SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Bebek

Patrik Dolovski

Josip Gagro

Tomislav Hirš

Ivan Oršolić

Tehnička dokumentacija aplikacije Athena

Varaždin, 2017.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

AIR1717:

Ivan Bebek, 0016114186

Patrik Dolovski, 0016110881

Josip Gagro, 0016110509

Tomislav Hirš, 0016110993

Ivan Oršolić, 0016114170

Github repozitorij

SCRUM dokumentacija

Tehnička dokumentacija aplikacije Athena

Mentor:

Doc.dr.sc. Zlatko Stapić

Varaždin, studeni 2017.

Sadržaj

1. Uvod

1.1. Svrha dokumenta

Svrha ovog dokumenta je detaljni opis tehničkih zahtjeva proizvoda. Aplikacija je razvijana na SCRUM metodi razvoja softvera, te kao takav pristup razvijanja softvera obuhvaća sustavni prikaz razvoja aplikacije *Athena*.

1.2. Korištene konvencije

Prilikom pisanja ovog dokumenta pratio se IEEE 830-1998 standard.

1.3. Ciljana publika dokumenta te preporuke za čitanje dokumenta

Publiku kojoj je dokument namijenjen čine prvenstveno razvojni tim, ali i treće osobe koje imaju želju za dublje razumijevanje funkcioniranja aplikacije *Athena*, njezinog dizajna i arhitekture.

Dokument se sastoji od tri cjeline, gdje svaka od njih opisuje poseban dio aplikacije. U prvom dijelu, odnosno uvodu, se opisuje sami dokument tehničke dokumentacije i osnovne informacije o *Athena* aplikaciji. U drugom dijelu je naveden općeniti opis proizvoda te se navode funkcije koje će *Athena* sadržavati. U trećem dijelu se navode i opisuju zahtjevi vanjskih sučelja.

1.4. Obujam aplikacije

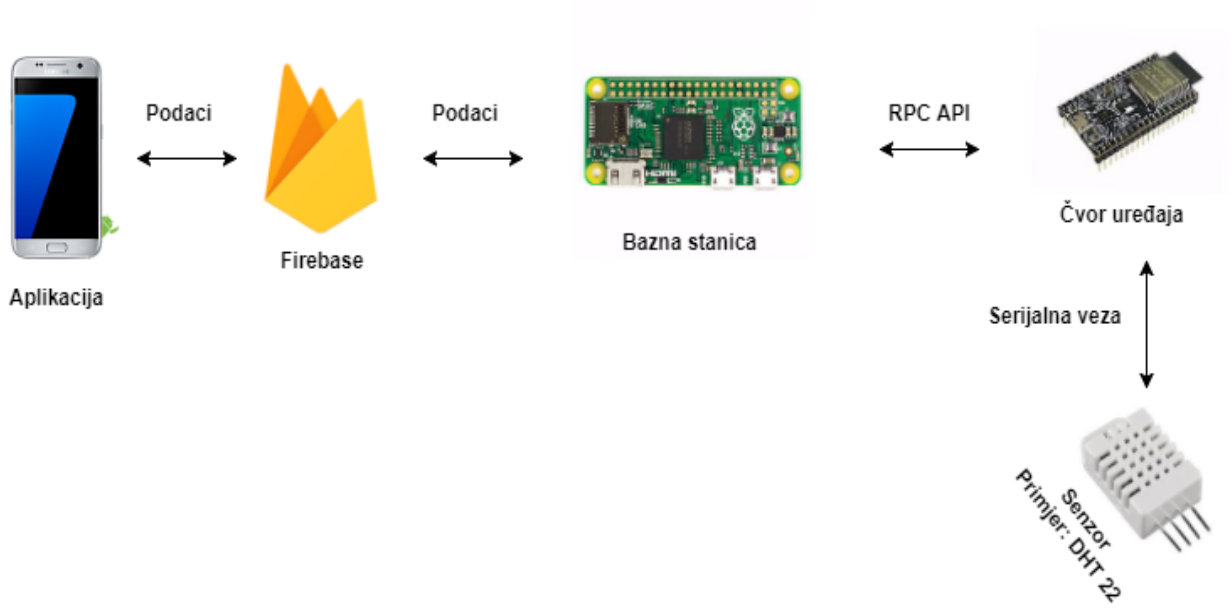
Athena je mobilna aplikacija namijenjena kontroliranju sustava za automatizaciju doma. Korisnici putem aplikacije imaju mogućnost dodavanja novih uređaja koje mogu kontrolirati, te pratiti njihovo stanje kroz statistiku. Uvjet za korištenje *Athene* je posjedovanje Facebook, Google ili Twitter računa preko koji se vrši prijava u sustav aplikacije. Osim mobilnog dijela, rad *Athene* se sastoji od raznog sklopovlja koje omogućava dohvaćanje informacija za obradu aplikaciji, te raznih pozadinskih servisa.

2. Općeniti opis

2.1. Perspektiva proizvoda

Pojavom Interneta omogućena je brza razmjena informacija s jednog mjesta. Razvojem tehnologije i sklopovlja omogućeno je spajanje raznih uređaja na Internet, te slanja informacija s uređaja na internet. Aplikacija *Athena* se koristi takvim mogućnostima, odnosno povezivanjem uređaja na internet i slanja informacija. Putem aplikacije je omogućena kontrola uređaja unutar doma i uvid u stanje doma.

U priloženoj slici je prikazan način na koji funkcionira aplikacija, odnosno put kroz koji informacije prolaze od kontroliranog uređaja do aplikacije.



Slika 1 - Pojednostavljeni dijagram sustava

2.2. Funkcije proizvoda

- Prijava korisnika
- Pregled uređaja
- Dodavanje i brisanje uređaja i bazne stanice
- Kontrola uređaja
- Pregled stanja doma
- Statistika

2.2.1 Tablica s pojmovima

Termin	Značenje	Primjer
Aplikacija	Aplikacija koja se izvodi na mobilnom uređaju, odnosno na Android operacijskom sustavu. Pomoću aplikacije korisnik koristi proizvod i njegove funkcionalnosti.	*Pomoću aplikacije korisnik se prijavljuje u sustav, s kojim mu se otvaraju navedene funkcije aplikacije*
Firebase	Platforma za razvoj mobilnih i web aplikacija, razvijena od strane Google-a. Koristi se za pohranu i dohvaćanje podataka, autentikaciju korisnika preko socijalnih mreža i sl.	*Prijava putem Facebook računa*
Bazna stanica	Ugradbeno računalo (Raspberry Pi Zero W) koje kontrolira uređaje, propagira podatke na njih te ih sinkronizira sa Firebase-om.	-
Čvor uređaja	ESP32 Mikrokontroler spojen s fizičke strane na uređaj koji kontroliramo preko senzora ili drugog sklopovskog sučelja.	-
Senzor	Preko senzora se dohvaćaju podaci koji se dalje šalju do aplikacije kojom se obavljaju navedene funkcije. Primjer senzora DHT 22	*Očitavanje temperature prostorije*

2.3. Korisničke klase i osobine

Aplikacija je namijenjena za obične korisnike, stoga nisu potrebna neka posebna predznanja, pa nam je dovoljno korisnike podijeliti u dvije klase:

Naziv klase	Plaćanje	Osobine
Obični korisnik	Bez plaćanja	Ova klasa ima prikaz svih osnovnih funkcionalnosti aplikacije, te joj se prikaz statistika senzora sačinjava od samo brojčanog načina.
Privilegiran korisnik	S plaćanjem	Ova klasa isto tako ima prikaz svih osnovnih funkcionalnosti aplikacije, ali joj prikaz statistike senzora sačinjava od brojčanog načina, a i isto tako i grafičkog načina.

2.4. Radno okruženje

Programski proizvod radi se u razvojnom okruženju Android Studio, u jezicima Java, XML i Gradle Script, operacijski sustav Android, koji sačinjava glavnu aplikaciju. Ona komunicira s bazom podataka. Za bazu podataka koristi se Firebase. Bazna stanica je Raspberry Pi Zero koja radi na operacijskom sustavu Linux. Na baznoj stanici se koristi Node JS. Ona komunicira s bazom podataka, a isto tako i sa čvorom uređaja (ESP8266). Čvor uređaja koristi operacijski sustav Mongoose OS. Na njemu se isto tako koristi Node JS.

U tablici su napisani svi elementi gore navedeni radi lakše preglednosti:

Naziv Dijela Sustava	Korištene Tehnologije
Glavna aplikacija	Razvojne tehnologije: Java, XML, Gradle Script Operativni sustav: Android
Bazna stanica	Razvojne tehnologije: Node JS Operativni sustav: Linux-based

Čvor uređaja (ESP8266)	Razvojne tehnologije: Node JS Operativni sustav: Mongoose OS

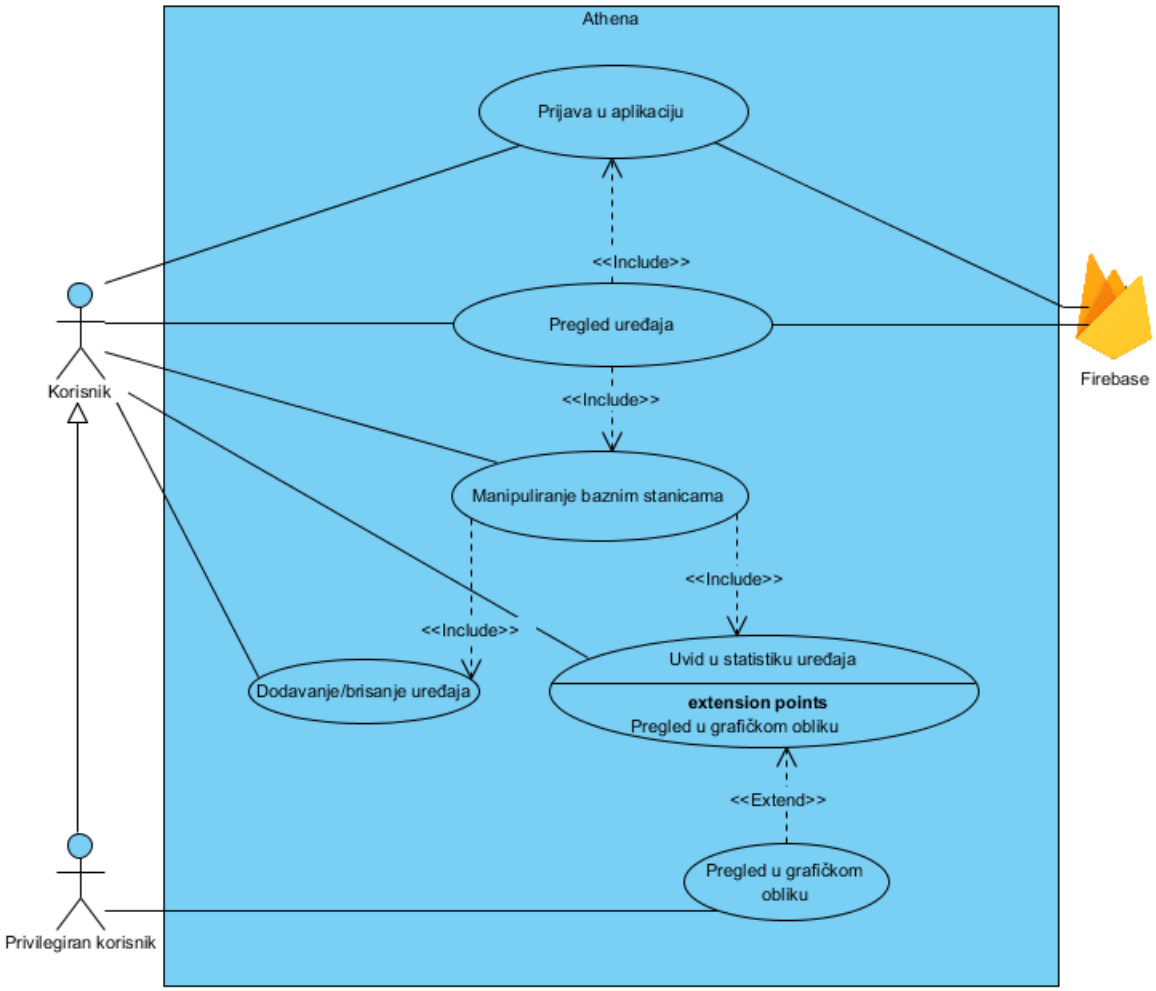
2.5. Ograničenja implementacije i dizajna

Rad sa ugradbenim sklopovljem uvijek predstavlja izazov zbog inherentnih ograničenja takvih uređaja, najviše u pogledu ograničenih specifikacija, poput male memorije ili brzine procesora/mikrokontrolera. Zbog ograničenja samih čvorova uređaja, određene dijelove implementacije, poput komunikacije sa Firebase servisom te pohranjivanje podataka moramo mitigirati na drugi uređaj, u ovom slučaju na baznu stanicu, Raspberry Pi. Ovakav način predstavlja nekoliko tehničkih problema, od sinkronizacije uređaja, prvotnog povezivanja te pouzdane komunikacije, koji ujedno predstavljaju najveće probleme prilikom same implementacije projekta. U ovom slučaju, vrijedi tvrdnja da je lanac jak samo koliko i njegova najslabija karika, te će se implementacija morati prilagoditi najnižoj razini sklopovlja, ugradbenim čvorovima uređaja te pomoću posrednika omogućiti njihovu komunikaciju te razmjenu podataka sa najvišom razinom sklopovlja, mobilnim uređajem i Firebase servisom.

Osim toga, sam dizajn arhitekture sadržava mnoge pogodnosti i fleksiblinost, od automatske sinkronizacije podataka između bazne stanice i mobilnog uređaja, do velikih dijelova već gotovih implementacija od strane samog Firebase-a.

3. Značajke sustava

3.1. Dijagram slučajeva korištenja



Slika 2 - Dijagram slučajeva korištenja Athena sustava

3.1.1. Slučajevi korištenja za korisnike aplikacije

Za početak korištenja aplikacije potrebno se prijaviti u aplikaciju koja se izvodi preko Firebase servisa. Nakon prijave, korisnik ima mogućnost pregledavanja uređaja koji sadržava funkcionalnost „Manipuliranje baznim stanicama“. Taj slučaj je od velike važnosti za aplikaciju jer njime dodajemo ili brišemo bazne stanice koji su centralni dio hardverskog dijela sustava. Na prethodno dodanu baznu stanicu korisnik može

dodavati ili brisati uređaje koje želi automatizirati. Posljednji slučaj korištenja je uvid u statistiku automatiziranih uređaja čiji je zadani prikaz u brojčanom obliku

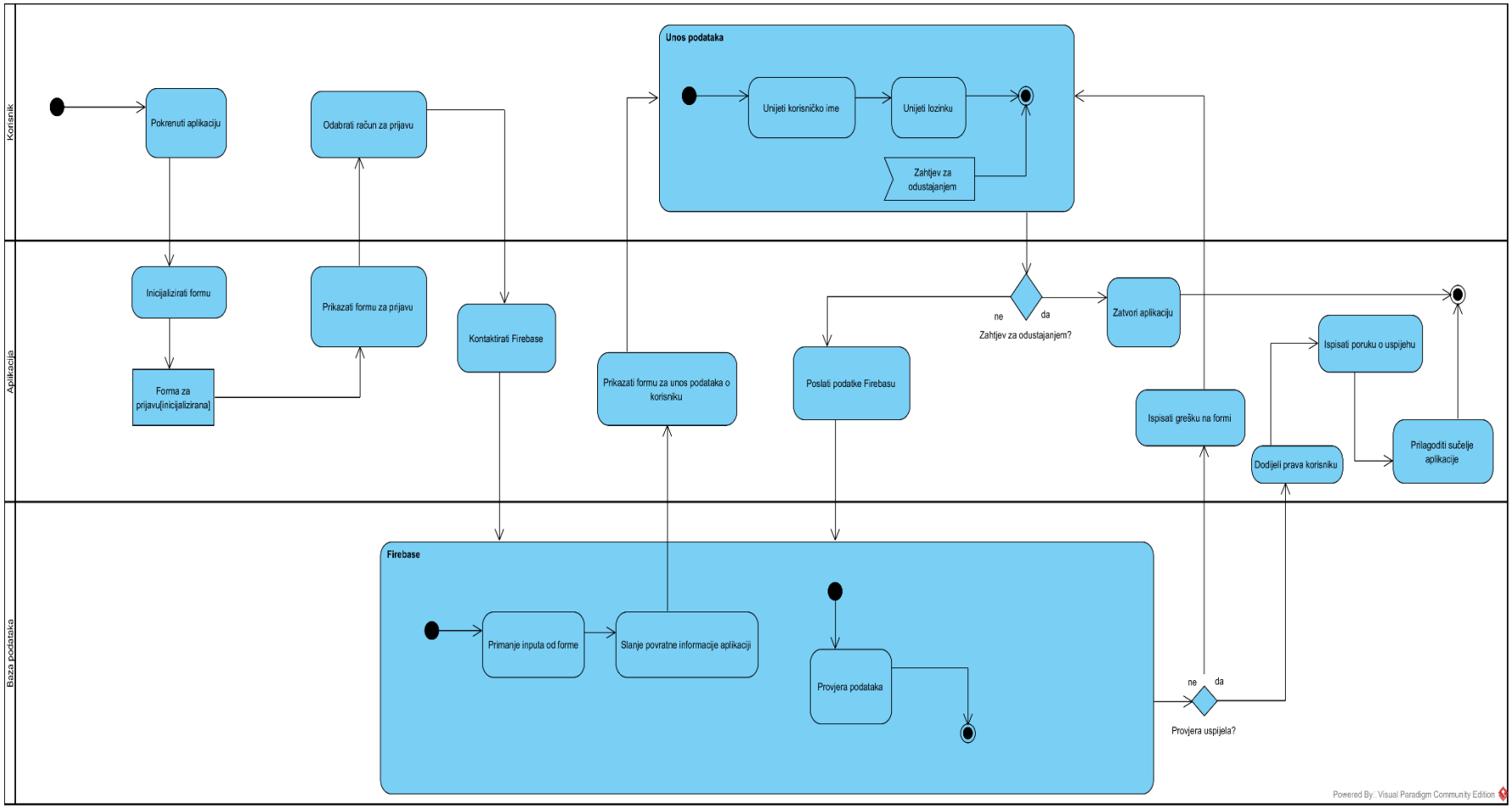
3.1.2. Slučaj korištenja za privilegirane korisnike aplikacije

Privilegirani plaćeni korisnici aplikacije uz sve prethodno navedene slučaje imaju napredniji prikaz statistike njihovih automatiziranih uređaja u obliku grafičkog prikaza.

3.2. Dijagram aktivnosti

3.2.1. Prijava u sustav

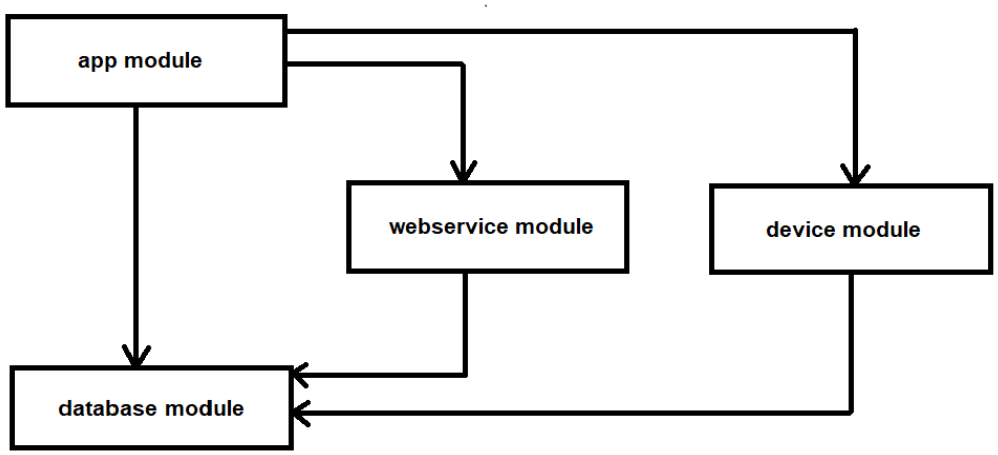
Na slici je prikazan dijagram aktivnosti prijave u sustav. Korisnik da bi se prijavio u sustav prvo mora pokrenuti samu aplikaciju, nakon čega mu se otvara forma za prijavu. Korisnik ima mogućnost biranja s kojim će se računom prijaviti, gdje se nakon odabira računa kontaktira Firebase. Kada Firebase vrati uspješnu povratnu informaciju aplikaciji ona prikazuje formu za unos podataka. Korisnik unosi svoje podatke (korisničko ime i lozinku), ali i može i odustati od prijave. Nakon što korisnik unese podatke aplikacija šalje podatke Firebase gdje on provjerava njihovu ispravnost. Ako provjera nije uspjela korisnik ima mogućnost ponovnog unosa podataka. Ako su podaci ispravni aplikacija dodjeluje prava korisniku, ispisuje poruku o uspjehu i otvara se početno sučelje aplikacije.



Slika 3– Dijagram aktivnosti login

3.3. Dijagram modula

Na slici je prikazan dijagram modula kojim je prikazana modularnost sustava Athena. Dijagramom je prikazana podjela i komunikacija među modulima te su tako prikazane i ovisnosti pojedinih modula o drugima.

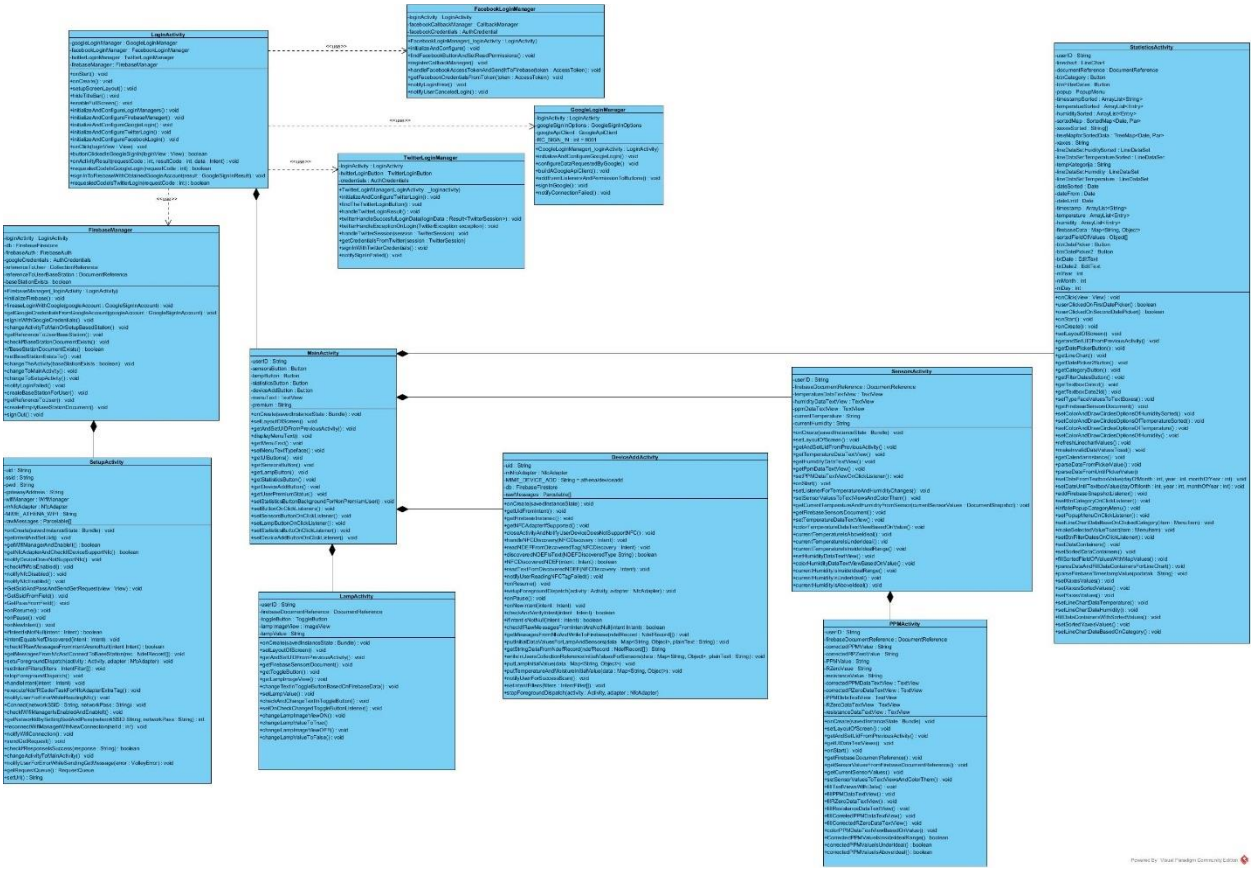


Slika 4 - Dijagram modula

Glavni modul je app modul koji komunicira sa svima i koji je svima nadređen, unutar njega će biti implementirana funkcionalnost sustava, tj. glavnina aplikacije. Drugi modul je webservice modul unutar kojega će biti implementirane prijave s raznim servisima koji će unutar ovog sustava biti (Google, Facebook, Twitter), korisnik će se nakon uspješne prijave zapisati u bazu te će se tako rukovati s korisnicima aplikacije. Device modul će biti modul unutar kojega će biti definirano rukovanje s uređajima koji će se nalaziti unutar sustava, te će dodatno uređaji komunicirati s bazom kako bi se ne bi svaki put ponovno morali očitovati da su se priključili nego da zapravo postoji zapis u bazi kako bi se ubrzalo prepoznavanje uređaja. Database modul biti će modul unutar kojega će biti implementirano rukovanje s Firebase bazom, te će se tu rukovati s podacima koji kruže unutar aplikacije te koji će se koristiti za razne akcije.

3.4 Dijagram klasa

Na slici je prikazan dijagram klasa sustava Athena. Dijagramom klasa pokazujemo ovisnost klasa sustava, te njihove attribute i metode. Zbog veličine slike dijagram je također dostupan online.



4. Prijava u sustav (Login)

Kao što je i ranije spomenuto prijava u samu aplikaciju odvijati će se preko Firebase-a, jedna od glavnih ideja i razloga korištenja prijave putem društvenih mreža je i taj da svaki korisnik koji uopće instalira aplikaciju koja se nudi mora minimalno imati napravljen Google račun. Vođeni tom informacijom odlučili smo se za ovakav način prijave, koji se oslanja na vanjske servise. Upravo ovako, riješeno je pitanje sigurnosti koje se sada oslanja na ostale servise od aplikacija koje će biti navedene u nastavku i koje se koriste za prijavu.

4.1. LoginActivity

LoginActivity je početna aktivnost, koja se otvara prilikom samog pokretanja aplikacije. Kako i ime kaže korisnik se na ovoj aktivnosti prijavljuje u aplikaciju. Korisnik ima mogućnost odabire račun s kojim se želi prijaviti, odnosno mogućnost prijave u sustav aplikacije putem Google, Facebook ili Twitter računa.

Prilikom kreiranja Login aktivnosti poziva se funkcija setupScreenLayout kojom se sakriva naslov aplikacije, te se omogućava aplikaciji zauzimanje površine cijelog zaslona. U kreiranju aktivnosti se po *default-u* postavlja vizualni izgled aktivnosti sa setContentView funckijom. Također, prilikom kreacije se izvršava funkcija initializeAndConfigureLoginManagers kojom se inicijalizira LoginManager koji će biti opisan u nastavku.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setupScreenLayout();
    Twitter.initialize(this);

    setContentView(R.layout.activity_login);
    initializeAndConfigureLoginManagers();
}
```

Metoda InitializeAndConfigureLoginManager kako samo ime kaže služi za inicijalizaciju i konfiguraciju LoginManagera, odnosno služi za inicijalizaciju managera za Firebase, Google, Twitter i Facebook. Manageri omogućavaju prijavu u sustav prilikom klika na gumb s računom ovisno o pritisnutom gumbu. Svaki od managera će biti opisan u nastavku.

```
private void initializeAndConfigureLoginManagers() {
    initializeAndConfigureFirebaseManager();
    initializeAndConfigureGoogleLogin();
    initializeAndConfigureTwitterLogin();
    initializeAndConfigureFacebookLogin();
}

private void initializeAndConfigureFirebaseManager() {
    firebaseManager = new FirebaseManager(this);
}
private void initializeAndConfigureGoogleLogin() {
    googleLoginManager = new GoogleLoginManager(this);
}
private void initializeAndConfigureTwitterLogin() {
    twitterLoginManager = new TwitterLoginManager(this);
}
private void initializeAndConfigureFacebookLogin() {
    facebookLoginManager = new FacebookLoginManager(this);
}
```

Sljedeći prikazani kod prikazuje prijavu u aplikaciju na temelju odabranog gumba. Klikom se šalje određeni requestCode preko kojega slijedi prijava u sustav.

```
private boolean buttonClickedIsGoogleSignIn(View loginView){
    return loginView.getId() == R.id.google_button;
}

@Override
public void onActivityResult(int requestCode, int resultCode, Intent data)
{
    super.onActivityResult(requestCode, resultCode, data);

    if (requestedCodeIsGoogleLogin(requestCode)){
        GoogleSignInResult googleSignInAttempt =
Auth.GoogleSignInApi.getSignInResultFromIntent(data);
        if (googleSignInAttempt.isSuccess())
            signInToFirebaseWithObtainedGoogleAccount(googleSignInAttempt);
    }

    else if(requestedCodeIsTwitterLogin(requestCode))

twitterLoginManager.twitterLoginButton.onActivityResult(requestCode,
resultCode, data);

    else

facebookLoginManager.facebookCallbackManager.onActivityResult(requestCode,
resultCode, data);
}

private boolean requestedCodeIsGoogleLogin(int requestCode) {
    return requestCode == GoogleLoginManager.RC_SIGN_IN;
}

private void signInToFirebaseWithObtainedGoogleAccount(GoogleSignInResult
result) {
    GoogleSignInAccount account = result.getSignInAccount();
    firebaseManager.firebaseLoginWithGoogle(account);
}

private boolean requestedCodeIsTwitterLogin(int requestCode) {
    return requestCode == TwitterAuthConfig.DEFAULT_AUTH_REQUEST_CODE;
}
```

4.1. TwitterLoginManager

Prilikom početka programiranja bilo koje vrste autentikacije s Firebase-om potrebno je unutar aplikacijskog dijela u datoteci gradle dodati ovisnost za Firebase autentikaciju, prikaz koda:

```
compile 'com.google.firebase:firebase-auth:11.4.2'
```

Da bi se aplikacija kao takva mogla uopće povezati te da bi radila autentikacija kako spada, potrebno je registrirati aplikaciju kao razvojnu unutar aplikacije Twitter te dohvatiti API ključ te API sigurni ključ, koji će se kasnije koristiti za samu autentikaciju. Kada se unutar grafičkog sučelja koje nudi Firebase omogući Twitter login obavljen je administrativni dio te se može prijeći na kodiranje sam logike.

Da bi se krenulo sa samim programiranjem potrebno je vizualno predočiti korisniku gumb koji bi služio za samu prijavu, i u sklopu toga se koristi „TwitterLoginButton“, koji se dodaje unutar izgleda same aktivnosti, prikaz koda:

```
<com.twitter.sdk.android.core.identity.TwitterLoginButton
    android:id="@+id/button_twitter_login"
    android:layout_width="213dp"
    android:layout_height="82dp"
    android:layout_gravity="center_horizontal"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintHorizontal_bias="0.502"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.449" />
```

Ovo je standardni način prikaza koji se koristi za dodavanje gumba, u svakom od zasebnih prijava koristi se sličan princip te će se zbog jednostavnosti prikazati samo ovaj gumb, naravno ovaj gumb nema sva svojstva koja su mu potrebna da bi se pozicionirao kao što je trenutno unutar aplikacije no također zbog preglednosti koda izbačen je i taj dio. U nastavku će biti opisane metode koje se koriste unutar ove klase. U konstruktoru klase prosljeđuje se LoginActivity, te se poziva funkcija initializeAndConfigureTwitterLogin kojim inicijaliziramo Twitter preko Login aktivnosti.

```
public TwitterLoginManager(LoginActivity _loginActivity){
    this.loginActivity = _loginActivity;
    initializeAndConfigureTwitterLogin();
}

private void initializeAndConfigureTwitterLogin() {
    Twitter.initialize(loginActivity);
    findTheTwitterLoginButton();
    handleTwitterLoginResult();
}
```

Kod inicijalizacije je potrebno hendlati rezultate prijave na Twitter. Unutar funkcije handleTwitterSession dešava se par aktivnosti, a one se izvršavaju nakon uspješne prijave pomoću Twitter-a, obavi se razmjena OAuth pristupnog tokena i OAuth tajne za kredencijale od Firebase-a, te se nakon toga obavi autentikacia s Firebase-om koristeći Firebase kredencijale, kod se može pronaći na stranicama Firebase-a kao

standardni kod za rukovanje s ovim dijelom razvoja aplikacije te ga iz tog razloga nećemo dodatno prikazivati.

```
private void handleTwitterLoginResult() {
    twitterLoginButton.setCallback(new Callback<TwitterSession>() {

        @Override
        public void success(Result<TwitterSession> loginData) {
            twitterHandleSuccesfulLoginData(loginData);

        }
        @Override
        public void failure(TwitterException exception) {
            twitterHandleExceptionOnLogin(exception);

        }
    });
}

private void twitterHandleSuccesfulLoginData(Result<TwitterSession>
loginData) {
    Log.d("TwitterDebug", "twitterLogin:success" + loginData);
    handleTwitterSession(loginData.data);
}

private void twitterHandleExceptionOnLogin(TwitterException exception) {
    Log.w("TwitterDebug", "twitterLogin:failure", exception);
}

private void handleTwitterSession(TwitterSession session) {
    getCredentialsFromTwitter(session);
    signInWithTwitterCredentials();
}

private void getCredentialsFromTwitter(TwitterSession session) {
    credentials = TwitterAuthProvider.getCredential(
            session.getAuthToken().token,
            session.getAuthToken().secret);
}

private void signInWithTwitterCredentials() {

loginActivity.firebaseManager.firebaseAuth.signInWithCredential(credentials
)
            .addOnCompleteListener(loginActivity, new
OnCompleteListener<AuthResult>() {
                @Override
                public void onComplete(@NonNull Task<AuthResult> task) {
                    if (task.isSuccessful()) {

loginActivity.firebaseManager.changeActivityToMainOrSetupBasedOnBaseStation
();
                    }
                    else
                        notifySignInFailed();
                }
            });
}

private void notifySignInFailed() {
    Toast.makeText(loginActivity, "Authentication failed.",
            Toast.LENGTH_SHORT).show();
}
```

4.2. FacebookLoginManager

Kao i za prijavu putem Twitter-a, tako je i za Facebook potrebno definirati standardni gumb koji će se koristiti za daljnje kodiranje te za postavljanje potrebnih mogućnosti. Prvo se dodaju potrebne ovisnosti za Facebook, te se sinkronizira sami projekt kako bi se sve potrebne postavke aktivirale. Nakon toga se standardno dohvaća gumb koji je namijenjen prijavi putem Facebooka, te se definiraju pristupna prava koja će imati Facebook, primjer koda:

```
private void findFacebookButtonAndSetReadPermissions() {
    LoginButton mFacebookSignInButton =
              loginActivity.findViewById(R.id.facebook_button);
    mFacebookSignInButton.setReadPermissions("email", "public_profile");
}
```

Nakon toga se registrira CallbackManager, te se uz pomoć njega rukuje s rezultatima prijave putem Facebook-a, prikaz koda:

```
private void registerCallbackManager(){
    LoginManager.getInstance().registerCallback(facebookCallbackManager,
            new FacebookCallback<LoginResult> () {

                @Override
                public void onSuccess(LoginResult loginResult) {
                    handleFacebookAccessTokenAndSendItToFirebase(loginResult
                              .getAccessToken());
                }

                @Override
                public void onCancel() {
                    notifyUserCanceledLogin();
                }
                @Override
                public void onError(FacebookException exception) {
                    notifyLoginError();
                }
            });
}
```

Postupak je sličan kao i kod prijave za Twitter uz razliku da se ovdje prilikom registriranja Callback-a uzima sama instanca koja je ranije kreirana, te se dodatno kreira novi FacebookCallback kako bi se definiralo što se radi kada se dogodi uspješna prijava, kada se prijava prekine te kada se pojavi greška unutar prijave, funkcije koje su ovdje implementirane su „onSuccess“, „onCancel“, „onError“. Unutar funkcije „onSuccess“ definira se rukovanje s pristupnim tokenom koji se koristi unutar autentikacije s Facebook-om, taj kod se u potpunosti može pronaći na stranicama koje nudi dokumentacija Firebase-a na stranici https://firebase.google.com/docs/auth/android/facebook-login. Rukovanje tokenom je napravljeno tako

da se dohvate kredencijali iz tokena, te se zatim na temelju njih preko firebaseManager iz loginActivity-a postavlja onCompleteListener koji provjerava je li radnja uspješna te na temelju uspješnosti preusmjeruje korisnika na glavnu aktivnost ili na setup aktivnost ovisno o tome ima li korisnik zapisanu baznu stanicu na firebase-u.

```
private void handleFacebookAccessTokenAndSendItToFirebase(AccessToken token) {
    getFacebookCredentialsFromToken(token);
    loginActivity.firebaseManager.firebaseAuth
        .signInWithCredential(facebookCredentials)
            .addOnCompleteListener(loginActivity,
                  new OnCompleteListener< AuthResult >() {
                @Override
                public void onComplete(@NonNull Task< AuthResult > task) {
                    if (task.isSuccessful())
                        loginActivity.firebaseManager
                                .changeActivityToMainOrSetupBasedOnBaseStation();
                    else
                        notifyLoginError();
                }
            });
}
```

Naravno unutar prikazane implementacije dodatno se može proširiti funkcionalnosti prijave raznim pozdravnim porukama i slično, no taj dio je ostavljen za kasnije kako bi se kasnije poklopio vizualni identitet same aplikacije sa raznim mogućnostima koje će grafičko sučelje same aplikacije nuditi korisniku.

4.3. GoogleLogin Manager

Prije samog početka rada s prijavom putem Googla, potrebno je dodati ovisnost za Google Sign-in unutar aplikacijske razine gradel-a, prikaz koda:

```
compile 'com.google.android.gms:play-services-auth:11.4.2'
```

Nakon dodavanja ovisnosti potrebno je sinkronizirati projekt kako bi sve radilo kako je i namijenjeno te je nakon toga potrebno specificirati SHA-1 otisak aplikacije unutar konzole Firebase-a, nakon toga se omogućuje prijava putem Googla unutar sučelja koje nudi Firebase. Nakon toga se unutar izgleda aktivnosti dodaje gumb koji će biti korišten za prijavu putem Googla, te se može početi s kodiranjem. Prvo što je potrebno učiniti je konfigurirati što će se prilikom prijave zahtijevati od korisnika i to se može učiniti koristeći metodu configureDataRequestedByGoogle, prikaz koda slijedi:

```
private void configureDataRequestedByGoogle() {
    googleSignInOptions = new GoogleSignInOptions.Builder(GoogleSignInOpti-
ons.DEFAULT_SIGN_IN)
            .requestIdToken(loginActivity.getString(R.string.client_id))
            .requestEmail()
            .build();
}
```

- DEFAULT_SIGN_IN – dohvaća osnovni profil i identifikator korisnika
- requestIdToken – Specificira da se zahtjeva ID token od autenticiranog korisnika
- requestEmail – Zahtjeva email kako bi se spremio unutar Firebase-a

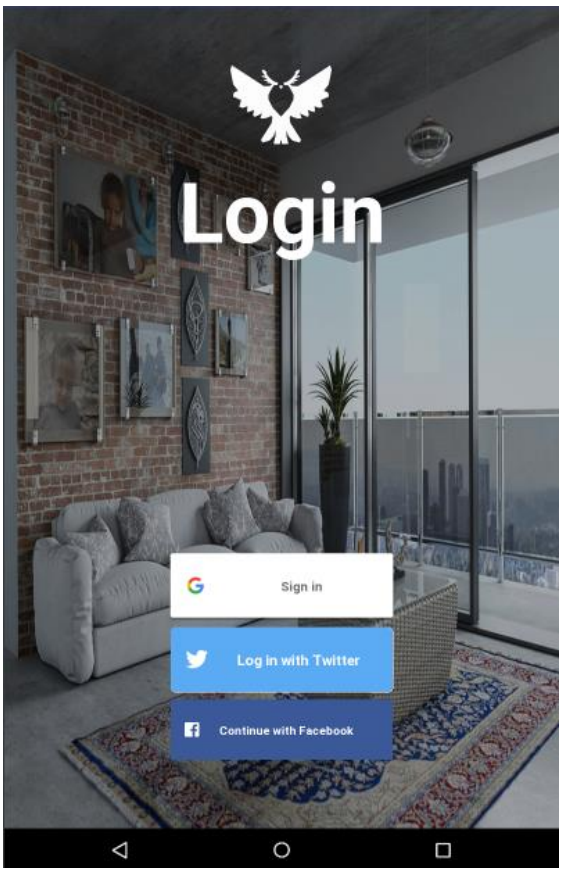
Nakon konfiguracije zahtjeva, potrebno je napraviti API klijent s pristupom do API-ja koji nudi Google prijava. EnableAutoManage omogućava automatsko upravljanje životnim ciklusom samog zahtjeva.

```
private void buildAGoogleApiClient(){
    googleApiClient = new GoogleApiClient.Builder(loginActivity)
            .enableAutoManage(loginActivity, new GoogleApiClient.OnConnec-
tionFailedListener() {
                @Override
                public void onConnectionFailed(@NonNull ConnectionResult
connectionResult) {
                    notifyConnectionFailed();}})
            .addApi(Auth.GOOGLE_SIGN_IN_API, googleSignInOptions)
            .build();
}
```

Funkcija „onConnectionFailed“ koristi se za automatsko upravljanje, te omogućuje rukovanje s greškom koja se desi prilikom neuspješnog spajanja. Korištenjem addApi specificira se koji su API-ji potrebni tj. koji API-ji se zahtijevaju.

Sama funkcija za prijavu koja se poziva prilikom pritiska za gumb za prijavu putem Google-a je prikazan kroz nekoliko sljedećih linija koda:

```
public void signInGoogle() {
    Intent signInIntent = Auth.GoogleSignInApi.getSignInIntent(googleApiC-
lient);
    loginActivity.startActivityForResult(signInIntent, RC_SIGN_IN);
}
```



Login aplikacije

5. Kreiranje početnog setup-a

Početni setup služi kao okosnica između korištenja aplikacije i prijave, tj. služi kao „most“ koji će omogućiti da korisnik poveže raspberry pi i mobitel kako bi se obavio prijenos podataka o mreži da se omogući spajanje na Internet te kako bi se poslao i univerzalni identifikator korisnika koji će kasnije omogućiti da se dohvaćaju i učitavaju podaci za prijavljenog korisnika. Jedan slučaj korištenja je kada je korisnik već registriran tj. kada ima registriranu baznu stanicu, njemu se nakon prijave i provjere omogući prijelaz na izbornik. Drugi slučaj korištenja je kada korisnik nema obavljen početni setup, tada mu se otvara nova aktivnost na kojoj može unijeti podatke o mreži koji će se poslati raspberry pi-u.

5.1. Prijava korisnika ovisno o tome ima li zapisanu baznu stanicu na firebase-u

Prilikom uspješne prijave prvo se dohvaća referenca na korisnika, te se dohvaća referenca na baznu stanicu. Nakon toga se provjerava postoji li document zapisa bazne stanice za dotičnog korisnika, te se ovisno o postojanju toga zapisa, korisnika proslijeđuje na Main activity ili na Setup activity. Reference na korisnika i na baznu stanicu se dohvaćaju iz firebase-a.

```
void changeActivityToMainOrSetupBasedOnBaseStation(){
    getReferenceToUser();
    getReferenceToUserBaseStation();
    checkIfBaseStationDocumentExists();
    changeTheActivity(ifBaseStationDocumentExists());
}
```

Slika 5. Promjena aktivnosti ovisno o zapisu bazne stanice

Nakon što se učita putanja unutar varijable potrebno je dodati slušatelj događaja koji će biti u stanju da prati svaki događaj koji se dogodi kada se uspješno izvrši ova akcija, kako je ova akcija osjetljiva na rezultat potrebno ju je osigurati s try catch blokovima kako si se spriječilo rušenje aplikacije u suprotnom, taj dio neće biti prikazan unutar same dokumentacije radi jednostavnosti. Ukoliko se ova akcija uspješno obavi potrebno je nadjačati metodu koja se pokreće kada se obavi sami zadatak. Nakon toga se provjerava i uspješnost izvršavanja zadatka te ukoliko je zadatak uspješno obavljen u varijablu se sprema rezultat samog zadatka. Ako se u toj varijabli nešto nalazi, drugim riječima ako sami dokument postoji radi se ispis u log koji omogućuje provjeru rada te nakon toga kreiranje Intenta koji će pokrenuti novu aktivnost. Nova aktivnost je izbornik koji je nakon toga povezan s korisnikom te spreman za korištenje. Funkcijom finish() radi tako da zatvori aktivnost koja je bila prije pokrenuta kako se korisnik ne bi mogao ponovno vratiti s back buttonom na aktivnost za prijavu, te kako bi se omogućio prividni backstack. Rad prvog dijela aktivnosti je prikazan sljedećim linijama koda:

```
private void checkIfBaseStationDocumentExists() {
    referenceToUserBaseStation.get()
        .addOnCompleteListener(new OnCompleteListener<DocumentSnapshot>() {
        @Override
        public void onComplete(@NonNull Task<DocumentSnapshot> firestoreQuery) {
            if (firestoreQuery.isSuccessful()) {
                try {
                    DocumentSnapshot baseStationDocument = firestoreQuery.getResult();
                    if (baseStationDocument != null)
                        setBaseStationExistsTo(true);
                    else
                        setBaseStationExistsTo(false);
                }
                catch (Exception e) {
                    setBaseStationExistsTo(false);
                }
            }
            else
                notifyLoginFailed();
        }
    });
}
```

Slika 6. Provjera ima li prijavljeni korisnik zapisanu baznu stanicu u firebase-u

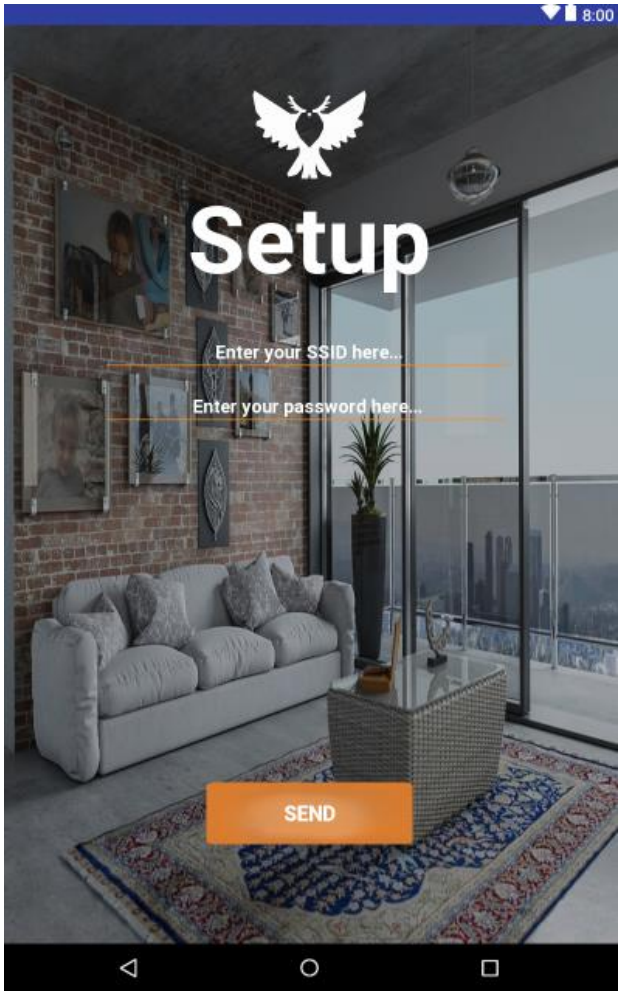
Nakon provjere zapisa bazne stanice, korisnik se preusmjeruje na MainActivity ili na SetupActivity ovisno o postojanju zapisa. Poziv nove aktivnosti je odrađen tako da se dohvaća ID korisnika, stvara se novi Intent kao nova aktivnost. U Intent se postavlja korisnikov ID kao extra, te se pokreće nova aktivnost i završava se Login aktivnost.

```
private void changeTheActivity(boolean baseStationExists) {
    if(baseStationExists)
        changeToMainActivity();
    else
        changeToSetupActivity();
}

private void changeToMainActivity() {
    String userUID = firebaseAuth.getUid();
    Intent intent = new Intent(loginActivity, MainActivity.class);
    intent.putExtra("userUID", userUID);
    loginActivity.startActivity(intent);
    loginActivity.finish();
}

private void changeToSetupActivity() {
    String userString = firebaseAuth.getUid();
    Intent intent = new Intent(loginActivity, SetupActivity.class);
    intent.putExtra("user",userString);
    loginActivity.startActivity(intent);
    loginActivity.finish();
}
```

Slika 7. Prijava u sustav te pozivanje sljedeće aktivnosti

5.2. Registriranje bazne stanice pod dotičnog korisnika u firebase

Kada se korisnik prijavi u sustav ili registrira po prvi put te kada mu se kroz prethodno navedene linije provjeri dostupnost bazne stanice, te kada se uvidi da korisnik nema baznu stanicu kreće se s pokretanjem aktivnosti za setup koji je potrebno odraditi kako bi se baznoj stanici omogućilo povezivanje na Internet, te kako bi ona mogla kreirati svoj hotspot koji će omogućiti senzorima da se spoje na njega te da tako komuniciraju s baznom stanicom. Korisnik mora unijeti podatke vezane za mrežu na koju se očekuje da će bazna stanica moći uvijek pristupiti . Prikaz aktivnosti koja se aktivira kad se pojavi ovaj zahtjev:



Slika 8. Prikaz aktivnosti setup-a

Nakon što se prikaže aktivnost koja je potrebna korisnik unosi u dostupna polja SSID i lozinku svojeg kućnog wifi-a, te skenira NFC tag na baznoj stanici, kako bi mu se aplikacija preko wifi-ja spojila na baznu stanicu. Nakon spajanja, korisnik pritiskom na gumb SEND šalje GET zahtjev baznoj stanici koji u

parametrima sadrži SSID i lozinku kućnog wifi-ja, kako bi se bazna stanica mogla povezati na internet, te tako pristupiti firebase-u. Slanje GET zahtjeva je detaljnije objašnjeno u poglavlju 10.

6. Izbornik - MainActivity

Nakon uspješne prijave u sustav korisniku se otvara glavni izbornik gdje ima tri opcije. To su pregledavanja senzora, upravljanja lampom te uvida u statistiku. Na samom kreiranju aktivnosti ona mora preuzeti Uid koji mu je proslijedila prethodna aktivnost. Da bi forma primila taj podatak mora prilikom kreiranja aktivnosti pozvati getIntent() funkciju te iz tog Intenta dohvatiti poslani String sa getStringExtra funkcijom. Taj Uid se ne koristi na ovoj aktivnosti ali je potreban za sve ostale aktivnosti kojima se može pristupiti putem glavnog izbornika koji će im slati taj podatak. Kod dohvaćanja korisnikovog Uid-a:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setLayoutOfScreen();
    getAndSetUIDFromPreviousActivity();

    displayMenuText();
    getUIButtons();
    setButtonOnClickListeners();
    getUserPremiumStatus();
}
```

Da bi forma primila taj podatak mora prilikom kreiranja aktivnosti pozvati getIntent() funkciju te iz tog Intenta dohvatiti poslani String sa getStringExtra funkcijom. Taj Uid se ne koristi na ovoj aktivnosti ali je potreban za sve ostale aktivnosti kojima se može pristupiti putem glavnog izbornika koji će im slati taj podatak. Kod dohvaćanja korisnikovog Uid-a:

```
private void getAndSetUIDFromPreviousActivity() {
    Intent intent = getIntent();
    userID = intent.getStringExtra("userUID");
}
```

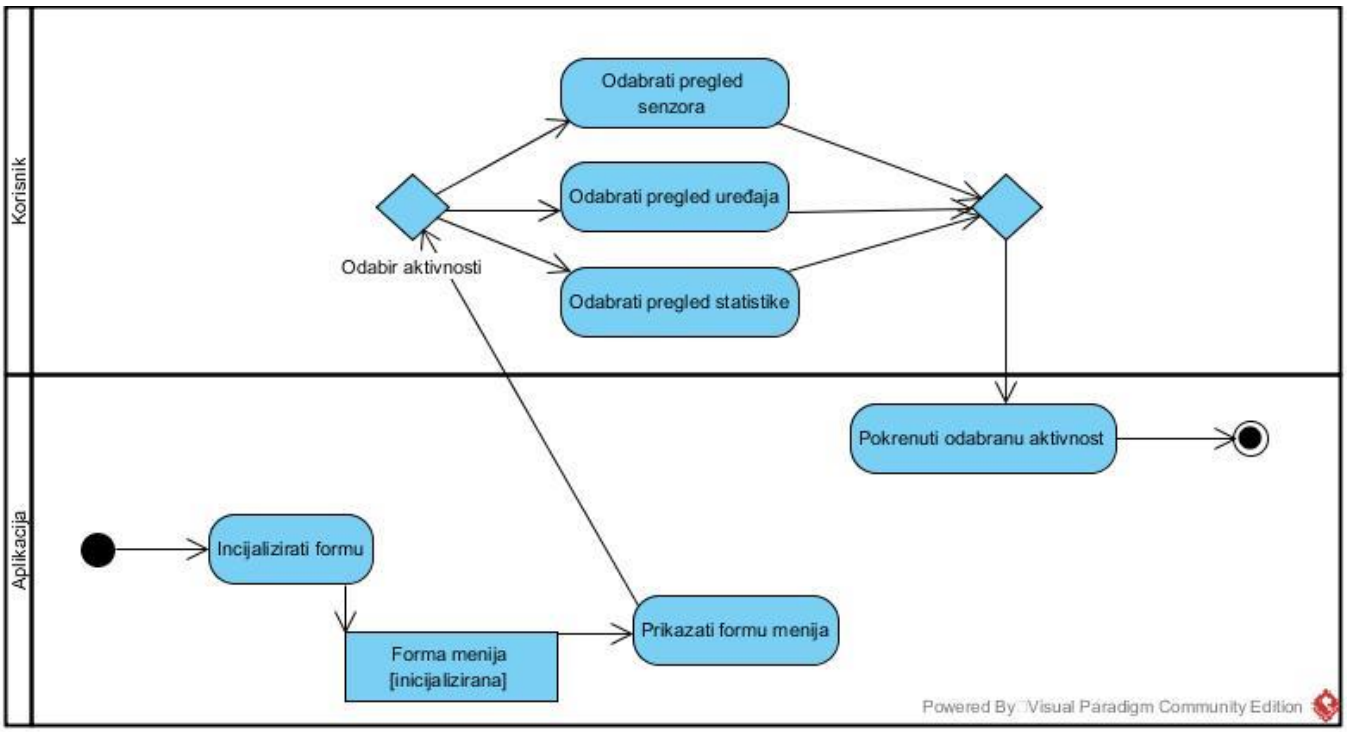
Izbornik je realiziran preko tri gumba preko kojih se pristupa drugim aktivnostima. Primjer XML koda gumba:

```
<Button
    android:id="@+id/lampButton"
    android:layout_width="@dimen/_100sdp"
    android:layout_height="@dimen/_90sdp"
    android:layout_alignParentEnd="true"
    android:layout_alignTop="@+id/sensorsButton"
    android:layout_marginEnd="39dp"
    android:layout_marginTop="80dp"
    android:background="@drawable/button_lamp"
    tools:layout_editor_absoluteX="86dp"
    tools:layout_editor_absoluteY="333dp" />
```

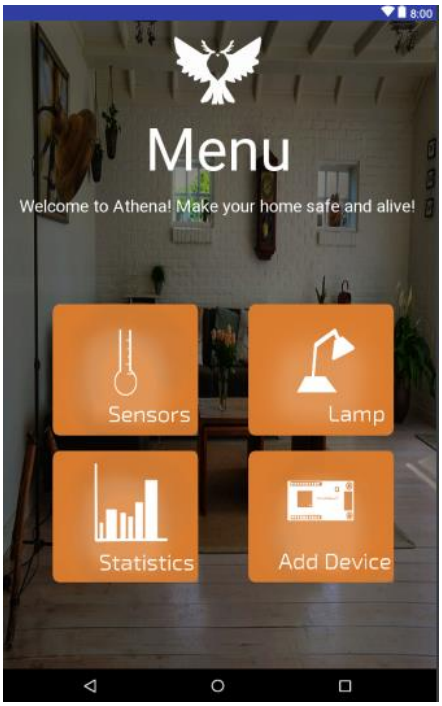
Bitna metoda u ovoj klasi je getUserPremiumStatus koja dohvaća korisnika i na osnovu dohvaćenih podataka zaslon se prilagođava statusu korisnika. Konkretno, premium korinsik ima mogućnost uvida u statistiku dok običnog korinsika gumb prebaci na Google Store.

```
private void getUserPremiumStatus() {

    DocumentReference documentReference = FirebaseFirestore.getIns-
tance().collection("users").document("stattest");

    documentReference.get().addOnCompleteListener(new OnCompleteListener<Do-
cumentSnapshot>() {
        @Override
        public void onComplete(@NonNull Task<DocumentSnapshot> task) {

            if(task.isSuccessful()) {

                DocumentSnapshot documentSnapshot = task.getResult();
                premium= documentSnapshot.getString("premium");

                setStatisticsButtonBackgroundForNonPremiumUser();
            }
        }
    });
}
```

Na slici je prikazan dijagram aktivnosti glavnog menija. Do glavnog menija dolazi se nakon uspješne prijave u aplikaciju s korisničkim računom Google-a, Facebook-a ili Twittera. Aktivnost inicijalizira formu na kojoj se iscrtavaju 3 gumba. Pomoću ta 3 gumba biramo funkcionalnosti aplikacije. Ispis podataka o senzorima, ispis stanja lampe te ispis statistike. Klikom na gumb pokrećemo zadanu aktivnost.



Slika 9 - Dijagram aktivnosti glavnog izbornik

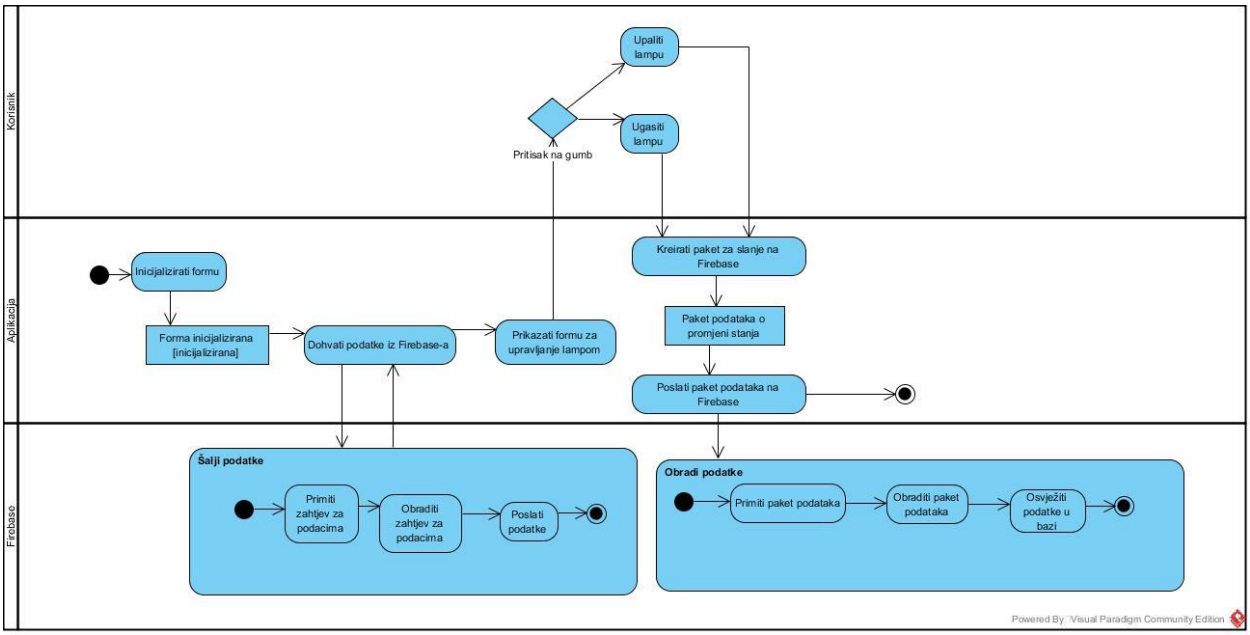


Slika 10 - UI glavnog izbornika

7. Upravljanje lampom

Klikom na gumb *Lamp* na glavnom izborniku otvara se aktivnost na kojoj se prikazuje stanje uređaja lampe ukoliko ona postoji. Prilikom kreiranja aktivnosti ona povlači podatak s Firebase-a o stanju lampe. Kao i u slučaju glavnog izbornika, da bi prikazali podatak o stanju lampe za određenog korisnika, potrebno je preuzeti Intent i iz njega pročitat Uid na temelju kojeg znamo s kojim korisnikom Firebase mora komunicirati. Jednostavno uključivanje/isključivanje lampe izvedeno je pomoću listenera dodanog na „toggle“ gumb.

```
private void setOnCheckChangedToggleButtonListener() {
    toggleButton.setOnCheckedChangeListener(new CompoundButton.OnCheckedChan-
geListener() {
        @Override
        public void onCheckedChanged(CompoundButton buttonView, boolean but-
tonIsChecked) {
            if(buttonIsChecked){
                changeLampImageViewON();
                changeLampValueToTrue();
            }
            else{
                changeLampImageViewOFF();
                changeLampValueToFalse();
            }
        }
    });
}
```



Slika 11 - Dijagram aktivnosti upravljanja lampom

Aktivnost započinje inicijalizacijom forme nakon koje aplikacija dohvaća podatke o stanju lampe s Firebase servisa, te ih prikazuje korisniku. Korisnik pritiskom na „toggle“ gumb može po potrebi uključivati i isključivati lampu. Svakim klikom na gumb aplikacija šalje paket podataka Firebase servisu koji obrađuje podatak i osvježava stanje.

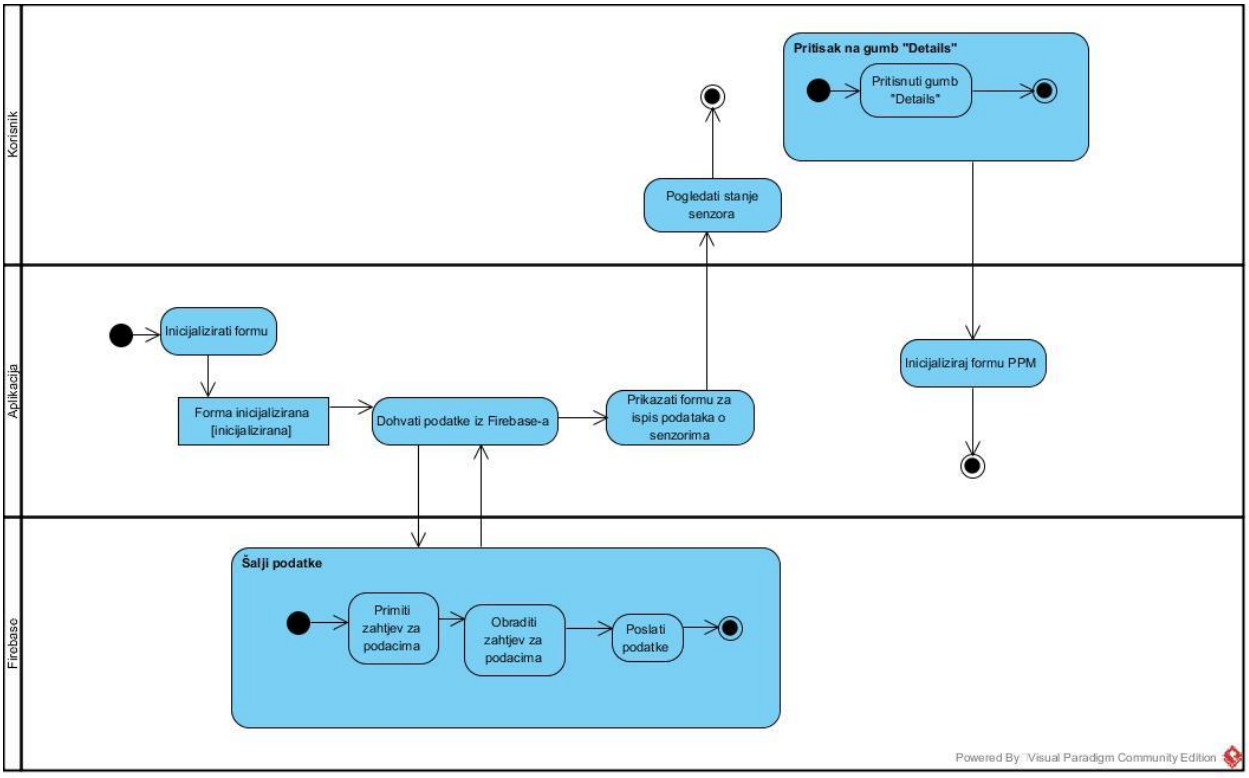
Slika 12 - UI upravljanja lampom

8. Prikaz podataka dohvaćenih senzorom

Klikom na gumb *Sensors* na glavnom izborniku otvara se aktivnost na kojoj se prikazuju podaci spremljeni unutar Firebase-a. Konkretno, to su podaci koje dohvaćaju senzori kao što su temperatura (eng. *temperature*), vlažnost (eng. *humidity*) i kvaliteta zraka (*PPM*). Kao i u glavnom izborniku potrebno je prilikom kreiranja aktivnosti preuzeti Intent i podatak Uid-a koji prosljeđuje izbornik aktivnosti senzora. Nakon toga potrebno je dohvatiti podatke iz Firebase-a. Podaci se dohvaćaju u stvarnom vremenu, tako se izmjenom podatka unutar baze automatski mijenja i na formi aplikacije. Kod dohvaćanja podataka iz Firebase-a:

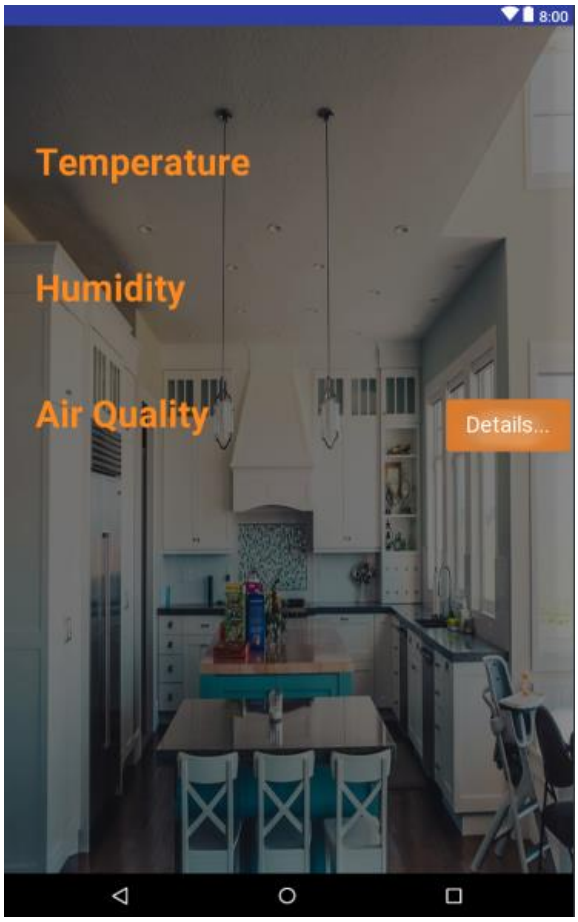
```
private void getFirebaseSensorsDocument() {
    firebaseDocumentReference = FirebaseFirestore.getInstance().collec-
tion("users").document("test123").collection("Uredaji").document("senzor");
}
```

Nakon toga se podaci vrednuju odnosno uspoređuju se s određenim vrijednostima da bi se podaci ispisali u određenoj boji. Parametar PPM se ne ispisuje na ovoj formi, jer on sadrži dodatne podatke koji se ispisuje na drugoj formi. Klikom na *Details* prelazimo na drugu formu gdje se prikazuju podaci PPM-a.



Slika 13 - Dijagram aktivnosti prikazivanja podataka senzora

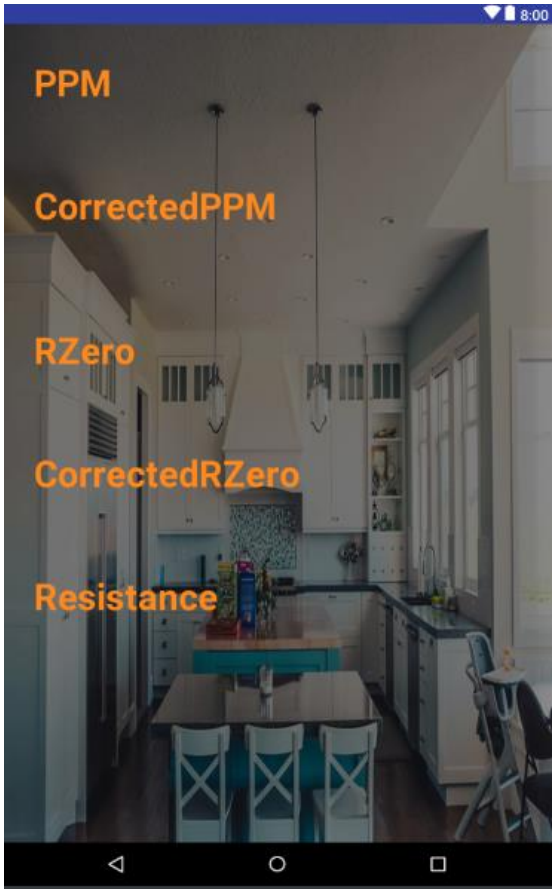
Aktivnost započinje inicijalizacijom forme nakon koje aplikacija dohvaća stanje senzora s Firebase servisa. Dohvaćene podatke ispisuje na formu te ih prikazuje korisniku koji dobiva uvid u stanja senzora. Korisnik može pritisnuti na tipku „Details“ koja inicijalizira novu formu za prikaz PPM-a.



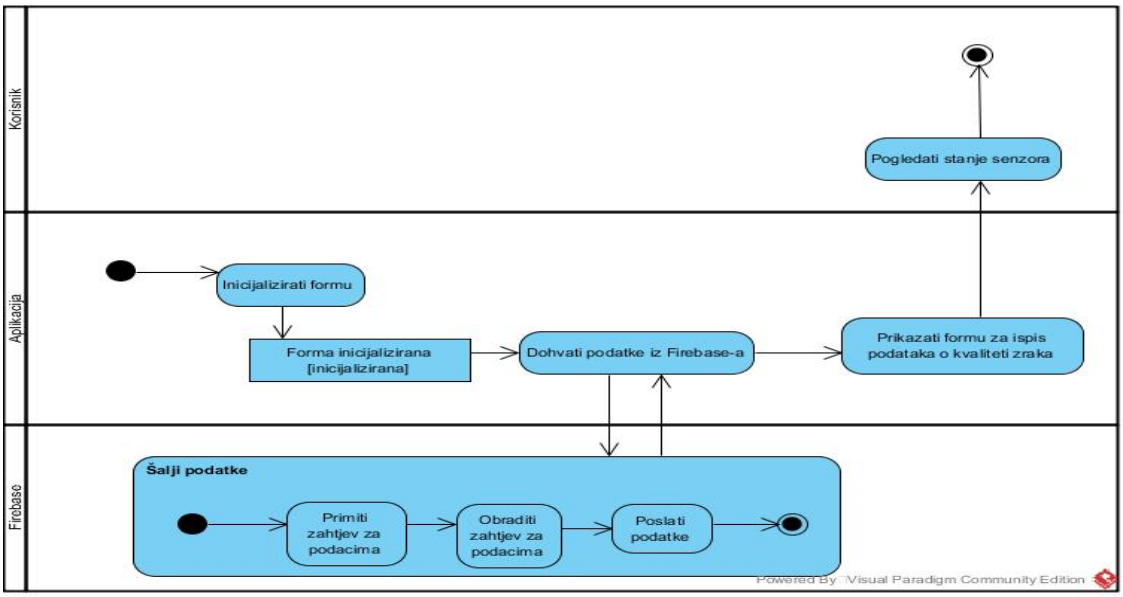
Slika 14 - UI prikaza podataka senzora

9. PPMActivity

PPMActivity je aktivnost koja prikazuje senzorom prikupljene podatke o zraku. Prikazuju se podaci PPM-a, CorrectedPPM-a, RZero, CorrectedRZero i Resistane. Aktivnost funckionira na isti način kao i SensorsActivity pa da bi izbjegli redundaciju nikakav detaljan kod neće biti prikazan za ovu aktivnost.



Slika 15 - UI prikaza podataka zraka



Slika 16 – Dijagram aktivnosti PPMActivity

Aktivnost započinje inicijalizacijom forme nakon koje aplikacija dohvaća stanje senzora s Firebase servisa. Dohvaćene podatke ispisuje na formu te ih prikazuje korisniku koji dobiva uvid u stanje kvalitete zraka.

10. StatisticsActivity

Klikom na gumb *Statistics* u glavnom izborniku otvara se aktivnost *StatisticsActivity* na kojoj se prikazuje niz vrijednosti određenih senzora u određenom vremenskom periodu. Pokretanjem aktivnosti povlače se podaci s Firebase servisa i sortiraju se u setove podataka temeljenih na kategoriji senzora.

```
public void addFirebaseSnapshotListener(){
    documentReference.addSnapshotListener(this, new EventListener<DocumentSnapshot>()
{
        @Override
        public void onEvent(DocumentSnapshot documentSnapshot,
FirebaseFirestoreException e) {
            if (documentSnapshot.exists()) {

                firebaseData  = documentSnapshot.getData();
                setDataContainers();
                fillSortedFieldOfValuesWithMapValues();
                parseDataAndFillDataContainersForLineChart();
                getLineChart();
                setXaxesVales();
                setYaxesValues();
                setColorAndDrawCirclesOptionsOfTemperature();
                setColorAndDrawCirclesOptionsOfHumidity();
                setLineChartDataTemperature();
                getCategoryButton();
                setBtnCategoryOnClickListener();
                getFilterDatesButton();
                setBtnFilterDatesOnClickListener();


            }
        }
    });
}
```

Na temelju odabira određene kategorije u padajućem izborniku koji se pojavljuje na zaslonu pritskom na gumb *Category,* u *LineChart* se učitava određeni set podataka i postavljaju se opcije iscrtavanja.

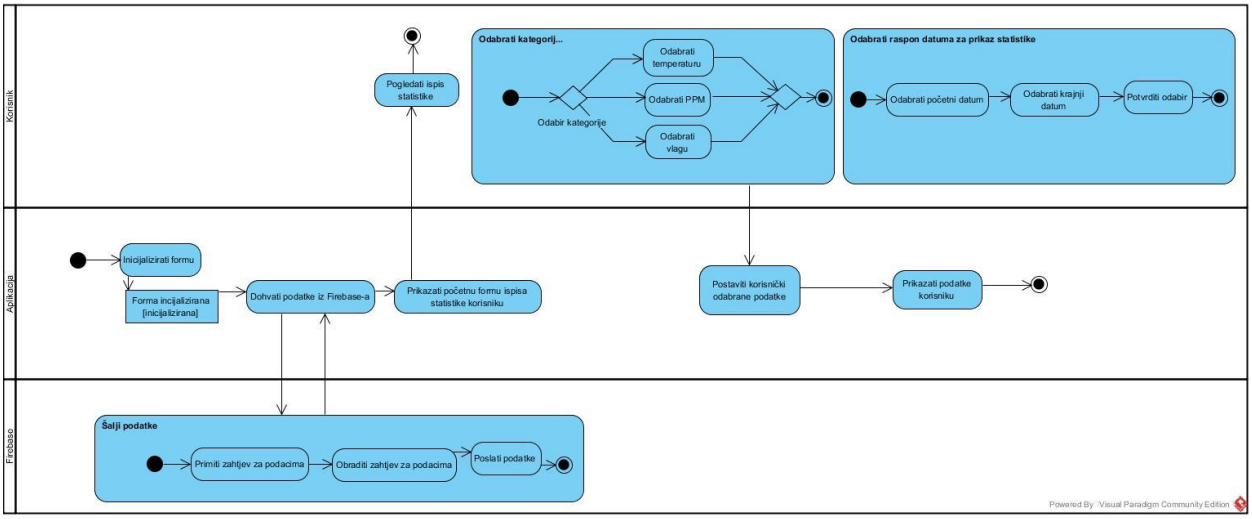
```
public void setPopupMenuOnClickListener() {

    popup.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
        public boolean onMenuItemClick(MenuItem item) {

            setLineChartDataBasedOnClickedCategory(item);
            makeSelectedValueToast(item);

            return true;
        }
    });
}

private void setLineChartDataBasedOnClickedCategory(MenuItem item) {
    switch(item.getTitle().toString()) {
        case "Humidity":
            setLineChartDataHumidity();
            refreshLinechartValues();
            tempKategorija = "Humidity";
            break;

        case "Temperature":
            setLineChartDataTemperature();
            refreshLinechartValues();
            tempKategorija = "Temperature";

        default: break;

    }
}
```

Prilikom odabira početnog i krajnjeg datuma perioda podataka koje prikazujemo, prethodno dohvaćeni podaci se sortiraju uz pomoć *SortedMap-e* te se učitavaju i zatim iscrtavaju u *LineChart* objektu.

```
private void setBtnFilterDatesOnClickListener() {

    btnFilterDates.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {

            if(dateFrom.before(dateUntil))

            {

                setSortedDataContainers();
                fillDataContainersWithSortedValues();
                setXaxesSortedValues();
                setSortedYaxesValues();
                setColorAndDrawCirclesOptionsOfHumiditySorted();
                setColorAndDrawCirclesOptionsOfTemperatureSorted();
                setLineChartDataBasedOnCategory();

            }

            else {
                makeInvalidDateValuesToast();
            }

        }
    });
}
```

```
private void fillDataContainersWithSortedValues() {
    int iterate = 0;
    for (SortedMap.Entry<Date, Par> red : sortedMap.entrySet()) {

        timestampSorted.add(red.getKey().toString());
        temperatureSorted.add(new
Entry(Float.parseFloat(red.getValue().getTemperatura()), iterate));
        humiditySorted.add(new Entry(Float.parseFloat(red.getValue().getVlaga()),
iterate));
        iterate++;

    }
}
```

Slika 17 – UI prikaz Statistike



Slika 18 – Dijagam aktivnosti StatusActivity

Aktivnost započinje inicijalizacijom forme, te zatim aplikacija dohvaća podatke senzora s Firebase sustava. Aplikacija korisniku prikazuje zadane podatke temperature. Korisnik može odabrati ispis jedne od zadanih kategorija senzora. Također korisnik može definirati vremenski period za koji želi ispis stanja senzora.

11. Čitanje NFC tagova

Za čitanje NFC tagova potrebna nam je klasa NdefReaderTask koja proširuje AsyncTask kako bi čitanje NFC tagova bilo asinkrono i odvijalo se u pozadini. Overridana je metoda doInBackground koja tijekom skeniranja NFC taga, dobiva taj tag kao parametar, te iz njega dobiva NDEF zapis, te vraća tekst tako da izvrši metodu readTextNDEFRecordsFromTagData.

```
@Override
protected String doInBackground(Tag... scannedNFCTagContents) {
    Ndef NFCTagData = Ndef.get(scannedNFCTagContents[0]);
    if (NFCTagDoesNotSupportNDEF(NFCTagData)){
        return null;
    }
    return readTextNDEFRecordsFromTagData(NFCTagData);
}
```

Metoda readTextNDEFRecordsFromTagData dobiva NDEF zapise it NFC taga, tako da dohvati cache-anu NDEF poruku, te čita tekstualne zapise iz NDEF zapisa, tako da prolazi po njima, te ako su zapisani s dobrim encoding-om, čita tekst, te ga dekodira.

```
private String readTextNDEFRecordsFromTagData(Ndef NFCTagData) {
    getNDEFRecordsFromTagData(NFCTagData);
    return readTextRecordsFromNDEFRecords();
}

private void getNDEFRecordsFromTagData(Ndef nfcTagRecord) {
    NdefMessage ndefMessage = nfcTagRecord.getCachedNdefMessage();
    NDEFRecords = ndefMessage.getRecords();
}

private String readTextRecordsFromNDEFRecords() {
    for (NdefRecord NDEFRecord : NDEFRecords) {
        if(NDEFRecordContainsText(NDEFRecord)){
            try {
                return readNDEFRecordText(NDEFRecord);
            } catch (UnsupportedEncodingException e) {
                notifyNFCReadError();
            }
        }
    }
    return null;
}
```

Metoda za čitanje NDEF zapisa dobiva byte array tako da dohvati payload od NDEF zapisa, te ga zatim dekodira tako da dohvati textEncoding iz njega, te jezični kod, koji zatim služe da bi se payload dekodirao prema zadanome jeziku.

```
private String readNDEFRecordText(NdefRecord NDEFRecord) throws
        UnsupportedEncodingException {
    byte[] NDEFRecordPayload = NDEFRecord.getPayload();
    return decodeNDEFRecordPayloadToString(NDEFRecordPayload);
}

private String decodeNDEFRecordPayloadToString(byte[] NDEFRecordPayload) throws
        UnsupportedEncodingException {
    getTextEncodingFromRecordPayload(NDEFRecordPayload[0]);
    getLanguageCodeLengthFromRecordPayload(NDEFRecordPayload[0]);
    return decodeStringUsingTheSpecifiedCharset(NDEFRecordPayload);
}

private void getTextEncodingFromRecordPayload(byte NDEFRecordPayloadFirstByte) {
    NDEFRecordTextEncoding =
              ((NDEFRecordPayloadFirstByte & 128) == 0) ? "UTF-8" : "UTF-16";
}

private void getLanguageCodeLengthFromRecordPayload(byte NDEFRecordPayloadFirstByte) {
    NDEFRecordLanguageCodeLength =  NDEFRecordPayloadFirstByte & 0063;
}

private String decodeStringUsingTheSpecifiedCharset(byte[] NDEFRecordPayload) throws
       UnsupportedEncodingException {
    int languageCodeAndFirstSymbolOffset = NDEFRecordLanguageCodeLength + 1;
    int stringLengthWithoutHeaderSymbols = NDEFRecordPayload.length -
                languageCodeAndFirstSymbolOffset;
    return new String(NDEFRecordPayload, languageCodeAndFirstSymbolOffset,
                stringLengthWithoutHeaderSymbols, NDEFRecordTextEncoding);
}
```

U aktivnosti koja koristi čitanje NFC tag-ova unutar funkcije OnCreate pozivamo metodu koja pokušava dohvatiti NFC adapter, te provjerava ako je isti NULL, u tome slučaju uređaj na kojem se izvodi aplikacija ne podržava NFC, pa stoga treba završiti aplikaciju.

```
private void getNFCAdapterIfSupported() {
    mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
    if (mNfcAdapter == null)
        closeActivityAndNotifyUserDeviceDoesNotSupportNFC();
}
```

Nakon toga pokrećemo metodu koja obavlja upravljanje Intentom. Metoda uzima intent, zatim provjerava je li intent jednak ACTION_NDEF_DISCOVERED. Nakon toga dohvaćamo tip intenta, te provjeravamo je li tip jednak MIME_DEVICE_ADD, u slučaju da je, čita se tekst it NDEF zapisa, a ako nije, obavještava se korisnik o neuspjelom čitanju. Čitanje teksta se odvija tako da dohvatimo NFC tag, te kreiramo novi NFCTagTextReader, te njemu proslijeđujemo dohvaćen tag.

```
private void handleNFCDiscovery(Intent NFCDiscovery) {
    if (NFCDiscoveredNDEF(NFCDiscovery)) {
        readNDEFFromDiscoveredTag(NFCDiscovery);
    }
}

private void readNDEFFromDiscoveredTag(Intent NFCDiscovery) {
    String NDEFDiscoveredType = NFCDiscovery.getType();
    if (discoveredNDEFIsText(NDEFDiscoveredType))
        readTextFromDiscoveredNDEF(NFCDiscovery);
    else
        notifyUserReadingNFCTagFailed();
}

private boolean discoveredNDEFIsText(String NDEFDiscoveredType) {
    return MIME_DEVICE_ADD.equals(NDEFDiscoveredType);
}

private boolean NFCDiscoveredNDEF(Intent intent) {
    return NfcAdapter.ACTION_NDEF_DISCOVERED.equals(intent.getAction());
}

private void readTextFromDiscoveredNDEF(Intent NFCDiscovery) {
    Tag DiscoveredNFCTag = NFCDiscovery.getParcelableExtra(NfcAdapter.EXTRA_TAG);
    new NFCTagTextReader().execute(DiscoveredNFCTag);
}
```

Nakon toga overridamo onNewIntent te prilikom prihvata novog intenta provjeravamo i verificiramo intent. Nakon čega provjeravamo jesu li raw poruke iz intenta različite null. Te nakon toga iz raw messages dohvaćamo NDEF poruke iz kojih dobivamo NDEF zapise i šaljemo ih metodi za zapisivanje na firebase.

```
@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    if (checkAndVerifyIntent(intent)) {
        if (checkIfRawMessagesFromIntentAreNotNull(intent)) {
            NdefMessage[] messages = new NdefMessage[rawMessages.length];
            for (int i = 0; i < rawMessages.length; i++) {
                messages[i] = (NdefMessage) rawMessages[i];
                NdefRecord[] ndefRecord = messages[i].getRecords();
                getMessagesFromNfcAndWriteToFirebase(ndefRecord);
            }
        }
    }
}

private boolean checkAndVerifyIntent(Intent intent) {
    return ifIntentIsNotNull(intent) && NFCDiscoveredNDEF(intent);
}

private boolean ifIntentIsNotNull(Intent intent) {
    return intent != null;
}
```

Iz NDEF zapisovog payloada dohvaćamo byte array, te ga pretvaramo u string. Nakon toga kreiramo novu HashMap-u te u nju zapisujemo inicijalne vrijednosti za lampu ili senzore, te u pod prijavljenog korisnika u firebase zapisujemo te vrijednosti za lampu ili senzor.

```
private void getMessagesFromNfcAndWriteToFirebase(NdefRecord[] ndefRecord) {
    byte[] byteArray = ndefRecord[0].getPayload();
    String payloadString = new String(byteArray);
    Map<String, Object> data = new HashMap<>();
    putInitialDataValuesForLampAndSensors(data, payloadString);
    writeInUsersCollectionReferenceInitialValuesForSensors(data, payloadString);
    notifyUserForSuccessScan();
}

private void putInitialDataValuesForLampAndSensors(Map<String, Object> data,
          String plainText) {
    if(plainText.equals("lampa"))
        putLampInitialValue(data);
    else if (plainText.endsWith("senzor"))
        putTemperatureAndMoistureInitialValue(data);
}

private void writeInUsersCollectionReferenceInitialValuesForSensors(Map<String,
          Object> data, String plainText) {
    CollectionReference usersCollectionRef = db.collection("users").document(uid)
                                                .collection("Uredaji");
    usersCollectionRef.document(plainText).set(data);
}

private void putLampInitialValue(Map<String, Object> data) {
    data.put("upaljen","false");
}

private void putTemperatureAndMoistureInitialValue(Map<String, Object> data) {
    data.put("temperatura", "0");
    data.put("vlaga", "0");
}
```

Na sljedećoj slici je prikazan UI za aktivnost dodavanja novih uređaja za korisnika na firebase.

Slika 19 - UI prikaza podataka senzora

12. Spajanje wifi-jem na baznu stanicu i slanje GET zahtjeva

Kako bismo se spojili na hotspot kojega odašilje bazna stanica, trebamo prvo dohvatiti WifiManager-a preko WIFI_SERVICE konteksta. Te vraćamo true/false ovisno o uspješnosti paljenja wifi-a. Ukoliko je wifi već upaljen, metoda vraća true.

```
private boolean getWifiManagerAndEnableIt() {
    wifiManager = (WifiManager) this.getApplicationContext()
                            .getSystemService(Context.WIFI_SERVICE);
    return wifiManager.setWifiEnabled(true);
}
```

Nakon što pročitamo NFC tag, u kojem su zapisane SSID i lozinka od hotspota kojeg odašilje bazna stanica, šaljemo ih metodi Connect koja provjerava je li wifi uključen, nakon čega dohvaća network ID preko SSID-a i lozinke, te radi promjenu trenutno aktivne konekcije kako bi se aplikacija spojila na baznu stanicu. Te poziva metodu koja obavještava korisnika o spajanju na baznu stanicu.

```
public void Connect(String networkSSID, String networkPass)
{
    checkIfWifiManagerIsEnabledAndEnableIt();
    int netId = getNetworkIdBySettingSsidAndPass(networkSSID, networkPass);
    reconnectWifiManagerWithNewConnection(netId);
    notifyWifiConnection();
}
```

Metoda za dohvačanje network ID-a kreira novu wifi konfiguraciju u koju zatim zapisuje proslijeđeni SSID i lozinku, koje formatira tako da im doda navodnike. Nakon čega metoda vraća konfiguraciju kao novu wifi mrežu, tj. ID nove mreže.

```
private int getNetworkIdBySettingSsidAndPass(String networkSSID, String networkPass) {
    WifiConfiguration conf = new WifiConfiguration();
    conf.SSID = String.format("\"%s\"", networkSSID);
    conf.preSharedKey = String.format("\"%s\"", networkPass);
    return wifiManager.addNetwork(conf);
}
```

Metoda za rekonekciju prekida vezu s trenutno povezanom wifi mrežom, zatim postavlja novu mrežu preko proslijeđenog network ID-a, te ponovno pokreće wifi manager koji se zatim povezuje s novo dodanom mrežom.

```
private void reconnectWifiManagerWithNewConnection(int netId) {
    wifiManager.disconnect();
    wifiManager.enableNetwork(netId, true);
    wifiManager.reconnect();
}
```

12.1. Slanje GET zahtjeva baznoj stanici

Nakon što se povežemo s baznom stanicom, trebamo poslati GET zahtjev u kojeg upisujemo SSID i lozinku našeg kućnog wifi-a, kako bi se bazna stanica mogla povezati na firebase. Prvo kreiramo i pokrećemo RequestQueue. Nakon njega postavljamo url adresu na koju će se poslati GET zahtjev, te na url dodajemo i GET parametre. Poslije toga stvaramo StringRequest, odabiremo metodu kao GET, postavlamo url, te zadajemo response listenera koji će nam vratiti odgovor bazne stanice na naš GET zahtjev, te error response listenera koji će vratit grešku ako se dogodila. Zatim u RequestQueue dodajemo naš string request, te u slučaju da nismo dobili response, javljamo korisniku da provjeri polja za unos, te pokuša ponovno poslati GET zahtjev. U slučaju da je response jednak SUCESS, pozivamo metodu koja mijenja aktivnost na Main Activity.

```
public void sendGetRequest() {
    RequestQueue mRequestQueue = getRequestQueue();
    mRequestQueue.start();
    String url = setUrl();
    StringRequest stringRequest = new StringRequest(Request.Method.GET, url,
            new Response.Listener<String>() {
                @Override
                public void onResponse(String response) {
                    if(checkIfResponseIsSuccess(response))
                        changeActivityToMainActivity();
                }
            }, new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
            notifyUserForErrorWhileSendingGetMessage(error);
        }
    });
    mRequestQueue.add(stringRequest);
    notifyUserToCheckFieldsAndTryToSendAgain();
}
```

13. Sklopovlje

U prvom dijelu ovog poglavlja ćemo opisati implementaciju mrežnih servisa koje omogućuju kontroliranje te očitavanje podataka sa uređaja čvorova, te njihovu mrežnu konfiguraciju.

Nakon samih uređaja čvorova, opisati ćemo sve dijelove bazne stanice vezane uz komunikaciju sa mobilnom aplikacijom, uređajima čvorova te Firestore-om.

13.1. Mongoose OS

Mongoose OS je operacijski sustav otvorenog koda namijenjen razvoju ugrađenih, odnosno IOT (eng. Internet of Things) uređaja. Podržava razne mikrokontrolere, uključujući i naše ESP32 i ESP8266 uređaje.

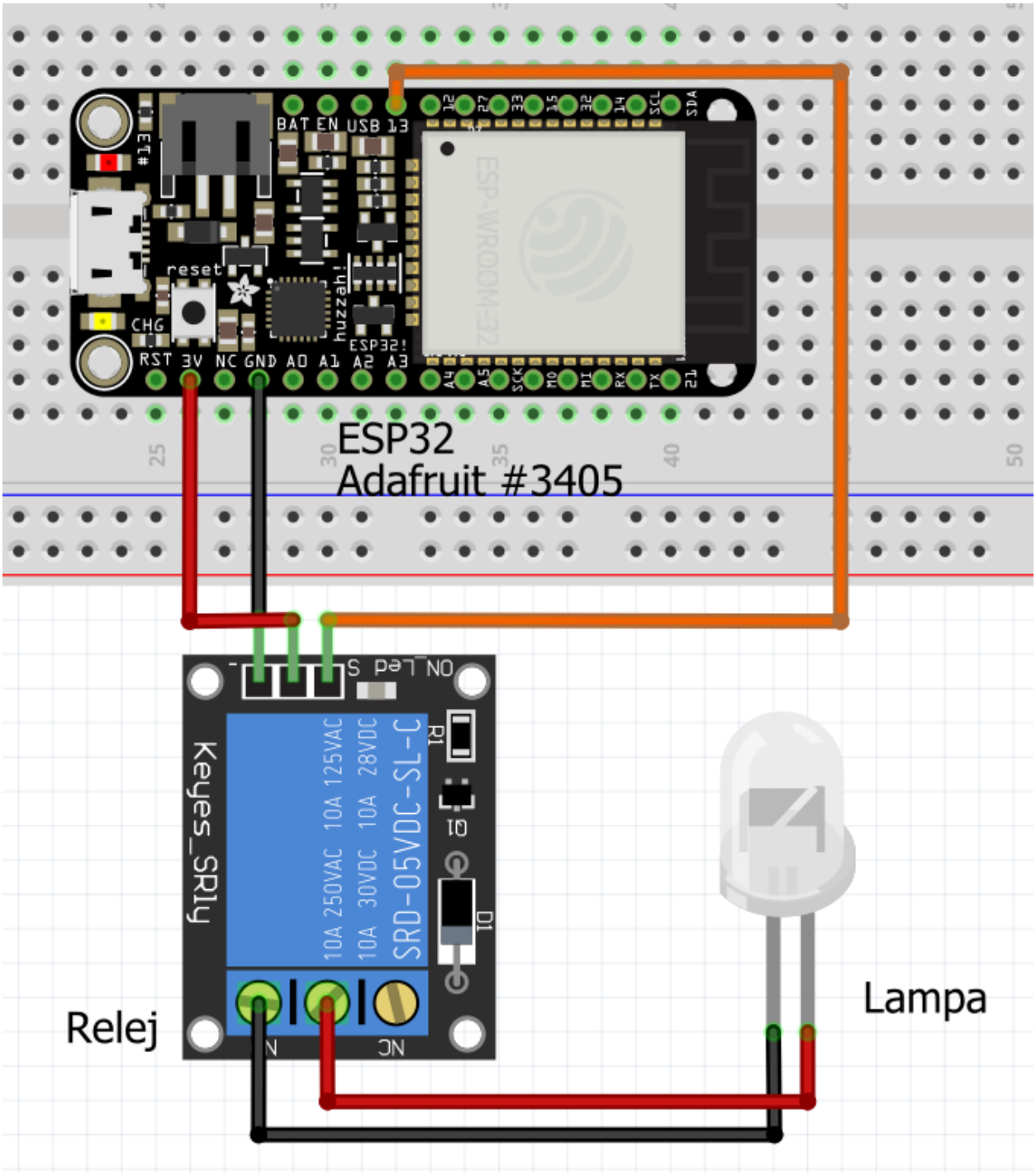
Programski jezici u kojima ga možemo programirati je JavaScript i C/C++. JavaScript se pokreće koristeći mJS – limitirani JavaScript engine, namijenjen mikrokontrolerima sa ograničenim resursima koji omogućuje malen 'footprint' te interoperabilnost sa C/C++ kodom koristeći *Foreign Function Interfacing*.

Mongoose nam omogućava full flash enkripciju i komunikaciju putem TLS v1.2 kriptografskog protokola koji omogućava sigurnu komunikaciju između naših uređaja.

Na nižoj razini, omogućava komunikaciju sa sklopovljem putem GPIO, SPI, I2C i mnogih drugih protokola.

Na kraju, jedna od najboljih mogućnosti koju nam pruža Mongoose je tzv. RPC (Remote Procedure Calls) servis, koji nam omogućava pozivanje C ili Javascript funkcije putem MQTT, HTTP i još nekolicine protokola.

13.2. Lampa (ESP32, Relej)



Slika 20 – Shematika lampe (ESP32, relej, lampa)

Ground (GND) pin releja spajamo sa Ground (GND) pinom ESP32. Voltage In (VIN) pin releja spajamo sa 3v pinom ESP32. Signal pin releja spajamo sa pinom 13 ESP32 (digitalni pin 0). Normally Open (NO) pin releja spajamo sa „fazom“ lampe, a GND pin releja spajamo sa uzemljenjem lampe.

```
let relayPin = 13;
GPIO.set_mode(relayPin, GPIO.MODE_OUTPUT);
GPIO.write(relayPin, 0);

RPC.addHandler('lampa', function(args) {
GPIO.toggle(relayPin);
});
```

Slika 21 - Lampa init.js

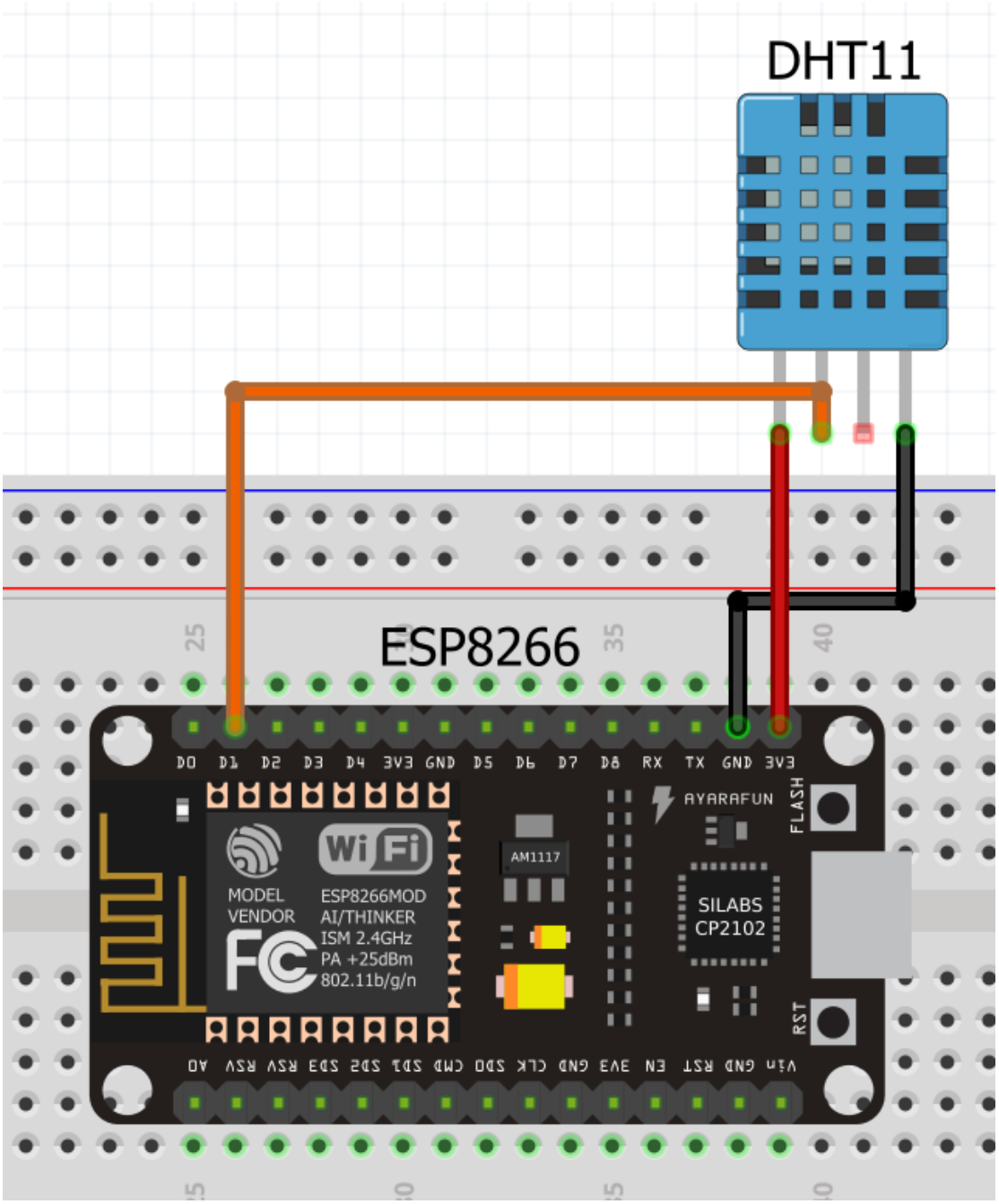
Na ESP32 Mongoose OS prilikom pokretanja (init.js) skripte definira relej koji se nalazi na digitalnom ulazu 1 (pin 13), te ga postavlja u način pisanja. Nakon toga dodajemo RPC handler pod nazivom *lampa* koji prilikom pozivanja mijenja stanje na digitalnom ulazu 1 koristeći funkciju GPIO.toggle. Ukoliko je stanje digitalnog ulaza 1 visoko (1), relej će propustiti struju do lampe te ju upaliti. Obrnuto vrijedi.

```
"wifi": {
    "sta": {
      "enable": true,
      "ssid": "BaznaStanica",
      "pass": "Athena1950!",
      "user": "",
      "anon_identity": "",
      "cert": "",
      "key": "",
      "ca_cert": "",
      "ip": "192.168.42.150",
      "netmask": "255.255.255.0",
      "gw": "192.168.42.1",
      "nameserver": "",
      "dhcp_hostname": ""
    }
```

Slika 22 - Mrežna konfiguracija lampe

Uređaj *lampa* ima definiran statički ip *192.168.42.150*, te njezinom RPC handleru možemo pristupiti preko *192.168.42.150/RPC/lampa* na privatnoj podmreži bazne stanice.

13.3. DHT11 - Senzor temperature i vlage



Slika 23 – Shematika DHT11 senzora za temperaturu i vlagu

Voltage Drain (VDD) te Ground (GND) pin DHT11 senzora spajamo sa 3v3 i GND pinom ESP8266 uređaja.

Data (DHT) pin DHT11 senzora spajamo sa D1 pinom ESP8266 uređaja.

```
let pin = 16;

let dht = DHT.create(pin, DHT.DHT11);
dht.begin();

RPC.addHandler('temperatura', function(){
  let t = dht.readTemperature(0, 0);
  return t;
});

RPC.addHandler('vlaga', function(){
  let h = dht.readHumidity(0);
  return h;
});
```

Slika 24 - DHT11 init.js

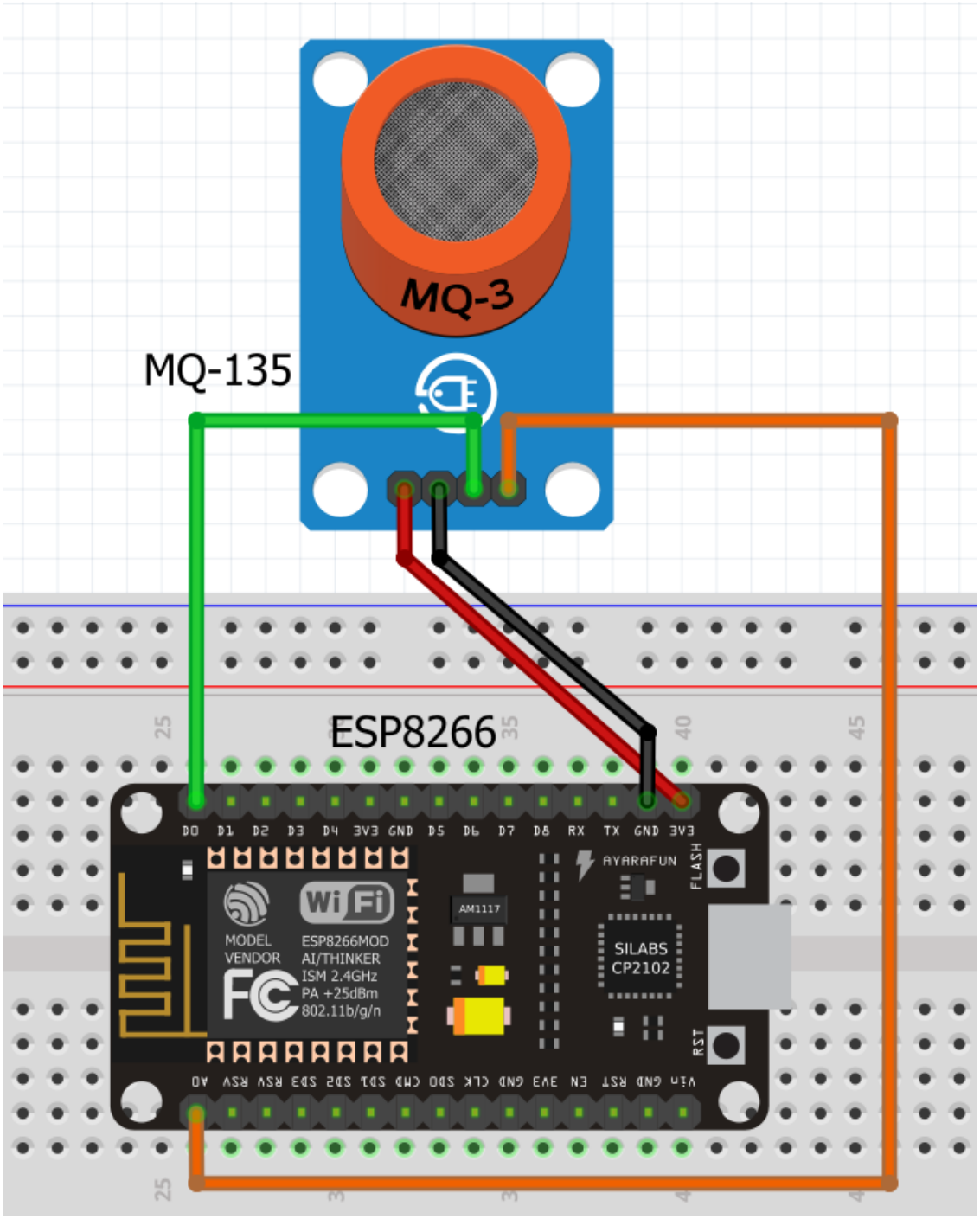
Na početku *init.js* skripte inicijaliziramo pin na kojemu se nalazi *data input* DHT11 senzora, u ovom slučaju pin 16 (D0 – digitalni pin 0), te inicijaliziramo *dht* objekt koristeći *DHT* biblioteku.

Dodajemo dva RPC handlera, jedan za temperaturu a jedan za vlagu, koji čitaju vrijednosti koristeći ranije spomenuti *dht* objekt.

```
"wifi": {
    "sta": {
      "enable": true,
      "ssid": "BaznaStanica",
      "pass": "Athena1950!",
      "user": "",
      "anon_identity": "",
      "cert": "",
      "key": "",
      "ca_cert": "",
      "ip": "192.168.42.151",
      "netmask": "255.255.255.0",
      "gw": "192.168.42.1",
      "nameserver": "",
      "dhcp_hostname": ""
    }
```

Slika 25 - Mrežna konfiguracija DHT11

13.4. MQ-135 - Senzor kvalitete zraka



Slika 26 – Shematika MQ-135 senzora kvalitete zraka

Kao i prilikom spajanja lampe i DHT11 senzora, Voltage Drain (VDD) te Ground (GND) pinove MQ-135 senzora spajamo sa GND i 3v3 pinovima ESP8266 uređaja. Digital Output i Analog Output (DO i AO) pinove senzora spajamo sa D0 i A0 pinovima ESP8266 uređaja.

```
MQ.MQ135.attach(0);

let rzero =0;
let correctedRZero = 0;
let resistance = 0;
let ppm = 0;
let correctedPPM = 0;
let temperature = 20;
let humidity = 10;


RPC.addHandler('airQuality', function(){
  RPC.call("http://192.168.42.151/RPC/", 'temperatura', '', function (resp, ud) {
  temperature = JSON.stringify(resp);
  print(temperature);
  }, null);

  RPC.call("http://192.168.42.151/RPC/", 'vlaga', '', function (resp, ud) {
  humidity = JSON.stringify(resp);
  print(humidity);
  }, null);

  rzero = MQ.MQ135.getRZero();
  correctedRZero = MQ.MQ135.getCorrectedRZero(temperature, humidity);
  resistance = MQ.MQ135.getResistance();
  ppm = MQ.MQ135.getPPM();
  correctedPPM = MQ.MQ135.getCorrectedPPM(temperature, humidity);
  return ":" + JSON.stringify(correctedPPM) + ":" + JSON.stringify(correctedRZero) +
":" + JSON.stringify(ppm) + ":" + JSON.stringify(rzero) + ":"  +
JSON.stringify(resistance) + ":";
});
```

Slika 27 – MQ-135 init.js

Na početku *init.js* skripte deklariramo i inicijaliziramo varijable potrebne za funkcioniranje senzora:

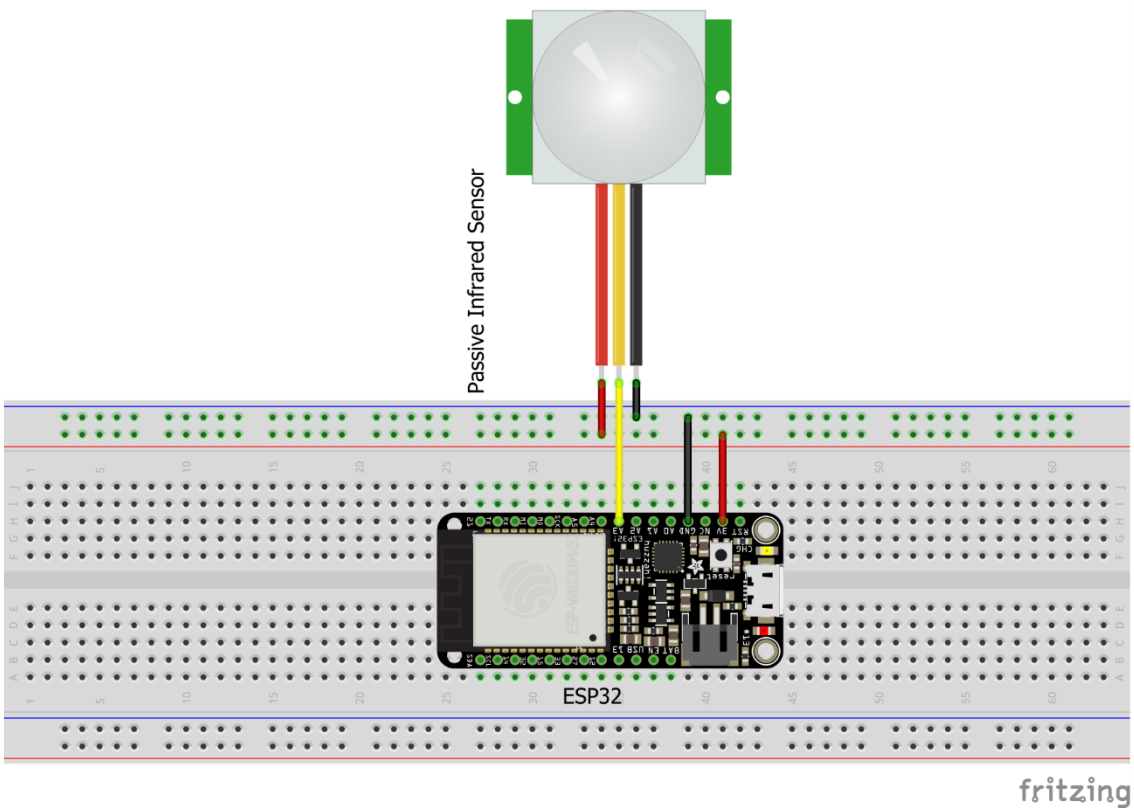
- RZero: Početni otpor senzora na 100PPM-a NH3 u čistom zraku (početni otpor, PPM)
- CorrectedRZero: Ispravljena vrijednost RZero uzimajući u obzir trenutnu temperaturu i vlagu u zraku
- Resistance: Početna vrijednost otpora na senzoru (moguće mijenjati pomoću potenciometra)
- PPM i CorrectedPPM: Količina čestica u zraku na milion (PPM – parts per milion), te ispravljena vrijednost uzimajući u obzir trenutnu temperaturu i vlagu u zraku
- Temperature i Humidity koje senzor dobija od drugog (DHT11) senzora preko RPC servisa

Dodajemo RPC handler pod nazivom *airQuality* koji poziva RPC DHT11 senzora kako bi dobio trenutnu vlagu i temperaturu u zraku. Ukoliko senzor nije dostupan, koriste se defaultne vrijednosti propisane za senzor. Nakon toga pomoću MQ.MQ135.get*NazivVrijednosti* dobivamo trenutne vrijednosti koje senzor očitava, te ih vraćamo kao String delimitiran dvotočkama, koje bazna stanica poslije parsira.

```
"wifi": {
    "sta": {
      "enable": true,
      "ssid": "BaznaStanica",
      "pass": "Athena1950!",
      "user": "",
      "anon_identity": "",
      "cert": "",
      "key": "",
      "ca_cert": "",
      "ip": "192.168.42.152",
      "netmask": "255.255.255.0",
      "gw": "192.168.42.1",
      "nameserver": "",
      "dhcp_hostname": ""
    }
```


Slika 28 - Mrežna konfiguracija MQ-135

13.5. PIR – Passive Infrared Sensor (Pasivni Infracrveni Senzor)



Slika 29 - Shematika MQ-135 senzora kvalitete zraka

Kao i prilikom spajanja lampe i ostalih senzora, Voltage Drain (VDD) te Ground (GND) pinove PIR senzora spajamo sa GND i 3v3 pinovima ESP32 uređaja. Output pin senzora spajamo sa A3 pinom ESP32 uređaja.

```
let pirSensor = 4;
GPIO.set_mode(pirSensor, GPIO.MODE_INPUT);

RPC.addHandler('detectMotion', function(){
  let motion = GPIO.read(pirSensor);
  let motion = JSON.stringify({
   data: {
     movement: motion
   }});
  return motion;
});
```

Slika 30 - PIR init.js

Na početku *init.js* skripte deklariramo i inicijaliziramo varijablu pirSensor koja definira GPIO pin na kojemu je priključen Output pin senzora.

Dodajemo RPC handler pod nazivom *detectMotion* koji dohvaća trenutnu vrijednost senzora, koja je "visoka" (odnosno jedan) ukoliko je pokret otkriven u krugu od 3 metra od senzora u protekloj minuti, ili "niska" (odnosno nula) ukoliko nije bilo pokreta u istom krugu u protekloj minute.


```
"wifi": {
    "sta": {
      "enable": true,
      "ssid": "BaznaStanica",
      "pass": "Athena1950!",
      "user": "",
      "anon_identity": "",
      "cert": "",
      "key": "",
      "ca_cert": "",
      "ip": "192.168.42.153",
      "netmask": "255.255.255.0",
      "gw": "192.168.42.1",
      "nameserver": "",
      "dhcp_hostname": ""
    }
```


Slika 31 - Mrežna konfiguracija MQ-135

14. Skripte na baznoj stanici

14.1. Bluetooth Agent

Skripta koja pomoću *ps aux* skenira sve trenutne procese filtrirajući ih po nazivu simple-agent, te ih sprema u varijablu rezultat. Ukoliko je rezultat prazan, odnosno *simple-agent* nije već pokrenut, pomoću *hciconfig* naredbe postavlja *hci0* interface u *PSCAN* i *ISCAN* način rada, laički rečeno, čini Bluetooth modul vidljivim. *Sspmode 0* isključuje Simple Pairing mode, pošto uređaj želimo spajati koristeći naš pin, koji smo ručno definirali unutar agenta. Nakon toga, pokreće se *simple-agent* koji dolazi uz bluez programsku podršku za Bluetooth.

```
---------------------- /usr/bin/btscript.sh ----------------------

#!/bin/sh
result=`ps aux | grep -i "simple-agent" | grep -v "grep" | wc -l`
if [ $result -ge 0 ]; then
    sudo hciconfig hci0 piscan
    sudo hciconfig hci0 sspmode 0
    sudo /usr/bin/python /usr/src/bluez-5.43/test/simple-agent &
else
    echo "BT Agent already started"
fi
```

Jedina izmjena koju smo načinili nad simple-agent skriptom koja dolazi uz bluez program, je definiranje našeg pina, koji je trenutno hardkodiran, a nalazi se unutar funkcije RequestPinCode koja se poziva nakon što neki uređaj zatraži uparivanje sa baznom stanicom preko simple-agenta.

```
---------------------- /usr/src/bluez-5.43/test/simple-agent ----------------------
 @dbus.service.method(AGENT_INTERFACE,
                                        in_signature="o", out_signature="s")
        def RequestPinCode(self, device):
                print("RequestPinCode (%s)" % (device))
                set_trusted(device)
                return "1950"
```

14.2. Provjera.js

Sljedeća skripta za dani UID provjerava koje sve uređaje korisnik posjeduje, te pokreće skripte koje su zadužene za komunikaciju i upravljanje tim uređajima. Također, skripta sadrži dva asinkrona slušatelja koji prate promjene nad čitavom kolekcijom korisnika te nad dokumentom lampa, ukoliko ju korisnik posjeduje. Ako se desi ikakva promjena nad čitavom kolekcijom korisnika, slušatelji pokreću skripte zadužene za upravljanje tim uređajima, kako bi se počeli kontrolirati novododani uređaji.

Prvi dio skripte sadrži ovisnosti te deklaracije i inicijalizacije potrebnih varijabli. *spawn* omogućava pokretanje shell skripti koristeći Node.Js, admin omogućava komunikaciju sa Firebase servisom, *http* omogućava slanje HTTP zahtjeva, *urlLampa* definira statički ip te putanju do RPC-a na kojemu se nalazi uređaj lampa, *serviceAccount* definira putanju do privatnog ključa koji omogućava pristupanje Firebase projektu.

senzorRef, *kvalitetaZraka* i *lampaRef* definiraju relativne putanje do tri moguća uređaja, DHT11 senzora, lampe te MQ-135 uređaja.

```
const {spawn} = require('child_process');
const admin = require('firebase-admin');
const http = require("http");
const urlLampa = "http://192.168.42.150/RPC/lampa";
var serviceAccount = require("./firestore.json");

admin.initializeApp({credential: admin.credential.cert(serviceAccount)});

var db = admin.firestore();
var uid = process.argv[2];

const senzorRef =
db.collection('users').doc(uid).collection('Uredaji').doc('senzor');
const kvalitetaZraka =
db.collection('users').doc(uid).collection('Uredaji').doc('kvalitetaZraka');
const lampaRef = db.collection('users').doc(uid).collection('Uredaji').doc('lampa');
```

Nad lampom je definiran onSnapshot koji služi kao listener. On vidi kada se stanje u firebaseu promijenilo, te provjerava novo stanje, ukoliko je novo stanje true, radi prazan GET request na url na 192.168.42.150/RPC/lampa zanemarivši sve podatke koje bi request vratio. Lampa se pali/gasi kada se GET request izvrši.

```
function slusajLampu() {

    var observer = lampaRef.onSnapshot(docSnapshot => {
        if (docSnapshot.data().upaljen == "true") {
            http.get(urlLampa, res => {
                res.setEncoding("utf8");
                let body = "";
                res.on("data", data => {});
                res.on("end", () => {});
            });
            console.log("Lampa je upaljena!");

        } else {
            http.get(urlLampa, res => {
                res.setEncoding("utf8");
                let body = "";
                res.on("data", data => {});
                res.on("end", () => {});
            });
            console.log("Lampa je ugašena!");
        }

    }, err => {console.log(`Encountered error: ${err}`);});
}
```

Funkcija koja se pokreće putem node.JS-a kada observer primjeti da se dogodila promjena nad određenim dokumentom. U principu ova skripta pretražuje ima li user sa danim UID-om lampu, senzor za temperaturu i senzor za kvalitetu zraka, ukoliko isti postoje pokreću se skripte koje očitavaju podatke za one senzore koji postoje, čita te podatke sa zadanih adresa te iste podatke sprema na Firebase, dodatno je napravljeno osvježavanje svakih jedne minute.

```
function pokreniSkripte() {

    var getDoc = senzorRef.get()
        .then(doc => {
            if (!doc.exists) {console.log('Dokument senzor ne postoji!');}
            else {const deploySh = spawn('sh', ['/home/pi/athena/senzor.sh']);}
        })
        .catch(err => {console.log('Error getting document', err);});

    var getDoc = kvalitetaZraka.get()
        .then(doc => {
            if (!doc.exists) {console.log('Dokument kvalitetaZraka ne postoji!');}
            else {const deploySh = spawn('sh',
['/home/pi/athena/kvalitetaZraka.sh']);}
        })
        .catch(err => {console.log('Error getting document', err);});

    var getDoc = lampaRef.get()
        .then(doc => {
            if (!doc.exists) {console.log('Dokument lampa ne postoji!');} else
{slusajLampu();}
        })
        .catch(err => {console.log('Error getting document', err);});

}
```

Definiranje observera koji će ukoliko se ikakav dokument doda ili promijeni, provjera se odvija tako da se odredi kolekcija i document u kojemu se očekuje da će se nalaziti kolekcija, a koja se nalazi kao dio tog korisnika (u našem slučaju to je uređaj koji je registriran za tog korisnika) da ponovno pokrene funkciju pokreniSkripte() kako bi provjerio je li dodan neki novi uređaj.

```
var doc = db.collection('users').doc(uid);
var observer = doc.onSnapshot(docSnapshot => {pokreniSkripte();},
    err => {console.log(`Encountered error: ${err}`);}
    );
}
```

14.3. Podaci.sh

Skripta koja se pokreće kada stignu podaci preko bluetootha na GATT server, radi na način da uz pomoć *awk* tj. naredbe koja traži I mijenja tekst uzima i u nultu stavku stavlja ono što se nalazi u stavci koja je potrebna i koja se može odabrati iz stavki koje su ranije razdjeljene I određene datoteke u ovom slučaju datoteke naziva podaci. Nakon što se odradi taj korak sprema ssid, pwd i uid unutar zadanog direktorija u zadane fileove. Nakon ovog koraka radi se učitavanje u varijable ranije zadanih vrijednosti koji se kasnije koriste za kreiranje zapisa u wpa_supplicant.conf fileu.

```
---------------------- /home/pi/when-changed/podaci.sh ----------------------

#!/bin/bash
awk -F'"' '$0=$2' /home/pi/GATT/python-gatt-server/podaci > /home/pi/GATT/python-
gatt-server/ssid
awk -F'"' '$0=$4' /home/pi/GATT/python-gatt-server/podaci > /home/pi/GATT/python-
gatt-server/pwd
awk -F'"' '$0=$6' /home/pi/GATT/python-gatt-server/podaci > /home/pi/GATT/python-
gatt-server/uid
ssid=$(cat /home/pi/GATT/python-gatt-server/ssid)
pwd=$(cat /home/pi/GATT/python-gatt-server/pwd)
wpa_passphrase ${ssid} ${pwd} > /etc/wpa_supplicant/wpa_supplicant.conf
```

14.4. Bazna_stanica.sh i registriraj_baznu_stanicu.py

Skripta koja se pokreće kada novi UID dođe preko Bluetootha na Baznu stanicu, pokreće registriraj_baznu_stanicu.py na tog korisnika sa tim UID-jem

```
---------------------- /home/pi/when-changed/bazna_stanica.sh ----------------------

#!/bin/bash
uid=$(cat /home/pi/GATT/python-gatt-server/uid)
python /home/pi/athena/registriraj_baznu_stanicu.py ${uid}
```

Skripta koja za dani UID registrira baznu stanicu pod tog usera. Importa firestore, te se spaja s podacima zapisanim u Athena-4fdb9da08e48.json. Skripta zatim uzima kolekciju iz Firebase-a te pod users, pronalazi njegov ID, koji je njoj proslijeđen u argumentu, i u Uređaji registrira document Bazna Stanica.

```
---------------------- /home/pi/athena/registriraj_baznu_stanicu.py ------------------
---
from google.cloud import firestore
import requests
import sys
import os
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "/home/pi/athena/Athena-
4fdb9da08e48.json"
def unesiUredaj(argv):

    db = firestore.Client()
    users_ref = db.collection(u'users')
    dokument = users_ref.get()
    postoji = False

    for dok in dokument:
        if argv == dok.id:
            postoji = True

    if postoji:
        try:
            bazna_ref =
db.collection(u'users').document(u'{}'.format(argv)).collection(u'Uredaji').document(
u'Bazna Stanica')
            bazna_ref.set(
                {
                        "setup":"done"
                })
        except:
            print "Pogresni argumenti"
    else:
        print "User ne postoji"

if __name__ == '__main__':
    from sys import argv
    unesiUredaj(argv[1])
```

14.5. Hostapd i hostapd.conf

Koristeći naredbu *hostapd -dd /putanja/do/datoteke.conf* pokrećemo pristupnu točku (AP) sa postavkama navedenim u datoteci.

Naš Hostapd konfiguracijski file koji pokreće AP sa zadanim postavkama:

```
interface=wlan0
driver=nl80211
ssid=BaznaStanica
country_code=HR
hw_mode=g
channel=1
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=Athena1950!
wpa_key_mgmt=WPA-PSK
```

14.6. kvalitetaZraka.sh i .py

Skripta koju pokreće provjera.js ako vidi da user na kojega je registrirana bazna stanica ima senzor za kvalitetu zraka

```
---------------------- /home/pi/athena/kvalitetaZraka.sh ----------------------
#!/bin/bash
uid=$(cat /home/pi/GATT/python-gatt-server/uid)
python /home/pi/athena/kvalitetaZraka.py ${uid} &
```

Skripta koja svaku minutu pinga senzor kvalitete zraka i updatea podatke na dani uid koji se skripti proslijeđuje kao parametar. Skripta iz kolekcije unutar users dohvaća trenutno prijavljenog korisnika čiji user ID joj je proslijeđen kao parametar, zatim u kolekciji Uređaji u dokumentu kvalitetaZraka zapisuje podatke koje potom dohvati sa senzora kojemu šalje zahtjev na url adresu sa slike.

```
---------------------- /home/pi/athena/kvalitetaZraka.py ----------------------
from google.cloud import firestore
import requests
import sys
import os
import time
starttime=time.time()
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "/home/pi/athena/Athena-
4fdb9da08e48.json"
def pingajUredaj(argv):

    db = firestore.Client()
    doc_ref =
db.collection(u'users').document(u'{}'.format(argv)).collection(u'Uredaji').document(
u'kvalitetaZraka')
    r = requests.get('http://192.168.42.152/RPC/airQuality')
    airQuality = r.text
    correctedPPM = airQuality.split(":")[1]
    correctedRZero = airQuality.split(":")[2]
    ppm = airQuality.split(":")[3]
    rzero = airQuality.split(":")[4]
    resistance = airQuality.split(":")[5]

doc_ref.set({u'CorrectedPPM':correctedPPM,u'CorrectedRZero':correctedRZero,u'PPM':ppm
,u'RZero':rzero,u'Resistance':resistance})

if __name__ == '__main__':
    from sys import argv
    print(argv[1])
    while True:
        pingajUredaj(argv[1])
        time.sleep(60.0 - ((time.time() - starttime) % 60.0))
```

14.7. senzor.sh i .py

Skripta koju pokreće provjera.js ako vidi da user na kojega je registrirana bazna stanica ima senzor za vlagu i temperaturu

```
---------------------- /home/pi/athena/senzor.sh ----------------------
#!/bin/bash
uid=$(cat /home/pi/GATT/python-gatt-server/uid)
python /home/pi/athena/senzor.py ${uid} &
```

Skripta koja svaku minutu pinga senzor za temperaturu i vlagu i updatea podatke na dani uid koji se skripti proslijeđuje kao parametar. Skripta dohvaća u kolekciji korisnika prema njegovom user ID-u, te u kolekciji Uređaji u dokumentu sensor zapisuje podatke koje dobiva od senzora temperature i vlage tako da dohvati njihove podatke s url adresa.

```
---------------------- /home/pi/athena/senzor.py ----------------------
from google.cloud import firestore
import requests
import sys
import os
import time
starttime=time.time()
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "/home/pi/athena/Athena-
4fdb9da08e48.json"
def pingajUredaj(argv):

    db = firestore.Client()
    doc_ref =
db.collection(u'users').document(u'{}'.format(argv)).collection(u'Uredaji').document(
u'senzor')
    r = requests.get('http://192.168.42.151/RPC/temperatura')
    temp = r.json()
    r = requests.get('http://192.168.42.151/RPC/vlaga')
    vlaga = r.json()
    doc_ref.set({u'temperatura': u''+str(temp)+'', u'vlaga': u''+str(vlaga)+''})
    print({u'temperatura': u''+str(temp)+'', u'vlaga': u''+str(vlaga)+''})
if __name__ == '__main__':
    from sys import argv
    print(argv[1])
    while True:
        pingajUredaj(argv[1])
        time.sleep(60.0 - ((time.time() - starttime) % 60.0))
```

15. Testiranje aplikacije

Za testiranje aplikacije odabrano je komponentno testiranje iz razloga, što je potreba za jediničnim testovima smanjena činjenicom da je većina koda u aplikacije ovisna o vanjskim servisima, kao što su Firebase, različiti menadžeri za pojedine prijave u aplikaciju. Primjer je prijava u Google koji bih naveo kao referentni, iz razloga jer se i za ostale prijave koriste menadžeri koji su definirani u javno dostupnoj dokumentaciji pružatelja servisa. Kako je velik dio koda koji se izvršava unutar aplikacije ovisan o vanjskim servisima, taj dio nije testiran jer bi za kreiranje jediničnih testova bilo potrebno mockati podatke koji su uvelike ovisni o tim istim servisima. Shodno tome bih naveo da bi za testiranje jedne funkcije bilo potrebno pozvati na desetke drugih te bi mock takvih podataka predstavljao problem. Iz navedenog razloga odlučilo se da će se testiranje aplikacije biti obavljano kao komponentno testiranje. Za obavljanje ove vrste testiranja aplikacije koristio se espesso programski okvir. Za omogućavanje korištenja istog potrebno je dodati ovisnosti za prevođenje testova u androidu i to na razini gradel-a za aplikaciju. Primjer koda slijedi:

```
androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
    exclude group: 'com.android.support', module: 'support-annotations'
})
```

15.1. LoginActivityTest

Nakon dodavanja ovisnosti u projekt, testirane su različite komponente aplikacije, prvo je testirana prijava u samu aplikaciju a test se sastoji od provjera postojanja različitih elemenata u ovom primjeru gumbova na samoj aktivnosti, razlog testiranja samo ovog dijela je taj da ukoliko gumb postoji sve nakon se predaje vanjskom servisu na obradu te se taj dio ne testira na unutar našeg dijela aplikacije. Test je na kraju rezultirao uspjehom ukoliko svi elementi postoje, te neuspjehom ukoliko jedan, dva ili sva tri gumba ne postoje na samoj aktivnosti prilikom obavljanja testiranja. Kod je trenutno spremljen unutar paketa team.athena(androidTest) unutar klase za testiranje *LoginActivityTest*. Unutar klase nalazi se i pravilo po kojem određuje koja će se aktivnost testirati, primjer koda slijedi:

```
@Rule
public ActivityTestRule<LoginActivity> mActivityTestRule = new
ActivityTestRule<>(LoginActivity.class);
```

Nakon pravila definira se i funkcija za testiranje unutar same klase, kod funkcije je prikazan kroz nekoliko slijedećih linija:

```
@Test
public void loginActivityTest() {
    delayToMatchAppExecutionDelay();
    checkIfGoogleLogInButtonExists();
    checkIfTwitterLogInButtonExists();
    checkIfFacebookLogInButtonExists();

}
```

Kao što se može vidjeti definirana funkcija sadrži dodatne funkcije koje su definirane za potrebe preglednosti koda testiranje za koju se smatra da je jednako važna kao i preglednost i čitljivost koda same aplikacije koja se pokreće na zadanoj platformi. Sve funkcije koje se definiraju unutar ove funkcije koja je javna su privatne te su u njima definirane akcije koje se rade u testovima. Prva funkcija koja se pokreće je *delayToMatchAppExecutionDelay()* i u njoj se definira potreban zastoj unutar aplikacije da bi se nadoknadilo kašnjenje pokretanja testa, ovaj dio je standardno postavljen na šezdeset sekundi no za potrebe obavljanja ovih testova mi smo stavili da on bude jednu sekundu tj. tisuću milisekundi kao parametar za potrebnu funkciju. Primjer koda za navedenu funkciju slijedi:

```
private void delayToMatchAppExecutionDelay() {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Druga na popisu funkcija je *checkIfGoogleLogInButtonExists(),* unutar nje je definiran raspored unutar okvira koji će se prikazati, nakon toga se gleda svaki element koji ima zadani id koji se proslijedi kao parametar te se pozove funkcija isDisplayed() koja služi kao provjera dali taj zadani element u ovom slučaju gumb postoji na samoj aktivnosti, nakon toga se radi provjera koja vidi dali je pronađeno podudaranje, prikaz koda funkcije slijedi:

```
private void checkIfGoogleLogInButtonExists(){
    ViewInteraction frameLayout = onView(
            allOf(withId(R.id.google_button), isDisplayed()));
    frameLayout.check(matches(isDisplayed()));
}
```

Kako je dio koda za testiranje sljedeća dva gumba koja služe za prijavu na Facebook i Twitter identičan prethodno navedenom te radi iste provjere, taj dio se neće posebno prikazivati.

15.2. SetupActivityTestGet

Za testiranje aktivnosti za postavljanje mreže preko koje će se bazna stanica spajati na Internet korisnika te preko kojega će dalje izmjenjivati podatke s firebase bazom podataka. Prilikom kreiranja testa te funkcije za test pod imenom *setupActivityTestGet()* u kojoj su navedene ostale funkcije koje će se kasnije pregledavati, te koje su tako postavljene radi preglednosti i čitljivosti samog testa. Prije samog testiranja potrebno je bilo postaviti pravilo koje određuje koja će se aktivnost pokrenuti prilikom pokretanja testa, prikaz koda slijedi:

```
@Rule
public ActivityTestRule<SetupActivity> mActivityTestRule = new
ActivityTestRule<>(SetupActivity.class);
```

Nakon postavljanja pravila definirana je i ranije nevedena funkcija koja će se pokretati prilikom pokretanja test za koju kod i slijedi:

```
@Test
public void setupActivityTestGet() {
    delayToMatchAppExecutionDelay();
    setTextOfSsidTextElement();
    setTextOfPasswordTextElement();
    performClickOnSendButtonElement();

}
```

Prva funkcija koja se nalazi u funkciji za testiranje je *delayToMatchAppExecutionDelay(),* koja je već ranije objašnjena te se iz toga razloga neće ponovno objašnjavati. Nakon nje slijedi funkcija za postavljanje teksta unutar elementa za unos, prvo se unosi dio za SSID mreže koja će se postavljati, prvo što se unutar samog testa odradi je zamjena teksta koji se nalazi kao standardni unutar tog elementa te se zatvara tipkovnica uređaja, dio koda kojim je to realizirano slijedi:

```
private void setTextOfSsidTextElement() {
    ViewInteraction appCompatEditText = onView(
            allOf(withId(R.id.ssidText), isDisplayed()));
    appCompatEditText.perform(replaceText("testSSID"), closeSoftKeyboard());
}
```

Sljedeće je postavljanje teksta za Password unutar polja namjenjenog za to. Ovaj dio teksta će se također slati unutar zahtjeva prema baznoj stanici. Primjer funkcije nije potrebno detaljno prikazivati jer je identičan prethodno navedenom. Sljedeća po redu je funkcija *performClickOnSendButtonElement()* koja testira pritisak gumba unutar aplikacije te se nakon pritiska aplikacija zatvara te se test završava uspješno ukoliko su svi koraci uspješno obavljeni. Primjer koda ove funkcije slijedi:

```
private void performClickOnSendButtonElement() {
    ViewInteraction appCompatButton = onView(
            allOf(withId(R.id.sendButton), withText("Send"), isDisplayed()));
    appCompatButton.perform(click());
}
```

Kao što se može vidjeti ovdje je dodatno definiran i tekst koji se nalazi unutar gumba kako bi se aktivirao samo taj gumb te tako dodatno osiguralo pokretanje baš tog gumba, putem funkcije perform() obavlja se željena akcija, a to je u ovom slučaju pritisak na gumb.

15.3. MainActivityAddDeviceTest

Ovaj test primarno služi za provjeru kako radi dodavanje uređaja unutar aplikacije, te dali postoje svi elementi potrebni za obavljanje ove aktivnosti. Prvo što je potrebno uraditi je definirati pravilo pomoću kojeg će se aktivirati glavna aktivnost unutar same aplikacije kako bi se prikazao gumb potreban za dodavanje uređaja te kako bi se dodatno ušlo i u tu aktivnost, te testirale sve mogućnosti na toj aktivnosti. Prikaz pravila za pokretanje aktivnosti slijedi:

```
@Rule
public ActivityTestRule<MainActivity> mActivityTestRule = new
ActivityTestRule<>(MainActivity.class);
```

Nakon definiranja pravila za pokretanje aktivnosti prilikom testiranja, napravljena je funkcija za testiranje u koj se nalaze ostale funkcije, funkcije su izdvojene kao posebne da bi se omogućila bolja preglednost koda samog testa, prikaz koda slijedi:

```
@Test
public void mainActivityAddDeviceTest() {
    delayToMatchAppExecutionDelay();
    checkIfDeviceAddButtonIsDisplayed();
    performClickOnDeviceAddButton();

}
```

Kao što se može vidjeti unutar same funkcije postoje ranije definirane funkcije koje su slične, jedina bitna razlika je navedeni id pomoću kojega se izvršava pretraga elementa na samoj aktivnost. Ulazak na samu aktivnost je prikazan no obavljanje radnje nije bilo moguće prikazati kroz komponentne testove zbog potrebe fizičkog prislanjanja NFC taga mobitelu s instaliranom aplikacijom kako bi se dodao novi uređaj korisniku, te spremio na za to predviđeno mjesto na firebase-u.

15.4. MainActivityLampTest

Prilikom testiranja akcije upravljanja lampom unutar same aplikacije bilo je potrebno provjeriti dali postoji sami gumb za lampu te ukoliko on postoji izvrši se pritisak te se nakon toga obavlja testno upravljanje akcijama nad lampom te pristisci na gumb unutar iste aktivnosti. Prvo što je bilo potrebno je postaviti pravilo za pokretanje početne aktivnosti za testiranje a to je u ovom slučaju bila glavna aktivnost aplikacije. Prikaz pravila slijedi:

```
@Rule
public ActivityTestRule<MainActivity> mActivityTestRule = new
ActivityTestRule<>(MainActivity.class);
```

Nakon postavljanja pravila definirana je funkcija za testiranje unutar aplikacije, unutar koje se dodatno definiraju ostale funkcije koje su realizirane na sljedeći način radi preglednosti i čitljivosti koda za testiranje. Prikaz koda slijedi:

```
@Test
public void mainActivityLampTest() {
    delayToMatchAppExecutionDelay();
    checkIfLampButtonElementIsDisplayed();
    performClickOnLampButtonElement();
    delayToMatchAppExecutionDelay();
    checkIfToggleButtonIsDisplayed();
    performClickOnToggleOFFButtonElement();
    delayToMatchAppExecutionDelay();
    performClickOnToggleONButtonElement();
}
```

Kao što se može vidjeti iz prikaza funkcija, unutar jedne veće za testiranje funkcije su identične kao i one u prethodnim klasama za testiranje uz jedinu razliku, a to je ta da se id elementa koji se traži razlikuje od funkcije do funkcije, te se iz tog razloga neće detaljno prikazivati poseban kod za prikazane funkcije.

15.5. MainActivitySensorsTest

Prilikom testiranja aktivnosti pregleda vrijednosti koje senzori očitavaju unutar prostorije korisnika, potrebno je provjeriti postojanje gumba za senzore, nakon toga se obavlja pritisak tog gumba. Nakon pritiska pokreće se nova aktivnost na kojoj se provjerava postojanje određenih elemenata, u ovom slučaju to su elementi za prikaz vrijednosti koje se trebaju prikazati na aktivnosti te se nakon toga provjeraja postojanje gumba za detalje, nakon čega se izvršava i pritisak na gumb koji otvara novu aktivnost pregleda detalja, na toj se aktivnosti dodatno provjerava postojanje elemenata za detalje koji služe za prikaz vrijednosti senzora za kvalitetu zraka. Prilikom pokretanja ovog testa potrebno je

postaviti pravilo za pokretanje potrebne aktivnosti a to je glavna aktivnost aplikacije, prikaz koda slijedi u nastavku:

```
@Rule
public ActivityTestRule<MainActivity> mActivityTestRule = new
ActivityTestRule<>(MainActivity.class);
```

Nakon postavljanja pravila potrebno je kreirati funkciju koja će se testirati unutar same klase, kao i u prethodnim klasama, ta funkcija sadži druge funkcije koje su kreirane iz razloga bolje preglednosti i čitljivosti koda namjenjenog za pokretanje testova, prikaz koda te funckije slijedi u nastavku:

```
@Test
public void mainActivitySensorsTest() {
    checkIfSensorsButtonElementIsDisplayed();
    delayToMatchAppExecutionDelay();
    performClickOnSensorsButtonElement();
    delayToMatchAppExecutionDelay();
    checkIfTemperatureDataIsDisplayed();
    checkIfHumidityDataIsDisplayed();
    checkIfTextViewPpmDataIsDisplayed();
    performClickOnPpmDataTextViewElement();
    delayToMatchAppExecutionDelay();
    checkIfPPMDataIsDisplayed();
    checkIfCorrectedPPMDataIsDisplayed();
    checkIfRZeroDataIsDisplayed();
    checkIfCorrectedRZeroDataIsDisplayed();
    checkIfResistanceDataIsDisplayed();
}
```

Kao i u prethodnoj klasi ove funkcije nije potrebno dodatno objašnjavati iz razloga jer su prethodno navedeno identične funkcije uz iznimku drgačijeg id-a koji se koristi za pronalazak elementa na samoj aktivnosti. Ukoliko su sve funkcije zadovoljene te ukoliko je funkcija za testiranje uspješno obavljena test se smatra gotovim te uspješno obavljenim.

15.6. MainActivityStatisticsTest

Testiranje komponente pregleda statistike za premium korisnika aplikacije se obavlja tako da se pokrene glavna aktivnost aplikacije, na kojoj se prilikom pronalaska definiranog gumba izvrši i pritisak na isti. Nakon pritiska gumba otvara se nova aktivnost na kojoj se može gledati statistika podataka za očitanja senzora i to za podatke koji su prikazani kod korisnika, korisnik može gledati i podatke u određenom vremenu tj. za određeni vremenski raspon. Prvo što je potrebno postaviti prilikom samog testiranja je pravilo koje definira da se pokreće glavna aktivnost aplikacije kako bi se moglo doći do potrebne aktivnosti, prikaz koda slijedi:

```
@Rule
public ActivityTestRule<MainActivity> mActivityTestRule = new
ActivityTestRule<>(MainActivity.class);
```

Nakon što se postavi pravilo potrebno je definirati i funkciju koja će se testirati, a unutar koje su kao i ranije pozivane ostale funkcije koje će se izvršavati po redu, ovaj način prikaza je odabran kako bi se poboljšala čitljivost i preglednost koda klase za testiranje. Dodatno važno za napomenuti je da je ova klasa bila najkompliciranija za testiranje upravo iz razloga jer je postojao veliki broj elemenata na samoj aktivnosti. Prikaz koda funkcije slijedi:

```
@Test
public void mainActivityStatisticsTest() {
    delayToMatchAppExecutionDelay();
    checkIfStatisticsButtonIsDisplayed();
    performClickOnStatisticsButtonElement();
    delayToMatchAppExecutionDelay();
    checkIfButtonCategoryIsDisplayed();
    checkIfButtonDateIsDisplayed();
    checkIfButtonDate2IsDisplayed();
    checkIfButtonOdDoIsDisplayed();
    checkIfLineChartIsDisplayed();
    performClickOnButtonCategory2Element();
    performClickOnTitleElementWithNameHumidity();
    performClickOnButtonWithTextDateFrom();
    performClickOnButtonWithTextOK4();
    performClickOnButtonWithTextDateUntil();
    performClickOnButtonWithTextOK();
    performClickOnButtonOdDoWithShowText();
    performClickOnButtonCategory8Element();
    performClickOnTitleElementWithNamePPM();
    performClickOnButtonCategoryElement();
    performClickOnTitleElementWithNameTemperature();
    delayToMatchAppExecutionDelay();
}
```

Kao i u prethodno navedenoj klasi nije potrebno dodatno objašnavati funkcije koje se nalaze unutar funkcije za testiranje jer su one identične s onima ranije navedenim, uz ključnu razliku id-a prema kojemu se pronalazi potrebni element na pokrenutoj aktivnost unutar aplikacije. Uspješno testiranje se obavi u slučaju da se uspješno izvrše sve funkcije koje su navedene prethodno te po prethodno navedenom poretku.