Razvoj modularne Android biblioteke

Perišić, Luka

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

Sveučilišni studij

RAZVOJ MODULARNE ANDROID BIBLIOTEKE

Završni rad

Luka Perišić

Osijek, 2019

SADRŽAJ

1. UVOD

Početkom projekta izrade aplikacije za Android operacijski sustav, potrebno je postaviti početne parametre i implementirati funkcionalnosti koje su zajedničke s prijašnjim ili budućim projektima iste arhitekture. Kako bi se ubrzao taj početak implementacije novog projekta osmišljena je biblioteka koja omogućuje automatsko postavljanje osnovnih parametara projekta, spajanje korisnika, postavljanje lokalne baze podataka, analitiku, pristup kameri i senzorima. Biblioteka kao takva se sastoji od već napisanog programskog koda i podataka, a korisnik, točnije programer, može ručno dodati programsku biblioteku i izravno koristiti njezinu funkcionalnost kako bi postigao željeni rezultat brže, smanjio kompleksnost koda ili pak automatizirao neki od potrebnih procesa bez pisanja ponavljajućeg (*engl. Boilerplate*) programskog koda [1].

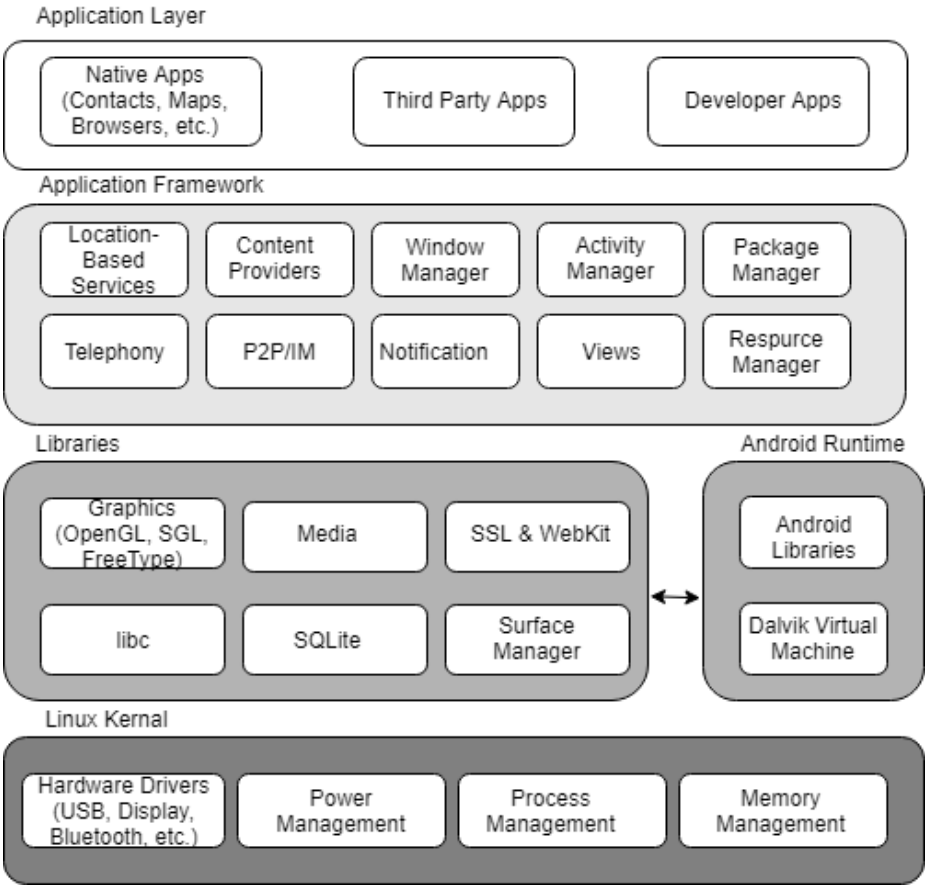
Razvijena biblioteka omogućava povezivanje s vanjskim servisom prema dogovorenim pravilima koji su uvjet funkcionalnosti i uspješne komunikacije. U drugom poglavlju su opisane korištene tehnologije i pomoćne biblioteke potrebne za razvoj biblioteke i potrebne za razvoj projekta koji koristi razvijenu biblioteku. U trećem poglavlju su definirani korisnički zahtjevi i specifikacije kao osnovna pravila funkcioniranja i pravila po kojima je smišljena arhitektura programskog koda. Nakon definiranja korisničkih zahtjeva opisan je razvoj biblioteke koji podrazumijeva korištenje tehnologija i pomoćnih biblioteka navedenih u drugom poglavlju.

Cilj završnog rada je prikazati stečena znanja iz područja razvoja programske podrške objektno orijentiranim načelima, prikazati postupak implementacije modularne arhitekture i učitavanja iste na vanjski repozitorij korištenjem najnovijih dostupnih tehnologija za razvoj biblioteke Android operacijskog sustava.

2. PREGLED KORIŠTENIH TEHNOLOGIJA

2.1. Android operacijski sustav

Android se prvi puta na tehnološkom radaru pojavljuje 2005. godine kada tvrtka Google kupuje tvrtku Android. Tada se još nije puno znalo o Androidu niti što Google planira napraviti s njim. Tek 2007. Google najavljuje prvu pravu platformu otvorenog koda (*engl. Open Source*) za mobilne uređaje. Android operacijski sustav je tehnologija otvorenog koda koja se dnevno pokreće na više od dvije milijarde uređaja. Ova se tehnologija sastoji od različitih komponenti i omogućuje razvojnim programerima da rade odvojeno [2]. Arhitektura Android operacijskog sustava se sastoji od nekoliko slojeva, to pokazuje slika 2.1. , ali za razvoj biblioteke posebno je važan aplikacijski sloj.



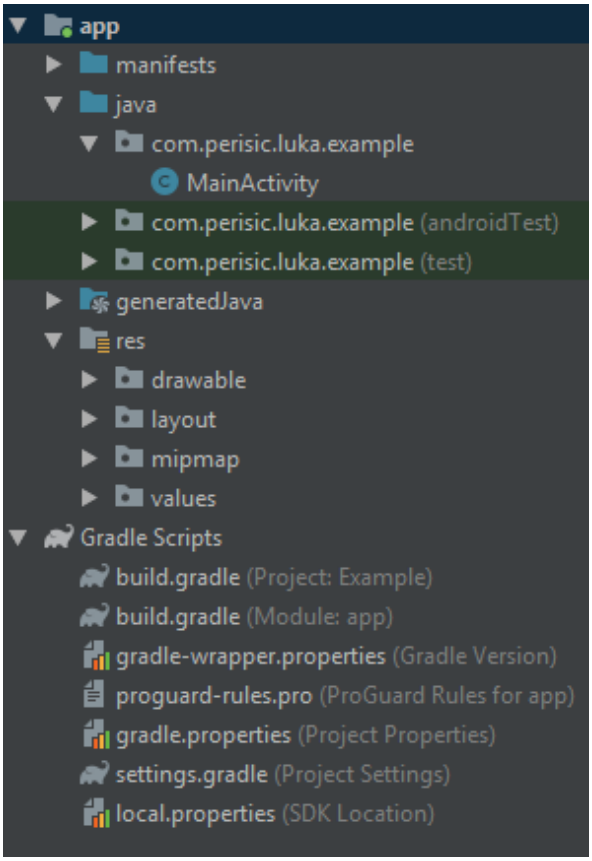
Slika 2.1. *Arhitektura Android operacijskog sustava*

2.2. Razvojno okruženje Android Studio

Android Studio je službeni *Integrated Development Environment* (IDE) za razvijanje Android aplikacija. Baziran je na IntelliJ platformi koja pruža poboljšanje u uređivanju koda i dodatne alate

za razvijanje Android aplikacija [3]. Jedna od značajki koje Android Studio posjeduje je Gradle građenje sustava [4]. Ova značajka je vrlo bitna za razvoj Android aplikacija jer je građenje aplikacija s ovim alatom, fokusiranim na fleksibilnost i učinkovitost, jednostavnije i također je omogućeno lakše uređivanje skripti za građenje.

Kreiranje novog projekta s jedinstvenim imenom paketa koji predstavlja aplikaciju rezultira strukturom datoteka (slika 2.2). Iz naveden slike se vidi kako se unutar imena paketa nalazi Android aktivnost koja predstavlja jedan ekran unutar prozora aplikacije i brine o kreiranju prozora u koji se stavljaju različiti elementi grafičkog korisničkog sučelja [5]. Također unutar se te strukture mogu nalaziti i ostale datoteke potrebne za razvijanje aplikacije. Poseban dio strukture bitan za razvijanje biblioteke je dio koji se nalazi unutar Gradle skripti (*engl. Gradle Scripts*) i u njemu se nalaze dvije *build.gradle* datoteke od kojih je jedna skripta za građenje na nivou aplikacije i jedna skripta koja služi za građenje *app* modula. Dodavanjem novog modula u projekt, Android Studio kreira i novu *build.gradle* skriptu.



Slika 2.2. *Struktura datoteka novog Android projekta*

Iz primjera *build.gradle* datoteke (*programski kod 2.1.*) se vidi kako kreiranjem novog modula izgleda jedna osnovna generirana skripta. Naredba *applicationId* unutar bloka *android* definira ime paketa, a unutar bloka *dependencies* se nalaze ovisnosti modula na vanjske android biblioteke.

Naredbom implementation se uvodi vanjska biblioteka čije se funkcionalnosti koriste i vidljive su samo modulu u čijoj gradle skripti se nalaze. Ako se koristi naredba *api* funkcionalnosti su otvorene za korištenje i modulima koji ovise o modulu gdje se nalazi navedena naredba. Naredbe *testImplementation*, *androidTestImplementation* i *annotationProcessor*, koje je objašnjene u sljedećem potpoglavlju, ne posjeduju svojstvo tranzitivne ovisnosti i prilikom dodavanja ovisnosti na drugi modul koji sadrži biblioteke kojima su potrebne ovisnosti s *annotationProcessor* naredbom, potrebno ih je zatim dodati u gradle skriptu gdje se nalazi ovisnost na drugi modul.

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 28
    defaultConfig {
        applicationId "com.perisic.luka.example"
        minSdkVersion 23
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
                    'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
}
```

Programski kod 2.1. *Build.gradle skripta app modula*

2.3. Java programski jezik

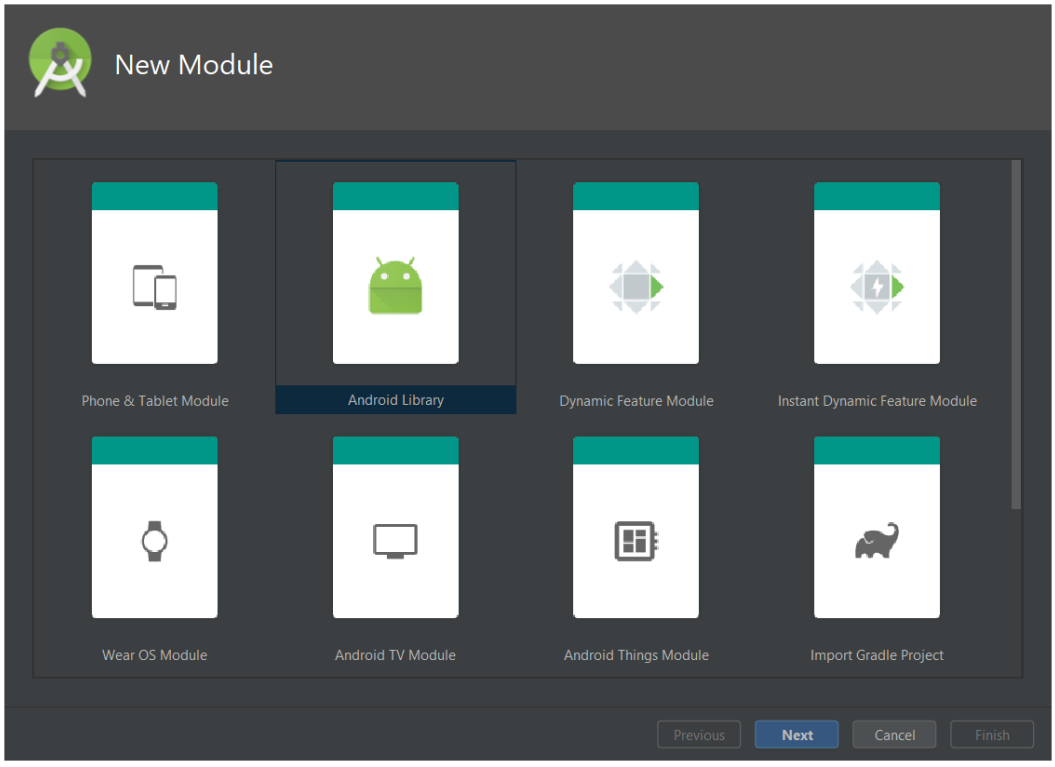
Java je objektno orijentirani, programski jezik baziran na klasama. Za razvoj su zaslužni James Gosling i Patrick Naughton iz tvrtke Sun Microsystems. Prvo prikazivanje programskog jezika je bilo 23. svibnja 1995. godine. Danas je aktualna verzija Java 12. Prednost Jave naspram ostalih programskih jezika je što se programi pisani u Javi mogu izvoditi bez ikakvih promjena na svim operacijskim sustavima za koje postoji JVM (*engl. Java Virtual Machine*) [6].

Java pruža odličan stupanj sigurnosti i pouzdanosti zahvaljujući VM-u (*engl. Virtual Machine*) i zatvorenom okolišu u kome svaki program operira (rad programa se ograničava samo na VM). Java sučelje (*engl. interface*) korišteno u razvoju biblioteke je predložak/nacrt klase i može sadržavati apstraktne metode bez tijela u svrhu postizanja pune apstrakcije i višestrukog

nasljeđivanja. Anotacije pridružuju meta-informacije paketima, klasama, metodama, atributima, parametrima, konstruktorima ili lokalnim varijablama tako da bi se prilikom izvođenja programskog koda dodijelili dodatni meta podatci. Od korištenih svojstava Java programskog jezika korišteni su i lambda izrazi koji izražavaju primjere funkcionalnih sučelja (sučelje s jednom apstraktnom metodom), implementiraju jednu apstraktnu metodu i stoga implementiraju funkcionalna sučelja.

2.4. JitPack repozitorij paketa

JitPack je repozitorij paketa za JVM ili Android projekte. Kreira projekte na zahtjev i poslužuje spremne za korištenje .jar ili .arr artefakte [7]. Nakon kreiranja android projekta potrebno je kreirati novi android modul, prilikom kreiranja odabrati Android Library (slika 2.3.) i u sljedećem koraku unijeti željeno ime.



Slika 2.3. *Prozor za kreiranje novog Android modula*

U korijenskoj *build.gradle* datoteci potrebno je dodati ovisnost (*engl. dependency*) (programski kod 2.2.).

```
buildscript {
    //...
    dependencies {
        classpath 'com.github.dcendents:android-maven-gradle-plugin:2.1'
        //...
    }
}
```

Programski kod 2.2. *Ovisnost na potreban dodatak*

U *build.gradle* datoteci novog Android modula potrebno je omogućiti dodatak (*engl. plugin*) i navesti ime grupe uz pomoć kojega se kasnije referencira ovisnost (programski kod 2.3.).

```
apply plugin: 'com.github.dcendents.android-maven'

group='com.github.lperisicz.AndroidModularStarter'
```

Programski kod 2.3. *Omogućen dodatak*

Nakon toga potrebno je kreirati izdanje (*engl. release*) na GitHub-u s nekim imenom i željeni modul će biti dostupan za korištenje (programski kod 2.4.).

```
api "com.github.username.project_name:module_name:TAG"
```

Programski kod 2.4. *Generalni primjer korištenja objavljene zavisnosti*

2.5. Room baza podataka

Room je odabrana biblioteka za rad s bazom podataka jer djeluje kao apstrakcijski sloj iznad ugrađene SQLite baze podataka i omogućava jednostavno korištenje generirajući kod potreban za komunikaciju s ugrađenom bazom [8]. Postupak korištenja se sastoji od kreiranja klase entiteta i stavljanjem anotacije *@Entity*, koja predstavlja jednu tablicu podataka (programski kod 2.5.), zatim objekta pristupanja domeni (*engl. Data Access Object,* DAO) koja je još jedna anotirana klasa i predstavlja posrednika između SQLite baze s metodama koje sadrže anotacije *@Insert*, *@Query*, *@Delete* i druge. Implementacija objekta pristupanja domeni prikazana je u programskom kodu 2.6.

Svaki entitet mora imati definiran barem jedan atribut s anotacijom *@PrimaryKey* koja služi da bi SQLite baza znala koji atribut klase predstavlja primarni ključ. Kao što anotacija *@Entity* ima atribut *tableName,* ime tablice koja se kreira, tako i pojedini atribut može imati anotaciju @ColumnInfo u kojem se predaje ime kolone, s atributom *name,* i kao takva će biti upisana u SQLite bazi. Klasa anotirana sa *@Dao sadrži* anotaciju *@Query* u kojoj je napisana SQL (*engl. Structured Query Language*) naredba koja opisuje zadani upit na bazu i vraća definirani rezultat.

```java
@Entity(tableName = "UserData")
public class UserModel {

    @PrimaryKey
    @ColumnInfo(name = "id")
    private int id;

    @ColumnInfo(name = "username")
    private String username;

    @ColumnInfo(name = "email")
    private String email;

}
```

Programski kod 2.5. *Klasa koja predstavlja podatke o korisniku*

```java
@Dao
public abstract class UserModelDao implements BaseModelDao<UserModel> {

    @Query("SELECT * FROM UserData")
    public abstract UserModel getUser();

    @Query("DELETE FROM UserData")
    public abstract void deleteUser();

    @Transaction
    public void saveUser(UserModel userModel) {
        deleteUser();
        insert(userModel);
    }

}
```

Programski kod 2.6. *UserModelDao klasa objekta pristupanja domeni*

2.6. Retrofit klijent

Retrofit je REST klijent za Android i Javu koji omogućava relativno jednostavno dohvaćanje i slanje JSON (*engl. JavaScript Object Notation*) ili drugih strukturiranih podataka preko servisa baziranih na REST-u [9]. Kako bi se definiralo biblioteci kako dohvatiti podatke prvo je potrebno definirati metode Retrofit servisa. Svaka metoda mora sadržavati HTTP anotaciju koja opisuje metodu zahtjeva i relativnu putanju [10]. Postoji 5 ugrađenih anotacija [11]:

- *@GET*
- *@POST*
- *@PUT*
- *@DELETE*
- *@HEAD*

Relativna putanja definira se kao atribut jednoj od navedenih anotacija (programski kod 2.7.). U istom programskom kodu definirana je i povratna vrijednost unutar izlomljenih zagrada

bibliotekom predefinirane generične klase *Call* uz koju se pri pozivu predaje uzvratni poziv (*engl. Callback*) koji se okida kada se HTTP zahtjev izvrši.

```
@POST("post/{id}")
Call<Post> getPost(int id);
```

Programski kod 2.7. *Primjer Retrofit metode*

Prema programskom kodu 2.8. vidljivo je kreiranje Retrofit objekta kojemu je predan osnovni dio putanje na vanjski REST servis i kreiranje Retrofit servisa *PostService* koji je sučelje koje sadrži metodu iz programskog koda 2.7. Pozivom metoda iz kreiranog servisa se dohvaćaju ili šalju podatci na vanjski servis.

```
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("https://sails-test-1.herokuapp.com/api/")
        .build();

PostService service = retrofit.create(PostService.class);
```

Programski kod 2.8. *Primjer kreiranja Retrofit objekta i Retrofit servisa*

2.7. Dagger injekcija ovisnosti

Dagger biblioteka je po definiciji brzi ubrizgavač zavisnosti (*engl. Dependency Injector*) za Java i Android okružje. Kako bi se shvatio koncept ubrizgavanja zavisnosti potrebno je poznavati princip inverzije kontrole (*engl. Inversion of control*). Ovaj princip predstavlja slučaj kada objekti neke klase ne kreiraju objekte druge klase o kojima zavise, već te objekte dobivaju iz vanjskih izvora, u ovom slučaju iz Dagger okvira (*engl. Framework*). Prema inverziji kontrole objekt jedne klase ne treba znati kako kreirati objekte druge klase o kojoj zavisi i Dagger nam zbog toga pruža mehanizme ubrizgavanja zavisnosti u konstruktor klase, njezine atribute i u metodu. Kako bi biblioteka znala gdje ubrizgavati zavisnosti potrebno je dodati anotaciju *@Inject* na mjesto ubrizgavanja. Programski kod 2.9. pokazuje primjer ubrizgavanja objekta klase *String* u konstruktor klase *Example*.

```
class Example {

    private final String dependency;

    @Inject
    public Example(String dependency) {
        this.dependency = dependency;
    }

}
```

Programski kod 2.9. *Primjer klase sa zavisnosti u konstruktoru*

Nakon toga je potrebno zadovoljiti zavisnosti gdje Dagger zauzima memoriju za potrebne objekte. Ako biblioteka sama zna kako kreirati objekt ona će to napraviti. U drugim slučajevima

postoji mogućnost da je zavisnost sučelje, može pripadati nekoj drugoj vanjskoj biblioteci ili je jedan od argumenata potreban za kreiranje objekta jedan prethodno navedeni slučaj. Taj problem se rješava uvođenjem Dagger modula (*engl. Module)*, klase s anotacijom *@Module*, te dodavanjem metoda koje služe kao usluživači (*engl. Provider*). Programski kod 2.10. prikazuje modul *Example* i njegovu metodu *provideDependency* koja uslužuje potreban objekt. Anotacija *@Singletone* je ugrađena anotacija koja govori kako pri svakom kreiranju objekta koji ima zavisnost na neki objekt, odnosno usluživač, će uslužiti uvijek isti objekt, odnosno objekt s jedinstvenom instancom (*engl. Singleton*).

```
@Module
public abstract class Example {

    @Singleton
    @Provides
    static String provideDependency() {
        return "Example";
    }

}
```

Programski kod 2.10. *Primjer Dagger modula koji uslužuje zavisnost*

Ako su nam potrebna dva ili više različitih objekata iste klase, samim usluživanjem metodama (programski kod 2.10.), biblioteka ne razumije gdje koji objekt treba uslužiti. Taj se problem rješava uvođenjem kvalifikatora. Kvalifikatori su anotacije koje su anotirane s anotacijom *@Qualifier*. Programski kod 2.11. pokazuje kvalifikatore *Name* i *Surname*. U istom programskom kodu se vidi i primjer modula koji uslužuje dva objekta klase *String*. Isti objekti se ubrizgavaju u konstruktor klase *Example* i uz pomoć kvalifikatora biblioteka zna koji objekt treba uslužiti. Nakon definiranja potrebnih modula i njihovih zavisnosti, biblioteka generira programski kod koji imitira onaj koji bi korisnik ručno mogao unijeti i pri tome pazi na jednostavnost i brzinu izvođenja.

```java
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface Name {

}

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface Surname {

}

@Module
public abstract class Example {

    @Provides
    @Singleton
    @Name
    static String provideName() {
        return "Ime";
    }

       @Provides
    @Singleton
    @Surname
    static String provideName() {
        return "Prezime";
    }

}

@Module
public abstract class Example {

    private String name;
    private String surname;

    @Inject
    public Example(@Name String name, @Surname String surname) {
        this.name = name;
        this.surname = surname;
    }

}
```

Programski kod 2.11. *Primjer korištenja kvalifikatora*

3. DEFINIRANJE KORISNIČKIH ZAHTJEVA I SPECIFIKACIJA

3.1. Korisnički zahtjevi

Prije početka implementacije potrebno je utvrditi koji su korisnički zahtjevi i zahtjeve koje korisnik mora ispuniti, ako želi koristiti biblioteku. Korisnik najefikasnije i uz minimalno programskog koda želi započeti novi projekt u kojem postoje funkcionalnosti definirane korisničkim zahtjevima.

3.2. Autorizacija korisnika

Prema korisničkim zahtjevima je utvrđeno kako se autorizacija vrši preko pristupnih tokena i korisniku je omogućeno automatsko rukovanje s pristupnim tokenom unutar biblioteke bez dodatnog programskog koda. Autorizacija se sastoji od para tokena:

- Pristupni token (*engl. Access token*) koji se šalje u zaglavlju sa svakim zahtjevom na servis gdje je definirano da je on potreban. Servisu služi kao ključ uz pomoć kojeg zna tko šalje zahtjev i sukladno tome zna kakav rezultat treba poslati.
- Token za osvježavanje (*engl. Refresh token*) koji postoji zbog sigurnosnih razloga.

Pristupni token ima svoj životni vijek čija je duljina definirana servisom i ako slučajno dođe do napada na sustav, odgovorne će osobe samo do isteka pristupnog tokena imati pristup na servis [12]. Zbog isteka pristupnog tokena postoji poziv na servis gdje se u tijelu poziva šalje token za osvježavanje i u odgovoru dobiva novi par tokena. Kako bi biblioteka uspješno odrađivala posao, omogućeno je automatsko slanje poziva za osvježavanje i spremanje pristupnih tokena u lokalnu bazu podataka kako ne bi korisnik prilikom sljedećeg paljenja aplikacije ponovno morao prolaziti kroz proces autorizacije.

3.3. Predefinirani zahtjevi na servis

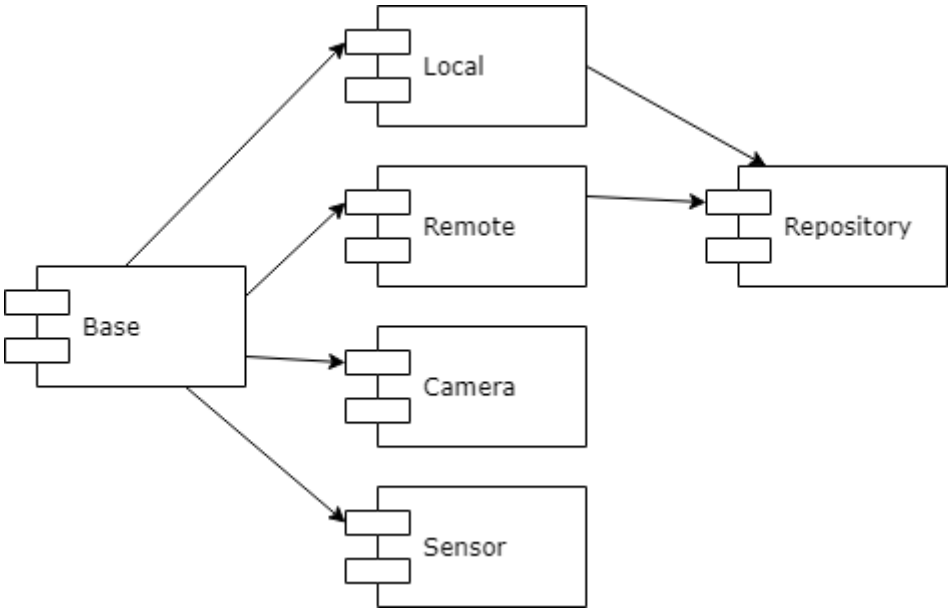
Korisničkim zahtjevima je definirana autorizacija koja se sastoji od para tokena, ali kako bi korisnik dobio tokene potrebno je omogućiti mehanizam registracije i pristupa već registriranim korisnicima aplikacije. Kako se postupak ne razlikuje između aplikacija, korisničkim zahtjevima je određeno da su svi pozivi na servis povezani s autorizacijom implementirani unutar biblioteke. Kako svaka aplikacija napravljena uz pomoć biblioteke može kontaktirati s različitim servisima, tako je i potrebno dodati korisniku mogućnost postavljanja osnovne putanje na servis.

4. RAZVOJ MODULARNE ANDROID BIBLIOTEKE

4.1. Arhitektura biblioteke

Kako su korisnički zahtjevi takvi da je omogućeno pristupanje pojedinom modulu uz pomoć JitPack dodatka, biblioteka je podijeljena na više modula. Arhitekturu čine sljedeći moduli (slika 4.1.):

- *Base* modul predstavlja modul zajednički za sve ostale module i sadrži zavisnosti koje su potrebne svim modulima.
- *Local* modul predstavlja dio biblioteke koja se brine o lokalnoj bazi i omogućuje pristup te također zavisi na modul *Base*.
- *Remote* modul sadrži sučelja za komunikaciju s vanjskim servisom.
- *Repository* modul predstavlja apstrakcijski sloj iznad dohvaćanja i spremanja podataka. Koristi sučelja za komunikaciju sa servisom iz *Remote* modula i operacije s lokalnom bazom iz *Local* modula.
- *Camera* modul sadrži ovisnosti na biblioteku za fotografiranje i zbog korisničkog sučelja zavisi na modul *Base*
- *Sensor* modul omogućava očitavanje senzora i sadrži zavisnost na *Base* modul



Slika 4.1. *Prikaz arhitekture biblioteke*

4.2. Base modul

Kako svi moduli posjeduju zajedničke zavisnosti, te iste zavisnosti odvojene su u poseban modul. Odvajanje zavisnosti olakšava praćenje i verzioniranje s jednog mjesta. Također promjenom implementacije određenih dijelova modula svi koji imaju zavisnost na modul *Base* će dobiti novu implementaciju.

4.3. Local modul

Ovaj modul brine o operacijama nad lokalnom bazom podataka i zbog zahtjeva korisnika on omogućava spremanje i dohvaćanje podataka o korisniku i podataka o pristupnim tokenima. Kako bi se spremilo i upravljalo korisnikovim podatcima kreirana je klasa *UserModel* s anotacijom *@Entity*. Prema programskom kodu 4.2. korisnik sadrži atribut *id*, koji predstavlja primarni ključ u tablici te atribute *username* i *email*.

```
@Entity(tableName = "UserData")
public class UserModel {

    @PrimaryKey
    @ColumnInfo(name = "id")
    private int id;

    @ColumnInfo(name = "username")
    private String username;

    @ColumnInfo(name = "email")
    private String email;

    //...

}
```

Programski kod 4.1. *Podatci o korisniku unutar lokalne baze*

Pri kreiranju se koristi pomoćno generično sučelje *BaseModelDao* (programski kod 4.2.) kojega će naslijediti svako kreirano *Dao* sučelje jer svaki objekt pristupanja domeni posjeduje mogućnost umetanja, ažuriranja i brisanja podataka. Prilikom prevođenja kompajler zamjenjuje generični tip parametra *T* s predanom klasom.

```java
public interface BaseModelDao<T> {

    @Insert(onConflict = REPLACE)
    void insert(T model);

    @Update
    void update(T model);

    @Delete
    void delete(T model);

}
```

Programski kod 4.2. *Pomoćno sučelje za objekte pristupanja domeni*

Prema programskom kodu 4.3. kreirana je i klasa *TokenModel* s dodatnim atributom *id* kojemu je vrijednost uvijek 1 zato što pristupni token ne posjeduje taj atribut i u svakom trenutku postoji samo jedan objekt spremljen u lokalnoj bazi. Umetanjem novog pristupnog tokena u lokalnu bazu, stari se briše i umeće novi pod istim primarnim ključem.

```java
@Entity(tableName = "TokenData")
public class TokenModel {

    @PrimaryKey
    @ColumnInfo(name = "id")
    private int id = 1;

    @ColumnInfo(name = "token")
    private String token;

    @ColumnInfo(name = "refreshToken")
    private String refreshToken;

    //...

}
```

Programski kod 4.3. *Podatci pristupnih tokena unutar lokalne baze podataka*

Prema pravilima iz poglavlja 2.5 kreirani su objekti pristupanja domeni za podatke o korisniku i podatke o pristupnim tokenima. Programski kod 4.4. prikazuje kreirani objekt pristupanja domeni za podatke od korisniku. Odjavljivanjem trenutnog korisnika aplikacije i prijave novoga je potrebno obrisati stare podatke i zapisati nove. Kako bi se omogućila takva funkcionalnost kreirana je metoda *saveUser* koja prvo briše sve podatke o korisniku iz baze i zatim upisuje novoga korisnika.

```
@Dao
public abstract class UserModelDao implements BaseModelDao<UserModel> {

    @Query("SELECT * FROM UserData")
    public abstract UserModel getUser();

    @Query("DELETE FROM UserData")
    public abstract void deleteAllData();

    @Transaction
    public void saveUser(UserModel userModel) {
        deleteAllData();
        insert(userModel);
    }

}
```

Programski kod 4.4. *Klasa objekta pristupanja domeni podataka o korisniku*

Korisničkim zahtjevom je definirano da kreiranje i usluživanje objekta lokalne baze i Dao objekata obavlja biblioteka. To je omogućeno dodavanjem Dagger modula *BaseRoomModule* (programski kod 4.5.). Modul uslužuje lokalnu bazu metodom koja prima kontekst aplikacije, ime baze i klasu koja predstavlja lokalnu bazu. Također uslužuje objekt klase *UserModelDao* uz usluženu lokalnu bazu.

```
@Module
abstract class BaseRoomModule {

    @Provides
    @Singleton
    static BaseLocalDatabase provideLocalDatabase(
            @ApplicationContext Context applicationContext,
            @DatabaseName String databaseName,
            @DatabaseClass Class<? extends BaseLocalDatabase> databaseClass
    ) {
        return Room.databaseBuilder(applicationContext, databaseClass, databaseName)
                .allowMainThreadQueries()
                .fallbackToDestructiveMigration().build();
    }

    @Provides
    @Singleton
    static UserModelDao provideUserModelDao(BaseLocalDatabase baseLocalDatabase) {
        return baseLocalDatabase.userModelDao();
    }
```

Programski kod 4.5. *Dagger modul koji uslužuje BaseLocalDatabase i UserModelDao*

4.4. Remote modul

Svi podatci dolaze s vanjskog servisa i kako bi se podatci mogli primati i slati potreban je Retrofit klijent. Također klijentskim zahtjevima je definirano da kreiranje servisa obavlja biblioteka, stoga je kreiran novi Dagger modul. Iz poglavlja 2.6 je vidljivo kako Retrofit biblioteka omogućava slanje različitih struktura podataka i ovisno o kojoj strukturi je riječ je potrebno dodati biblioteku koja pruža konverter za rukovanje potrebnom strukturom. U ovom slučaju se koristi

biblioteka *Gson* koja upravlja sa JSON strukturom podataka. Konverter se uslužuje kao programskom kodu 4.6.

```
@Provides
@Singleton
static Converter.Factory provideConverterFactory() {
    return GsonConverterFactory.create(
            new GsonBuilder()
                    .setLenient()
                    .create()
    );
}
```

Programski kod 4.6. *Metoda koja uslužuje konverter JSON strukture podataka*

Uz konverter je potreban i HTTP klijent (programski kod 4.7.). Prima *ServiceInterceptor*, koji se brine da prije svakog zahtjeva na vanjski servis u zaglavlje zahtjeva (*engl. Header*) stavi pristupni token. Također prima i *TokenAuthenticator* koji zahtjeva i sprema novi token za osvježavanje kada se pojavi odgovor sa servisa koji govori da je pristupni token istekao.

```
@Provides
@Singleton
static OkHttpClient provideOkHttpClient(
        ServiceInterceptor serviceInterceptor,
        TokenAuthenticator tokenAuthenticator
) {
    return new OkHttpClient.Builder()
            .addInterceptor(serviceInterceptor)
            .authenticator(tokenAuthenticator)
            .build();
}
```

Programski kod 4.7. *Metoda koja uslužuje HTTP klijent*

Retrofit klijent se potom kreira uz pomoć usluženog HTTP klijenta i konvertera. Programski kod 4.8. pokazuje usluživanje Retrofit klijenta uz dodatno usluženu osnovnu rutu koja na sebi sadrži kvalifikator *@BaseUrl*. Usluživanje osnovne rute, prema klijentskim zahtjevima, omogućuje klijentska strana prilikom korištenja *Remote* modula.

```
@Provides
@Singleton
static Retrofit provideRetrofit(
        OkHttpClient okHttpClient,
        Converter.Factory converterFactory,
        @BaseUrl String baseUrl
) {
    return new Retrofit.Builder()
            .baseUrl(baseUrl)
            .addConverterFactory(converterFactory)
            .client(okHttpClient)
            .build();
}
```

Programski kod 4.8. *Metoda koja uslužuje Retrofit klijent*

Nakon kreiranja klijenta je potrebno kreirati sučelja koja će omogućiti komunikaciju sa servisom. Prvo je kreirano sučelje *AuthService* (programski kod 4.9.) koje sadrži metode za osvježavanje tokena i spajanje korisnika. Dodatci na rute poziva su izvučeni u posebnu klasu.

```
public interface AuthService {

    @Headers(Config.NO_JWT_AUTH_HEADER)
    @POST(ROUTE_REFRESH_TOKEN)
    Call<BaseResponse<RefreshTokenResponse>> refreshToken(
        @Body RefreshTokenRequest refreshTokenRequest
    );

    @Headers(Config.NO_JWT_AUTH_HEADER)
    @POST(ROUTE_LOGIN)
    Call<BaseResponse<LoginResponse>> login(@Body LoginRequest request);

}
```

Programski kod 4.9. *Sučelje za komunikaciju sa servisom*

Kako bi klijent uz pomoć kreiranog Gradle modula mogao koristiti servise koji su kreirani u biblioteci, kreiran je novi Dagger modul nazvan *BaseServiceModule* koji uslužuje objekte servisa. Programski kod 4.10. prikazuje kreirani Dagger modul i metodu koja uslužuje servis uz usluženi Retrofit klijent (programski kod 4.9.).

```
@Module
public abstract class BaseServiceModule {

    @Provides
    @Singleton
    static AuthService providesAuthService(@Authentication Retrofit retrofit){
        return retrofit.create(AuthService.class);
    }
}
```

Programski kod 4.10. *Dagger modul za usluživanje servisa za autorizaciju*

4.5. Repository modul

Nakon implementacije *Local* i *Remote* modula preostaje implementacija *Repository* modula. Ovaj modul sakriva funkcionalnosti dohvaćanja i spremanje podataka od korisnika i samim time daje jednostavnost promijene implementacije pružajući interakciju preko sučelja. Prema zahtjevu korisnika, prvi podatak koji se vraća nakon poziva na servis je podatak sa statusom učitavanje i zbog tog je kreirano sučelje *BaseRepository* (programski kod 4.11.) kojega nasljeđuje svaki repozitorij. U sučelju se nalazi metoda *executeCall* koja prima Retrofit poziv i uzvratni poziv. Metoda obavlja poziv na servis i na drugoj niti (*engl. Thread*) dobiva rezultat koji se postavlja u povratnu vrijednost. Početnu vrijednost koju metoda vraća je osnovni podatak s praznim *data* atributom i statusom učitavanje. Sukladno rezultatu poziva na servis postavlja se povratna vrijednost. U slučaju da je dobiven rezultat u metodi *onResponse* povratnoj vrijednosti se postavlja

vrijednost odgovora, a ako dobiveni rezultat sadrži grešku, metodom *parseRetorofitErrorBody* iz pomočnog sučelja, se postavlja vrijednost sa statusom greška.

```
default <T> LiveData<BaseResponse<T>> executeCall(Call<BaseResponse<T>> call, @NonNull Observer<T> callback) {
    NetworkLiveDataSource<T> apiResponse = new NetworkLiveDataSource<>();
    call.enqueue(new Callback<BaseResponse<T>>() {
        @Override
        public void onResponse(@NonNull Call<BaseResponse<T>> call, @NonNull Response<BaseResponse<T>> response) {
            if (response.body() != null) {
                if (response.body().getData() != null) {
                    callback.onChanged(response.body().getData());
                }
                apiResponse.setValue(response.body(), BaseData.Status.DONE);
            } else if (response.errorBody() != null) {
                apiResponse.setValue(parseRetrofitErrorBody(response.errorBody()), BaseData.Status.ERROR);
            }
        }

        @Override
        public void onFailure(@NonNull Call<BaseResponse<T>> call, @NonNull Throwable t) {
            apiResponse.setValue(new BaseResponse<>(BaseData.Status.ERROR));
        }
    });
    return apiResponse.startWithLoading();
}
```

Programski kod 4.11. *Pomoćna metoda iz BaseRepository sučelja*

U programskom kodu 4.12. je prikaz repozitorija za autorizaciju kojeg predstavlja sučelje s metodom za prijavu korisnika u sustav.

```
public interface AuthRepository extends BaseRepository {

    LiveData<BaseResponse<LoginResponse>> login(LoginRequest loginRequest);

}
```

Programski kod 4.12. *Sučelje repozitorija za autorizaciju korisnika*

Metoda *login* prima jedan parametar koji predstavlja nužne podatke koji se šalju na servis kako bi obavili uspješnu prijavu. U ovom slučaju to je email adresa i lozinka koji se nalaze u objektu klase *LoginRequest*. Metoda vraća *LiveData* klasu nositelja podataka klase unutar izlomljenih zagrada. U ovom slučaju nosi podatke klase *LoginResponse*. Ova klasa ima mogućnost promatranja od strane Android aktivnosti ili fragmenta gdje promatrač dobiva zadnju postavljenu vrijednost kada im se promijeni stanje. U programskom kodu 4.13. je prikazana implementacija repozitorija koji u konstruktoru prima servis za autorizacija i *TokenModelDao* klasu koja brine o dohvaćanju i spremanju pristupnih token iz lokalne baze podataka. Prilikom korisnikovog zahtjeva za prijavu prema naslijeđenoj metodi *executeCall* (programski kod 4.11.) korisnik u početku dobiva podatak kojem je status za učitavanje i unutar povratnog poziva dobije rezultat sa servisa i naknadno u predanom povratnom pozivu sprema pristupni token i token za osvježavanje uz pomoć metoda iz *TokenModelDao* klase.

```
public class AuthRepositoryImpl implements AuthRepository {

    private AuthService authService;
    private TokenModelDao tokenModelDao;

    @Inject
    AuthRepositoryImpl(AuthService authService, TokenModelDao tokenModelDao) {
        this.authService = authService;
        this.tokenModelDao = tokenModelDao;
    }

    @Override
    public LiveData<BaseResponse<LoginResponse>> login(LoginRequest loginRequest) {
        return executeCall(
                authService.login(loginRequest),
                loginResponse -> tokenModelDao.insert(
                        new TokenModel(
                                loginResponse.getToken(),
                                loginResponse.getRefreshToken())
                )
        );
    }
}
```

Programski kod 4.13. *Implementacija repozitorija za autorizaciju korisnika*

4.6. Sensor modul

Većina uređaja pokretana Android operacijskim sustavom posjeduju ugrađene senzore za orijentaciju, pokret, osvjetljenje i druge. Ovi senzori omogućavaju dohvaćanje očitanja s velikom preciznošću i točnošću. Android platforma podržava tri kategorije senzora [13]:

- Senzori za pokret
- Okolišni senzori
- Senzori položaja

Paket android.hardware ugrađen u standardni androidx paket koji je nužan pri kreiranju novog Android projekta omogućuje pristup klasama i sučeljima:

- *SensorManager* klasa koja se koristi za kreiranje instanci servisa koja omogućuje razne metode za pristup i dohvaćanje senzora i registraciju slušatelja događaja (*engl. Event listener*). Također pruža nekoliko konstanti koje definiraju točnost i stopu prikupljanja podataka sa senzora.
- *Sensor* klasa koje se kreira za određeni senzor koji se koristi.
- *SensorEvent* klasa zaslužena za kreiranje instanci događaja koji vraćaju očitanja sa senzora.
- *SensorEventListener* sučelje koje omogućuje slušanje na promjene vrijednosti očitavanja i promijene točnosti senzora.

Korisničkim zahtjevima je definirano da se biblioteke mora pobrinuti o mjerenju vrijednosti osvjetljenja tako da pruža pristup novoj Android aktivnosti koja se brine o dohvaćanju podataka i spremanju podataka pri povratku na prethodnu aktivnost. Kako bi se očitale vrijednosti sa senzora prvo je potrebno kreirati instancu *SensorManager* klase i instancu klase *Sensor* uz pomoć prethodno navedene instance. Kreiranje objekata klasa se vrši unutar metode *onCreate* (programski kod 4.14.).

```
private Sensor lightSensor;
private SensorManager sensorManager;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_sensor);
    ...
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    lightSensor = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
}
```

Programski kod 4.14. *Kreiranje instanci SensorManager i Sensor klase*

Nakon kreiranja potrebnih instanci se registrira sučelje slušatelj događaja za čiju je instancu potrebno implementirati dvije metode *onSensorChanged* koja sluša na promjenu očitanja senzora i *onAccuracyChanged* metodu koja sluša na promjenu točnosti očitavanja. Registracija slušatelja se poziva metodom *registerListener* koja prima sučelje za slušanje, senzor i ugrađenu konstantu stope skupljanja podataka unutar metode *onResume*. Nasljeđivanjem sučelja aktivnost implementira prethodno navedene metode i sprema zadnju očitanu vrijednost u varijablu *luxValue* (programski kod 4.15.).

```
public class SensorActivity extends AppCompatActivity implements SensorEventListener {

    ...

    private float luxValue = 0f;

    @Override
    protected void onResume() {
        super.onResume();
        sensorManager.registerListener(
            this,
            lightSensor,
            SensorManager.SENSOR_DELAY_NORMAL
        );
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
        luxValue = event.values[0];
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {

    }

}
```

Programski kod 4.15. *Registracija slušatelja i spremanje zadnje očitane vrijednosti*

Prema programskom kodu 4.16. pri povratku iz aktivnosti kao rezultat se postavlja vrijednost zadnjeg očitavanja koju zatim dobiva prethodna aktivnost u metodi *onActivityResult*.

```
Intent intent = new Intent();
intent.putExtra(RESULT_KEY, luxValue);
setResult(RESULT_OK, intent);
finish();
```

Programski kod 4.16. *Spremanje vrijednosti osvjetljenja prilikom povratka*

4.7. Camera modul

Korisničkim zahtjevima je određeno kako biblioteka treba samostalno brinuti o fotografiranju u posebnoj Android aktivnosti i pri povratku vratiti putanju do slike spremljene na uređaju. Kako bi se implementiralo fotografiranje potrebno je dodati ovisnosti na biblioteku CameraX čija je verzija odvojena u glavnu *build.gradle* skriptu (programski kod 4.17) [14].

```
//CAMERA
implementation "androidx.camera:camera-core:${camerax_version}"
implementation "androidx.camera:camera-camera2:${camerax_version}"
```

Programski kod 4.17. *Postavljanje ovisnosti na CameraX biblioteku*

Prema dokumentaciji biblioteke prvo je potrebno dodati komponentu *TextureView*. Prema programskom kodu 4.18. unutar *activity_camera.xml* datoteke koja predstavlja zaslon aktivnosti dodana je navedena komponenta. Atributom *android:layout_width* je vrijednost širine komponente postavljena na vrijednost širine zaslona, a visina, postavljajući atribut

android:layout_height na „*0dp*“ i atribut *app:layout_constrintDimensionRation* na vrijednost „*1:1*“, je postavljena na jednaku vrijednost kao i širina tako dobivajući kvadratni oblik.

```
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".CameraActivity">

    <TextureView
        android:id="@+id/texture_view_finder"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintDimensionRatio="1:1"/>
...
```

Programski kod 4.18. *Izgled TextureView komponent unutar activity_camera.xml datoteke*

Prema programskom kodu 4.19. metodom *startCamera* unutar aktivnosti se postavljaju početni parametri kamere. Nakon kreiranja objekta klase *PreviewConfig* kojem je postavljen omjer širine i visine te veličina koja odgovara veličini *TextureView* komponente, se kreira instanca klase *Preview* koja omogućava prikaz slike koju kamera trenutno snima. Iz primjera koda se također vidi kreiranje instance *ImageCaptureConfig* klase kojom se postavljaju parametri za fotografiranje. Uz pomoć navedene instance se kreira objekt *ImageCapture* klase s metodom za slikanje koja prima datoteku u koju se sprema slika i povratni poziv kada se obavi fotografiranje. Unutar povratnog poziva se vraća iz aktivnosti i kao rezultat se postavlja lokacija slike u uređaju. Pozivom statičke metode *bindToLifecycle* iz klase *CameraX se* registrira prikaz trenutne slike i akcija za fotografiranje. Metoda *startCamera* se poziva u aktivnosti unutar metode *onCreate*.

```
private void startCamera() {
    PreviewConfig previewConfig = new PreviewConfig.Builder()
            .setTargetAspectRatio(new Rational(1, 1))
            .setTargetResolution(new Size(SCREEN_WIDTH, SCREEN_HEIGHT))
            .build();

    Preview preview = new Preview(previewConfig);

    ImageCaptureConfig imageCaptureConfig = new ImageCaptureConfig.Builder()
            .setTargetAspectRatio(new Rational(1, 1))
            .setCaptureMode(ImageCapture.CaptureMode.MIN_LATENCY).build();

    ImageCapture imageCapture = new ImageCapture(imageCaptureConfig);
    findViewById(R.id.button_capture).setOnClickListener(v -> {
        imageCapture.takePicture(
                file,
                new ImageCapture.OnImageSavedListener() {
                    @Override
                    public void onImageSaved(@NonNull File file) {
                        ...
                    }

                    @Override
                    public void onError(...){
                        ...
                    }
                });
    });
    CameraX.bindToLifecycle(this, preview, imageCapture);
}
```

Programski kod 4.19. *Postavljanje fotografiranja i pregleda kamere na komponenti*

5. ZAKLJUČAK

Korištenjem biblioteke se olakšava izrada novih Android aplikacija i ubrzava proces implementacije ostalih značajki. Funkcionalnosti zajedničke svim projektima su odvojene i olakšana je naknadna promjena i proširivanje biblioteke bez nužnog mijenjanja klijentskog programskog koda. Cilj ovog završnog rada bio je izraditi funkcionalnu i modularnu biblioteku za izradu Android aplikacija te tako smanjiti vrijeme kreiranja i automatizirano ubrzati proces realizacije.

Izrada završnog rada zahtijevala je detaljno planiranje modularne arhitekture i praćenje pravila za čisti i održivi programski kod. Nadalje, omogućena komunikacija s bibliotekom je sakrivena preko sučelja pružajući tako dodatan sloj sigurnosti. Biblioteka je učitana na vanjski repozitorij i kao takva je dostupna za korištenje svima putem mreže. Vanjski repozitorij olakšava praćenje i pretragu repozitorija po verzijama. Modularnost omogućava programerima odvojeno sudjelovanje na daljnjem poboljšanju biblioteke ili razvoju aplikacije koja ju koristi kako bi se omogućio čisti i organizirani programski kod.

LITERATURA

[1] Technopedia, Software Library, https://www.techopedia.com/definition/3828/software-library, lipanj 2019.

[2] I. Krajci, D. Cummings, Android on x86, Apress, Berkley, CA, 2013.

[3] Jetbrains, IntelliJ IDEA, https://www.jetbrains.com/idea/, lipanj 2019.

[4] Developers, Configure your build, https://developer.android.com/studio/build/index.html, lipanj 2019.

[5] Developers, Activity, https://developer.android.com/reference/android/app/Activity, lipanj 2019.

[6] Javapoint, Java tutorial, https://www.javatpoint.com/internal-details-of-jvm, kolovoz 2019.

[7] JitPack.io, https://jitpack.io/docs/, lipanj 2019.

[8] Tutorialspoint, Android – SQLite Database, https://www.tutorialspoint.com/android/android_sqlite_database.htm, lipanj 2019.

[9] RESTfulAPI, What is REST?, https://restfulapi.net, kolovoz 2019.

[10] W3schools, What is HTTP?, https://www.w3schools.com/whatis/whatis_http.asp, kolovoz 2019.

[11] Square, Retrofit, https://square.github.io/retrofit/, kolovoz 2019.

[12] Auth0, Understanding Refresh Tokens, https://auth0.com/learn/refresh-tokens/, lipanj 2019.

[13] Developers, Sensors Overview, https://developer.android.com/guide/topics/sensors/sensors_overview, srpanj 2019.

[14] Developers, CameraX Overview, https://developer.android.com/training/camerax, srpanj 2019.

SAŽETAK

Cilj ovog rada je opisati korištenje biblioteka na Android platformi, prikazati postupak učitavanja biblioteke na vanjski repozitorij i implementaciju modularne arhitekture biblioteke koja omogučava automatsko postavljanje osnovnih parametara projekta, spajanje korisnika, postavljanje lokalne baze, pristup kameri i senzorima. U prvom dijelu su objašnjene sve tehnologije i pomoćne biblioteke potrebne za razvoj. Zatim su definirani korisnički zahtjevi i specifikacije kao osnovna pravila funkcioniranja i pravila po kojima je smišljena arhitektura programskog koda. Naposljetku je objašnjen postupak implementacije programskog koda i postavljanja početnih parametara projekta.

Ključne riječi: Android, biblioteka, Java, moduli, REST

ABSTRACT

The goal of this assignment was to describe the use of libraries on the Android platform, to present the process of loading a library to an external repository and show the implementation of a modular library architecture that allows automatic setting of basic project parameters, connecting users, setting up a local database, accessing cameras and sensors. The first section explains all the technologies and support libraries needed for development. Then, user requirements and specifications are defined as the basic rules of operation and rules by which the code architecture is designed. Finally, the process of implementing the code and setting the initial project parameters is explained.

Keywords: Android, Java, library, modules, REST

ŽIVOTOPIS

Luka Perišić rođen je 17.10.1997. u Slavonskom Brodu. Pohađao je osnovnu školu „Matija Antun Reljković“, u Cerni. Nakon završetka osnovne škole upisuje „Gimnaziju Matije Antuna Reljkovića“, u Vinkovcima, prirodoslovno-matematički smjer. 2016. godine polaže maturu, te iste godine upisuje preddiplomski sveučilišni studij na Fakultetu za elektrotehniku, računarstvo i informacijske tehnologije Osijek, smjer Računarstvo. 2018. godine postaje stipendist STEM stipendije na temelju uspješnosti. Trenutno radi godinu dana kao student Android developer u tvrtki Gauss Development. Također posjeduje određeno znanje u govoru čitanju i pisanju engleskog jezika. Posjeduje određeno znanje o programskim jezicima Java, Kotlin, JavaScript i C++ te znanje o razvijanju Android aplikacija i pozadinskih servisa.

Luka Perišić