inačicu. To je izvršeno interpolacijom znakovnog niza. Inačice se zatim definiraju u istoj datoteci ali pod blokom *ext* u kojem je omogućeno definiranje konstanti koje se mogu koristiti u *build* procesu. Definiranje inačica biblioteka prikazano je slikom 4.3.

```
ext {
    cardViewVersion = "1.0.0"
    materialVersion = "1.2.0-alpha05"
    retrofitVersion = "2.6.0"
    okHttpLoggingInterceptorVersion = "4.0.0"
    koinVersion = "2.0.1"
    glideVersion = "4.9.0"
    swipeRefreshLayoutVersion = "1.1.0-beta01"
    openStreetMapVersion = "6.1.0"
}
```

Slika 4.3 Inačice biblioteka korištenih za razvoj Android mobilne aplikacije

Nakon definiranja dodatnih biblioteka, odnosno ovisnosti, potrebno je izvršiti takozvanu sinkronizaciju projekta. Android Studio razvojno okruženje pokazuje poruku pri vrhu uređivača teksta kojom naznačuje da je došlo do promjene *build* datoteke te da se pritiskom na gumb može izvršiti sinkronizacija. Tim postupkom automatski se preuzimaju točno određene inačice biblioteka te ih razvojno okruženje priprema za korištenje.

U nastavku su pojašnjene korištene biblioteke.

4.3.2 Biblioteka Card View

Prema [38], biblioteka Card View služi kao element aplikacijskog sučelja u kojemu se mogu nalaziti ostali elementi sučelja. Na te elemente tada je moguće primijeniti razmak od rubova i drugih elemenata, odmah ispune (engl. *padding*) od ruba elementa te položaj u odnosu na druge elemente sučelja korištenjem ograničenja (engl. *view constraints*). Idealan slučaj korištenja jest kada se u aplikaciji mora prikazati lista stavki koje zatim poprimaju dojam fizičke kartice koja

prikazuje ispunjen sadržaj. Dojam kartice postiže se iscrtavanjem sjene oko kartice. Također se nudi mogućnost definiranja polumjera zaobljenosti ruba kartice, definiranje pozadine kartice te visinu kartice spram pozadine. U razvoju aplikacije iskoristila se mogućnost kreiranja kartice zaobljenog ruba kako bi kartica imitirala ponašanje gumba s određenom visinom od baznog (engl. *root*) *viewa*. Biblioteku održava tvrtka Google.

4.3.3 Biblioteka Material

Material biblioteka za cilj ima uvesti elemente sučelja kojim se olakšava razvoj korisničkih sučelja i funkcionalnih komponenti. Biblioteka prati istoimeni pristup dizajnu sučelja poznatijih po imenu *Material Design*. Takav pristup razvoju sučelja zanimljiv je tvrtkama kojima je cilj kreiranje brenda i njegove lake uočljivosti na različitim platformama. Prema [39], biblioteku razvijaju programski inženjeri kao i dizajneri korisničkog iskustva (engl. *User Experience Designers, UX Designers*). Cilj UX dizajnera jest kreirati sučelje koje je intuitivno, odnosno lako za korištenje. U obzir uzimaju količinu potrebnih koraka da se odradi nekakva fundamentalna radnja unutar aplikacije te redizajniranje sučelja ako testovima sučelja uoče otežanost korištenja kod korisnika.

4.3.4 Biblioteka Swipe Refresh Layout

Biblioteka kojom se unutar aplikacije na raspolaganje dobije element sučelja koji se može ugraditi unutar drugih elemenata sučelja. Uobičajeno zauzima cijelu površinu zaslona i sadrži listu elemenata. Dok je prikaz liste na vrhu, dodatnim povlačenjem prsta prema dnu zaslona, pojavljuje se element sučelja koji naznačuje radnju osvježavanja sadržaja liste. Funkcija osvježavanja se ne poziva sve dok korisnik ne odmakne prst od zaslona, a da je prethodno izveo radnju povlačenja prsta prema dnu zaslona. Više se može vidjeti u [40].

4.3.5 Biblioteka Glide

Glide [41] je biblioteka zadužena za učitavanje slika i videa unutar Android aplikacija. Biblioteka također podržava pohranu dohvaćenih slika s interneta na internu ili eksternu memoriju kako bi se minimizirao mrežni promet. U tom slučaju bi interna i eksterna memorija služile kao priručne memorije. U slučaju da se trebaju učitavati slike iz lokalne memorije, biblioteka ne zahtijeva definiranje dozvola unutar *AndroidManifest* datoteke. U slučaju da se slike učitavaju preko interneta potrebno je dodati adekvatnu dozvolu, a ako se želi omogućiti automatsko pozivanje zahtjeva, ukoliko se prvotni zahtjevi nisu uspjeli izvršiti, također je potrebno dodati dozvolu unutar *AndroidManifest* datoteke [42]. Biblioteka također podražava manipuliranje slike kroz transformacije poput centriranja slika u izvornoj veličini i izrezivanja dijelova koji se ne mogu

prikazati u odabranom elementu sučelja ili sužavanje slike tako da se pozicionira unutar elementa specificirane veličine. Cijelu funkcionalnost biblioteke moguće je iskoristiti pišući samo jednu liniju koda.

4.3.6 Biblioteka Open Street Map

Za prikazivanje mape unutar detalja ponude korištena je biblioteka otvorenog koda *osmdroid*, implementiranoj tako da funkcionira nad *Open Street Map*, neovisno od Googleovih usluga i Google Mapa. Prema [43], ova biblioteka omogućuje gotovu jednaku funkcionalnost poput Maps SDK kojeg razvija Google. Za potrebe izrade aplikacije korištena je mogućnost specificiranja koordinata i dodavanje oznake na kartu s kratkim opisom te je svoju svrhu u potpunosti ispunila iako nudi mnoštvo drugih funkcionalnosti.

4.3.7 Biblioteka Koin

Koin [44] je biblioteka razvijena s ciljem pojednostavljivanja uvrštavanja, injektiranja ovisnosti (engl. *dependency injection*, *DI*) u kod. Svrha DI-a jest omogućiti slabije vezivanje koda koje će rezultirati lakšim otkrivanjem i otklanjanjem grešaka. Točnije, funkcionalnosti i odgovornosti odvojene su u klase, a komunikacija se ostvaruje pomoću sučelja. To znači da ako jedan objekt poziva metode iz sučelja koje su implementirane u drugom objektu, da se implementacija metoda u drugom objektu može mijenjati bez da se izvrše promjene u kodu pozivatelja. Također se olakšava i testiranje jer sve što tester treba napraviti jest kreirati svoju klasu s implementiranim metodama i svojstvima iz sučelja kako bi tim objektom imitirao pravi objekt u produkciji. Nakon definiranja takvog objekta, potrebno je promijeniti i definiciju injektiranog objekta unutar DI kontejnera (engl. *DI container*) koji generira objekte kada se zatraže. Koin omogućava injektiranje pomoću *singletona* te *factoryja*. *Singleton* predstavlja objekt koji se kreira samo prvi put kada ga se zahtijeva, a u svim ostalim slučajevima vraća se točno ta instanca koja je kreirana prilikom prvog poziva. *Factory* predstavlja metodu generiranja ovisnosti poput stvarne tvornice u kontekstu da se prilikom svakog zahtijevanja ovisnosti kreira novi objekt. Dokumentacija je raspoloživa na [45], a u potpoglavlju za razvoj pokazano je podešavanje i korištenje Koina u razvoju Android aplikacije.

4.3.8 Biblioteka Retrofit

Retrofit biblioteku razvija američka tvrtka Square [46]. Predstavlja klasu kroz koju se kreira objekt koji implementira vlastito definirano sučelje mrežnog usluga. Sučeljem se definiraju nazivi metoda, putanje do resursa, adekvatna HTTP metoda, zaglavlja, parametri te sadržaj za slanje pri

komunikaciji s poslužiteljem, tj. API na kojeg će se spajati. Ako se prilikom definiranja navedenog sučelja očekuje rezultat, potrebno je definirati objekt u kojeg se vrši deserijalizacija odgovora poslužitelja. Prilikom kreiranja *retrofit* klijenta, potrebno je definirati HTTP klijenta, bazni URL API-ja koji se koristi te pretvarača (engl. *converter*) koji vrši serijalizaciju i deserijalizaciju podataka za svaki zahtjev. Retrofit za slanje HTTP zahtjeva koristi OkHttp biblioteku također kreiranu od tvrtke Square. Dodatna ovisnost sa 4.4 koja se koristi u aplikaciji jest *OkHttp Logging Interceptor* koji se koristi za potrebe razvoja aplikacije tako da prilikom svakog zahtjeva unutar *Logcat* konzole zapiše sadržaj tijela HTTP zahtjeva kojeg je poslužitelj vratio kao odgovor. Od podržanih pretvarača, prema [47], *Retrofit* podržava *Gson, Jackson, Moshi* te *Simple XML*. U ovome radu korišten je *Gson* konverter kreiran od Googlea [48]. Točni implementacijski detalji dani su sljedećim potpoglavljima.

4.4 Razvoj Android aplikacije

4.4.1 Postavljanje i karakteristike Android projekta

Kako bi se započeo razvoj aplikacije potrebno je otvoriti razvojno okruženje Android Studio te odabrati *File->New->New Project*. Zatim je iz kartice *Phone and Tablet* potrebno odabrati *Empty Activity* te odabrati *Next*. Na novootvorenom zaslonu potrebno je unijeti naziv aplikacije, ime paketa, tj. jedinstveno ime paketa, putanju do direktorija koji je korijenski direktorij za cijelu aplikaciju, programski jezik kojim se aplikacija razvija, te minimalna inačica Android operacijskog sustava kojeg aplikacija podržava. Za jedinstveno ime paketa preporučuje se koristiti reverzni URL koji počinje od vrhovne domene, imena tvrtke, a završava na najmanjoj domeni, odnosno na imenu aplikacije. Bitno je osigurati jedinstvenost imena paketa u slučaju da se u budućnosti aplikaciju želi distribuirati putem *Google Play* platforme čime aplikacija dobiva potencijalno veliko tržište iako je aplikaciju moguće distribuirati bez navedene platforme. Odabirom minimalne inačice Android operacijskog sustava kojeg aplikacija podržava postavlja se ograničenje na klase koje se mogu koristiti prilikom razvoja, rukovanje dozvolama itd. Android Studio također pokazuje postotni udio korištenog operacijskog sustava na mobilnim uređajima što može pomoći pri odabiru minimalno podržane inačice. Za razvoj ove aplikacije korištena je minimalna API razina 26, odnosno, Android 8.0 poznatiji kao Oreo. Nakon unosa i odabira prethodno navedenih podataka, može se odabrati *Finish* nakon čega se pokreće generiranje projekta. *Gradle* sustav za gradnju projekta preuzet će ovisnosti, odnosno biblioteke od kojih se svaki projekt mora sastojati te će također pripremiti hijerarhiju projekta. Nakon inicijalnog kreiranja projekta potrebno je dodati ovisnosti u *build.gradle* datoteku vezanu za aplikaciju prema

slikama 4.4 i 4.3 te izvršiti sinkronizaciju projekta prilikom koje *Gradle* sustav pronalazi, preuzima i uvrštava u projekt.

Nakon postavljanja projekta može se pristupiti razvoju. Potrebno je odabrati *Project* s lijeve strane sučelja te iz otvorenog prozora iz padajućeg izbornika odabrati *Android*. U prozoru se zatim prikazuju dva direktorija. Brzu ekspanziju direktorija moguće je ostvariti odabirom direktorija i pritiskom tipke '*'. Pod *app* direktorijem otkrivaju se dodatni direktoriji – *manifests, java, res*. *Manifests* direktorij sadržava *AndroidManifest.xml* datoteku zaduženu za opis glavnih stavki aplikacije te dozvola. Direktoriji za organizaciju projekta te datoteke izvornog koda postavljaju se pod direktorij „*java-> <jedinstveni naziv paketa definiran prilikom kreiranja projekta>“. Res* direktorij služi za pohranu resursa koji su korišteni za izgradnju aplikacije te korisničkog sučelja. Unutar tog direktorija, nakon kreiranja novog projekta, nalaze se direktoriji *drawable, layout, mipmap* te *values*. Prema [49], *drawable* direktorij služi za pohranu grafičkih resursa unutar aplikacije u *.png, .9.png, .jpg, .gif* formatima te onima specificiranim pomoću XML-a što uključuje slike definirane SVG formatom te definicije stanja koja su svojstvena Android platformi [50]. Unutar *layout* direktorija definiraju se XML datoteke kojima se opisuju cijela sučelja ili elementi sučelja koji će se prikazivati na mobilnim uređajima. Uobičajeno se definiraju zasloni za aktivnosti i fragmente, glavnim gradivnim blokovima Android aplikacija. *Mipmap* direktorijem definirane su ikone aplikacije koje će se pokazivati na uređajima s različitom gustoćom piksela zaslona. Predstavlja direktorij koji se najmanje koristi tijekom razvoja aplikacija jer se ikone mogu definirati i direktorijem *drawable*. *Values* direktorijem definiraju se XML datoteke *colors, styles, strings* kojima su specificirane boje, stilovi te znakovni nizovi koji se koriste unutar aplikacije. Dodatno je moguće definirati *arrays, dimens, public* i ostale XML datoteke. *Arrays* datotekom omogućeno je definiranje niza znakova ili cijelih brojeva, *dimens* datotekom omogućeno je specificiranje često korištenih dimenzija unutar sučelja pod određenim nazivom tako da se mogući zahtjev promjene dimenzije može izvršiti iz centraliziranog mjesta. *Public* datotekom je moguće definirati konstantan identifikator određenih resursa ukoliko to aplikacija zahtijeva. Dodatni direktoriji koji mogu biti definirani unutar *res* direktorija obuhvaćaju *anim, menu* te *font* direktorije. *Anim* direktorijem definirane su XML datoteke kojima se animiraju elementi korisničkog sučelja u aplikaciji, *menu* direktorijem definiran je sadržaj izbornika koji se zatim u aplikaciji mogu prikazivati u padajućem izborniku ili *navigation draweru*. *Font* direktorijem specificirani su različiti fontovi koji će se koristiti unutar aplikacije. Podržani tipovi fontova jesu *.ttf, .otf* ili *.ttc*. Prema naputku iz [49], pojedinačni resursi moraju biti postavljeni u adekvatne direktorije unutar *res* direktorija jer u protivnom dolazi do kompilacijske greške.

AndroidManifest datoteka ključna je prilikom razvoja mobilnih aplikacija. Datoteka se mora nalaziti u korijenskom direktoriju izvornih datoteka, a nužna je za alat *Gradle* prilikom gradnje projekta, Android operacijski sustav te Google Play [51]. Korijenski element datoteke je *manifest* koji mora sadržavati naziv aplikacijskog paketa te prostor imena koji se koristi za definiranje ostalih elemenata unutar korijenskog elementa. Pomoću naziva aplikacijskog paketa *Gradle* pronalazi datoteke izvornog koda tijekom izgradnje projekta. Prema [52], također služi za definiranje prostora imena klase koja sadržava sve resurse u aplikaciji s generiranim identifikatorima. Ta klasa zove se *R* klasa, a generirana je tijekom izgradnje projekta. Klasa nudi nekoliko svojstava pomoću kojih se može pristupiti definiranim resursima aplikacije, a u razvoju se najčešće koriste *id*, *layout*, *string*, *drawable* te *menu*. Ako je naziv aplikacijskog paketa *com.example.ferit*, tada je putanja do *R* klase *com.example.ferit.R*. Naziv aplikacijskog paketa također služi poput baznog URL-a koji se nadodaje na relativne putanje aktivnosti definiranih unutar *manifest* datoteke. *AndroidManifestom* definirane su sve ključne komponente aplikacije poput aktivnosti, usluga, *broadcast receivera* te *content providera*. Definirane su i dozvole (engl. *permissions*) ukoliko se nešto potražuje od sustava ili ako se može narušiti korisnikova privatnost. *AndroidManifest* datotekom također su specificirani nužni zahtjevi za hardverom uređaja na kojima bi se aplikacija pokretala. Navedeno omogućava *Google Playu* da uređajima koji nemaju traženu funkcionalnost ne prikazuje aplikacije koje ne mogu pokretati.

Prema [53], postoje tri tipa dozvola na Android platformi no najčešće su korištena dva tipa – normalne dozvole (engl. *normal permissions*) i opasne dozvole (engl. *dangerous permissions*). Dozvole koje pripadaju u kategoriju normalnih dozvola automatski se odobravaju prilikom instalacije aplikacije na mobilni uređaj te se ne pita korisnika za odobravanje. Korisnik također ne može ukinuti takvu dozvolu. Normalnim dozvolama obuhvaćene su radnje kojima se ne može narušiti privatnost korisnika. Opasnim dozvolama obuhvaćena je interakcija s korisničkim podacima poput kontakata ili e-mailova, pohrana i uvrštavanje određenih podataka u hijerarhiju direktorija te one radnje koje mogu utjecati na rad ostalih aplikacija na mobilnom uređaju. Za opasne dozvole potrebno je tražiti odobrenje korisnika neposredno prije mjesta u aplikaciji gdje je ta dozvola potrebna za ostvarivanje funkcionalnosti. Time se osigurava da korisnik razumije kontekst u kojem se koristi tražena dozvola te se povećava vjerojatnost odobrenja radnje. Primjeri normalnih dozvola unutar Android platforme bili bi: *ACCESS_NETWORK_STATE, INTERNET, RECEIVE_BOOT_COMPLETED, SET_WALLPAPER*. Neke od opasnih dozvola bile bi: *ACCESS_COARSE_LOCATION, CALL_PHONE, CAMERA, RECORD_AUDIO, WRITE_EXTERNAL_STORAGE*. Na [54] vidljiv je potpun opis normalnih i opasnih dozvola.

Kao što je ranije spomenuto, *AndroidManifestom* potrebno je definirati sve usluge, primatelje razašiljanja (engl. *broadcast receiver*) te pružatelje sadržaja (engl. *content provider*). Uslugama na Android platformi definirane su komponente [55] koje se izvršavaju u pozadini bez korisničkog sučelja. U tu kategoriju pripadaju radnje izrade sigurnosnih kopija, osvježavanje sadržaja za prikazivanje u aplikaciji putem interneta, komunikacija s drugim procesima itd. Usluge na Android platformi podijeljene su na one u prvom planu (engl. *foreground*), u pozadini (engl. *background*) i one čiji je životni vijek (engl. *lifecycle*) vezan za postojanje komponenti koje ga koriste, a zovu se *bound* usluge. *Foreground* usluge moraju prikazati korisniku informaciju da se izvode u obliku notifikacije koja se prikazuje u gornjem dijelu zaslona. *Foreground* usluge uobičajeno uključuju reprodukciju glazbe, snimanje zvuka ili zaslona. Pozadinskim uslugama omogućeno je ažuriranje lokalne baze podataka, priprema podataka za sljedeće korisnikovo učitavanje itd. Vezane usluge (engl. *bound services*) usluge omogućuju klijent-poslužitelj način komunikacije između procesa. Takvi usluge su aktivne sve dok postoji barem jedan proces koji ih koristi. Primatelji razašiljanja predstavljaju posebne komponente koje se razvijaju unutar aplikacije, a cilj im je pokrenuti nekakav zadatak kada se pojavi događaj na koji su se registrirali. Prema [56], registracije pojedinih aplikacija na određene događaje vodi Android operacijski sustav. Kada se određeni događaj pojavi, Android operacijski sustav šalje *broadcast* poruku svim aplikacijama koje su se registrirale. Tako primjerice Android operacijski sustav šalje *broadcast* nakon uključivanja mobilnog uređaja, promjene mrežne raspoloživosti, nakon prikapčanja na punjač, uključivanje zrakoplovnog načina rada itd. Potpuni popis moguće je vidjeti na [57]. Tako se primjerice prikapčanjem na punjač može pokrenuti energetski zahtjevnija operacija poput obrade podataka ili izrade sigurnosne kopije i slanje na Googleove poslužitelje ako postoji internetska veza. Prema [58], pružatelj sadržaja predstavlja komponentu kojom je drugim aplikacijama omogućen pristup podacima poput relacijske baze podataka. Omogućava siguran pristup podacima te standardno sučelje preko kojega izmjenjuju podaci.

Za potrebe razvoja definirane su četiri dozvole od kojih su dvije normalne razine, *INTERNET* i *ACCESS_NETWORK_STATE*, te dvije opasne dozvole, *ACCESS_FINE_LOCATION* i *WRITE_EXTERNAL_STORAGE*. Nakon toga se atributima elementa definira korijenska klasa aplikacije, ikone aplikacije, naziv te tema. Za specificiranje svakog atributa korišten *android* prostor imena. Za korijensku klasu aplikacije korištena je relativna putanja po hijerarhiji direktorija koja na kraju izgleda *com.example.cityalarm_android_kotlin.RootAppActivity*. Specifikacija ikona odrađena je referenciranjem *drawable* direktorija za resurse aplikacije sa specificiranjem imena *ic_launcher*. Tema primijenjena na cijelu aplikaciju specificirana je referencirajući *style*

datoteku unutar *values* direktorija. Na slici 4.5 prikazan je navedeni stil. Unutar *application* oznake mogu se vidjeti definicije aktivnosti koje se koriste za rad aplikacije. Svaka aktivnost minimalno mora imati specificiran naziv koji navodi lokaciju do klase koja rukuje korisničkim sučeljem. Za svaku aktivnost osim pozdravnog zaslona (engl. *splash screen*) postoji definiran atribut *label* kojim su referencirani znakovni nizovi unutar *strings* datoteke resursa. Za neke aktivnosti definirani su atributi *screenOrientation*, *windowSoftInputMode* te posebna tema kojom se skriva podrazumijevana alatna traka (engl. *action bar*) unutar aplikacije vidljivog na slici 4.5.

```
…
<style name="AppTheme"
        parent="Theme.AppCompat.Light.DarkActionBar">
    <!-- Customize your theme here. -->
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>

</style>

<style name="NoActionBar"
        parent="Theme.AppCompat.Light.NoActionBar">
    <item name="android:colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="windowActionBar">false</item>
    <item name="windowNoTitle">true</item>
</style>
…
```

Slika 4.5 Prikaz dijela kreiranih stilova

Postavljanjem *screenOrientation* atributa ograničena je aktivnost da ne reagira na promjene orijentacije uređaja te time da ne zahtijeva ponovno iscrtavanje sučelja pošto je aplikacija zamišljena da za korištenje samo u okomitom prikazu. *WindowSoftInputMode* atributom postavljenim na *adjust pan* vrijednost, upućuje se Android operacijski sustav da pomakne trenutno odabrani element za unos teksta prema gore kako ga tipkovnica ne bi prekrila kada se prikaže na zaslonu mobilnog uređaja. Dodatne opcije atributa moguće je vidjeti na [59]. Na *UserHomeScreen* aktivnosti uklonila se alatna traka jer ju je potrebno kreirati programskim putem zbog korištenja posebnog izbornika. Pozdravni zaslon u *AndroidManifest* datoteci sadržava oznaku *intent-filter* s dvije dodatne oznake: *action* i *category*. Tim oznakama specificiran je pozdravni zaslon kao prvi zaslon aplikacije kada korisnik pokrene aplikaciju.

4.4.2 Opis glavnih komponenti sučelja Android aplikacija

Osnovni gradivni element Android aplikacija predstavljaju korisnička sučelja. Korisnička sučelja moraju se udomiti na komponente Android platforme koji im to omogućuju, a riječ je o aktivnostima (engl. *activities*) te fragmentima (engl. *fragments*).

Aktivnost predstavlja glavni element Android aplikacija jer na raspolaganje daje zaslon uređaja u kojeg se iscrtava korisničko sučelje [60]. Aktivnosti tako mogu zauzimati cijeli zaslon, neki njegov

dio ili iscrtavati povrh već neke prikazane aktivnosti. Uobičajeno jest da aktivnost zauzima cijeli zaslon mobilnog uređaja. Na korisnikov zahtjev otvaranja aplikacije uvijek se pokreće i prikazuje jedna aktivnost. Od te jedne aktivnosti moguće je vršiti prijelaze na ostale aktivnosti unutar vlastite aplikacije ili koristiti aktivnosti drugih aplikacija za ostvarivanje korisničke naredbe. Svaka aktivnost ima svoj životni vijek kojim upravlja Android operacijski sustav na naredbe korisnika ili u iznimnim situacijama kada sustav nema slobodne memorije pa je prisiljen zatvoriti aktivnost. Prema [61], postoji sedam metoda povratnih poziva kojim Android operacijski sustav upravlja životnim vijekom određene aktivnosti, a riječ je o metodama *onCreate, onStart, onResume, onPause, onStop, onDestroy, onRestart* [61].

U svakoj aktivnosti nužno je postojanje metode *onCreate* koja je zapravo ulazna točka u određenu aktivnost slično *main* metodama na ostalim platformama. Ostale metode su opcionalne i moguće ih je implementirati u slučaju da se treba izvršiti nekakva radnja kada ih sustav pozove. *OnCreate* metodom kreiraju se korisnička sučelja, postavljaju elementi sučelja te iniciraju radnje koje učitavaju podatke. Nakon poziva slijedi poziv *onStart* metode te tada aktivnost postaje vidljiva korisniku na zaslonu mobilnog uređaja. Nakon poziva *onStart* mogući su pozivi na *onResume* i *onStop* metode. Pozivom o*nResume* aktivnost je došla u prvi plan te korisnik može zadavati određene zadatke koje želi izvršiti. *OnPause* metodom aktivnost nije više u prvom planu te se može iskoristiti prilika da se pohrani stanje aktivnosti i pohrane određeni podaci u priručnu memoriju ukoliko je potrebno. Prema [61], potrebno je da metoda brzo završi jer se druga aktivnost ne može pokrenuti sve dok prva metoda nije završena. Nakon poziva *onPause* metode mogu slijediti pozivi na *onResume* i *onStop* metode. *OnStop se* poziva kada aktivnost nije u prvom planu Poslije poziva *onStop* mogu slijediti pozivi na *onRestart* te *onDestroy*. *OnRestart* poziva se ako se prethodno zaustavljena aktivnost ponovno želi vratiti u prvi plan, a *onDestroy* uklanja aktivnost iz memorije. *OnDestroy* se poziva nakon eksplicitnog poziva na *finish* metodu aktivnosti ili kada sustav nema dovoljno memorije za normalan rad.

Pokretanje drugih aktivnosti može obuhvaćati uporabu kamere, otvaranje web preglednika na pojedini URL, otvaranje Google Maps aplikacije na određene koordinate itd. Pokretanje drugih aktivnosti ostvaruje se kreiranjem instance *Intent* klase. Kreirani *intent* objekt može biti eksplicitan ili implicitan te se također može specificirati pokreće li se druga aktivnost radi obavljanja zadatka i vraćanja rezultata ili se pokreće kako bi se upotpunila vlastita aplikacija [62]. Eksplicitnim *intentom* mora se znati točan naziv aktivnosti koja se želi pokrenuti unutar vlastite ili druge aplikacije, te ako se žele proslijediti podaci u obzir se mora uzeti i podržani tip podataka druge aktivnosti. Implicitnim *intentom* ne specificira se aktivnost koja se želi pokrenuti, nego je potrebno

specificirati radnju (engl. *action*) koja treba biti obavljena od druge kompatibilne aktivnosti te opcionalno podaci koji se prosljeđuju. Takav *intent* zatim se predaje Android operacijskom sustavu koji pregledava aplikacije na mobilnom čije aktivnosti mogu ispuniti traženi scenarij te kreira listu koja se korisniku pojavi na donjem dijelu zaslona. Korisnik tada prema osobnim preferencijama odabire aplikaciju koja će obaviti zadatak. Najkorištenije radnje predstavljaju *ACTION_VIEW* te *ACTION_SEND*. Neki implicitni *intentovi* zahtijevaju specifikaciju URL-a s uvrštenim podacima za obradu, a neki zahtijevaju korištenje posebne metode *putExtra* nad *intent* objektom. *PutExtra* metodom potrebno je definirati parove ključ-vrijednost koje će odredišna aktivnost moći iskoristiti za dohvaćanje poslanih podataka. Ključevi se uobičajeno definiraju pomoću konstanti na tipu *Intent*. Tako se ključem *Intent.EXTRA_TITLE* definira naslov prozora sustava za odabir aplikacije koja izvršava određeni *intent*, a ključem *EXTRA_TEXT* definira se sadržaj poruke kojim određena aktivnost treba rukovati.

Fragmenti predstavljaju alternativni način prikazivanja korisničkog sučelja. Svaki fragment zahtijeva uvrštavanje u *ViewGroup* objekt ili nekog njegovog nasljednika koji primjerice može biti *FrameLayout* [63]. Glavna karakteristika fragmenata jest da ih se može prikazivati više unutar jedne aktivnosti, iako je moguće prikazivanje samo jednog fragmenta preko cijelog korisničkog sučelja. Većina aplikacije napravljena je pomoću jedne aktivnosti unutar koje se izmjenjuju fragmenti koji predstavljaju korisnikova odredišta odabirom u *navigation draweru*. Prema [63], nužno je da klasa koja rukuje određenim fragmentom nasljeđuje klasu *Fragment*. Pošto su fragmenti zaslužen za prikazivanje korisničkog sučelja, također imaju slične metode životnog vijeka poput aktivnosti, ali uz male dodatke. Tako postoje dodatne *onAttach, onCreateView, onViewCreated, onDestroyView* te *onDetach* metode povratnih poziva koje poziva Android operacijski sustav. *OnAttach* metoda u fragmentu poziva se nakon što je fragment pridružen aktivnosti. Budući da se aktivnost kojoj je fragment pridružen šalje kao parametar metodi *onAttach*, moguće je definirati aktivnost kao slušatelja (engl. *listener*) koji reagira na nekakvu radnju učinjenu unutar fragmenta. Prema [64], *onCreateView* metodom potrebno je sustavu predati identifikator kojim se generira sučelje fragmenta definirano XML datotekom iz *res->layout* direktorija. Za učitavanje podataka i konfiguriranje fragmenta preporuča se korištenje *onViewCreated* metode koja se poziva neposredno nakon kreiranja fragmenta. *OnDestroyView* poziva se prilikom uklanjanja sučelja iz fragmenta, a *onDetach* poziva se nakon uklanjanja veze s aktivnošću koja je udomljavala fragment.

Izmjena fragmenata unutar *ViewGroup* ili *FrameLayouta* uvrštenih unutar aktivnosti vrši se transakcijama (engl. *transaction*). Transakcijom je potrebno specificirati u koji element sučelja se

uvrštava fragment, te sam fragment. Transakcije mogu dodavati, zamjenjivati te uklanjati fragmente iz navigacijskog stoga.

U aplikaciju je potrebno dodati korijensku aktivnost za ispravnu konfiguraciju *Koin* biblioteke za injektiranje ovisnosti. Za dodavanje aktivnosti potrebno je unutar Android Studija desnim klikom odabrati naziv paketa definiranog prilikom definiranja projekta te odabrati *New->Kotlin File/Class*. U novootvorenom prozoru potrebno je unijeti naziv datoteke u koju se pohranjuje klasa. Odabrano ime jest *RootAppActivity*. Također se kreirala istoimena klasa koja nasljeđuje klasu *Application*. Nasljeđivanjem *Application* klase osigurava se da je *RootAppActivity* klasa koja se pokreće prije svih drugih klasa, *broadcast receivera* itd. Idealno je mjesto za definiranje varijable koja sadrži instancu te klase kako bi se moglo pristupiti nužnom aplikacijskom kontekstu iz bilo koje klase unutar aplikacije. Varijabla koja se definira jest privatna samo za navedenu klasu, a instancu klase može se dobiti kreiranom metodom. Također je bitno napomenuti da se pridruživanje instance klase *RootAppActivity* varijabli vrši samo jedanput, tj. prilikom samog pokretanja aplikacije te navedeno predstavlja *singleton* instancu. U navedenoj klasi potrebno je konfigurirati biblioteku *Koin* pomoću koje se vrši injektiranje ovisnosti unutar aplikacije. To je moguće definiranjem lambde na *startKoin* metodi biblioteke. Pošto je jedini parametar metode *lambda* na tipu *KoinApplication (KoinApplication.() -> Unit)*, tu je lambdu unutar Kotlin programskog jezika moguće napisati izvan zagrada te se zagrade mogu ukloniti iz poziva funkcije. Unutar lambde je pristup svim metodama *KoinApplication* klase. Za razvoj aplikacije korištena je metoda *modules* kojom se definiraju sve konfigurirane komponente unutar liste koje će *Koin* moći razriješiti i injektirati na tražena mjesta. Također se pri definiranju *Koina* može specificirati kontekst kojeg će *Koin* koristiti u svom radu. Cijela *RootAppActivity* klasa može se vidjeti programskim kodom 4.1. Iz programskog koda je vidljivo da su definirana četiri modula od kojih svaki prati svoje odgovornosti. Riječ je o *domain* modulu koji prati scenarije odnosno slučajeve korištenja (engl. *use case, interactor*) aplikacije, *networking* modul kojim je konfiguriran *Retrofit* klijent, *preferences* modul koji generira *singleton* instance klase *SharedPreferences*. *Presentation* modulom definirani su svi prezenteri korišteni za ostvarivanje MVP arhitekture aplikacije. Svaki od modula započinje definiranjem varijable naziva tog modula te pridruživanje *Koin module* lambde navedenoj varijabli. Unutar modula može biti definiran *factory* ili *single* način kreiranja određene instance nekog tipa. Kod svake takve metode unutar para zagrada mora biti definiran naziv sučelja pomoću kojega *Koin* na mjestu injektiranja generira pravilan objekt koji implementira navedeno sučelje. Tako su, koristeći *factory*, generirani *domain* i *presentation* moduli, dok su *networking* i *preferences* moduli napravljeni pomoću definicije *single*. Za

```
class RootAppActivity : Application() {
    companion object {
        private lateinit var instance: RootAppActivity
        fun getApplicationContext() = instance
    }

    override fun onCreate() {
        super.onCreate()
        instance = this
        startKoin {
            modules(
                listOf(
                    domainModule,
                    networkingModule,
                    preferencesModule,
                    presentationModule
                )
            )
            androidContext(this@RootAppActivity)
        }
    }
}
```

Programski kod 4.1 Prikaz RootAppActivity klase

rješavanje potrebnih ovisnosti za neku od klasa definiranih u navedenim modula, korištena je *get* metoda unutar *Koina* te nije potrebno eksplicitno specificirati tip podatka koji obrađuje. *Koin* ima ugrađenu podršku za pronalazak traženih tipova, a ako tijekom pokretanja aplikacije nije pronađen adekvatan objekt, dolazi do *runtime* iznimke. Nakon definiranja korijenske aktivnosti aplikacije, kreirane su pomoćne klase i metode koje olakšavaju razvoj. Tako je kreirana metoda proširenja na tipu *FragmentActivity* kako bi bila omogućena izmjena učitanih fragmenata pomoću jednostavnog poziva metode. U protivnom bi se kod trebao bespotrebno duplicirati na svakom mjestu gdje treba biti izvršena izmjena, dok je na ovaj način omogućena centralizacija tog koda. Kreirane su dvije dodatne metode proširenja pomoću kojih se, u ovisnosti prisustva URL-a unutar dohvaćene ponude, preuzima slika ili učitava podrazumijevana slika pomoću Glide biblioteke. Implementirane su i metode kojima se olakšava rukovanjem dozvolama korisnika. Kreirane su metode proširenja povrh *Context* klase za jednostavno prikazivanje *toast* poruka. Metodama se predaje tekstualni sadržaj koji treba biti prikazan unutar poruke ili se predaje identifikator na znakovni resurs iz *R* klase. Kao drugi parametar implementirano je dugo trajanje poruke kao podrazumijevanu vrijednost tog parametra, no moguće je specificirati drugu vrijednost koja se koristi unutar poziva metode. Za prikazivanje i sakrivanje elementa kreirane su dvije pomoćne metode proširenja definirane povrh *View* klase. Naposljetku definirana je opća (engl. *generic*) metoda kojom se olakšava pokretanje drugih aktivnosti unutar aplikacije koristeći prethodno opisani implicitni *intent*. Prikaz metode vidljiv je programskim kodom 4.2.

Programskim kodom 4.2 može se vidjeti da je kao prvi parametar kreiranja *Intent* objekta korišten aplikacijski kontekst definiran kao *singleton* unutar *RootAppActivity*, a kao drugi parametar generički tip klase na koju upućuje *intent* objekt. Zatim je na instanci *intent* objekta metodom *run*

```
fun <T> activityChanger(destinationClass: Class<T>) {
    Intent(RootAppActivity.getApplicationContext(), destinationClass).run {
        flags = Intent.FLAG_ACTIVITY_NEW_TASK
        startActivity(RootAppActivity.getApplicationContext(), this, null)
    }
}
```

Programski kod 4.2 Prikaz metode za izmjenu trenutno prikazane aktivnosti

specificirana posebna zastavica kojom je Android operacijskom sustavu naznačeno da mora pokrenuti potpuno novu aktivnost te izvrši kreirani *intent* pozivom *startActivity* metode. Kao parametri *startActivity* metode proslijeđen je aplikacijski kontekst, pod *this* parametrom kreirani *Intent* objekt te *null* parametrom kojim je specificirana odsutnost posebnih opcija prilikom pokretanja tražene aktivnosti.

4.4.3 Razvoj zaslona Android aplikacije

Kao prvi zaslon korištena je aktivnost koja predstavlja pozdravni zaslon. Dodavanje nove aktivnosti u projekt može biti izvršeno na dva načina. Prvim načinom moguće je desnim klikom na naziv paketa odabrati *New->Kotlin File/Class* te imenovati datoteku koja pohranjuje novu klasu. Zatim je u toj datoteci potrebno definirati klasu koja nasljeđuje *AppCompatActivity* te je potrebno ručno napraviti zapis klase unutar *AndroidManifest* datoteke. Za kreiranje datoteke korisničkog sučelja potrebno je desnim klikom odabrati direktorij *res->layout* te odabrati *New->Layout Resource File*, te dodijeliti ime datoteke korisničkog sučelja. Drugi način kreiranja aktivnosti jest *New->Activity->Empty Activity*. Nakon toga otvoren je prozor kojim se specificira naziv aktivnosti, te se automatski generira naziv datoteke korisničkog sučelja koje se može promijeniti ako je potrebno. Odabirom *Finish* tipke Android Studio generira zapis za aktivnost unutar *AndroidManifest* datoteke, kreira izvornu datoteku aktivnosti te datoteku korisničkog sučelja u *layout* direktoriju. Kako bi se uklonilo pozivanje *onCreate* metode za aktivnosti i *onCreateView* metode prilikom kreiranja fragmenata, kreirane se su dvije apstraktne klase koje nasljeđuje svaka novo kreirana aktivnost ili fragment. Programskim kodom 4.3 vidljiva je implementacija apstraktne klase *BaseActivity*.

```
abstract class BaseActivity : AppCompatActivity() {
    abstract fun getLayoutResourceId(): Int
    abstract fun setUpUi(savedInstanceState: Bundle?)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(getLayoutResourceId())
        setUpUi(savedInstanceState)
    }
}
```

Programski kod 4.3 Prikaz BaseActivity apstraktne klase

Sa slike je moguće vidjeti da svaka klasa koja naslijedi *BaseActivity* morati dati implementaciju *getLayoutResourceId* te *setUpUi* metoda. *GetLayoutResourceId* metodom specificira se

identifikator resursa iz *layout* datoteke pomoću *R* klase te se ta metoda poziva unutar povratnog poziva *onCreate* metode prilikom pokretanja aplikacije. *SetContentView* metoda generira korisničko sučelje te ga prikazuje na zaslonu mobilnog uređaja. Nakon kreiranja sučelja pozvana je kreirana implementacija *setUpUi* metode u klasi koja nasljeđuje *BaseActivity*. Slično se događa s *BaseFragment* apstraktnom klasom. Ondje je specificirana samo metoda kojom se dohvaća vrijednost identifikatora korisničkog sučelja koji predstavlja fragment. Zatim se, koristeći napuhivač sučelja (engl. *layout inflater*), generira sučelje fragmenta iz kreiranih XML datoteka. Tako se za kreiranje pozdravne aktivnosti koristila apstraktna klasa *BaseActivity*. Unutar datoteke sučelja dodan je element sučelja *ImageView* kojim je iscrtan logo na pozdravnom zaslonu te je također specificirana pozadina cijele aktivnosti. Na samom elementu *ImageView* potrebno je specificirati širinu i visinu elementa te ograničenja sučelja koja pozicioniraju navedeni element s obzirom na element roditelja, koji je u ovom slučaju *Constraint Layout*. Nakon specificiranja korisničkog sučelja potrebno je specificirati sučelja koja će implementirati prezenter i *view*, tj. klasa aktivnosti. Za organiziranje datoteka izvornog koda kreirani su paketi koji pohranjuju srodne datoteke. Pošto je unutar Kotlin programskog jezika moguće definirati sučelja unutar sučelja, odabran je takav pristup specificiranja ugovora (engl. *contract*) kojim su objedinjena sučelja prezentera i pogleda. Nakon definicije sučelja, potrebno ih je implementirati u klasama prezentera i pogleda te unutar konstruktora prezentera omogućiti korištenje slučaja korištenja koji omogućava provjeru je li korisnik prijavljen. Također je potrebno izvršiti registraciju sučelja prezentera i konkretne implementacije unutar Koin modula kako bi se mogao injektirati na mjestu korištenja. Registracija prezentera prikazana je programskim kodom 4.4.

```
factory<SplashActivityContract.Presenter> { SplashActivityPresenter(get()) }
```

Programski kod 4.4 Registracija prezentera unutar presentation modula

Nakon definiranja svih metoda unutar aktivnosti pozdravnog zaslona, unutar aktivnosti pozdravnog zaslona potrebno je injektirati vrijednost varijable prezenter na prethodno kreiranu implementaciju klase prezentera. Navedeno se ostvaruje koristeći *inject* metodu za učitavanje ovisnosti. Korištenje metode prikazano je programskim kodom 4.5.

```
private val presenter by inject<SplashActivityContract.Presenter>()
```

Programski kod 4.5 Prikaz korištenja inject metode

Sa slike je vidljivo da se potražuje tip koji implementira sučelje *SplashActivityContract.Presenter* te će se rezolucija i injektiranje ovisnosti izvršiti prilikom prvog korištenja navedene varijable. Navedeno se događa unutar *setUpUi* metode gdje se mora postaviti povratni poziv unutar danog prezentera na aktivnost tipa *SplashActivityContract.View*. Takav način učitavanja gdje varijabla

poprima vrijednost prilikom prvog poziva naziva se lijenim učitavanjem *(engl. lazy loading).* Nakon toga potrebno je pokrenuti radnju unutar prezentera koja odlučuje o tome je li korisnik prijavljen. Rezultat te radnje propagira se kroz povratni poziv na aktivnost iz samog prezentera. U aktivnosti se zatim preusmjerava korisnika na zaslon za prijavu ili središnju točku aplikacije kojom su učitane ponude. Za potrebe izmjene trenutno prikazane aktivnosti korištena je kreirana metoda prikazana programskim kodom 4.2. Nakon kreiranja pozdravnog zaslona kreirani su zasloni za prijavu i registraciju korisnika. Zaslon za prijavu znatno je lakše kreirati od zaslona za registraciju zbog manjeg broja zahtjeva za korisničkim podacima. U aplikaciju je zatim potrebno dodati aktivnosti na jedan od prethodno dva opisana načina. Nakon dodanih aktivnosti, potrebno je unutar *AndroidManifest* datoteke specificirati atribut nad kreiranom aktivnošću da se pojavom tipkovnice spriječi prekrivanje elementa prilikom unosa teksta.

Sučelje za prijavu korisnika sastoji se od elementa za prikaz loga, dva elementa za unos teksta, gumba za prijavu, gumba za odlazak na aktivnost registracije te indikatora pozadinske aktivnosti (engl. *spinner*). Navedeni elementi uvršteni su u *ConstraintLayout* korijenski element gdje su se definirali atributi koji čine taj element aktivnim (u fokusu) ako je fokus prethodno bio negdje drugdje, poput polja za unos emaila. Pomoću promjene trenutnog fokusa unutar aktivnosti moguće je izvršiti sakrivanje tipkovnice ako je korisnik prethodno imao otvorenu tipkovnicu fokusiranu na određeno polje za unos. Zatim su unutar sučelja dodani preostali elementi. Polja za unos korisničkog emaila i lozinke specifična su po tome što dolaze iz *Material* biblioteke. Riječ je o elementu *TextInputLayout* koji unutar sebe mora sadržavati *TextInputEditText.* Koristeći navedene elemente omogućen je prijenos upute za adekvatan unos informacija, na gornji rub elementa *TextInputLayout* nakon što korisnik započne unos sadržaja u *TextInputEditText* element. Za kreiranje stiliziranog gumba korišten je element *CardView* koji unutar sebe sadrži *TextView* za opis radnje koju gumb započinje. Za potrebe prikazivanja animacije prilikom izvršavanja mrežnih zahtijeva, unutar posebne XML datoteke specificiran je *spinner* kojeg je moguće uvrstiti unutar svake druge XML datoteke sučelja. Svim elementima potrebno je specificirati širinu, visinu, ograničenja, a onima kojima se želi upravljati iz programskog koda, potrebno je definirati jedinstvene identifikatore koristeći *id* atribut. Unutar aktivnosti koja rukuje prijavom korisnika, potrebno je naslijediti *BaseActivity* te implementirati metode *getLayoutResourceId* te *setUpUi*. Deklariranje prezentera slično je onome unutar pozdravne aktivnosti, no moraju se definirati nova sučelja koja odgovaraju trenutnoj uporabi unutar postupka prijave korisnika. Odgovornost prezentera jest da unutar svoje klase sadržava povratni poziv na aktivnost kako bi mogao slati poruke, odnosno naredbe prema aktivnosti. Također je potrebno da prezenter reagira na pritisak

tipke kojeg aktivnost proslijedi u prezenter. Dužnost aktivnosti jest navigirati do zaslona prikaza ponuda ako je prijava uspješna, prikazati grešku ako nije te prikazivanje nedeterminističkog *spinnera*. Stoga su odgovornosti navedene kao deklaracije metoda unutar sučelja *view*. Za validaciju sadržaja unutar polja za unos emaila i lozinke korištena je provjera da postoji barem neki znak jer se od korisnika očekuje da zna svoje podatke za prijavu. Kada je forma ispunjena, omogućen je pritisak tipke za prijavu te odabirom navedene tipke korisnik započinje radnju unutar prezentera koja pokreće radnju u pripadnom scenariju korištenja, a zatim u razini repozitorija.

Registracija korisnika realizirana je korištenjem jedne aktivnosti koja u sebi sadržava *FrameLayout* element u kojeg su uvršteni potrebni fragmenti za uspješno registriranje korisnika. Unutar sučelja aktivnosti definirani su samo nedeterministički *spinner*, te navedeni *FrameLayout*. Za aktivnost registracije, kao i za svaki od fragmenata u postupku registracije definirani su ugovori, te napravljene implementacije istih u prezenterima i aktivnostima. Prvim fragmentom u postupku registracije korisnik unosi osobne podatke poput imena, emaila, lozinke, datuma rođenja, spola te države za koju ga zanimaju ponude. Za svako od polja za unos tekstualnog sadržaja kreiran je *TextWatcher* koji se poziva na svaku promjenu sadržaja unutar tekstualnog elementa za kojeg je vezan. *TextWatcher* dodaje se metodom *addTextChangedListener* te je proslijeđen objekt koji rukuje promjenom sadržaja. Za ovaj fragment korišteni su validatori za svako od polja. Tako primjerice za korisničko ime postoji ograničenje da mora sadržavati vrijednost, za email mora biti ispunjen određeni regularni izraz (engl. *regular expression, regex*), polje za unos lozinke također zahtjeva ispunjenje određenog regularnog izraza, a ponovljena lozinka da je jednaka prvotno unesenoj. Za ostala polja i elemente za unos vrijednosti potrebno je samo postojanje vrijednosti kada se vrijednost promijeni s postavljene proizvoljne vrijednosti. Razlika s obzirom na aktivnost za prijavu korisnika jest postojanje elemenata sučelja kojima su prikazane greške i uspjesi prilikom validacije ispod određenih elemenata za unos tekstualnog sadržaja. Odabir datuma rođenja omogućen je pritiskom na gumb, a pozivanje određene radnje mora se definirati u takozvanom *click listeneru*. *Click listener* mora biti definiran na komponenti koja ima specificiran atribut *id* unutar sučelja pojedine aktivnosti ili fragmenta, a postavljen je metodom *setOnClickListener* kojoj se predaje *lambda* koju će pozvati sustav kada se zaista dogodi klik na određen element sučelja. Unutar *lambde* potrebno je definirati poziv na metode kako bi postavljanje *listenera* bilo što kompaktnije. Unutar metode potrebno je definirati instancu *Calendar* klase te je potrebno definirati varijablu koja predstavlja dijalog u kojem se otvora odabir datuma. Navedena varijabla je tipa *DatePickerDialog*, a za kreiranje dijaloga potrebno je unutar konstruktora proslijediti kontekst, odnosno trenutnu aktivnost u kojoj je otvoren dijalog za odabir datuma, povratni poziv koji je

pozvan nakon uspješnog postavljanja datuma unutar dijaloga te datum koji je odabran kada se dijalog otvori. Nakon inicijalizacije varijable s instancom *DatePickerDialoga,* potrebno je još ograničiti najveći datum koji se može odabrati. Zatim je za pokazivanje dijaloga pozvana metoda *show* na instanci varijable. Nakon što korisnik odabere datum, zatvoren je dijalog te pozvan povratni poziv specificiran u konstruktoru dijaloga. U tom povratnom pozivu validiran je odabir te korisnik ima mogućnost ispunjavanja ostatka forme. Na prvu registracijsku formu također su dodana dva gumba isključiva odabira (engl. *radio button*) kojima korisnik odabire spol. Međusobno isključive gumbe unutar sučelja potrebno je uvrstiti unutar *RadioGroup* elementa te grupi i gumbima dodijeliti identifikatore, veličinu i ograničenja. Na elementu *RadioGroup* također se može odabrati orijentacija te je za razvoj aplikacije odabrana horizontalna orijentacija. Reagiranje na odabir postavljeno je na grupu metodom *setOnCheckedChangeListener* gdje je specificiran povratni poziv. Za proces registracije također je potrebno učitati države unutar padajućeg izbornika koji se na Android platformi naziva *spinner*. Za potrebe izvršavanja slučaja korištenja koristio se kreirani prezenter. U slučaju uspješnog dohvaćanja informacija o državama, fragmentu se signalizira uspjeh te je za popunjavanje sadržaja padajućeg izbornika potrebno definirati adapter koji kreira *view* i pridružuje dobivene podatke kreiranom *viewu*. Za ostvarivanje navedenoga potrebno je kreirati vlastitu klasu koja nasljeđuje *ArrayAdapter* te preopteretiti metode *getView* i *getDropDownView*. Zatim je unutar tih metoda potrebno definirati poziv na metodu koja vraća ili generira traženi element sučelja. Ukoliko je prilikom sistemskog poziva na bilo koju od metoda parametar *view* jednak *null*, koristeći *inflater* generira se sučelje za samo jedan element unutar padajućeg izbornika. Ako parametar *view* sadrži vrijednost, samo se u element padajućeg izbornika zapišu podaci koji prikazuju na sučelju. Za upotpunjavanje rada padajućeg izbornika kreirana je instanca klase adaptera te je ta klasa postavljena kao adapter samoga *spinnera*. Za detekciju odabira pojedine države, također je kreiran povratni poziv.

Uspješnim unosom potrebnih podataka validator omogućuje korisniku odlazak na drugi zaslon gdje odabire lokacije za koje želi vidjeti određene ponude. Podaci uneseni na prvoj registracijskoj formi se pomoću povratnog poziva šalju na aktivnost koja sadrži registracijske fragmente. Povratni poziv na aktivnost specificiran je unutar *onAttach* metode unutar fragmenta gdje je kao parametar dobivena aktivnost koja će udomiti fragment. Tako je za specificiranje varijable povratnog poziva kreirano sučelje pomoću kojega se komunicira s aktivnošću koja implementira navedeno sučelje. Za pohranu privremenih registracijskih podataka korišten je *builder* razvojni obrazac gdje se prilikom završetka popunjavanja registracijske forme kreira konkretna klasa s konstantnim vrijednostima korisničkih podataka. Za novi fragment također je potrebno specificirati sučelja od

kojih je jedno implementirano unutar prezentera, a drugo unutar aktivnosti. Korisničko sučelje ovog fragmenta sastoji se od komponente *RecyclerView*, nedeterminističkog *spinnera* te gumba za odlazak na zadnji zaslon registracije. Komponenta *RecyclerView* služi za prikazivanje elemenata koji imaju jednaku organizaciju sučelja, no prikazuju različite podatke. Bitno svojstvo *RecyclerView* komponente je ponovno korištenje, recikliranje, prethodno kreiranih elemenata sučelja kada nestanu s vidljivog područja zaslona. Tada Android operacijski sustav pomoću *RecyclerView* adaptera prilagođava sadržaj za određeni element liste pomoću njegova indeksa i polja iz kojeg su uzeti podaci. Učitavanje gradova izvršeno je unutar prezentera za drugi registracijski zaslon. Argument za učitavanje gradova proslijeđen je iz aktivnosti registracije na potvrdu ispunjenja prvog registracijskog zaslona. Pošto su prilikom potvrde ispunjenja prvog zaslona podaci poslani putem povratnog poziva aktivnosti, aktivnosti je tako omogućeno postavljanje potrebnih podataka te inicijalizacija novog fragmenta koji je zadužen za učitavanje gradova. Za učitavanje novog fragmenta unutar postojećeg *FrameLayouta* korišteno je proširenje *showFragment* pokazano početkom ovog potpoglavlja. Unutar *onViewCreated* metode novog fragmenta, dohvaća se spremljen identifikator države pomoću kojega se dohvaćaju gradovi za tu državu. Unutar *onViewCreated* metode postavlja se povratni poziv prezentera na fragment kako bi mogao slati poruke fragmentu, te se nakon toga započinje radnja dohvaćanja gradova. Prezenter fragmentu šalje poruku da prikaže nedeterministički *spinner* te sam poziva slučaj korištenja koji rukuje mrežnim zahtjevom te metodama povratnih poziva za uspjeh, odnosno neuspjeh dohvaćanja gradova za određenu državu. U slučaju uspješnog dohvaćanja gradova, mora se pripremiti polje koje će pohraniti dohvaćene gradove te polje koje će pohraniti korisnikove odabrane gradove. Nakon pohrane dohvaćenih gradova, signalizira se fragmentu da prestane prikazivati nedeterministički *spinner* te da se podaci mogu učitati unutar *recycler view* adaptera. Za kreiranje adaptera gradova potrebno je kreirati vlastitu klasu adaptera te naslijediti *RecyclerView.Adapter* klasu kojoj se unutar „<>“ mora proslijediti tip koji popunjava svaki od redaka liste, a koji se opet mora specificirati vlastito definiranom klasom. Tako potpuna definicija tipa kojeg adapter mora naslijediti izgleda *RecyclerView.Adapter<LocationViewHolder>*, a implementiran je i *LocationViewHolder*. Zatim se unutar adaptera mora postaviti polje unutar kojeg se spremaju gradovi koje treba prikazati. Da bi adapter obavljao svoju odgovornost rukovanja elementima sučelja na zahtjev Android operacijskog sustava, potrebno je preopteretiti *onCreateViewHolder*, *onBindViewHolder* te *getItemCount* metode. *GetItemCount* metodu najlakše je implementirati jer mora vratiti samo broj elemenata koje sadrži polje kako bi se odredio broj potrebnih elemenata sučelja za njihovo prikazivanje i recikliranje. Unutar *onCreateViewHolder* metode potrebno je generirati sučelje za pojedini element liste. Element liste također mora imati svoje sučelje koje se

definira XML-om. Korijenski element sučelja za prikaz liste mora imati definiran atribut *layout_height* s vrijednošću *wrap_content* jer u protivnom jedan element liste unutar *recycler viewa* zauzima cijeli zaslon mobilnog uređaja. Unutar tog elementa zatim se uvrštavaju *TextView* i *ImageView* elementi kako bi se prikazao naziv države pri lijevom rubu zaslona te status selekcije na desnoj strani zaslona. Nakon postavljanja ograničenja i ostalih atributa, fokus se može vratiti u adapter *recycler viewa*. Sučelje se generira koristeći *LayoutInflater* te metodu *inflate*. *Inflate* metodi mora se predati *id* resursa koji predstavlja element liste, *parent* parametar koji je dobiven kao parametar *onCreateViewHolder* metode, te zastavica *attachToRoot* koja se mora postaviti na *false*. *AttachToRoot* zastavica mora se postaviti u *false* jer je odgovornost Android operacijskog sustava da doda generirani element unutar *recycler viewa*, a ne programera. Nakon generiranja sučelja, metoda *onCreateViewHolder* mora vratiti instancu *view holdera* definiranog prilikom specificiranja nasljeđivanja adaptera. Stoga se konstruktoru klase *LocationViewHolder* prosljeđuje generirani *view* u kojeg se uvrštavaju podaci. Navedeni *view holder* dobiva se kao parametar metode *onBindViewHolder* gdje se sučelje mora popuniti podacima. Nad dobivenim *view holderom* zatim se poziva definirana metoda *bind* kojom se prosljeđuje određen element liste iz adaptera te referenca na funkciju koja se poziva kada se registrira odabir pojedinog elementa liste unutar *recycler viewa*. Navedenu deklaraciju funkcije potrebno je navesti i u konstruktoru samog adaptera prilikom čijeg kreiranja se prosljeđuje funkcija koja se poziva. Pošto se prosljeđuje funkcija u konstruktoru adaptera te metodi unutar *view holdera* potrebno je definirati funkcijski tip u kojem se pohranjuje referenca na funkciju. U ovome slučaju funkcijski tip jest *(Location, Int) -> Unit*. Iz funkcijskog tipa može se vidjeti da su parametri proslijeđene funkcije *Location*, te nekakav cijeli broj, a povratni tip podatka je *Unit*, što znači da metoda ne vraća nikakvu vrijednost. Unutar adaptera je također definirana metoda *setData* koja prima listu gradova, postavlja listu gradova adaptera na dobivenu listu te obavještava Android operacijski sustav metodom *notifyDataSetChanged* da je potrebno ponovno iscrtavanje *recycler viewa* kojem je adapter pridružen. Nakon poziva *notifyDataSetChanged*, sustav poziva redom metode *getItemCount*, *onCreateViewHolder* te *onBindViewHolder* kako bi ponovno mogao prikazati *recycler view* ali s ažuriranim podacima. *LocationViewHolder* klasa mora naslijediti klasu *RecyclerView.ViewHolder* i proslijediti *view* kojeg dobije kao parametar te se nasljeđuje sučelje *LayoutContainer* kojim se omogućuje pristup elementima sučelja *view holdera* kako bi im se dodijelili podaci te pridružio povratni poziv za selekciju određenog elementa. Implementiranjem adaptera i *view holdera* koji prikazuje vrijednost jednog elementa liste, potrebno je unutar fragmenta za lokacije definirati adapter i upravitelj rasporeda na elementu *recycler view*. Programskim kodom 4.6 prikazana je definicija *recycler viewa*.

```
locations.apply {
    adapter = locationsAdapter
    layoutManager = LinearLayoutManager(context, LinearLayoutManager.VERTICAL, false)
    addItemDecoration(DividerItemDecoration(context, LinearLayoutManager.VERTICAL))
}
```

Programski kod 4.6 Postavljanje recycler view komponente

Nakon što korisnik klikne na jedan element liste unutar *recycler viewa* povratnim pozivom prosljeđuju se grad i pozicija prezenteru da na temelju tih informacija odabrani grad pohrani u svoju listu odabranih gradova. Prezenter rukuje odabirom najviše tri grada te uklanjanjem odabira. Za svaki odabir i uklanjanje odabira potrebno je ponovno iscrtavanje elementa *recycler viewa* kako bi se korisniku vizualno reprezentirao njegov odabir. Odabirom jednog grada omogućuje mu se navigacija na zadnji zaslon registracije. Nakon odabira nastavka procesa povratnim pozivom prosljeđuju se odabrani gradovi do aktivnosti te se provodi tranzicija do zadnjeg registracijskog fragmenta gdje korisnik odabire kategorije za koje je zainteresiran. Postupak razvoja korisničkog sučelja, sučelja i implementacija sučelja za prezenter i *view* jednaki su razvoju drugog registracijskog zaslona uz uklanjanje ograničenja najvećeg mogućeg odabira stavaka iz liste. Za završavanje postupka registracije potrebno je odabrati barem jednu kategoriju te se pritiskom na tipku za registraciju šalju zadnji podaci na aktivnost koja je udomljavala fragmente te kreira konkretan tip korisničkog registracijskog zahtjeva te se poziva prezenter aktivnosti da započne scenarij registracije. U slučaju uspješne registracije, korisniku se prikazuje *toast* poruka te ga se preusmjerava na zaslon za prijavu.

Nakon prijave korisniku se otvara centralna aktivnost aplikacije u kojoj se učitavaju preostali fragmenti. Sučelje ove aktivnosti kreirano je koristeći *DrawerLayout* element kao korijenski element sučelja. Karakteristika tog elementa jest da može udomiti najviše jedan dodatni element sučelja uz obavezni *NavigationView*. *NavigationView* zadužen je za učitavanje resursa izbornika iz *menu* mape koji se prikazuje kada korisnik odabere ikonu izbornika ili kada napravi gestu povlačenja prsta od lijevog ruba zaslona. Dodatni element sučelja uobičajeno je *ConstraintLayout* unutar kojega se zatim mogu uvrštavati hijerarhije elemenata. *ConstraintLayout* moguće je postaviti da zauzima cijelu površinu zaslona postavljanjem atributa *layout_width* i *layout_height* na *match_parent* te se isti atributi moraju primijeniti i za korijenski element. Unutar *ConstraintLayouta* specificira se element *Toolbar* koji prikazuje ikonu za otvaranje izbornika, naslov fragmenta koji je trenutno prikazan na zaslonu te dodatne ikone ovisno o prikazanom fragmentu. Dodatno uz element *Toolbar* uvršten je i *FrameLayout* kojemu su ograničenja postavljena da zauzima preostali raspoloživi prostor zaslona uređaja. Također se koristi za udomljavanje fragmenata pomoću izvršavanja transakcija. Kako bi se unutar glavne aktivnosti mogao otvarati izbornik, potrebno je pomoću metode *setSupportActionBar* postaviti kreirani

Toolbar kao alatnu traku aktivnosti. Zatim je potrebno definirati varijablu koja pohranjuje objekt kreiran klasom *ActionBarDrawerToggle*. Konstruktoru te klase potrebno je predati aktivnost unutar koje se pokazuje izbornik, te *toolbar*. Navedena klasa unutar predanog *toolbara* uvrštava gumb te pokreće i zaustavlja animacije na otvaranje ili zatvaranje izbornika. Zatim je na *DrawerLayout* potrebno metodom *addDrawerListener* proslijediti definiranu varijablu te sinkronizirati stanje izbornika pozivom *syncState* na kreiranoj varijabli. Potrebno je zatim na *NavigationView* pomoću metode *setNavigationItemSelectedListener* proslijediti parametar *this* što znači da trenutna aktivnost prima informacije o odabranim stavkama unutar izbornika. Kako bi se to ostvarilo, potrebno je naslijediti i implementirati sučelje *NavigationView.OnNavigationItemSelectedListener* unutar aktivnosti. Potrebno je implementirati metodu *onNavigationItemSelected* s parametrom tipa *MenuItem* koji predstavlja odabranu stavku unutar izbornika. Unutar te metode vrše se transakcije na preostale fragmente unutar aplikacije.

4.5 Pregled razvoja mobilne aplikacije koristeći programski jezik Swift

4.5.1 Upravitelji biblioteka

Rukovanje bibliotekama za razvoj iOS aplikacija uvelike se razlikuje od onoga na Android platformi. Na Android platformi ovisnosti su centralizirane dok kod iOS platforme to ne mora biti slučaj. Za razvoj iOS mobilnih aplikacija postoji nekolicina takozvanih upravitelja paketa (engl. *package manager*) koji upravljaju ovisnostima, vanjskim bibliotekama projekta. Cilj upravitelja paketa jest ukloniti potrebu za ručnim upravljanjem ovisnostima u vidu ažuriranja, izgradnje i integriranja u vlastiti projekt. Uobičajeno je upravljati ovisnostima u par naredbi u terminalu. Tako su najpoznatiji upravitelji paketa *Cocoa Pods, Carthage* i *Swift Package Manager.*

Prema [9], upravitelj paketa *Carthage* daje korisniku na odgovornost postavljanje ovisnosti bez modifikacije projektnih datoteka i postavke izgradnje projekta poput *Cocoa Podsa*. *Carthage* predstavlja decentraliziran upravitelj ovisnostima. Razlika u usporedbi s centraliziranim upraviteljem ovisnostima jest u postojanju repozitorija koji programerima omogućava pregled raspoloživih biblioteka koje mogu koristiti [9]. Korištenjem *Carthage* odgovornost pronalaska adekvatnih biblioteka prepuštena je programerima te voditeljima projekta smanjuje potrebu održavanja dodatnog resursa za kojeg tvrde da je izvor centralnog kvara. Pošto u današnje vrijeme postoje usluge oblaka računarstva, omogućena je redundantnost centraliziranih upravitelja paketa. U slučaju da platforme za udomljavanje koda poput *GitLaba, GitHuba, Bitbucketa* i drugih ne funkcioniraju, ne bi funkcionirao ni centralizirani ni decentralizirani upravitelj paketa. *Carthage* za postavljanje ovisnosti u projekt zahtjeva mnogo ručnog postavljanja uz dio automatizacije

posla. Stoga je trenutno skupina korisnika mala te je koriste entuzijasti jer postoje jednostavniji načini da se omogući unošenje biblioteka. Dodatni nedostatak *Carthage* upravitelja paketa jest što ga ne podržava svaka biblioteka.

Swift Package Manager direktno je rukovanje bibliotekama omogućeno unutar Swift programskog jezika. Omogućava automatsko preuzimanje i povezivanje ovisnosti [65] kako bi se ubrzao razvoj aplikacija. Pošto je vezan uz programski jezik Swift moguće ga je koristiti na inačicama Linuxa i MacOS-a. Za konfiguriranje ovisnosti koristi *Package.swift* datoteku koja u sebi mora sadržavati uvezen *PackageDescription* modul. Ovisnosti je također moguće definirati i iz sučelja razvojnog okruženja gdje se treba specificirati URL do biblioteke te inačica koja se želi koristiti prilikom razvoja. Za više informacija potrebno je konzultirati [65] i [66].

Za izradu aplikacije korišten je *Cocoa Pods* upravitelj paketa. *Cocoa Pods* predstavlja centraliziran upravitelj paketa jer sadržava repozitorij registriranih biblioteka koje se mogu koristiti za razvoj aplikacija [67]. Cilj *Cocoa Podsa* jest omogućiti laku interakciju, otkrivanje i dodavanje novih biblioteka u repozitorij. *Cocoa Pods* pisan je jezikom Ruby te se istim jezikom opisuje datoteka u koju programer navodi potrebne biblioteke te datoteka kojom autori biblioteka obavještavaju centralni repozitorij da se dogodilo postavljanje nove ili ažuriranje postojeće biblioteke. Na [68] može se vidjeti prikaz centralnog repozitorija koji sadržava specifikacije raspoloživih biblioteka. Specifikacije se nazivaju *Podspec* ili *Spec*, [69], a sadržavaju informacije o nazivu, inačici, kratki opis te poveznicu na kojoj se nalazi izvorni kod biblioteke. Također sadržava informacije o licenci koja specificira način korištenja biblioteke, autoru, minimalne zahtjeve za platformama gdje se biblioteka može koristiti te inačice programskog jezika Swift u kojima se mogu koristiti. Primjer *Podspec* datoteke koja se koristi na glavnom repozitoriju, dan je prilogom 2 preuzetog iz [70].

Za korištenje *Cocoa Pods* unutar vlastitog projekta potrebno je putem terminala i gem upravitelja ovisnosti za macOS platformu u terminal upisati: *sudo gem install cocoapods*. Nakon instalacije potrebno je izvršiti komandu *pod setup* kojom se preuzima definicija *Podspec* datoteka s glavnog repozitorija kako bi se ubrzao proces traženja biblioteka. Nakon navedenih radnji *Cocoa Pods* spreman je za uvoz ovisnosti u razne projekte. Kako bi se u određenom projektu definirale biblioteke potrebno je kreirati novi projekt unutar Xcode razvojnog okruženja te otvoriti korijenski direktorij projekta unutar terminala. Zatim je potrebno utipkati *pod init* naredbu kojom se kreira *Podfile* kojim se specificiraju tražene biblioteke. Za pretragu biblioteka potrebno je koristiti tražilicu dostupnu na [8] ili koristiti terminal s naredbom *pod search <riječ_za_pretragu>*. Nakon pronalaska biblioteke potrebno je izvršiti registraciju ovisnosti gdje je inačica biblioteke proizvoljna. Ako inačica nije specificirana uvijek se preuzima najnovija inačica biblioteke. Tako

zapis za biblioteku Alamofire u *Podfile* datoteci izgleda: *pod 'Alamofire', '~> 5.0.1'*. Nakon definiranja svih potrebnih biblioteka za projekt, potrebno je pohraniti *Podfile* datoteku te unutar terminala izvršiti komandu *pod install* kojom se, prema [71], kreira *Podfile.lock* datoteka, kreira ili ažurira radni prostor projekta (engl. *workspace*), kreira dodatan projekt koji sadržava uvezene biblioteke, a kojeg je potrebno specificirati u prethodno navedenom radnom prostoru projekta. Sadržaj datoteke dan je slikom 4.6.

```
platform :ios, '12.0'

target 'cityalarm-swift' do
  use_frameworks!
  pod 'Alamofire', '~> 5.0.1'
  pod 'Kingfisher', '~> 5.13.0'
  pod 'Swinject'
  pod 'SwinjectStoryboard'
  pod 'Toast-Swift', '~> 5.0.1'
end
```

Slika 4.6 Prikaz Podfile datoteke

Od iznimne je važnosti nakon izvršavanja naredbe *pod install* pokretati Xcode projekt s ekstenzijom *.xcworkspace* jer sada ta datoteka sadržava kreirani projekt i projekt koji sadržava Cocoa Pods biblioteke o kojima ovisi kreirani projekt. U protivnom Xcode obavještava korisnika da ne može pronaći biblioteke o kojima ovisi korisnički projekt te se ne može izvršiti gradnja i pokretanje projekta. *Podfile.lock* datoteku izgrađuje *Cocoa Pods* sustav tijekom prvog injektiranja ovisnosti. Stoga je u slučaju korištenja alata za verzioniranje koda nužno da su *Podfile* i *Podfile.lock* uključene u taj alat jer je nužno instalirati iste ovisnosti na drugom računalu. Moguće je u sustav za verzioniranje koda uključiti direktorij u kojeg je *Cocoa Pods* preuzeo biblioteke kako bi se osiguralo da svaki sudionik projekta ima istu inačicu biblioteka te da je ne može slučajno promijeniti, no to je rjeđa praksa zbog ograničenog prostora na poslužitelju za verzioniranje koda. Prema [8], pokretanjem *pod install* na drugom računalu s dostupnim *Podfile* i *Podfile.lock* datotekama, *Cocoa Pods* poštuje ovisnosti unutar *Podfile.lock* datoteke neovisno o tome postoje li zapisi novijih biblioteka unutar centralnog repozitorija. *Cocoa Pods* preuzima i ugrađuje novije ovisnosti tek kada se promijene biblioteke ili inačice biblioteka unutar *Podfile* datoteke ili ako se u terminalu pokrene naredba *pod update*. *Cocoa Pods* osigurava instalaciju točnih biblioteka na drugim računalima provjerom sadržaja datoteke *Podfile.lock* koja sadržava *hash* vrijednosti *Podfile* datoteke kao i svakog *Podspeca*. Ukoliko se prilikom pokretanja „*pod install“ hash* trenutne *Podfile* datoteke i *hash* pohranjen unutar *Podfile.lock* datoteke ne podudaraju, *Cocoa*

Pods instalira one biblioteke i inačice specificirane u *Podfile* datoteci. Primjer *Podfile.lock* datoteke dan je prilogom 3.

Za označavanje inačica biblioteka koristi se takozvano semantičko verzioniranje (engl. *semantic versioning*) koje sadržava tri znamenke [72] [73]. Krajnje lijeva znamenka označava veću inačicu (engl. *major*), srednja manju inačicu (engl. *minor*) te krajnje desna znamenka inačicu zakrpe grešaka (engl. *patch*). Ako se novom inačicom uklanjaju greške unutar koda biblioteke, uvećava se broj zakrpe. Ako se doda nova funkcionalnost koja je kompatibilna s postojećim kodom, uvećava se manja inačica. Velika inačica uvećava se kada se u biblioteku dodaju ili promjene postojeće funkcionalnosti, odnosno API.

Tako primjerice *Cocoa Pods*, prema [74], podržava sljedeće načine definiranja inačica biblioteka unutar *Podfile* datoteke:

1. Izostavljanje inačice – u tom slučaju koristi se najnovija inačica biblioteke dostupna na centralnom repozitoriju.
2. Specificiranje inačice – nakon deklaracije naziva biblioteke specificira se točna inačica biblioteke koja se želi koristiti.
3. Korištenje relacijskih operatora – Koriste se navedeni relacijski operatori za specificiranje intervala inačica biblioteka ‘<’, ‘<=’, ‘>’, ‘>=’.
4. Korištenje tzv. *optimističnog* operatora – operator koji izgleda ‘~>’, a ima funkciju ograničiti najveću dostupnu inačicu biblioteke. Postoje tri načina uporabe od kojih jedan rezultira jednakim učinkom kao izostavljanje inačice. Načini uporabe glase:
    a. '~> 5.13.0' – pozivom *pod install* preuzima se i konfigurira inačica biblioteke sve do 5.14 ne uključujući 5.14.
    b. '~> 5.13' – pozivom *pod install* preuzima se i konfigurira inačica biblioteke sve do 6.0 ne uključujući 6.0.
    c. '~> 5' pozivom *pod install* preuzima se i konfigurira biblioteka s najvećom dostupnom inačicom.

Iz priloga 3 može se vidjeti i tranzitivna ovisnost dvaju biblioteka, a riječ je o biblioteci *SwinjectStoryboard* i biblioteci *Kingfisher*. Pod stavkom *dependencies* redom stoje ovisnosti koje su navedene u *Podfile* datoteci, a stavkom *Pods* nabrojane su ovisnosti i njihove tranzitivne ovisnosti ako postoje. Tako je u ovom slučaju za biblioteku *Kingfisher 5.13.0* potrebna dodatna biblioteka *Kingfisher/Core 5.13.0*, a za *SwinjectStoryboard 2.0.0* postojanje *Swinject* biblioteke do inačice 3.0.

Za razvoj iOS aplikacije korištene su biblioteke *Alamofire*, *Swinject*, *Kingfisher*, *Toast-Swift* koje su pojašnjene u nastavku.

4.5.2 Biblioteka Alamofire

Biblioteka Alamofire [75] zadužena je za apstrahiranje *URLSessiona*, Appleovog okvira za izvršavanje HTTP zahtjeva kojeg je moguće pokretati na macOS, iOS, tvOS i watchOS platformama. Alamofire omogućava izvršavanje zahtjeva, preuzimanje i slanje datoteka kao i rukovanje priručnom memorijom. Također omogućava rukovanjem autorizacijom, serijalizacijom i deserijalizacijom različitih tipova podatka. Prema [76], glavna funkcionalnost biblioteke dostupna je preko globalne varijable *AF* tipa *Session*. *Session* klasa direktno rukuje zahtjevima, njihovim raspoređivanjem, preusmjeravanjem i pohranom u priručnu memoriju. Pošto postoji globalna varijabla koja prati stanje svih trenutnih zahtjeva omogućeno je prekidanje svih nedovršenih zahtjeva. Prilikom kreiranja zahtjeva uvelike pomažu proširenja nad tipom *URLRequest* pomoću kojega se modelira API u sljedećem potpoglavlju. Alamofire također omogućava definiranje vlastitih validatora odgovora poslužitelja prilikom definiranja zahtjeva. Validator koji prvi registrira grešku signalizira pojavu greške ostatku lanca te se greška odmah propagira do mjesta za rukovanje odgovorom poslužitelja. Za dobivanje korisnog rezultata postoji nekoliko metoda poput *response*, *responseData*, *responseJSON* te *responseDecodable*. Unutar navedenih metoda, kao zadnji parametar definiraju se lambde, odnosno *closure-i* koji zatim rukuju dobivenim odgovorom tipa *DataResponse<Success, AFError>*.

4.5.3 Biblioteka Swinject

Biblioteka Swinject [77] koristi se za injektiranje ovisnosti unutar iOS aplikacija. Dobar dodatak je također SwinjectStoryboard biblioteka koja omogućava injektiranje ovisnosti u upravljače pogleda (engl. *view controller*) kreiranih iz tzv. *Storyboard* sustava za dizajn sučelja unutar Xcode razvojnog okruženja. Swinject također funkcionira tako da je potrebno definirati sučelje, odnosno, protokol na iOS platformi pomoću kojega se zahtijeva određena funkcionalnost od tipova koji implementiraju taj protokol. Prije korištenja ovisnosti unutar koda, potrebno je izvršiti registraciju protokola i objekata koji se kreiraju prilikom rezolucije ovisnosti. Također podržava kreiranje *singletonom* ili *factoryjem*. Za potrebe injektiranja unutar upravljača pogleda potrebno je izvršiti injektiranje pomoću svojstva, a ne konstruktora jer ne postoji direktan pristup konstruktoru upravljaču pogleda.

4.5.4 Biblioteka Kingfisher

Biblioteka Kingfisher [78] omogućuje asinkrono preuzimanje slika i pohranu u priručnu memoriju kako bi se povećala efikasnost mrežnog prometa. Za učitavanje slika potrebno je specificirati URL koji može predstavljati lokaciju na internetu ili u lokalnom spremniku. Također omogućava manipuliranje slikom prije nego se uvrsti u element sučelja. Za česte slučajeve korištenja, poput uvrštavanja slike unutar elementa *UIImageView*, postoje spremna proširenja kojima se u jednoj liniji koda može manipulirati slikama.

4.5.5 Biblioteka Toast-Swift

Bibliotekom Toast-Swift omogućeno je jednostavno prikazivanje *Toast* poruka na iOS platformi. Riječ je o obavijestima koje se pojavljuju pri dnu zaslona bez omogućene korisničke interakcije. Dodatna prednost koja ubrzava razvoj aplikacije jest prikazivanje jednostavnog indikatora aktivnosti povrh sučelja što je iznimno korisno prilikom izvršavanja mrežnih zahtjeva. Biblioteka je dostupna na [79].

4.5.6 Biblioteka SWRevealController

SWRevealViewController [80] ovisnost je kojom se ostvaruje segment *Material* dizajna koji se na Android platformi ostvaruje pomoću Swipe Refresh Layouta. SWRevealViewController klasom omogućava se definiranje prednjeg i pozadinskog upravljača pogleda. Pozadinski upravljač pogleda služi za iscrtavanje izbornika i odabir željenih radnji, a prednji upravljač pogleda služi za učitavanje odabranih upravljača. Kako bi se postigla jednaka funkcionalnost poput one na Android platformi u funkcionalnosti povlačenja prsta s lijevog ruba zaslona za otvaranje izbornika, na iOS platformi mora se kreirati posebni prepoznavatelj radnji (engl. *gesture recognizer*) koji se dodaje na upravljače pogleda na kojima je potrebno prikazati izbornik.

4.6 Razvoj iOS aplikacije

4.6.1 Postavljanje i karakteristike iOS projekta

Početak razvoja iOS aplikacije započinje otvaranjem razvojnog okruženja Xcode. Kreiranje projekta može se izvršiti s prikazanog pozdravnog zaslona odabirom *Create a new Xcode project* ili odabirom *File->New->Project*. Nakon odabira otvara se prozor na središtu zaslona kojim se odabire ciljana platforma aplikacije te odabir preddefiniranih obrazaca za projekt. U prozoru se odabire iOS operacijski sustav te *Single View App* te odabere *Next*. Otvara se novi prozor u kojemu je potrebno unijeti informacije o projektu. Potrebno je unijeti naziv aplikacije pod poljem *Product*

Name, osobu koja je zaslužena za kreiranje aplikacije pod padajućim izbornikom *Team* nije nužno unijeti, *Organization Name* polje može ostati prazno, a nužno je ispuniti polje *Organization Identifier*. Polje *Organization Identifier* ima istu svrhu kao i na Android platformi te je potrebno zadati reverzni URL koji jedinstveno identificira kreiranu aplikaciju. Također je kao razvojni programski jezik potrebno odabrati *Swift*, a pod stavkom *User Interface* potrebno je odabrati *Storyboard* te odabrati *Next*. Zatim je potrebno na otvorenom prozoru odabrati lokaciju na koju se pohranjuje projekt te opcionalno odabrati *Source Control* da se omogući verzioniranje koda. Uključivanje verzioniranja koda je opcionalno jer se navedeno može ostvariti manualnim putem. Nakon kreiranja projekta i zatvaranja prozora, potrebno je uputiti se u terminal te navigirati do direktorija u kojem se nalazi projekt. Ondje je potrebno izvršiti komandu *pod init* kako bi se pripremilo korištenje *CocoaPods* upravitelja ovisnostima. U *Podfile* datoteku potrebno je dodati ovisnosti specificirane slikom 4.6, pohraniti datoteku te unutar terminala, u korijenskom direktoriju projekta, pokrenuti komandu *pod install*. *Pod install* naredbom pronalaze se tražene ovisnosti na centralnom *CocoaPods* repozitoriju te ih se preuzima. Preuzimanjem svih specificiranih biblioteka *CocoaPods* kreira zaseban projekt koji sadržava sve ovisnosti te modificira prethodno kreirani projekt da ovisi o projektu koji sadržava sve ovisnosti što rezultira stvaranjem nove datoteke unutar projekta sa ekstenzijom *.xcworkspace*. Kako bi se aplikacija mogla razvijati s novim ovisnostima, potrebno je unutar Xcode sučelja odabrati *File->Open* te u prozoru pronaći datoteku sa ekstenzijom *.xcworkspace* te je otvoriti. Također je projekt potrebno otvarati na navedeni način od trenutka unosa ovisnosti *CocoaPods* sustavom. U prozoru Xcode sučelja može se vidjeti da postoje dva projekta. Projekt s bibliotekama ne bi se trebao ručno mijenjati te je sve promjene potrebno vršiti na prvotnom projektu. Odabirom projektne datoteke u središnjem prozoru otvaraju se karakteristike projekta i karakteristike *build targeta*. Tako se pod stavkom *build targeta,* pod karticom *General* nalaze postavke vezane za identitet aplikacije, informacije o tipu uređaja na kojima se aplikacija pokreće, ikonama i početnom zaslonu aplikacije te korišteni razvojni okviri i biblioteke. Pod stavkom identitet moguće je promijeniti naziv aplikacije koji se prikazuje korisniku na mobilnom uređaju, moguće je promijeniti naziv korijenskog direktorija projekta te semantičku inačicu aplikacije. Informacijama o tipu uređaja odabire se minimalna potrebna inačica iOS operacijskog sustava kako bi se aplikacija mogla pokrenuti, podržani uređaji koji uključuju iPhone ili iPad, datoteku sučelja koja se treba pokazati nakon pokretanja aplikacije, koje su podržane orijentacije uređaja te stavke poput stila statusne trake aplikacije te njezino skrivanje. Razlika spram Android platforme jest što se na iOS platformi podrazumijevana orijentacija može postaviti na centralnom mjestu, te ako određen upravljač pogleda zahtijeva rotaciju, može implementirati povratni poziv sustavskih metoda. Aplikacijska

ikona postavlja se na sliku sadržanu unutar *Assets.xcassets* direktorija. Također je moguće postaviti pozdravni zaslon koji se mora kreirati koristeći *Storyboard* sustav za razvoj sučelja aplikacije. Ako su prije otvaranja zaslona o karakteristikama projekta uvrštene ovisnosti koristeći *CocoaPods,* pod razvojnim okvirima vidljiv je uključen *Pods* razvojni okvir o kojem ovisi kreirani projekt. Ostalim karticama *build targeta* postavljaju se karakteristike potpisivanja aplikacije, informacija o detaljima projekta te postavke o izgradnji projekta. Potpisivanjem se upravlja certifikatima nužnim za potpisivanje aplikacije kako bi mogla biti spremna za postavljanje na *AppStore*, distribucijski centar aplikacija na Apple operacijskim sustavima. Za razvoj ove aplikacije nije bilo potrebno kreirati i uvesti certifikat za potpisivanje aplikacije. Detalje o projektu moguće je također vidjeti unutar *Info.plist* datoteke. Postavke odabrane na prethodnim zaslonima moguće je pronaći u mnoštvu drugih opcija koje je moguće promijeniti unutar postavki korištenih za izgradnju projekta, odnosno aplikacije. Važna postavka koja je ondje promijenjena jest inačica Swift programskog jezika koja je postavljena na inačicu 5 kako bi se mogla koristiti novija inačica Alamofire biblioteke koja ovisi o navedenoj inačici Swifta. Karakteristikama projekta moguće je promijeniti postavke koje se primjenjuju prilikom kreiranja novih *targeta* te je također moguće specificirati lokalizaciju znakovnih nizova korištenih unutar aplikacije.

Kreiranjem novog projekta generira se nekoliko osnovnih datoteka kako bi se omogućio razvoj aplikacija. Riječ je o *AppDelegate*, *ViewController*, *Main.storyboard*, *Assets.xcassets*, *LaunchScreen.storyboard* te *Info.plist* datotekama. *AppDelegate* klasa prema [81] predstavlja centralnu točku svake iOS aplikacije koja mora naslijediti i implementirati metodu *UIApplicationDelegate* protokola. *Singleton* instancu *UIApplication* klase kreira sustav prilikom pokretanja aplikacije te je toj instanci moguće pristupiti preko *shared* svojstva na *UIApplication* klasi [82]. Navedena klasa informira delegata, *AppDelegate,* na određene radnje poput pokretanja aplikacije te zaustavljanja aplikacije. Također se savjetuje da se instancom *UIApplication* klase treba provjeriti može li se pomoću URL-a otvoriti druga aplikacija kako bi se upotpunilo iskustvo korisnika, otvaranje drugih aplikacija, produljenje izvršavanja aplikacije u pozadini prilikom zatvaranja aplikacije te registriranjem na mrežni izvor notifikacija. Prema [81], preporuka je da se *AppDelegate* klasa koristi za inicijalizaciju bitnih podatkovnih struktura unutar aplikacije, reagiranje na događaje koji nisu usmjereni na određene upravljače pogleda, registriranje na razne usluge ovisno o tipu aplikacije te odabir scene (engl. *scene*) koja će biti prikazana na zaslonu mobilnog uređaja. Scena predstavlja korisničko sučelje koje treba pokazati na zaslonu mobilnog uređaja, a koje je kreirano kroz *Storyboard* alat na iOS mobilnoj platformi. Ekvivalent navedenog na Android platformi predstavlja sučelje aktivnosti ili fragmenata.

4.6.2 Opis glavnih komponenti sučelja iOS aplikacija

Upravljači pogleda na iOS platformi predstavljaju osnovni gradivni element svake aplikacije. Svaka scena mora imati pridružen upravljač pogleda ili kraće upravljač, koji rukuje učitavanjem, prikazivanjem te uništavanjem scene. Također su upravljači pogleda zaduženi za reagiranje na korisničke radnje, iniciranje obrade te na kraju prikaza rezultata korisniku. Koristeći arhitekturu MVP, upravljač pogleda promatra se kao razina pogleda te ne smije sadržavati kompleksnu logiku. Zadaća upravljača treba biti iniciranje radnji na razini prezentera, te prikazivanje rezultata dobivenih iz prezentera. Stoga se implementira *view* protokol pomoću kojega prezenter šalje poruke upravljaču putem povratnog poziva. iOS platforma također specificira metode životnog vijeka svakog upravljača unutar aplikacije slično Android platformi. Tako se iOS platformom definira pet metoda povratnih poziva koje poziva iOS operacijski sustav tijekom rada aplikacije. Riječ je o metodama *viewDidLoad*, *viewWillAppear*, *viewDidAppear*, *viewWillDisappear* te *viewDidDisappear* [83]. Bitno je napomenuti da nije potrebno implementirati niti jednu od navedenih metoda kako bi se pokrenula scena određenog upravljača i korisniku omogućila interakcija sa scenom. Korisnik bi mogao unositi tekst unutar polja za unos, no ako je potrebno omogućiti nekakvu validaciju unosa ili početak pretrage sadržaja, tu radnju se mora postaviti pokretanjem upravljača te je potrebno implementirati barem *viewDidLoad* metodu. Životni vijek upravljača može se vidjeti na [83].

Sustav v*iewDidLoad* metodu poziva kada se kreira i učita scena kreirana kroz *Storyboard.* Prije poziva *viewDidLoad* sustav osigurava da je svaki *outlet* iz scene prema upravljaču postavljen i spreman za korištenje. *Outletom* se omogućava programski pristup elementima sučelja tijekom rada aplikacije. Tako je moguće pristupiti svojstvima svakog elementa i mijenjati ih na proizvoljne vrijednosti. *ViewDidLoad* metoda poziva se jedanput nakon kreiranja sučelja scene, a navedeno se uobičajeno događa kada neki kod ili sustav pristupi *view* svojstvu upravljača. *ViewWillAppear* poziva se prije nego se kreirani *view* doda na hijerarhiju pogleda aplikacije. Preporuka Applea jest korištenje metode za iniciranje obrade koja se treba izvršiti prije pojave *viewa* no metodom se ne može osigurati da će *view* zapravo biti prikazan na zaslonu mobilnog uređaja. *ViewDidAppear* metodom signalizira se da je *view* uspješno dodan na hijerarhiju pogleda te se isto ne može osigurati da će biti prikazan na zaslonu. Neposredno prije uklanjanja *viewa* s hijerarhije pogleda aplikacije sustav poziva metodu *viewWillDisappear* kojom se trebaju pohraniti promjene na sučelju. *ViewDidDisapper* poziva se nakon uklanjanja *viewa* iz hijerarhije pogleda.

Storyboard alat omogućava kreiranje sučelja iOS mobilnih aplikacija. Predstavlja u potpunosti grafičko sučelje kojim se elementi dodaju na scenu te se tim elementima omogućava povezivanje

s kodom kako bi se upotpunilo korisničko iskustvo. Otvaranjem nekog *Storyboarda* rezultira otvaranjem cijelog novog prozora u centralnom sučelju Xcodea. Tako je moguće s lijeve strane otvorenog prozora vidjeti trenutne scene koje se nalaze na sučelju te način njihova povezivanja. Preporuka jest koristiti određeni *Storyboard* kako bi se objedinilo korisničko iskustvo za nekakav zadatak, primjerice registracije i prijave u aplikaciju. Tako se na najvišem hijerarhijskom sloju *Storyboarda* uobičajeno prikazuju upravljači, navigacijski upravljači (engl. *navigation controller*), kartični upravljači (engl. *tab controller*) te *split view* upravljači. Upravljači udomljavaju sučelja scena, a ostalim navedenim upravljačima funkcija je prikazati osnovne tipove upravljača na neki od standardiziranih načina. Navigacijski upravljač izmjenjuje upravljače jedno povrh drugih koristeći navigacijski stog, kartični upravljači pri dnu ili vrhu zaslona prikazuju nekoliko kartica kojima korisnik odabire odredišni upravljač. *Split view upravljači* služe za prikaz dva upravljača od kojih je jedan glavni (engl. *master*), a drugi detalj (engl. *detail*) koji se prikazuje odabirom određenog elementa na sučelju glavnog upravljača. Na scenu određenog upravljača dodavanje elemenata odrađuje se koristeći biblioteku elemenata (engl. *library*) unutar Xcode sučelja. Bibliotekom elemenata omogućava se dodavanje labela, tipki, gumba isključiva odabira, polja za unos tekstualnog sadržaja i mnoge druge komponente. Bibliotekom elemenata omogućeno je dodavanje ikona i slika definiranih unutar kataloga resursa unutar projekta. Prevlačenjem jednog od takvih elemenata, automatski se ugrađuju unutar elementa *ImageView*. Za pozicioniranje elemenata unutar scene nužno je koristiti *Auto Layout* sustav kojim se definira pozicija i veličina elemenata unutar scene. Dodavanje ograničenja ostvaruje se pritiskom tipke *ctrl* i povlačenjem miša dok je pritisnuta lijeva tipka miša do komponente u odnosu na koju se želi postaviti horizontalni ili vertikalni razmak. Također se može navedena radnja napraviti na samom elementu te se tada omogućava definiranje širine, visine te omjera veličina određenog elementa [84]. Svaka scena koja se nalazi unutar *Storyboarda* mora imati pridruženu klasu upravljača koji rukuje njenim prikazom na zaslonu mobilnog uređaja. Potrebno je otvoriti *inspector* prozor prečacem *cmd + option + 0*. Zatim je unutar tog prozora potrebno pronaći karticu *identity inspector* te pod *class* unijeti naziv klase koja će rukovati prikazom scene. Odlaskom do *connections inspectora* omogućava se pregled svih postavljenih *outleta*, radnji (engl. *action*) te prijelaza (engl. *segue*). Budući da na iOS platformi ne postoje identifikatori elemenata sučelja kao na Android platformi pomoću kojih se pristupa elementima, mijenjaju njihova svojstva, te radnje koje iniciraju, potrebno je definirati takozvane *outlete* i radnje. Prema [85], *outlet* je svojstvo klase koje na sebi ima anotaciju *@IBOutlet*. Navedena anotacija ne sadržava vrijednost te služi samo Xcodeu da odredi koje je svojstvo zaista *outlet*. Pomoću *outleta* dobiva se referenca na element scene kojim se zatim može manipulirati kroz mijenjanje svojstva ili pozivom metoda. Radnjom, prema [86], specificira

se dio koda upravljača koji se izvršava nakon neke korisnikove radnje unutar scene. Radnju se označava posebnom anotacijom *@IBAction.* Za dodavanje *outleta* i radnje unutar upravljača potrebno je otvoriti *Storyboard* i pomoćni uređivač (engl. *assistant editor*) koji pronalazi adekvatnu klasu upravljača ovisno o sceni koja je odabrana. Nakon otvaranja, potrebno je pritisnuti *ctrl* te lijevu tipku miša na proizvoljni element te povući miš do pomoćnog uređivača teksta gdje se zatim otvara novi prozor. U tom prozoru moguće je odabrati vrstu veze koja će se ostvariti između scene i upravljača kao i naziv radnje ili *outleta*. Razlika prilikom dodavanja *outleta* ili radnje jest što se prilikom definiranja *outleta* specificira tip svojstva, a kod definiranja radnje može se dodatno definirati element koji šalje radnju te događaj na kojem će radnja biti izvedena. Dodavanjem radnje direktno iz *Storyboarda* eliminira se ručno definiranje radnje metodom *addTarget* na određeni *outlet.* U slučaju da se želi na takav način definirati radnja, potrebno je unutar metode *addTartget* predati tri parametra. Prvi parametar definira objekt koji prima poruku, drugi parametar mora sadržavati vrijednost te se sastoji od selektora metode koja se poziva na pojavu događaja, te kao treći parametar mora se definirati događaj koji pokreće radnju. Drugi parametar, odnosno referenca na funkciju, mora se definirati unutar *#selector()* metode kojom se osigurava da se specificirana metoda može pronaći i pokrenuti tijekom rada aplikacije. Metoda koju se referencira mora ispred definicije funkcije sadržavati anotaciju *@objc.* Trećim parametrom definira se slučaj enumeracije na čiji se događaj želi pokrenuti radnja. Neki od slučajeva enumeracije su reagiranje na pritisak, reagiranje na podizanje prsta, povlačenje prsta itd.

Za izmjenu trenutno prikazanog upravljača na zaslonu mobilnog uređaja koriste se takozvani prijelazi. Prema [87], izvor radnje za prijelaz predstavlja gumb, korisnikova radnja ili nekakav događaj unutar aplikacije. Kreiranje prijelaza odrađuje se odabirom komponente koja izvršava prijelaz te držeći *ctrl* i lijevu tipku miša odabire odredišni upravljač koji se želi prikazati. Nakon toga dobiva se nekoliko opcija kojima se odabire željeni način prikaza novog upravljača. Raspoloživi prijelazi su *show, show detail, present modally* te *present as popover*. *Show* prijelazom prikazuje se novo kreirani upravljač preko postojećeg upravljača, no ne zauzima prostor cjelokupnog zaslona. Koristeći navedeni prijelaz unutar navigacijskog upravljača ima drugačiji način prikaza koji je pojašnjen u nastavku. *Show Detail* prijelaz koristi se samo unutar *split view upravljača* gdje se odredišni upravljač ugrađuje unutar prostora za prikaz detalja. *Present Modally* prijelaz isključivo prikazuje odredišni upravljač samo na dijelu zaslona, dok *Present as Popover* prijelaz prikazuje upravljač unutar manjeg prozora koji se pokazuje na zaslonu. Poseban oblik prijelaza jest *Unwind* prijelaz kojim se omogućava prijelaz na neki od prethodno prikazanih upravljača. Prijelaze je osim iz *Storyboarda* moguće kreirati programski gdje se programeru

omogućava kreiranje uvjetnih prijelaza te inicijalizacija pojedinih varijabli odredišnog upravljača. Bitno je napomenuti da se svakim prijelazom, osim *unwind* prijelazom, kreira nova instanca odredišnog upravljača uključujući pridruženu scenu.

Navigacijski upravljači osnovna su komponenta svake iOS aplikacije, a služe za prikazivanje proizvoljnih upravljača unutar aplikacije. Glavna karakteristika navigacijskih upravljača upravo je postojanje navigacijskog stoga unutar kojeg se uvrštavaju određeni upravljači koristeći *Show* prijelaz. Prema [88], navigacijski upravljač može prikazivati samo jedan upravljač, a mora imati definiran barem jedan upravljač koji se naziva korijenski upravljač. Navigacijski upravljač kreira se tako da se iz biblioteke elemenata prevuče na radnu površinu element naziva *NavigationController*. Zatim je potrebno s unesenog navigacijskog upravljača napraviti prijelaz na drugi upravljač te kada se otvori prozor za odabir prijelaza, odabrati svojstvo pod imenom *root view controller*. Odabirom navedenog svojstva odabrani upravljač prvi je prikazan unutar navigacijske hijerarhije navigacijskog upravljača. Prijelaze na ostale upravljače moguće je ostvariti korištenjem *show* prijelaza kojeg navigacijski upravljač interpretira dodavanjem odredišnog upravljača na navigacijsku hijerarhiju te njegovo prikazivanje u prvom planu. Korisnik zatim na navigacijskoj traci (engl. *navigation bar*) povrh zaslona ima opciju povratka na prethodno prikazani upravljač čime se trenutno prikazani upravljač uklanja s navigacijskog stoga. Navigacijskom trakom omogućava se dodavanje naslova za prikazani upravljač te se omogućava dodavanje gumba kojima bi se pokretale određene radnje unutar prikazanog upravljača.

Kako je pojašnjeno u prethodnom tekstu, unutar svake iOS aplikacije mora postojati klasa *AppDelegate* koja je također *singleton* instanca kao i objekt *UIApplication* klase. *AppDelegate* idealna je klasa za definiranje injektiranja ovisnosti unutar aplikacije jer se kreira prije bilo koje komponente koja bi mogla koristiti neku od ovisnosti. Za injektiranje ovisnosti unutar aplikacije koristi se biblioteka *Swinject* koja zahtjeva definiranje takozvanog kontejnera (engl. *container*) koji sadržava definicije komponenti za rezoluciju i injektiranje. Kako bi se izbjegla jedna velika funkcija koja sadržava registracije komponenti za cijelu aplikaciju, napravili su se odvojeni moduli koji prate registracije srodnih komponenti. Stoga je potrebno kreirati direktorij koji grupira module za injektiranje. Navedeno se ostvaruje pritiskom desnog klika miša te odabira *New Group* čime se omogućuje definiranje naziva direktorija te dodavanje novih ili premještanje postojećih datoteka. Tako su primjerice napravljeni različiti moduli za prezentere, slučajeve korištenja, modul repozitorija te *storyboard* modul. Zatim su kreirane zasebne Swift datoteke koje sadržavaju proširenja nad tipom *Container* koji predstavlja središnju točku *Swinject* biblioteke. Unutar proširenja definirane su metode unutar kojih se zatim vrši registracija protokola i njihovih

implementacija. Registracija komponenti zatim se može izvršiti unutar *AppDelegate* klase, a prikazana je programskim kodom 4.7.

```
static let container: Container = {
        let container = Container()
        container.preferencesModule()
        container.domainModule()
        container.repositoryModule()
        container.presentersModule()
        container.storyboardsModule()
        return container
}()
```

Programski kod 4.7 Prikaz specificiranja kontejnera za rezoluciju ovisnosti

Programskim kodom 4.7 također je demonstrirana karakteristika Swift programskog jezika da se postavi varijabla s nepromjenjivom vrijednošću tako što se definira *lambda* koja se izvršava prilikom kreiranja objekta koji sadržava navedenu varijablu. Nadalje, unutar *AppDelegate* klase potrebno je implementirati *window* varijablu iz sučelja *UIApplicationDelegate* kako bi se omogućio prikaz upravljača na zaslonu mobilnog uređaja. Unutar metode *application(_: didFinishLaunchingWIthOptions:)* odlučuje se o prikazu prvog aplikacijskog zaslona. Ovisno o odluci, poziva se prikladna metoda koja koristi *SwinjectStoryboard* biblioteku za kreiranje početnog upravljača te dodjeljivanja upravljača varijabli *window* kako bi se moglo prikazati korisničko sučelje. Jedna od metoda dana je u nastavku (Programski kod 4.8).

```
private func navigateTo(storyboard: String, withControllerName controller: String) {
        let storyboard = SwinjectStoryboard.create(name: storyboard, bundle: nil, container:
AppDelegate.container)
        let navigationController = storyboard.instantiateInitialViewController() as!
UINavigationController
        let rootViewController = storyboard.instantiateViewController(withIdentifier: controller) as
UIViewController
        navigationController.viewControllers = [rootViewController]
        window?.rootViewController = navigationController
        window?.makeKeyAndVisible()
}
```

Programski kod 4.8 Prikaz metode za navigaciju do storyboarda i upravljača definiranih parametrima metode

Na slici je vidljivo korištenje *SwinjectStoryboard* biblioteke koja se koristi za kreiranje tipa *SwinjectStoryboard* koji služi za kreiranje upravljača s injektiranim ovisnostima. Parametrima se mora specificirati naziv te kontejner koji sadrži definiciju ovisnosti. Budući da je prethodno kreiran kontejner, ovdje se predaje kao treći parametar metodi *create*. Zatim je kreiran navigacijski upravljač i upravljač koji se postavlja kao korijenski na definirani navigacijski upravljač. Metodom *instantiateViewController* kreira se upravljač s postavljenim identifikatorom unutar *Storyboarda*. Naposljetku se definiranoj varijabli *window* specificira korijenski upravljač te se poziva metoda za prikaz korisničkog sučelja *makeKeyAndVisible*. Za pomoć u razvoju također su kreirane metode proširenja. Tako je kreirana metoda proširenja *asDictonary* koja može biti korištena na tipovima podataka koji nasljeđuju *Encodable* protokol kako bi se omogućila serijalizacija mrežnih zahtjeva.

Točnije, pretvara objekt koji implementira protokol *Encodable* u parove ključ-vrijednost. Proširenje tipa *Encodable* te implementacija navedene metode dana je programskim kodom 4.9.

```
extension Encodable {
    func asDictionary() -> [String: Any] {
        var dictionary: [String:Any] = [:]
        do {
            let data = try JSONEncoder().encode(self)
            guard let tempDictionary = try JSONSerialization.jsonObject(with: data, options:
.allowFragments) as? [String: Any] else {
                throw NSError()
            }
            dictionary = tempDictionary
        } catch {
            print("Error while converting \(self) to dictionary: \(error)")
        }

        return dictionary
    }
```

Programski kod 4.9 Prikaz proširenja nad tipom Encodable s implementacijom metode asDictionary

Koristi se unutar mrežnog sloja aplikacije opisanog sljedećim poglavljem. Od pomoćnih metoda kreirana su i proširenja za pretvaranje datuma iz jednog formata u drugi. Za uporabu biblioteke *Kingfisher* unutar aplikacije kreirano je proširenje nad tipom *UIImageView* kako bi se unutar jedne linije koda omogućilo učitavanje slike pomoću URL-a. Metoda za učitavanje slika mora sadržavati URL do slike koja se želi prikazati, te slika koja se prikazuje u slučaju da se ne može dohvatiti zahtijevana slika specificirana URL-om. Kao i na Android platformi, kreirana su dva proširenja nad tipom *UIView* kojima se omogućava lako prikazivanje i sakrivanje elemenata sučelja. Za pomoć u razvoju kreirane su i tri dodatne metode. Jednom metodom specificira se izvođenje programskog koda na glavnoj niti programa. Drugom metodom omogućava se kreiranje prozora upozorenja kojim se obavještava korisnika o važnim događajima u aplikaciji dok se trećom metodom specificira otvaranje drugih aplikacija. Trećom metodom centralizira se generiranje URL-a te uvjetno grananje u ovisnosti može li se navedeni URL otvoriti na mobilnom uređaju.

4.6.3 Razvoj zaslona iOS aplikacije

Za razvoj aplikacije kreirani su resursi unutar *Assets.xcassets* direktorija koji predstavljaju slike i ikone korištene kroz cijelu aplikaciju. Svaki resurs slike ili ikone poželjno je definirati u tri raspoložive veličine kako bi sustav odabrao najbolju sliku za određeni zaslon mobilnog uređaja odnosno gustoću piksela. Svakom resursu moguće je odrediti naziv koji se zatim može koristiti unutar koda. Kao prvi zaslon kreira se pozdravni zaslon unutar *LaunchScreen.storyboard* datoteke. Na scenu su dodani resursi iz *Assets* direktorija pomoću biblioteke elemenata. Dodanim elementima potrebno je definirati ograničenja položaja i dimenzija kako bi se mogli pravilno iscrtati na zaslonu uređaja. Definiranjem pozdravnog zaslona potrebno je otići do datoteke projekta te pod karticom *General* i stavkom *App Icons and Launch Images* postaviti *Launch Screen File* na

prethodno kreirani pozdravni zaslon. Bitno je napomenuti da ukoliko je aplikacija bila pokretana na simulatoru prije definiranja datoteke pozdravnog zaslona, aplikaciju je potrebno ukloniti s uređaja kako bi se mogao vidjeti rezultat dodavanja pozdravnog zaslona. Provjera je li korisnik prijavljen unutar iOS aplikacija premještena je u *AppDelegate* unutar metode *application* jer je ondje nužno izvršiti odluku koji upravljač se prvi prikazuje na zaslonu. Ako je korisnik prijavljen prikazuje se zaslon s ponudama, a ako nije, prikazuje se zaslon za prijavu s kojeg također može pokrenuti registraciju.

Zasloni prijave i registracije korisnika odvojeni su od ostalih zaslona aplikacije kako bi se olakšala navigacija unutar *Storyboard* datoteke te razdvojile scene i upravljači na one kojima može pristupiti registrirani korisnik, odnosno neregistriran korisnik. Za dodavanje novog *Storyboarda* unutar aplikacije potrebno je izvršiti desni klik miša na neku stavku unutar projektnog prostora te odabrati *New File*. U novootvorenom prozoru potrebno je pronaći *Storyboard* te odabrati *Next*. Na sljedećem prozoru moguće je definiranje naziva datoteke te grupe odnosno direktorija u kojeg se dodaje navedena datoteka. Nakon unosa potrebno je odabrati gumb *Create* čime se kreira specificirani *Storyboard*. Nakon kreiranja potrebno je otvoriti kreiranu datoteku te iz biblioteke elemenata prenijeti *Navigation Controller* na pozadinu *Storyboarda*. Zatim je potrebno kreirati klasu koja upravlja korijenskim upravljačem navigacijskog upravljača, u ovom slučaju *LoginViewController*. Navedeno se ostvaruje desnim klikom miša na projektne datoteke te odabirom *New File*, no sada je potrebno odabrati *Cocoa Touch Class*. Novim prozorom potrebno je definirati naziv klase te klasu koja se nasljeđuje. U ovom slučaju odabran je *UIViewController,* Swift programski jezik te sljedećim zaslonom određen direktorij u kojeg je pohranjena klasa. Zatim je potrebno opet otvoriti kreirani *Storyboard* te odabrati korijenski upravljač navigacijskog upravljača i otvoriti *identity inspector* prozor s desne strane sučelja. Pod stavkom c*lass* potrebno je unijeti naziv prethodno kreirane klase koja rukuje navedenom scenom te je pod *storyboard ID* potrebno unijeti naziv korišten prilikom korištenja *Swinject Storyboard* biblioteke za kreiranje inicijalnog upravljača. Za kreiranje sučelja scene za prijavu korisnika, potrebno je iz biblioteke elemenata na scenu prevući dva polja za unos tekstualnog sadržaja pod imenom *Text Field*, te dva gumba. Na scenu je također dodan element za prikaz loga aplikacije *UIImageView*. Navedene elemente potrebno je pozicionirati i odrediti im ograničenja u odnosu na druge elemente te rub zaslona mobilnog uređaja. Poljima za unos tekstualnog sadržaja potrebno je postaviti sadržaj unutar polja *placeholder* u prozoru *attributes inspector*. U navedenom prozoru omogućeno je postavljanje boje teksta, poravnanje te bitne karakteristike koje utječu na ponašanje samog elementa kada se pojavi programska tipkovnica. Tako je potrebno na polja za unos emaila i lozinke

isključiti automatsku kapitalizaciju unesenog sadržaja, isključiti provjeru pravopisa te prikazati najadekvatniji tip tipkovnice. U slučaju polja za lozinku, moguće je odabrati opciju *Secure Text Entry* kojom se upisivani sadržaj maskira. Za gumb prijave korisnika definira se naziv, boja teksta i pozadine, te se isključuje ponašanje gumba koje se ponovno uključuje kada se validatorom potvrdi ispravnost forme. Drugi gumb dodan je na dno korisničkog sučelja te služi za okidanje prijelaza na zaslone registracije korisnika. Navedeni prijelaz u potpunosti se realizira unutar *Storyboard* sučelja jer ne postoji potreba za kondicionalnim prijelazom na drugi upravljač. Zatim je za osposobljavanje forme za prijavu korisnika potrebno otvoriti pomoćni uređivač kojim se unutar klase upravljača dodaju *outleti* i radnje s polja za unos te gumba za prijavu korisnika. Kako bi se konfiguriralo ponašanje polja za unos i gumba unutar procesa prijave korisnika, unutar *viewDidLoad* metode pozvane su metode kojima se vrši konfiguracija elemenata sučelja. Tako se za svako polje unosa definira delegat na trenutni upravljač te dodaje radnja koja validatoru prosljeđuje novi tekstualni sadržaj elemenata na promjenu njihova sadržaja. Definiranjem delegata na *Text Field* elemente unutar upravljača omogućeno je registriranje pritiska tipke *return* na tipkovnici ili promjene fokusa na drugi element za unos teksta. Na cijeli korijenski *view* dodan je prepoznavatelj dodira kako bi se sakrila tipkovnica nakon promjene fokusa s polja za unos teksta na korijenski *view*. Specificirane su i metode kojima se pomoću *outleta* na gumb prijave mijenja boja pozadine kako bi se naznačila validnost forme te se mijenja reagiranje gumba na odabir ovisno o validnosti forme. Za razdvajanje odgovornosti te ostvarivanja arhitekture MVP unutar aplikacije, izrađena su dva protokola – za upravljač i prezenter. Dodatno je unutar Swifta određeno da jedino klase mogu implementirati navedene protokole. Nakon implementacije protokola, njihovu registraciju potrebno je izvršiti u adekvatnim *Swinject* modulima. Tako se za rezoluciju prezentera unutar upravljača mora kreirati javna varijabla koje je tipa protokola kojeg dani upravljač mora implementirati. Stoga se unutar *storyboard* modula specificira registracija nad tipom upravljača kako bi se mogao injektirati odgovarajući prezenter. Unutar *Swinject* biblioteke za navedenu funkcionalnost koristi se metoda *storyboardInitCompleted* koja kao parametar prima tip upravljača za kojeg se specificira injektiranje. Zatim se drugim parametrom definira *lambda* kojoj su ulazni parametri razlučitelj (engl. *resolver*) te instanca upravljača. Zatim se na instanci upravljača pristupa svojstvu prezentera te koristi razlučitelj kako bi se odredio konkretan tip potreban za injektiranje. Navedeno je pokazano programskim kodom 4.10 .

```
storyboardInitCompleted(LoginViewController.self) { resolver, controller in
        controller.presenter = resolver.resolve(LoginPresenterContract.self)
}
```

Programski kod 4.10 Registriranje ovisnosti upravljača unutar storyboards modula

Za registraciju prezentera koristi se metoda *register* kojoj je prvi parametar tip sučelja kojeg prezenter implementira, a kao drugi parametar potrebno je definirati *lambdu*. Ova *lambda* sastoji se od jednog ulaznog parametra koji predstavlja razlučitelja koji se zatim koristi za kreiranje konkretnog tipa prezentera te razlučivanje njegovih ovisnosti. Prikazivanje *spinnera* određuje se unutar prezentera kada primi poruku od aktivnosti da započne proces prijave, a uklanjanje *spinnera* vrši se unutar metoda povratnih poziva pozvanih s višeg sloja. U slučaju uspješne prijave korisnika, šalje se poruka upravljaču da izvrši tranziciju na zaslon s raspoloživim ponudama. Kada upravljač primi poruku od prezentera, zahtijeva od kreirane metode unutar *AppDelegate* klase navigaciju do potrebnog upravljača i pridružene scene.

Proces registracije drugačije je odrađen u usporedbi s Android platformom. Na iOS platformi registracijski zasloni kreirani su koristeći upravljače od kojih svaki predstavlja jedan dio procesa registracije. Tok podataka osmišljen je *builder* razvojnim obrascem gdje prvi registracijski zaslon popuni podatke za koje je odgovoran te prilikom ručnog prijelaza na drugi zaslon, trenutni korisnički podaci unutar *buildera* proslijede se do prezentera drugog upravljača. Slično se događa s trećim upravljačem čiji prezenter prilikom prijelaza dobiva korisničke podatke s prethodna dva upravljača. U konačnici kada korisnik ispuni formu pritiskom na tipku registracije kreira se konkretna instanca korisničkih podataka koja se koristiti prilikom komunikacije s poslužiteljem. Za kreiranje prvog registracijskog zaslona bilo je potrebno dodati elemente potrebne za unos korisničkog imena, emaila, lozinke, datum rođenja, spola te države. Za unos korisničkog imena, emaila i lozinke koristi se prethodno opisani element *Text Field* te se također dodaju elementi *Label* kako bi se mogli prikazati greške prilikom validacije neposredno ispod određenog polja za unos teksta. Za odabir datuma rođenja te države kreirani su gumbi s vlastitim stilom na čiji se odabir programski generira sučelje pomoću kojega korisnik odabire podatke i napreduje u procesu registracije. Odabir spola vrši se koristeći element *Segmented Control*. Za sve navedene elemente potrebno je unutar upravljača povezati *outlete*, a na sve gumbe dodatno radnje pomoću kojih se iscrtava sučelje u slučaju odabira datuma rođenja ili nastavlja proces registracije. Na upravljač za registraciju zatim je potrebno definirati delegate *Text Field* elemenata te radnje kojima se sakriva tipkovnica. Navedene funkcionalnosti također su dodane unutar *viewDidLoad* metode te se započinje s dohvaćanjem država unutar prezentera kako bi bile spremne za generiranje sučelja odabira. Pravila validacije navedenih elemenata vrijede jednako kao na Android platformi te se tako u ovisnosti o greškama validacije prikazuju ili uklanjaju poruke ispod elemenata. Kako bi se kreirala komponenta odabira datuma, korištena je klasa *UIDatePicker* u kojoj se u konstruktoru mora definirati pravokutnik *CGRect*. *CGRect* klasa je iz *Core Graphics* okvira kojom se definira

prostor za iscrtavanje određenih elemenata sučelja programskim putem. Tako je konstruktoru te klase potrebno predati x i y koordinate u odnosu na koje se treba definirati pravokutno područje iscrtavanja. Područje iscrtavanja definira se preostalim dvama parametrima gdje se specificiralo da širina i visina područja moraju biti jednaki zaslonu mobilnog uređaja. Zatim je potrebno postaviti opciju *datePickerMode* kojom se definira da sučelje odabira prikazuje datum i vrijeme ili samo datum. U ovome slučaju definiran je odabir datuma (Programski kod 4.11).

```
datePicker = UIDatePicker(frame: CGRect(x: 0, y: 0, width: view.frame.width, height:
view.frame.height))
datePicker?.datePickerMode = .date
datePicker?.show()
```

Programski kod 4.11 Prikaz kreiranja UIDatePicker komponente

Potrebno je također definirati instancu *Calendar* klase te postaviti početni datum i minimalni datum koji se može odabrati sučeljem. Kako bi se kreirani *picker* mogao prikazati, potrebno je pristupiti varijabli *view* upravljača te pozvati metodu *addSubview* kojoj se predaje kreirani *date picker*. Pošto se navedeni *view* iscrtava preko cijelog zaslona, a navedeni upravljač se nalazi unutar navigacijskog upravljača, programski se kreira gumb *UIBarButtonItem* koji je vezan za navigacijsku traku. Pritiskom korisnika na navedeni gumb poziva se metoda koja dohvaća odabran datum iz *pickera* te ga formatira za prikaz unutar gumba koji je otvorio *picker* kako bi se korisnika obavijestilo o odabiru datuma, te se također pretvara u format koji je potreban za registraciju na poslužitelju. Također se unutar navedene metode *picker* uklanja iz *view* hijerarhije upravljača kako bi se mogao nastaviti proces registracije. Za odabir spola dodan je element *Segmented Control* kojim se može definirati proizvoljna količina elemenata od kojih je potrebno izvršiti jedinstven odabir. Komponenta je definirana da prilikom pojave zaslona nije odabran niti jedan element te čim korisnik odabere jednu vrijednost, komponenta se smatra da je prošla validacijski proces. Odabir željene države odvija se na nešto drugačiji način. Prilikom kreiranja sučelja scene gumb kojim se otvara zaslon za odabir države je onemogućen sve dok se od poslužitelja ne dobije uspješan odgovor koji sadržava popis država. Kada prezenter signalizira upravljaču da je mrežni zahtjev uspješno izvršen, omogućava se pritisak gumba za odabir države. Nakon korisnikovog odabira gumba, poziva se radnja kojom se kreira *UIPickerView* instanca klase te delegat koji kreira potrebne elemente sučelja te ih upotpunjuje podacima. *UIPickerView* konstruktoru i konstruktoru kreirane klase koja je delegat moraju se predati dimenzije zaslona mobilnog uređaja kako bi se kreiralo korisničko sučelje koje odgovara mobilnom uređaju. Unutar delegat klase potrebno je implementirati *UIPickerViewDataSource* i *UIPickerViewDelegate* protokole. Također se mora kreirati i javna varijabla kojom se iz upravljača specificira polje država koje se trebaju prikazati unutar sučelja. Stoga je potrebno preopteretiti metode *numberOfComponents(in:)* te

pickerView(_:numberOfRowsInComponent:) iz *UIPickerViewDataSource* protokola kojima se specificira broj odjeljaka (engl. *section*) unutar odabira te broj redaka u pojedinoj komponenti. Broj odjeljaka postavlja se na jedan, a broj redaka unutar tog odjeljka jednak je broju država unutar liste država. Iz *UIPickerViewDelegate* potrebno je preopteretiti metode *pickerView(_:rowHeightForComponent:)* te *pickerView(_:viewForRow:forComponent:reusing:)*. Prvom metodom specificira se visina pojedinog *viewa* koji udomljava *view* kreiran drugom metodom. Drugom metodom potrebno je kreirati varijablu u koju se sprema instanca *UIView* klase koju je potrebno kreirati *CGRect* objektom. Zatim je potrebno definirati labelu kojom se formatira tekst države, veličina fonta te dodati na prethodno kreiranu varijablu. Na kraju se iz metode vraća modificirani *view* koji sadržava labelu. Unutar upravljača se zatim na *picker* postavlja kreirani objekt kao delegat i izvor podataka. Kreiranje gumba unutar navigacijske trake jednak je onomu unutar odabira datuma. Nakon odabira države unutar koda dobiva se odabrani indeks *pickera* pomoću kojega se određuje odabrana država. Nakon popunjene forme, korisniku se omogućava pritisak gumba koji započinje prijelaz do drugog zaslona registracije. Za manualno specificiranje prijelaza potrebno je napraviti prijelaz s jednog upravljača na drugi te unutar *attribute inspectora* dodati jedinstveni identifikator prijelaza te tip prijelaza postaviti na *Show*. Zatim je u kodu, kada se korisniku omogući prijelaz te korisnik pritisne gumb, potrebno pozvati metodu *performSegue(withIdentifier:sender:)* gdje se prvim parametrom mora proslijediti znakovni niz koji je korišten za identifikator prijelaza. Sustav zatim poziva metodu *prepare(for:sender:)* koja u prvom parametru sadržava kreirani odredišni upravljač. Prije nego se pozovu metode za postavljanje informacija o državi te registracijskim podacima korisnika, potrebno je koristeći prvi parametar provjeriti je li identifikator prijelaza uistinu onaj koji odgovara prijelazu na drugi registracijski zaslon te je li kreirani odredišni upravljač upravo klasa odgovorna za rukovanje drugim registracijskim zaslonom. Ako su oba uvjeta ispunjena, nad odredišnim upravljačem pozivaju se metode kojima se unutar njegovog prezentera postavljaju vrijednosti države te registracijski podaci korisnika u obliku *builder* obrasca kojeg upotpunjuju sljedeća dva zaslona za registraciju korisnika.

Korisničko sučelje drugog registracijskog zaslona sastoji se od *Table View* elementa, specifikacije ćelije *Table View Cell* te gumba za prijelaz na završni registracijski zaslon. Ćelijom se definira sučelje komponente koja se koristiti za prezentiranje srodnih podataka koji uobičajeno dolaze unutar liste. Elementi predstavljaju labelu te ikonu kojom je signaliziran odabir pojedinog grada. Odgovornost *Table View* komponente jest u ponovnom korištenju ćelija ako je potrebno prikazati veliki broj elemenata neke liste. Tada se ponovno iskorištava sučelje ćelije koja je izašla izvan

vidljivog područja zaslona mobilnog uređaja kako bi se izbjegle energetski i memorijski zahtjevne operacije kreiranja novih ćelija. Rezultat funkcionalnosti ekvivalentan je *Recycler Viewu* na Android platformi uz razlike u implementaciji. Za ispravno funkcioniranje ćelije potrebno je definirati klasu, u ovom slučaju *CityTableViewCell*, koja nasljeđuje *UITableViewCell* klasu kako bi se moglo programski upravljati sadržajem pojedine ćelije. Nakon kreiranja navedene klase, potrebno je unutar *Storyboarda* odabrati ćeliju te odlaskom na *identity inspector* pod stavku *class* unijeti naziv prethodno kreirane klase kako bi se upravljalo sučeljem ćelije. Ćeliji je također potrebno definirati identifikator unutar *attributes inspectora* kako bi se isti tip ćelije mogao ponovno koristiti za prikazivanje sadržaja. Potrebno je koristeći pomoćni uređivač dodati *outlete* elemenata labele i ikone unutar klase za upravljanje ćelijom. U klasi je definirana javna metoda koja prima parametar grada i status selekcije te postavlja tekstualni sadržaj labele na naziv grada te u ovisnosti o odabiru grada, prikazuje ili sakriva ikonu selekcije. Definiranjem ćelije potrebno je prijeći na definiciju protokola te njihovu implementaciju unutar upravljača i prezentera. Zadaća prezentera drugog registracijskog zaslona jest dohvaćanje gradova određene države te rukovanje korisnikovim odabirom gradova. U upravljač je potrebno dodati *outlete* od *Table Viewa* dodanog unutar scene te gumba za nastavak registracije. Unutar metode *viewDidLoad* postavljeni su delegat i izvor podataka *outleta Table Viewa* da pokazuju na trenutni upravljač čime upravljač mora osigurati implementaciju *UITableViewDelegate* i *UITableViewDataSource* protokola. U navedenoj metodi iz prezentera započinje proces dohvaćanja gradova. Dok se očekuje odgovor poslužitelja na sučelju je prikazan *spinner*, a kada podaci stignu od poslužitelja, *spinner* se zaustavlja te obavještava upravljač da obnovi podatke unutar *Table Viewa*. Metoda koju poziva upravljač na *outletu Table Viewa* jest *reloadData*. Pozivom prethodne metode vrše se dva bitna poziva za popunjavanje *Table View* elementa - *tableView(_:numberOfRowsInSection:)* i *tableView(_:cellForRowAt:)*. Prvom metodom potrebno je vratiti broj potrebnih ćelija za prikazivanje svih stavki liste te se od prezentera potražuje broj dohvaćenih gradova dok se drugom metodom mora dohvatiti ćelija pomoću dodijeljenog identifikatora te *castati* u klasu koja je zadužena za sučelje ćelije. Zatim se nad instancom ćelije poziva metoda kojom se prosljeđuje grad za prilagodbu sadržaja ćelije. Naposljetku je potrebno vratiti popunjenu ćeliju iz metode kako bi mogla biti prikazana unutar *Table Viewa*. Korisnikov odabir pojedinog grada detektira se unutar metode *tableView(_:didSelectRowAt:)* gdje se prezenteru zatim prosljeđuje selekcija određenog retka te stavlja određeni grad u listu odabranih gradova. Metode korištene za definiranje *Table View* elementa dane su programskim kodom 4.12. Korisnika se ograničava na odabir najviše tri mjesta, te ukoliko želi odabrati više, prezenter obavještava upravljač da prikaže prikladnu poruku. Da bi se ispunila validacija, korisnik mora odabrati barem jedan grad. Kada korisnik odluči

```
extension SecondRegistrationScreenViewController: UITableViewDelegate, UITableViewDataSource {

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return presenter.getCitiesCount()
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "CityCell") as! CityTableViewCell
        cell.bind(city: presenter.getCity(atIndex: indexPath.row))
        return cell
    }

    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
        presenter?.setSelectedCity(cityIndexPath: indexPath)
    }
}
```

Programski kod 4.12 Prikaz metoda potrebnih za ispravno funkcioniranje Table View elementa

nastaviti proces registracije, opet je potrebno izvršiti prijelaz iz programskog koda te trećem upravljaču proslijediti dosadašnje korisničke stavke.

Trećim registracijskim upravljačem potrebno je korisniku unutar sučelja prikazati popis kategorija od kojih mora odabrati najmanje jednu, a gornja granica nije definirana. Proces razvoja scene, protokola i implementacije protokola jednak je razvoju drugog registracijskog zaslona. Nakon korisnikovog odabira barem jedne kategorije i odabirom gumba registracije, kreira se konkretni tip registracijskog zahtjeva koji se šalje iz prezentera u slučaj korištenja zadužen za registraciju korisnika. U slučaju uspješne registracije korisniku se prikazuje poruka te ga se *unwind* prijelazom vraća na zaslon za prijavu.

Nakon uspješne prijave korisnika se preusmjerava u centralni dio aplikacije koji je realiziran drugom *Storyboard* datotekom. Za prikaz svih upravljača unutar navedenog *Storyboarda* korištena je biblioteka *SWRevealViewController* kojom se ostvaruje prikaz navigacijskog izbornika te navigacija do odredišnog upravljača. Za uporabu navedene biblioteke potrebno je u prozor *Storyboarda* prenijeti jedan upravljač iz biblioteke elemenata te u *identity inspectoru* pod polje *class* definirati klasu *SWRevealViewController*. Na scenu je također potrebno dodati dva upravljača. Jedan upravljač koristi biblioteka za prikaz navigacijskog izbornika, a drugim upravljačem specificira se početni upravljač prilikom učitavanja *Storyboarda*. U kontekstu biblioteke navigacijski izbornik mora se definirati posebnim prijelazom kojem je potrebno dodijeliti identifikator *sw_rear* te klasu *SWRevealViewControllerSegueSetController*, a početni upravljač mora se definirati prijelazom s identifikatorom *sw_front* uz istu klasu kao prethodni prijelaz. Sučelje navigacijskog izbornika kreirano je uporabom *Table View* elementa te enumeracije koja modelira tekst, ikonu i nazive prijelaza koji se moraju izvršiti nakon korisnikova odabira. Prijelazi koji prikazuju odredišni upravljač u prvom planu iz navigacijskog izbornika moraju imati definiranu *SWRevealViewControllerSeguePushController* klasu.

4.7 Testiranje iOS aplikacije

Testiranje aplikacije obavljeno je koristeći ugrađenu podršku Xcode alata za testiranje jedinica koda. Prije dodavanja testnog paketa unutar projekta, potrebno je odabrati naziv paketa kojeg se želi testirati. Zatim je potrebno odabrati *File -> New -> Target* iz alatnog izbornika. Iz otvorenog prozora potrebno je odabrati *Unit Testing Bundle* te pritisnuti gumb *Next*. U novom prozoru potrebno je unijeti naziv paketa koji sadržava sve testne klase, programski jezik korišten za specifikaciju testova te odabrati odredišni projekt i metu ukoliko nisu automatski postavljene na željene veličine. Odabirom tipke *Finish* kreira se novi paket koji sadržava *Info.plist* datoteku u kojoj su sadržane osnovne informacije o *targetu* koji se pokreće, a koji služi za pokretanje testova. Nakon kreiranja paketa uputno je kreirati zasebne grupe i klase koje se odnose na testiranje određenih jedinica aplikacije kako bi bili jasno odvojeni na logičke cjeline. Kreirane su tri grupe koje sadržavaju testne klase za testiranje validatora prijave, registracije te promjenu korisničkih kategorija. Dodavanje testne klase izvršava se odabirom kreiranog paketa testova desnom tipkom miša te odabirom *New File -> Unit Test Case Class* te dodjeljivanjem naziva klase i odabirom klase roditelja na *XCTestCase*. Kako bi se pristupilo testiranju kreirane aplikacije, potrebno je unutar testne klase uvesti cijeli izvorni projekt ključnom riječi *import* te anotacijom *@testable*.

Uobičajena testna klasa sastoji se od minimalno tri metode. Metode za postavljanje parametara testa *setUp()*; metode koja predstavlja testni slučaj; te metode kojom se oslobađa memorija prethodno kreiranih objekata, *tearDown()*.

Prema [89] i [90], posebnost *XCTest* izvršitelja jest u tome što prije izvođenja testnih slučajeva *XCTest runtime* vrši pregled svih klasa koje nasljeđuju *XCTestCase* te kreira kolekciju svih takvih klasa. U slučaju da je potrebno testirati metode koje mijenjaju bitne konfiguracije unutar sustava, nužno je korištenje *setUp* i *tearDown* metoda kako bi svaki testni slučaj započeo na očekivanom stanju, a ne na stanju postavljenom od drugog testa. Navedeno ponašanje svojstveno je *XCTest* okruženju prema [91]. Također je bitno napomenuti da testne metode moraju biti imenovane prefiksom test, ne smiju imati parametre te ne smiju imati povratni tip. Nužno je pridržavanje prethodnih pravila kako bi *XCTest runtime* pronašao testne metode jer se metode izvršavaju abecednim redom. Testne metode potrebno je modelirati tako da se postave potrebne varijable, pozove metoda testne aplikacije s definiranom varijablom kako bi se zadnjim korakom mogao utvrditi uspjeh testa.

Za potrebe testiranja aplikacije kreirani su testovi za provjeru ispravnosti validatora prijave, registracije te promjene kategorija. Validatorima prijave nužno je osigurati postojanje barem nekih

vrijednosti unutar prijavne forme jer se pretpostavlja da korisnik zna potrebne podatke za prijavu. Registracijskim validatorom testirani su konkretni znakovni skupovi regularnim izrazima kojima utvrđuje ispravnost emaila i lozinke dok se za ostale elemente registracijske forme provjerava postojanje vrijednosti iz specificiranog skupa ili u slučaju ponovljene lozinke, jednakost s unesenom lozinkom. Korišten regularni izraz za email adrese ne ograničava korisnika na broj odjeljaka korisničkog imena te domene te osigurava postojanje znaka '@'. Regularni izraz korišten za lozinke nameće duljinu lozinke na minimalno osam znakova - slova i znakovi te minimalno jedna znamenka. Sveukupno je unutar testnih datoteka kreirano 43 testna slučaja kojima se pokrivaju granične i očekivane vrijednosti parametara.

Testiranje jedinica koda Android aplikacija izvršava se koristeći JUnit 4 testni okvir. Prema [92], korištenje JUnit inačice 4 omogućeno je kreiranje testnih klasa bez potrebe nasljeđivanja *TestCase* klase iz testnog okvira te je uklonjena potreba navođenja prefiksa *test* ispred naziva testnih metoda. Testne metode potrebno je anotirati s @Test te koristiti niz pretpostavki za odluku o uspješnosti testnog slučaja.

Alat kojim je omogućeno testiranje Android i iOS mobilnih aplikacija jest Appium web poslužitelja koji prima zahtjeve dobivene od klijenta te ih prosljeđuje određenom pogonskom programu za odabranu platformu gdje se zatim izvršavaju testovi.

5. USPOREDBA PLAFORMI ANDROID I IOS S NAGLASOM NA POSLUŽITELJSKU STRANU RJEŠENJA

Prilikom razvoja bilo koje vrste aplikacija moguće je pridržavati se istog arhitekturalnog stila što razvijene aplikacije dokazuju. Slojevi između *viewa* i repozitorija, odnosno mrežnog sloja, mogu biti razvijeni na identičan način. Razlike prilikom razvoja prvotno se prikazuju u načinu kreiranja korisničkih sučelja te interakciji korisničkog sučelja s kodom klase koja upravlja njihovim prikazivanjem. Razlike se također vide u određenim elementima korisničkog sučelja te njihovim konfiguriranjem. Tako je primjerice ekvivalent androidovom *RecyclerViewu TableView* na iOS platformi. Obje komponente zahtijevaju kreiranje sučelja ćelija za jedan element liste. Platforme se razlikuju i po načinu na koji se vrše tranzicije s jednog zaslona na drugi. Razlike se također vide u bibliotekama, a prvenstveno u bibliotekama za injektiranje ovisnosti te mrežnu komunikaciju. Da bi razvijene aplikacije izvršavale svoju zadaću, potrebno je postojanje internetske veze na mobilnim uređajima. Internetska mreža mora biti ostvarena koristeći *WIFI* ili podatkovni promet jer se u protivnom ne mogu učitavati sadržaji namijenjeni za posluživanje korisniku. Za razvoj aplikacija i komunikaciju s poslužiteljem korištene su dvije popularne biblioteke za olakšavanje razvoja mrežnih mobilnih aplikacija. Riječ je o biblioteci Retrofit za Android platformu te biblioteci Alamofire za iOS platformu. Retrofit za izvršavanje svojih funkcija koristi dvije ostale biblioteke, *OkHttp* i odabrani konverter odgovora poslužitelja. Retrofit omogućuje kreiranje asinkronih mrežnih zahtjeva na drugim nitima kako bi se izbjeglo blokiranje glavne niti koja je zadužena za iscrtavanje i osvježavanje korisničkog sučelja. U slučaju da se rade sinkroni zahtjevi, sučelje aplikacije ne bi odgovaralo na korisnikove radnje odlaska korak u nazad ili mijenjanje prozora. Na obje mobilne platforme postoji limit u kojem aplikacija mora reagirati na korisnikovu radnju, te ukoliko aplikacija ne rukuje korisničkim zahtjevom, operacijski sustav prikazuje dijalog iz kojega se korisniku omogućava prekid aplikacijskog procesa ili čekanje da se blokirajuća radnja izvrši.

5.1.1 Opis razvijenog mrežnog sloja na Android platformi

Za korištenje Retrofit biblioteke potrebno je ispravno izvršiti konfiguriranje *retrofit* klijenta i kreiranje sučelja koje reflektira API na kojeg se spaja *retrofit* klijent. Mrežni klijent potrebno je definirati unutar *Koin networking* modula. Kreiranje mrežnog klijenta prikazano je programskim kodom 5.1. Prvom linijom specifikacije mrežnog modula definira se konverter zahtjeva prema poslužitelju te odgovora poslužitelja prema kreiranoj aplikaciji. Metodom *create* nad klasom *GsonConverterFactory* kreira se podrazumijevana instanca *Gson* konvertera koja serijalizira i

```
single(named(GSON_CONVERTER_FACTORY)) { GsonConverterFactory.create() as Converter.Factory }

single(named(LOGGING_INTERCEPTOR)) {
    HttpLoggingInterceptor().apply {
        level = HttpLoggingInterceptor.Level.BODY
    }
}

single(named(OKHTTPCLIENT)) {
    OkHttpClient.Builder().apply {
        if (BuildConfig.DEBUG) addInterceptor(interceptor = get(named(LOGGING_INTERCEPTOR)))
        callTimeout(timeout = REQUEST_TIMEOUT, unit = TimeUnit.SECONDS)
    }.build()
}

single(named(RETROFIT)) {
    Retrofit.Builder()
        .client(get(named(OKHTTPCLIENT)))
        .baseUrl(BASE_URL)
        .addConverterFactory(get(named(GSON_CONVERTER_FACTORY)))
        .build()
}

single(named(RETROFIT_SERVICE)) {
    get<Retrofit>(named(RETROFIT)).create(CityAlarmService::class.java)
}
```

Programski kod 5.1 Kreiranje retrofit mrežnog klijenta i usluga

deserijalizira JSON transportni format u konkretne tipove podataka korištene unutar aplikacije. Ukoliko zaglavljem HTTP zahtjeva nije specificirano korišteno kodiranje, koristi podrazumijevano UTF-8 kodiranje znakova. Potrebno je izvršiti pretvorbu u tip *Converter.Factory* jer *Koin* ne može automatski odrediti tip konvertera te bi dolazilo do *runtime* iznimke prilikom rezolucije konvertera u daljnjem procesu kreiranja mrežnog klijenta. Nakon definicije konvertera, potrebno je definirati *HttpLoggingInterceptor* koji unutar konzole ispisuje sadržaj HTTP zahtjeva specificiranog postavljanjem *level* varijable. *Level* varijabla je tip enumeracije koji se sastoji od četiri slučaja. *NONE* slučajem ne zapisuje se nikakav sadržaj unutar konzole, *BASIC* ispisuje HTTP metodu, krajnju točku, inačicu protokola te status odgovora poslužitelja. *HEADERS* slučaj uz karakteristike BASIC slučaja ispisuje i HTTP zaglavlja, a *BODY* slučaj uz ispis *BASIC* slučaja ispisuje i sadržaj poslan unutar tijela HTTP odgovora te se može smatrati potpunim ispisivanjem sadržaja zahtjeva. Za izvršavanje mrežnih zahtjeva potrebno je kreirati *OkHttpClient builder* razvojnim obrascem. Dodatno se specificira presretač (engl. *intercepter*) mrežnih poziva koji se postavlja na prethodno definirani *HttpLoggingInterceptor*. Dodavanje presretača mrežnih poziva određuje se u odnosu na zastavicu postavljenju unutar *BuildConfig* datoteke. Također se povećava vrijeme unutar kojega se mora izvršiti zahtjev i sve potrebne pretvorbe metodom *callTimeout*. Nakon definiranja potrebnih karakteristika HTTP klijenta, *build* metodom kreira se konkretan tip spreman za korištenje. Nakon definiranja konvertera i HTTP klijenta, kreiran je *retrofit* klijent koji kreira uslugu. Unutar tog bloka koda može se vidjeti *Koin* rezolucija ovisnosti pomoću definicije naziva potraživanog tipa. Unutar *get* metode potrebno je specificirati *named* metodu s varijablom

koja pohranjuje znakovni niz ovisnosti koji je prethodno korišten prilikom definiranja tražene ovisnosti. Vidljivo je da se na *builder* razvojni obrazac Retrofita definiraju HTTP klijent, bazni URL koji se dinamički dodaje na relativne putanje usluga te konverter zahtjeva i odgovora poslužitelja. Naposljetku se nad *retrofit* klijentom poziva metoda *create* kojom se kreira instanca API-ja definirana sučeljem. Metoda *create* vrši provjeru je li uistinu riječ o sučelju te također provjerava da sučelje ne nasljeđuje druga sučelja čime se osigurava da su sve krajnje točke centralizirane u jednom sučelju. U slučaju detekcije suprotnog, dolazi do *runtime* iznimke. Specifikacija API-ja definira se koristeći anotacije nad definicijama metoda te njihovim parametrima unutar sučelja. Nazivom metoda specificira se naziv koji se koristi prilikom razvoja, a anotacijom na metodi određuje se HTTP metoda te krajnja točka API-ja na koju se usmjerava mrežni zahtjev. Podržane anotacije su *@GET*, *@POST*, *@PUT*, *@DELETE*. Za svaku od anotacija potrebno je postaviti relativnu putanju koju Retrofit nadovezuje na bazni URL ili cjelokupni URL te se u tom slučaju ne dodaje bazni URL. U slučaju da se podaci šalju poslužitelju potrebno ih je proslijediti kao parametar funkcije te dodati *@BODY* anotaciju ispred parametra. Ukoliko se neki od parametara treba proslijediti u listi parametara unutar samog URL-a, potrebno je takav parametar anotirati s *@QUERY* te u zagradama definirati ključ pod kojim se uvrštava anotiran parametar. U slučaju da se parametar mora uvrstiti unutar putanje URL-a prilikom izvršavanja zahtjeva, potrebno je unutar URL-a unutar vitičastih zagrada upisati identifikator parametra. Zatim je ispred parametra metode kojeg se želi unijeti unutar putanje URL-a definirati anotaciju *@PATH* te u zagradama navesti identifikator koji se unio unutar vitičastih zagrada URL-a. Programskim kodom 5.2 izložen je dio kreiranog sučelja za korištenje API-ja.

```
interface CityAlarmService {
    @POST("api/users/login")
    fun loginUser(@Body user: UserLoginRequest): Call<UserLoginResponse>

    @POST("api/users/register")
    fun registerUser(@Body user: UserRegisterRequest): Call<UserRegisterResponse>

    @GET("api/countries/all")
    fun getCountries(): Call<List<Country>>

    @GET("api/locations/all")
    fun getLocations(@Query("country") countryIso: String): Call<List<Location>>

    ...
}
```

Programski kod 5.2 Prikaz sučelja potrebnog za razvoj retrofit klijenta

Iz programskog koda 5.2 vidljivo je da neki od parametara metoda ne pripadaju u osnovne tipove programskog jezika Kotlin. Razlog je u slanju složenih tipova podataka poput korisničkih podataka kako bi se mogla izvesti točna operacija na poslužitelju. Poslužitelj također za zahtjev na različitim krajnjim točkama daje različite odgovore. Stoga je potrebno kreirati objekte i poželjno imenovati ih tako da im je jasna svrha unutar programa. Također ti tipovi podataka služe specificiranom

konverteru da ih serijalizira prije slanja, odnosno deserijalizira nakon primanja odgovora poslužitelja. Tako je primjerice programskim kodom 5.3 izložen model korišten prilikom prijave korisnika.

```
data class UserLoginResponse(
    val user: User,
    val categories: List<Category>,
    @SerializedName("city") val cities: List<City>,
    @SerializedName("share_url") val shareUrl: String?,
    @SerializedName("share_title") val shareTitle: String?,
    @SerializedName("share_message") val shareMessage: String?,
    @SerializedName("social_image") val socialImage: String?,
    @SerializedName("share_image") val shareImage: String?
)
```

Programski kod 5.3 Prikaz UserLoginResponse modela na Android platformi

Sa slike je vidljivo da se da je riječ o podatkovnoj klasi odnosno klasi kojoj je cilj samo držati srodne podatke bez ikakve obrade metodama. Vidljivo je također da *user*, *categories* i *cities* pohranjuju podatke koje je opet potrebno izraziti drugim modelom sve dok se ne dođe do elementarnih tipova podataka. Jedna od karakteristika *Gsona* jest mogućnost specificiranja različitog naziva svojstava unutar aplikacije i naziva koji se koristi izvan aplikacije, odnosno na poslužitelju. Time se osigurava uniforman prostor imena kroz cijelu aplikaciju. Dodjeljivanje različitih naziva ostvaruje se anotacijom *@SerializedName* te specificiranjem znakovnog niza koji se koristi na API-ju. Kako bi se unutar aplikacije ostvarili SR i DI principi kreirali su se dodatni slučajevi korištenja kao sučelja i dodatno njihove implementacije u konkretnim klasama. Tako sučelje slučaja korištenja za registraciju korisnika izgleda kao programski kod 5.4.

```
interface RegistrationUseCase {
    fun execute(
        body: UserRegisterRequest,
        onSuccess: SuccessLambda<UserRegisterResponse>,
        onRequestError: RequestError,
        onError: OnError
    )
}
```

Programski kod 5.4 Prikaz sučelja slučaja korištenja za registraciju korisnika

Prikazanim sučeljem specificira se sadržaj budućeg mrežnog zahtjeva koji je predstavljen *body* parametrom metode, te preostali parametri predstavljaju metode povratnih poziva koje će biti pozvane u ovisnosti o uspješnosti izvršavanja mrežnog zahtjeva. Za kreiranje implementacije sučelja potrebno je kreirati novu klasu koja nasljeđuje kreirano sučelje. U konstruktoru klase potrebno je definirati ovisnost na API uslugu prikazan programskim kodom 5.2 kako bi *Koin* mogao injektirati uslugu unutar klase te u konačnici omogućiti izvršavanje mrežnog zahtjeva. Unutar preopterećene *execute* metode sučelja potrebno je kreirati implementaciju metode koja poziva *registerUser* metodu sučelja *retrofit* klijenta. Stoga se na injektiranoj inačici usluge poziva prethodno navedena metoda te joj se predaje parametar *body*, a zatim se na toj metodi pozivaju

metode *execute* ili *enqueue*. *Execute* metodom kreirao bi se sinkroni mrežni zahtjev, a *enqueue* metodom kreira se asinkroni zahtjev koji se koristi u većini slučajeva. Nakon kreiranja sučelja slučaja korištenja i njegove implementacije, potrebno je izvršiti registraciju unutar *Koin domain* modula kako bi se slučajevi korištenja mogli pozvati iz odgovarajućih prezentera.

5.1.2 Opis razvijenog mrežnog sloja na platformi iOS

Na iOS platformi potreban je drugačiji pristup razvijanju mrežnog sloja aplikacije. Najpoznatija biblioteka za razvoj mrežnih aplikacija jest Alamofire. Alamofire biblioteka izgrađena je nad Appleovom *URLSession* klasom kako bi se aplikacijama omogućila uporaba interneta. Za korištenje biblioteke nije potrebno izvesti složene postupke kreiranja klijenta kao kod biblioteke Retrofit na Android platformi, ali je zato potrebno definirati zasebnu datoteku koja će enkapsulirati funkcionalnosti jednog API-ja. Bitna razlika spram Retrofit biblioteke jest da je Alamofire asinkrona biblioteka, te ne podržava kreiranje sinkronih zahtjeva. Za izvršavanje HTTP zahtjeva, biblioteka na korištenje daje globalnu varijablu *AF* postavljenu na *singleton* instancu *Session* klase. Omogućava kreiranje zahtjeva, pokretanje preuzimanja te slanja podataka. Za kreiranje zahtjeva koristila se metoda *request* metoda koja u biblioteci izgleda: „*request(_ convertible: URLRequestConvertible, interceptor: RequestInterceptor? = nil) -> DataRequest*“. Navedena metoda korištena je za kreiranje svakog mrežnog zahtjeva unutar iOS aplikacije. Predavanje prvog parametra predstavlja se enumeracijom koja je funkcionalni ekvivalent sučelju usluga potrebnog za *retrofit* klijent. Drugi parametar nije korišten, točnije, ostaje postavljen na *nil* vrijednost. Za enkapsuliranje rukovanja API-jem kreirana je posebna enumeracija, API usmjerivač (engl. *api router*) koji nasljeđuje protokol *URLRequestConvertible* koji zahtjeva implementaciju metode *asURLRequest. AsURLRequest* metodu poziva biblioteka Alamofire prilikom postupka kreiranja zahtjeva koja će u tom trenutku pokušati generirati *URLRequest* iz dane enumeracije koji će predati *URLSession* biblioteci na izvršavanje. Slučajevi API usmjerivača predstavljaju krajnje točke na koje su usmjereni pozivi te imaju pridružene vrijednosti pomoću kojih se modeliraju podaci koji si uvršteni u određena zaglavlja ili tijelo HTTP zahtjeva. Korištena HTTP metoda, krajnja točka, parametri tijela HTTP zahtjeva, parametri upita i dodatna zaglavlja za pojedini slučaj predstavljeni su privatnim obradnim svojstvima. Obradna svojstva za odabir HTTP metode moraju na trenutnom slučaju odrediti koji slučaj HTTP metode treba vratiti iz *HTTPMethod* enumeracije. Varijablom *path* određuje se relativna putanja do krajnje točka koja se mora koristiti prilikom specificiranog slučaja. Obradno svojstvo tijela HTTP zahtjeva, parametra upita te zaglavlja vraćaju isti tip podatka „*Parameters?*“ kojim se naznačuje da za određeni slučaj navedene varijable ne moraju imati vrijednost, tj. mogu vratiti *nil* te se navedeno svojstvo iskorištava prilikom generiranja

zahtjeva. Ako pojedini slučaj zahtijeva vraćanje vrijednosti, vrijednost se mora vratiti u obliku imenika kako bi se mogao dobiti dobro formatiran JSON dokument, specificirati metode zaglavlja ili parametri upita. Kako bi se minimizirale greške prilikom navođenja baznog URL-a, metoda zaglavlja ili parametara upita dobro je kreirati strukturu koja pohranjuje samo konstante za razvoj mrežne strane aplikacije. Unutar strukture zatim se mogu definirati enumeracije s pridruženim vrijednostima u koje se pohranjuju znakovni nizovi kako bi se izbjeglo ručno pisanje konstanti na mjestima korištenja. Naposljetku se svi navedeni elementi objedinjuju unutar *asURLRequest* metode gdje se izgrađuje zahtjev. Implementacija metode dana je programskim kodom 5.5.

```
func asURLRequest() throws -> URLRequest {
        let url = try NetworkingConstants.baseUrl.asURL()

        var urlRequest = URLRequest(url: url.appendingPathComponent(path))
        urlRequest.httpMethod = method.rawValue

        urlRequest.setValue(NetworkingConstants.DataType.json.rawValue, forHTTPHeaderField:
NetworkingConstants.Headers.contentType.rawValue)
        urlRequest.setValue(NetworkingConstants.DataType.json.rawValue, forHTTPHeaderField:
NetworkingConstants.Headers.acceptType.rawValue)

        if let additionalHeaders = additionalHeaders {
            urlRequest.setValue(additionalHeaders[NetworkingConstants.Headers.authentication.rawValue] as?
String, forHTTPHeaderField: NetworkingConstants.Headers.authentication.rawValue)
        }

        if let encodingParameters = parameters {
            do {
                urlRequest.httpBody = try JSONSerialization.data(withJSONObject: encodingParameters,
options: [])

                print(urlRequest)
            } catch {
                throw AFError.parameterEncodingFailed(reason: .jsonEncodingFailed(error: error))
            }
        }

        if let queryString = queryString {
            let encoding = URLEncoding.queryString
            do {
                try urlRequest = encoding.encode(urlRequest, with: queryString)
            } catch {
                throw AFError.parameterEncodingFailed(reason: .jsonEncodingFailed(error: error))
            }
        }

        return urlRequest
}
```

Programski kod 5.5 Prikaz implementacije asURLRequest metode

Na slici je vidljivo da se prvotno pokušava kreirati URL iz strukture mrežnih konstanti koji se zatim koristi u drugom redu za izgradnju potpune krajnje točke. Unutar *appendingPathComponent* može se vidjeti pozivanje obradnog svojstva *path* koji se evaluira tek prilikom navedenog poziva. Nakon toga specificirana je HTTP metoda zahtjeva te je uzeta osnovna vrijednost slučaja enumeracije koja vraća znakovnu reprezentaciju odabrane metode [93]. Neposredno ispod toga navodi se JSON tip podatka za zaglavlja *Content-Type* te *Accept-Type*. Trima *if* uvjetima određuju

se dodana zaglavlja, sadržaj tijela zahtjeva te parametri upita. Ako je bilo kojih od tih uvjeta *nil,* nikakva se vrijednost ne dodaje unutar *urlRequest* varijable.

Arhitekturalna razina repozitorija kreirana je kako slučajevi korištenja ne bi morali znati točno kreiranje Alamofire zahtjeva, validatora i deserijalizacije podataka. Štoviše, omogućeno je identično korištenje protokola kao na Android platformi gdje se sučelje dinamički implementira unutar *Retrofit* biblioteke. S gledišta slučajeva korištenja funkcioniraju jednako na obje platforme – dobivaju podatke s razine prezentera te im je jedina odgovornost pokrenuti proces dohvaćanja podataka i proslijediti rezultat na niži sloj. Slučajevi korištenja znaju da neki objekt implementira protokol o čijoj implementaciji ovise te da će dobiti podatke koje prosljeđuju kroz metode povratnih poziva. Protokolom repozitorija osigurava se jednako sučelje između slučajeva korištenja te repozitorija. To znači da pojavom zahtjeva za dohvaćanjem nekih podataka iz priručne memorije u kombinaciji s dohvaćenim podacima putem HTTP zahtjeva postaje odgovornost repozitorija, te da se razina slučajeva korištenja neće trebati mijenjati. Implementaciju sučelja repozitorija definira se novom klasom koja nasljeđuje navedeni protokol. Svakom metodom se zatim definira Alamofire poziv na prethodno kreirani API usmjerivač koji generira validan URL zahtjev. Programskim kodom 5.6 prikazana je metoda za registraciju korisnika.

```
func registerUser(_ user: UserRegistrationRequest, onSuccess: @escaping
SuccessLambda<UserRegistrationResponse>, onError: @escaping OnError) {
        AF.request(ApiRouter.registerUser(user:
user)).customRegistrationValidator().validate().responseDecodable(of:
UserRegistrationResponse.self, queue: .global(qos: .default)) { response in
            response.handleServerResponse(onSuccess: onSuccess, onError: onError)
        }
}
```

Programski kod 5.6 Prikaz implementacije metode registerUser unutar implementacije repozitorija

Za potrebu takvog zahtjeva kreirano je proširenje *customRegistrationValidator* nad tipom *DataRequest* kako bi se mogla detektirati greška tijekom pokušaja registracije te pripremiti poruka koja se pokazuje korisniku. *Validate* metoda dolazi od same biblioteke Alamofire, a zadužena je za validaciju statusnog koda i provjeru je li u zaglavlju sadržan tip podatka koji je proslijeđen u zaglavlje prilikom kreiranja mrežnog zahtjeva. Bitno je napomenuti da ako prvi validator podigne grešku, ostale metode koje slijede samo prosljeđuju tu grešku do mjesta povratnog poziva. Dobiveni odgovor poslužitelja pokušava se deserijalizirati pomoću *responseDecodable* metode gdje se specificira klasa u koju se treba izvršiti deserijalizacija te red izvršavanja kako bi se deserijalizacija odvijala na drugoj niti. Kao zadnji parametar specificira se povratni poziv koji dobiva *response* kao parametar koji je tipa *DataResponse*. Zatim se na tom tipu kreiralo proširenje koje će vratiti grešku ili uspješno deserijaliziran objekt na glavnu nit te se podaci vraćaju kroz

slučaj korištenja, prezenter pa do korisničkog sučelja. Nakon implementacije repozitorija, potrebno je izvršiti registraciju protokola repozitorija i implementaciju repozitorija kako bi bio spreman za injektiranje. Definicija protokola slučaja korištenja drugačija je od definicije sučelja Kotlin programskim jezikom. Programskim kodom 5.7 prikazana je definicija protokola za registracijski slučaj korištenja kako bi se moglo direktno usporediti programskim kodom 5.4.

```
protocol RegistrationUseCase: class {
    var networkingRepository: NetworkingRepository { get }
    init(networkingRepository: NetworkingRepository)
    func execute(
        body user: UserRegistrationRequest,
        onSuccess: @escaping SuccessLambda<UserRegistrationResponse>,
        onError: @escaping OnError)
}
```

Programski kod 5.7 Prikaz protokola slučaja korištenja za registriranje korisnika

Swift programskim jezikom može se ograničiti implementacija protokola samo na klase, što je učinjeno dodajući *class* ključnu riječ nakon imena protokola. Također je deklarirana varijabla tipa *NetworkingRepository* koju klasa mora implementirati te funkcionalnost klase može samo zatražiti vrijednost pohranjenu unutar varijable što je osigurano ključnom riječi *get*. Deklariran je konstruktor kojeg klasa mora implementirati, a koji se koristi za injektiranje ovisnosti klase. Metoda *execute* ima identičnu deklaraciju metodi unutar *Kotlina* s razlikom da se unutar metode koristi naziv parametra *user* te da je metodama povratnih poziva potrebno dodati *@escaping* prije definiranja tipa metode jer se time omogućava pozivanje povratnog poziva nakon izvršavanja mrežnog zahtjeva.

Definicija tipova podatka koji se dobiju od poslužitelja također se razlikuju od Android platforme, ali promjene nisu značajne. Za svrhu usporedbe s Android platformom (Programski kod 5.3), programskim kodom 5.8 prikazan je model koji se koristi za prijavu korisnika na iOS platformi. Usporedbom je vidljivo da programski jezik Swift ima drugačiju sintaksu za deklaraciju liste elemenata, ali da je deklaracija ostalih varijabli identična. Na iOS platformi potrebno je osigurati implementaciju *Codable* protokola kako bi se omogućila serijalizacija podataka i deserijalizacija odgovora poslužitelja u objekt pojedine klase. Ako definirani objekt ima nazive varijabli jednake kao odgovor poslužitelja, onda je dovoljno na strukturu ili klasu navesti implementaciju *Codable* protokola. U slučaju da se želi koristiti različit naziv varijable od onoga na poslužitelju, potrebno je unutar klase ili strukture implementirati enumeraciju *CodingKeys*. Enumeracija prvotno mora naslijediti *String* protokol, a zatim protokol *CodingKey*. Navođenjem *String* protokola omogućava se funkcionalnost Swifta kojom se definiraju nepromjenjive znakovne oznake na slučajevima enumeracije tzv. osnovne vrijednosti, a *CodingKey* upućuje sustavu da je moguće dohvatiti potrebne ključeve u procesu serijalizacije ili deserijalizacije. Dodjeljivanje naziva koji se koristi

```swift
struct UserLoginResponse: Codable {
    let user: User
    let categories: [LoginCategory]
    let cities: [LoginCity]
    let shareUrl: String?
    let shareTitle: String?
    let shareMessage: String?
    let socialImage: String?
    let shareImage: String?

    enum CodingKeys: String, CodingKey {
        case user
        case categories
        case cities = "city"
        case shareUrl = "share_url"
        case shareTitle = "share_title"
        case shareMessage = "share_message"
        case socialImage = "social_image"
        case shareImage = "share_image"
    }
}
```

Programski kod 5.8 Prikaz UserLoginResponse modela na iOS platformi

na poslužitelju na slučaj enumeracije vrši se znakom jednakost te specificiranjem znakovnog niza. Bitno je napomenuti da svi slučajevi *CodingKeys* enumeracije moraju imati iste nazive kao varijable koje se koriste u kodu klase ili strukture, a slučajevi se ne smiju izostaviti jer u protivnom nisu korišteni u serijalizaciji i deserijalizaciji.

U tablici 5.1 izloženi su kvalitativni kriteriji usporedbe razvoja mrežne arhitekturalne razine koristeći Retrofit i Alamofire biblioteke u razvoju Android i iOS mobilnih aplikacija. Kvalitativni kriteriji podijeljeni su u tri kategorije po složenosti implementacije, odnosno korištenja – nisku, srednju i visoku.

Tablica 5.1 Prikaz usporedbe Android i iOS mobilnih platrofmi prilikom razvoja mrežne arhitekturalne razine

Kriteriji usporedbe	Android platforma	iOS platforma
Kreiranje modela	Niska	Niska
Kreiranje mrežnog klijenta	Srednja	Niska
Kreiranje mrežnog sučelja poslužitelja	Niska	Visoka
Kreiranje interaktora	Niska	Niska
Kreiranje repozitorija	Niska	Niska
Razumljivost	Visoka	Srednja
Testabilnost	Srednja	Srednja
Proširivost	Visoka	Visoka
Uporabljivost mrežnih sučelja unutar repozitorija	Visoka	Visoka

6. ZAKLJUČAK

Zajedničke karakteristike Android i iOS platforme su slojevita arhitektura u kojoj razine preko jedinstvenog sučelja pristupaju uslugama druge razine koja implementira potraživane funkcionalnosti. Nakon opisa platformi opisani su programski jezici Java i Objective-C koji su korišteni za razvoj mobilnih aplikacija u začecima na operacijskim sustavima Android te iOS. Kotlin i Swift programski su jezici nove generacije koji su zamijenili prethodno spomenute programske jezike za razvoj mobilnih aplikacija. Karakterizira ih pojednostavljena sintaksa, jasnoća te sažetost koda koju nije moguće postići starijim programskim jezicima. Nadalje, u radu je opisana arhitektura MVP koja je korištena za razvoj mobilnih aplikacija u praktičnom dijelu rada. U radu su pojašnjene tehnike testiranja programske podrške. Dvije glavne razine testiranja programske podrške su funkcionalno i nefunkcionalno testiranje. Funkcionalnim testiranjem razvijeni program testira se na ispunjenje korisničkih zahtjeva. U funkcionalno testiranje pripada testiranje jedinica koda, integracijsko testiranje, testiranje sustava te testiranje prihvatljivosti razvijene programske podrške. Nefunkcionalnim testiranjem testira se pouzdanost, uporabljivost, performanse pri povećanom opterećenju te sigurnost. Nakon testiranja, pojašnjena su načela razvoja SOLID kojih se uputno pridržavati prilikom razvoja programskih projekata, jer vode do lakšeg održavanja projekta i povećanja fleksibilnosti. Pokazan je poslužitelj korišten za ostvarivanje funkcionalnosti razvijenih mobilnih aplikacija te su prikazani odgovori nekih zahtjeva kako bi se dobio uvid u oblik podataka korištenih u modelima aplikacija. Prikazan je razvoj Android i iOS mobilnih aplikacija primjenom arhitekture MVP kojom je jasno razdvojena odgovornost između sloja pogleda, modela te prezentera. Primjenom arhitekture MVP aplikacije se priprema na promjene zahtjeva te dodavanje novih funkcionalnosti uz bržu implementaciju zahtjeva. Usporedbom Android i iOS platforme prikazane su razlike i sličnosti prilikom definiranja modela, interaktora te repozitorija kojim se izvršavaju mrežni zahtjevi. Na Android platformi koristeći biblioteku Retrofit nužno je koristiti dodatnu biblioteku treće strane kojom se omogućuje serijalizacija i deserijalizacija podataka u komunikaciji s poslužiteljem. U ovom radu korištena je biblioteka Gson na platformi Android kojom je nužno anotirati varijable klase u slučaju da se nazivi razlikuju od onih korištenih na poslužitelju. Na platformi iOS ne postoji potreba za bibliotekama treće strane, jer postoji ugrađena podrška za serijalizaciju i deserijalizaciju koristeći *Codable* protokol. Također, postavljanje Retrofit klijenta je zahtjevnije u usporedbi s Alamofire klijentom, no Retrofit klijent pruža sažetiji opis sučelja poslužitelja, dok Alamofire klijent zahtijeva više postavljanja. Krajnje točke poslužitelja na mobilnoj platform iOS modelirane su

enumeracijom kako bi se postigla uporaba bibliotek Alamofire e na sličan način Retrofit biblioteci na Android platformi.

Unaprjeđenje rada može biti u poboljšanju sigurnosti aplikacije, korištenju notifikacija pri pojavi novih aktualnih ponuda za određenog korisnika te korištenje GPS tehnologije i notifikacije zasnovane na geolokaciji korištenjem ograničavanja geografskog područja.

SAŽETAK

U ovome radu uspoređen je razvoj mobilnih aplikacija za Android i iOS platforme. Obje aplikacije spajaju se na dostupni poslužitelj kako bi se mogla oblikovati korisnička sučelja. Uvodnim dijelom rada prikazane su najzastupljenije mobilne platforme današnjice te opis arhitekturalnih komponenti od kojih se sastoje. Opisano je korištenje programskih jezika Java, Objective-C s naglaskom na programske jezike Kotlin i Swift koji su ih zamijenili u razvoju aplikacija. Opisana je arhitektura MVP korištena za razvoj praktičnog dijela rada. Prikazan je kratki pregled metoda testiranja programske podrške kao i načela razvoja SOLID pri pisanju programskog koda. Opis razvoja svake od platformi započinje opisom korištenih biblioteka te ključnih elemenata potrebnih za razvoj aplikacija. Značajnije biblioteke korištene za razvoj Android aplikacije uključuju Retrofit i Koin, a za razvoj iOS aplikacije biblioteke Alamofire te Swinject. Na kraju rada dana je usporedba mobilnih platformi s gledišta mrežne razine arhitekture s kreiranim modelima, mrežnim klijentima, interaktorima i repozitorijima.

Ključne riječi: Android, iOS, Kotlin, MVP, Swift.

Comparative analysis of software approaches in development of mobile applications for Android and iOS

ABSTRACT

The development of Android and iOS applications is presented in this paper. Both applications connect to the server to fetch the data required for the creation of user interfaces. In the introductory part of this paper the two most popular mobile platforms are described along with the description of architectural components of each platform. The Java and Objective-C programming languages are described with the emphasis on Kotlin and Swift programming languages which replaced the former ones in the application development. MVP architecture, which was used for the application development, is also described. Furthermore, testing methods and SOLID principles are also described in the following section. Overview of application development starts with the explanation of libraries needed for the development after which the explanation of key application elements is given. The most important libraries used for Android application are Retrofit and Koin, while for iOS application those are Alamofire and Swinject. With the ending section the mobile platforms are compared with the emphasis on the network architecture layer with created models, network clients, interactors and repositories.

Keywords: Android, iOS, Kotlin, MVP, Swift.

LITERATURA

[1] R. C. Martin, Clean Code, Pearson, 2009.

[2] R. C. Martin, Clean Architecture, Pearson, 2018.

[3] »Eventbrite,« Dostupno: https://www.eventbrite.com/. Pristup ostvaren: 9.7.2020.

[4] »Meetup,« Dostupno: https://www.meetup.com/. Pristup ostvaren: 9.7.2020.

[5] »All Events In City,« Dostupno: vhttps://allevents.in/#. Pristup ostvaren: 9.7.2020.

[6] »Android Studio,« Dostupno: https://developer.android.com/studio. Pristup ostvaren: 11.7.2020.

[7] »Xcode,« Dostupno: https://developer.apple.com/xcode/. Pristup ostvaren: 11.7.2020.

[8] »CocoaPods.org,« Dostupno: https://cocoapods.org/. Pristup ostvaren: 23.3.2020.

[9] »Carthage,« Dostupno: https://github.com/Carthage/Carthage. Pristup ostvaren: 23.3.2020.

[10] »RxKotlin,« Dostupno: https://github.com/ReactiveX/RxKotlin. Pristup ostvaren: 11.7.2020.

[11] »RxSwift,« Dostupno: https://github.com/ReactiveX/RxSwift. Pristup ostvaren: 11.7.2020.

[12] »Platform Architecture,« Dostupno: https://developer.android.com/guide/platform. Pristup ostvaren: 21.4.2020.

[13] »The Android OS,« Dostupno: https://uniandes-se4ma.gitlab.io/books/chapter7/the-android-os.html. Pristup ostvaren: 21.4.2020.

[14] »Android Runtime (ART) and Dalvik,« Dostupno: https://source.android.com/devices/tech/dalvik. Pristup ostvaren: 21.4.2020.

[15] »Darwin-XNU,« Dostupno: https://github.com/apple/darwin-xnu. Pristup ostvaren: 23.4.2020.

[16] »XNU: The Kernel,« Dostupno: http://osxbook.com/book/bonus/ancient/whatismacosx/arch_xnu.html. Pristup ostvaren: 23.4.2020.

[17] »Accelerate,« Dostupno: https://developer.apple.com/documentation/accelerate. Pristup ostvaren: 23.4.2020.

[18] »DotNetTricks,« Dostupno: https://www.dotnettricks.com/learn/xamarin/understanding-xamarin-ios-build-native-ios-app. Pristup ostvaren: 23.4.2020.

[19] »JAVASOFT SHIPS JAVA 1.0,« Dostupno: https://web.archive.org/web/20070310235103/http://www.sun.com/smi/Press/sunflash/1996-01/sunflash.960123.10561.xml. Pristup ostvaren: 18.4.2020.

[20] »The Java® LanguageSpecification,« Dostupno: https://docs.oracle.com/javase/specs/jls/se14/jls14.pdf. Pristup ostvaren: 18.4.2020.

[21] »Objective-C Overview,« Dostupno: https://www.tutorialspoint.com/objective_c/objective_c_overview.htm. Pristup ostvaren: 18.4.2020.

[22] »Programming with Objective-C,« Dostupno: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html. Pristup ostvaren: 18.4.2020.

[23] »Kotlin 1.0 Released,« Dostupno: https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/. Pristup ostvaren: 14.4.2020.

[24] »The Kotlin Programming Language,« Dostupno: https://github.com/JetBrains/kotlin. Pristup ostvaren: 14.4.2020.

[25] »Kotlin,« Dostupno: https://kotlinlang.org/. Pristup ostvaren: 14.4.2020.

[26] »Kotlin/Native for Native,« Dostupno: https://kotlinlang.org/docs/reference/native-overview.html. Pristup ostvaren: 14.4.2020.

[27] »Learn Kotlin,« Dostupno: https://kotlinlang.org/docs/reference/. Pristup ostvaren: 14.4.2020.

[28] »Creation of Swift,« Dostupno: https://atp.fm/205-chris-lattner-interview-transcript/#swiftcreation. Pristup ostvaren: 16.4.2020.

[29] »The Swift Programming Language,« Dostupno: https://github.com/apple/swift. Pristup ostvaren: 16.4.2020.

[30] »About Swift,« Dostupno: https://docs.swift.org/swift-book/. Pristup ostvaren: 16.4.2020.

[31] »Infimum Android Handbook,« Dostupno: https://infinum.com/handbook/books/android/MVP. Pristup ostvaren: 8.7.2020.

[32] »MVP architecture,« Dostupno: https://blog.mindorks.com/android-mvp-architecture-extension-with-interactors-and-repositories-bd4b51972339. Pristup ostvaren: 19.4.2020.

[33] »Software Testing Methodologies,« Dostupno: https://smartbear.com/learn/automated-testing/software-testing-methodologies/. Pristup ostvaren: 20.4.2020.

[34] »About Postman,« Dostupno: https://www.postman.com/about-postman. Pristup ostvaren: 18.3.2020.

[35] »Markdown,« Dostupno: https://markdownlivepreview.com/. Pristup ostvaren: 18.3.2020.

[36] »Postman Learning Center,« Dostupno: https://learning.postman.com/docs/postman/launching-postman/introduction/. Pristup ostvaren: 18.3.2020.

[37] »Gradle,« Dostupno: https://docs.gradle.org/current/userguide/writing_build_scripts.html#sec:the_gradle_build_language. Pristup ostvaren: 21.3.2020.

[38] »Create a Card-Based Layout,« Dostupno: https://developer.android.com/guide/topics/ui/layout/cardview. Pristup ostvaren: 21.3.2020.

[39] »Material Components,« Dostupno: https://github.com/material-components/material-components-android. Pristup ostvaren: 21.3.2020.

[40] »Supporting Swipe-to-Refresh,« Dostupno: https://developer.android.com/training/swipe. Pristup ostvaren: 21.3.2020.

[41] »Glide,« Dostupno: https://github.com/bumptech/glide. Pristup ostvaren: 3.21.2020.

[42] »Glide - Download & Setup,« Dostupno: http://bumptech.github.io/glide/doc/download-setup.html#permissions. Pristup ostvaren: 21.3.2020.

[43] »Osmdroid,« Dostupno: https://github.com/osmdroid/osmdroid. Pristup ostvaren: 21.3.2020.

[44] »Koin,« Dostupno: https://github.com/InsertKoinIO/koin. Pristup ostvaren: 22.3.2020.

[45] »What is KOIN?,« Dostupno: https://doc.insert-koin.io/#/introduction. Pristup ostvaren: 22.3.2020.

[46] »Retrofit,« Dostupno: https://github.com/square/retrofit. Pristup ostvaren: 22.3.2020.

[47] »Retrofit - Introduction,« Dostupno: https://square.github.io/retrofit/. Pristup ostvaren: 22.3.2020.

[48] »Gson,« Dostupno: https://github.com/google/gson. Pristup ostvaren: 22.3.2020.

[49] »App resources overview,« Dostupno: https://developer.android.com/guide/topics/resources/providing-resources. Pristup ostvaren: 28.3.2020.

[50] »Drawable resources,« Dostupno: https://developer.android.com/guide/topics/resources/drawable-resource#StateList. Pristup ostvaren: 28.3.2020.

[51] »App Manifest Overview,« Dostupno: https://developer.android.com/guide/topics/manifest/manifest-intro. Pristup ostvaren: 28.3.2020.

[52] »Android Manifest - Package name and application ID,« Dostupno: https://developer.android.com/guide/topics/manifest/manifest-intro#package-name. Pristup ostvaren: 28.3.2020.

[53] »Permissions overview,« Dostupno: https://developer.android.com/guide/topics/permissions/overview. Pristup ostvaren: 28.3.2020.

[54] »Manifest.permission,« Dostupno: https://developer.android.com/reference/android/Manifest.permission. Pristup ostvaren: 28.3.2020.

[55] »Services overview,« Dostupno: https://developer.android.com/guide/components/services. Pristup ostvaren: 28.3.2020.

[56] »Broadcasts overview,« Dostupno: https://developer.android.com/guide/components/broadcasts. Pristup ostvaren: 28.3.2020.

[57] »Intent,« Dostupno: https://developer.android.com/reference/android/content/Intent#standard-broadcast-actions. Pristup ostvaren: 28.3.2020.

[58] »Content providers,« Dostupno: https://developer.android.com/guide/topics/providers/content-providers. Pristup ostvaren: 28.3.2020.

[59] »Activity,« Dostupno: https://developer.android.com/guide/topics/manifest/activity-element. Pristup ostvaren: 29.3.2020.

[60] »Introduction to Activities,« Dostupno: https://developer.android.com/guide/components/activities/intro-activities. Pristup ostvaren: 31.3.2020.

[61] »Documentation - Activity,« Dostupno: https://developer.android.com/reference/android/app/Activity. Pristup ostvaren: 31.3.2020.

[62] »Intents and Intent Filters,« Dostupno: https://developer.android.com/guide/components/intents-filters. Pristup ostvaren: 31.3.2020.

[63] »Fragments,« Dostupno: https://developer.android.com/guide/components/fragments. Pristup ostvaren: 31.3.2020.

[64] »Fragment,« Dostupno: https://developer.android.com/reference/androidx/fragment/app/Fragment. Pristup ostvaren: 31.3.2020.

[65] »Package Manager,« Dostupno: https://swift.org/package-manager/. Pristup ostvaren: 23.3.2020.

[66] »Swift Package Manager for iOS,« Dostupno: https://www.raywenderlich.com/7242045-swift-package-manager-for-ios. Pristup ostvaren: 23.3.2020.

[67] »CocoaPods,« Dostupno: https://github.com/CocoaPods/CocoaPods. Pristup ostvaren: 23.3.2020.

[68] »CocoaPods - Specs,« Dostupno: https://github.com/CocoaPods/Specs. Pristup ostvaren: 23.3.2020.

[69] »Specs and the Specs Repo,« Dostupno: http://guides.cocoapods.org/making/specs-and-specs-repo.html. Pristup ostvaren: 23.3.2020.

[70] »Alamofire.podspec.json,« Dostupno: https://github.com/CocoaPods/Specs/blob/master/Specs/d/a/2/Alamofire/5.0.1/Alamofire.podspec.json. Pristup ostvaren: 23.3.2020.

[71] »Using CocoaPods,« Dostupno: https://guides.cocoapods.org/using/using-cocoapods.html#what-is-happening-behind-the-scenes. Pristup ostvaren: 23.3.2020.

[72] »Semantic Versioning,« Dostupno: https://semver.org/. Pristup ostvaren: 23.3.2020.

[73] »CocoaPods Tutorial for Swift: Getting Started,« Dostupno: https://www.raywenderlich.com/7076593-cocoapods-tutorial-for-swift-getting-started#toc-anchor-008. Pristup ostvaren: 23.3.2020.

[74] »The Podfile,« Dostupno: https://guides.cocoapods.org/using/the-podfile.html. Pristup ostvaren: 23.3.2020.

[75] »Alamofire,« Dostupno: https://github.com/Alamofire/Alamofire. Pristup ostvaren: 24.3.2020.

[76] »Alamofire Reference,« Dostupno: https://alamofire.github.io/Alamofire/. Pristup ostvaren: 24.3.2020.

[77] »Swinject,« Dostupno: https://github.com/Swinject/Swinject. Pristup ostvaren: 24.3.2020.

[78] »Kingfisher,« Dostupno: https://github.com/onevcat/Kingfisher. Pristup ostvaren: 24.3.2020.

[79] »Toast-Swift,« Dostupno: https://github.com/scalessec/Toast-Swift. Pristup ostvaren: 24.3.2020.

[80] »SWRevealViewController,« Dostupno: https://github.com/John-Lluch/SWRevealViewController. Pristup ostvaren: 24.3.2020.

[81] »UIApplicationDelegate,« Dostupno: https://developer.apple.com/documentation/uikit/uiapplicationdelegate. Pristup ostvaren: 9.4.2020.

[82] »UIApplication,« Dostupno: https://developer.apple.com/documentation/uikit/uiapplication. Pristup ostvaren: 9.4.2020.

[83] »View Controllers,« Dostupno: https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/WorkWithViewControllers.html#//apple_ref/doc/uid/TP40015214-CH6-SW1. Pristup ostvaren: 9.4.2020.

[84] »Anatomy of a Constraint,« Dostupno: https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/AnatomyofaConstraint.html#//apple_ref/doc/uid/TP40010853-CH9-SW1. Pristup ostvaren: 10.4.2020.

[85] »Outlet,« Dostupno: https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/Outlet.html. Pristup ostvaren: 10.4.2020.

[86] »Target-Action,« Dostupno: https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/TargetAction.html#//apple_ref/doc/uid/TP40009071-CH3-SW1. Pristup ostvaren: 10.4.2020.

[87] »Using Segues,« Dostupno: https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/UsingSegues.html#//apple_ref/doc/uid/TP40007457-CH15-SW1. Pristup ostvaren: 10.4.2020.

[88] »UINavigationController,« Dostupno: https://developer.apple.com/documentation/uikit/uinavigationcontroller. Pristup ostvaren: 10.4.202.

[89] »iOS Unit Testing,« Dostupno: https://www.raywenderlich.com/960290-ios-unit-testing-and-ui-testing-tutorial. Pristup ostvaren: 17.6.2020.

[90] »Swift Tests,« Dostupno: https://qualitycoding.org/xctestcase-teardown/. Pristup ostvaren: 17.6.2020.

[91] »The Order of Unit Tests,« Dostupno: https://www.appsdeveloperblog.com/the-order-of-unit-tests-in-xcode/. Pristup ostvaren: 17.6.2020.

[92] »Local Unit Tests,« Dostupno: https://developer.android.com/training/testing/unit-testing/local-unit-tests. Pristup ostvaren: 30.6.2020.

[93] »HTTPMethod.swift,« Dostupno: https://github.com/Alamofire/Alamofire/blob/master/Source/HTTPMethod.swift. Pristup ostvaren: 6.4.2020.

PRILOZI

Prilog 1. Pdf i docx diplomskog rada (na CD-u)

Prilog 2. Prikaz *Podspec* datoteke


```
{
    "name": "Alamofire",
    "version": "5.0.1",
    "license": "MIT",
    "summary": "Elegant HTTP Networking in Swift",
    "homepage": "https://github.com/Alamofire/Alamofire",
    "authors": {
    "Alamofire Software Foundation": "info@alamofire.org"
},
    "source": {
    "git": "https://github.com/Alamofire/Alamofire.git",
    "tag": "5.0.1"
},
    "documentation_url": "https://alamofire.github.io/Alamofire/",
    "platforms": {
    "ios": "10.0",
    "osx": "10.12",
    "tvos": "10.0",
    "watchos": "3.0"
},
    "swift_versions": [
    "5.0",
    "5.1"
    ],
    "source_files": "Source/*.swift",
    "frameworks": "CFNetwork",
    "swift_version": "5.1"
}
```

Prilog 3. Prikaz *Podfile.lock* datoteke

```
PODS:
  - Alamofire (5.0.1)
  - Kingfisher (5.13.0):
    - Kingfisher/Core (= 5.13.0)
  - Kingfisher/Core (5.13.0)
  - Swinject (2.7.1)
  - SwinjectStoryboard (2.2.0):
    - Swinject (~> 2.6)
  - Toast-Swift (5.0.1)

DEPENDENCIES:
  - Alamofire (~> 5.0.1)
  - Kingfisher (~> 5.13.0)
  - Swinject
  - SwinjectStoryboard
  - Toast-Swift (~> 5.0.1)

SPEC REPOS:
  trunk:
    - Alamofire
    - Kingfisher
    - Swinject
    - SwinjectStoryboard
    - Toast-Swift

SPEC CHECKSUMS:
  Alamofire: 41c162d645ba6a78e7fd030fcdaf6aa394ca39a0
  Kingfisher: 0d334cada987fcddbe9b5cec516406e1ee9d4748
  Swinject: ddf78b8486dd9b71a667b852cad919ab4484478e
  SwinjectStoryboard: 32512ef16c2b0ff5b8f823b23539c4a50f6d3383
  Toast-Swift: 9b6a70f28b3bf0b96c40d46c0c4b9d6639846711

PODFILE CHECKSUM: 42edcd48c5d9d1aba5caa52013d4c5d9a4f732d2

COCOAPODS: 1.8.4
```

POPIS POJMOVA

Android okolina za pokretanje aplikacija (*Android Runtime*): Okolina koja prevodi Dalvik bajtkod u instrukcije koje će se izvršavati na fizičkom procesoru. 3

Boolean: Tip podatka koji pohranjuje logičke vrijednosti istinu (true) i neistinu (false). 5

Char: Tip podatka koji pohranjuje neki od alfanumeričkih znakova. Za zapis koristi 8 bita. Adekvatan za ASCII znakove. Za internacionalne znakove, odnosno korištenje UNICODE skupa znakova, koristi se tip podatka string. 5

Dalvik bajtkod (engl. *dalvik executable bytecode,DEX*): Bajtkod kojeg korsiti Dalvik virtualni stroj za generiranje strojnog koda 3

Double: Tip podatka koji pohranjuje decimalne brojeve. Za zapis se koriste 64 bita te je raspon vrijednosti između 2.3E-308 do 1.7E+308. 5

Float: Tip podatka koji pohranjuje decimalne brojeve. Zapis uobičajeno pomoću 32 bita. Raspon vrijednosti između 1.2E-38 do 3.4E+38 5

Imenici (engl. *dictionary*): Tip podatka koji pohranjuje vrijednosti u parovima ključ vrijednost. Prilikom korištenja imenika tip podatka za ključ te tip podatka za vrijednost moraju biti istog tipa podatka koji se koristio prilikom definiranja imenika. 8

Int: Tip podatka koji pohranjuje cijele brojeve. Uobičajeno je zapis pomoću 32 bita. Raspon vrijednosti između -2,147,483,648 do 2,147,483,647 5

Napuhivač sučelja (engl. *inflater*): Mehanizam kojim se na Android platformi generiraju aplikacijska sučelja te povezuju na glavnu aktivnost koja se prikazuje na zaslonu mobilnog uređaja. 40

Obradno svojstvo (engl. *computed property*): Svojstvo na čiji će se poziv izvršiti obrada te vratiti rezultat. Slično funkciji koja ne prima parametar, a vraća vrijednost; sažetiji zapis. 13

Pohranjivo svojstvo (engl. *stored property*): Svojstvo koje služi za pohranu konstantne ili promjenjive vrijednosti tijekom izvođenja programa. 13

Povratni poziv (engl. *callback*): Poziv kojim se tijek izvođenja vraća u pozivajući kod. 15

Prevođenje prije pokretanja (engl. *ahead of time,AOT)*: Vrsta prevođenja gdje se cijeli izvorni kod podliježe prevođenju za određenu računalnu arhitekturu prije pokretanja na fizikalnom uređaju. Rezultira bržim izvođenjem programa u usporedbi s prevođenjem tijekom izvođenja, JIT. 3

prevođenje tijekom izvođenja (engl. *just in time,JIT)*: Metoda izvršavanja i prevođenja programskog koda u trenutcima kada je potrebna implementirana funkcionalnost, a koja još nije prevedena. Suprotan pristup JIT prevođenju jest prevođenje prije izvršavanja, AOT 3

Promatrač promjene vrijednosti (engl. *observer*): Promatrač koji izvršava nekakvu dodijeljenu radnju prilikom promjene vrijednosti nad određenim svojstvom. 13

String: Tip podatka koji pohranjuje znakovne nizove. Ako programski jezik omogućava rad s UNICODE znakovima, moguć je zapis internacionalnih znakova. Programski jezik može ograničiti količinu znakova koja se pohranjuje unutar takvog tipa podatka. 11

ŽIVOTOPIS

Petar Marić rođen je 19. travnja 1996. godine u Osijeku. 2011. godine upisuje III. Gimnaziju u Osijeku te je završava 2015. godine. Smjer Računarstvo upisuje 2015. godine na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. 2018. godine dobiva Dekanovu nagradu za uspjeh u studiranju. Iste godine upisuje diplomski smjer Programsko inženjerstvo na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku.