Upravljanje i nadzor rada robotske ruke korištenjem IoT platforme i Android aplikacije

Blavicki, Anamarija

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

Upravljanje i nadzor rada robotske ruke korištenjem IoT platforme i Android aplikacije

Diplomski rad

Anamarija Blavicki

Osijek, 2019.

Sadržaj

1. UVOD

Tvrtka Ericsson Nikola Tesla je razvila prototip sustava koji omogućava upravljanje i praćenje rada robotske ruke na daljinu. Rad ruke se prati pomoću niza senzora integriranih na ruci. Podatci sa senzora se prikupljaju i šalju na IoT platformu. Svrha ovog rada je razviti Android mobilnu aplikaciju koja će omogućiti dodatan način praćenja rada ruke, a ujedno i učiniti podatke prikupljene sa senzora dostupnijima. Aplikacija će također olakšati upravljanje ruke budući da će omogućiti pokretanje predefiniranih radnji te pomicanje svakog pojedinačnog zgloba.

Rad je koncipiran u pet poglavlja kroz koja je opisan proces razvoja Android aplikacije za nadzor i upravljanje robotske ruke. Nakon kratkog uvoda u rad te opisa zadatka diplomskog rada u prvom poglavlju, u drugom poglavlju su opisane tehnologije korištene za razvoj rada – Android OS, ASP.NET SignalR tehnologija, IoT (Internet of Things) Accelerator te AMQ protokol. U trećem poglavlju se opisuje realizacija aplikacije. Prvenstveno je predstavljena ideja i osnovni koncept zadatka te je prikazan cjelokupni sustav koji sadrži robotsku ruku s Arduino mikroupravljačem, Android aplikaciju te IoT platformu. U tom poglavlju je detaljnije opisan način povezivanja aplikacije s platformom u svrhu nadzora senzora integriranih na robotskoj ruci. U sljedećem, četvrtom, poglavlju, opisan je način povezivanja aplikacije i robotske ruke u svrhu upravljanja ruke. Na kraju je prikazano testiranje aplikacije te njene realizirane funkcionalnosti.

1.1. Zadatak diplomskog rada

Glavni zadatak je izraditi Android aplikaciju za upravljanje i nadzor rada robotske ruke. Aplikacija treba imati mogućnost grafičkog prikazivanja podataka sa senzora preko IoT platforme u stvarnom vremenu, a neki od senzorskih podataka su višeosno inercijalno ubrzanje (akcelerometar), višeosno Zemljino gravitacijsko ubrzanje (žiroskop), temperatura zgloba i temperatura okoline. Aplikacija također treba imati mogućnost grafičkog prikaza povijesnih podataka sa senzora korištenjem IoT plaftorme (npr. posljednji sat, dan, tjedan).

Unutar aplikacije se moraju moći konfigurirati parametri za spajanje na IoT platformu (identifikator mrežnog posrednika, sigurnosni ključevi) te se moraju moći dodavati i brisati senzori za prikaz podataka. Što se tiče upravljanja ruke, aplikacija treba imati mogućnost ručne kontrole ruke poput mijenjanja položaja svakog pojedinačnog pokretnog zgloba – ruka ima šest zglobova,

odnosno šest stupnjeva slobode kretanja. Također trebaju postojati predefinirane rutine ruke poput zaustavljanja ili pokretanja te ubrzavanja ili usporavanja rada ruke.

2. ANALIZA KORIŠTENIH TEHNOLOGIJA

Unutar poglavlja su detaljnije opisane tehnologije korištene pri izradi aplikacije kako bi se omogućilo lakše razumijevanje ostatka rada. Prvenstveno je opisan Android operacijski sustav te njegova službena razvojna okolina – Android Studio unutar koje je aplikacija realizirana. Jezik koji je korišten za razvoj je Java. Za uspostavu komunikacije između servera i klijenta, korištena je SignalR biblioteka koja se temelji na WebSocket protokolu za razmjenu podataka u stvarnom vremenu. Korištena platforma je Ericsson IoT Accelerator [1].

2.1. Android operacijski sustav

Android operacijski sustav je otvoreni OS koji omogućava korisnicima prepravljanje, mijenjanje i poboljšavanje izvornog koda i njegovog sadržaja. Aplikacije su napisane pomoću *Software Development Kit*a (SDK) te svaka verzija Androida ima svoj SDK koji sadrži alate za programiranje kao što su *debugger*, softverske biblioteke, emulator, dokumentaciju i razne primjere koji služe kao pomoć pri razvoju. Razvojni jezik aplikacija može biti Java ili Kotlin te se one pokreću pomoću Dalvik virtualnog stroja budući da na uređajima ne postoji Java virtualni stroj koji služi za izvršavanje Java bajtnog koda. [2]

2.1.1. Android Studio

Android Studio je službena integrirana razvojna okolina (engl. *Integrated Development Enviroment* – IDE) za aplikacije namijenjene za Android operacijski sustav. Zasniva se na ItelliJ IDEA softveru pomoću kojega se olakšava razvoj aplikacija te pruža podrška pri razvoju. Dostupan je brzi emulator s brojnim funkcionalnostima za testiranje aplikacija bez stvarnog uređaja, *Instant Run* – pokretanje aplikacija s novim promjenama bez *build*anja novog APK, integracija sustava za kontrolu verzije poput GitLaba, Lint alati pomoću kojih se prati kvaliteta koda itd. Android Studio ima *Gradle build* sustav temeljen na JVM (engl. *Java Virtual Machine*) *build* sustavima te tako izvorne programske datoteke komprimira u jednu *.apk* datoteku koja se koristi pri implementaciji aplikacija na Android uređaje. [3]

Od 2018. godine, dostupna je nova verzija Android *Support* biblioteka – *AndroidX* kojom se dodaju nove funkcionalnosti i unaprjeđuju stare kao npr. mijenjanje imena svih paketa (biblioteka)

u konzistentna imena (s početkom *androidx.**). S obzirom na promjene, potrebno je napraviti migraciju postojećih projekata na AndroidX, što se omogućava u Android Studiu, no projekt mora imati API verziju 28. 54[4]

Projekti su prikazani u Android načinu prikaza koji se sastoji od tri glavne cjeline – *manifest, java* i *res* direktorije. *Manifest* direktorij sadrži jednu od temeljnih datoteka Android aplikacije – *AndroidManifest.xml* unutar koje se nalaze svi meta-podatci aplikacije, struktura aplikacije, aktivnosti, servisi, dozvole aplikacije te također ikonu i verziju aplikacije.

Java direktorij sadrži sve aktivnosti (engl. *activitiy*) koje predstavljaju pojedine zaslone aplikacije, kao i ostale potrebne java datoteke koje pomažu pri funkcionalnom radu aplikacije te su sve datoteke unutar ovog direktorija napisane u java programskom jeziku.

Res direktorij sadrži sve resurse potrebne za rad aplikacije – izgled aplikacije (engl. *layout*), slike koje se koriste u aplikaciji, boje, dimenzije aplikacije i slično. Izgled aplikacije se definira pomoću XML opisnog jezika te se on također koristi i za većinu datoteka u *res* direktoriju. [1]

2.2.ASP.NET SignalR

SignalR je biblioteka za ASP.NET developere pomoću koje se olakšava korištenje web funkcionalnosti aplikacija u stvarnom vremenu. Funkcionalnosti u stvarnom vremenu se odnose na mogućnost servera da šalje podatke klijentima onog trenutka kad ti podatci budu dostupni, dakle klijent ne mora zatražiti podatke od poslužitelja. SignalR se najčešće koristi u aplikacijama koje omogućuju razmjenu poruka između klijenata (*chat*), no u ovom slučaju se koristi kako bi se korisniku prikazali podatci sa senzora u trenutku kada su poslani na poslužitelj.

Biblioteka koristi *WebSocket* protokol koji predstavlja način prijenosa podataka kojim se ostvaruje kontinuirana, dvosmjerna komunikacija između poslužitelja i klijenta. Za korištenje spomenutog protokola, potrebno je da obje strane (i poslužitelj i klijent) podržavaju WebSocket način rada.

SignalR API (engl. *Application Programming Interface*) podržava dva modela komunikacije između poslužitelja i klijenta – *persistent connections* i hubove. Konekcija predstavlja jednostavnu završnu točku za slanje poruka jednom primatelju, grupi ili *broadcast* poruka. Takva konekcija omogućuje developeru pristup komunikacijskom protokolu niske razine koji SignalR koristi.

Hubovi su češće korišteni te se temelje na API-ju od konekcija i omogućavaju klijentu i poslužitelju da izravno pozivaju metode jedan na drugom. Ovaj model također omogućava

klijentima da proslijede parametre koji su zahtijevani za pojedine metode od strane poslužitelja. Hubovi rade na način da kada poslužiteljska strana pozove metodu na klijentskoj strani, šalje se paket podataka unutar kojega se nalazi ime i parametri pozvane metode. Ta metoda mora biti definirana na klijentskoj strani kako bi se primili poslani podatci.

U SignalR tehnologiji postoje tri vrste veze (konekcije) – SignalR konekcija, transportna konekcija te fizička konekcija. SignalR konekcija predstavlja logičku vezu između URL-a poslužitelja i klijenta koju karakterizira jedinstveni konekcijski identifikacijski broj (engl. *connection ID*). Pomoću podataka koji se prenose tom vezom se omogućava spajanje transportne konekcije koja se odnosi na logičku vezu između poslužitelja i klijenta održavanu od strane jednog transportnog API-ja (*WebSocket*i, događaji poslani od strane poslužitelja, *forever frame* ili *long-polling*). SignalR koristi transportni API kako bi uspostavio transportnu vezu, a API ovisi o postojanoj fizičkoj vezi. Fizička veza se odnosi na fizičke spojne elemente mreže, tj. žice, bežične signale, rutere koji omogućavaju komunikaciju između klijenta i poslužitelja. Dakle, za postojanje SignalR konekcije bitno je postojanje fizičke veze između klijenta i poslužitelja na koju se oslanja transportna veza koja je nužna za SignalR vezu. [5]

U sljedećem potpoglavlju je opisan način rada WebSocket protokola koji je, u ovom slučaju, podržan i od strane klijenta (Android aplikacije) i od strane poslužitelja (Ericsson IoT Accelerator).

2.2.1. WebSocket protokol

WebSocket tehnologija je komunikacijski protokol koji omogućava dvosmjernu komunikaciju između poslužitelja i klijenta preko jedne TCP veze. Koristi se za ostvarivanje komunikacije u stvarnom vremenu zbog mogućnosti slanja podataka bez prethodnog zahtjeva za istima.

WebSocket tehnologija se javlja kao rješenje problema HTTP (engl. *HyperText Transfer Protocol*) komunikacije – HTTP zahtjevi dodaju veliku količinu zaglavlja (engl. *header*) pri slanju podataka što rezultira u velikoj količini dodatnog prometa te se pojavljuje veliko kašnjenje između servera i klijenta, kao i opterećenje obje strane.

Protokol se sastoji od dva dijela – handshake kojim se započinje komunikacija te prijenos podataka. Klijent uspostavlja WS konekciju putem procesa WS handshake – počinje kao uobičajeni HTTP GET zahtjev serveru (omogućava poslužiteljima da primaju i HTTP i WebSocket zahtjeve) te se pomoću HTTP *Upgrade* zaglavlja označava zahtjev za WebSocket komunikacijom. Server odgovara klijentu te, ako podržava WebSocket protokol, kroz Upgrade

zaglavlje obavještava klijenta da je komunikacija uspostavljena. Time se HTTP veza zamjenjuje WebSocket vezom koja koristi istu TCP/IP konekciju te razmjena podataka može započeti s bilo koje strane. Za zatvaranje veze, jedna strana mora poslati odgovarajući okvir za zatvaranje, a razlog zatvaranja nije obavezan iako se može navesti.

Dakle, WebSocket protokol podržava stalnu konekciju između klijenta i servera te na taj način omogućava slanje podataka u stvarnom vremenu. Koristi se *socket*ima koji predstavljaju stalne komunikacijske kanale između dva kraja – klijenta i poslužitelja, poslužitelj automatski šalje klijentu sve pristigle podatke. [6]

2.3.IoT Accelerator

IoT Accelerator predstavlja IoT platformu u vlasništvu tvrtke Ericsson te je značajan dio platforme i DDM – Device and Data Managements koji omogućava upravljanje uređajima i podatcima te brzi razvoj IoT aplikacija. Komunikacija između platforme i aplikacija se vrši pomoću IP protokola te UDP/TCP transportnog protokola.

DDM se sastoji od tri cjeline – upravljanje uređajima, upravljanje podatcima te procesiranje u stvarnom vremenu. Upravljanje podatcima omogućuje spremanje i praćenje vrijednosti pojedinih senzora te je također dostupan REST API koji korisnicima (aplikacijama) omogućava pristup podatcima. Za pristup je potrebna identifikacija korisnika koja se vrši pomoću API-ključa. Kompletan popis API zahtjeva se može naći na [7].

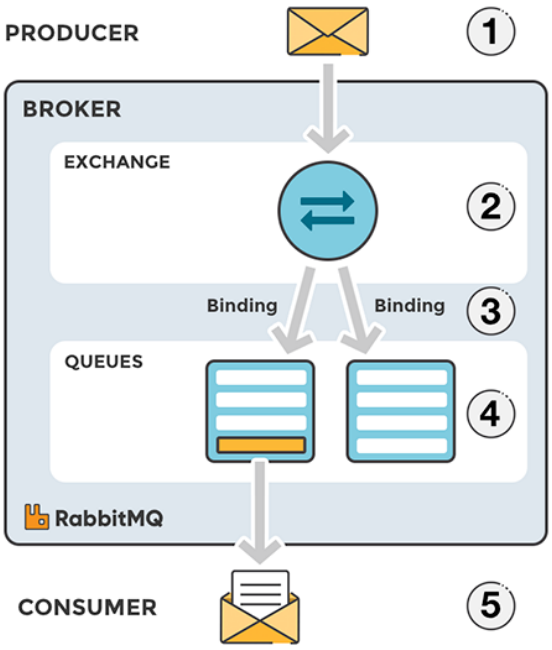
Komunikacija između platforme i korisnika se kriptira unutar komunikacijskih kanala te platforma podržava protokole otvorenog koda – HTTPS, AMPQ i MQTT protokoli.

Platforma je razvijena na način da može podržavati širok spektar aplikacija, različitih korisničkih slučajeva poput povezivanje zgrada ili „pametnih“ kuća, „pametni“ gradovi, „pametno“ parkiranje te mnogi drugi. [8]

2.4.Advanced Message Queuing Protocol – AMQP

AMQP je protokol otvorenog standarda koji radi na aplikacijom sloju te omogućava pouzdano slanje i primanje poruka. Protokol se sastoji od tri entiteta – pošiljatelj (engl. *producer*), primatelj (engl. *consumer*) i broker. Pošiljatelj generira i šalje poruke te poruke unutar brokera prvo idu na

exchange koji ima zadatak primanja i umjeravanja poruka na odgovarajući red. Odgovarajući red se odabire pomoću definiranih parametara unutar poruke. Zatim se za prijenos poruke stvara *binding* tj. poveznica između *exchange*a i reda čekanja (engl. *queue*). Svaki red čekanja ima svoju vezu, ali može biti povezan s više exchangea i s više primatelja. Poruke se zatim spremaju u redove sve dok ih primatelj ne preuzme. Na slici Slika 2.1. je prikazan proces slanja poruke.



Slika 2.1. AMQP proces slanja poruke[9]

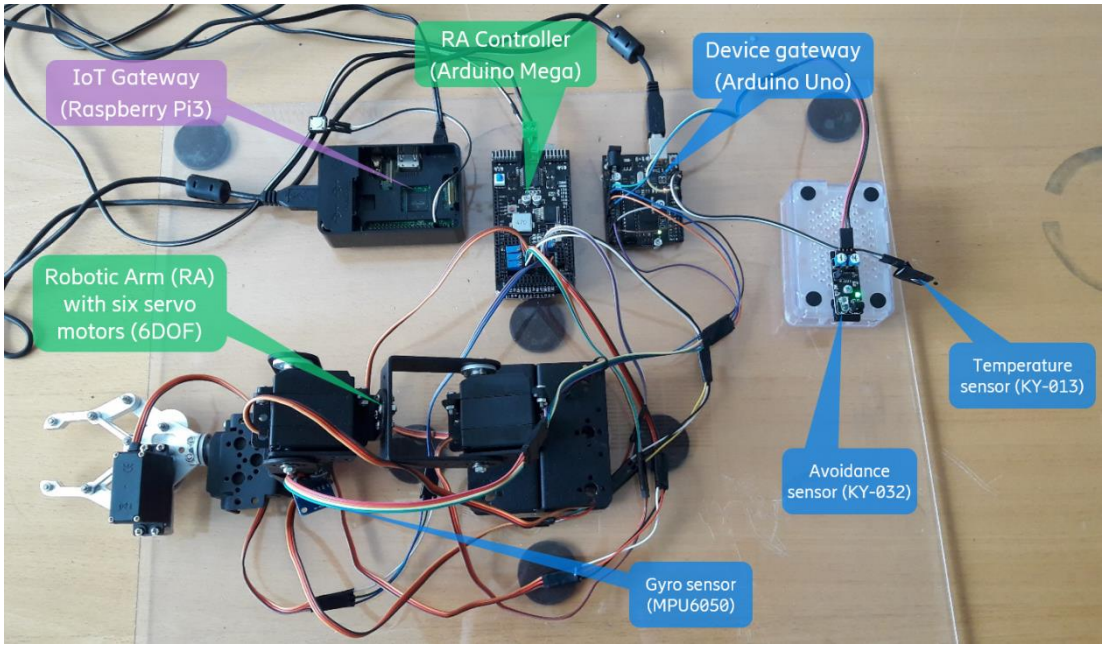
Kako bi se osigurao primitak poruka, postoje poruke potvrde koje primatelj šalje kada primi poruku. Broker briše poruku iz reda čekanja u kojemu je bila tek kada primi potvrdu od primatelja. [10].

3. REALIZACIJA MOBILNE APLIKACIJE – NADZOR

Unutar ovog poglavlja je detaljnije objašnjen tijek razvoja dijela mobilne aplikacije zaduženog za nadziranje podataka prikupljenih sa senzora na robotskoj ruci. Prvenstveno je navedena ideja i osnovni koncepti za izradu aplikacije, dijagram komponenti cjelokupnog sustava te je prikazan wireframe – početni izgled aplikacije. Također je detaljnije opisana *Room* biblioteka koja predstavlja način spremanja podataka koji su potrebni za dohvaćanje vrijednosti sa IoT platforme. Ta komunikacija s platformom se vrši pomoću *Retrofit* HTTP klijenta te je način rada *Retrofit*a detaljnije objašnjen u nastavku. Na kraju je naveden način spajanja sa IoT platformom u stvarnom vremenu koristeći SignalR.

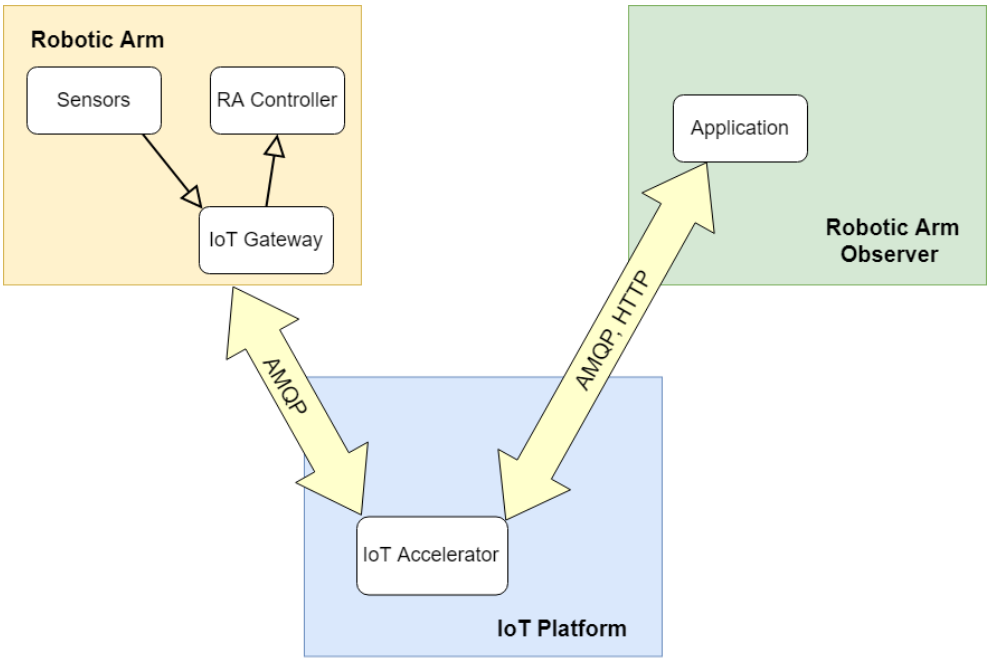
3.1. Ideja i osnovni koncept

Ideja za izradu aplikacije dolazi od tvrtke Ericsson Nikola Tesla te njihove potrebe za Android aplikacijom koja će olakšati nadzor rada robotske ruke te njeno upravljanje. Na ruci se nalazi niz integriranih senzora te se podatci sa senzora prikupljaju i šalju na IoT platformu. Na slici Slika 3.1. je prikazan hardverski dio prototipa.



Slika 3.1. Prototip

Dakle, potrebno je razviti Android aplikaciju za već razvijenu robotsku ruku sa senzorima čije se vrijednosti pomoću Arduina prikupljaju i šalju na Ericssonovu IOTA (Internet of Things Accelerator) platformu. Aplikacija treba dohvaćati podatke s platforme te imati mogućnost prikaza podataka iz prošlosti kao i u stvarnom vremenu pomoću SignalR tehnologije. Također, aplikacija mora imati mogućnost upravljanja robotske ruke tj. mogućnost mijenjanja položaja svakog zgloba na ruci. Ta komunikacija će se ostvariti pomoću AMQP-a (engl. *Advanced Message Queuing Protocol*). Na slici Slika 3.2. je prikazan dijagram komponenti sustava te njihove povezanosti.

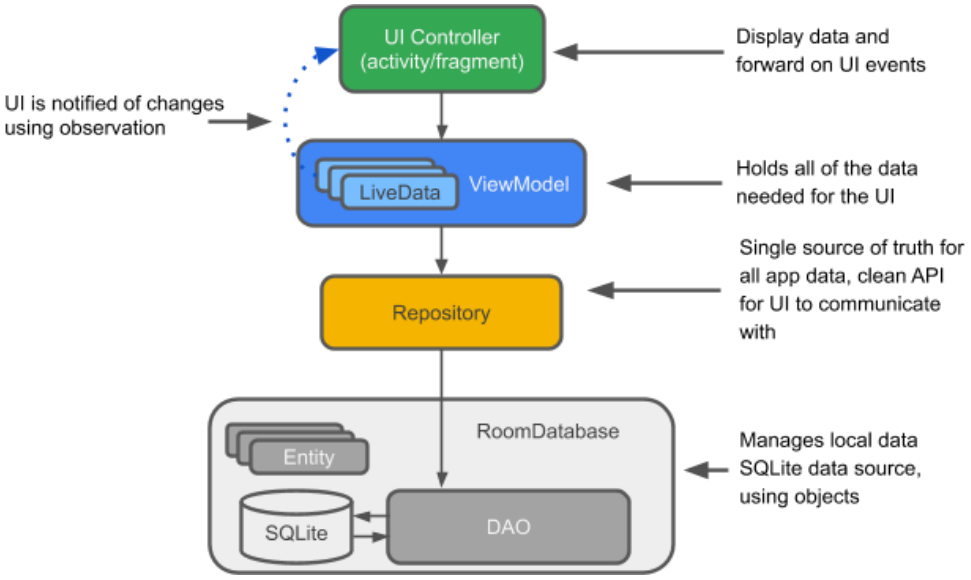


Slika 3.2. Dijagram komponenti

Aplikacija je podijeljena u dva glavna dijela – dio za nadzor rada robotske ruke te dio za upravljanje robotske ruke. Dio za nadzor rada se odnosi na praćenje vrijednosti prikupljenih sa senzora integriranih na robotskoj ruci npr. senzor za temperaturu, akcelerometar, žiroskop. Aplikacija također ima mogućnost dodavanja novih senzora na popis za praćenje kao i brisanje senzora s popisa, a podatci potrebni za dodavanje novih senzora su definirani na IOTA-i. Drugi dio, za upravljanje ruke, sadrži dva načina upravljanja – pokretanje ruke s predefiniranim pokretima – automatskim pokretima te pokretanje svakog zgloba (servomotora) ruke posebno – manualno pokretanje.

3.2.Android Architecture Components

Komponente Android arhitekture (engl. *Android Architecture Components*) predstavljaju skup Android biblioteka koje omogućavaju lakše razvijanje, testiranje i održavanje aplikacija. Također se brinu za postojanost podataka kao i za upravljanje životnog ciklusa podataka. Preporučene komponente arhitekture su *LiveData, ViewMoodel* i *Room* biblioteke te je na slici Slika 3.3. prikazana njihova međusobna povezanost.



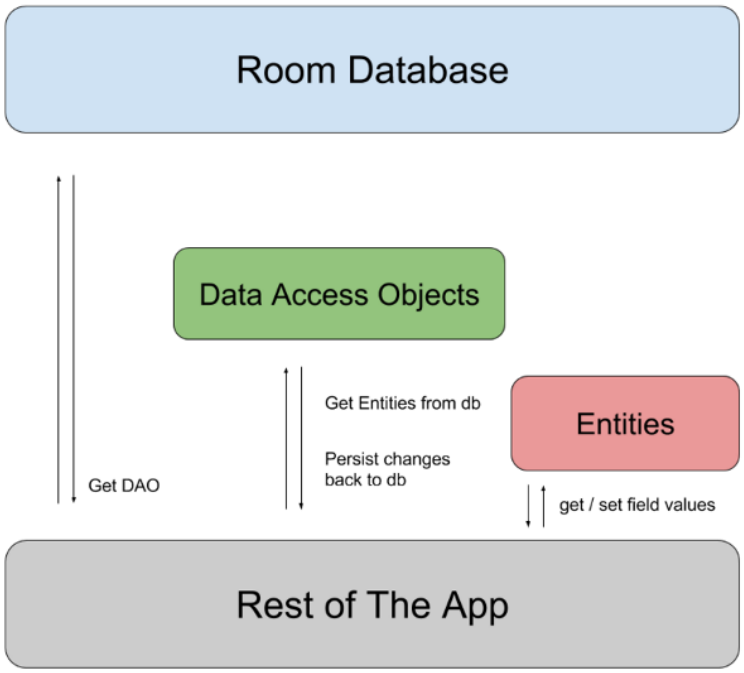
Slika 3.3 Komponente arhitekture[11]

Dakle, Room baza podataka služi za lokalno spremanje podataka koristeći određene objekte te predstavlja sloj na SQLite bazi podataka. Ona sadrži objekt za pristup podatcima (engl. *Data Access Objects* – DAO) pomoću kojeg komunicira s repozitorijem. Pomoću klase repozitorija (engl. *Repository*) se obavljaju sve operacije s podatcima, ona zapravo predstavlja poveznicu između DAO-a i lokalne baze podataka (unutar koje su spremljeni podatci) te olakšava korištenje različitih niti za upite. Iznad repozitorija se nalazi *ViewModel* klasa koja pruža podatke korisničkom sučelju (engl. *User Interface* – UI). Dakle *ViewModel* služi kao sredstvo komunikacije između repozitorija i UI-a, a sadrži i *LiveData* klasu koja pomaže pri praćenju promjena podataka. [11]

3.2.1. Room

Room je biblioteka koja predstavlja apstraktni sloj iznad SQLite baze podataka te se pomoću nje olakšava pristup i rad sa spremljenim podatcima. Sastoji se od tri glavne komponente – baze

podataka koja predstavlja pristupnu točku za interakciju i rad s relacijskim podatcima aplikacije, entiteta koji predstavlja tablicu u bazi i objekata za pristup podatcima (DAO) koji sadrže metode za pristup bazi. Na slici Slika 3.4. je prikazana arhitektura Room biblioteke na kojoj se vide njene komponente i njihova povezanost.



Slika 3.4. Komponente Room biblioteke [12]

Za korištenje Room biblioteke unutar Android aplikacije, potrebno je u *build.gradle* datoteku dodati ovisnost (engl. *dependency*):

```
//room database
implementation 'androidx.room:room-runtime:2.1.0'
annotationProcessor 'androidx.room:room-compiler:2.1.0'
```

Slika 3.5. Ovisnosti Room biblioteke

Za spremanje podataka, potrebno je napraviti klasu te ju označiti sa *@Entity* anotacijom. Time se, u bazi podataka, stvara tablica s nazivom klase te se objekti te klase spremaju unutar istoimene tablice. Atributi tablice su atributi klase te ih moguće označiti sa *@ColumnInfo* anotacijom kojom se omogućuje postavljanje naziva stupca unutar tablice. [13]

Na slici Slika 3.6. je prikazana klasa *Sensor* koja predstavlja jednu od tablica korištenu za potrebe rada. Sa slike se može uočiti da u bazi podataka postoji tablica s nazivom „sensors“, a njeni entiteti su „name“ (ime senzora), „description“ (opis senzora), „id“ (identifikacijski broj senzora) i „url“ (adresa senzora).

```
@Entity(tableName = "sensors")
public class Sensor {

    @ColumnInfo(name = "name")
    private String name;
    @ColumnInfo(name = "description")
    private String description;
    @PrimaryKey
    @NonNull
    @ColumnInfo(name = "id")
    private String ID;
    @ColumnInfo(name = "url")
    private String resourceUrl;
}
```

Slika 3.6. Klasa Sensor

Pojedine atribute tablice je moguće i dodatno definirati pomoću anotacije *@PrimaryKey* koja označava primarni ključ tablice, *@NonNull* koja označava da atribut ne smije biti *null* ili prazan. Također je moguće ignorirati (ne dodavati u tablicu) pojedine atribute s anotacijom *@Ignore*. [13]

Nakon što je kreirana tablica, potrebno je napraviti objekt za pristup podatcima (DAO). DAO se definira kao Java sučelje (engl. *interface*) te se označava s anotacijom *@Dao*. Unutar tog sučelja se nalazi popis metoda pomoću kojih se pristupa podatcima unutar baze. Na slici Slika 3.7. se mogu vidjeti definirane metode za dodavanje, brisanje i mijenjanje pojedinačnog senzora kao i metode za dohvaćanje i brisanje svih senzora – sve metode su osnovne CRUD (engl. *Create, Read, Update, Delete*) operacije te se koriste *@Insert*, *@Delete*, *@Update* i *@Query* anotacije. Također je moguće pisati vlastite SQL upite pomoću *@Query* anotacije. [14]

```
@Dao
public interface SensorDao {

    @Insert
    void insert(Sensor sensor);

    @Delete
    void delete(Sensor sensor);

    @Update
    void update(Sensor sensor);

    @Query("DELETE FROM sensors")
    void deleteAll();

    @Query("SELECT * FROM sensors ORDER BY name ASC")
    LiveData<List<Sensor>> getAllSensors();
}
```

Slika 3.7. Objekt za pristup podatcima (DAO)

Posljednja komponenta koju je potrebno napraviti je klasa koja predstavlja bazu podataka. Ta klasa mora biti apstraktna klasa (može se naslijediti, ali se ne može instancirati) koja nasljeđuje

RoomDatabase klasu, označena *@Database* anotacijom te se mora naznačiti koje klase predstavljaju njene entitete tj. tablice što je u ovom slučaju *Sensor.class*. Na slici Slika 3.8. se vidi klasa SensorRoomDatabase koja je definirana kao *singleton* kako bi se spriječilo višestruko otvaranje baze podataka. [15]

```
@Database(entities = {Sensor.class}, version = 1)
public abstract class SensorRoomDatabase extends RoomDatabase {

    public abstract SensorDao sensorDao();

    private static volatile SensorRoomDatabase INSTANCE;

    static SensorRoomDatabase getDatabase(final Context context){
        if(INSTANCE == null){
            synchronized (SensorRoomDatabase.class){
                if(INSTANCE == null){
                    INSTANCE =
Room.databaseBuilder(context.getApplicationContext(),
                            SensorRoomDatabase.class, "sensor_database")
                            .build();
                }
            }
        }
        return INSTANCE;
    }
}
```

Slika 3.8. SensorRoomDatabase klasa

Room baza podataka se u aplikaciji koristi kako bi korisnik mogao na jednostavan način dodavati i uklanjati senzore čije vrijednosti može nadzirati.

3.2.2. Lifecycle-Aware Components

Komponente svjesne životnog ciklusa omogućavaju praćenje promjena te izvode radnje na temelju promjene životnog ciklusa neke druge komponente (npr. aktivnosti ili fragmenta). Za korištenje ovih komponenti potrebno je u *gradle.build* datoteku dodati ovisnosti sa slike Slika 3.9.

```
//lifecycle components
implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0'
annotationProcessor 'androidx.lifecycle:lifecycle-compiler:2.0.0'
```

Slika 3.9. Ovisnosti lifecycle komponenti

Za potrebe aplikacije korištene su dvije klase iz biblioteke životnog ciklusa (engl. *Lifycycle library*) – *LiveData* i *ViewModel* klase. *LiveData* klasa predstavlja rješenje za kontinuirano praćenje podataka. Na ovaj način se omogućava ažuriranje UI-a čim se dogodi neka promjena na podatcima (poput dodavanja novog senzora ili mijenjanja atributa postojećeg senzora).

Za potrebe aplikacije, *LiveData* klasom se omota popis svih senzora koji se nalaze u bazi, dakle koristi se pri dohvaćanju svih senzora kako bi korisnici uvijek bili dostupni najnoviji podatci. Takav upit je prikazan na slici ispod (Slika 3.10.) te se on, pomoću *LiveData* klase, izvršava asinkrono na pozadinskoj niti (engl. *background thread*) budući da Room biblioteka ne dopušta izvršavanje upita na glavnoj niti (engl. *main thread*). [16]

```
@Query("SELECT * FROM sensors ORDER BY name ASC")
LiveData<List<Sensor>> getAllSensors();
```

Slika 3.10. Primjer upita s LiveData tipom varijable

ViewModel klasa se koristi kao način komunikacije između repozitorija i korisničkog sučelja, također se koristi za razmjenu podataka između fragmenata. Dakle, ta klasa služi kao spremnik za podatke potrebne za korisničko sučelje te se brine o njihovom životnom ciklusu i u slučaju mijenjanja konfiguracije aplikacije kao npr. rotacije zaslona. *ViewModel* klasa također koristi *LiveData* popis senzora kako bi se taj popis spremio u *cache* memoriju te se na taj način izbjegava bespotrebno dohvaćanje podataka iz baze u slučaju da nema promjena. [17] Na slici Slika 3.11. je prikazana implementacija *ViewModel*a.

```
public class SensorViewModel extends AndroidViewModel {

    private SensorRepository mRepository;
    private LiveData<List<Sensor>> mAllSensors;

    public SensorViewModel(@NonNull Application application) {
        super(application);
        mRepository = new SensorRepository(application);
        mAllSensors = mRepository.getAllSensors();
    }

    public LiveData<List<Sensor>> getAllSensors() {
        return mAllSensors;
    }

    public void insert(Sensor sensor) {mRepository.insert(sensor);}
    public void delete(Sensor sensor) {mRepository.delete(sensor);}
    public void update(Sensor sensor) {mRepository.update(sensor);}

}
```

Slika 3.11. Implementacija ViewModel klase

Povezivanje korisničkog sučelja i baze podataka se izvršava pri pokretanju aplikacije te se u početnoj aktivnosti (*MainActivity.java*), u *OnCreate* metodi, kreira *ViewModel* pomoću *ViewModelProviders* klase (Slika 3.12.).

```
sensorViewModel = ViewModelProviders.of(this).get(SensorViewModel.class);
```

Slika 3.12. Definiranje ViewModel objekta

Isti *ViewModel* se koristi i u *SensorsListFragment.java* datoteci, gdje se implementira i promatrač (Slika 3.13.) na promjene podataka te se on izvršava svaki put kada se promatrani podatci promijene, npr. brisanje senzora s popisa (sSlika 3.14.).

```
sensorViewModel.getAllSensors().observe(this, adapter::setSensors);
```

Slika 3.13. Implementiranje ViewModel promatrača

```
sensorViewModel.delete(adapter.getSensor(position));
```

Slika 3.14. Pozivanje delete metode ViewModela

3.3.Retrofit

Retrofit biblioteka predstavlja REST (engl. *Representational State Transfer*) klijenta za Javu i Android koji olakšava slanje i dohvaćanje strukturiranih (npr. u JSON formatu – engl. *JavaScript Object Notation*) podataka s nekog poslužitelja. Komunikacija između klijenta (aplikacije) i poslužitelja se izvršava korištenjem OkHttp biblioteke. Na slici ispod (Slika 3.15.) su prikazane ovisnosti Retrofit biblioteke.

```
//retrofit
implementation 'com.squareup.retrofit2:retrofit:2.5.0'
implementation 'com.squareup.retrofit2:converter-gson:2.5.0'
```

Slika 3.15. Ovisnosti Retrofit biblioteke

Za korištenje Retrofita prvenstveno je potrebno napraviti sučelje unutar kojega se definiraju željeni upiti prema poslužitelju te se time stvara HTTP API u kao Java sučelje. Svaki upit mora sadržavati HTTP anotaciju (GET, POST, PUT, DELETE ili HEAD) s željenom metodom i relativnim URL-om (engl. *Uniform Resource Locator*). [18] U sučelju SensorAPI.java su kreirana dva upita – za dohvaćanje vrijednosti senzora tj. mjerenja sa senzora te za dohvaćanje *realtime* tokena sa IoT platforme.

Kako bi se dohvatile vrijednosti pojedinog senzora, potrebno je specificirati *X-DeviceNetwork* koji predstavlja ključ mreže tj. identifikator mreže i *Authorization* token koji predstavlja ključ za autorizaciju te su oba parametra definirana je od strane IOTA-e. Ta dva parametra se nalaze u zaglavlju upita, a u putanju upita se postavljaju *resourceId* koji je ID pojedinog senzora te početno vrijeme (*start*) koje je potrebno predati u obliku *unixtimestamp*a u milisekundama. Na slici Slika

3.16 je prikazan upit za dohvaćanje vrijednosti senzora iz prošlosti te su definirani sve prethodno objašnjeni parametri.

```
@GET("api/v3/measurements/{resourceId}/since/{start}")
Call<Measurement> getAllMeasurements(
        @Header("X-DeviceNetwork") String deviceNetwork,
        @Header("Authorization") String token,
        @Path("resourceId") String resourceId,
        @Path("start") long unixTime
);
```

Slika 3.16. HTTP upit za dohvaćanje vrijednosti senzora

Na slici Slika 3.17. je prikazan rezultat prethodno opisanog upita te se moguće vidjeti kako se dobiva JSON objekt koji sadrži ID senzora te polje objekata unutar kojih se nalazi vremenska oznaka te vrijednost senzora. Odgovor na zahtjev je dobiven putem Swagger API frameworka. [7]


```
{
  "id": "7fbf70e3-07f5-4637-999f-c68a77eac12b",
  "v": [
    {
      "UnixTimestamp": 1561300776000,
      "DateTime": "2019-06-23T14:39:36+00:00",
      "v": 19
    },
    {
      "UnixTimestamp": 1561300876000,
      "DateTime": "2019-06-23T14:41:16+00:00",
      "v": 23
    },
    {
      "UnixTimestamp": 1561300884000,
      "DateTime": "2019-06-23T14:41:24+00:00",
      "v": 29
    }
  ]
}
```


Slika 3.17. Odgovor na HTTP zahtjeva

Na temelju rezultata sa slike Slika 3.17., generirane su klase koje predstavljaju JSON modele – *Measurement* i *Value* klase prikazane na slikama ispod (Slika 3.18. i Slika 3.19.). Za pretvaranje JSON stringova u Java objekte, korištena je *Google*ova *Gson* biblioteka koja također omogućava i pretvaranje Java objekata u JSON stringove. Atribute unutar klase potrebno je anotirati *@Expose* anotacijom kojom se odlučuje koje atribute treba uzimati u obzir te anotacijom *@SerializedName* u slučaju da se ime varijable u Javi razlikuje od imena u JSON objektu.

```
public class Measurement {

    @Expose
    private String id;
    @SerializedName("v")
    @Expose
    private List<Value> values = null;

}
```

Slika 3.18. Measurement model

```
public class Value {

    @SerializedName("UnixTimestamp")
    @Expose
    private Long unixTimestamp;
    @SerializedName("DateTime")
    @Expose
    private String dateTime;
    @SerializedName("v")
    @Expose
    private Float value;

}
```

Slika 3.19. Value model

Nakon kreiranih potrebnih klasa, pomoću metode *setupRetrofit()*, prikazana na slici Slika 3.20., je implementiran Retrofit unutar aplikacije. Kreiranjem instance Retrofita se povezuje kreirano API sučelje s HTTP upitima te se navodi URL IoT platforme (baseUrl) koji se dohvaća iz dijeljenih postavki aplikacije (dohvaća se iz *Shared Preferences* i sprema u statičku varijablu *IoTLocation*), klijent koji se koristi za izvršavanje upita (OkHttp) te način serijalizacije i deserijalizacije objekata (Gson).

```
private void setupRetrofit() {
    Retrofit retrofit = new Retrofit.Builder()
            .baseUrl(IoTLocation)
            .client(new OkHttpClient())
            .addConverterFactory(GsonConverterFactory.create(new GsonBuilder().se-
rializeNulls().create()))
            .build();

    api = retrofit.create(SensorAPI.class);
}
```

Slika 3.20. Metoda setupRetrofit()

Na slici Slika 3.21. je prikazana predaja potrebnih parametara definiranim zahtjevom *getAllMeasurements()* te se metodom *enqueue(retrofit2.Callback<T>)* asinkrono izvršava zahtjev. Parametri koji se predaju metodi su statičke varijable unutar kojih su spremljeni podatci iz dijeljenih postavki aplikacije te su te varijable dostupne kroz cijelu aplikaciju. Također su

definirane i dvije *callback* metode – *onResponse* koja se poziva pri primljenom HTTP odgovoru; te *onFailure* u slučaju da se dogodi pogreška pri komunikaciji s poslužiteljem. Unutar *onResponse* metode se nastavlja rad s dobivenim podatcima. [19]

```
Call<Measurement> call = api.getAllMeasurements(
        identifier,
        "Bearer " + key,
        clickedSensorId,
        startTime
);

call.enqueue(new Callback<Measurement>() {
    @Override
    public void onResponse(Call<Measurement> call, Response<Measurement> response){
        Measurement measurement = response.body();
        if(measurement!= null) {
            List<Value> values = measurement.getValues();
            //do stuff
        }
        else {
            //do stuff
        }
    }
    @Override
    public void onFailure(Call<Measurement> call, Throwable t) {
        System.out.println(t);
    }
});
```

Slika 3.21. Pozivanje HTTP zahtjeva

Zahtjev za dohvaćanje *realtime* tokena se vrši analogno dohvaćanju vrijednosti senzora – metodi je potrebno predati mrežni identifikator te token koji služi za autorizaciju. Na slici ispod (Slika 3.22.) je prikazan opisani zahtjev.

```
@POST("api/v3/deviceNetwork/realtimeToken")
Call<ResponseBody> getRealtimeToken(
        @Header("X-DeviceNetwork") String deviceNetwork,
        @Header("Authorization") String token
);
```

Slika 3.22. Zahtjev za dohvaćanje realtime tokena

3.4.SignalR

Za integraciju SignalR-a, korištena je *ASP.NET SignalR – java client* biblioteka [20]. Budući da je ta biblioteka zastarjela i arhivirana, nije moguće dodati njenu ovisnost u aplikaciju, nego ju je potrebno dodati kao modul (u Android Studiu – File > Import Module). Na taj način se automatski u datoteku *settings.gradle* dodaju imena dodanih module. Pri *build*u aplikacije *Gradle* obuhvaća

samo module koji se nalaze unutar *settings.gradle* datoteke. Nakon dodavanja potrebnih modula za SignalR, u *settings.gradle* se nalaze sljedeći moduli:

```
include ':app', ':signalr-client-sdk', 'signalr-client-sdk-android'
```

Slika 3.23. Dodavanje SignalR modula

S obzirom da je navedena biblioteka arhivirana, ona se više ne održava te ju je prvenstveno bilo potrebno ažurirati korištene klase i biblioteke. Također je bilo potrebno napraviti izmjenu u *WebocketTransport.java* datoteci te je na slici Slika 3.24. prikazan kod koji je dodan u datoteku kako bi se omogućilo korištenje WebSocket protokola. Dakle, dodana je zamjena protokola predanog URL-a – HTTP protokol se mijenja u WS (*WebSocket*) protokol, a HTTPS (engl. *HyperText Transfer Protocol Secure*) u WSS (*WebSocket Secure*) protokol.

```
if(url.startsWith("https://")){
    isSsl = true;
    url = url.replace("https://", "wss://");
} else if(url.startsWith("http://")){
    url = url.replace("http://", "ws://");
}
```

Slika 3.24. Izmjene u SignalR biblioteci

Cijeli proces povezivanja s poslužiteljem putem SignalR-a je implementiran unutar *SignalRHubConnection.java* klase. Za povezivanje klijenta (aplikacije) i poslužitelja (IOTA-e), potrebno je napraviti hub konekciju i definirati *proxy* koji je posrednik između klijenta i poslužitelja. Kod ispod prikazuje definiranje konekcije i *proxy*ja (Slika 3.25.). Može se uočiti kako predani URL nema nastavak „/signalr“ budući da biblioteka to sama dodaje pri provjeri valjanosti URL-a. Pri definiranju konekcije, dodan je *logger* kako bi svi koraci pri spajanju bili dokumentirani i dostupni za provjeru pri razvijanju aplikacije što uvelike olakšava pronalazak pogrešaka i praćenje toka konekcije. [21]

```
hubConnection = new HubConnection("https://lab-api.ddm.iot-
accelerator.ericsson.net", "", true, logger);

hubProxy = hubConnection.createHubProxy("measurementHub");
```

Slika 3.25. Definiranje konekcije s APPIoT platformom

HubProxy omogućava da poslužitelj poziva metode koje su definirane na klijentu. To se omogućava s metodom *on*() pomoću koje se registrira događaj tj. metoda koja može biti pozvana.

Kod ispod (Slika 3.26.) prikazuje definiranje metode *NewMeasurement* koja se poziva pri slanju novih podataka od poslužitelja klijentu. Dakle, unutar te metode se deserijalizira dobiveni odgovor te se izvlače podatci potrebni za dodavanje nove točke na graf.

```
hubProxy.on("NewMeasurement", message -> {
    try {
        Gson gson = new Gson();
        String json = gson.toJson(message);
        JSONObject responseObject = new JSONObject(json);
        addEntry(chart, responseObject);
    } catch (Exception e) {
        e.printStackTrace();
    }
}, Object.class);
```

Slika 3.26. Metoda NewMeasurement

Kao što je već spomenuto u potpoglavlju 2.2., SignalR ima nekoliko mogućih načina transporta podataka. Budući da je WebSocket tehnologija jedina koja omogućava pravu dvosmjernu komunikaciju između klijenta i poslužitelja te je podržana od obje strane komunikacije, upravo ona je korištena za prijenos podataka (Slika 3.27.).

```
transport = new WebsocketTransport(hubConnection.getLogger());
hubConnection.start(transport);
```

Slika 3.27. Definiranje načina komunikacije

Također je moguće pozivanje metoda definiranih na poslužitelju od strane klijenta te je za to potrebno pozvati *invoke()* metodu na definiranom hub proxyju. U aplikaciji se, nakon uspješnog povezivanja (*hubConnection.connected()* metoda), pozivaju dvije metode sa poslužitelja – metoda za autentikaciju („*Authenticate*“) i metoda za dodavanje senzora („*AddSensor*“). Uz obje metode je potrebno predati i odgovarajuće parametre (*identifier* je statička varijabla čija je vrijednost dohvaćena iz dijeljenih postavki aplikacije, a *realtimeToken* je dohvaćen s IOTA-e) te su obje metode bez povratne vrijednosti što se može vidjeti na slici Slika 3.28.

```
hubConnection.connected(() -> {
    hubProxy.invoke("Authenticate", identifier, realtimeToken);
    hubProxy.invoke("AddSensor", resourceId);
});
```

Slika 3.28. Pozivanje metoda definiranih na poslužitelju

Za povezivanje putem SignalR-a, potrebno je napraviti instancu *SignalRHubConnection* klase te pozvati njenu metodu *startSignalR()*. Također je potrebno predati *realtime* token, ID senzora te instancu grafa na koji će se iscrtavati vrijednosti primljene od poslužitelja. Ta metoda se poziva unutar *RealtimeDataFragment*a, tj. pri njegovom stvaranju, ali tek nakon uspješnog dohvaćanja *realtime* tokena.

Unutar klase *SignalRHubConnection* se nalazi i metoda „*addEntry*“ za dodavanje novih mjerenja na graf u stvarnom vremenu. Metodi se predaje graf na koji se iscrtavaju vrijednosti te JSON objekt koji je dohvaćen sa servera i predstavlja novo pristiglu vrijednost tj. mjerenje. Unutar tog objekta se nalazi polje „*v*“ koje predstavlja vrijednost mjerenja te polje „*Unixtimestamp*“ koje predstavlja vrijeme u kojem je vrijednost nastala. Te dvije vrijednosti se prosljeđuju „*addEntry*“ metodi kao X i Y vrijednosti novog unosa (engl. *entry*) koji se iscrtava na graf. Konstruktor novog unosa za parametre prima dvije *float* vrijednosti (X i Y) što predstavlja problem budući da je *unixtimestamp* varijabla tipa *long*. Iz tog razloga, predaje se razlika između vremena dva mjerenja u sekundama. Prvo je potrebno dohvatiti vrijeme zadnjeg mjerenja te ga spremiti u varijablu *time*, zatim se od vremena svakog novog mjerenja oduzima varijabla *time* i dijeli se s tisuću kako bi se dobile sekunde. Opisani postupak je prikazan na kodu ispod (Slika 3.29.).

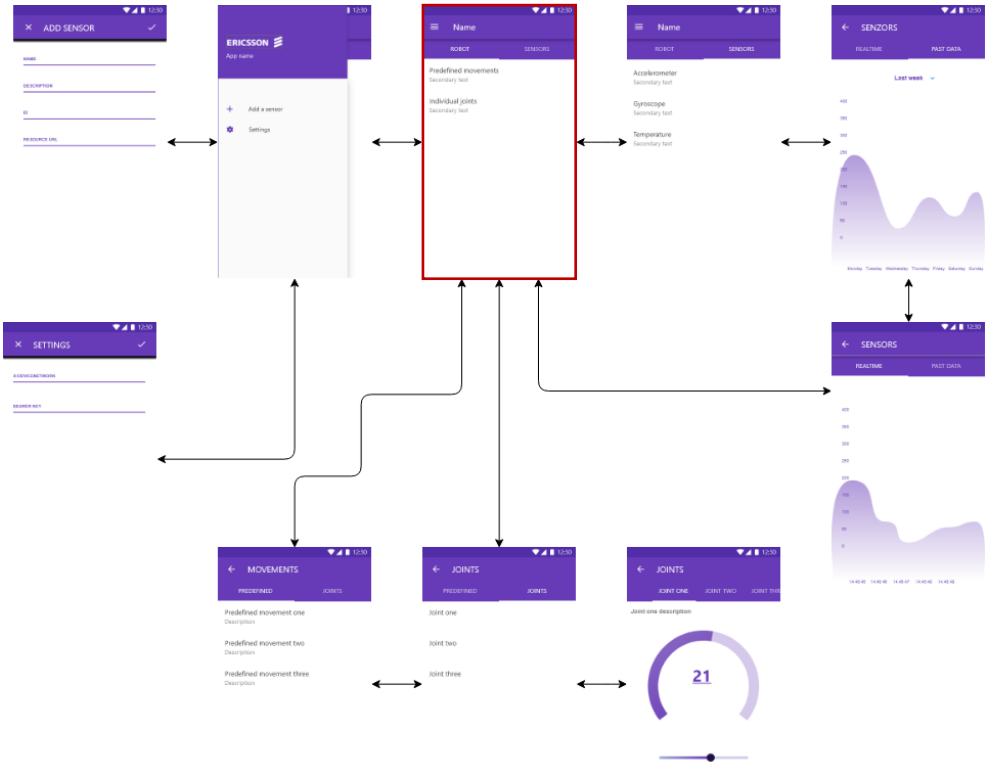
```
private void addEntry(LineChart chart, JSONObject jsonObject) {

    LineData data = chart.getData();
    long unixtimestamp = 0;
    float value = 0;
    try {
        unixtimestamp = ((Double)jsonObject.get("UnixTimestamp")).longValue();
        value = ((Double) jsonObject.get("v")).floatValue();
    }
    catch (Exception e){
        e.printStackTrace();
    }

    if(data != null) {
        ILineDataSet set = data.getDataSetByIndex(0);
        if(set == null){
            set = createSet();
            data.addDataSet(set);
        }
        if(time==0)
            time=unixtimestamp;
        float x = (float)((unixtimestamp-time)/1000);
        data.addEntry(new Entry(x, value), 0);

        hideProgressDialog();

        data.notifyDataChanged();
        chart.notifyDataSetChanged();

        chart.setVisibleXRangeMaximum(50);
        chart.moveViewToX(data.getEntryCount()-10);
    }
}
```

Slika 3.29. Metoda za dodavanje novih mjerenja na graf u stvarnom vremenu

Opisani postupak pretvorbe X vrijednosti mjerenja vrijedi i za prikazivanje mjerenja iz prošlosti budući da su i njihova vremena zapisana u obliku *unixtimestamp*a u milisekundama.

3.5. Izrada aplikacijskog sučelja

Prije početka izrade aplikacijskog sučelja, kreiran je *mock-up* aplikacije kojim se definirao izgled aplikacije. Na slici ispod (Slika 3.30.) je prikazan *mock-up* koji prikazuje prvu pretpostavku izgleda aplikacije te taj izgled može biti izmijenjen kroz realizaciju. Crvenom bojom je označen početni zaslon.



Slika 3.30. Mock-up aplikacije

3.5.1. ViewPager i RecyclerView

Aplikacija je pomoću *ViewPager* rasporeda podijeljena u tri dijela – nadzor vrijednosti senzora, automatsko pokretanje i manualno pokretanje ruke. *ViewPager* omogućava korisniku da se kreće kroz ponuđene kartice pokretima u desno ili u lijevo te je za njegovo korištenje potrebno, u XML datoteku, dodati <*ViewPager*> element te <*TabLayout*> element koji osigurava horizontalni prikaz kartica. U aplikaciji se *ViewPager* koristi unutar *MainActivity* i *SensorChartActivity* aktivnosti. Unutar svake kartice je implementiran njoj odgovarajući fragment koji omogućava prikaz podataka. Za prikaz fragmenata je potrebno implementirati *PagerAdapter* unutar kojega su definirane metode za popunjavanje i kretanje unutar *ViewPagera*. [22]

Unutar *SensorChartActivity*ja je implementiran prilagođeni *ViewPager* radi poboljšanja korisničkog iskustva pri korištenju grafova. Kreirana je klasa *CustomViewPager* (koja je proširenje klase *ViewPager*) pomoću koje se onemogućava mijenjanje fragmenata povlačenjem prstom (*swipe*anjem), već je potrebno odabrati određenu karticu s naslovom. Unutar *CustomViewPager*a se nalaze metode (vidljive na kodu ispod – Slika 3.31.) koje prepisuju predefinirane metode te se pomoću njih onemogućuje mijenjanje kartica povlačenje prstom.

```
@Override
public boolean onInterceptTouchEvent(MotionEvent ev) {
    if(this.enabled)
        return super.onInterceptTouchEvent(ev);
    return false;
}

@Override
public boolean onTouchEvent(MotionEvent ev) {
    if(this.enabled)
        return super.onTouchEvent(ev);
    return false;
}

public void setPagingEnabled(boolean enabled){
    this.enabled = enabled;
}
```

Slika 3.31. Metode unutar CustomViewPagera

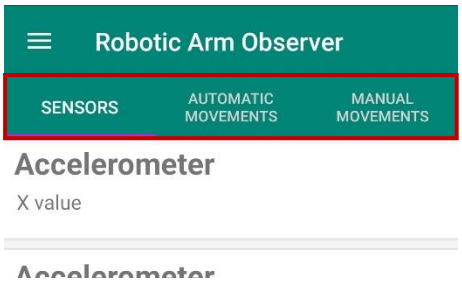
Nakon implementacije prilagođenog *ViewPager*a, u aktivnosti gdje se on koristi je potrebno metodi „*setPagingEnabled*“ predati „*false*“ vrijednosti kako bi se isključilo pomicanje fragmenata povlačenjem prstom.

```
chartViewPager.setPagingEnabled(false);
```

Slika 3.32. Onemogućavanje listanje kartica

Razlog posebnog *ViewPager*a je sprječavanje otežanog približavanja i pomicanja grafova budući da se graf i kartice *ViewPager*a pomiču istim pokretima.

Fragmenti predstavljaju dio korisničkog sučelja te se mogu smatrati kao modularni dio neke aktivnosti koji ima vlastiti životni ciklus. Za potrebe aplikacije su kreirani *SensorsListFragment*, *JointsListFragment* i *MovementsListFragment* koji se koriste unutar *MainActivity*ja za prikaz popisa dodanih senzora i popisa mogućih izbora (automatskih i manualnih pokreta) za upravljanje ruke te svaki od navedenih fragmenata predstavlja jednu karticu unutar TabLayouta (označene crvenom bojom na slici Slika 3.33.). Fragment *MovementsListFragment* (kartica *automatic movements*) služi za prikaz mogućih predefiniranih pokreta ruke (start, pause i stop), a *JointsListFragment* (kartica *manual movements*) služi za prikaz izbora zglobova (base, arm, elbow, wrist, hand i gripper) koji se mogu pomicati.



Slika 3.33. Prikaz kartica TabLayouta

Unutar *SensorChartActivity*ja su korišteni *PastDataFragment* i *RealtimeDataFragment* koji služe za prikaz grafova s podatcima iz prošlosti i podataka u stvarnom vremenu te i oni također predstavljaju kartice TabLayouta. Na slici Slika 3.34. je prikazano postavljanje *TabLayouta* (horizontalni dizajn za prikaz kartica), *ViewPager*a te odgovarajućih fragmenata unutar glavne aktivnosti – *MainActivity.class*.

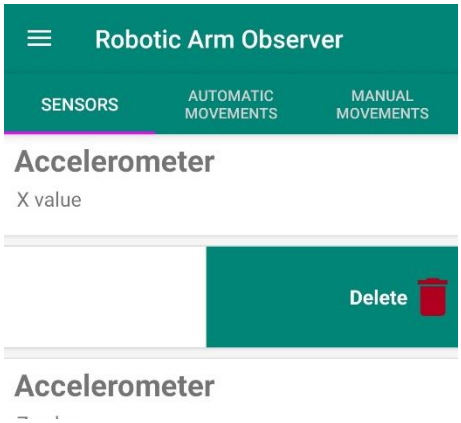
```
viewPagerAdapter = new ViewPagerAdapter(getSupportFragmentManager());
sensorListFragment = new SensorsListFragment();
robotListFragment = new RobotListFragment();

viewPagerAdapter.addFragment(sensorListFragment);
viewPagerAdapter.addFragment(movementsFragment);
viewPagerAdapter.addFragment(jointsFragment);

viewPager.setAdapter(viewPagerAdapter);
tabLayout.setupWithViewPager(viewPager);

tabLayout.getTabAt(0).setText("Sensors");
tabLayout.getTabAt(1).setText("Automatic movements");
tabLayout.getTabAt(2).setText("Manual movements");
```

Slika 3.34. Implementacija ViewPagera

Popisi unutar spomenutih fragmenata se prikazuju pomoću *RecyclerView*a koji omogućava dodavanje velikog skupa podataka unutar jednog popisa. *RecyclerView* je naprednija i fleksibilnija inačica *ListView*a koja omogućava promjenu izgleda redaka, animaciju elemenata te samostalno učitavanje podataka po potrebi.

Stavke unutar popisa su zapravo objekti *RecyclerView.ViewHolder* klase te *RecyclerView* kreira onoliko objekata koliko je prikazano stavki na popisu. Tim objektima se upravlja putem adaptera koji povezuje poglede s podatcima te napuhuje pojedini redak unutar popisa. Za svaki popis je potrebno kreirati novi adapter budući da se podatci unutar popisa razlikuju, a time se razlikuju i pojedine *ViewHolder* klase. [23]

Kako bi se ostvarila interakcija s *RecyclerView*om, potrebno je implementirati određeni osluškivač (engl. *listener*). Osluškivač se može implementirati kao posebno sučelje te je takav način korišten za dodavanje *swipe* pokreta u lijevo kako bi se obrisao pojedini senzor (Slika 3.35.).



Slika 3.35. Swipe u lijevo za brisanje senzora

Drugi način implementacije osluškivača je unutar *ViewHolder* klase te je taj način korišten za dodavanje osluškivača na odabir (pritisak) pojedinog retka. Dakle, *ViewHolder* klasa mora implementirati *View.OnClickListener*, a prikazana je na slici Slika 3.36. Dakle, na korisnikov odabir retka, putem pozicije se određuje koji senzor je odabran te se otvara nova aktivnost s podatcima senzora.

```
public class SensorViewHolder extends RecyclerView.ViewHolder implements
View.OnClickListener{

    private TextView tvListName;
    private TextView tvListDescription;
    public RelativeLayout viewBackground, viewForeground;

    private SensorViewHolder(@NonNull View itemView) {
        super(itemView);
        itemView.setOnClickListener(this);
        tvListName = itemView.findViewById(R.id.tvListName);
        tvListDescription = itemView.findViewById(R.id.tvListDescription);
        viewBackground = itemView.findViewById(R.id.background);
        viewForeground = itemView.findViewById(R.id.foreground);
    }
    @Override
    public void onClick(View v) {
        int position = getLayoutPosition();
        Sensor sensor = sensorList.get(position);
        Intent intent = new Intent(context, SensorChartActivity.class);
        context.startActivity(intent);
    }
}
```

Slika 3.36. Implementacija osluškivača

3.5.2. Postavke

Unutar postavki aplikacije, korisnik ima mogućnosti postavljanja parametara poput lokacije IoT platforme, mrežnog identifikatora, sigurnosnog ključa te početne pozicije ruke. Ti parametri se spremaju unutar aplikacije pomoću objekta dijeljenih postavki (engl. *Shared Preferences)* koji pokazuje na datoteku unutar koje su spremljeni navedeni parametri u obliku ključ-vrijednost. Ovaj način spremanja podataka je pogodan za male količine podataka te omogućava spremanje i mijenjanje podataka te njihovu dosljednost.

U *MainActivity*ju su definirane statičke varijable unutar kojih se spremaju vrijednosti postavki. Pri pokretanju aplikacije, dohvaćaju se vrijednosti koje su spremljene unutar dijeljenih postavki te se određuju njihove uobičajene vrijednosti koje se koriste prilikom prvog pokretanja aplikacije. Dohvaćene vrijednosti se spremaju unutar statičkih varijabli (vidljivo na slici Slika 3.37.).

```
private void setSettings() {
    SharedPreferences sharedPref=getSharedPreferences(getString(R.string.sha-
red_pref_settings), MODE_PRIVATE);
    IoTLocation = sharedPref.getString(getString(R.string.shared_pref_location),
"https://lab-api.ddm.iot-accelerator.ericsson.net/");
    identifier = sharedPref.getString(getString(R.string.shared_pref_identifier),
"613b7124-e2db-4e76-9feb-102a869bd497");
    key = sharedPref.getString(getString(R.string.shared_pref_key), get-
String(R.string.token));
    baseValue = sharedPref.getString(getString(R.string.shared_pref_base), "85");
    armValue = sharedPref.getString(getString(R.string.shared_pref_arm), "120");
    elbowValue = sharedPref.getString(getString(R.string.shared_pref_elbow),
"120");
    wristValue = sharedPref.getString(getString(R.string.shared_pref_wrist), "90");
    handValue = sharedPref.getString(getString(R.string.shared_pref_hand), "90");
    gripperValue = sharedPref.getString(getString(R.string.shared_pref_gripper),
"50");
    added = sharedPref.getBoolean("added", false);
}
```

Slika 3.37. Dohvaćanje vrijednosti postavki pri pokretanju aplikacije

Spomenute varijable se koriste kroz aplikaciju gdje god je potrebno koristiti lokaciju IoT platforme, mrežni identifikator ili sigurnosni ključ npr. prilikom dohvaćanja podataka sa platforme. Početne vrijednosti ruke se šalju na platformu prilikom pokretanja aplikacije i kako bi se ruka postavila u određeni početni položaj. Ruka također mijenja položaj pri svakoj izmjeni početnih vrijednosti.

Posljednja varijabla *added* predstavlja zastavicu koja služi za popunjavanje popisa senzora pri prvom pokretanju aplikacije. Zastavica je prvenstveno postavljena na *false* vrijednost te se prilikom prvog pokretanja aplikacije poziva metoda „addSensors()“, kojom se popunjava popis potrebnim podatcima o pojedinom senzoru, a zastavica se postavlja na *true* vrijednost kako se

popunjavanje ne bi više izvršavalo. Na ovaj način je omogućeno samo jedno izvršavanje spomenute metode budući da nije potrebno dodavati iste senzore više puta. Na slici ispod (Slika 3.38.) je prikazan kod za provjeru zastavice te je prikazana metoda kojom se popunjava popis senzora.

```
if(!added) {
    addSensors();
    editor.putBoolean("added", true).apply();
}

private void addSensors(){
    sensorViewModel.insert(new Sensor("Accelerometer", "X value", "e582ea88-0488-
4b5c-8788-a998caf24507", "19356/3313/0/5702"));
...
    sensorViewModel.insert(new Sensor("Arm angle", "Z value", "8d28cbd4-514a-42da-
9142-a7321c5a6267", "19356/34324/0/5704"));
}
```

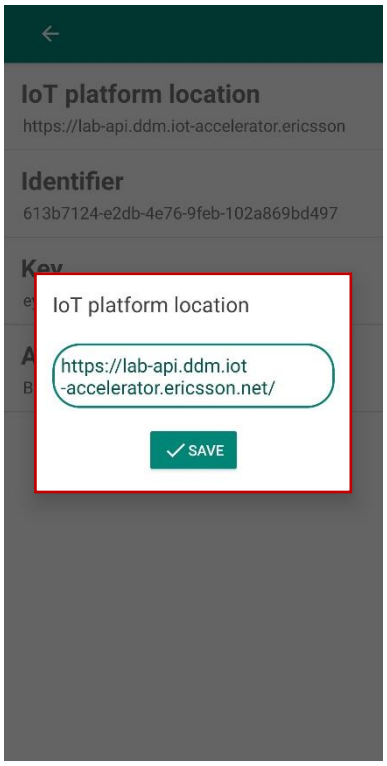
Slika 3.38. Popunjavanje popisa senzorima

Postavke aplikacije su prikazane unutar *RecyclerView*a koji olakšava prikaz postavki te njihovu izmjenu. Prilikom odabira određene postavke tj. odabira lokacije IoT platforme, identifikatora mreže i sigurnosnog ključa, korisniku se otvara prozor (engl. *dialog*) unutar kojega je potrebno unijeti tražene podatke za odabranu postavku. Ovisno o odabranoj postavci, uneseni podatci se spremaju na određeno polje unutar dijeljenih postavki te se pomoću metode „*apply()*“ izmijenjeni podatci spremaju na memoriju (vidljivo na kodu ispod – Slika 3.39. te Slika 3.40.). Tako spremljeni podatci će biti postojeći i nakon izlaska i ponovnog pokretanja aplikacije. Prije spremanja se provjerava je li polje za unos prazno te u slučaju da je, korisniku se prikazuje upozorenje i onemogućava se spremanje praznog polja.

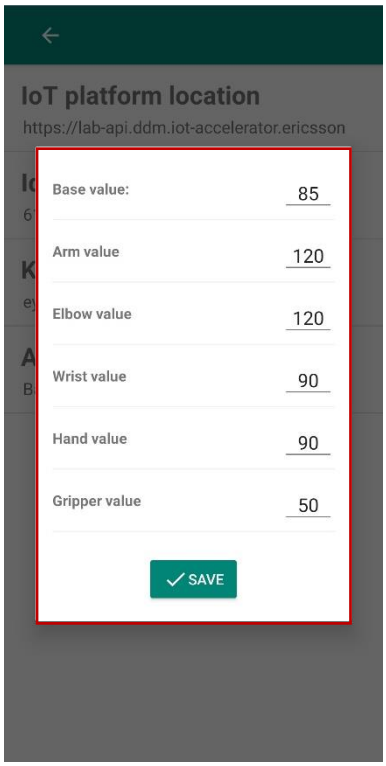
```
private void showInputDialog(int position){
    Dialog inputDialog = new Dialog(context);
    inputDialog.setContentView(R.layout.dialog_input);
    TextView tvDialogTitle = inputDialog.findViewById(R.id.tvDialogTitle);
    tvDialogTitle.setText(data[position]);
    EditText inputBox = inputDialog.findViewById(R.id.input_box);
    inputBox.setText(values[position]);
    Button saveButton = inputDialog.findViewById(R.id.btn_save);
    saveButton.setOnClickListener(view->{
        if(inputBox.getText().toString().trim().isEmpty())
            inputBox.setError("Please enter a value!");
        else {
            switch (position) {
                case 0:
                    IoTLocation = inputBox.getText().toString();
                    value.setText(inputBox.getText().toString());
                    editor.putString(context.getString(R.string.shared_pref_loca-
tion), inputBox.getText().toString()).apply();
                    inputDialog.dismiss();
                    break;
                case 1:
                    identifier = inputBox.getText().toString();
                    value.setText(inputBox.getText().toString());
                    editor.putString(context.getString(R.string.shared_pref_identi-
fier), inputBox.getText().toString()).apply();
                    inputDialog.dismiss();
                    break;
                case 2:
                    key = inputBox.getText().toString();
                    value.setText(inputBox.getText().toString());
                    editor.putString(context.getString(R.string.shared_pref_key),
inputBox.getText().toString()).apply();
                    inputDialog.dismiss();
                    break;
            }
        }
    });
    inputDialog.show();
}
```

Slika 3.39. Kod za prikaz prozora za unos postavki

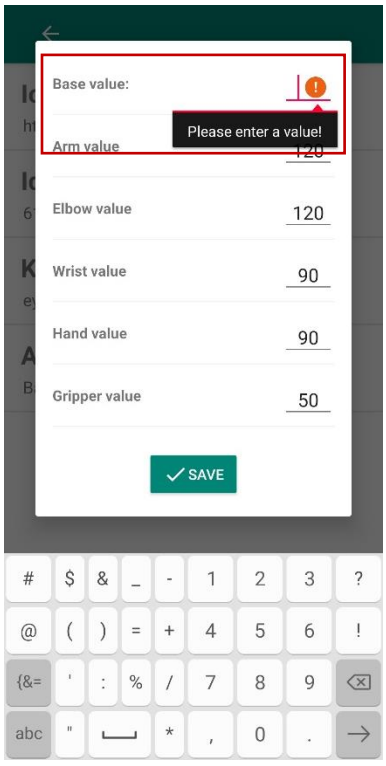
Prilikom odabira posljednje postavke (izmjena početne pozicije ruke tj. početnih vrijednosti pojedinog servomotora – zgloba), korisniku se otvara poseban prozor unutar kojega se mogu mijenjati početne vrijednosti svih zglobova (Slika 3.41.). Svaki zglob mora imati postavljenu vrijednost, dakle korisnik u svako polje mora unijeti određenu vrijednost (može biti i nula). U slučaju da unutar nekog polja nema vrijednosti, korisniku se ispisuje pogreška s porukom da je određeno polje prazno (Slika 3.42.).



Slika 3.40. Unos postavki



Slika 3.41. Unos početnih vrijednosti zglobova



Slika 3.42. Error u slučaju praznog polja

Nakon provjere svakog polja, vrijednosti se spremaju u statičke varijable koje su definirane u *MainActivity*ju (Slika 3.43.) te prilikom povratka na početni zaslon aplikacije te promjene položaja svakog zgloba šalju na server i ruka se pomiče na određeni položaj.

```
private void showPositionDialog(){
    Dialog positionDialog = new Dialog(context);
    positionDialog.setContentView(R.layout.dialog_position);
    EditText etBaseValue = positionDialog.findViewById(R.id.etBaseValue);
    ...
    Button saveButton = positionDialog.findViewById(R.id.btn_save);

    etBaseValue.setText(baseValue);
    ...

    saveButton.setOnClickListener(v -> {
        if(etBaseValue.getText().toString().isEmpty())
            etBaseValue.setError("Please enter a value!");
        else{
            baseValue = etBaseValue.getText().toString();
            editor.putString(context.getString(R.string.shared_pref_base),
                 baseValue).apply();
        }
        ...
    if(!etBaseValue.getText().toString().isEmpty() &&
       !etArmValue.getText().toString().isEmpty() &&
       !etElbowValue.getText().toString().isEmpty() &&
       !etWristValue.getText().toString().isEmpty() &&
       !etHandValue.getText().toString().isEmpty() &&
       !etGripperValue.getText().toString().isEmpty())
        positionDialog.dismiss();
});
    positionDialog.show();
}
```

Slika 3.43. Kod za unos početnih vrijednosti zglobova

3.5.3. Provjera dostupnosti interneta

U aplikaciji se provjera dostupnosti interneta izvršava pomoću klase *InternetChecker* unutar koje se nalazi metoda koja vraća *boolean* vrijednost postoji li internet konekcija ili ne. Za provjeru mreže je potrebno dodati dopuštenje za pristup mreži te dopuštenje za povezivanje s internetom u *Manifest* datoteku:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
```

Slika 3.44. Dopuštenja za pristup mreže i povezivanje s internetom

Metoda „*hasNetworkConnection*“ dohvaća sve aktivne veze i provjerava ima li uređaj pristup internetu putem mobilne ili WIFI mreže te je prikazana na slici ispod (Slika 3.45.).

```
public boolean hasNetworkConnection() {
    boolean haveConnectedWifi = false;
    boolean haveConnectedMobile = false;

    ConnectivityManager cm = (ConnectivityManager)
context.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo netInfo = cm.getActiveNetworkInfo();
    if(netInfo != null){
        if (netInfo.getTypeName().equalsIgnoreCase("WIFI"))
            if (netInfo.isConnected())
                haveConnectedWifi = true;
        if (netInfo.getTypeName().equalsIgnoreCase("MOBILE"))
            if (netInfo.isConnected())
                haveConnectedMobile = true;

    return haveConnectedWifi || haveConnectedMobile;
}
```

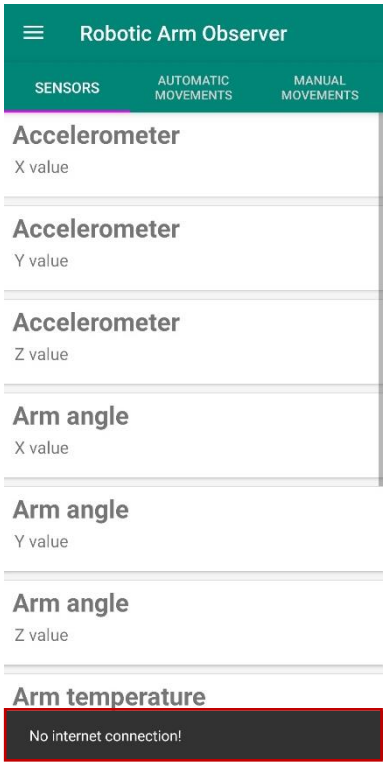
Slika 3.45. Metoda za provjeru povezanosti s internetom

Pomoću spomenute metode se unutar svake aktivnosti provjerava postojanost internetske veze te u slučaju da uređaj nije povezan s internetom, aplikacija ne izvršava metode kojima je potrebna konekcija s internetom. Na slici ispod (Slika 3.46.) je prikazan kod korišten pri kreiranju fragmenta za prikaz podataka u stvarnom vremenu te se može vidjeti kako se metode za povezivanje s IOTA-om uopće ne pozivaju u slučaju da uređaj nije spojen na Internet.

```
InternetChecker checker = new InternetChecker(context);
if(!checker.hasNetworkConnection()){
    Snackbar.make(view.findViewById(R.id.flRealtime), "No internet connection!",
2000).show();
}
else {
    showProgressDialog();
    createConnection();
    setupRealtimeLineChart();
}
```

Slika 3.46. Provjera povezanosti s internetom i ispis upozorenja

Na slici Slika 3.47. je prikazano upozorenje koje se prikazuje u slučaju da uređaj nije povezan s internetom.



Slika 3.47. Upozorenje da uređaj nije povezan s internetom

Provjera interneta se vrši iz razloga što aplikacija u velikom postotku ovisi postojanosti povezanosti s internetom. U slučaju da nema internetske konekcije, nije moguće dohvaćanje podataka s IOTA platforme (bilo u stvarnom vremenu ili podataka iz prošlosti) kao ni slanje podataka tj. naredbi na platformu (manualnih ili automatskih pokreta).

3.6. *MPAndroidChart* biblioteka

MPAndroidChart biblioteka je besplatan softver koji omogućava jednostavno korištenje raznovrsnih grafova za Android aplikacije. Za implementaciju unutar projekta, potrebno je dodati sljedeće ovisnosti:

```
repositories {
    maven { url "https://jitpack.io" }
}
//chart library
implementation 'com.github.PhilJay:MPAndroidChart:v3.1.0'
```

Slika 3.48. Dodavanje ovisnosti MPAndroidCharta

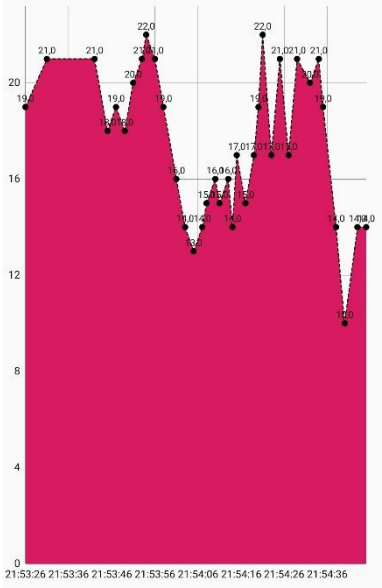
Za dodavanje grafova, potrebno je u željenu XML datoteku dodati pogled (engl. *view*) tj. *widget* grafa. U aplikaciji se koriste *LineChart widget*i i za prikaz podataka iz prošlosti (unutar *PastDataFragment.class*) i za prikaz podataka u stvarnom vremenu (unutar *RealtimeDataFragment.class*). LineChart widget predstavlja graf koji iscrtava vrijednosti te ih povezuje crtom.

Biblioteka omogućava mijenjanje dizajna grafova kao npr. dodavanje naslova, uređivanje X i Y osi, dodavanje legende, dodavanja animacija prikaza točaka te mijenjanje boja iscrtanih točaka. Također omogućava dodavanje osluškivača na grafove, tako da korisnik može pomicati prikaz grafova (približavati i udaljavati prikaz). Na slici Slika 3.49. su navedene neke od izmjena koje su se koristile za graf koji prikazuje podatke iz prošlosti.

```
xAxis.setPosition(XAxis.XAxisPosition.BOTTOM);
xAxis.setValueFormatter(new ValueFormatter() {

YAxis yAxis = lineChart.getAxisLeft();
yAxis.setAxisMinimum(0f);
lineChart.getAxisRight().setEnabled(false);

// disable legend
lineChart.getLegend().setEnabled(false);

//enable touch gestures
lineChart.setTouchEnabled(true);

//scaling both axis at the same time
lineChart.setPinchZoom(true);

// set listeners
lineChart.setOnChartValueSelectedListener(this);

// create marker to display box when values are selected
MyMarkerView mv = new MyMarkerView(context, R.layout.custom_marker_view);

// Set the marker to the chart
mv.setChartView(lineChart);
lineChart.setMarker(mv);

// enable scaling and dragging
lineChart.setDragEnabled(true);
lineChart.setScaleEnabled(true);

//draw points over time
lineChart.animateX(1500);
```

Slika 3.49. Definiranje izgleda grafa

Na slici Slika 3.50. je prikazan izgled grafa koji se temelji na prethodnom kodu.



Slika 3.50. Izgled grafa

Također je dodan marker koja se pojavljuje pri odabiru neke točke s grafa. Unutar markera se ispisuje vrijeme i datum (format „HH:mm:ss dd.MM.yyyy“) te vrijednost odabranog mjerenja tj. točke na grafu. Izgled markera je prikazan na slici Slika 3.51.

Slika 3.51. Izgled markera za graf

Budući da se vrijeme očitavanja vrijednosti senzora dobiva u obliku *unixtimestamp*a u milisekundama, za X os je potrebno formatirati dobiveni oblik u „HH:mm:ss“ oblik vremena. Dakle, grafu je predana razlika u sekundama između vremena novog mjerenja i vremena prvog prikazanog mjerenja.

Iz tog razloga je potrebno toj razlici dodati vrijeme prvog prikazanog mjerenja te pomnožiti s tisuću kako bi se dobilo vrijeme novog mjerenja u milisekundama (*unixtimestamp* u milisekundama) te se to vrijeme u milisekundama formatira u „HH:mm:ss“ i prikazuje na graf. Kod ispod prikazuje opisani postupak (Slika 3.52.).

```
xAxis.setValueFormatter(new ValueFormatter() {
    private final SimpleDateFormat mFormat = new SimpleDateFormat("HH:mm:ss",
Locale.ENGLISH);

    @Override
    public String getFormattedValue(float value) {
        long millis = time + (long)value * 1000;
        return mFormat.format(new Date(millis));
    }
});
```

Slika 3.52. Formatiranje X osi

4. REALIZACIJA MOBILNE APLIKACIJE – UPRAVLJANJE

Upravljanje robotske ruke se vrši na dva načina – upravljanje automatskim radnjama i upravljanje pojedinačnim zglobovima ruke tj. pojedinačnog servomotora. Zbog toga su kreirana dva modela podataka – „*Movement*“ i „*Joint*“ model koji sadrže podatke o automatskim i manualnim pokretima. Popis *movementList*, temeljen na modelu *Movement*, (Slika 4.1.) sadrži moguće automatske pokrete tj. ime i opis pokreta te naredbu koju je potrebno poslati serveru kako bi se izvršio taj pokret.

```
private List<Movement> addMovements(){
    movementList = new ArrayList<>();
    movementList.add(new Movement("Start", "start", "CMD:AUTO:START"));
    movementList.add(new Movement("Pause", "pause", "CMD:AUTO:PAUSE"));
    movementList.add(new Movement("Stop", "stop", "CMD:AUTO:STOP"));
    return movementList;
}
```

Slika 4.1. Dodavanje automatskih pokreta na popis

Popis za manualne radnje je analogan popisu za automatske radnje te se također sastoji od imena, opisa te naredbe potrebne za pokretanje. Ruka se sastoji od šest servomotora tj. sadrži šest stupnjeva slobode (engl. *Degrees of Freedom* – DOF). Stupnjevi slobode se odnose na broj osi po kojima se tijelo (u ovom slučaju ruka) može kretati u trodimenzionalnom prostoru. [24]

Tih šest zglobova je na serveru definirano kao baza, ruka, lakat, zglob, šaka i hvataljka te se unutar aplikacije koriste engleski nazivi. Na slici Slika 4.2. je prikazano dodavanje zglobova i njihovih parametara.

```
private List<Joint> addJoints(){
    jointList = new ArrayList<>();
    jointList.add(new Joint("Base", "Base description", "CMD:MANUAL:BASE:"));
    jointList.add(new Joint("Arm", "Arm description", "CMD:MANUAL:ARM:"));
    jointList.add(new Joint("Elbow", "Elbow description", "CMD:MANUAL:ELBOW:"));
    jointList.add(new Joint("Wrist", "Wrist description", "CMD:MANUAL:WRIST:"));
    jointList.add(new Joint("Hand", "Hand description", "CMD:MANUAL:HAND:"));
    jointList.add(new Joint("Gripper", "Gripper description",
"CMD:MANUAL:GRIPPER:"));
    return jointList;
}
```

Slika 4.2. Dodavanje zglobova ruke na popis

U ovom slučaju, budući da korisnik određuje pokret zgloba, naredbe za pokretanje nisu dovršene te će njihova nadogradnja razraditi kasnije u sljedećem potpoglavlju.

4.1.RabbitMQ

RabbitMQ predstavlja broker poruka koji prima i prosljeđuje podatke u obliku binarnih poruka. Broker je instaliran na korištenom serveru te se za implementaciju RabbitMQ-a unutar Android operacijskog sustava, koristi se RabbitMQ Java biblioteka koja se dodaje sljedećom ovisnosti:

```
//RabbitMQ
implementation 'com.rabbitmq:amqp-client:5.7.2'
```

Slika 4.3. Dodavanje RabbitMQ ovisnosti

Aplikacija pomoću RabbitMQ-a šalje poruke tj. naredbe na server te se na serveru nalazi modul za direktnu kontrolu ruke koji čita primljene poruke i realizira ih. Za slanje poruka je potrebno implementirati pošiljatelja (*producer*) i primatelja (*consumer*) te memoriju unutar koje se spremaju poruke za slanje (*queue*). [25]

Za uspostavljanje konekcije sa serverom te slanje poruka na server je kreirana RabbitMQConnection.java klasa. Povezivanje sa serverom se vrši pomoću *ConnectionFactory*ja koji omogućava kreiranje *Connection* instanci – AMQP konekcija. Takve konekcije se brinu za odabir protokola i autentikaciju te se koriste za otvaranje kanala, registriranje životnog ciklusa događaja te zatvaranje konekcija koje se više ne koriste. *ConnectionFactory* omogućava konfiguraciju parametara konekcije poput IP adrese, akreditacije (korisničko ime i lozinka), broj porta itd. Na slici ispod (Slika 4.4.) se može vidjeti konfiguracija instanci konekcija te se ona poziva prilikom svakog pokretanja aktivnosti koje šalju naredbe serveru. [26]

```
private void startConnection(){
    factory = new ConnectionFactory();
    factory.setHost("129.192.68.121");
    factory.setUsername("pdm_user");
    factory.setPassword("pass4pdm");
    factory.setPort(5672);
}
```

Slika 4.4. Konfiguracija konekcije sa serverom

Za slanje poruka je potrebno kreirati pošiljatelja koji uzima poruke iz unutarnjeg reda čekanja, a u slučaju pogreške ih ponovno postavlja u red tj. na kraj reda. Potrebno je kreirati kanal pomoću kojega se izvršava komunikacija s brokerom. Metoda „*sendMessage*“ (prikazana na slici ispod) prima određenu poruku koja se želi poslati te se, putem kanala, poruka šalje na broker u bajtovima. Unutar metode je određen naziv reda čekanja u kojemu se spremaju poruke spremne za slanje, također se postavlja *routingKey* na temelju kojeg se određuje odredište poruke. *Routing key* se

uspoređuje s *binding key*om postavljenim na serveru te se poruka šalje u slučaju podudaranja. [27] Opisana metoda je prikazana ispod na slici Slika 4.5.

```
public void sendMessage(String message){
    try {
        connection = factory.newConnection();
        channel = connection.createChannel();
        channel.queueDeclare("ra1_cmd_queue", false, false, false, null);
        channel.confirmSelect();

        try {
            channel.basicPublish("", "ra1_cmd_queue", null, message.getBytes());
            channel.waitForConfirmsOrDie();
        } catch (Exception e) {
            e.printStackTrace();
            try {
                queue.putFirst(message);
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            }
        }
    } catch (IOException | TimeoutException e) {
        e.printStackTrace();
    }
}
```

Slika 4.5. Slanje poruke (naredbe) na server

Unutar aktivnosti koja šalje naredbe na server kreirana je nit (engl. *Thread*) pomoću koje se izvršava pozadinska obrada podataka. Svaka nit mora implementirati metodu *run()* koja odrađuje posao izvan glavne niti. U ovom slučaju, unutar te metode se dohvaća prva poruka u redu čekanja te se poziva *sendMessage* metoda koja šalje podatke tj. dohvaćenu poruku (naredbu) serveru. [25]

Na slici Slika 4.6. je vidljiva metoda unutar koje se kreira nova nit te se ta metoda nalazi u *JointActivity*ju za slanje naredbi za manualne pokrete i u *MovementListAdapter*u za slanje naredbi za automatske pokrete.

```
public void publishToAMQP(){
    publishThread = new Thread(() -> {
        while(true){
            String message;
            try {
                message = queue.takeFirst();
                rabbitMQConnection.sendMessage(message);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    publishThread.start();
}
```

Slika 4.6. Spremanje poruke u red čekanja

4.1.1. Croller biblioteka [28]

Ruka se sastoji od šest zglobova tj. šest servomotora koji se mogu zakretati za 180 stupnjeva. Aplikacija koristi klizač (engl. *seekbar*) kako bi omogućila korisniku određivanje kuta zakretanja pojedinog motora te takvi pokreti predstavljaju manualne pokrete ruke. Croller biblioteka omogućava korištenje kružnog klizača - omogućava korisniku kružno povlačenje gumba kako bi mijenjao postavljenu vrijednost. Za korištenje je potrebno dodati sljedeću ovisnost:

```
//Circular seekbar
implementation 'com.sdsmdg.harjot:croller:1.0.7'
```

Slika 4.7. Dodavanje ovisnosti kružnog klizača

Vrijednosti koje se mogu odabrati na klizaču se nalaze u rasponu od 0 do 180 budući da se servomotori koji kontroliraju zglobove ruke mogu zakretati od 0 do 180 stupnjeva. Vrijednost napretka klizača je postavljena na temelju definirane početne vrijednosti svakog zgloba (vrijednost definirana u postavkama aplikacije te se prilikom pokretanja aplikacije zglob – servomotor pomiče na tu vrijednost tj. na taj položaj). Također je postavljen osluškivač pomoću kojega se prati pomicanje gumba. Pri svakom pomaku, trenutna vrijednost se upisuje na oznaku ispod klizača, a pri završetku pomicanja se odabrana vrijednost dodaje na kraj naredbe manualnih pokreta koja se zatim šalje na server. Na slici ispod (Slika 4.8.) je prikazan kod za prethodno objašnjene postupke.

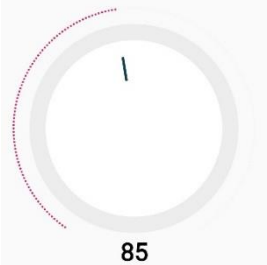
```
jointCroller = findViewById(R.id.jointCroller);
jointCroller.setIsContinuous(false);
jointCroller.setMin(0);
jointCroller.setMax(180);
if(!value.isEmpty())
    jointCroller.setProgress(Integer.parseInt(value));
jointCroller.setLabelSize(100);
jointCroller.setOnCrollerChangeListener(new OnCrollerChangeListener() {
    @Override
    public void onProgressChanged(Croller croller, int progress) {
        croller.setLabel(Integer.toString(progress));
    }

    @Override
    public void onStartTrackingTouch(Croller croller) { }

    @Override
    public void onStopTrackingTouch(Croller croller) {
        String amqpMessage = message.concat(Integer.toString(croller.getPro-
gress()));
        publishMessage(amqpMessage);
    }
});
```

Slika 4.8. Konfiguracija kružnog klizača

Klizač se u XML-u dodaje pomoću <*Croller*> elementa te je unutar tog elementa definiran i izgled klizača. Na kodu ispod (Slika 4.9.) se može vidjeti postavljanje boje klizača, kao i njegovog radijusa i debljine pojedinih elemenata te je na slici Slika 4.10. prikazan konačni izgled kružnog klizača.

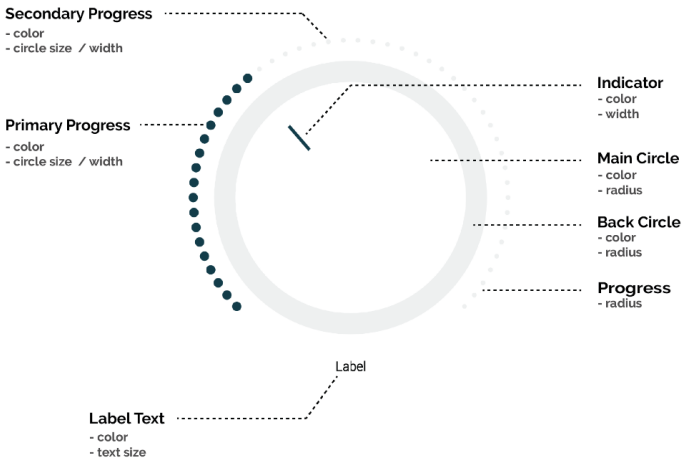
```
<com.sdsmdg.harjot.crollerTest.Croller
    android:id="@+id/jointCroller"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_centerInParent="true"
    app:back_circle_color="#EDEDED"
    app:indicator_color="#0B3C49"
    app:indicator_width="10"
    app:label_color="#000000"
    app:main_circle_color="#FFFFFF"
    app:progress_primary_color="@color/colorAccent"
    app:progress_secondary_color="#EEEEEE"/>
```

Slika 4.9. Element kružnog klizača u XML-u



Slika 4.10. Izgled kružnog klizača

Na slici ispod (Slika 4.11.) su prikazani svi elementi klizača te njihovi atributi koji se mogu mijenjati. Ti atributi se mogu mijenjati i unutar java koda, no unutar aplikacije je, radi jednostavnosti koda, to odrađeno u XML-u.



Slika 4.11. Atributi kružnog klizača – Crollera[28]

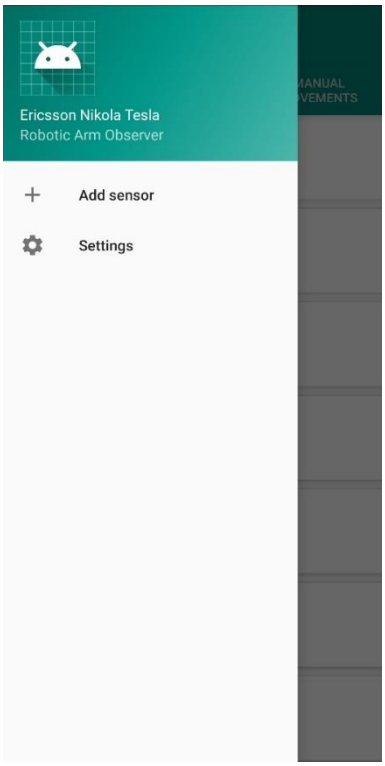
4.2.Testiranje aplikacije

Nakon što je aplikacija realizirana, potrebno ju je testirati kako bi se potvrdila funkcionalnost svih zahtjeva aplikacije. Kroz proces izrade za pokretanje aplikacije je korišten stvarni Android uređaj – Huawei P20.

Pri pokretanju aplikacije, pojavljuje se početni zaslon na kojemu se nalaze tri kartice – *Sensors* unutar koje se nalazi popis svih senzora čije vrijednosti se mogu promatrati; *Automatic movements* koja prikazuje popis svih automatskih kretnji ruke; te *Manual movements* na kojoj se nalazi popis zglobova ruke (servomotora) koje je moguće manualno pomicati. Na slici Slika 4.12. je prikazan početni zaslon te je prikazana prva kartica *sensors*.

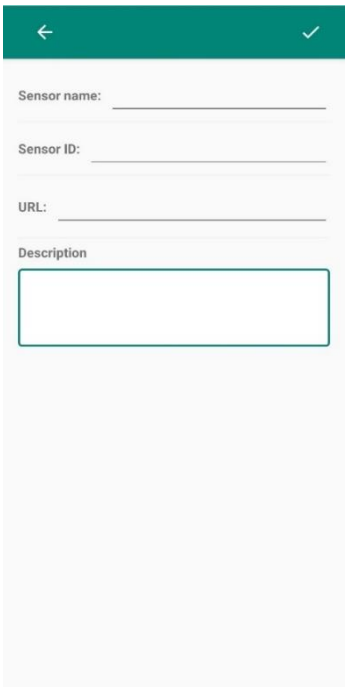
Slika 4.12. Početni zaslon

U gornjem lijevom kutu (na početnom zaslonu – Slika 4.12.) se nalazi gumb za dodatni izbornik unutar kojega se nalazi opcija za dodavanje novih senzora na popis te postavke aplikacije. Izbornik s navedenim opcijama je prikazan na slici ispod (Slika 4.13.).



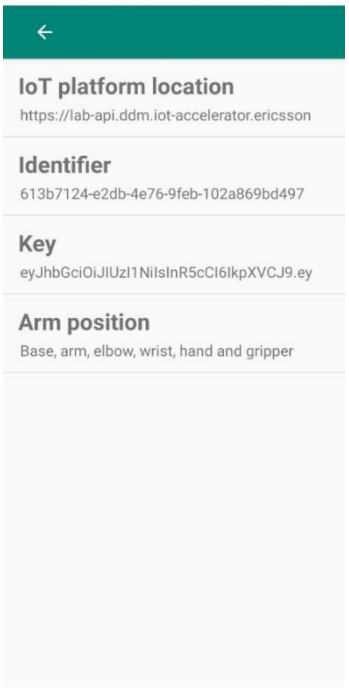
Slika 4.13. Izbornik

Dodavanje senzora se vrši putem zaslona na slici Slika 4.14. te je potrebno unijeti ime, opis, ID i URL senzora. Svi spomenuti parametri su definirani na APPIoT platformi. Nakon ispravnog unošenja podataka te njihovog spremanja na uređaj, dodani senzor se prikazuje na popisu senzora.



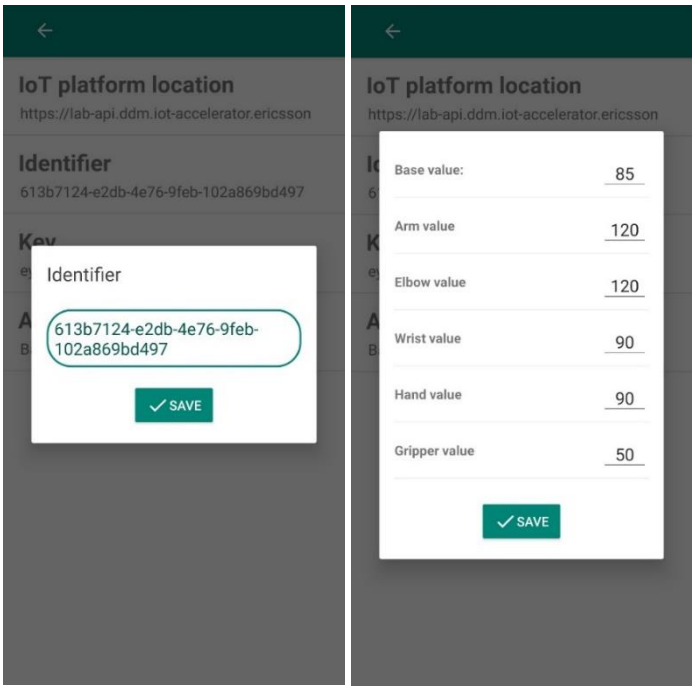
Slika 4.14. Dodavanje senzora

U postavkama aplikacije je moguće mijenjati razne parametre koji se koriste kroz cijelu aplikaciju – lokacija IoT platforme, mrežni identifikator, sigurnosni ključ te početne vrijednosti pojedinog zgloba (Slika 4.15.). Te početne vrijednosti predstavljaju poziciju na koju se svaki servomotor postavi prilikom pokretanja aplikacije.



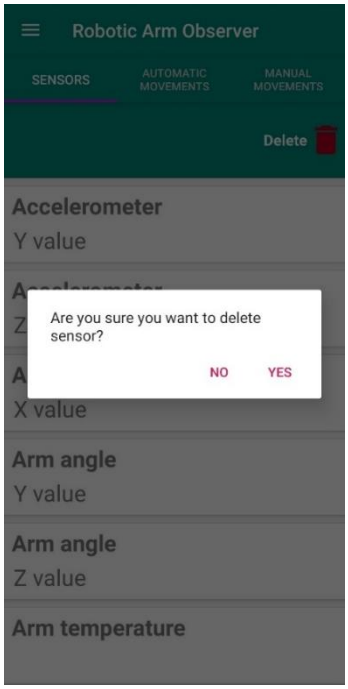
Slika 4.15. Postavke aplikacije

Odabirom određene postavke, korisniku se prikazuje prozor u kojemu se može unijeti odabrana postavka tj. vrijednost ili parametar (Slika 4.16.).



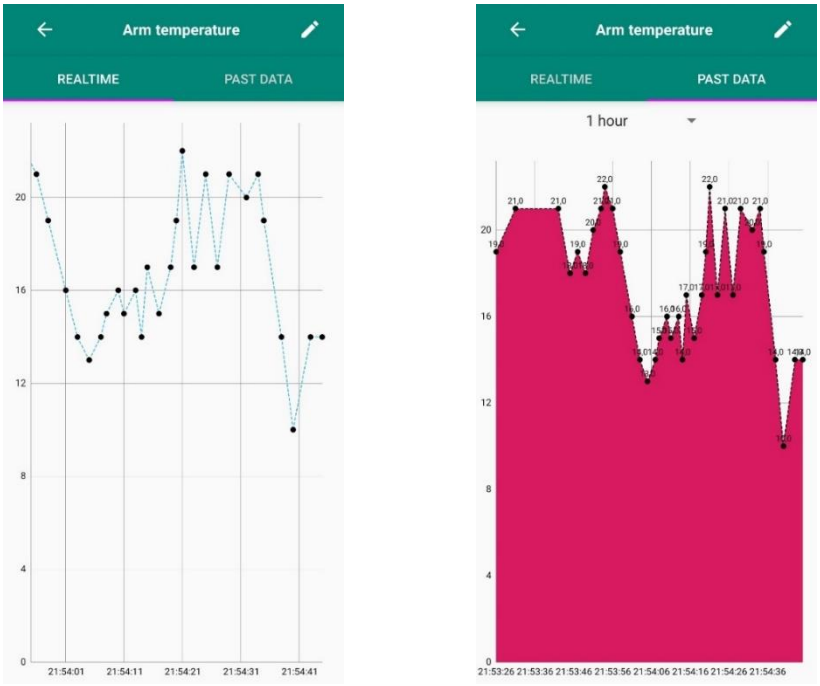
Slika 4.16. Unos parametara postavki

Na početnom zaslonu, svaki senzor ima svoje ime i opis te ga je moguće izbrisati pomicanjem pojedinog retka na lijevu stranu (Slika 4.17.). Prilikom pomicanja retka, korisnika se pita je li siguran da želi obrisati s popisa odabrani senzor.



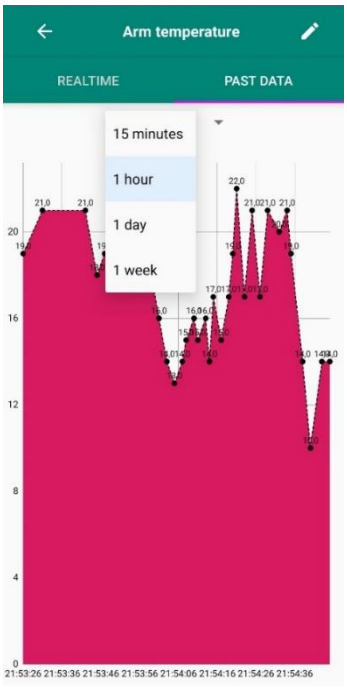
Slika 4.17. Brisanje senzora s popisa

Odabirom određenog senzora (klikom na senzor) se prikazuje novi zaslon (Slika 4.18.) sa grafovima te karticama za prikaz podataka u stvarnom vremenu (*realtime*) i podataka iz prošlosti (*past data*).



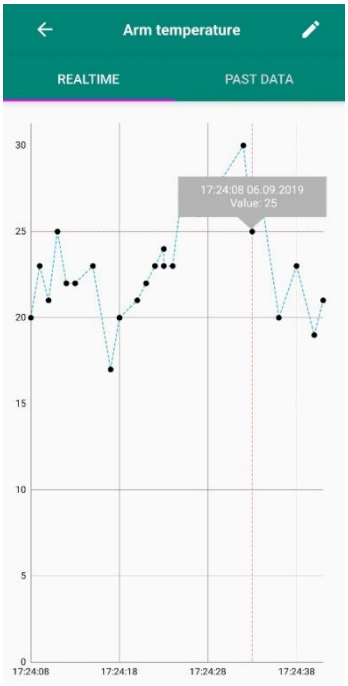
Slika 4.18. Grafovi za prikaz podataka

Na grafu za prikaz podataka u stvarnom vremenu, podatci se iscrtavaju kako stižu na platformu, a za podatke iz prošlosti postoji izbornik s rasponom dohvaćanja podataka (Slika 4.19.) – mogu se dohvatiti podatci na platformi spremljeni u zadnjih petnaest minuta, u zadnjem satu, danu te zadnjem tjednu.



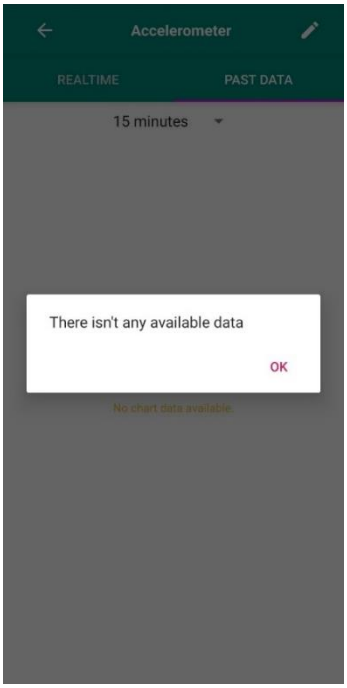
Slika 4.19. Izbornik raspona vremena

Korisnik može odabrati pojedine točke (mjerenja) na grafu te se time prikazuje marker unutar kojega se nalazi vrijeme i datum mjerenja te vrijednost mjerenja (Slika 4.20.)



Slika 4.20. Marker s dodatnim podatcima

U slučaju da ne postoje dostupni podatci (razlog nepostojanja podataka može biti da ruka ne radi ili da podatci još nisu dostupni na platformi – postoji kašnjenje od 5 minuta), korisniku se prikazuje prozor na sliciSlika 4.21.



Slika 4.21. Obavijest da nema dostupnih podataka

U gornjem desnom kutu (na slici Slika 4.18.) se nalazi opcija za uređivanje podataka senzora. Zaslon za uređivanje senzora je prikazan na slici Slika 4.22. te se može uočiti kako je onemogućeno mijenjanje ID-a senzora iz razloga što ID predstavlja primarni ključ svakog senzora.

Slika 4.22. Uređivanje podataka senzora

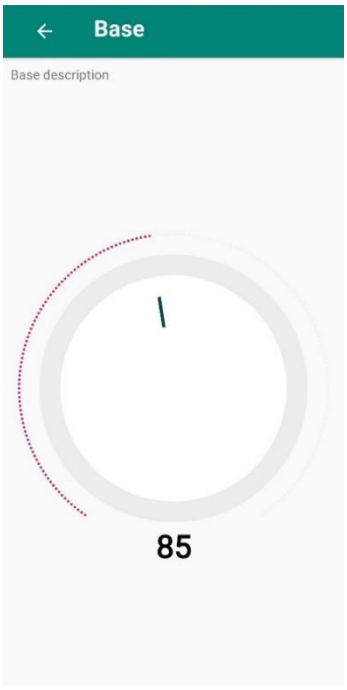
Druga kartica sadrži popis s automatskim pokretima ruke; to podrazumijeva pokretanje, zaustavljanje i pauziranje pojedine radnje ruke (Slika 4.23.) koji se omogućuju odabirom željene radnje.

Slika 4.23. Automatski pokreti ruke

Posljednja kartica sadrži popis svih zglobova koji se nalaze na ruci tj. popis servomotora koji su nazvani po zglobovima kako bi se lakše razlikovali (Slika 4.24.). Kao što se može vidjeti sa slike, pored svakog zgloba je napisana njegova uobičajena vrijednost koja je definirana u postavkama aplikacije.

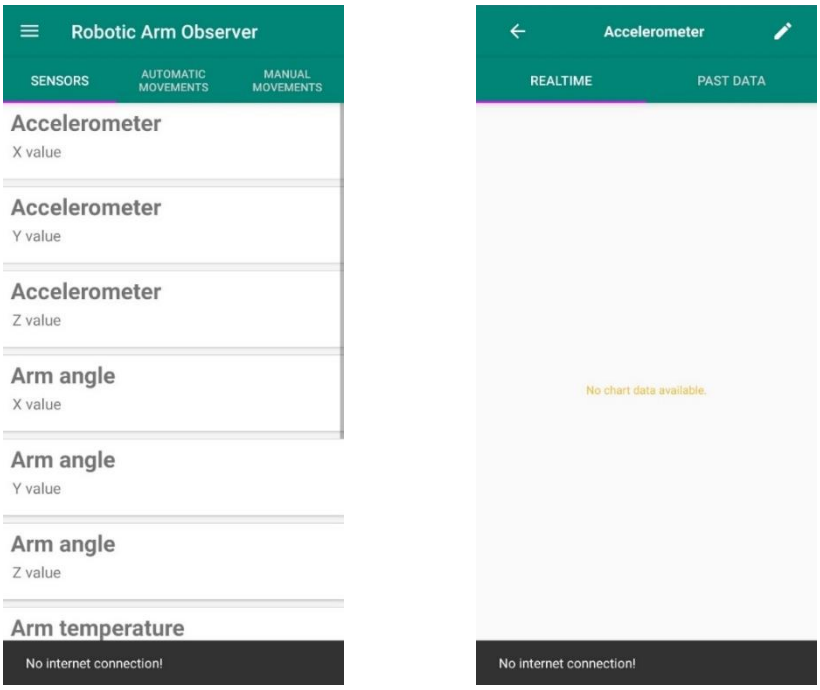
Slika 4.24. Manualni pokreti rukom

Odabirom pojedinog zgloba, korisniku se otvara zaslon sa kružnim klizačem pomoću kojega se može mijenjati položaj odabranog zgloba (Slika 4.25.).



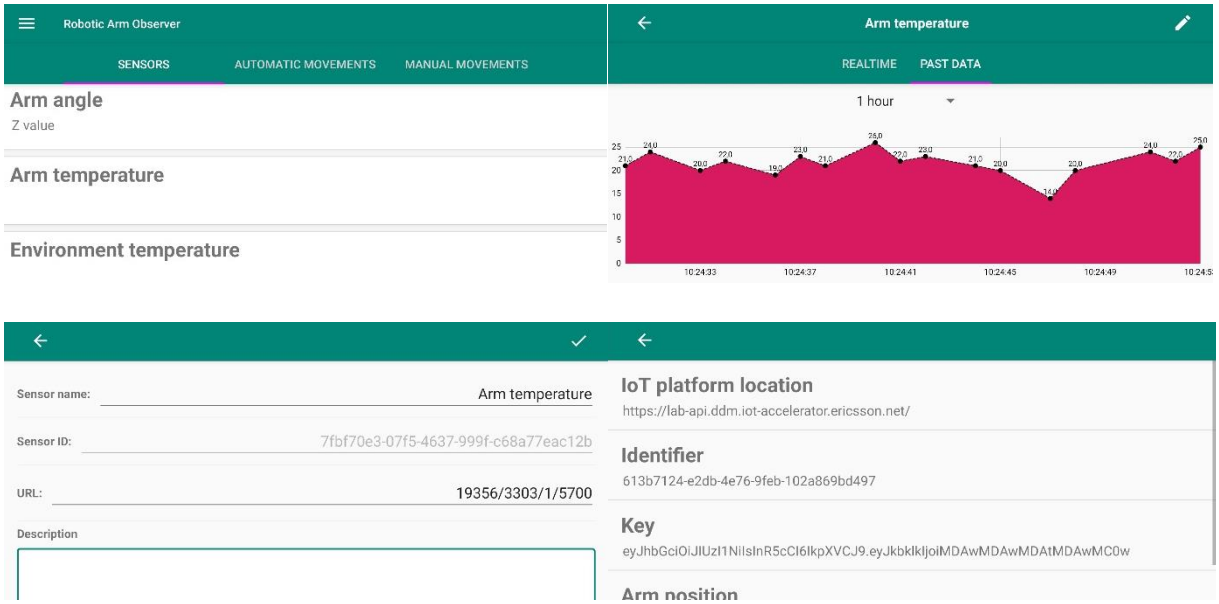
Slika 4.25. Kružni klizač za zglob

Bitno je napomenuti kako je aplikaciji potrebna Internet konekcija kako bi ispravno radila. U slučaju da mobilni uređaj nije spojen na Internet, korisniku se prikazuje obavijest (prikazane na slici Slika 4.26. – na dnu ekrana) da nema Internet konekcije te nije moguće dohvaćanje bilo kakvih podataka, kao ni slanje podataka na IOTA-u.



Slika 4.26. Obavijest da uređaj nije povezan s internetom

Aplikacija također ima mogućnost rada u horizontalnom položaju uređaja. Na slikama ispod su prikazani zasloni u takvom načinu.



Slika 4.27. Prikaz rada aplikacije u horizontalnom načinu

5. ZAKLJUČAK

Zahvaljujući sve većem tehnološkom napretku, omogućena je realizacija raznih uređaja i aplikacija koje olakšavaju svakidašnje poslove. Tvrtka Ericsson Nikola Tesla osmislila je prototip sustava koji omogućava upravljanje i praćenje rada robotske ruke na daljinu. Kako bi se dalje olakšalo praćenje ruke, razvijena je Android mobilna aplikacija te je njen razvoj bio i cilj ovog rada. Aplikacija omogućava praćenje rada robotske ruke čime osigurava dostupnost podataka te njeno udaljeno upravljanje.

Na ruci se nalazi niz integriranih senzora te se podatci sa senzora prikupljaju pomoću Arduino mikroupravljača i šalju na IoT platformu. Korištena IoT platforma je Ericsson IoT Accelerator. Razvoj aplikacije je zahtijevao korištenje tehnologija poput ASP.NET SignalR, AMQP te razvojno okruženje Android Studio. SignalR omogućava korištenje web funkcionalnosti aplikacija te je korišten za dohvaćanje i prikazivanje podataka u stvarnom vremenu. Tehnologija koristi WebSocket protokol pomoću kojega se ostvaruje kontinuirana, dvosmjerna komunikacija između poslužitelja i klijenta (u ovom slučaju između aplikacije i Ericsson IoT Accelerator platforme). AMQP protokol je korišten za slanje upravljačkih poruka na platformu. Unutar aplikacije se protokol koristi za slanje naredbi za pomicanje robotske ruke.

Kroz rad je prikazan razvoj aplikacije te je na kraju potvrđena njena funkcionalnost i ispravnost.

Aplikaciju je moguće i dodatno poboljšati i unaprijediti dodavanjem raznih mogućnosti za upravljanje rukom. Jedan od prijedloga je mogućnost dodavanja novog načina upravljanja korištenjem kamere i upravljanje relativnom pozicijom hvataljke u prostoru (lijevo/desno, gore/dolje, naprijed/nazad).

LITERATURA

[1] Ericsson, IoT Accelerator [online], Ericsson, 2019., dostupno na: https://www.ericsson.com/en/portfolio/iot-and-new-business/iot-solutions/iot-accelerator [11.06.2019.]

[2] C. Aliferi, Android Programming Cookbook [online], Exelixis Media P.C., 2016., dostupno na: http://enos.itcollege.ee/~jpoial/allalaadimised/reading/Android-Programming-Cookbook.pdf [11.06.2019.]

[3] Android Developers, Meet Android Studio [online], Android Developers, dostupno na: https://developer.android.com/studio/intro/index.html [12.06.2019.]

[4] Android Developers, AndroidX Overview [online], Android Developers, dostupno na: https://developer.android.com/jetpack/androidx [20.06.2019.]

[5] P. Fletcher, Introduction to SignalR [online], Microsoft, 2014., dostupno na: https://docs.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr, [21.06.2016.]

[6] I. Fette, A. Melnikov, The WebSocket Protocol [online], Internet Engineering Task Force (IETF), 2011., dostupno na: https://tools.ietf.org/html/rfc6455 [11.06.2019.]

[7] Ericsson, IOTA Swaagger, dostupno na: https://iotabusinesslab-api.sensbysigma.com/swagger/ui/index, [22.06.2019.]

[8] Ericsson, Device & Data Management: Part of IoT Accelerator [online], Ericsson, 2018., dostupno na: https://docs.ddm.iot-accelerator.ericsson.net/ [11.06.2019.]

[9] L. Johansson, Part 1: RabbitAMQP for Begginers – What is RabbitAMQP [online], CloudAMQP, 2015. dostupno na: https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html, [28.06.2019.]

[10] RabbitMQ, AMQP 0-9-1 Model Explained [online], RabbitMQ, dostupno na: https://www.rabbitmq.com/tutorials/amqp-concepts.html, [28.06.2019.]

[11] Google Developers Codelabs, Android Room with a View [online], Google Developers Codelabs, dostupno na: https://codelabs.developers.google.com/codelabs/android-room-with-a-view/#0, [24.06.2019.]

[12] Android Developers: Save dana in a local database using Room [online], Android Developers dostupno na: https://developer.android.com/training/data-storage/room/index.html, [25.06.2019.]

[13] Android Developers Guides, Defining data using Room entities [online], Android Developers, dostupno na: https://developer.android.com/training/data-storage/room/defining-data.html, [25.06.2019.]

[14] Android Developers, Accessing data using Room DAOs [online], Android Developers, dostupno na: https://developer.android.com/training/data-storage/room/accessing-data.html, [25.06.2019.]

[15] Android Developers, Database [online], Android Developers, dostupno na: https://developer.android.com/reference/androidx/room/Database.html, [25.06.2019.]

[16] Android Developers, Handling Lifecycles with Lifecycle-Aware Components [online], Android Developers, dostupno na: https://developer.android.com/topic/libraries/architecture/lifecycle, [25.06.2019.]

[17] Android Developers Guides, ViewModel [online], Android Developers, dostupno na: https://developer.android.com/topic/libraries/architecture/viewmodel, [26.06.2019.]

[18] L. Vogel, Using Retrofit 2.x as REST client [online], Vogella, 2018., dostupno na: https://www.vogella.com/tutorials/Retrofit/article.html, [22.03.2019.]

[19] Square, Retrofit [online], Square GitHub, dostupno na: https://square.github.io/retrofit/, [23.06.2019.]

[20] SignalR java-client [online], dostupno na: https://github.com/SignalR/java-client, [23.06.2019.]

[21] B. Gaster, ASP.NET SignalR Hubs API Guide [online], Microsoft, dostupno na: https://docs.microsoft.com/en-us/aspnet/signalr/overview/guide-to-the-api/hubs-api-guide-net-client, [23.06.2019.]

[22] Android Developers Guides, Create swipe views with tabs [online], Android Developers, dostupno na: https://developer.android.com/guide/navigation/navigation-swipe-view, [25.06.2019.]

[23] Android Developers Guides, Create a List with RecyclerView [online], Android Developers, dostupno na: https://developer.android.com/guide/topics/ui/layout/recyclerview, [25.06.2019.]

[24] Technopedia, Six Degrees of Freedom (6DOF) [online], Technopedia, dostupno na: https://www.techopedia.com/definition/12702/six-degrees-of-freedom-6dof, [20.08.2019.]

[25] L. Johansson: Get started with RabbitMQ on Android (Android Studio) [online], CloudAMQP, 2015. dostupno na: https://www.cloudamqp.com/blog/2015-07-29-rabbitmq-on-android.html, [20.08.2019.]

[26] RabbitMQ, Java Client API Guide [online], RabbitMQ, dostupno na: https://www.rabbitmq.com/api-guide.html#connecting, [20.08.2019.]

[27] RabbitMQ, Java Tutorials [online], RabbitMQ, dostupno na: https://www.rabbitmq.com/tutorials/tutorial-one-java.html, [20.08.2019.]

[28] H. S. Oberai, Croller [online], GitHub, 2017., dostupno na: https://github.com/harjot-oberai/Croller, [20.08.2019.]

SAŽETAK

Cilj diplomskog rada je bio realizirati Android aplikaciju koja korisnicima omogućava dohvaćanje vrijednosti senzora koji su postavljeni na robotsku ruku te omogućava upravljanje ruke. Vrijednosti senzora se dohvaćaju sa Ericssonove IoT Accelerator platforme te aplikacija omogućava dohvaćanje i nadziranje podataka u stvarnom vremenu, kao i podataka iz prošlosti. Za dohvaćanje u stvarnom vremenu, korištena je ASP.NET SignalR tehnologija, a za dohvaćanje podataka iz prošlosti HTTP zahtjevi. Aplikacija također omogućuje upravljanje rukom ili predefiniranim pokretima ili pomicanjem pojedinačnih zglobova. Komunikacija između aplikacije i Arduina koji upravlja robotskom rukom se vrši pomoću AMQP-a. Aplikacija je izrađena u Android Studio razvojnom okruženju te korisnicima olakšava nadzor i upravljanje robotske ruke.

Ključne riječi: Android, Android Studio, Java, ASP.NET SignalR, IoT Accelerator, AMQP, RabbitMQ

ABSTRACT

Robotic arm Control System using IoT platform and Android Application

The main goal of this thesis was to develop an Android application which enables users to manage and supervize the values of multiple sensors integrated on a robotic arm. It also enables users to control the arm using the application. The values from the sensors are stored on Ericsson's IoT Accelerator platform and the application can show realtime data or retrieve past dana. ASP.NET SignalR technology was used for getting the data in realtime and HTTP requests were used for retrieving the past data. The application also enables controlling the arm by using predefined movements or by moving each joint individually. Communication between the application and the Arduino microcontroller is achieved using the AMQP. The enviroment used for developing the application was Android Studio which is the official integrated development enviroment for Android operating systems.

Keywords: Android, Android Studio, Java, ASP.NET SignalR, IoT Accelerator, AMQP, RabbitMQ

ŽIVOTOPIS

Anamarija Blavicki je rođena 30. srpnja 1996. godine u Slavonskom Brodu, Hrvatska. Godine 2002. započinje osnovnoškolsko obrazovanje u OŠ Earl M. Lawson, Leavenworth, Kansas, SAD. Nakon završenog prvog razreda, obrazovanje nastavlja u OŠ Orašje, Bosna i Hercegovina sve do sedmog razreda. 2008. godine se ponovo seli u Sjedinjene Američke države gdje dovršava svoje osnovnoškolsko obrazovanje u Walter Colton Middle School, Monterey, California. Nakon osnovne škole, pohađa jezičnu gimnaziju Matija Mesić u Slavonskom Brodu te 2014. upisuje Elektrotehnički fakultet u Osijeku, smjer Elektrotehnika, a nakon prve godine se opredjeljuje za smjer Komunikacije i Informatika. Preddiplomski studij Elektrotehnike završava 2017. godine s temom završnog rada „Primjena informacijsko-komunikacijske tehnologije kao promocijskog alata u svrhu realizacije marketinške strategije na primjeru hrvatske turističke destinacije“ koji izrađuje u suradnji s Nacionalnim parkom Paklenica. Tijekom ljeta 2017. sudjeluje na Ljetnom kampu (*Summer Camp*) – ljetna radionica tvrtke Ericsson Nikola Tesla d.d. u Zagrebu gdje stječe praktično iskustvo u istraživanju i razvoju pod vodstvom Ericssonovih stručnjaka. Nakon uspješno stečene titule prvostupnice Elektrotehnike, upisuje sveučilišni diplomski studij Elektrotehnike, smjer Komunikacije i Informatika, izborni blok Mrežne tehnologije. Na diplomskog studiju sudjeluje u izradi projekta fakultetskog natječaja Pro-Student „Feritoskop – Android bežični osciloskop s pomoću Arduino mikroprocesora“. 2018. godine se ponovno prijavljuje na Ljetni kamp tvrtke Ericsson Nikola Tesla d.d. te u istoj tvrtki odrađuje stručnu praksu na posljednjoj godini studija. 2019. godine dobiva nagradu za najbolju studenticu na Fakultetu Elektrotehnike, Računarstva i Informacijskih Tehnologija od strane Lions Cluba, Osijek. Iste godine dobiva i Rektorovu nagradu za seminarski rad iz kolegija Internet objekata pod nazivom „SmartLock“.

Anamarija Blavicki

PRILOZI

CD

- Rad u .doc i pdf formatu
- Android studio projekt
- .apk datoteka aplikacije