

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Karmelo Mrvica, Ivan Novosel,

Marko Vrščak, Mateo Zović,

Jakov Begović

Studij: Informacijski i poslovni sustavi

SMART WINDOW STATE SENSOR

PROJEKT

Mentor/Mentorica:

mag. inf. Lovro Posarić

Varaždin, siječanj 2023.

Posebne zahvale mentoru projekta asistentu mag. inf. Lovri Posariću na izdašnoj pomoći i neprestanoj dostupnosti.

Karmelo Mrvica, Ivan Novosel,

Marko Vrščak, Mateo Zović,

Jakov Begović

Izjava o izvornosti

Izjavljujemo da je naš projekt izvorni rezultat našeg rada te da se u izradi istoga nismo koristili drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Sažetak

Smart Window State Sensor (pametni senzor za detekciju stanja prozora) je projekt koji ima za svrhu obavještavanje korisnika o potrebi za otvaranjem prozora. Potreba za otvaranjem prozora proizlazi iz informacija o kvaliteti zraka unutar određene prostorije i kvaliteti zraka izvan prostorije. Kvaliteta zraka ovisi o količini ugljikovog dioksida koji je prisutan u sastavu zraka – što je manje ugljikovog dioksida prisutno u zraku je to zrak bolji. Uređaji korisnika se mogu pretplatiti na ovu temu (topic) te time mogu primati obavijesti o potrebi za otvaranjem i zatvaranjem prozora putem Android aplikacije. Za detekciju je li prozor otvoren ili nije korišten je senzor udaljenosti, koji mjeri stupanj otvorenosti prozora. Ova implementacija senzora udaljenosti predviđena je za mjerjenje otvorenosti prozora u položaju kip (ventus). Sve navedene komponente zajedno komuniciraju i razmjenjuju podatke i informacije putem Azure Cloud servisa. Podaci senzora o kvaliteti zraka u prostoriji i izvan nje se dohvaćaju putem internetske stranice Smart Citizen API.

Ključne riječi: pametni senzor, prozor, kvaliteta zraka, senzor udaljenosti, Android

aplikacija, Azure Cloud servis, podaci

Sadržaj

1. Uvod.....	1
2. Modeli.....	2
2.1. Općenita arhitektura sustava	2
2.1.1. Dasduino CONNECTPLUS (ESP32)	3
2.1.2. Senzor udaljenosti – HC-SR04 Ultrasonic Sensor.....	4
2.1.3. Senzori kvalitete zraka i Smart Citizen	6
2.1.4. Microsoft Azure Cloud	7
2.1.5. Android Studio	9
2.2. Složena arhitektura sustava.....	10
2.3. Komunikacijski dijagrami.....	12
2.4. Dijagram aktivnosti.....	16
2.5. Shema (dijagram) spajanja	27
3. Izvršni programi.....	28
3.1. Senzor udaljenosti.....	28
3.2. Logic App servisi.....	30
3.3. Function App servisi.....	31
3.3.1. IoT Hub Trigger Function App.....	31
3.3.2. Time Trigger Function App	37
3.4. Android aplikacija.....	41
4. Kratki troškovnik i finansijska analiza	50
5. Korisnička dokumentacija.....	65
6. Zaključak	77
Popis literature.....	78
Popis slika	79

1. Uvod

Pojam Smart Window State Sensor predstavlja senzor koji služi za prepoznavanje ili detekciju stanja prozora – je li prozor otvoren ili zatvoren. Ovakva tematika savršeno opisuje zamisao IoT (eng. Internet of Things) uređaja. Cilj IoT uređaja, i cijelog njihovog sustava, jest da obične predmeta i stvari iz svakidašnjeg života pretvaraju u „pametne“ stvari. Riječ „pametni“ (eng. smart) u ovom kontekstu označava davanje novih vrijednosti predmetu iz fizičkog svijeta, koje dosad nije imao. Time se predmet podiže na jednu višu razinu, gdje pruža dodatne funkcionalnosti u usporedbi s klasičnim predmetom. U našem primjeru prozor bi predstavljao jedan klasični predmet iz stvarnog svijeta, koji sam po sebi nije pametan. Uloga prozora je da omogući ulazak svjetlosti u kuću te protok zraka po potrebi, u slučaju ako je prozor otvoren. Pametni prozor, s druge strane, uz pomoć senzora prikuplja podatke i ovisno o prikupljenim podacima, može izvršavati određene radnje koje klasični prozor ne posjeduje. Na primjer, kada navečer padne mrak, prozor može detektirati nedostatak svjetlosti izvan kuće te prema tome zaključiti da mora spustiti zastore, ili recimo ukoliko je prozor cijeli dan otvoren navečer će se automatski zatvoriti, zato jer je uvečer temperatura zraka dosta niska, te ne bi bilo poželjno da se prostorija kuće jako ohladi. U našem slučaju, pametni prozor imat će mogućnost slanja obavijesti korisniku o potrebi za ručnim otvaranjem ili zatvaranjem prozora. Kako je već i prije navedeno, predmet da bude „pametan“ mora prikupljati podatke iz okoline, što je zadaća senzora. Pametni prozor okružen je senzorima za mjerjenje kvalitete zraka te senzorom udaljenosti koji daje informacije o tome je li prozor otvoren ili nije. Svi oni zajedno čine jednu, naizgled jednostavnu cjelinu povezanih uređaja, međutim u praksi i nije baš tako. Uređaji međusobno moraju komunicirati putem različitih protokola, razmjenjivati podatke i informacije, pamtitи stanja i slično, gdje je krajnji rezultat svega toga poruka obavijesti koju korisnik prima na svojem mobilnom uređaju. Za jednog promatrača, ovaj sustav uređaja može djelovati jednostavno, međutim potrebno je puno znanja i razumijevanja o različitim aspektima djelovanja ovog složenog sustava. Projekt će biti podijeljen u više cjeline, od kojih svaka objašnjava na koji način je implementirana pojedina funkcionalnost pametnog prozora, počevši od prikupljanja podataka sa senzora, pa sve do zajedničke komunikacije uređaja preko oblaka (Clouda) te slanja prikupljenih podataka na Android uređaj korisnika.

2. Modeli

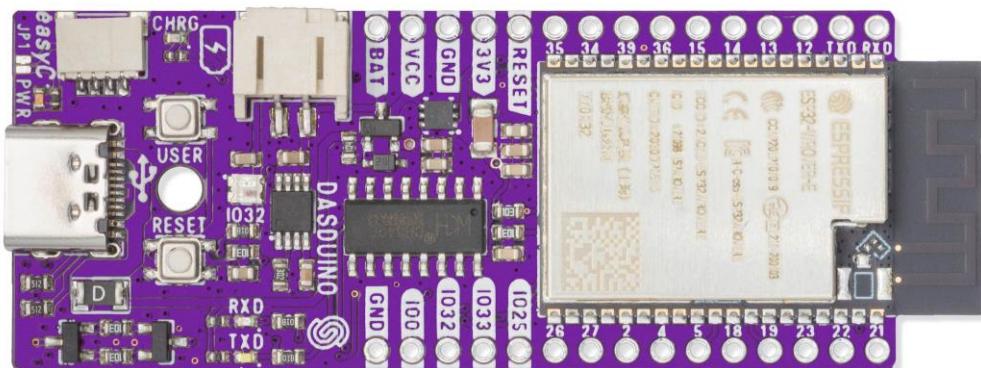
Pojam modela u tehničkom svijetu označava pojednostavljenu stvarnost. Model prikazuje pojednostavljenu komponentu sustava / cjeline, koja je sastavni dio promatranog predmeta (objekta). Uloga modela je da se na jednostavan način prikaže i dobije razumijevanje o promatranom objektu, njegovom sastavu, konstrukciji i funkcionalnostima. U ovome projektu, model projekta će biti prikazan kroz općenitu arhitekturu sustava, komunikacijske dijagrame, dijagrame aktivnosti te shemu spajanja senzora udaljenosti.

2.1. Općenita arhitektura sustava

Općenita arhitektura sustava sastavljena je od fizički prisutnih komponenti te onih komponenti koje su dostupne u oblaku (cloud). Fizički prisutne komponente, koje smo trebali spojiti jesu Dasduino CONNECTPLUS pločica, senzor udaljenosti te „prozor“. Komponente koje su već spojene i samostalno funkcioniraju kao zasebna cjelina (ali su izvan našeg fizičkog doseg), jesu senzori za kvalitetu zraka, koji se nalaze u uredu u Dublinu u Irskoj. Fizički nevidljiva komponenta i poveznica svih navedenih uređaja ove arhitekture je Microsoft Azure Cloud. Putem ove komponente uređaji i senzori razmjenjuju podatke, informacije i poruke te time zajedno čine funkcionalnu sredinu. Microsoft Azure nudi mogućnosti zapisa podataka u baze podataka, izradu funkcija i okidača te daje mogućnost vizualnog prikaza podataka putem grafikona. Osim obrade i prikaza podataka, Microsoft Azure upravlja slanjem obavijesti na mobitel putem Android aplikacije. Ukratko rečeno, Microsoft Azure Cloud predstavlja središte arhitekture, koji povezuje sve uređaje i senzore, obrađuje podatke i šalje odgovarajuće obavijesti korisnicima Android aplikacije. Slika arhitekture sustava biti će prikazana nešto kasnije, ali za početak, potrebno je prvo opisati komponente i objasniti osnovne principe kako funkcioniraju.

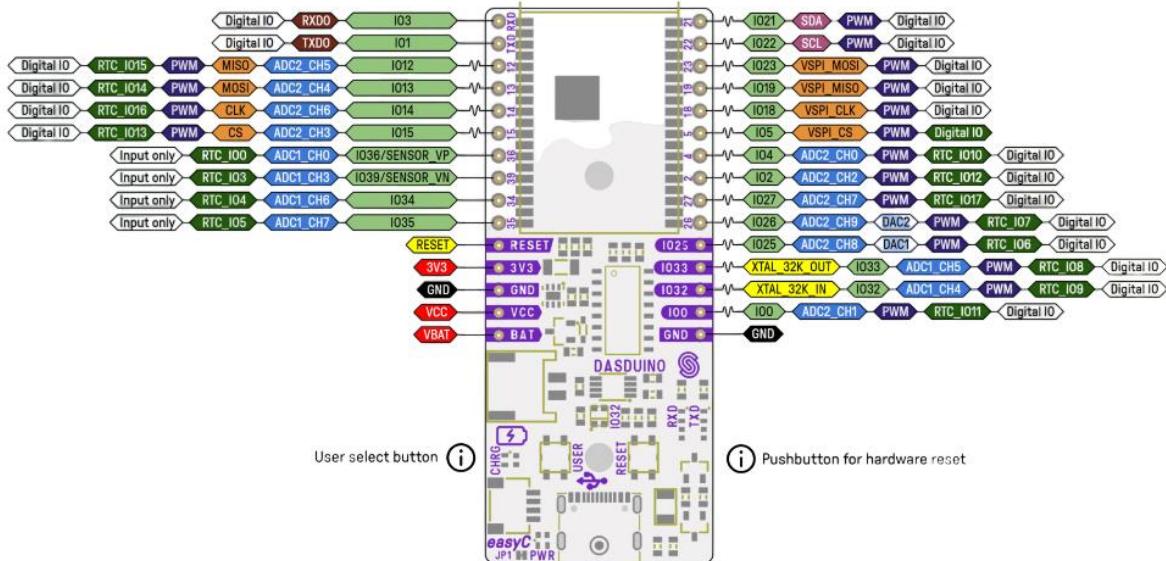
2.1.1. Dasduino CONNECTPLUS (ESP32)

Dasduino CONNECTPLUS pločica je električka pločica razvijena od hrvatske kompanije Soldered Electronics. Soldered Electronics nalazi se u Osijeku, a bavi se razvojem, dizajniranjem te proizvodnjom vlastitih električkih proizvoda. Jedan od njihovih proizvoda jest Dasduino pločica. Postoje različite vrste pločica Dasduina, koje se razlikuju po čipu koji koriste i električkoj shemi. U sljedećem dijelu će biti opisana pločica korištena u ovome radu - Dasduino CONNECTPLUS (ESP32).



Slika 1: Dasduino CONNECT PLUS

Dasduino CONNECTPLUS pločica koristi mikročip ESP32-WROVER-E, te koristi napon rada 3.3V (integrirani 5V regulator). Pločica se sastoji od 30 pinova, od kojih neki rade s digitalnim signalima, analognim te onih pinova koji imaju mogućnost upravljanja širinom (duljinom) impulsa (PWM). Na sljedećoj slici [Slika 2] je prikazana Dasduino CONNECTPLUS pločica skupa sa svojima konekcijama.

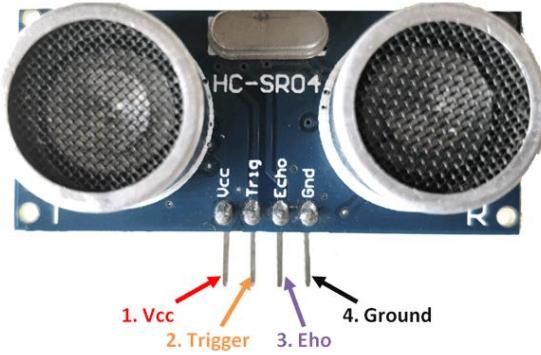


Slika 2: Dasduino CONNECTPLUS (ESP32) - Prikaz konekcija

Na prikazu konekcija se može uočiti da većina pinova sadrži opciju upravljanja širinom impulsa (PWM – Pulse Width Modulation), a samim time, ti pinovi se mogu koristiti za rad s analognim signalima. Skoro svi pinovi omogućavaju rad s ulaznim i izlaznim signalima (IO – Input Output).

2.1.2. Senzor udaljenosti – HC-SR04 Ultrasonic Sensor

Senzor udaljenosti ili ultrazvučni senzor je senzor za mjerjenje udaljenosti pomoću ultrazvučnih valova. Glava senzora emitira ultrazvučni val koji se odbija od nekog predmeta ili objekta te se nazad vraća do glave senzora. Ultrazvučni senzor mjeri udaljenost objekta mjeranjem vremena između emisije i prijema, odnosno između slanja i primanja signala. Za ispravan rad senzora udaljenosti potrebno je pravilno spojiti četiri pina samog senzora. Vcc pin služi za napajanje senzora, a najčešće je to 5 V. Trigger pin se koristi za slanje signala, dok se Echo pin koristi za primanje signala. Ground pin se, kao i obično, spaja na minus to jest uzemljenje, a služi za zatvaranje strujnog kruga. Izgled senzora udaljenosti i njegovih pinova prikazan je na sljedećoj slici [Slika 3].



Slika 3: HC-SR04 Ultrasonic Sensor - Prikaz pinova

HC-SR04 ultrazvučni senzor udaljenosti, u teoriji, može izmjeriti udaljenost između 2 cm i 450 cm, dok je praksi ta udaljenost puno manja – od 2 cm do 80 cm. Senzor računa udaljenost s preciznošću od 3 mm. Za izračun udaljenosti između objekta i senzora koristi se formula:

$$L = \frac{1}{2} * T * C$$

gdje L predstavlja udaljenost, T vrijeme proteklo između slanja i primanja signala, te C koji predstavlja brzinu zvuka. Umnožak se množi s $\frac{1}{2}$ kako bi se dobila vrijednost udaljenosti za jedan smjer (a ne za oba). Vrijednost koja se uzima za brzinu zvuka je 330m/s.



Slika 4: princip rada ultrazvučnog senzora

2.1.3. Senzori kvalitete zraka i Smart Citizen

Senzori kvalitete zraka, kao što im i samo ime govori, služe za mjerjenje raznih parametara kvalitete zraka. U te parametre ulaze podaci o temperaturi, tlaku zraka, vlažnosti, koncentracija štetnih čestica u zraku, količina ugljikovog dioksida u zraku, te količina buke i svjetlosti u zraku / prostoriji gdje se senzor nalazi. Ovi senzori su dio projekta SmartCitizen (Pametni građani) koji imaju za cilj omogućiti širokoj populaciji ljudi praćenje raznih parametra kvalitete okoliša u svrhu građanske, obrazovne ili znanstvene aktivnosti. Glavna ideja projekta SmartCitizen je da očitanja senzora budu svima dostupna diljem svijeta, te da se postoji mogućnost kupnje i ugradnje senzora u vlastitom okruženju, tvoreći tako jedan složeni sustav povezanih uređaja. Moguće je pristupiti tim uređajima i dobiti uvid u njihova očitanja putem internetske stranice Smart Citizen [3].



Slika 6: primjer senzora kvalitete zraka



Slika 5: instalacija senzora za kvalitetu zraka



Slika 7: Smart Citizen - karta umreženih uređaja u Evropi

Nakon što se uređaj ugradi i instalira potrebno ga je spojiti na mrežu, kako bi bio vidljiv na internetskoj stranici te mogao normalno komunicirati s ostalim uređajima. Detaljne upute o tome kako se ugrađuje, instalira i spaja sam uređaj dostupne su na internetu. Također, mogu se naći različite varijante uređaja za mjerjenje kvalitete zraka, koji nama trenutno nisu bitni, već je dovoljno da uređaj ima ugrađeni senzor za mjerjenje razine CO₂ (ugljikovog dioksida) u zraku. Senzor s kojeg mi očitavamo vrijednosti je tipa AMS CCS811.

Measurement	Units	Sensors
Air temperature	°C	Sensirion SHT-31
Relative Humidity	% REL	Sensirion SHT-31
Noise level	dBA	Invensense ICS-434342
Ambient light	Lux	Rohm BH1721FVC
Barometric pressure	Pa	NXP MPL3115A26
Equivalent Carbon Dioxide	ppm	AMS CCS811
Volatile Organic Compounds	ppb	AMS CCS811
Particulate Matter PM 1 / 2.5 / 10	µg/m ³	Planttower PMS 5003

Slika 8: vrste senzora ugrađenih u uređaj za mjerjenje kvalitete zraka

2.1.4. Microsoft Azure Cloud

Microsoft Azure je inicijalno razvijen kao PaaS (Platform as a Service), dok danas pruža preko 200 različitih servisa u oblaku (Cloudu), klasificirajući se tako kao SaaS (Service as a Service) i IaaS (Infrastructure as a Service). Microsoft Azure ili kraće, samo Azure, je zadužen za prikupljanje podataka uređaja i senzora te spremanje tih podataka na Cloud. Cloud predstavlja prostor za pohranu podataka putem interneta. Podaci mogu biti „sirovi“ (dohvaćeni s uređaja), a mogu biti i obrađeni i analizirani, koji se prikazuju putem različitih grafikona, izvještaja, sortiranih lista i slično. Azure je fizički nevidljiva komponenta, međutim on je ujedno i najbitnija komponenta IoT sustava jer se preko njega ostvaruje gotovo sva komunikacija između pametnih uređaja. Azure svojim servisima nudi korisniku različite mogućnosti dohvaćanja, pohrane i slanja podataka putem različitih komunikacijskih protokola.



Slika 9: Microsoft Azure logo i prikaz ikona nekih od servisa koje nudi

Ono po čemu je Azure zasigurno poznat jest po složenosti i širokom rasponu mogućnosti koje nudi. U Azuru postoji mogućnost kreiranja baze podataka u koju se zapisuju podaci očitani sa senzora. Moguće je napraviti vlastiti server na kome će se izvoditi više različitih baza podataka. Zatim, ukoliko se iz očitanih podataka sa senzora želi dohvatiti samo specifičan podatak, dohvatit će se putem servisa Logic App ili putem servisa Function App. Ukoliko se želi komunicirati s uređajem koji je povezan na Azure, može se putem IoT Huba. Moguće je prikazati i virtualnu repliku stvarnog uređaja u virtualnom okruženju – primjer digitalnog blizanca (Digital Twin Platform). Uglavnom, servisa ima mnogo, a u dalnjem tekstu će kratko biti opisani samo oni koji su korišteni pri izradi ovog rada.

Logic App servis služi za izradu logike rada s podacima, a glavna mu je karakteristika da je low-code servis što u prijevodu znači da ima malo kodiranja i pisanja koda, već se bazira na vizualnom prikazu i sastavljanju koda. Koristi se za dohvaćanje podataka sa Smart Citizen API stranica senzora.

SQL Database servis služi za pohranu podataka, koji se kasnije mogu čitati, koristiti se za donošenje odluka, provođenje analize podataka i slično.

Function App servis predstavlja glavnu logiku zaduženu za ispravno izvođenje, dohvaćanje i uspostavu komunikacije među uređajima i drugim Azure servisima. Function App je vrlo slična servis kao i Logic App samo što on zahtjeva pisanje koda, te nije vizualno interaktivn kao i Logic App. Function App se može koristiti u svrhu postavljanja okidača, izvođenje funkcija po ispunjenu nekog od uvjeta ili pak za druge potrebe.

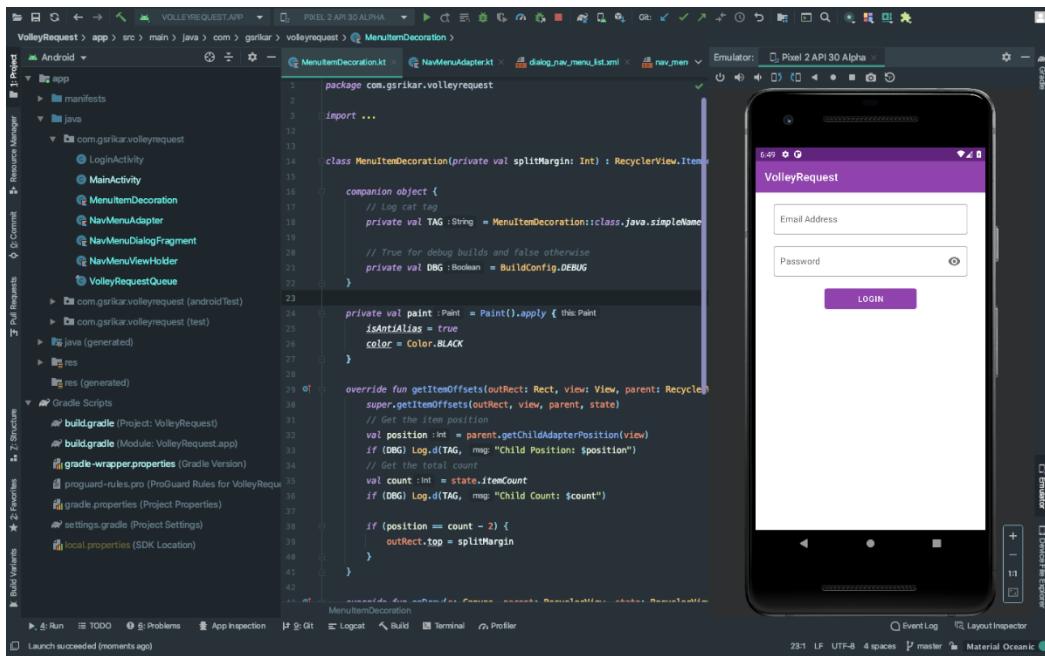
IoT Hub servis je poveznica između uređaja i Function App servisa. Dakle, bilo koji IoT uređaj spojen na IoT Hub, šalje poruke preko IoT Hub koji nadalje proslijeđuje te poruke preko Function App servisa te ih tada Function App može obrađivati i izvršavati definirane radnje. IoT Hub u suštini omogućuje jednostavnije i sigurnije povezivanje s ostalim Azure servisima.

2.1.5. Android Studio

Android mobilnu aplikaciju moguće je izraditi pomoću razvojnog okruženja Android Studio. Android Studio koristi dva programska jezika za programiranje same aplikacije, a to su Java, ili noviji Kotlin. Kotlin je ustvari nastao kao pojednostavljena zamjena za Javu. Iako je Kotlin noviji jezik, Android Studio nudi mogućnost pisanja i Java i Kotlin koda u istom projektu, dakle moguće je pri izradi jedne aplikacije koristiti dva različita programska jezika. Android Studio pruža sve potrebno za izradu mobilnih aplikacija, od dodavanja običnog teksta i naslova, do kreiranja gumbova, prikaza slika, slanja i primanja obavijesti, kreiranje grafikona i još mnogih drugih opcija. Vrlo korisna funkcionalnost jest da se uz pisanje samog koda, aplikacija može i testirati na pravom Android uređaju. Uređaj je potrebno povezati putem kabla i skinuti potrebni softver za rad s Android Studiom te se zatim na zaslonu mobitela prikazuje izrađena aplikacija. Ukoliko korisnik nema Android uređaj, može koristiti virtualni uređaj koji mu Android Studio sam pokreće unutar svoje razvojne okoline. Na taj način programer uvijek može testirati aplikaciju radi li određena funkcionalnost kako treba ili ne.



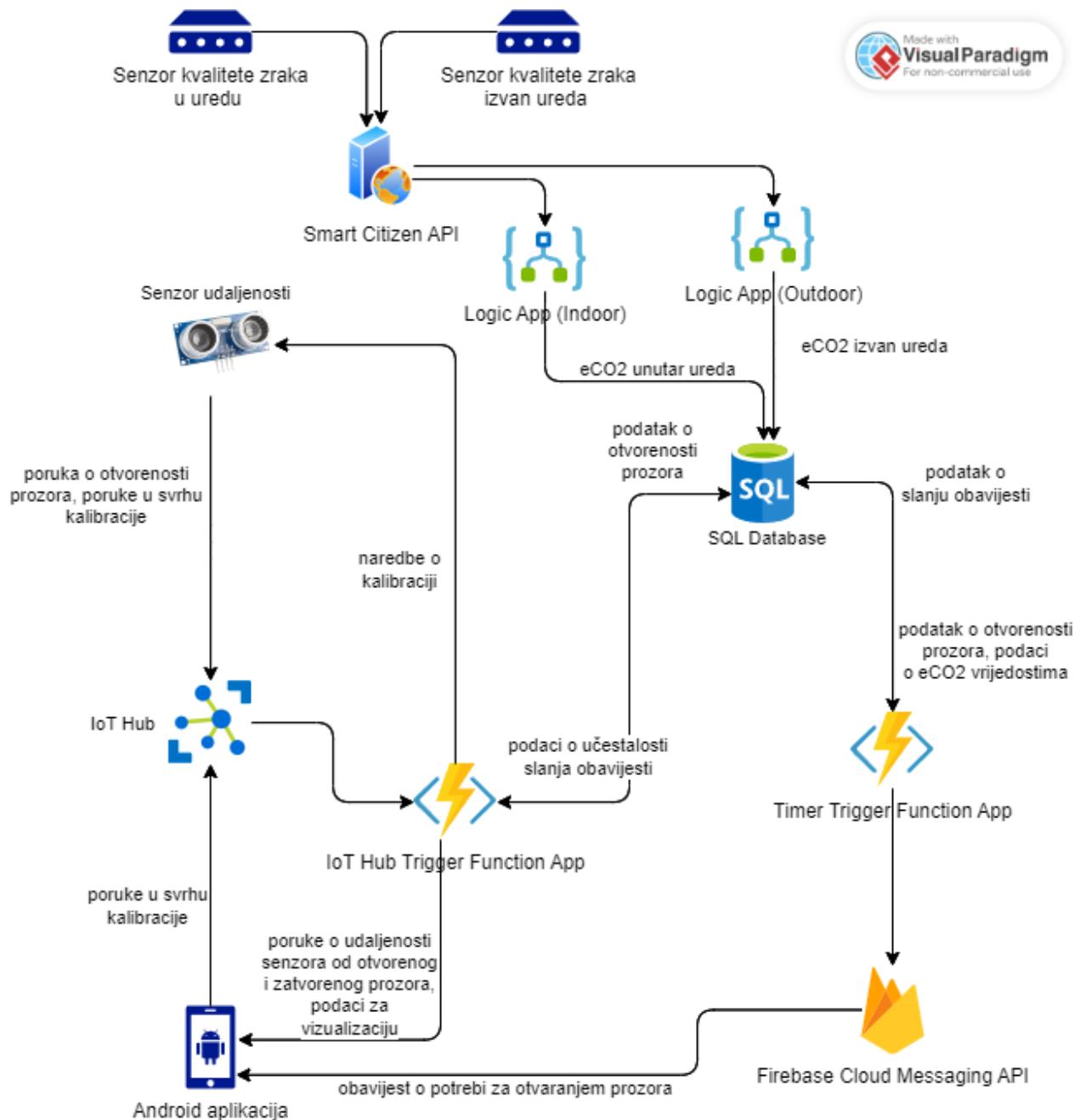
Slika 10: Android Studio logo



Slika 11: Android Studio razvojna okolina

2.2. Složena arhitektura sustava

Složena arhitektura sustava sastavljena je od svih prethodno opisanih uređaja, a pokazuje smjer komunikacije i poruka koja se izmjenjuju između uređaja, Azure servisa te Android aplikacije. Sljedeći dijagram arhitekture sustava izrađen je u online alatu Visual Paradigm Online [6].



Slika 12: složena arhitektura sustava Smart Window

Ikone na prikazanom dijagramu mogu se podijeliti u dvije kategorije, ikone Microsoft Azure Cloud servisa i sve ostale ikone koje prikazuju povezane uređaje. Povezanih uređaja nema puno, to su senzori kvalitete zraka u uredu i izvan ureda, senzor udaljenosti te Android uređaj na kojem se izvodi Android aplikacija. Ti uređaji su fizički prisutni te predstavljaju krajnji dio arhitekture sustava. Unutarnju arhitekturu sustava čini Azure Cloud servisi i to dva Logic App servisa, IoT Hub, dva Function App servisa, i SQL Database servis.

Senzori za kvalitetu zraka neprestano mjere kvalitetu zraka u uredu koji se nalazi u Irskoj. Podatke koje prikupe šalju na Smart Citizen API stranicu u JSON formatu poruke. Na navedenoj stranici se ti podaci zapisuju, međutim prikazani su u vrlo nečitljivom formatu, stoga ih je potrebno pretvoriti u razumljiviji način prikaza podataka. Ovu funkciju obavljaju Logic App servisi, konkretno, skupljaju podatke o koncentriranosti eCO2 u zraku. Podatak o navedenom parametru dobivaju šaljući POST zahtjev na Smart Citizen API servis svaku minutu.

Jednom kada Logic App servisi dohvate traženu vrijednost pohranjuju ju u SQL Database servis. SQL Database se sastoji od dvije baze podataka, jedna za prikupljanje vrijednosti kvalitete zraka, a druga zadužena za pohranjivanje vrijednosti je li prozor otvoren ili zatvoren.

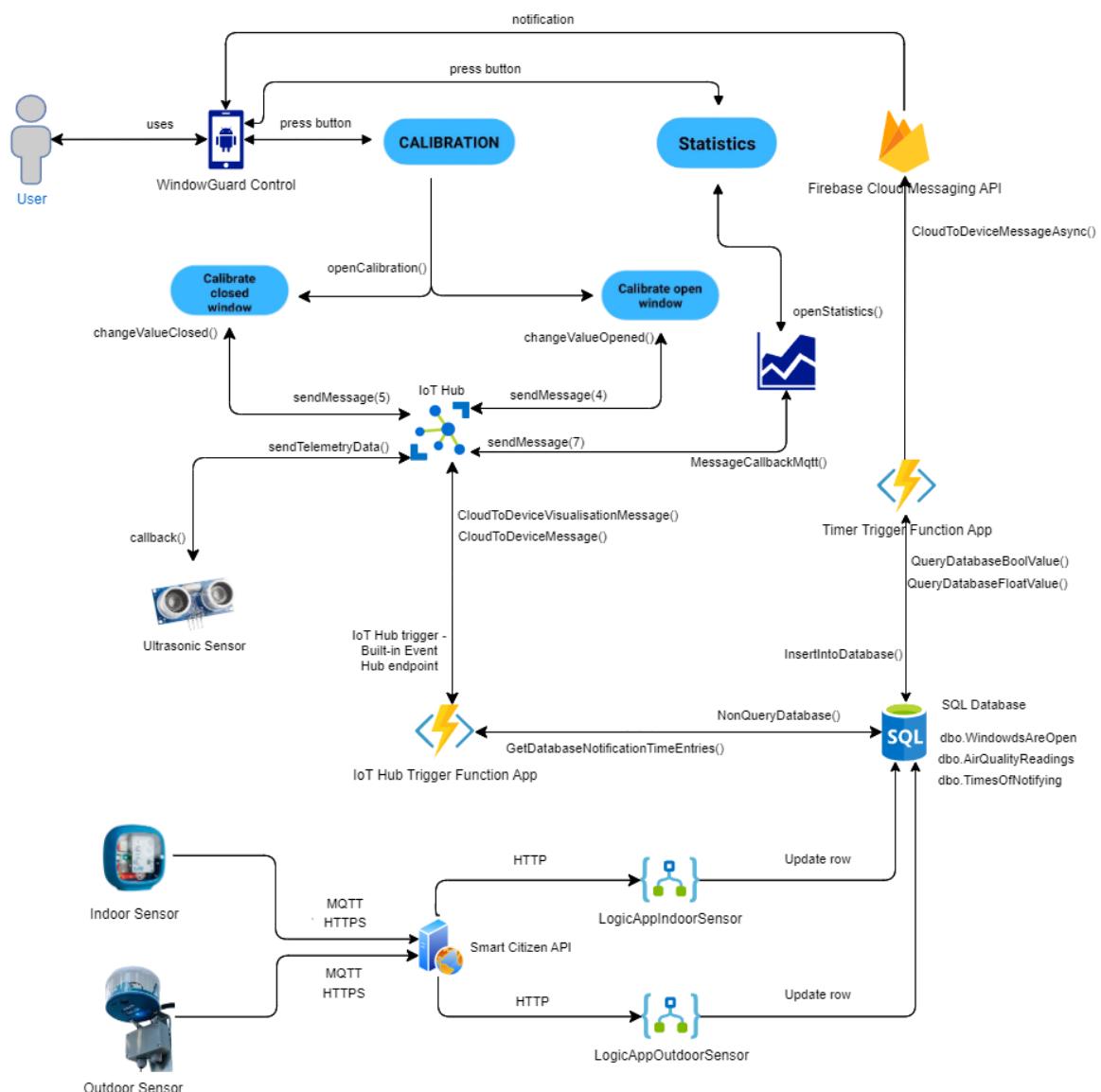
Praćenje otvorenosti prozora omogućuje senzor udaljenosti. Senzor šalje podatke o udaljenosti od prozora te se ti podaci dohvaćaju preko IoT Hub servisa. IoT Hub servis služi kao poveznica između senzora udaljenosti i Function App servisa i Android uređaja. Drugim riječima, on omogućava dvosmjernu komunikaciju između uređaja i Cloud-a. Još jedna od uloga senzora udaljenosti jest to da ima mogućnost kalibracije. Kalibracija funkcioniра na način da senzor izmjeri udaljenost do otvorenog prozora, i udaljenost do zatvorenog prozora, pritom spremajući ih u posebne varijable. Nadalje, preko Function App logike se zapisuje podatak u bazu o tome je li prozor otvoren ili zatvoren na temelju dohvaćenih vrijednosti.

Android uređaj služi za slanje naredbi o kalibraciji senzora, te za primanje različitih obavijesti i poruka putem Function App servisa.

Povezanost svih ovih dijelova arhitekture ostvarena je pomoću Function App servisa. Oni izvode različite funkcije i okidače, upravljaju komunikacijom i prosleđuju poruke i podatke. IoT Hub Trigger Function App se aktivira svaki put kada se pošalje device-to-cloud poruka na IoT Hub. On u bazu prosleđuje poruku je li prozor otvoren ili ne. Timer Trigger Function App se aktivira svakih 5 minuta. Njegova svrha je provjeravanje je li nužno poslati obavijest preko Firebase Cloud Messaging API servisa na Android uređaj. Ako je, onda pošalje obavijest. On iz baze podataka čita podatke je li prozor već otvoren i uspoređuje eCO2 koncentraciju unutar i izvan ureda. Firebase Cloud Messaging API servis je zamjena za Notification Hub servis koji nažalost, uz niz pokušaja nije radio.

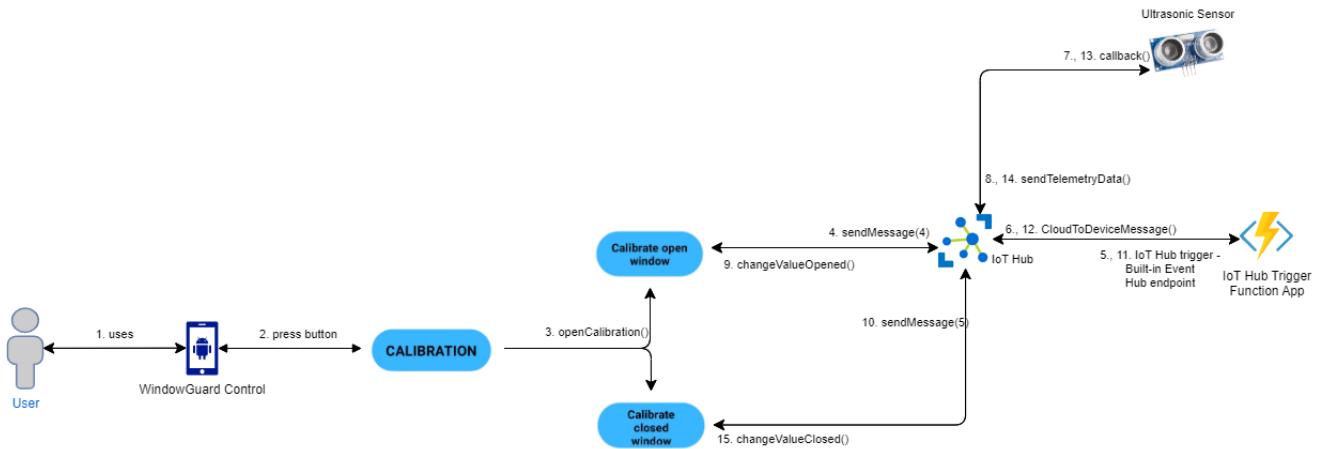
2.3. Komunikacijski dijagrami

Komunikacijski dijagrami su vrsta dijagrama koji prikazuju komunikaciju između različitih objekata. Cilj ovih dijagrama nije prikazivanje složene strukture, već da se iz slike može iščitati koji objekt komunicira s kojim i na koji način je ta komunikacija ostvarena. Na slijedećoj slici 13 je prikazan komunikacijski dijagram cijele arhitekture sustava, dok će u nastavku biti prikazani pojedinačni dijelovi ovog dijagrama sa popratnim rednim brojevima operacija kako bi se jednostavnije moglo ukazati na bitne dijelove.



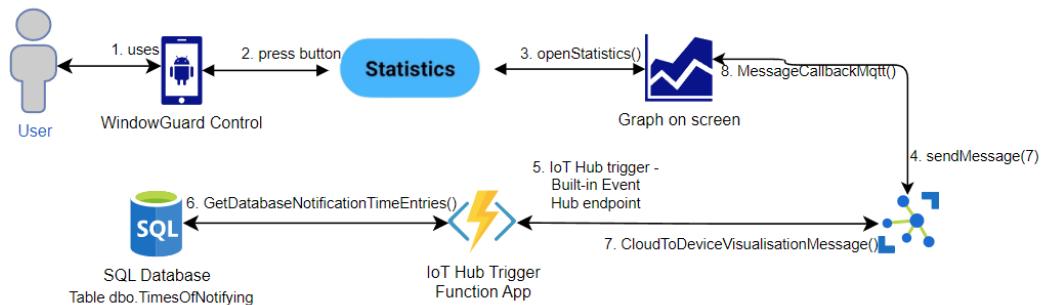
Slika 13: komunikacijski dijagram cijele arhitekture sustava

Prvi pojednostavljeni dijagram prikazuje komunikaciju između korisnika i aplikacije, dok aplikacija u pozadini komunicira s nizom Azure servisa. Korisnik se služi mobitelom te pritiskom na gumb Calibration mu se otvara novi zaslon s mogućnostima kalibracije otvorenog i zatvorenog prozora. Prilikom odabira jedne od ponuđenih opcija, uspostavlja se komunikacija s IoT Hub servisom, koji nadalje posreduje u komunikaciji između Function App servisa, senzora udaljenosti i sučelja aplikacije koju korisnik koristi. Function App servis komunicira s senzorom udaljenosti tražeći ga da izmjeri udaljenost i pošalje vrijednost. Očitana vrijednost se šalje preko IoT Hub servisa i preko funkcija Function App servisa te se pohranjuje u SQL bazu podataka. Nadalje, Function App servis po potrebi traži vrijednosti zapisane u bazu podataka te ih prikazuje na zaslonu aplikacije, gdje se vrijednosti izmjerene udaljenosti prikazuju korisniku.



Slika 14: komunikacijski dijagram - Kalibracija prozora

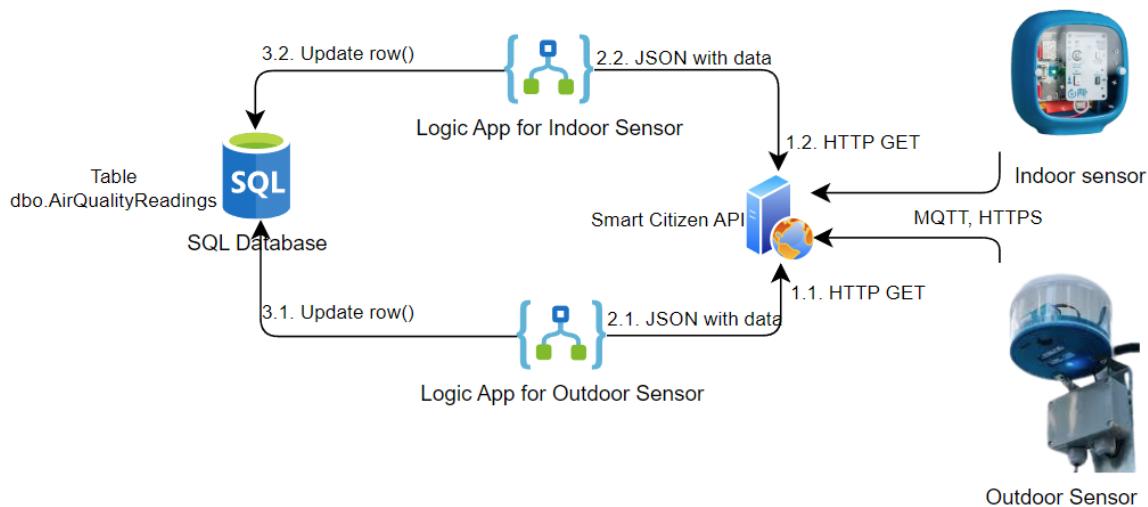
Sljedeći dijagram prikazuje komunikaciju koja se odvija kada korisnik aplikacije pritisne gumb Statistics. Isto kao i u prošlom dijagramu, korisnik interaktivno komunicira s aplikacijom, samo što ovaj put koristi opciju statistike, koja prikazuje grafikon poslanih i primljenih poruka obavijesti. Grafikon u aplikaciji komunicira s Function App servisom te od njega traži da dohvati



Slika 15: komunikacijski dijagram - Prikaz grafikona

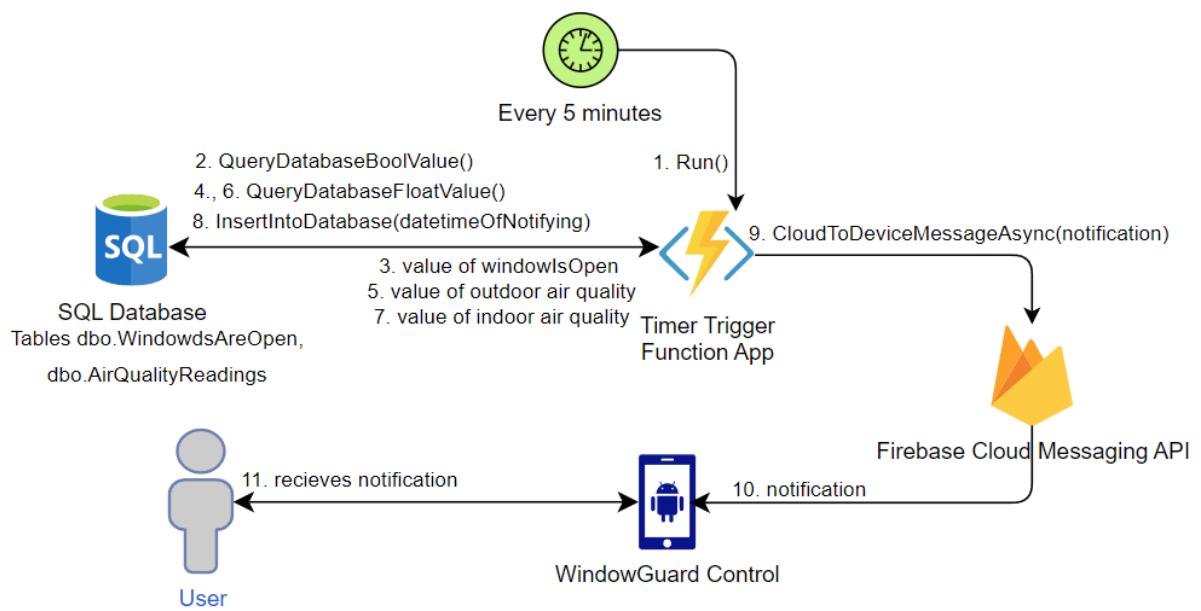
podatke iz SQL baze podataka, te da ih proslijedi nazad u aplikaciju. Ovdje su dohvaćeni podaci vidljivi korisniku u obliku grafa.

Treći komunikacijski dijagram prikazuje proces dohvaćanja podataka sa senzora kvalitete zraka, koji se nalaze unutar i izvan ureda u Dublinu u Irskoj. Oba senzora za komunikaciju sa Smart Citizen API-em koriste MQTT i HTTPS protokole, dok se komunikacija između Smart Citizen API i Logic App servisa na Azuru odvija putem HTTP protokola. Postoje dva LogicApp servisa, gdje jedan dohvata podatke sa senzora unutar ureda, dok drugi dohvata podatke sa senzora izvan ureda. Podatke koje dohvate šalju putem naredbe ažuriranja reda u SQL bazi podataka, gdje se ti podaci i skladište. GET zahtjev se ostvaruje svakih 5 minuta.



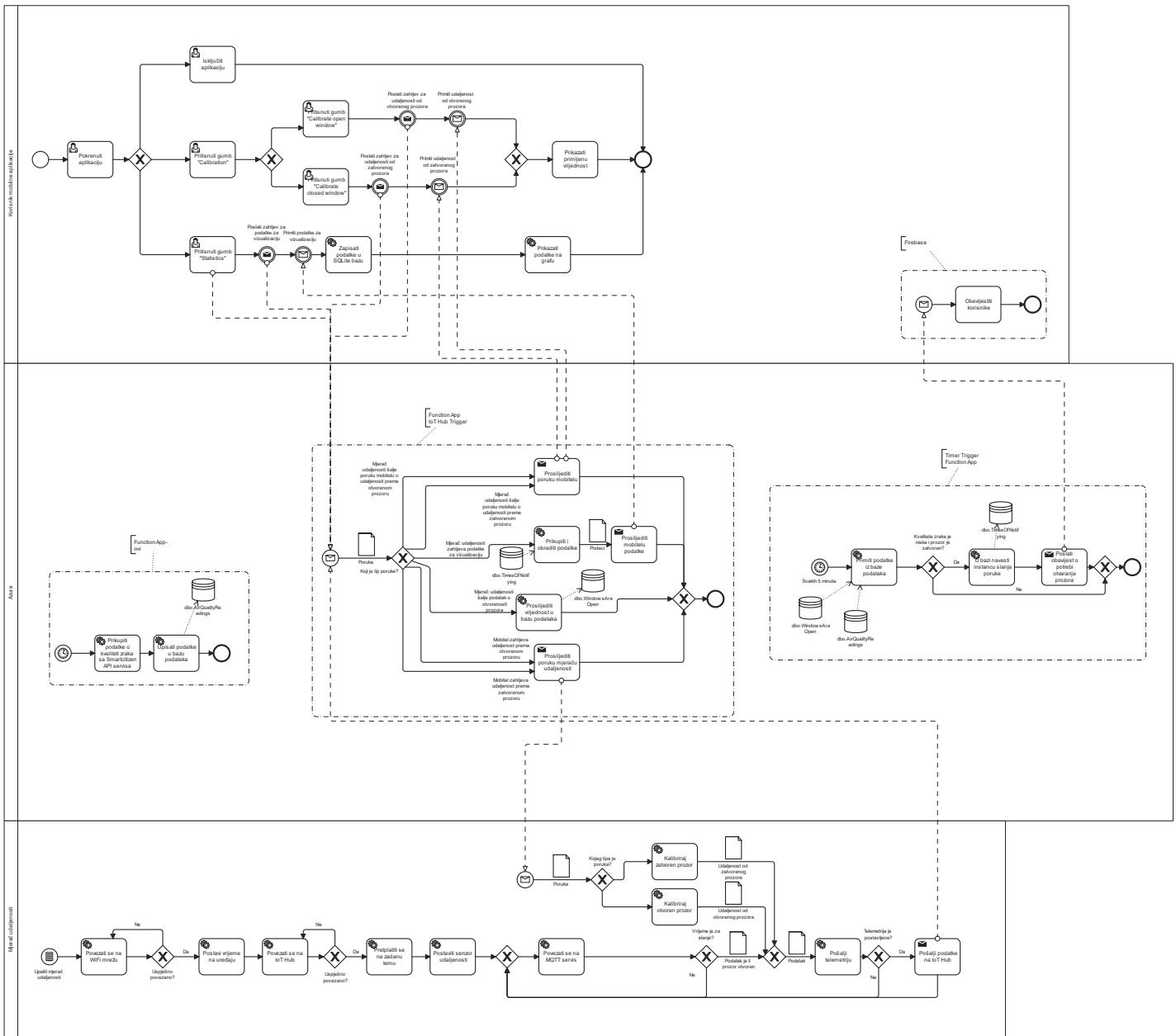
Slika 16: komunikacijski dijagram - Dohvaćanje vrijednosti sa senzora kvalitete zraka

Zadnji dijagram komunikacije pokazuje interakciju između korisnika, aplikacije, SQL baze podataka i njihovih posrednika Function App i Firebase Cloud servisa. Function App dohvata vrijednosti iz baze podataka te ih uspoređuje te ovisno tome odlučuje treba li se poslati obavijest korisniku ili ne. Odnosno, u slučaju da je potrebno otvoriti prozor izvršit će se koraci 8-11. U slučaju da je kvaliteta zraka u prostoriji loša, tada Function App komunicira s Firebase Cloud servisom, koji kreira i šalje obavijest o potrebi za otvaranjem prozora na Android aplikaciju. Android aplikacija vizualno prikazuje korisniku obavijest.



Slika 17: komunikacijski dijagram - Kreiranje obavijesti

2.4. Dijagram aktivnosti



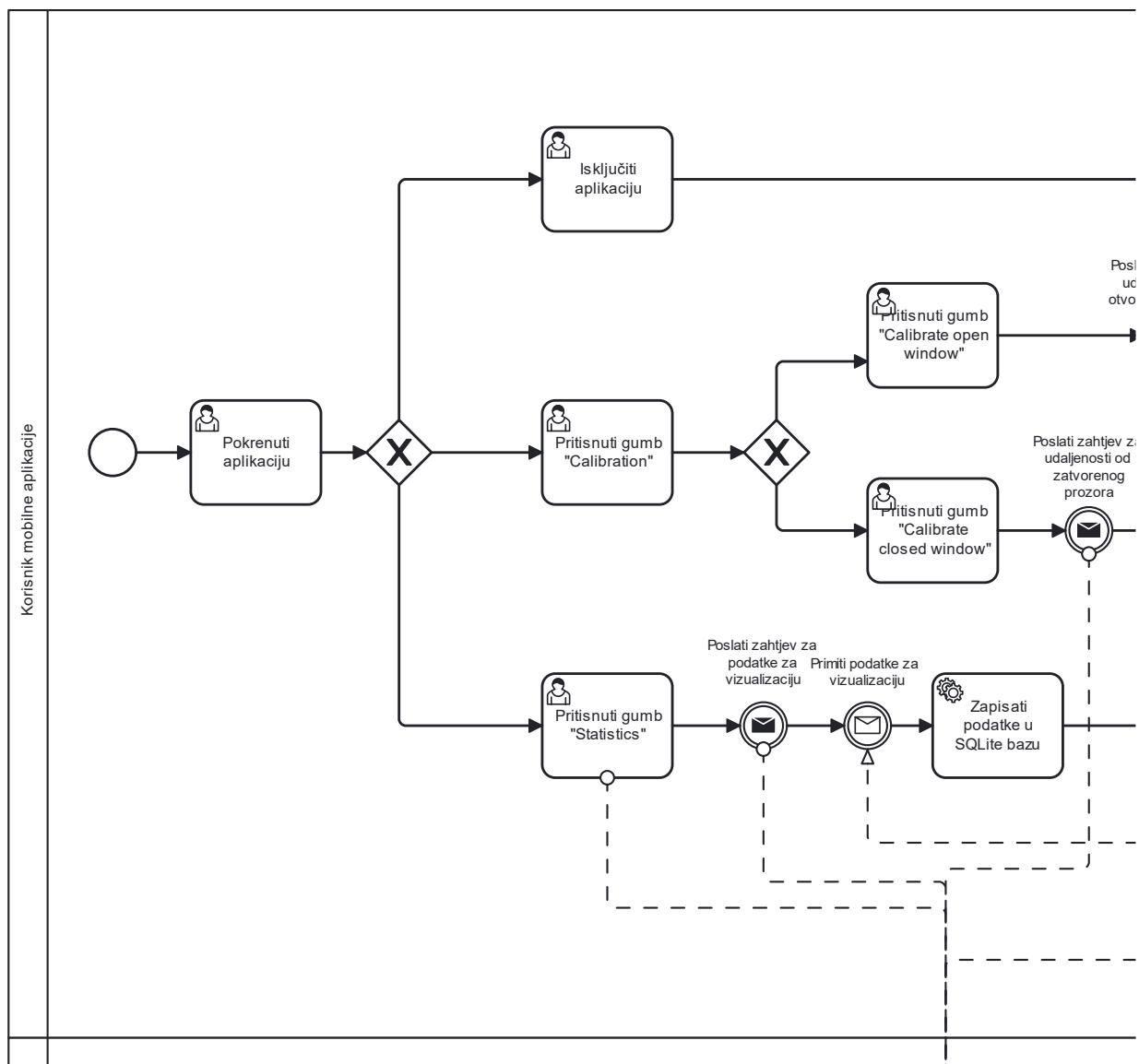
Slika 18: dijagram aktivnosti arhitekture sustava

U našem dijagramu aktivnosti koristit ćemo BPMN (Business Process Model Notation) te za cilj imamo prikazati poslovne procese, učesnike (sudionike), tokove, odluke i aktivnosti. Glavni proces naziva se Obavljanje korisnika o potrebi za otvaranjem prozora. Dijagram aktivnosti sastoji se od jednog bazena (eng. pool) i jedne staze u kojoj je učesnik sam korisnik, a drugi bazu se sastoji od tri staze u kojima su učesnici: senzor kvalitete zraka, Azure i senzor udaljenosti. Na slijedećoj slici [Slika 17] će biti prikazan BPMN dijagram aktivnosti u cijelosti, dok će nakon objašnjenja cijelog dijagrama biti stavljene slike uvećanog dijagrama kako bi se lakše mogle promotriti određene aktivnosti.

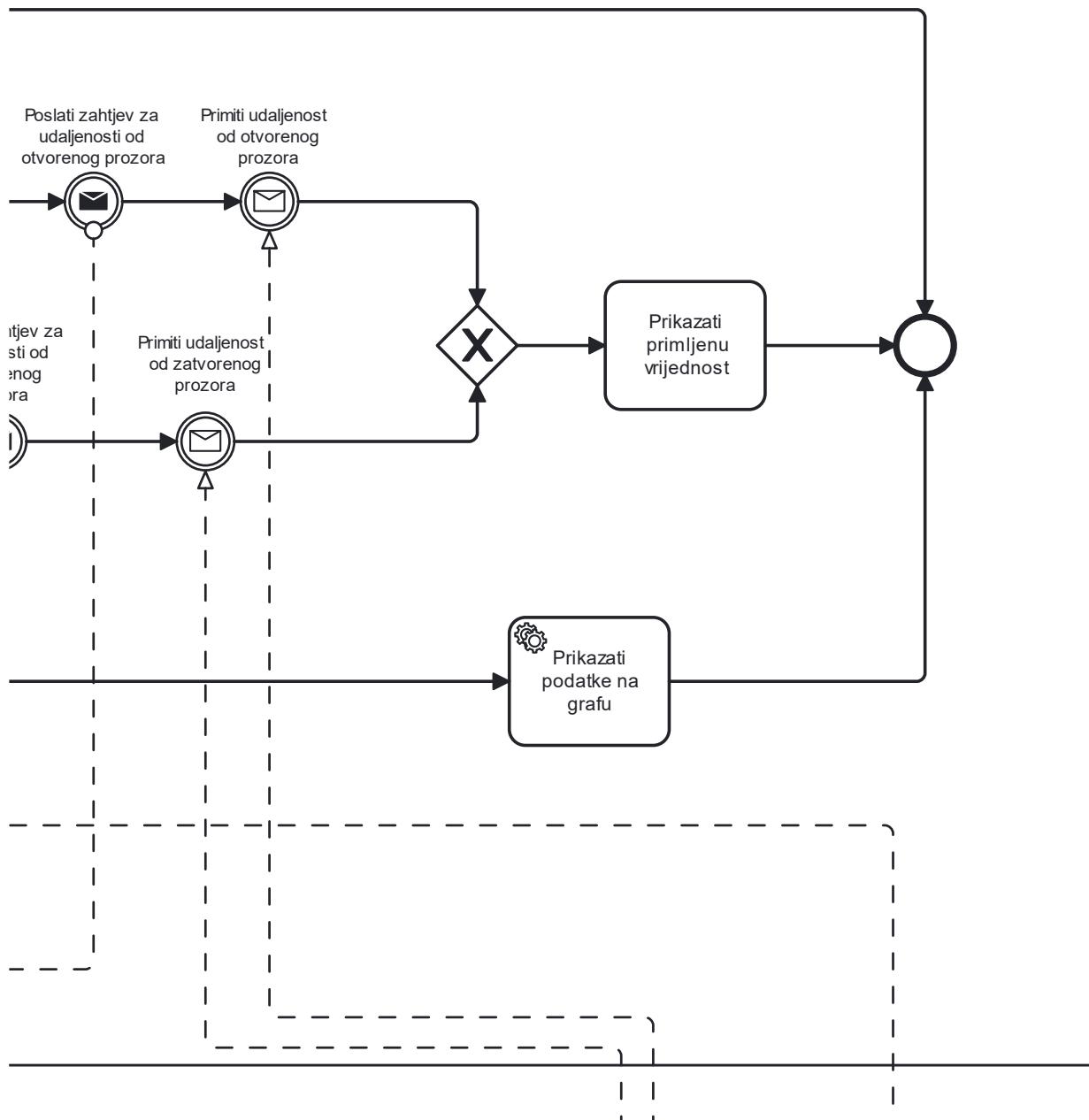
Na samom početku korisnik na svom mobilnom uređaju pokreće aplikaciju. Nakon pokretanja aplikacije korisnik ima tri mogućnosti: Pritisnuti gumb Calibration, pritisnuti gumb Statistics ili isključiti aplikaciju gdje korisnik isključuje aplikaciju i završava proces. Pritiskom na gumb Statistics na mobilnom uređaju, iz Azura se poziva Timer Trigger Function App koji se izvršava svakih pet minuta. Informacije koje će se koristiti u Statisticsi aktivnost Timer Trigger Function App čita iz baze podataka koja se zove TimesOfNotifying u koju se isto preko aktivnosti Timer Trigger Function App podaci o broju poslanih obavijesti korisniku svaki sat zapisuju u bazu. Aktivnost Timer Trigger Function App koja se nalazi u Azure-u šalje statističke vrijednosti korisniku na mobilni uređaj. Nakon što korisnik primi statističke vrijednosti one se korisniku prikazuju i time završava proces. Ukoliko korisnik na mobilnom uređaju pritisne gumb Callibrate open window pokreće se aktivnost Pokrenuti IoT Hub koja se nalazi na Azure-u. Aktivnost Pokrenuti IoT Hub poziva aktivnost Pozvati IoT Hub Trigger Function App koja iz Azure-a poziva senzor udaljenosti. U novoj stazi koja se naziva senzor udaljenosti pomoću aktivnosti Izmjeriti udaljenost otvorenog prozora mjeri se udaljenost. Nakon što se udaljenost izmjeri, ona se šalje korisniku. Kad korisnik primi izmjerenu vrijednost ona se pomoću aktivnosti Prikazati vrijednost udaljenosti prikazuju korisniku na mobilnom uređaju. Ukoliko korisnik na mobilnom uređaju pritisne gumb Callibrate close window pokreće se aktivnost Pokrenuti IoT Hub koja se nalazi na Azure-u. Aktivnost Pokrenuti IoT Hub poziva aktivnost Pozvati IoT Hub Trigger Function App koja iz Azure-a poziva senzor udaljenosti.

U novoj stazi koja se naziva senzor udaljenosti pomoću aktivnosti Izmjeriti udaljenost zatvorenog prozora mjeri se udaljenost. Nakon što se udaljenost izmjeri, ona se šalje korisniku. Kad korisnik primi izmjerenu vrijednost ona se pomoću aktivnosti Prikazati vrijednost udaljenosti prikazuju korisniku na mobilnom uređaju. U stazi koja se naziva Senzor kvalitete zraka mjeri se kvaliteta zraka. Svakih pet minuta aktivnost Izmjeriti kvalitetu zraka u uredu mjeri kvalitetu zraka i pomoću aktivnosti pohranjuje izmjerene vrijednosti u bazu koja se naziva AirQualityReadings. Ista stvar se događa i s mjeranjem kvalitete zraka izvan ureda, te se nakon mjerjenja vrijednosti i pomoću aktivnosti Pohraniti vrijednosti, pohranjuju vrijednosti u AirQualityReadings bazu. Kako bi se korisnika obavještavalo kada je potrebno otvoriti odnosno zatvoriti prozor, zadužena je aktivnost Usporediti kvalitetu zraka. Aktivnost Pozvati Timer

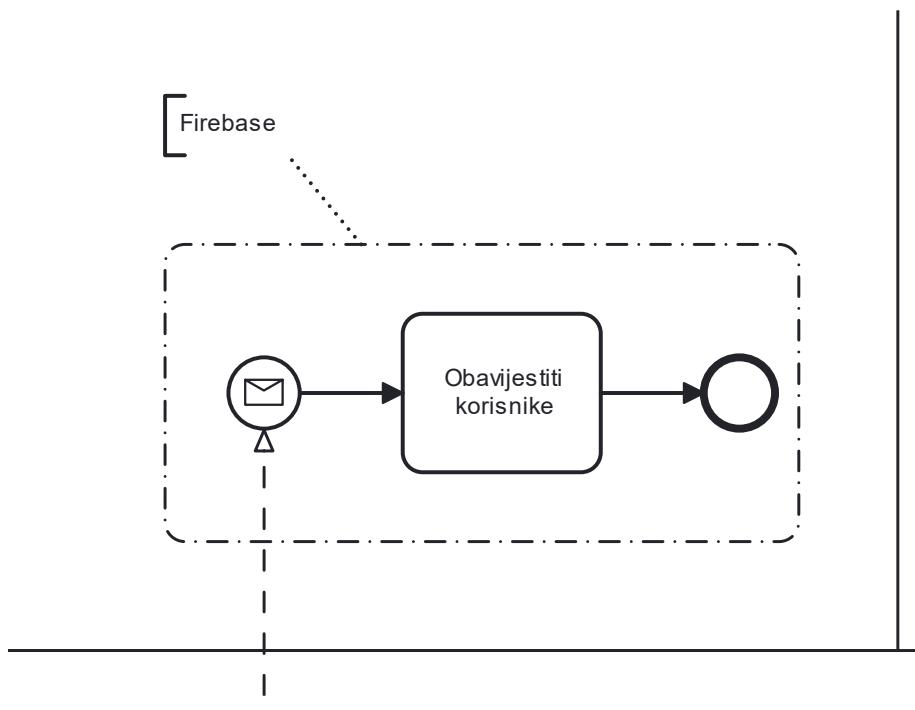
Trigger Function App čita iz baze AirQualityReadings vrijednosti kvalitete zraka u uredu i izvan njega te vrijednosti o otvorenosti, odnosno zatvorenosti prozora iz WindowsAreOpen baze. Pročitane vrijednosti iz baza aktivnost Pozvati Timer Trigger Function App proslijedi aktivnosti usporediti kvalitetu zraka. Ukoliko je kvaliteta zraka izvan ureda veća od kvalitete zraka u uredu i prozor je zatvoren, javlja se korisniku da otvorи prozor. Ukoliko je kvaliteta zraka u uredu veća od kvalitete zraka izvan ureda i prozor je otvoren, javlja se korisniku da zatvori prozor. Aktivnost Usporediti kvalitetu zraka šalje obavijest korisniku koji ju prima. Nakon što korisnik primi obavijest, ista se prikazuje i time završava proces.



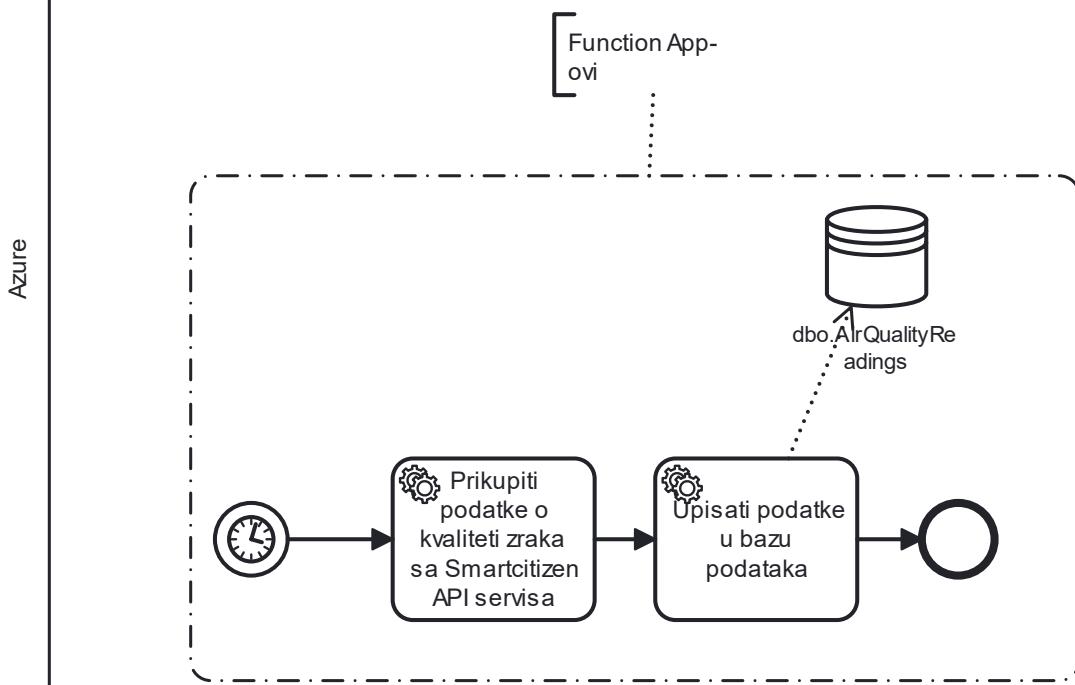
Slika 19: staza korisnika mobilne aplikacije 1/3



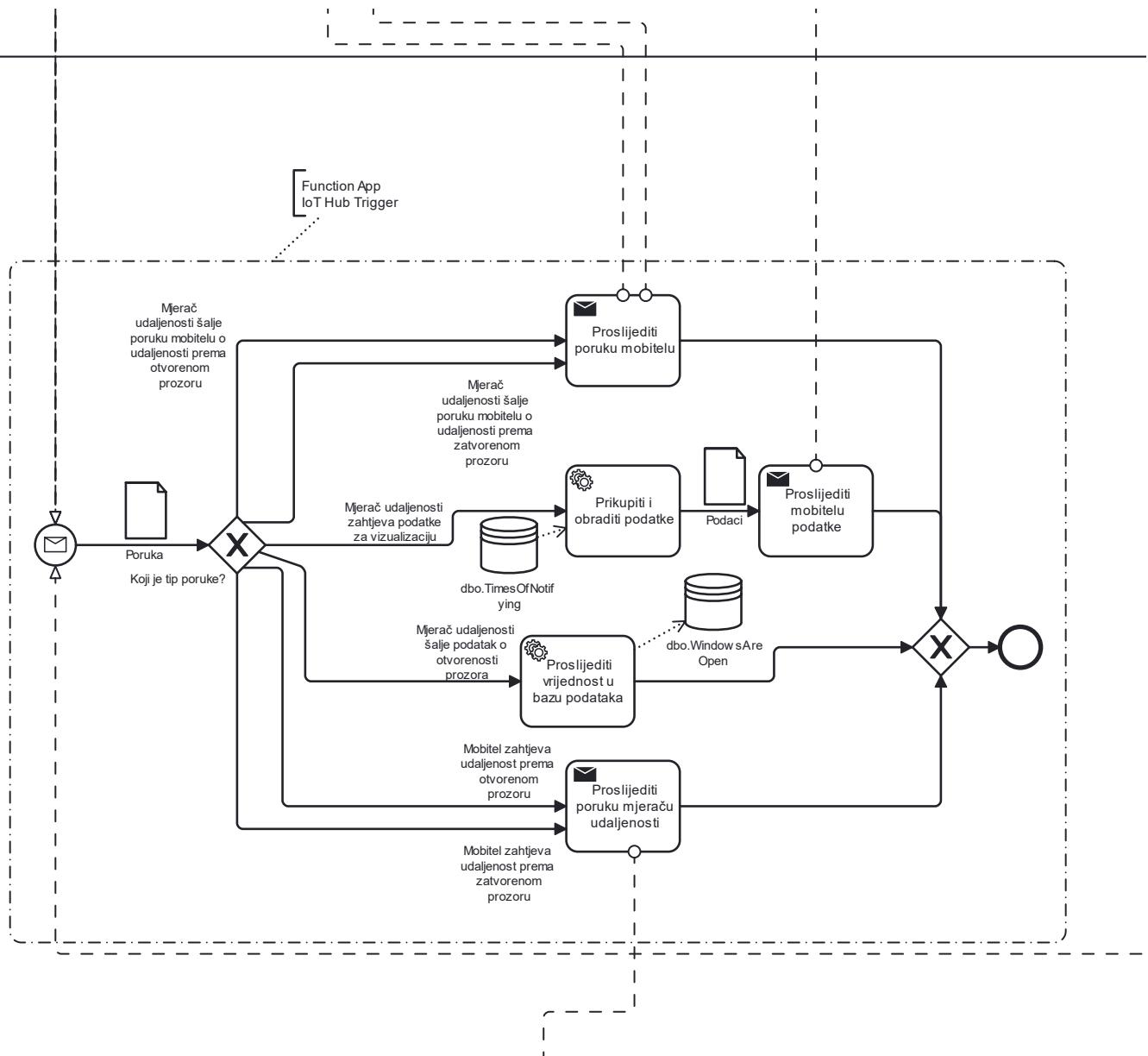
Slika 20: staza korisnika mobilne aplikacije 2/3



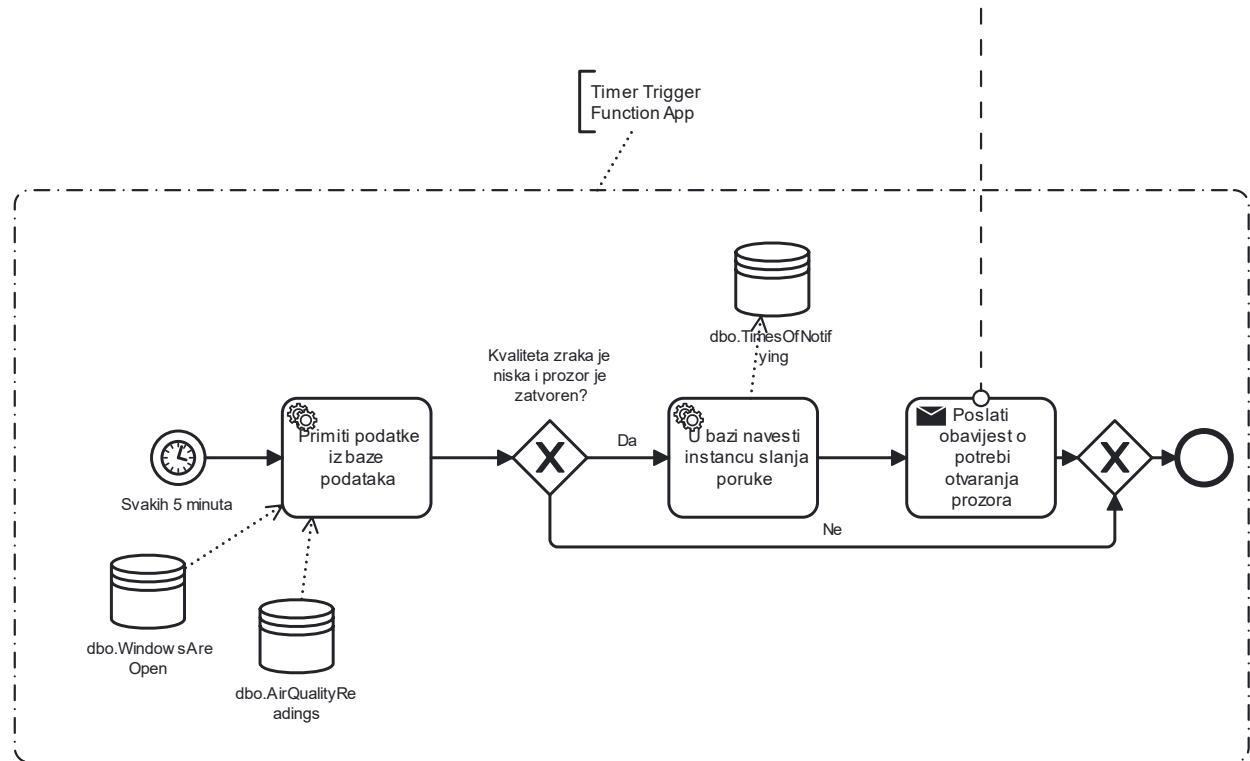
Slika 21: staza korisnika mobilne aplikacije 3/3



Slika 22: staza Azure 1/3

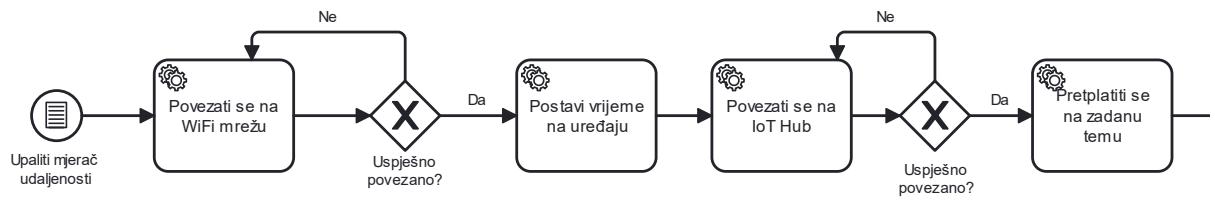


Slika 23: staza Azure 2/3

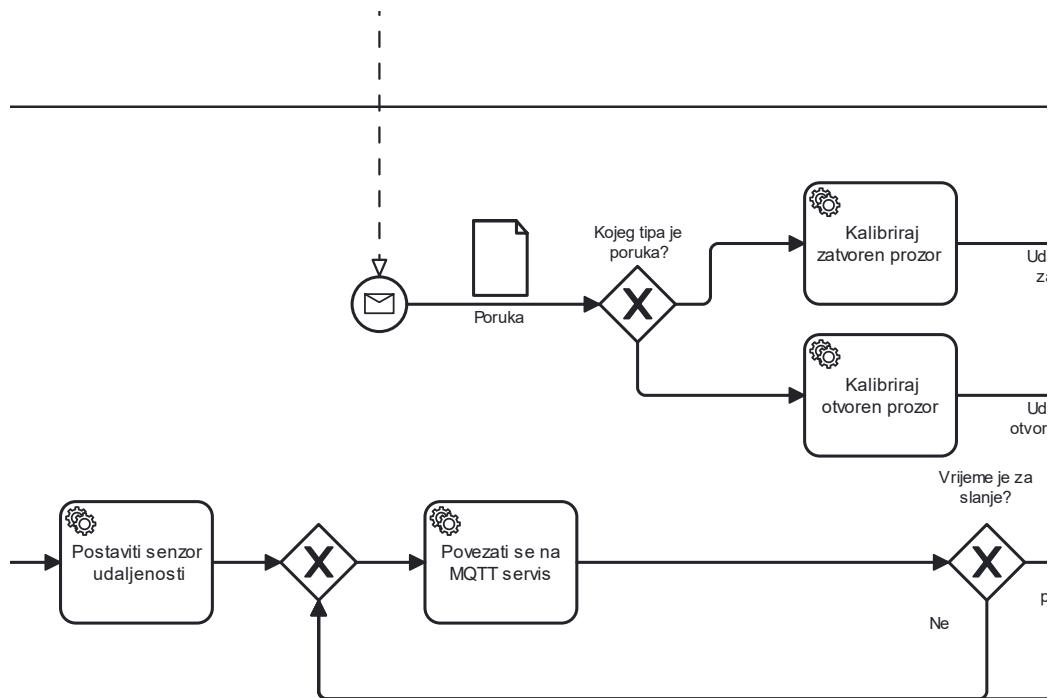


Slika 24: staza Azure 3/3

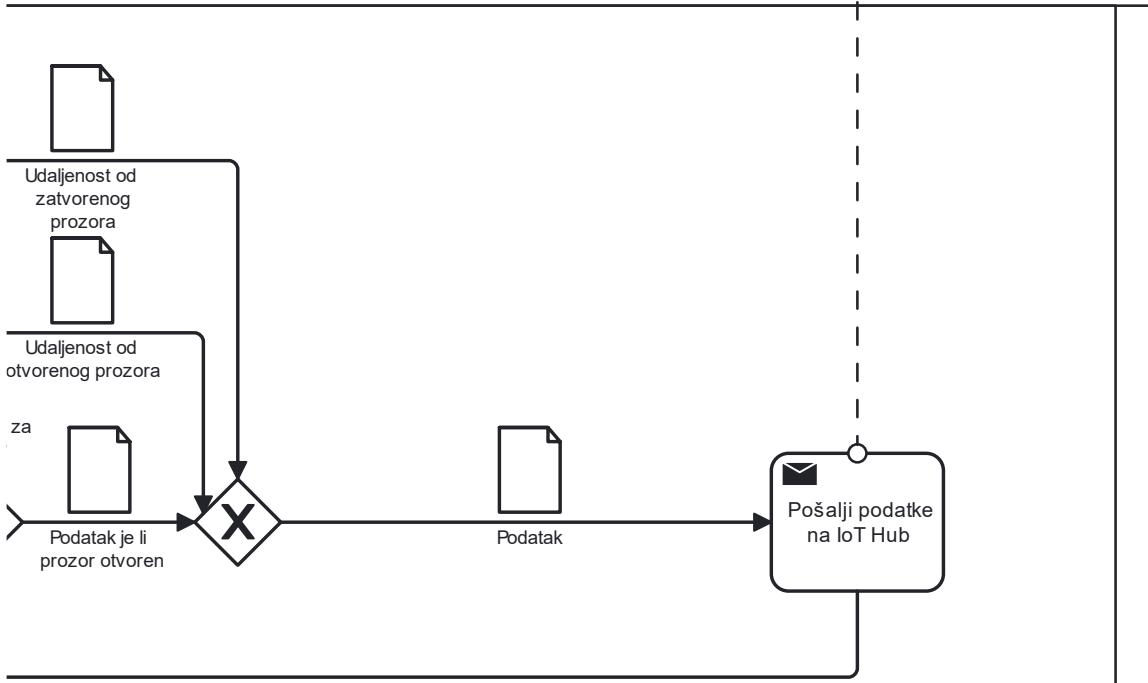
Mjerač udaljenosti



Slika 26: staza mjerača udaljenosti 1/3



Slika 25: staza mjerača udaljenosti 2/3

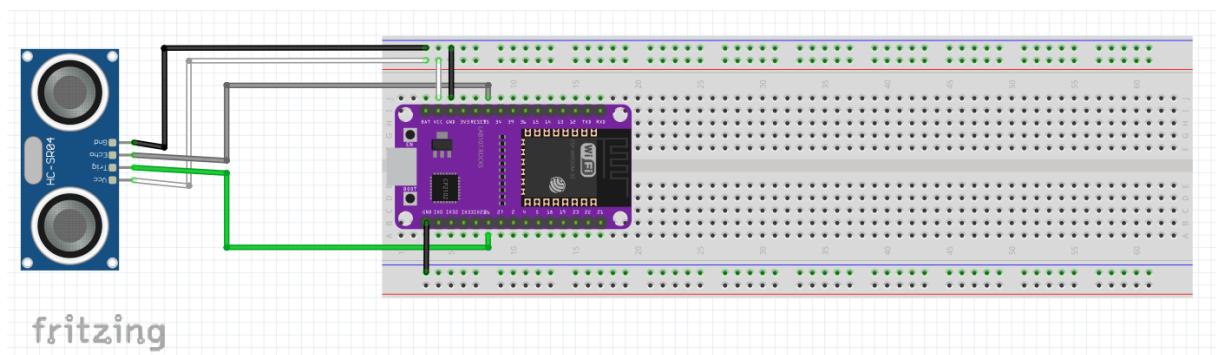
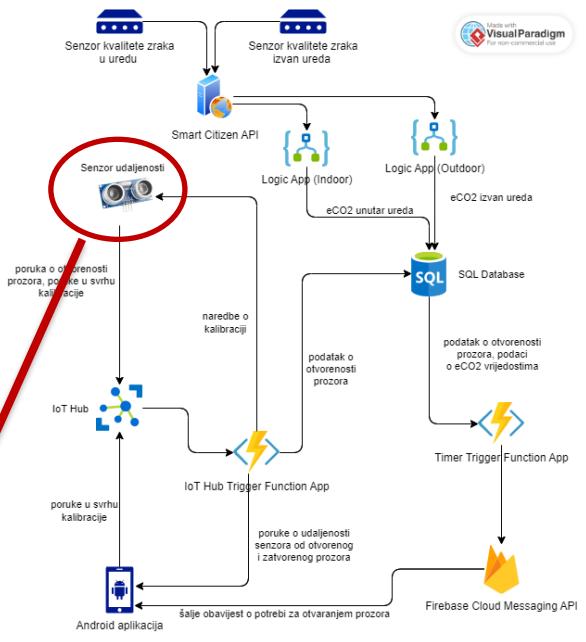


Slika 27: staza mjerača udaljenosti 3/3

2.5. Shema (dijagram) spajanja

Shema spajanja prikazana je pomoću dijagrama izrađenog u Fritzing alatu [7]. Bitna napomena je da jedino što treba spojiti fizički je senzor udaljenosti, stoga ovo poglavlje projekta pokriva samo taj dio. Ostali uređaji su izvan našeg dosega (senzori kvalitete zraka), te s njih samo čitamo podatke, pa iz tog razloga ne postoji shema spajanja tih uređaja.

Iz priložene slike možemo uočiti da je ultrazvučni senzor spojen na pinove Dasduino CONNECT PLUS pločice. Vcc senzora je spojen na Vcc pin pločice iz kojeg dobiva potreban napon za rad, te isto tako je i GND (uzemljenje) spojeno na GND pin pločice. Trigger senzora je spojen na GPIO (General Pin Input Output) 26, dok je Echo pin spojen na pin 35, koji je predviđen samo za primanje vrijednosti (Input only).



Slika 28: shema spajanja senzora udaljenosti

3. Izvršni programi

Nakon prikazanih komponenti arhitekture sustava, komunikacijskih dijagrama i dijagrama aktivnosti te razumijevanja sheme spajanja senzora udaljenosti, vrijeme je za prikaz programskega koda koji je zaslužan za ispravno izvođenje sustava. Programska kod se može podijeliti u više izvršnih programa, iz jednostavnog razloga, jer je pisan u više različitih programskih okruženja. U sljedećim poglavljima su ukratko opisani i objašnjeni izvršni programi senzora udaljenosti, Logic App servisa, Function App servisa te Android aplikacije.

3.1. Senzor udaljenosti

Izvor koda: [9]. Prilikom izrade koda za senzor udaljenosti, poslužili smo se materijalima koje smo izradili za vrijeme trajanja vježbi te smo prema predlošku iz vježbi izradili kod za vlastiti slučaj. S obzirom na navedeno, kod neće biti u cijelosti prikazan, već samo neki njegovi dijelovi.

Izvedba senzora udaljenosti koristi dvije datoteke zaglavlja i tri cpp datoteke od kojih SerialLogger datoteke služe za jednostavno korištenje serijske komunikacije za slanje informacija i grešaka u konzolu (naredbeni redak) te ispis trenutnog vremena, dok datoteke AzIoTSasToken omogućuju generiranje, provjeru isteka i dohvaćanje trenutnog SAS tokena potrebnog za autentifikaciju uređaja na Azure IoT Hub klijentu.

Zadaća glavnog programa (main) je uspostavljanje senzora udaljenosti da mjeri udaljenost te da šalje telemetrijske podatke na Azure platformu putem MQTT protokola. Na vrhu programa je prisutno definiranje varijabla, a može se istaknuti da se u strukturu MessageStruct pohranjuju informacije o vrsti poruke i izmjerenoj udaljenosti.

```
struct MessageStruct {
    int messageType;
    double distanceMeasured;
} RecievedMessageStruct;
```

Prilikom inicijalizacije u setup dijelu, inicijalizira se vrijeme, povezuje se WiFi, konfigurira se Azure IoT Hub klijent te se povezuje i konfigurira MQTT klijent. Na sljedećem primjeru koda prikazano je pozivanje funkcija i metoda u setup dijelu.

```

void setup() {
    setupWiFi();
    initializeTime();

    if (initIoTHub()) {
        connectMQTT();
        mqttReconnect();
    }

    az_result res = az_iot_hub_client_telemetry_get_publish_topic(&client, NULL, publishTopic, 200, NULL );
    // The receive topic isn't hardcoded and depends on chosen properties, therefore we need to use
    // az_iot_hub_client_telemetry_get_publish_topic()

    // Setup ultrasonic sensor
    pinMode(trigPIN, OUTPUT);
    pinMode(echoPIN, INPUT);

    Logger.Info("Setup done");
}

```

U funkciji `loop()` provjerava se povezanost s MQTT brokerom (posrednikom), izvršava se MQTT petlja, te se periodički šalju telemetrijski podaci (svakih 10 sekundi).

```

void loop() [
    if(!mqttClient.connected()) mqttReconnect();

    mqttClient.loop();

    if(nextTime <= millis()){
        sendTelemetryData(1);

        nextTime = millis() + 10000; // + 10 seconds
    }
]

```

Još bi bilo vrijedno za izdvojiti funkciju `callback()`, koja se poziva kada stigne poruka na temu (topic) na koju je uređaj pretplaćen putem MQTT protokola. Iz priloženog koda može se vidjeti kako se poziva funkcija `JsonParse` koja je odgovorna za parsiranje JSON formata poruke i dobivanje relevantnih podataka. Na temelju primljenih podataka se dalje izvršavaju odgovarajuće akcije.

```

void callback(char *topic, byte *payload, unsigned int length) {
    payload[length] = '\0';
    String message = String((char*)payload);

    Logger.Info(message);

    MessageStruct Result = JsonParse(message);

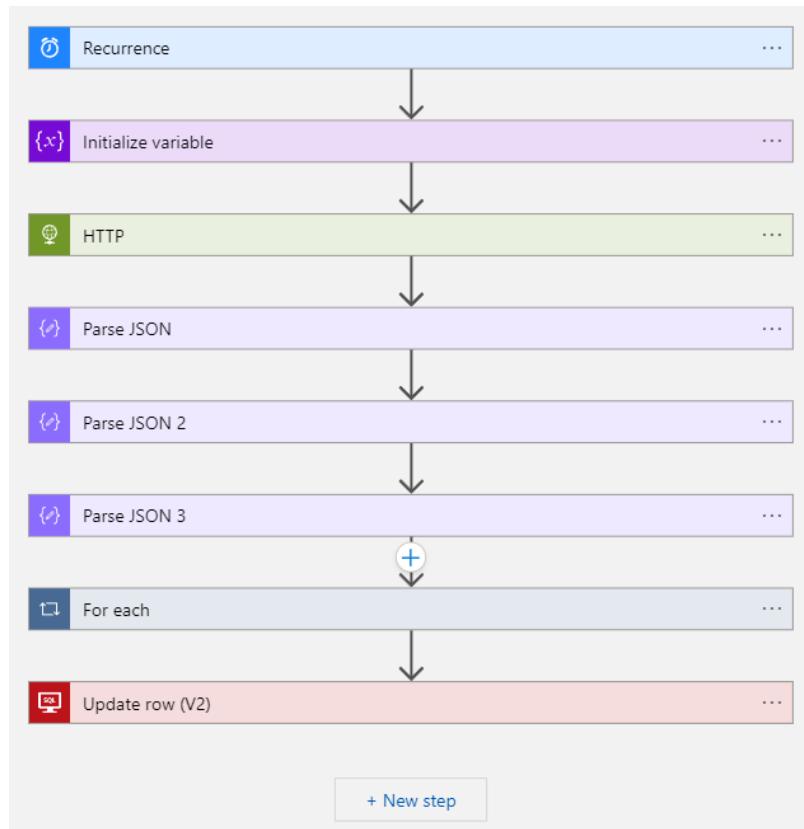
    Logger.Info("messageType: " + String(Result.messageType) + " distanceMeasured: " + String(Result.distanceMeasured));

    // if the messageType is 4 Dasduino measures distance to open window
    // if the messageType is 5 Dasduino measures distance to closed window
    switch (Result.messageType) {
        case 4: Kalibriraj(1); sendTelemetryData(2); break;
        case 5: Kalibriraj(2); sendTelemetryData(3); break;
    }
}

```

3.2. Logic App servisi

Iako Logic App servisi ne sadrže izvršni kod, već su vizualna reprezentacija koda u Azure Cloud sučelju, svejedno će biti prikazani u svrhu dobivanja razumijevanja na koji način funkcioniraju. Logic App unutarnjeg senzora kvalitete zraka ima istu logiku kao i Logic App vanjskog senzora, stoga će cijeli logika Logic App servisa biti prikazan na jednom primjeru. Vizualni prikaz Logic App servisa dan je sljedećom slikom.



Slika 29: Logic App servis - prikaz koraka (logike)

Logic App servis se oblikuje na način da se dodaje dio po dio logike. Smjer strelica govori u kojem smjeru podaci putuju. Prvi dio Recurrence (Ponavljanje) postavlja interval u kojem se dohvaća poruka primljena sa senzora kvalitete zraka. HTTP dio definira s koje web stranice se dohvaća podatak, a Parse JSON dohvaća podatak u JSON obliku. Na slici se može vidjeti kako su korištena tri uzastopna koraka Parse JSON, to je zato što podatak koji tražimo se nalazi unutar više JSON zapisa to jest, kada se promatra struktura dobivene poruke, podatak o razini eCO2 u zraku se nalazi na dnu strukture. Korak For each traži id uređaja koji odgovara broju 112, jer je to senzor čije vrijednosti mi očitavamo. Update row ažurira podatke u prethodno kreiranoj bazi podataka, pritom upisujući pročitanu vrijednost o razini eCO2.

ID	value
1	1168
2	1844

Slika 30: prikaz upisanih vrijednosti razine eCO2 u zraku u bazu podataka

Pod ID stupcem vidimo brojeve 1 i 2. Broj 1 označava očitanu vrijednost unutarnjeg senzora kvalitete zraka, a pod brojem 2 se nalazi vrijednost vanjskog senzora kvalitete zraka. Viša (veća) vrijednost predstavlja goru kvalitetu zraka, odnosno to znači da je u zraku prisutna veća razina ugljikovog dioksida.

3.3. Function App servisi

Već je i prije spomenuto da su Function App servisi srž arhitekture, te nema sumnje da je i logika iza njih veoma složena. Korištena su dva Function App servisa, koji će biti opisani zasebno, prvo IoT Hub Trigger Function App, a potom Time Trigger Function App.

3.3.1. IoT Hub Trigger Function App

Struktura Function App servisa je oblikovana u tri klase Root, ValuesByHour i IoTHubTriggerCSharp1. U Root klasu se samo upisuju podaci o tipu poruke (određena cjelobrojnom vrijednosti – id) i izmjerena vrijednost (decimalni zapis vrijednosti). Root klasa predstavlja strukturu podataka JSON poruke koja se očekuje kao ulaz u funkciju. ValuesByHour je isto jednostavna klasa kao i Root klasa, samo što ova klasa predstavlja model podataka koji sadrži listu cjelobrojnih brojeva obavijesti po satu.

```
public class Root
{
    public int messageType { get; set; }
    public double distanceMeasured { get; set; }
}

public class ValuesByHour
{
    public List<int> numOfNotificationsByHour { get; set; }
}
```

Klasa IoTHubTriggerCSharp1 je zadužena za izvođenje logike cijelog okidača (triggera) Function App servisa. Glavna funkcija je Run(), koja u svojem kodu poziva ostale metode, kao na primjer funkcije ForwardFlagToDatabase(), CloudToDeviceMessage(), i CloudToDeviceVisualisationMessage(). Funkcija Run() se pokreće kada se dogodi događaj u IoT Hub servisu. Ovisno o ispunjenom uvjetu izvršava se neka od preostalih funkcija.

Run() funkcija ispituje vrijednost dobivenog tipa poruke koji može biti od broja 1 do 5 i broj 7. U slučaju 1 ispisuje se poruka u konzolu o uspješnom primitku poruke te se poziva funkcija ForwardFlagToDatabase() koja se koristi za ažuriranje baze podataka ovisno o primljenoj vrijednosti zastavice (flag value). Poziva se funkcija NonQueryDatabase() koja uspostavlja konekciju sa bazom, izvršava SQL upite, te naposljetku zatvara konekciju s bazom podataka.

```
if (!string.IsNullOrEmpty(msg))
{
    Root myDeserializedClass = JsonConvert.DeserializeObject<Root>(msg);

    switch(myDeserializedClass.messageType){
        case 1: // Distance measurer is updating the windowIsOpen flag in the database
            logger.LogInformation($"GOT 1");
            bool flagValue = myDeserializedClass.distanceMeasured > 0.5;
            logger.LogInformation($"The flag is: {flagValue}");
            ForwardFlagToDatabase(flagValue);

            break;
    }
}

public void ForwardFlagToDatabase(bool flagValue){
    string query;
    if(flagValue){
        query = $"UPDATE [dbo].[WindowsAreOpen] SET windowIsOpen=1 WHERE ID=1;";
    }else{
        query = $"UPDATE [dbo].[WindowsAreOpen] SET windowIsOpen=0 WHERE ID=1;";
    }

    logger.LogInformation($"The query is: {query}");

    NonQueryDatabase(query);
}
```

```

public void NonQueryDatabase(string query){
    using SqlConnection connection = new(databaseConnectionString);
    connection.Open();

    logger.LogInformation($"Opened connection to database");

    using (SqlCommand command = new(query, connection))
    {
        var rows = command.ExecuteNonQuery();

        logger.LogInformation($"{rows} rows were updated in the database");
    }

    connection.Close();
}

```

Ukoliko je primljena poruka s vrijednošću 2 ili 3 tada znamo da senzor udaljenosti šalje obavijest o izmjerenoj udaljenosti otvorenog prozora (id=2) ili zatvorenog prozora (id=3). U konzoli se ispisuje poruka o uspješno primljenoj poruci te se poziva funkcija CloudToDeviceMessage(). CloudToDeviceMessage() se koristi za slanje poruka uređajima putem IoT Hub servisa. Ova funkcija stvara objekt ServiceClient koja se koristi za slanje poruka. Unutar CloudToDeviceMessage() metode se poziva još jedna metoda SendCloudToDeviceMessageAsync(), koja asinkrono šalje poruke uređajima koristeći stvoreni ServiceClient. Poruke se šalju u JSON obliku.

```

case 2: // Distance measurer is sending a notification regarding the length towards opened
         //window to the mobile phone
logger.LogInformation($"GOT 2");
logger.LogInformation($"The distance to opened is: {myDeserializedClass.distanceMeasured}");
CloudToDeviceMessage(androidAppsID, 2, myDeserializedClass.distanceMeasured);

break;

case 3: // Distance measurer is sending a notification regarding the length towards closed
         //window to the mobile phone
logger.LogInformation($"GOT 3");
logger.LogInformation($"The distance to closed is: {myDeserializedClass.distanceMeasured}");
CloudToDeviceMessage(androidAppsID, 3, myDeserializedClass.distanceMeasured);

break;

```

```

public void CloudToDeviceMessage(string deviceId, int mesType, double distanceMes){
    logger.LogInformation("Send Cloud-to-Device message");
    serviceClient = ServiceClient.CreateFromConnectionString(ioTHubConnectionString);

    SendCloudToDeviceMessageAsync(deviceId, mesType, distanceMes).Wait();
}

private async Task SendCloudToDeviceMessageAsync(string deviceId, int mesType, double distMeasured)
{
    Root messageToSend = new()
    {
        messageType = mesType,
        distanceMeasured = distMeasured
    };

    string jsonString = JsonConvert.SerializeObject(messageToSend);
    logger.LogInformation("Sending message: " + jsonString);
    var commandMessage = new Message(Encoding.ASCII.GetBytes(jsonString));
    await serviceClient.SendAsync(deviceId, commandMessage);
}

```

Primljene poruke čija je id vrijednost 4 ili 5 se izvršavaju vrlo slično kao i prethodno opisane poruke s id vrijednosti 2 ili 3. Pozivaju se funkcije CloudToDeviceMessage() i SendCloudToDeviceMessageAsync(), međutim bitna razlika je da su poruke ovog tipa poslane s mobilnog uređaja, koji traži da od senzora udaljenosti da izmjeri i pošalje vrijednost udaljenosti otvorenog / zatvorenog prozora.

```

case 4: // The mobile phone is instructing the distance measurer to measure the distance
         // towards the open window
logger.LogInformation($"GOT 4");
logger.LogInformation($"Mobile phone is requesting distance towards open");
CloudToDeviceMessage(distanceMeasurerID, 4, myDeserializedClass.distanceMeasured);

break;

case 5: // The mobile phone is instructing the distance measurer to measure the distance
         // towards the closed window
logger.LogInformation($"GOT 5");
logger.LogInformation($"Mobile phone is requesting distance towards closed");
CloudToDeviceMessage(distanceMeasurerID, 5, myDeserializedClass.distanceMeasured);

break;

```

Poruka koja sadrži id s vrijednosti 7, označava da mobilni uređaj zahtjeva (traži) podatke o notifikacijama u svrhu vizualizacije podataka putem grafikona. Poziva se funkcija CloudToDeviceVisualisationMessage(), koja dalje poziva ostale metode, kako bi se na kraju ispravo prikazao dijagram na zaslonu korisnika aplikacije. U nastavku je dan programski kod svih metoda koje se pozivaju i koriste za izvršavanje ove funkcionalnosti.

```

case 7: // The mobile phone is requesting data about notification times
    logger.LogInformation($"GOT 7");
    logger.LogInformation($"Mobile phone is requesting data for visualisation");

    CloudToDeviceVisualisationMessage();
    break;

```

```

private void CloudToDeviceVisualisationMessage(){

    // Query for all times of notifying in the last 24 hours (in UTC+1 time)
    List<DateTime> datetimeResults = GetDatabaseNotificationTimeEntries();

    List<int> numOccurrences = ProcessDatabaseResults(datetimeResults);

    string jsonString = CreateStringFromObject(numOccurrences);

    logger.LogInformation("Sending message: " + jsonString);

    SendCloudToDeviceVisualisationMessageAsync(jsonString).Wait();
}

```

```

private List<DateTime> GetDatabaseNotificationTimeEntries(){
    using SqlConnection connection = new(databaseConnectionString);
    connection.Open();

    string datetimeResultsQuery =
    "SELECT datetimeOfNotifying FROM [dbo].[TimesofNotifying] WHERE datetimeOfNotifying >= DATEADD(HOUR, -23, GETUTCDATE());";
    // the query is designed for UTC+1 timezone

    List<DateTime> datetimeResults = new();

    logger.LogInformation("Getting the values");
    using (SqlCommand command = new(datetimeResultsQuery, connection))
    {
        //logger.LogInformation("command is set");
        using SqlDataReader reader = command.ExecuteReader();
        //logger.LogInformation("reader was executed");
        while (reader.Read())
        {
            datetimeResults.Add(Convert.ToDateTime(reader.GetValue(0)));
        }
    }

    connection.Close();
    return datetimeResults;
}

```

```

private static List<int> ProcessDatabaseResults(List<DateTime> datetimeResults)
{
    // For all values in datetimeResults aggregate those so you get the number of entries for every hour in a day
    // Initialize a List<int> to store occurrences for each hour
    List<int> numOfOccurrences = new(24);

    // Initialize the list with zeros for each hour
    for (int i = 0; i < 24; i++)
    {
        numOfOccurrences.Add(0);
    }

    // Grouping datetimeResults by hour and counting occurrences
    foreach (DateTime dt in datetimeResults)
    {
        numOfOccurrences[dt.Hour]++;
    }

    return numOfOccurrences;
}

private static string CreateStringFromObject(List<int> numOfOccurrences)
{
    ValuesByHour messageToSend = new()
    {
        numOfNotificationsByHour = numOfOccurrences
    };

    return JsonConvert.SerializeObject(messageToSend);
}

```

```

private static async Task SendCloudToDeviceVisualisationMessageAsync(string jsonString)
{
    var commandMessage = new Message(Encoding.ASCII.GetBytes(jsonString));
    serviceClient = ServiceClient.CreateFromConnectionString(ioTHubConnectionString);

    await serviceClient.SendAsync(androidAppsID, commandMessage);
}

```

Ukratko rečeno, može se reći da IoT Hub Trigger Function App servis prima poruke iz IoT Hub servisa, analizira njihov sadržaj, ažurira SQL bazu podataka i šalje natrag poruke mobilnim uređajima ovisno o sadržaju primljenih poruka. Primljene poruke se prikazuju u vidu izmjerениh vrijednosti udaljenosti senzora ili u obliku statističkih podataka prikazanih grafikonom.

3.3.2. Time Trigger Function App

Time Trigger Function App je podijeljen u tri klase, klasa NotificationToSend – sadrži podatke potrebne za prijavu u Firebase Cloud, klasa ValuesForVisualisation – omogućuje pohranu liste nizova znakova, čiji se podaci kasnije koriste za prikaz grafikona, te klasa TimerTriggerCSharp1 - sadrži svu logiku zaduženu za ispravno izvođenje servisa. Klasa NotificationToSend sastavljena je samo od inicijalizacije varijabli u koje se pohranjuju vrijednosti tipa string, dok je klasa ValuesForVisualisation sastavljena od varijable u koju se pohranjuje lista znakova.

```
public class NotificationToSend
{
    public string type { get; set; }
    public string project_id { get; set; }
    public string private_key_id { get; set; }
    public string private_key { get; set; }
    public string client_email { get; set; }
    public string client_id { get; set; }
    public string auth_uri { get; set; }
    public string token_uri { get; set; }
    public string auth_provider_x509_cert_url { get; set; }
    public string client_x509_cert_url { get; set; }
    public string universe_domain { get; set; }
}

public class ValuesForVisualisation
{
    public List<string> timesOfNotifying { get; set; }
}
```

Za glavno izvođenje klase TimerTriggerCSharp1 zadužena je funkcija Run(), koja u svojoj strukturi poziva funkcije QueryDatabaseBoolValue(),QueryDatabaseFloatValue(), CloudToDeviceMessageAsync(), CreateInputCommand() i InsertIntoDatabase(). Funkcija Run() se izvršava svaki put kada se aktivira mjerač vremena (timer). Timer je postavljen da se aktivira svake minute. Zatim nakon provjere je li ostvarena konekcija prema Azure SQL bazi podataka upisuju se SQL upiti u string varijable. Nadalje otvara se veza prema SQL bazi podataka (connection.Open()) te se izvršavaju upit za dobivanje informacija o položaju prozora, te upiti za dohvaćanje podataka o kvaliteti zraka unutar i izvan prostorije.

```

public void Run([TimerTrigger("0 */1 * * * *")]TimerInfo myTimer, ILogger logger, ExecutionContext context)
{
    log = logger;
    log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now}");

    // Check if the connection string is null or empty
    if (string.IsNullOrEmpty(databaseConnectionString))
    {
        log.LogInformation("Azure SQL Database connection string is missing or empty.");
        return;
    }

    string queryForWindow = "SELECT windowIsOpen FROM WindowsAreOpen WHERE ID=1;";
    string queryForOutdoorQuality = "SELECT value FROM [dbo].[AirQualityReadings] WHERE ID=2;";
    string queryForIndoorQuality = "SELECT value FROM [dbo].[AirQualityReadings] WHERE ID=1;";

    using SqlConnection connection = new(databaseConnectionString);

    connection.Open();

    bool windowIsOpen = QueryDatabaseBoolValue(connection, queryForWindow);
    log.LogInformation($"Window is open: {windowIsOpen}");

    float qualityOutdoors = QueryDatabaseFloatValue(connection, queryForOutdoorQuality);
    log.LogInformation($"Outdoor quality: {qualityOutdoors}");

    float qualityIndoors = QueryDatabaseFloatValue(connection, queryForIndoorQuality);
    log.LogInformation($"Indoor quality: {qualityIndoors}");
}

```

Na slijedećem primjeru koda su prikazane funkcije koje izvršavaju SQL upite i vraćaju rezultate kao bool ili float vrijednost.

```

private static bool QueryDatabaseBoolValue(SqlConnection connection, string queryForWindow){
    using SqlCommand command = new(queryForWindow, connection);
    object result = command.ExecuteScalar();

    return (bool)result;
}

private static float QueryDatabaseFloatValue(SqlConnection connection, string queryForWindow){
    using SqlCommand command = new(queryForWindow, connection);
    object result = command.ExecuteScalar();

    return (float)result; // Unable to cast object of type 'System.Single' to type 'System.Double'
}

```

Funkcija Run() ima još jedan dio koda koji ispituje kvalitetu zraka i stanje prozora. Ako je kvaliteta zraka unutar prostorije veća od 1200 i prozor je zatvoren, onda se šalje poruka na Android uređaj o potrebi za otvaranjem prozora. Taj dio se izvršava putem metoda CloudToDeviceMessageAsync(), u kojoj se poziva nova metoda CreateJsonForGoogleAuth(). Prva metoda se koristi za slanje asinkronih zahtjeva za slanje poruke, te se u njoj postavlja poruka koja će se poslati, uključujući naslov i tijelo poruke te temu (topic) na koju se poruka šalje. Druga metoda stvara JSON objekt s podacima potrebnim za autentifikaciju aplikacije pri korištenju Firebase usluge.

```
if(qualityIndoors > 1200 && !windowIsOpen){ // if higher than 1200 ppm and window is closed
    _ = CloudToDeviceMessageAsync();
```

```
public async Task CloudToDeviceMessageAsync()
{
    var defaultApp = FirebaseApp.Create(new AppOptions()
    {
        Credential = GoogleCredential.FromJson(CreateJsonForGoogleAuth()),
    });

    log.LogInformation(defaultApp.Name); // [DEFAULT] usually

    var message = new FirebaseAdmin.Messaging.Message()
    {
        Notification = new Notification
        {
            Title = "Open the window",
            Body = "Air quality in your room has dropped. Please open your window."
        },
        Topic = "Notifications"
    };

    var messaging = FirebaseMessaging.DefaultInstance;
    var result = await messaging.SendAsync(message);

    log.LogInformation(result);
}
```

```
private string CreateJsonForGoogleAuth()
{
    MessageToSend messageToSend = new()
    {
        type = "",
        project_id = "",
        private_key_id = "",
        private_key = "",
        client_email = "",
        client_id = "",
        auth_uri = "",
        token_uri = "",
        auth_provider_x509_cert_url = "",
        client_x509_cert_url = "",
        universe_domain = ""
    };

    string jsonString = JsonConvert.SerializeObject(messageToSend);
    log.LogInformation(jsonString);
    return jsonString;
}
```

Naposljeku, pozivaju se metode CreateInputCommand() i InsertIntoDatabase() koje izvršavaju SQL naredbu za umetanje novih vrijednosti u bazu podataka u koju se upisuje datum i vrijeme poslane poruke. Potom se zatvara veza prema SQL bazi podataka (connection.Close()), nakon što su operacije s bazom završene.

```
if(qualityIndoors > 1200 && !windowIsopen){ // if higher than 1200 ppm and window is closed
    _ = CloudToDeviceMessageAsync();

    // Update the database - add entry for this notification
    SqlCommand insertDatetimeOfNotifyingCommand = CreateInputCommand(connection);
    InsertIntoDatabase(insertDatetimeOfNotifyingCommand);
}

connection.Close();
```

```
private static SqlCommand CreateInputCommand(SqlConnection connection)
{
    string inputCommand = "INSERT INTO [dbo].[TimesOfNotifying] VALUES(@insertSmallDatatype);";
    // Example: "INSERT INTO [dbo].[TimesOfNotifying] VALUES('2024-01-30 00:01');";

    SqlCommand command = new(inputCommand, connection);

    command.Parameters.Add(new SqlParameter("@insertSmallDatatype", SqlDbType.SmallDateTime)
    {
        Value = DateTime.UtcNow.AddHours(1) // UTC+1, independent of location of execution
    });

    return command;
}
```

```
public void InsertIntoDatabase(SqlCommand commandToExecute){
    log.LogInformation("Inserting into database");

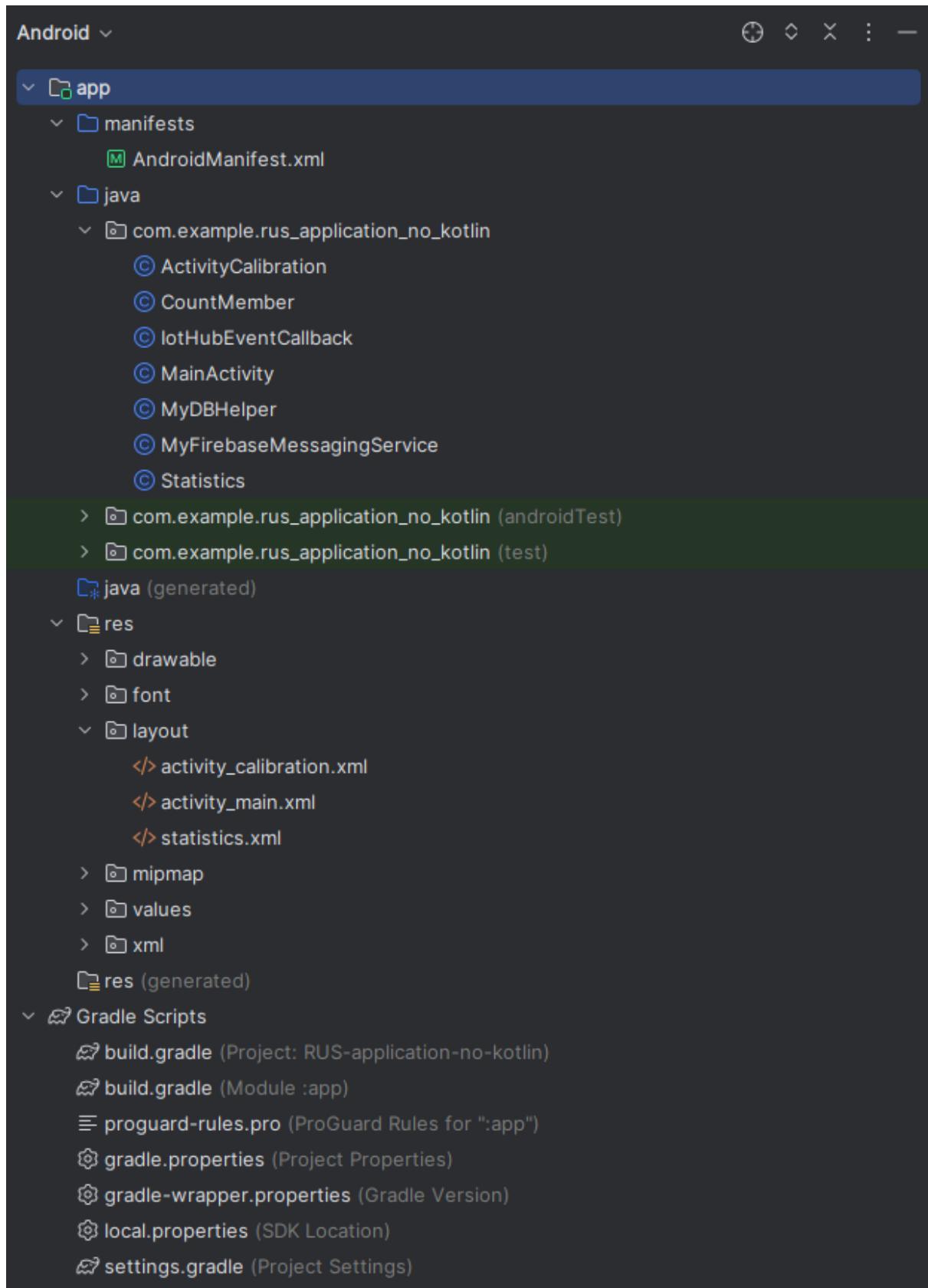
    using SqlCommand command = commandToExecute;
    var rows = command.ExecuteNonQuery();

    log.LogInformation($"{rows} rows were updated in the database");
}
```

Ukratko sažeto, može se reći da se Function App servis periodički izvršava svaku minutu. Kada se aktivira, povezuje se s Azure SQL bazom podataka kako bi dohvatio informacije o stanju prozora i kvaliteti zraka unutar i izvan prostorije. Na temelju prikupljenih informacija, donosi odluku te o tome obavještava korisnika Android uređaja da otvori prozor radi poboljšanja kvalitete zraka u prostoriji. Također, zapisuje vrijeme i datum kreirane obavijesti, koja se zatim spremi u zasebnu bazu podataka, te služi kao uvid u statističke podatke (grafikon).

3.4. Android aplikacija

Za realizaciju ovog projekta smo se odlučili za android mobilnu aplikaciju izrađenu putem Android studia. Odlučili smo se za rad u Javi za programiranje samih funkcionalnosti, te smo koristili xml za vizualni identitet aplikacije. Struktura tih elemenata izgleda sljedeće:



3.4.1. MainActivity

Main activity je java class koji se pokreće prvi pri otvaranju aplikacije.

Metoda onCreate se izvršava pri kreiranju klase. Unutar nje se nalazi kod za dobivanje NFC tokena za Firebase, te OnClickListener-i za dugmad koja se nalaze na tom početnom zaslonu. setContentView određuje kakav raspored će naša stranica imati putem xml datoteke.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    MyDBHelper dbHelper = new MyDBHelper(context: this);
    dbHelper.SetupDatabase();
    FirebaseApp.initializeApp(context: this);
    subscribeTopics();
    askNotificationPermission();

    FirebaseMessaging.getInstance().getToken().addOnCompleteListener(new OnCompleteListener<String>() {
        @Override
        public void onComplete(@NonNull Task<String> task) {
            if (!task.isSuccessful()) {
                System.out.println("Fetching FCM registration token failed");
                return;
            }

            // Get new FCM registration token
            String token = task.getResult();

            Log.d(tag: "TOKEN", token);
        }
    });

    textView = findViewById(R.id.Title);

    button1 = findViewById(R.id.calibration);
    button1.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) { openCalibration(); }
    });

    button2 = findViewById(R.id.statistics);
    button2.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) { openStatistics(); }
    });
}
```

Metode openCalibration() i openStatistics() služe za otvaranje drugog class-a tj. stranice naše aplikacije.

```
+ usage
public void openCalibration() {
    Intent intent= new Intent( packageContext: this, ActivityCalibration.class);
    startActivity(intent);
}

1 usage
public void openStatistics(){
    Intent intent= new Intent( packageContext: this, Statistics.class);
    startActivity(intent);
}
```

Unutar tog class-a se pri prvom pokretanju također korisnik preplaćuje na Notifications topic preko kojega firebase zna kojim sve uređajima treba poslati notifikaciju. Osim toga korisniku se nudi pop-up sa zahtjevom za dopuštenje za prikaz notifikacija.

```
1 usage
private void subscribeTopics() {
    // [START subscribe_topics]
    FirebaseMessaging.getInstance().subscribeToTopic("Notifications")
        .addOnCompleteListener(new OnCompleteListener<Void>() {
            @Override
            public void onComplete(@NonNull Task<Void> task) {
                String msg = "Subscribed";
                if (!task.isSuccessful()) {
                    msg = "Subscribe failed";
                }
                Log.d( tag: "TOPIC", msg);
            }
        });
    // [END subscribe_topics]
}
```

```
1 usage
private final ActivityResultLauncher<String> requestPermissionLauncher =
    registerForActivityResult(new ActivityResultContracts.RequestPermission(), isGranted -> {
        if (isGranted) {
            // FCM SDK (and your app) can post notifications.
        } else {
            // Inform user that that your app will not show notifications.
        }
    });
1 usage
private void askNotificationPermission(){
    // This is only necessary for API level >= 33 (TIRAMISU)
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        if (ContextCompat.checkSelfPermission(context: this, Manifest.permission.POST_NOTIFICATIONS) ==
            PackageManager.PERMISSION_GRANTED) {
            // FCM SDK (and your app) can post notifications.
        } else{
            // Directly ask for the permission
            requestPermissionLauncher.launch(Manifest.permission.POST_NOTIFICATIONS);
        }
    }
}
```

3.4.2. ActivityCalibration

Na početku klase ćemo definirati neke varijable koje će nam kasnije biti od pomoći unutar koda.

```
3 usages
public TextView CalculationOpened;
3 usages
public static TextView CalculationClosed;
2 usages
public Button button_back;

1 usage
public static float DistanceOpened;
1 usage
public static float MessageType;
1 usage
public static float DistanceClosed;
```

Funkcija “sendMessage” u ovom kodu služi za slanje poruka IoT hubu putem MQTT protokola. Prvo se definira connection string za IoT Hub, koji sadrži informacije o hostu, uređaju i dijeljenom ključu. Nakon toga, stvara se “DeviceClient” objekt s MQTT protokolom i otvara veza prema IoT Hubu.

U funkciji se postavlja i callback (“MessageCallbackMqtt”), koji će biti pozvan kada stigne poruka od IoT Huba. Zatim se kreira JSON objekt poruke koji sadrži vrijednost parametra “value”, dok se “distanceMeasured” postavlja na nulu.

Nakon toga, poruka se šalje putem “deviceClient.sendEvent(msg)”, te se čeka 2 sekunde kako bi se omogućilo vrijeme za odgovor od IoT Huba. Iz primljene poruke izvlače se vrijednosti “messageTypeReturn” i “distanceMeasured”.

Na temelju vrijednosti “messageTypeReturn”, pozivaju se određene metode (“changeValueClosed” ili “changeValueOpened”). Vjerojatno ove metode služe za ažuriranje korisničkog sučelja ili obavljanje drugih radnji ovisno o vrsti primljene poruke.

Na kraju, veza prema IoT Hubu se zatvara (“deviceClient.close()”). Ovaj kod čini dio aplikacije koja uspostavlja komunikaciju s IoT Hubom putem MQTT protokola, šalje poruke te obrađuje odgovore dobivene od IoT Huba.

```

2 usages
public void sendMessage(int value) {
    String iotHubConnectionString =
"HostName=HUB-RUS-jbegovic21.azure-devices.net;DeviceId=Androids;SharedAccessKey=ApQkG6BAwHwEcPRXicVy0BcekniuS2Ys6AIoTIG2F7U=";
    DeviceClient deviceClient = new DeviceClient(iotHubConnectionString, IoTHubClientProtocol.MQTT);

    try {
        deviceClient.open( withRetry: true);
        Log.d( tag: "IoTHub", msg: "Successfully connected");

        // Set up message callback
        MessageCallbackMqtt callback = new MessageCallbackMqtt();
        Counter counter = new Counter( num: 0);
        deviceClient.setMessageCallback(callback, counter);

        // Create JSON message
        JSONObject jsonMessage = new JSONObject();
        jsonMessage.put( name: "messageType", value);
        jsonMessage.put( name: "distanceMeasured", value: 0);

        // Send JSON message
        Message msg = new Message(jsonMessage.toString());
        msg.setContentType("application/json");
        msg.setMessageId(UUID.randomUUID().toString());
        Log.d( tag: "IoTHub", jsonMessage.toString());

        deviceClient.sendEvent(msg);
        Log.d( tag: "IoTHub", msg: "Successfully sent the message");

        // Wait for response (you may want to handle this asynchronously)
        // For simplicity, we'll just wait for a short duration here.
        Thread.sleep( millis: 2000);

        double messageTypeReturn = extractValue(parseJsonToString(new String(msg.getBytes(), Message.DEFAULT_IOTHUB_MESSAGE_CHARSET)));
        double distanceMeasured = extractValue2(parseJsonToString(new String(msg.getBytes(), Message.DEFAULT_IOTHUB_MESSAGE_CHARSET)));
        Log.d( tag: "ISPIS ZA CALLBACK", msg: messageTypeReturn + " " + distanceMeasured);

        // Handle the received message
        // You can update UI or perform other actions based on the received message.

        counter.increment();

        if (messageTypeReturn == 3) {
            changeValueClosed(distanceMeasured);
        }
        else if (messageTypeReturn == 2) {
            changeValueOpened(distanceMeasured);
        }

    } catch (IotHubClientException | JSONException | InterruptedException e) {
        Log.e( tag: "IoTHub", msg: "Error: " + e.getMessage());
    } finally {
        deviceClient.close();
    }
}

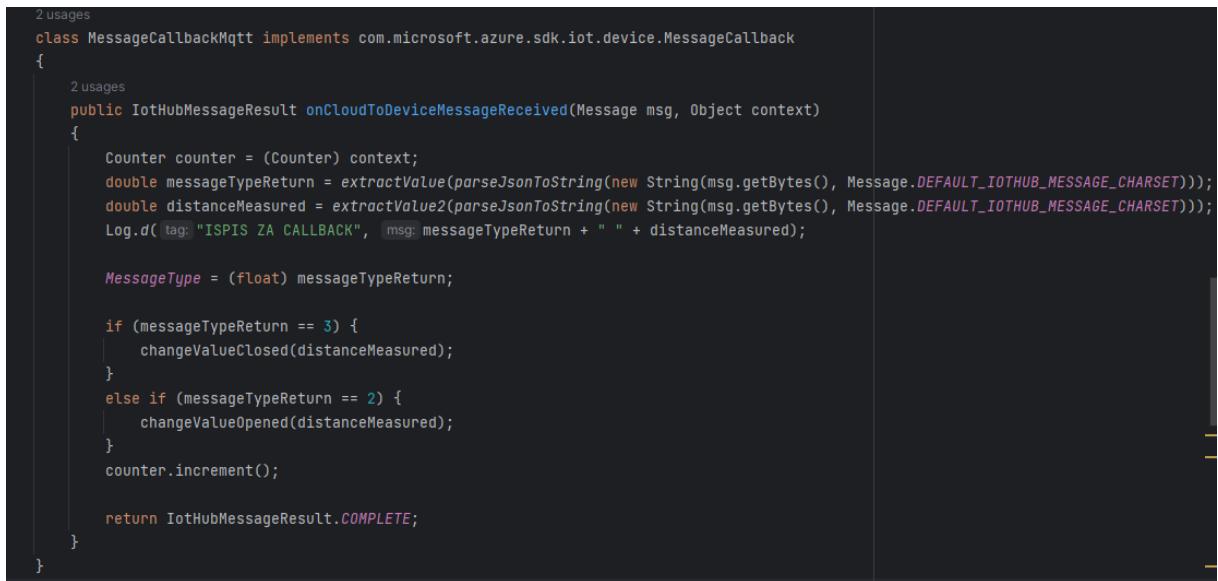
```

Klasa “MessageCallbackMqtt” predstavlja implementaciju sučelja “com.microsoft.azure.sdk.iot.device.MessageCallback” te se koristi za obradu poruka koje stižu od IoT Huba. Metoda “onCloudToDeviceMessageReceived” poziva se kada primi poruku od IoT Huba, a njezina implementacija obuhvaća nekoliko ključnih koraka.

Prvo, iz primljene poruke se izvlače podaci, obično u JSON formatu, koristeći metode poput “parseJsonToString” i “extractValue”. Zatim se ti podaci logiraju pomoću “Log.d”. Vrijednost “messageTypeReturn” iz poruke se pohranjuje u varijablu “MessageType”.

Nakon toga, provjerava se vrijednost "messageTypeReturn". Ovisno o njoj, pozivaju se određene metode poput "changeValueClosed" ili "changeValueOpened". Ovo vjerojatno služi za prilagodbu stanja ili ažuriranje korisničkog sučelja u skladu s primljenom porukom.

Brojač "counter" se inkrementira, vjerojatno za potrebe praćenja broja primljenih poruka. Konačno, metoda vraća "IoTHubMessageResult.COMPLETE", što označava da je poruka uspješno obrađena. Ovaj dio koda predstavlja ključnu logiku za rukovanje porukama iz IoT Huba, omogućavajući aplikaciji da reagira na promjene u stvarnom vremenu.



```
2 usages
class MessageCallbackMqtt implements com.microsoft.azure.sdk.iot.device.MessageCallback
{
    2 usages
    public IoTHubMessageResult onCloudToDeviceMessageReceived(Message msg, Object context)
    {
        Counter counter = (Counter) context;
        double messageTypeReturn = extractValue(parseJsonToString(new String(msg.getBytes(), Message.DEFAULT_IOTHUB_MESSAGE_CHARSET)));
        double distanceMeasured = extractValue2(parseJsonToString(new String(msg.getBytes(), Message.DEFAULT_IOTHUB_MESSAGE_CHARSET)));
        Log.d(tag: "ISPIS ZA CALLBACK", msg: messageTypeReturn + " " + distanceMeasured);

        MessageType = (float) messageTypeReturn;

        if (messageTypeReturn == 3) {
            changeValueClosed(distanceMeasured);
        }
        else if (messageTypeReturn == 2) {
            changeValueOpened(distanceMeasured);
        }
        counter.increment();

        return IoTHubMessageResult.COMPLETE;
    }
}
```

Metoda "parseJsonToString" ima zadatku pretvoriti JSON niz u ekvivalentan string koristeći Jackson biblioteku. Evo opisa njezine funkcionalnosti:

Metoda prima JSON niz ("jsonString") kao ulaz i pokušava ga parsirati. Prvo, stvara se instanca "ObjectMapper", koji je dio Jackson biblioteke za rad s JSON podacima. Zatim se koristi "objectMapper.readTree(jsonString)" kako bi se JSON niz pretvorio u "JsonNode", interno reprezentiranje JSON strukture u Jackson biblioteci.

Nakon dobivanja "JsonNode" objekta, on se pretvara natrag u string pomoću "jsonNode.toString()". Konačno, rezultirajući string se vraća kao rezultat metode.

U slučaju bilo kakve greške tijekom parsiranja, hvata se izuzetak ("Exception e"), ispisuje se njegov stog iznimaka ("e.printStackTrace()"), i metoda vraća "null".

```
private static String parseJsonToString(String jsonString) {  
    try {  
        // Create ObjectMapper instance  
        ObjectMapper objectMapper = new ObjectMapper();  
  
        // Parse JSON string to JsonNode  
        JsonNode jsonNode = objectMapper.readTree(jsonString);  
  
        // Convert JsonNode to String  
        return jsonNode.toString();  
    } catch (Exception e) {  
        e.printStackTrace();  
        return null;  
    }  
}
```

Metode “extractValue” i “extractValue2” imaju zadatku izvući vrijednost određenog polja (“messageType”) iz JSON niza i vratiti je kao double. Evo opisa njegove funkcionalnosti:

Metoda prima JSON niz (“jsonString”) kao ulaz i pokušava ga parsirati koristeći Jackson biblioteku. Prvo, stvara se instanca “ObjectMapper”, koji je dio Jackson biblioteke za rad s JSON podacima. Zatim se koristi “objectMapper.readTree(jsonString)” kako bi se JSON niz pretvorio u “JsonNode”, interno reprezentiranje JSON strukture u Jackson biblioteci.

Nakon dobivanja “JsonNode” objekta, iz njega se izvlači vrijednost polja “messageType” pomoću “jsonNode.get("messageType").asDouble()”. Ova vrijednost se zatim vraća kao rezultat metode.

```

2 usages
private static double extractValue(String jsonString) {
    try {
        // Create ObjectMapper instance
        ObjectMapper objectMapper = new ObjectMapper();

        // Parse JSON string to JsonNode
        JsonNode jsonNode = objectMapper.readTree(jsonString);

        // Extract value from JsonNode
        double messageType = jsonNode.get("messageType").asDouble();
        // Return the extracted value
        return messageType;
    } catch (Exception e) {
        e.printStackTrace();
        return 0;
    }
}

2 usages
private static double extractValue2(String jsonString) {
    try {
        // Create ObjectMapper instance
        ObjectMapper objectMapper = new ObjectMapper();

        // Parse JSON string to JsonNode
        JsonNode jsonNode = objectMapper.readTree(jsonString);

        // Extract value from JsonNode
        double messageType = jsonNode.get("distanceMeasured").asDouble();
        // Return the extracted value
        return messageType;
    } catch (Exception e) {
        e.printStackTrace();
        return 0;
    }
}

```

Ovaj sljedeći dio koda predstavlja postavljanje listenera za dva gumba u korisničkom sučelju. Prvi gumb je povezan s kalibracijom zatvorenog položaja (“calibrationClosedButton”), dok je drugi gumb povezan s kalibracijom otvorenog položaja (“calibrationOpenedButton”). Evo opisa što se događa u ovom dijelu koda:

1. "calibrationClosedButton": Povezuje se s gumbom iz korisničkog sučelja koji ima ID "R.id.calibration_closed". Kada se klikne na ovaj gumb, pokreće se "onClick" metoda koja šalje poruku IoT Hubu pomoću "sendMessage(5)". Ova poruka vjerojatno označava zahtjev za kalibracijom zatvorenog položaja. Odmaknuti komentarizirani dio ("// changeValueClosed()") sugerira da bi također moglo doći do ažuriranja sučelja ili neke druge akcije vezane uz kalibraciju zatvorenog položaja.

2. "calibrationOpenedButton": Povezuje se s gumbom iz korisničkog sučelja koji ima ID "R.id.calibration_opened". Kada se klikne na ovaj gumb, pokreće se "onClick" metoda koja šalje poruku IoT Hubu pomoću "sendMessage(4)". Ova poruka vjerojatno označava zahtjev za kalibracijom otvorenog položaja. Odmaknuti komentarizirani dio ("// changeValueOpened()") sugerira da bi također moglo doći do ažuriranja sučelja ili neke druge akcije vezane uz kalibraciju otvorenog položaja.

U osnovi, ovi gumbi služe za pokretanje kalibracije za određene položaje, a funkcionalnost vezana uz slanje poruka i potencijalno ažuriranje sučelja se poziva kada korisnik pritisne određeni gumb.

```
// DUGME KALIBRIRAJ ZATVOREN
Button calibrationClosedButton = findViewById(R.id.calibration_closed);
calibrationClosedButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        sendMessage(value: 5);
        changeValueClosed();
    }
});

// DUGME KALIBRIRAJ OTVOREN
Button calibrationOpenedButton = findViewById(R.id.calibration_opened);
calibrationOpenedButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        sendMessage(value: 4);
        changeValueOpened();
    }
});
```

Ove metode se koriste za postavljanje teksta na određenim tekstualnim poljima ("CalculationClosed" i "CalculationOpened") na temelju dobivene udaljenosti prilikom kalibracije.

Ova metoda koristi "runOnUiThread" kako bi se osiguralo da se ažuriranje sučelja izvrši u glavnoj dretvi. U "run" metodi, dobivena udaljenost ("DistanceClosed") se zaokružuje prema gore na dvije decimale i rezultirajuća vrijednost se postavlja kao tekst na "CalculationClosed" TextView elementu, praćena dodatkom " cm".

Ova metoda ima sličnu strukturu kao i prethodna, s time da se primljena udaljenost ("DistanceOpened") također zaokružuje prema gore na dvije decimale, te se rezultat postavlja kao tekst na "CalculationOpened" TextView element, opet praćen dodatkom " cm".

```
// Update the changeValueClosed() method as follows:  
2 usages  
public void changeValueClosed(double DistanceClosed) {  
    runOnUiThread(new Runnable() {  
        1 usage  
        double roundedValueDistanceClosed = Math.ceil(DistanceClosed * 100) / 100;  
        @Override  
        public void run() { CalculationClosed.setText(roundedValueDistanceClosed + " cm"); }  
    });  
}  
  
// Update the changeValueOpened() method as follows:  
2 usages  
public void changeValueOpened(double DistanceOpened) {  
    runOnUiThread(new Runnable() {  
        1 usage  
        double roundedValueDistanceOpened = Math.ceil(DistanceOpened * 100) / 100;  
        @Override  
        public void run() { CalculationOpened.setText(roundedValueDistanceOpened + " cm"); }  
    });  
}
```

3.4.3. Statistics

Statistics je aktivnost koja služi kako bi se prikazala statistika broja poruka poslanih određeni dan. Na njoj se nalaze gumbi button_back koji služi za vraćanje na glavni meni i CountGlobal koji služi kao globalna varijabla za spremanje podataka poslanih od Function app-a. OnCreate je metoda koja se poziva za kreiranje aktivnosti. Kako se kreira izgled aplikacije iz setContentView(R.layout.statistics), nakon toga se poziva metoda sendMessage(7) kojom se Function app-u šalje poziv za dobivanje podataka za graf. Na kraju se samo poziva metoda setOnClickListener sa gumbom za natrag i metodom openMainActivity() kako bi se vratili na glavni activity pritiskom na njega.

```
public class Statistics extends AppCompatActivity {

    2 usages
    public Button button_back;

    3 usages
    public static ArrayList<Integer> CountGlobal;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.statistics);

        sendMessage( value: 7);

        button_back = findViewById(R.id.button_back);
        button_back.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) { openMainActivity(); }
        });
    }
}
```

OpenMainActivity metoda otvara glavni activity.

```
private void openMainActivity() {
    Intent intent = new Intent( packageContext: this, MainActivity .class);
    startActivity(intent);
}
```

MessageCallbackMqtt klasa u sebi sadrži metodu onCloudToDeviceMessageReceived. Poruka koja se u ovoj funkciji dobiva sadrži podatke za graf. Metoda onCloudToDeviceMessageReceived se poziva kada Function app pošalje podatke. U polje CountArray tipa int se spremaju svi brojevi koji predstavljaju broj poruka od 0:00 do 23:00 sata. Pozivaju se metode parseJsonToString i nakon toga extractValue kako bi se izvukle vrijednosti brojeva i spremile u polje CountArray. Nakon toga u CountGlobal spremi vrijednost varijable CountArray kako bi podaci ostali zapamćeni i mogli se iskoristiti za poslije. MakeGraph() metoda se poziva kada se želi napraviti linearni graf na temelju podataka u CountGlobal, a counter.increment() je metoda klase Counter koja služi kao brojač za poruke koje dobivamo od onCloudToDeviceMessageReceived.

```

class MessageCallbackMqtt implements com.microsoft.azure.sdk.iot.device.MessageCallback
{
    2 usages
    public IoTHubMessageResult onCloudToDeviceMessageReceived(Message msg, Object context)
    {
        Counter counter = (Counter) context;
        Log.i( tag: "Poruka", new String(msg.getBytes(), Message.DEFAULT_IOTHUB_MESSAGE_CHARSET));
        ArrayList<Integer> CountArray = extractValue(parseJsonToString(new String(msg.getBytes(), Message.DEFAULT_IOTHUB_MESSAGE_CHARSET)));
        CountGlobal = CountArray;

        Log.i( tag: "CountCheck", msg: "CountStringGlobal je " + CountGlobal.get(1));
        MakeGraph();
        counter.increment();

        return IoTHubMessageResult.COMPLETE;
    }
}

```

MakeGraph() metoda služi za prikazivanje linearног grafa. Otvara se MyDBHelper objekt koji služi za pristupanje podacima SQLite baze. SQLite baza je korišтena kako bi podaci o statistici bili trajno spremljeni i kada se aplikacija ugasi. For petlja koja se ponavlja 24 puta koristi metodu UpdateById objekta dbHelper kako bi na temelju vrijednosti u polju CountGlobal ažurirala 24 redaka koji se nalaze u bazi. Svaki redak ima id od 0 do 23 koji označava sate od 0:00 do 23:00, a vrijednost Count označava koliko se poruka u tom satu poslalo. Kada se vrijednosti u bazi ažuriraju tada se stvara polje dataPoints tipa Entry, tip podatka korišten za unose vrijednosti u grafu. Koristi te for petlja koja 24 puta ubacuje vrijednosti sa metodom add(). Koristi se dbHelper.SelectById(i) kako bi se dobile vrijednosti iz baze. Podaci tipa Entry se spremaju u dataPoints, a nakon toga stvaramo novi objekt LineDataSet na temelju dataPoints. LineDataSet tip objekta označava set podataka koji predstavljaju jednu crtu u grafu. Postavljamo boju na plavu i postavljamo da je tekst uz Entry vrijednosti plav. Nakon toga se kreira objekt LineData ne temelju lineDataSet varijable tipa LineDataSet. lineData je kolektivni objekt za podatke grafa koji može sadržati više LineDataSet-ova. Na kraju se postavljaju podaci na chart objekt tipa LineChart koji označava element unutar aktivnosti sa id vrijednosti lineChartOne. Postavljaju se podaci, javlja se grafu da su ostvarene promjene i pomoću invalidate() metode se natjera graf da se ponovno nacrtava.

```

public void MakeGraph() {
    MyDBHelper dbHelper = new MyDBHelper( context: this);
    ArrayList<Entry> dataPoints = new ArrayList<>();

    for (int i = 0; i < 24; i++) {
        dbHelper.UpdateById(i, CountGlobal.get(i));
    }

    for (int i = 0; i < 24; i++) {
        dataPoints.add(new Entry(i, dbHelper.SelectById(i)));
    }

    LineDataSet lineDataSet = new LineDataSet(dataPoints, label: "Number of Sent Messages");
    lineDataSet.setColor(Color.rgb( red: 0, green: 186, blue: 255));
    lineDataSet.setValueTextColor(Color.BLUE);

    LineData lineData = new LineData(lineDataSet);
    lineData.setDrawValues(true);

    LineChart chart = (LineChart)findViewById(R.id.lineChartOne);
    chart.setData(lineData);

    Description description = new Description();

    description.setText("Message Graph");

    chart.setDescription(description);

    chart.getData().notifyDataChanged();
    chart.invalidate();
}

```

SendMessage metoda prima argument value i spaja se na IoT Hub. Šalje se vrijednost poruke sa messageType = 7 na temelju value-a, a nakon toga se čeka poruka za callback.

```
public void sendMessage(int value) {
    String iotHubConnectionString = "HostName=HUB-RUS-jbegovic21.azure-devices.net;DeviceId=Androids;SharedAccessKey=ApQkG6BAwHwEcPRXicVyoBceknIuS2Ys6AIoTI62F7U=";
    DeviceClient deviceClient = new DeviceClient(iotHubConnectionString, IoTHubClientProtocol.MQTT);

    try {
        deviceClient.open( withRetry: true);
        Log.d( tag: "IoTHub", msg: "Successfully connected");

        // Set up message callback
        MessageCallbackMqtt callback = new MessageCallbackMqtt();
        Counter counter = new Counter( num: 0);
        deviceClient.setMessageCallback(callback, counter);

        // Create JSON message
        JSONObject jsonMessage = new JSONObject();
        jsonMessage.put( name: "messageType", value);
        jsonMessage.put( name: "distanceMeasured", value: 0);

        // Send JSON message
        Message msg = new Message(jsonMessage.toString());
        msg.setContentType("application/json");
        msg.set messageId(UUID.randomUUID().toString());
        Log.d( tag: "IoTHub", jsonMessage.toString());

        deviceClient.sendEvent(msg);
        Log.d( tag: "IoTHub", msg: "Successfully sent the message");
        // Wait for response (you may want to handle this asynchronously)
        // For simplicity, we'll just wait for a short duration here.
        // izbrisano
        Thread.sleep( millis: 2000);
        // Handle the received message
        // You can update UI or perform other actions based on the received message.
        counter.increment();

    } catch (IoTHubClientException | JSONException | InterruptedException e) {
        Log.e( tag: "IoTHub", msg: "Error: " + e.getMessage());
    } finally {
        deviceClient.close();
    }
}
```

ExtractValue je slična prošloj implementaciji, no u ovom slučaju koristi se for petlja kako bi se dobile vrijednosti i spremile u polje brojeva. Nakon toga se vraća polje brojeva.

```
private static ArrayList<Integer> extractValue(String jsonString) {  
    try {  
        // Create ObjectMapper instance  
        ObjectMapper objectMapper = new ObjectMapper();  
  
        // Parse JSON string to JsonNode  
        JsonNode jsonNode = objectMapper.readTree(jsonString);  
        JsonNode jsonArray = jsonNode.get("numOfNotificationsByHour");  
  
        ArrayList<Integer> messageType = new ArrayList<>();  
        // Extract value from JsonNode  
        for (int i=0;i<j jsonArray.size();i++) {  
            int Value = jsonArray.get(i).asInt();  
            Log.i( tag: "Vrijednost", msg: i + " je " + Value);  
            messageType.add(Value);  
        }  
        // Return the extracted value  
        Log.i( tag: "Message Type Prijevod", msg: "Poruka: " + messageType);  
        return messageType;  
    } catch (Exception e) {  
        e.printStackTrace();  
        return null;  
    }  
}
```

3.4.4. MyDBHelper

Ova klasa služi za pristup SQLite bazi i njeno korištenje. Kreiraju se neke osnovne varijable za postavljanje baze, poput DATABASE_NAME, DATABASE_VERSION i TABLE_NAME. Te tri varijable se koriste za kreiranje baze i tablice imena „Statistika“, a nakon toga KEY_ID i KEY_COUNT označavaju stupce za ID vrijednost i Count vrijednost. One se sve pozivaju u metodama MyDBHelper, konstruktor SQLite baze, a pri kreiranju baze poziva se metoda OnCreate.

```
public class MyDBHelper extends SQLiteOpenHelper {

    1 usage
    private static final String DATABASE_NAME = "Statistika";
    1 usage
    private static final int DATABASE_VERSION = 4;
    6 usages
    private static final String TABLE_NAME = "Statistika";

    4 usages
    private static final String KEY_ID = "ID";
    3 usages
    private static final String KEY_COUNT = "Count";
    2 usages
    public MyDBHelper(@Nullable Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {

        db.execSQL("CREATE TABLE "+TABLE_NAME+" ("+KEY_ID+" INTEGER PRIMARY KEY, "
            + KEY_COUNT + " INTEGER)");
    }
}
```

OnUpgrade metoda služi pri mijenjanju vrijednosti DATABASE_VERSION na veću vrijednost, tj. mijenjanju vrijednosti. Briše se stara baza i dodaje se nova sa OnCreate.

```
@Override  
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
  
    db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);  
  
    onCreate(db);  
}
```

SetupDatabase je metoda koja služi za postavljanje baze pri pokretanju aplikacije. Ona se pokreće pri dolasku u Main aktivnost. Ako u bazi ne postoji redak sa ID od 0 do 23, tada se na to mjesto dodaje novi redak sa ID vrijednosti koja nedostaje i Count vrijednosti 0.

```
public void SetupDatabase() {  
    SQLiteDatabase database = this.getWritableDatabase();  
  
    for (int i=0;i<24;i++) {  
        Cursor cursor = database.query(TABLE_NAME, columns: null, selection: KEY_ID + " = " + i, selectionArgs: null, groupBy: null, having: null, orderBy: null);  
        boolean exists = cursor.moveToFirst();  
        if (!exists) {  
            ContentValues values = new ContentValues();  
            values.put(KEY_ID, i);  
            values.put(KEY_COUNT, 0);  
            database.insert(TABLE_NAME, nullColumnHack: null, values);  
        }  
    }  
    database.close();  
}
```

SelectById je metoda za vraćanje vrijednosti koja se nalazi na poziciji id koja je argument te metode. Ako se nalazi vrijednost, tada varijabla Count postaje ta vrijednost, a ako ne postoji vraća se 0.

```
public int SelectById(int id) {  
    SQLiteDatabase database = this.getReadableDatabase();  
    Cursor cursor = database.query(TABLE_NAME, columns: null, selection: KEY_ID + " = " + id, selectionArgs: null, groupBy: null, having: null, orderBy: null);  
    int Count = 0;  
    if (cursor.moveToFirst()) {  
        Count = cursor.getInt(columnIndex: 1);  
    }  
  
    cursor.close();  
    database.close();  
    return Count;  
}
```

UpdateById je naredba koja ažurira vrijednost na temelju argumenta id i count. Ažurira se vrijednost, a vraća se vrijednost retka koji je ažuriran ako se ažurirao.

```
public int UpdateById(int id, int count) {
    SQLiteDatabase database = this.getWritableDatabase();
    ContentValues values = new ContentValues();
    values.put(KEY_COUNT, count);

    int rowsUpdated = database.update(TABLE_NAME, values, whereClause: "id = " + id, whereArgs: null);
    database.close();
    return rowsUpdated;
}
```

3.4.5. XML

Ovaj XML layout definira izgled glavnog ekrana ("Main") u Android aplikaciji. Ključne stavke u ovom layoutu uključuju:

1. ConstraintLayout: Koristi se kao korijenski layout, pružajući snažnu podršku za postavljanje elemenata sučelja pomoću ograničenja.
2. TextView (Title): Prikazuje naslov na sredini ekrana, označen kao "windowguard_control". Font, veličina, i boja teksta su prilagođeni, a pozadinska slika postavljena je na "background2".
3. Button (calibration): Gumb za pokretanje kalibracije. Stiliziran je s određenim fontom, veličinom, i bojom teksta, te ima određenu visinu i poziciju u odnosu na druge elemente. Boja pozadine i efekti klika su također definirani.
4. Button (statistics): Gumb za prikaz statističkih podataka. Ima slična svojstva kao i gumb za kalibraciju, ali je postavljen na dno ekrana.
5. ImageView (imageView3): Prikazuje logo aplikacije. Postavljen je u gornji desni kut ekrana i ima definiranu širinu i visinu.

Ostale XML datoteke su na istom ili vrlo sličnom principu te su im samo promijenjene vrijednosti ograničenja, veličina i slično.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/Statistika"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/background2"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id>Title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentStart="true"
        android:layout_alignParentTop="true"
        android:layout_centerInParent="true"
        android:layout_marginTop="100dp"
        android:fontFamily="@font/roboto_bold"
        android:gravity="center"
        android:text="WindowGuard Control"
        android:textColor="#000000"
        android:textSize="48sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.483"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
```

```
<Button
    android:id="@+id/calibration"
    android:layout_width="wrap_content"
    android:layout_height="80dp"
    android:layout_alignParentTop="true"
    android:layout_centerHorizontal="true"
    android:layout_marginBottom="110dp"
    android:backgroundTint="#38B6FF"
    android:fontFamily="@font/roboto_bold"
    android:text="Calibration"
    android:textColor="#000000"
    android:textSize="24sp"
    app:layout_constraintBottom_toTopOf="@+id/statistics"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:rippleColor="#000000" />

<Button
    android:id="@+id/statistics"
    android:layout_width="wrap_content"
    android:layout_height="65dp"
    android:layout_alignParentBottom="true"
    android:layout_centerHorizontal="true"
    android:layout_marginBottom="125dp"
    android:backgroundTint="#38B6FF"
    android:fontFamily="@font/roboto_bold"
    android:text="Statistics"
    android:textColor="#000000"
    android:textSize="20sp"
    app:iconTint="#000000"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent" />
```

```
<ImageView  
    android:id="@+id/imageView3"  
    android:layout_width="118dp"  
    android:layout_height="119dp"  
    android:layout_marginTop="85dp"  
  
    android:layout_marginEnd="16dp"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintTop_toTopOf="@+id>Title"  
    app:srcCompat="@drawable/logo_transparent" />  
  
</androidx.constraintlayout.widget.ConstraintLayout>
```

Samo ćemo napomenuti element grafa u XML datoteci za ekran “Statistics”

Ovaj XML element definira “LineChart” iz biblioteke MPAndroidChart, koji se koristi za prikazivanje linija na grafikonu.

Ovaj element predstavlja konfiguraciju “LineChart” widgeta, koji će biti smješten unutar “ConstraintLayout”-a i koji će prikazivati podatke u obliku linija na grafikonu. Grafikon će zauzimati određeni prostor između navedenih granica i bit će pozicioniran relativno prema drugim elementima unutar “ConstraintLayout”-a.

```
<com.github.mikephil.charting.charts.LineChart  
    android:id="@+id/lineChartOne"  
    android:layout_width="332dp"  
    android:layout_height="285dp"  
    app:layout_constraintBottom_toTopOf="@+id/button_back"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/textViewTitle"  
    app:layout_constraintVertical_bias="0.135" />
```

4. Kratki troškovnik i financijska analiza

Troškovi izrade ove aplikacije dolaze iz sljedećih elemenata: senzor za udaljenost, senzor za kvalitetu zraka, Dasduino pločica, održavanje servera te ljudski rad pri samoj izradi aplikacije.

Cijena jedne Dasduino CONNECTPLUS pločice iznosi 13.95€, senzor kvalitete zraka CCS811 iznosi 19.95€ te senzor za mjerjenje udaljenosti za koji smo koristili ultrazvučni modul HC-SR04 kojemu je cijena 3.95€. Sveukupno to iznosi 37.85€

Mjesečna cijena osnovnog održavanja servera iznosi oko 39.78€ mjesečno. U slučaju da želimo imati uz održavanje i podršku serveru dostupnu 24/7, tada ta cijena može prelaziti 65.61€. Uzmemo li tu drugu opciju to će nas na kraju iznositi 787.32€ godišnje

Cijena izrade same aplikacije sa njenim funkcionalnostima bi iznosila 3 300€. U to je uračunat rad tri osobe koje bi radile na aplikaciji, UI dizajner, android developer i Dasduino logic developer koji bi radili na aplikaciji jedan mjesec. Njihova plaća bi iznosila otprilike 1 100€ te smo tu vrijednost uzeli pri izračunu.

Uzmemo li sve navedeno u obzir zaključujemo da je izgradnja same aplikacije 3 300€ te dodatnih 787.32€ godišnje za održavanje i podršku serveru.

Za izradu jednog uređaja senzora povezanih na Dasduino koštati će nas 37.85€.

Za izgradnju n broja jedinica, moramo jedino množiti sa cijenom izrade jedinice, jer je aplikacija ista te server koji plaćamo na godišnjoj bazi također vredni za više uređaja.

1 uređaj – 3 337.85€ + 787.32€ godišnje

2 uređaja – 3 375.70€ + 787.32€ godišnje

5 uređaja – 3 489.25€ + 787.32€ godišnje

10 uređaja – 3 678.50€ + 787.32€ godišnje

5. Korisnička dokumentacija

Pri prvoj instalaciji korisniku se nudi opcija prihvatanja ili odbijanja primanja obavijesti na mobilni uređaj. U slučaju da želi primati notifikacije o tome kada bi trebao otvoriti ili zatvoriti prozor, korisnik će stisnuti opciju da prihvata.

WindowGuard Control



Calibration



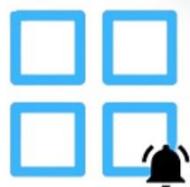
Želite li aplikaciji **WindowGuard Control** dopustiti
da vam šalje obavijesti?

Dopusti

Nemoj dopustiti

Korisniku je pri otvaranju aplikacije prezentiran početni ekran sa nazivom aplikacije te sa dva gumba, „Calibration” i “Statistics”.

WindowGuard Control



Calibration

Statistics

Pritiskom na prvi gumb „Calibration“, otvara se drugi prikaz gdje je prikazano sljedeće: gumb za kalibriranje otvorenog prozora („Calibrate open window“), gumb za kalibriranje zatvorenog prozora („Calibrate closed window“) te dva polja u kojima se prikazuju vrijednosti udaljenosti otvorenog i zatvorenog prozora.

CALIBRATION



0.0 cm

0.0 cm

Calibrate
opened window

Calibrate
closed window

Back

Korisnik zatim odabire želi li kalibrirati tj. postaviti vrijednost udaljenosti za otvoren ili zatvoren prozor pritiskom na određeni gumb na kojem je to indicirano. Taj gumb šalje naredbu preko cloud servisa senzoru za očitavanje udaljenosti da udaljenost i izmjeri. On to ponovno šalje aplikaciji te se to zatim prikazuje u sukladnom polju za prikaz.

CALIBRATION



46.2 cm

0.0 cm

Calibrate
opened window

Calibrate
closed window

Back

CALIBRATION



46.2 cm 8.34 cm

Calibrate
opened window

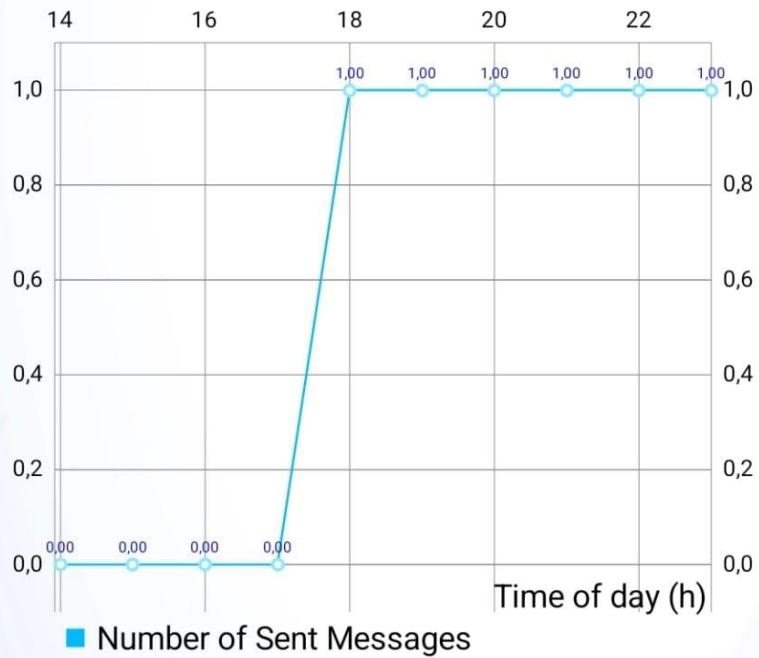
Calibrate
closed window

Back

Na tom zaslonu se također još nalazi gumb „Back“ za povratak na početni zaslon.

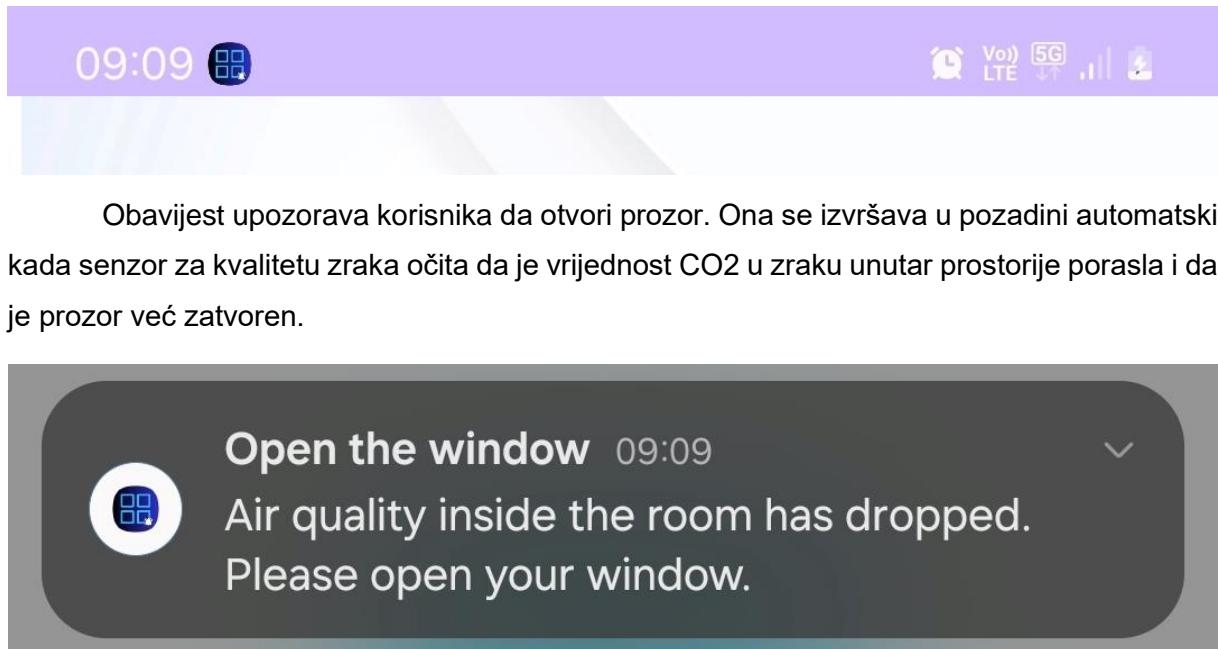
Trećem prikazu pristupamo pritiskom na gumb „Statistics“ te na njemu imamo prikazane grafove sa vrijednosti brojem poslanih poruka kroz vrijeme.

Statistics



Back

Osim dva prethodno opisana zaslona, aplikacija također prima obavijesti.



6. Zaključak

Smart Window State Sensor predstavlja primjer složenog umreženog sustava koji omogućuje praćenje i upravljanje stanjem prozora putem Android uređaja. Prvo su prikazane pojedinačne komponente, kao što su Dasduino CONNECTPLUS, senzor udaljenosti, senzori kvalitete zraka, Microsoft Azure Cloud servisi i Android aplikacija. Potom su te komponente prikazane kao složeni sustav, njihov međusobni način komuniciranja, kao i aktivnosti koje se izvode u promatranom trenutku događaja. Sve opisane aktivnosti i slijed njihovog izvršavanja uvelike olakšava razumijevanje napisanog programskog koda. Kod je generalno pisan u C# jeziku, zaslužan za povezivanje Azure Cloud servisa i uređaja, dok je primarno Java programski jezik bio prisutan za implementaciju Android aplikacije. Ono što je bitno za uočiti je da Microsoft Azure Cloud servisi predstavljaju ključnu infrastrukturu za povezivanje svih komponenti sustava. Kroz različite servise poput Logic App, Function App i SQL Database servisa, omogućeno je prikupljanje, obrada i analiza podataka te upravljanje komunikacijom između mjernih uređaja (senzora) i Android aplikacije. Nadalje, Android aplikacija predstavlja interaktivno sučelje između korisnika i pametnog prozora, omogućujući im interakciju sa senzorom udaljenosti i to samo (iz perspektive korisnika) putem mobilnog telefona. Tako primjerice kroz aplikaciju, korisnici mogu kalibrirati senzore, pregledavati statističke podatke i primati obavijesti o stanju prozora.

Može se istaknuti da su ključni element ovog projekta pametni prozor koji integrira različite senzore kako bi prikupio informacije o okolini i omogućio korisnicima da na temelju tih informacija donose odluke o otvaranju i zatvaranju prozora. Senzori za kvalitetu zraka pružaju uvid u podatke o kvaliteti zraka unutar i izvan prostorije, dok senzor udaljenosti omogućuje praćenje stanja prozora. Sve u svemu, rješenje koje Smart Window State Sensor pruža, odnosno pametni senzor za detekciju stanja prozora, jest rješenje za praćenje i upravljanje stanjem prozora putem pametnih uređaja, ističući važnost Cloud tehnologije. Cloud tehnologija ne samo da je središte komunikacije, već se na njenim principima bazira i sama smisao razvoja umreženih sustava. Umreženi sustavi nisu ništa više nego grupa senzora i uređaja koji međusobno komuniciraju i razmjenjuju podatke putem odgovarajućeg protokola, baš kao i našem slučaju Smart Window State Sensorsa.

Popis literature

- [1] Dasduino CONNECTPLUS. (bez dat.). Pristupano 24. siječnja, 2024, sa <https://soldered.com/hr/proizvod/dasduino-connectplus/>
- [2] HC-SR04 Ultrasonic Sensor Working, Pinout, Features & Datasheet. (bez dat.). Pristupano 24. siječnja, 2024, sa <https://components101.com/sensors/ultrasonic-sensor-working-pinout-datasheet>
- [3] Smart Citizen. (bez dat.). Pristupano 27. siječnja, 2024, sa <https://smartcitizen.me/kits/>
- [4] Cloud Computing Services | Microsoft Azure. (bez dat.). Pristupano 27. siječnja, 2024, sa <https://azure.microsoft.com/en-us/>
- [5] Android Mobile App Developer Tools – Android Developers. (bez dat.). Pristupano 28. siječnja, 2024, sa <https://developer.android.com/>
- [6] Visual Paradigm - Online Productivity Suite. (bez dat.). Pristupano 28. siječnja, 2024, sa <https://online.visual-paradigm.com/>
- [7] Welcome to Fritzing. (bez dat.). Pristupano 28. siječnja, 2024, sa <https://fritzing.org/>
- [8] Web-based tooling for BPMN, DMN, CMMN, and Forms | bpmn.io. (bez dat.). Pristupano 30. siječnja, 2024, sa <https://bpmn.io/>
- [9] „laboratory107/LabSession-RUS-Device-Empty“. Pridstupljeno: 31. siječanj 2024. [Na internetu]. Dostupno na: <https://github.com/laboratory107/LabSession-RUS-Device-Empty>
- [10] „ChatGPT“. Pridstupljeno: 31. siječanj 2024. [Na internetu]. Dostupno na: <https://chat.openai.com>

Popis slika

Slika 1: Dasduino CONNECT PLUS.....	3
Slika 2: Dasduino CONNECTPLUS (ESP32) - Prikaz konekcija.....	4
Slika 3: HC-SR04 Ultrasonic Sensor - Prikaz pinova	5
Slika 4: princip rada ultrazvučnog senzora	5
Slika 5: instalacija senzora za kvalitetu zraka	6
Slika 6: primjer senzora kvalitete zraka	6
Slika 7: Smart Citizen - karta umreženih uređaja u Europi.....	6
Slika 8: vrste senzora ugrađenih u uređaj za mjerjenje kvalitete zraka	7
Slika 9: Microsoft Azure logo i prikaz ikona nekih od servisa koje nudi.....	8
Slika 10: Android Studio logo.....	9
Slika 11: Android Studio razvojna okolina.....	9
Slika 12: složena arhitektura sustava Smart Window.....	10
Slika 13: komunikacijski dijagram cijele arhitekture sustava	12
Slika 14: komunikacijski dijagram - Kalibracija prozora.....	13
Slika 15: komunikacijski dijagram - Prikaz grafikona.....	13
Slika 16: komunikacijski dijagram - Dohvaćanje vrijednosti sa senzora kvalitete zraka.....	14
Slika 17: komunikacijski dijagram - Kreiranje obavijesti	15
Slika 18: dijagram aktivnosti arhitekture sustava	16
Slika 19: staza korisnika mobilne aplikacije 1/3	19
Slika 20: staza korisnika mobilne aplikacije 2/3	20
Slika 21: staza korisnika mobilne aplikacije 3/3	21
Slika 22: staza Azure 1/3.....	22
Slika 23: staza Azure 2/3.....	23
Slika 24: staza Azure 3/3.....	24
Slika 25: staza mjerača udaljenosti 2/3.....	25
Slika 26: staza mjerača udaljenosti 1/3.....	25
Slika 27: staza mjerača udaljenosti 3/3.....	26
Slika 29: shema spajanja senzora udaljenosti	27
Slika 30: Logic App servis - prikaz koraka (logike)	30
Slika 31: prikaz upisanih vrijednosti razine eCO2 u zraku u bazu podataka.....	31